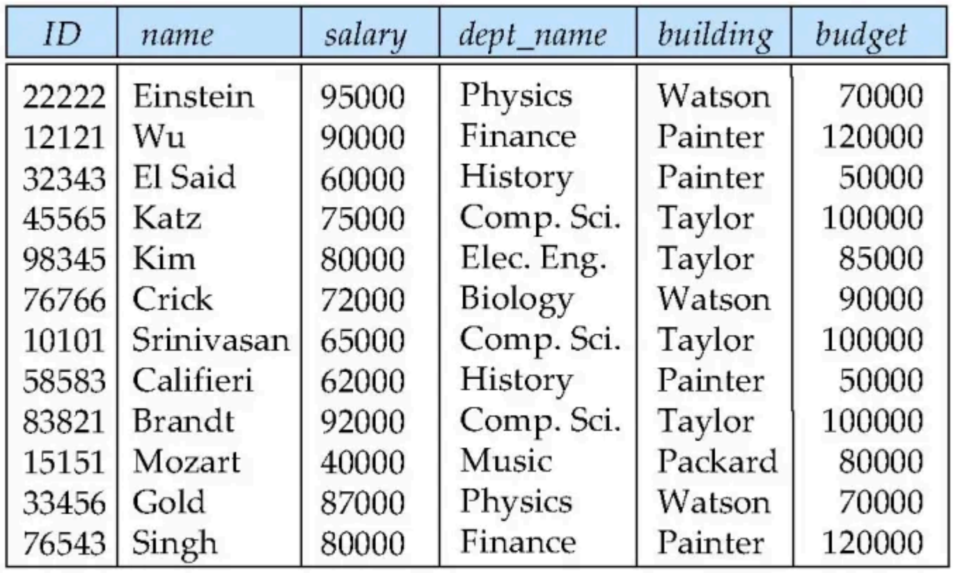
Normalization (DBMS) - 1

Created	@March 31, 2025 4:03 PM
Tags	

Anomalies in Databases:

1. Insertion Anomaly: New department with no instructors will introduce null to the ID column.
2. Update Anomaly: Updating budget of Finance dept will have to be done multiple times, else DB will become inconsistent.
3. Deletion Anomaly: Deleting Mozart's record removes all information about the Music Dept!



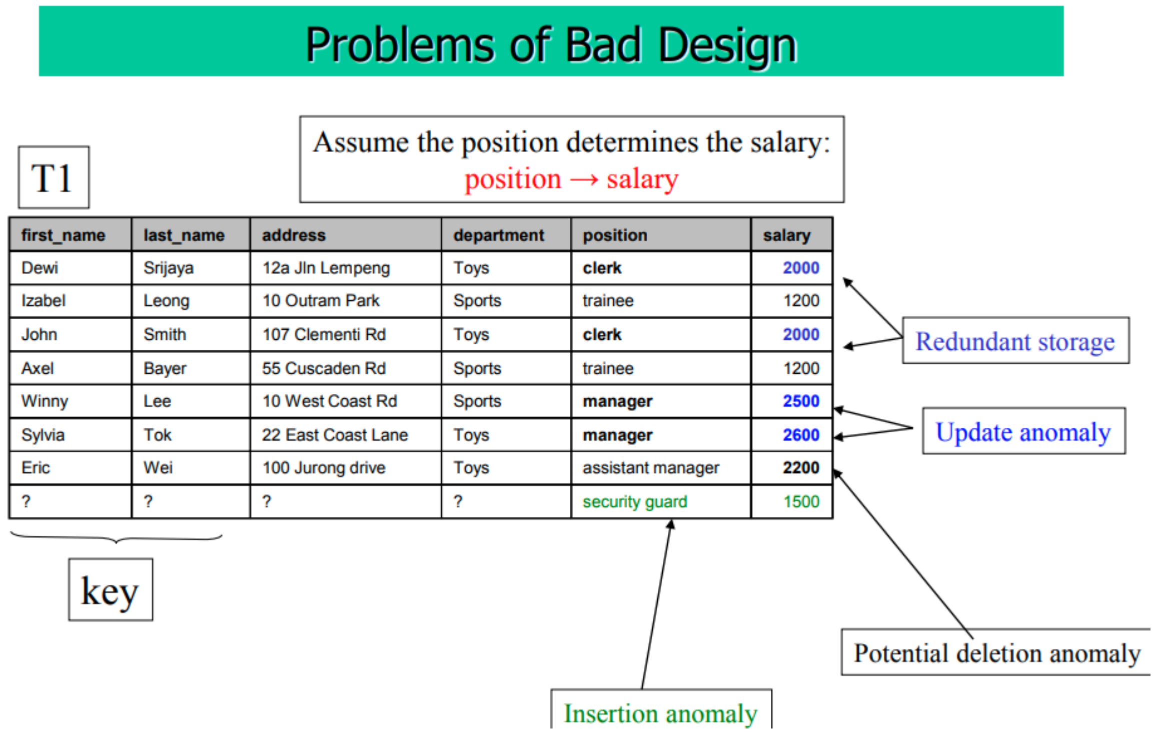
ID	*name*	*salary*	*dept_name*	*building*	*budget*
22222	Einstein	95000	Physics	Watson	70000
12121	Wu	90000	Finance	Painter	120000
32343	El Said	60000	History	Painter	50000
45565	Katz	75000	Comp. Sci.	Taylor	100000
98345	Kim	80000	Elec. Eng.	Taylor	85000
76766	Crick	72000	Biology	Watson	90000
10101	Srinivasan	65000	Comp. Sci.	Taylor	100000
58583	Califieri	62000	History	Painter	50000
83821	Brandt	92000	Comp. Sci.	Taylor	100000
15151	Mozart	40000	Music	Packard	80000
33456	Gold	87000	Physics	Watson	70000
76543	Singh	80000	Finance	Painter	120000

Fig. 1: in_dep schema

Other problems:

1. Budget info is repeated multiple times.
2. Even if one value of budget goes wrong, DB will be inconsistent.

3. Another example:



Problems of Bad Design

T1

Assume the position determines the salary:
position → salary

first_name	last_name	address	department	position	salary
Dewi	Srijaya	12a Jln Lempeng	Toys	**clerk**	**2000**
Izabel	Leong	10 Outram Park	Sports	trainee	1200
John	Smith	107 Clementi Rd	Toys	**clerk**	**2000**
Axel	Bayer	55 Cuscaden Rd	Sports	trainee	1200
Winny	Lee	10 West Coast Rd	Sports	**manager**	**2500**
Sylvia	Tok	22 East Coast Lane	Toys	**manager**	**2600**
Eric	Wei	100 Jurong drive	Toys	assistant manager	**2200**
?	?	?	?	security guard	1500

Features of a Good Relational Design: Should not have all these problems!

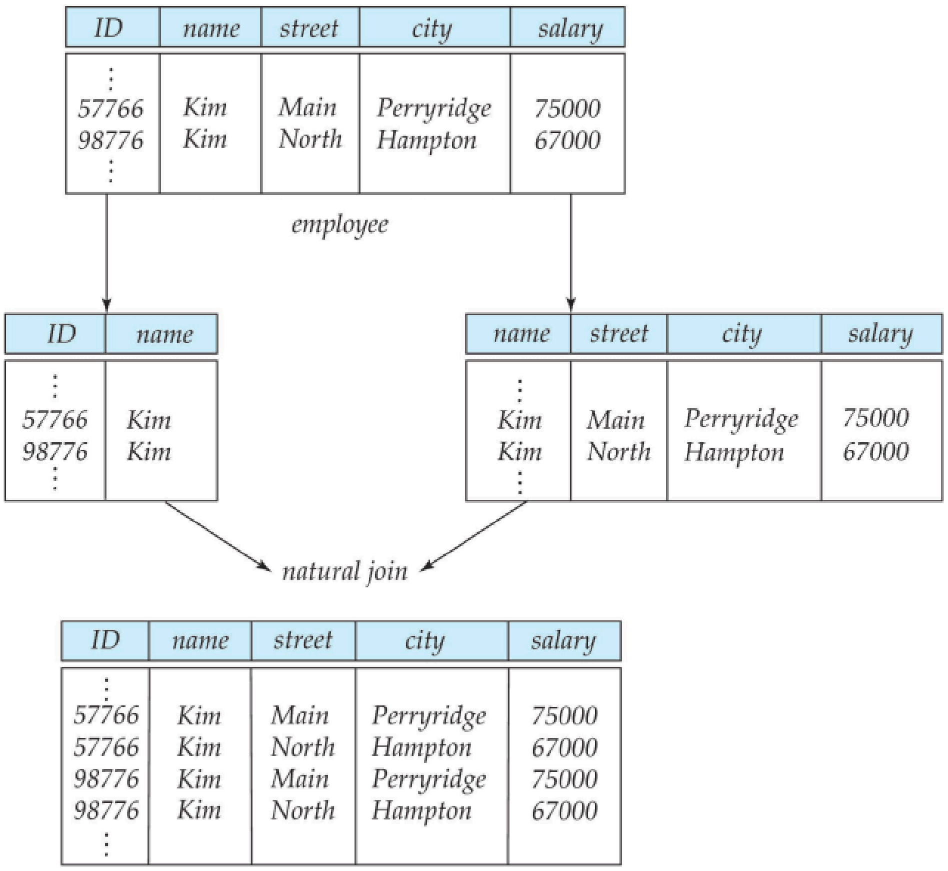
Note: Remember the original design(Lab Assignment 1) does not have these problems; we can insert, delete, update, at per our requirements.

But how to go from this table to that design? **DECOMPOSITION**!

Decomposition:

1. The in_dep schema can be decomposed into an **instructor** schema, and a **department** schema.
2. Any schema, in general, that shows repitition of information, may have to be decomposed into several smaller schemas.
3. But not all decompositions are useful!
4. Suppose we decompose ***employee(ID, name, street, city, salary)*** into

employee1 (ID, name) and ***employee2 (name, street, city, salary)***



ID	name	street	city	salary
⋮				
57766	Kim	Main	Perryridge	75000
98776	Kim	North	Hampton	67000
⋮				

ID	name
⋮	
57766	Kim
98776	Kim
⋮	

name	street	city	salary
⋮			
Kim	Main	Perryridge	75000
Kim	North	Hampton	67000
⋮			

ID	name	street	city	salary
⋮				
57766	Kim	Main	Perryridge	75000
57766	Kim	North	Hampton	67000
98776	Kim	Main	Perryridge	75000
98776	Kim	North	Hampton	67000
⋮				

5. We don't know which Kim has which ID! Kim with ID=57766 does not live in North Hampton, but SQL does not know!!
6. This is a lossy decomposition - although we're getting MORE rows after the natural join, we are not gaining anything - rather, we are losing information!
7. We should aim for lossless decompositions:

- Let R be a relation schema and let R_1 and R_2 form a decomposition of R . That is R = R_1 U R_2
- We say that the decomposition is a **lossless decomposition** if there is no loss of information by replacing R with the two relation schemas R_1 U R_2
- Formally,

$$\Pi_{R_1}(r) \bowtie \Pi_{R_2}(r) = r$$

- And, conversely a decomposition is lossy if

$$r \subset \Pi_{R_1}(r) \bowtie \Pi_{R_2}(r) = r$$

8. Example of lossless decomposition:

- Decomposition of $R = (A, B, C)$

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

A	B	C
α	1	A
β	2	B

r

A	B
α	1
β	2

$\Pi_{A,B}(r)$

B	C
1	A
2	B

$\Pi_{B,C}(r)$

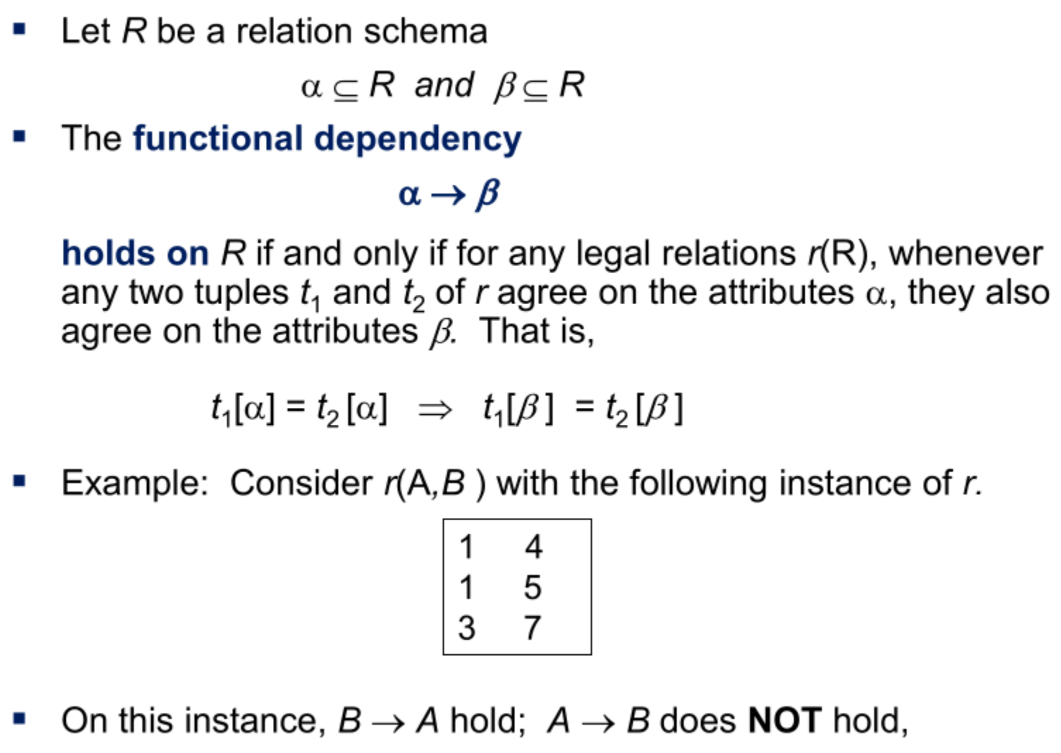
$\Pi_A(r) \bowtie \Pi_B(r)$

A	B	C
α	1	A
β	2	B

9. When decomposition is lossless, we preserve all our original data.
10. How to ensure a decomposition is lossless? - Preserve all original data, reduce redundancy.
11. How to ensure that these two conditions are satisfied? Follow the Functional Dependencies!

Functional Dependencies:

1. There are usually a variety of constraints (rules) on the data in the real world.
2. For example, some of the constraints that are expected to hold in a university database are:
    a. Students and instructors are uniquely identified by their ID.
    b. Each student and instructor has only one name.
    c. Each instructor and student is (primarily) associated with only one department.
    d. Each department has only one value for its budget, and only one associated building.
3. An instance of a relation that satisfies all such real-world constraints is called a legal instance of the relation;
4. A legal instance of a database is one where all the relation instances are legal instances.
    a. Constraints on the set of legal relations.
    b. Require that the value for a certain set of attributes determines uniquely the value for another set of attributes.
    c. **A functional dependency is a generalization of the notion of a key. What does it mean? Let's first understand FDs below:**



- Let *R* be a relation schema

$$\alpha \subseteq R \text{ and } \beta \subseteq R$$

- The **functional dependency**

$$\alpha \rightarrow \beta$$

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 α, they also agree on the attributes β. That is,

$$t_1[\alpha] = t_2[\alpha] \Rightarrow t_1[\beta] = t_2[\beta]$$

- Example: Consider *r*(A,*B*) with the following instance of *r.*

1	4
1	5
3	7

- On this instance, $B \rightarrow A$ hold; $A \rightarrow B$ does **NOT** hold,

5. K is a superkey for relation schema R if and only if K → R
6. K is a candidate key for R if and only if
   - K → R, and
   - for no α ⊂ K, α → R
7. Consider the schema: in_dep (ID, name, salary, dept_name, building, budget), we can consider {ID, dept_name} as a super-key.
8. So, this FD holds: ID, dept_name → name, salary, building, budget
9. We know these FDs to hold too: dept_name→ building, ID→ building
10. But what about this: dept_name → salary ?
11. **This does not hold!**
12. We have to check which FDs hold on which relations.
    a. ***If a relation r is legal under a set F of functional dependencies, we say that r satisfies F.***

Trivial Functional Dependencies:

$\alpha \rightarrow \beta$ is trivial if $\beta \subseteq \alpha$

NUMERICAL on FDs:

1. Let's take a look at this dependency: `Student_ID → Student_Name`

   Consider a student table:

   Students Table:

Student ID	Student Name
101	Sam
102	Emma
103	Sam

   In this table, two students share the name "Sam," but their distinct Student IDs differentiate them.

   The functional dependency "Student_ID → Student_Name" highlights that Student ID uniquely identifies Student Name.

Task: Validating Dependencies

Amoung the following cases, select the cases that are valid.

Case 1:

Student ID	Student Name
101	Sam
102	Sam

Case 2:

Student ID	Student Name
103	Emma
104	John

Case 3:

Student ID	Student Name
105	Lily

105	Lily

Case 4:

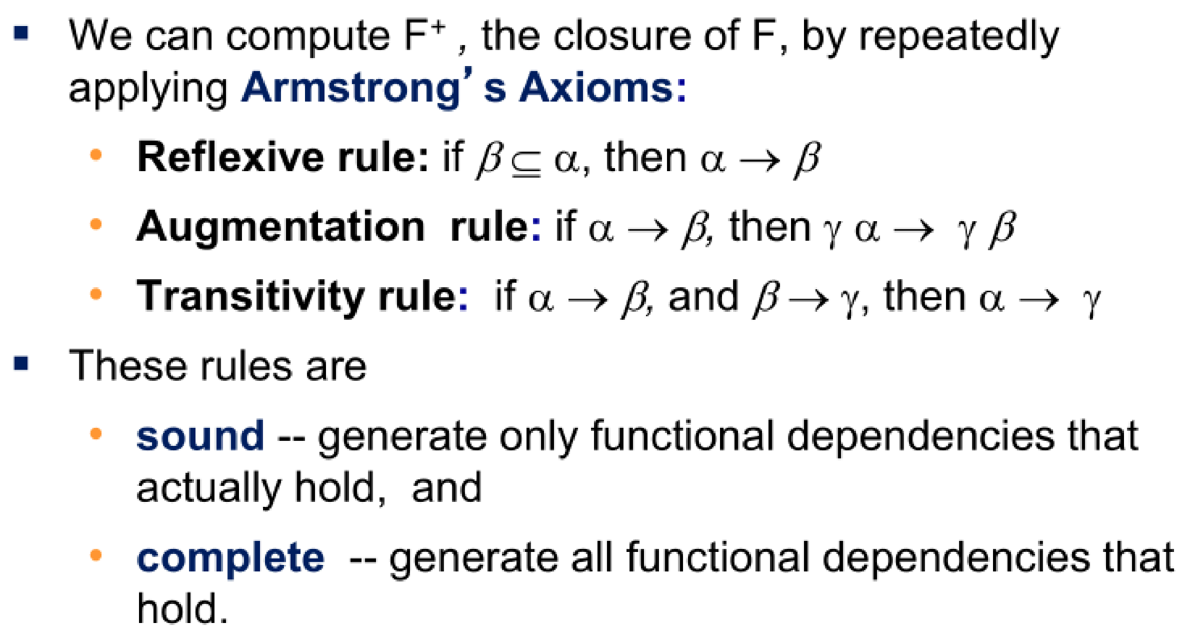
Student ID	Student Name
106	Max
106	Alex

Select all the cases which are valid (1,2,3).

```
Case 1 Valid - Different IDs prevent confusion despite same names.
Case 2 Valid - Different IDs and names avoid ambiguity.
Case 3 Valid - Same student with same name and ID.
Case 4 Invalid - Contradicts "Student_ID → Student_Name" as same ID ha
s different names.
```

Closure of a Set of Functional Dependencies

1. The set of all functional dependencies logically implied by F is the closure of F.
    a. Denoted by F+.
    b. ARMSTRONG's AXIOMS:



- We can compute F^+, the closure of F, by repeatedly applying **Armstrong's Axioms:**
    - **Reflexive rule:** if $\beta \subseteq \alpha$, then $\alpha \rightarrow \beta$
    - **Augmentation rule:** if $\alpha \rightarrow \beta$, then $\gamma\,\alpha \rightarrow \gamma\,\beta$
    - **Transitivity rule:** if $\alpha \rightarrow \beta$, and $\beta \rightarrow \gamma$, then $\alpha \rightarrow \gamma$
- These rules are
    - **sound** -- generate only functional dependencies that actually hold, and
    - **complete** -- generate all functional dependencies that hold.

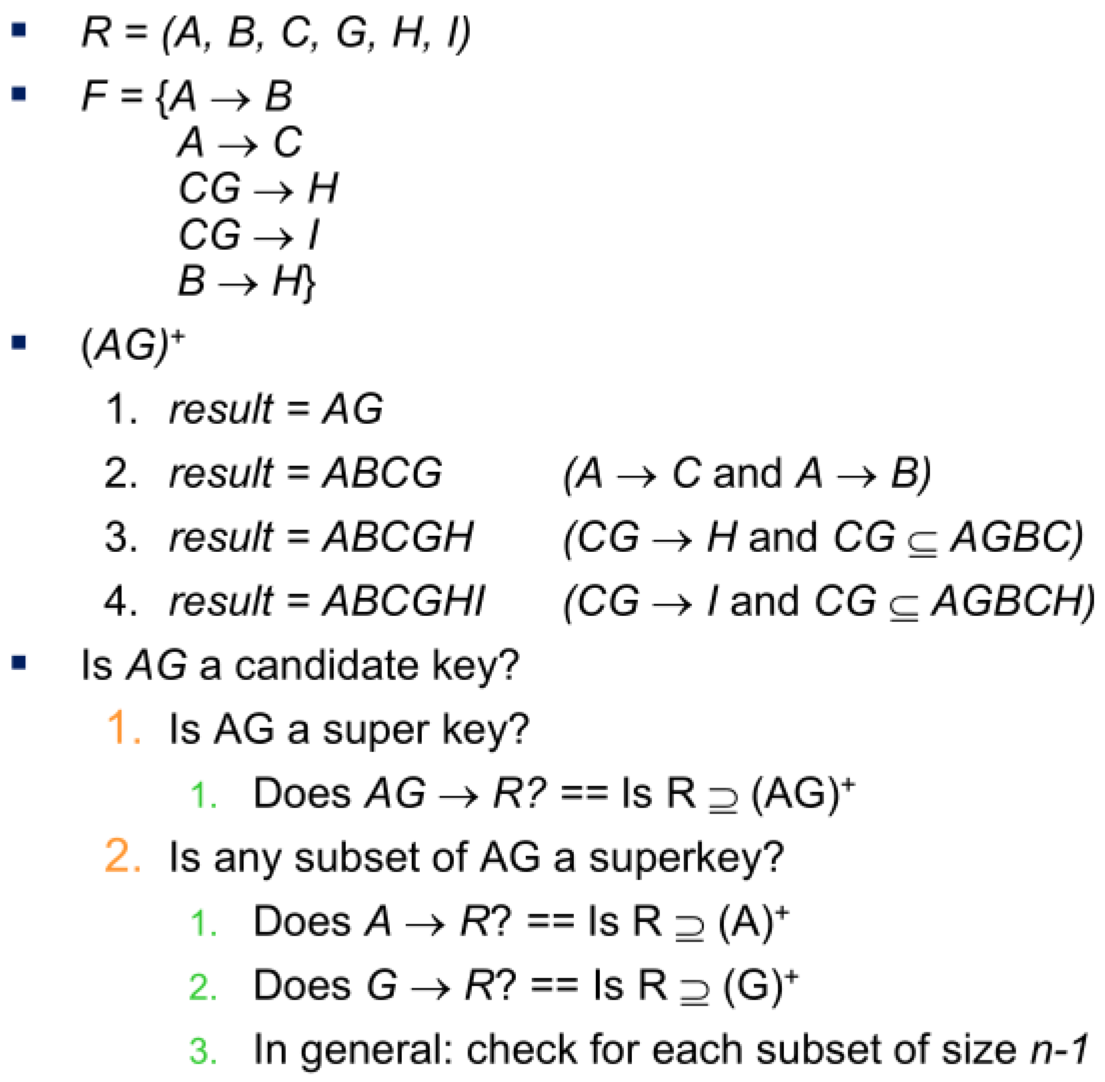
2. Example:

- *R = (A, B, C, G, H, I)*
  $F = \{ A \rightarrow B$
  $A \rightarrow C$
  $CG \rightarrow H$
  $CG \rightarrow I$
  $B \rightarrow H\}$
- Some members of F^+
  - $A \rightarrow H$
    - by transitivity from $A \rightarrow B$ *and* $B \rightarrow H$
  - $AG \rightarrow I$
    - by augmenting $A \rightarrow C$ with G, to get $AG \rightarrow CG$
      and then transitivity with $CG \rightarrow I$
  - $CG \rightarrow HI$
    - by augmenting $CG \rightarrow I$ to infer $CG \rightarrow CGI$,
      and augmenting of $CG \rightarrow H$ to infer $CGI \rightarrow HI$,
      and then transitivity

3. More rules:

- Additional rules:
  - **Union rule**: If $\alpha \rightarrow \beta$ holds and $\alpha \rightarrow \gamma$ holds, then $\alpha \rightarrow \beta \gamma$ holds.
  - **Decomposition rule**: If $\alpha \rightarrow \beta \gamma$ holds, then $\alpha \rightarrow \beta$ holds and $\alpha \rightarrow \gamma$ holds.
  - **Pseudotransitivity rule**:If $\alpha \rightarrow \beta$ holds and $\gamma \beta \rightarrow \delta$ holds, then $\alpha \gamma \rightarrow \delta$ holds.

4. **Closure of Attribute Sets:** 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.
5. Example:



- $R = (A, B, C, G, H, I)$
- $F = \{A \rightarrow B$
  $A \rightarrow C$
  $CG \rightarrow H$
  $CG \rightarrow I$
  $B \rightarrow H\}$
- $(AG)^+$
  1. *result* = AG
  2. *result* = $ABCG$ $(A \rightarrow C$ and $A \rightarrow B)$
  3. *result* = $ABCGH$ $(CG \rightarrow H$ and $CG \subseteq AGBC)$
  4. *result* = $ABCGHI$ $(CG \rightarrow I$ and $CG \subseteq AGBCH)$
- Is AG a candidate key?
  1. Is AG a super key?
     1. Does $AG \rightarrow R$? == Is $R \supseteq (AG)^+$
  2. Is any subset of AG a superkey?
     1. Does $A \rightarrow R$? == Is $R \supseteq (A)^+$
     2. Does $G \rightarrow R$? == Is $R \supseteq (G)^+$
     3. In general: check for each subset of size *n-1*

6. Uses of Attribute Closure:
   a. **Testing for superkey:**
      i. To test if α is a superkey, we compute α+, and check if α+ contains all attributes of R.
   b. **Testing functional dependencies:**

        a. To check if a functional dependency α → β holds (or, in other words, is in F+), just check if β ⊆ α+.

            a. That is, we compute α+ by using attribute closure, and then check if it contains β.

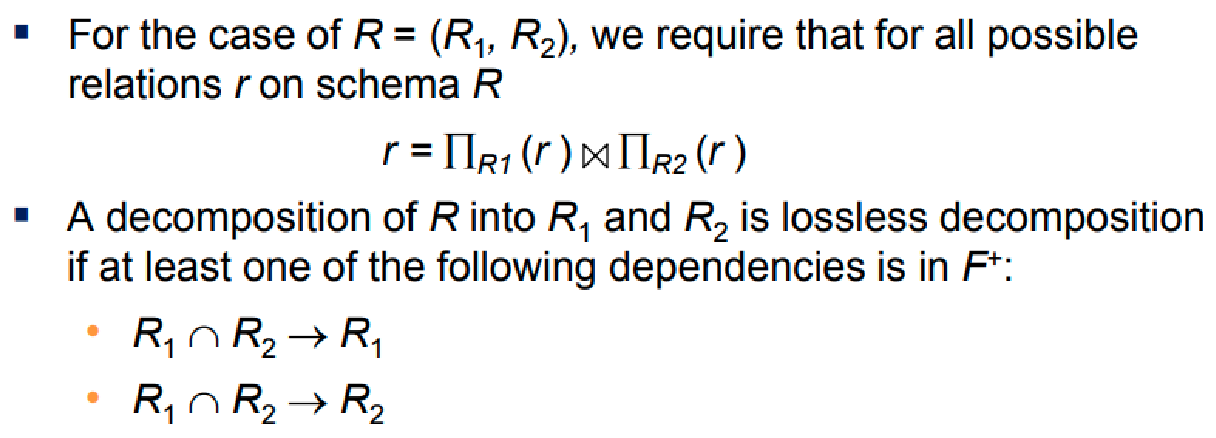
                a. Is a simple and cheap test, and very useful

    c. **Computing closure of F:**

        a. ***<u>For each γ ⊆ R, we find the closure γ+, and for each S ⊆ γ+, we output a functional dependency γ → S.</u>***

Lossless Decomposition:

1. Rules for checking lossless decomposition:

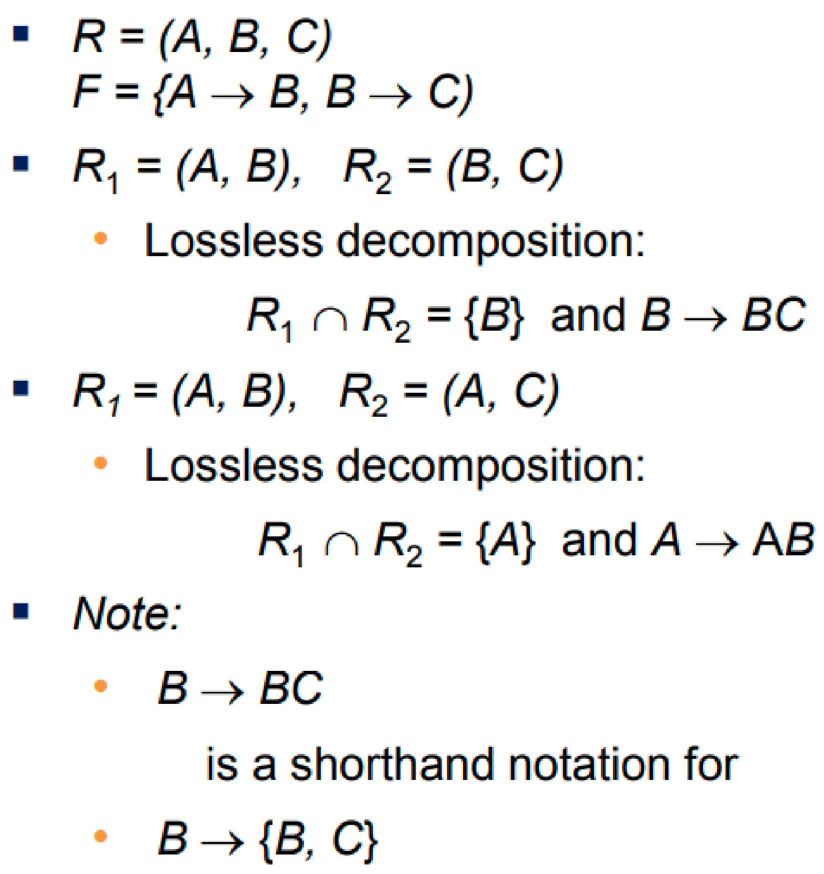


- For the case of $R = (R_1, R_2)$, we require that for all possible relations r on schema R

$$r = \Pi_{R1}(r) \bowtie \Pi_{R2}(r)$$

- A decomposition of R into R_1 and R_2 is lossless decomposition if at least one of the following dependencies is in F^+:
    - $R_1 \cap R_2 \rightarrow R_1$
    - $R_1 \cap R_2 \rightarrow R_2$

3. Example:

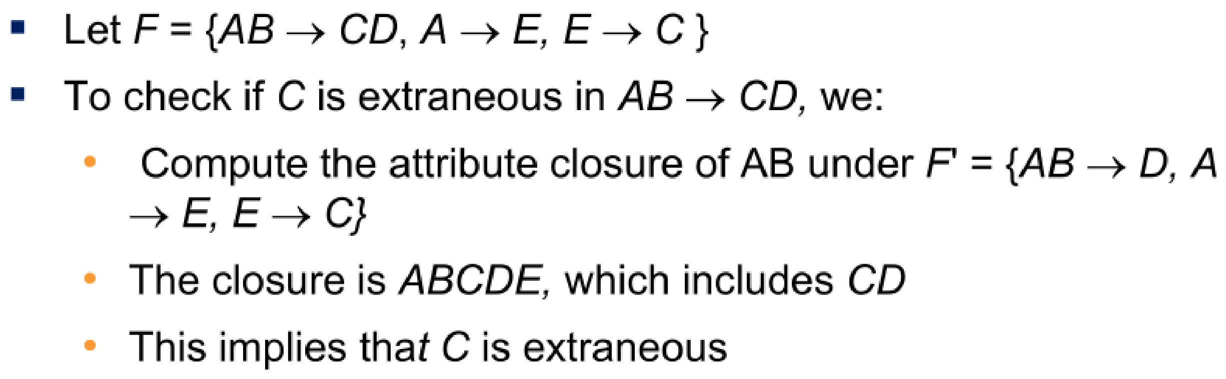


- $R = (A, B, C)$
  $F = \{A \rightarrow B, B \rightarrow C)$
- $R_1 = (A, B), \quad R_2 = (B, C)$
  - Lossless decomposition:
    $R_1 \cap R_2 = \{B\}$ and $B \rightarrow BC$
- $R_1 = (A, B), \quad R_2 = (A, C)$
  - Lossless decomposition:
    $R_1 \cap R_2 = \{A\}$ and $A \rightarrow AB$
- *Note:*
  - $B \rightarrow BC$
    is a shorthand notation for
  - $B \rightarrow \{B, C\}$

Canonical Cover: A minimal set of FDs

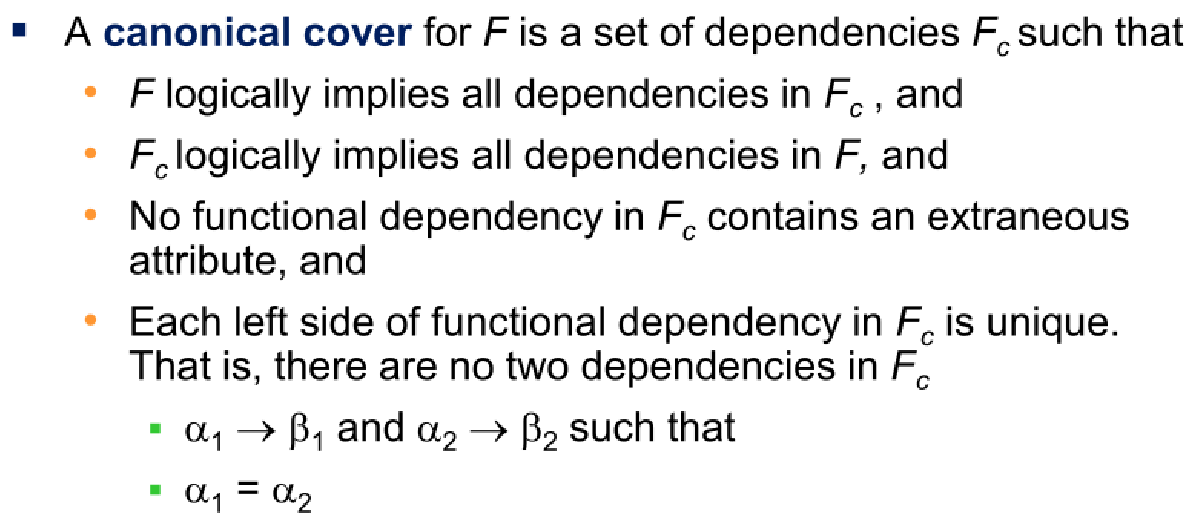
1. Suppose that we have a set of functional dependencies F on a relation schema.
2. Whenever a user performs an update on the relation, the database system must ensure that the update does not violate any functional dependencies; that is, all the functional dependencies in F are satisfied in the new database state.
   a. If an update violates any functional dependencies in the set F, the system must roll back the update.
   b. We can reduce the effort spent in checking for violations by testing a simplified set of functional dependencies that has the same closure as the given set.
   c. This simplified set is termed the canonical cover.
3. Generally we have to remove extraneous attributes.

4. An attribute of a functional dependency in F is *extraneous* if we can remove it without changing F+
    a. Example of Extraneous Attribute:



- Let $F = \{AB \rightarrow CD, A \rightarrow E, E \rightarrow C\}$
- To check if C is extraneous in $AB \rightarrow CD$, we:
  - Compute the attribute closure of AB under $F' = \{AB \rightarrow D, A \rightarrow E, E \rightarrow C\}$
  - The closure is *ABCDE,* which includes *CD*
  - This implies that *C* is extraneous

5. Definition of Canonical Cover:

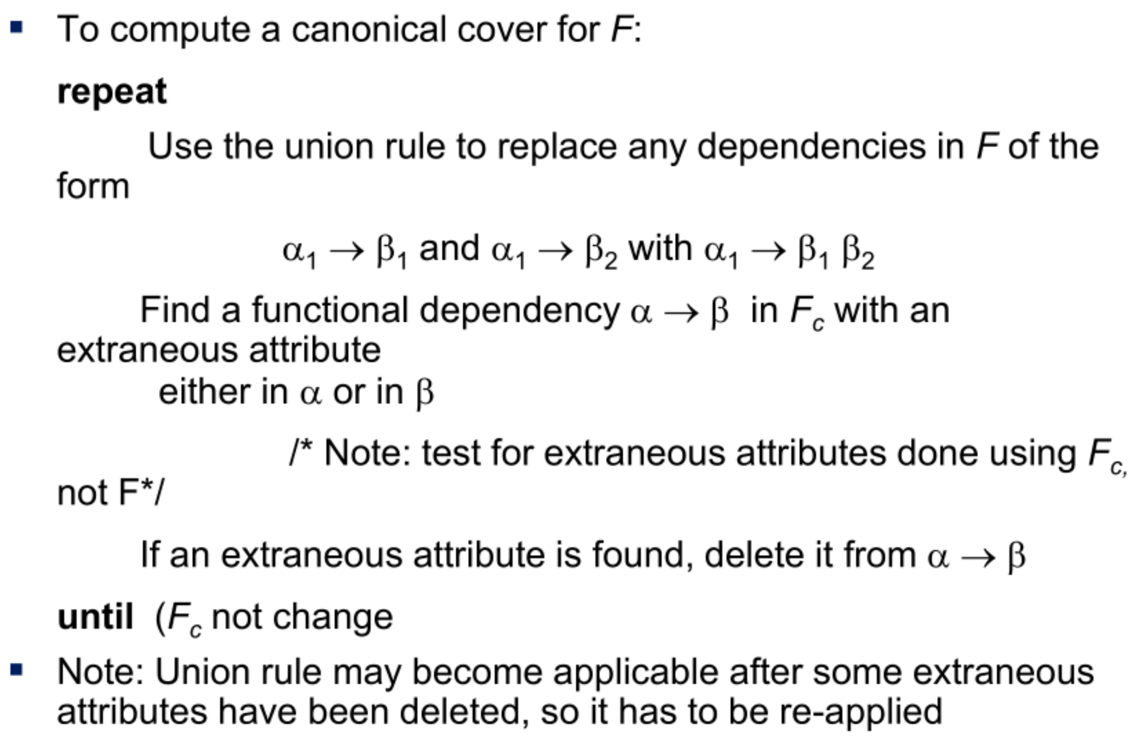


- A **canonical cover** for F is a set of dependencies F_c such that
  - F logically implies all dependencies in F_c , and
  - F_c logically implies all dependencies in F, and
  - No functional dependency in F_c contains an extraneous attribute, and
  - Each left side of functional dependency in F_c is unique. That is, there are no two dependencies in F_c
    - $\alpha_1 \rightarrow \beta_1$ and $\alpha_2 \rightarrow \beta_2$ such that
    - $\alpha_1 = \alpha_2$

6. Removing an attribute from the left side of a functional dependency could make it a stronger constraint.
    a. F = {AB → C, A → D, D → C}
    Then we can show that F logically implies A → C, making B extraneous in AB → C.
7. Removing an attribute from the right side of a functional dependency could make it a weaker constraint.

   a. For example, if we have AB → CD and remove C, we get the possibly weaker result AB → D.
   b. It may be weaker because using just AB → D, we can no longer infer AB → C.
   c. But, depending on what our set F of functional dependencies happens to be, we may be able to remove C from AB → CD
      safely.
   d. For example, suppose that F = { AB → CD, A → C}.
      i. Then we can show that even after replacing AB → CD by AB → D, we can still infer $AB → C and thus AB → CD.

10. Computing Canonical Cover:

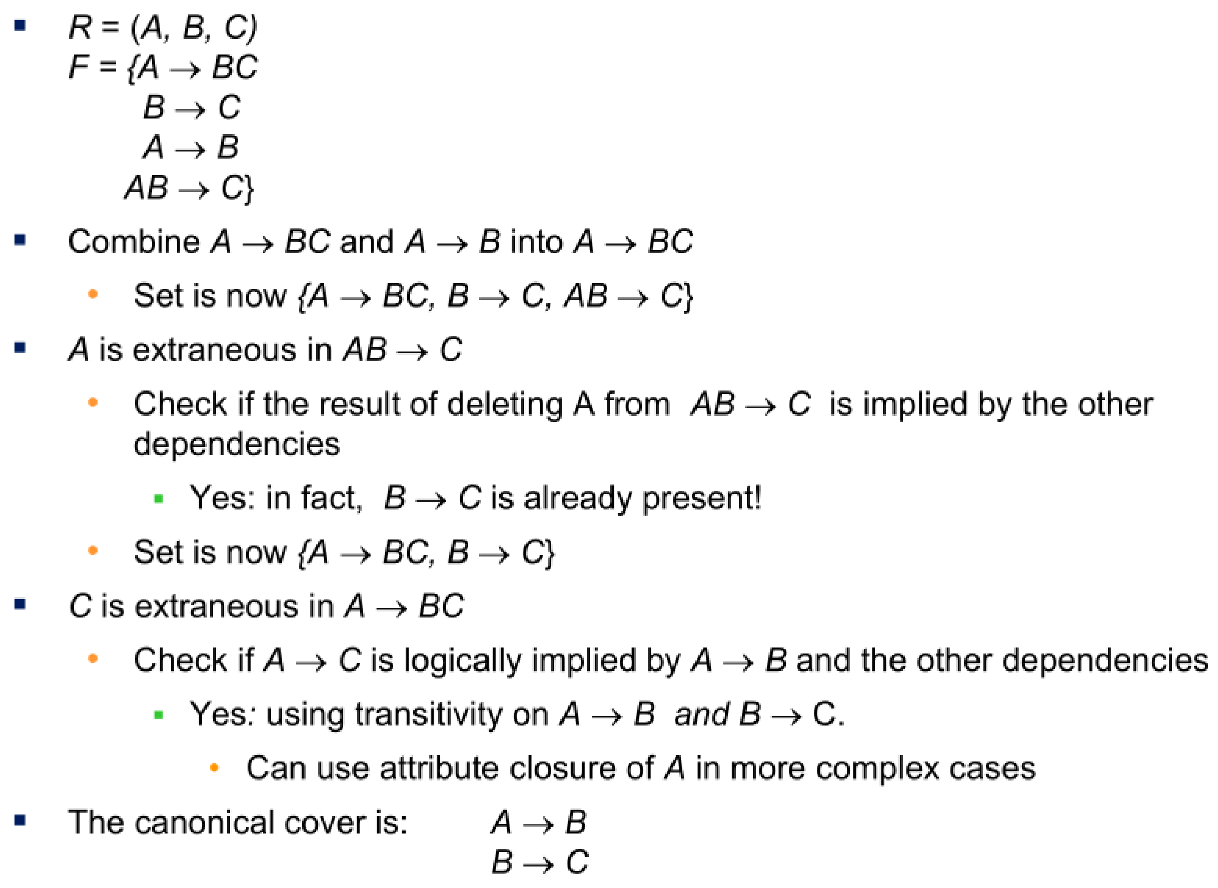


- To compute a canonical cover for *F*:

```
repeat
        Use the union rule to replace any dependencies in F of the form
                $\alpha_1 \rightarrow \beta_1$ and $\alpha_1 \rightarrow \beta_2$ with $\alpha_1 \rightarrow \beta_1 \beta_2$
        Find a functional dependency $\alpha \rightarrow \beta$ in $F_c$ with an extraneous attribute
            either in $\alpha$ or in $\beta$
                /* Note: test for extraneous attributes done using $F_c$, not F*/
        If an extraneous attribute is found, delete it from $\alpha \rightarrow \beta$
until ($F_c$ not change
```

- Note: Union rule may become applicable after some extraneous attributes have been deleted, so it has to be re-applied

11. ***<u>Example of computing a canonical cover:</u>***

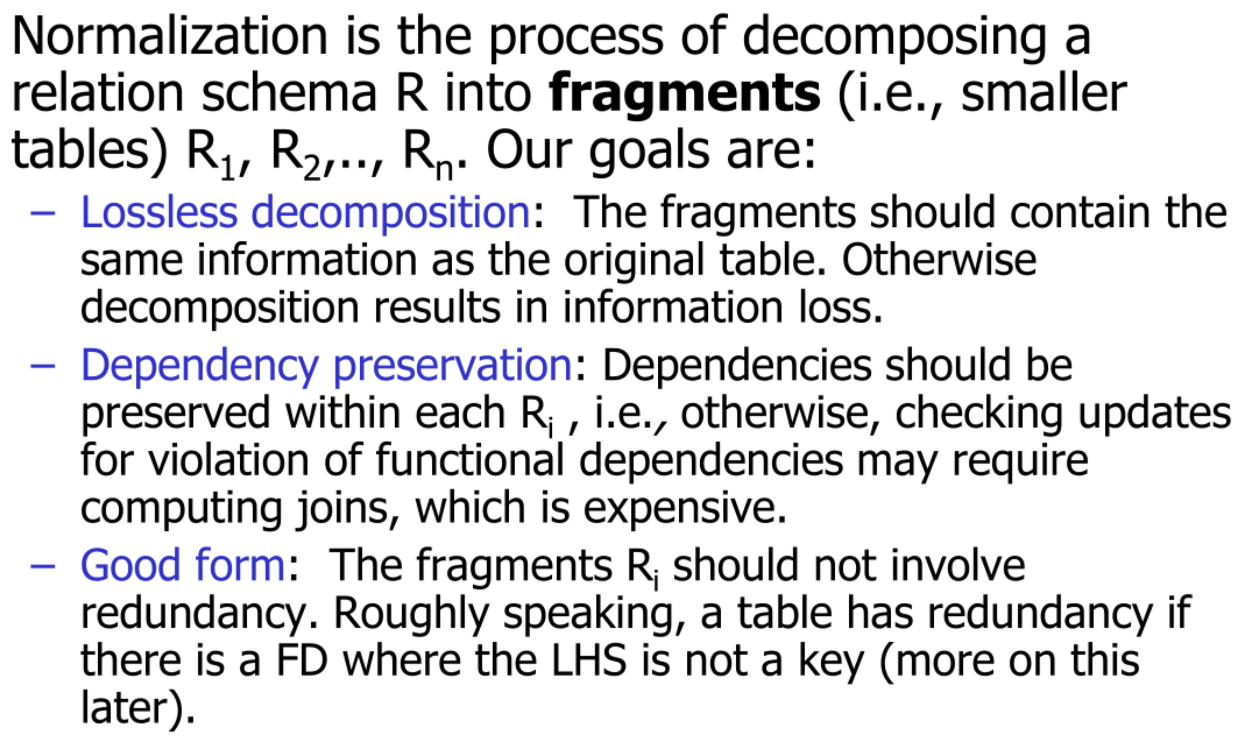


- $R = (A, B, C)$
  $F = \{A \rightarrow BC$
  $B \rightarrow C$
  $A \rightarrow B$
  $AB \rightarrow C\}$
- Combine $A \rightarrow BC$ and $A \rightarrow B$ into $A \rightarrow BC$
  - Set is now $\{A \rightarrow BC, B \rightarrow C, AB \rightarrow C\}$
- A is extraneous in $AB \rightarrow C$
  - Check if the result of deleting A from $AB \rightarrow C$ is implied by the other dependencies
    - Yes: in fact, $B \rightarrow C$ is already present!
  - Set is now $\{A \rightarrow BC, B \rightarrow C\}$
- C is extraneous in $A \rightarrow BC$
  - Check if $A \rightarrow C$ is logically implied by $A \rightarrow B$ and the other dependencies
    - Yes: using transitivity on $A \rightarrow B$ and $B \rightarrow C$.
      - Can use attribute closure of A in more complex cases
- The canonical cover is: $A \rightarrow B$
  $B \rightarrow C$

Dependency Preservation:

1. If testing a functional dependency can be done by considering just one relation, then the cost of testing this constraint is low
2. When decomposing a relation it is possible that it is no longer possible to do the testing without having to perform a Cartesian Product/join.
3. A decomposition that makes it computationally hard to enforce functional dependency is said to be NOT dependency preserving.

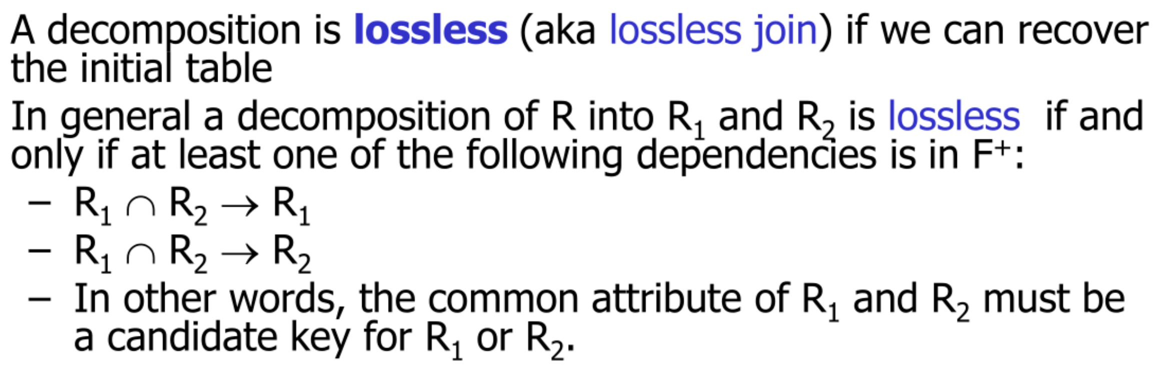
5. Normalization Theory:



Normalization is the process of decomposing a relation schema R into **fragments** (i.e., smaller tables) R_1, R_2,.., R_n. Our goals are:

- Lossless decomposition: The fragments should contain the same information as the original table. Otherwise decomposition results in information loss.
- Dependency preservation: Dependencies should be preserved within each R_i , i.e., otherwise, checking updates for violation of functional dependencies may require computing joins, which is expensive.
- Good form: The fragments R_i should not involve redundancy. Roughly speaking, a table has redundancy if there is a FD where the LHS is not a key (more on this later).

8. Lossless Join:

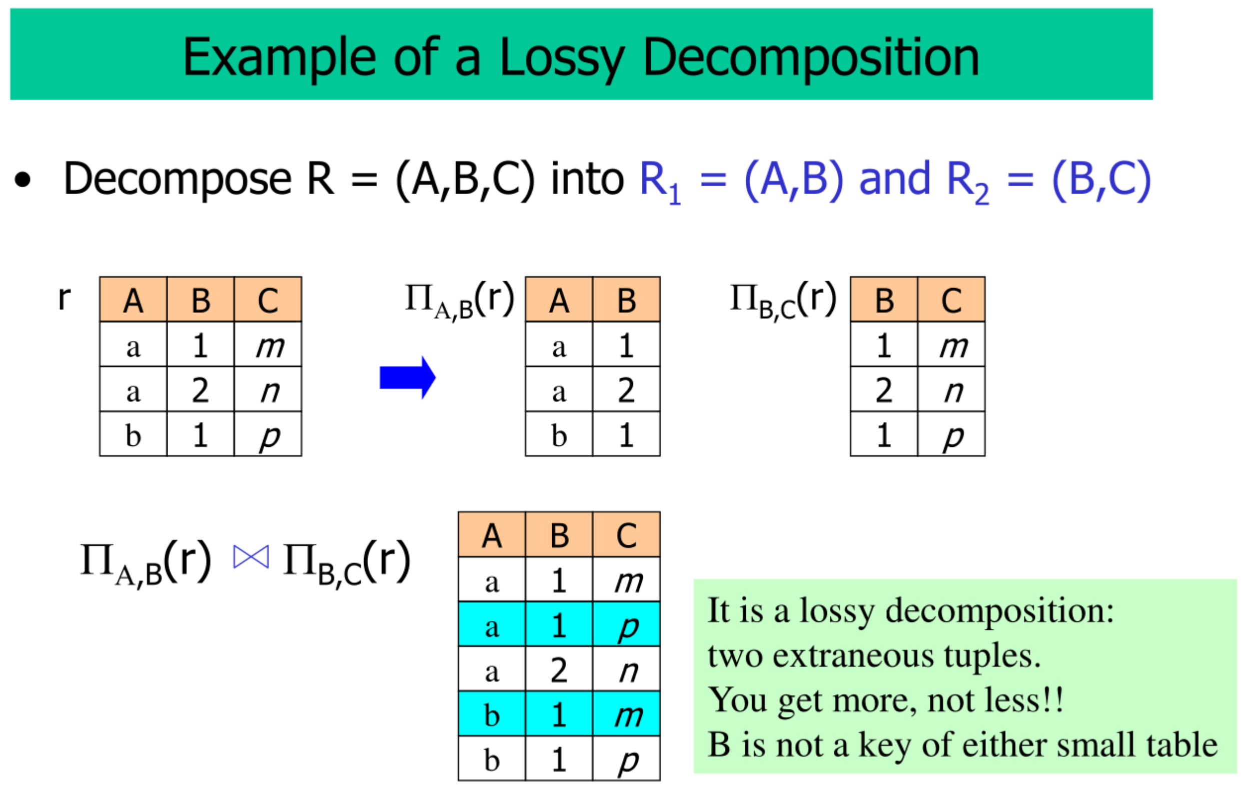


A decomposition is **lossless** (aka lossless join) if we can recover the initial table

In general a decomposition of R into R_1 and R_2 is lossless if and only if at least one of the following dependencies is in F^+:

- $R_1 \cap R_2 \rightarrow R_1$
- $R_1 \cap R_2 \rightarrow R_2$
- In other words, the common attribute of R_1 and R_2 must be a candidate key for R_1 or R_2.

9. Lossy:



Example of a Lossy Decomposition

- Decompose R = (A,B,C) into R_1 = (A,B) and R_2 = (B,C)

r

A	B	C
a	1	*m*
a	2	*n*
b	1	*p*

$\Pi_{A,B}(r)$

A	B
a	1
a	2
b	1

$\Pi_{B,C}(r)$

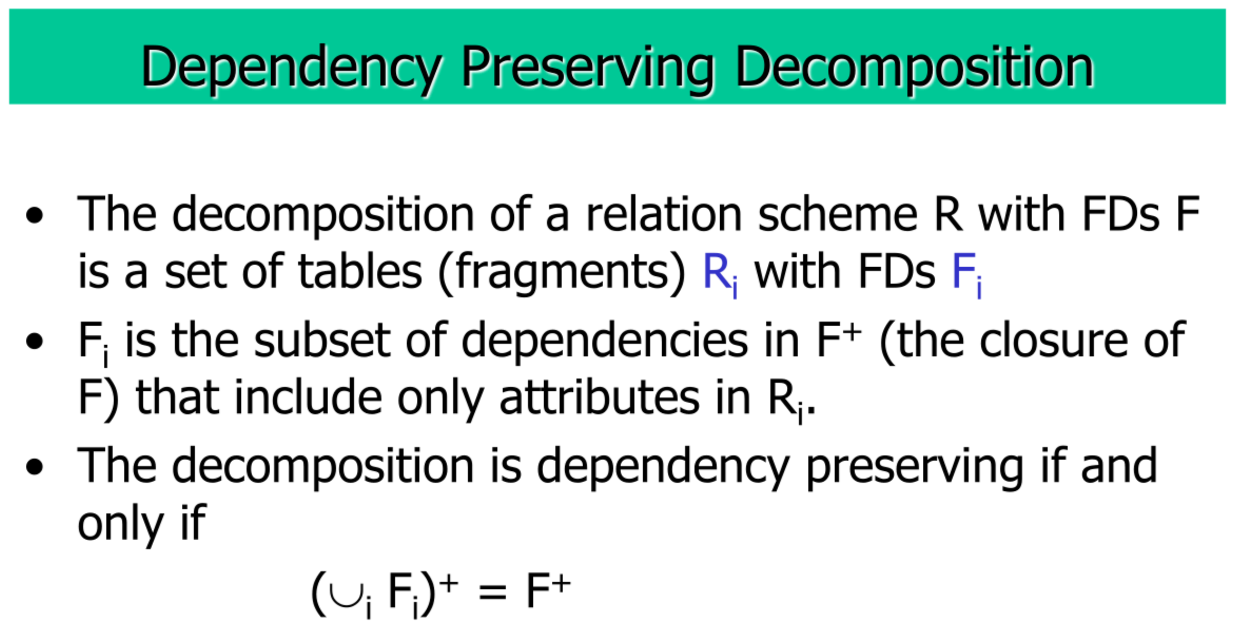
B	C
1	*m*
2	*n*
1	*p*

$\Pi_{A,B}(r) \bowtie \Pi_{B,C}(r)$

A	B	C
a	1	*m*
a	1	*p*
a	2	*n*
b	1	*m*
b	1	*p*

It is a lossy decomposition:
two extraneous tuples.
You get more, not less!!
B is not a key of either small table

10. **Lossless: AC, BC; C is a key for both of them!**
11. DP:

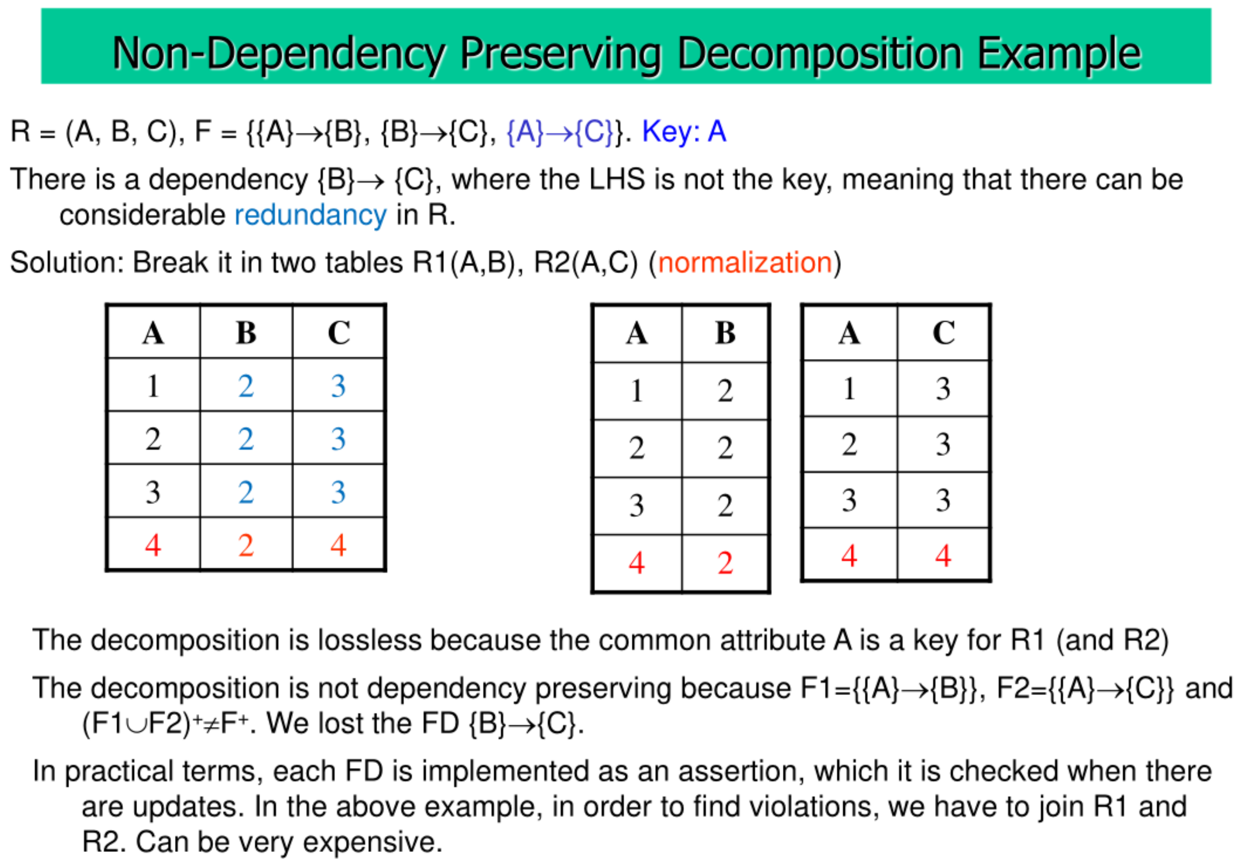


Dependency Preserving Decomposition

- The decomposition of a relation scheme R with FDs F is a set of tables (fragments) R_i with FDs F_i
- F_i is the subset of dependencies in F^+ (the closure of F) that include only attributes in R_i.
- The decomposition is dependency preserving if and only if

$$(\cup_i F_i)^+ = F^+$$

12. Non-DP example:



Non-Dependency Preserving Decomposition Example

R = (A, B, C), F = {{A}→{B}, {B}→{C}, {A}→{C}}. Key: A

There is a dependency {B}→ {C}, where the LHS is not the key, meaning that there can be considerable redundancy in R.

Solution: Break it in two tables R1(A,B), R2(A,C) (normalization)

A	B	C
1	2	3
2	2	3
3	2	3
4	2	4

A	B
1	2
2	2
3	2
4	2

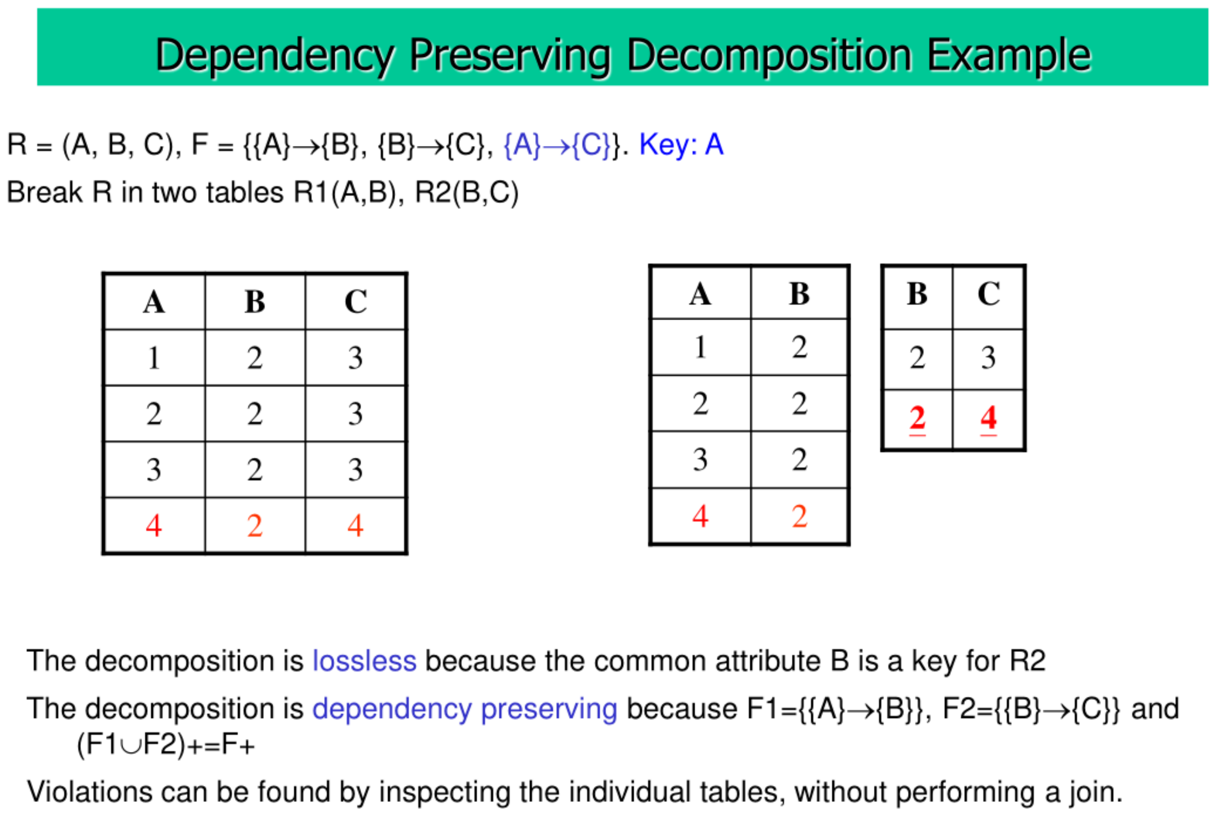
A	C
1	3
2	3
3	3
4	4

The decomposition is lossless because the common attribute A is a key for R1 (and R2)

The decomposition is not dependency preserving because F1={{A}→{B}}, F2={{A}→{C}} and $(F1 \cup F2)^+ \neq F^+$. We lost the FD {B}→{C}.

In practical terms, each FD is implemented as an assertion, which it is checked when there are updates. In the above example, in order to find violations, we have to join R1 and R2. Can be very expensive.

13. DP example:



Dependency Preserving Decomposition Example

R = (A, B, C), F = {{A}→{B}, {B}→{C}, {A}→{C}}. Key: A

Break R in two tables R1(A,B), R2(B,C)

A	B	C
1	2	3
2	2	3
3	2	3
4	2	4

A	B
1	2
2	2
3	2
4	2

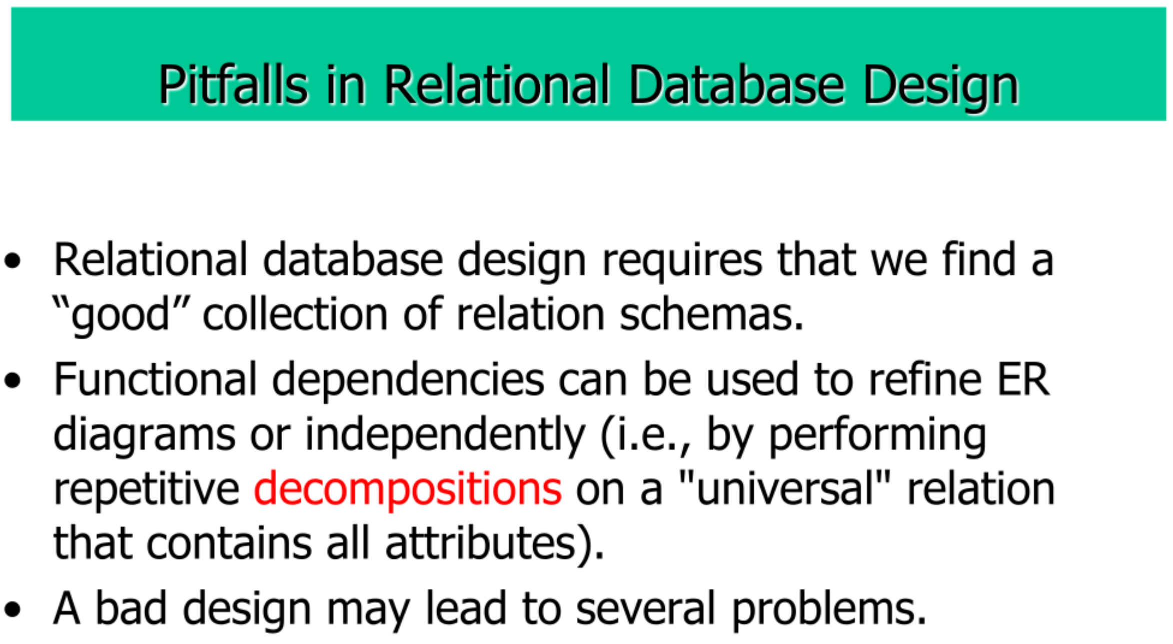
B	C
2	3
2	**4**

The decomposition is lossless because the common attribute B is a key for R2

The decomposition is dependency preserving because F1={{A}→{B}}, F2={{B}→{C}} and (F1∪F2)+=F+

Violations can be found by inspecting the individual tables, without performing a join.

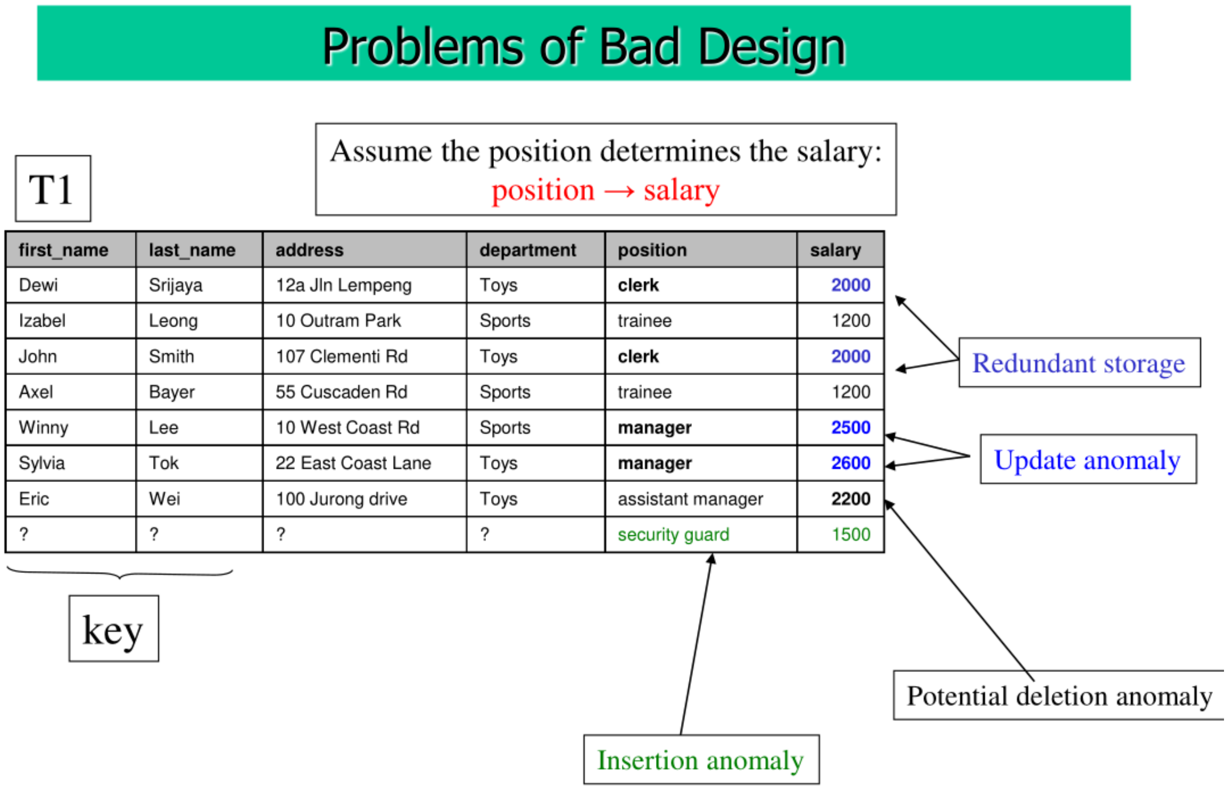
14. Potential Problems:



Pitfalls in Relational Database Design

- Relational database design requires that we find a "good" collection of relation schemas.
- Functional dependencies can be used to refine ER diagrams or independently (i.e., by performing repetitive decompositions on a "universal" relation that contains all attributes).
- A bad design may lead to several problems.

15. Bad design:



Problems of Bad Design

T1

Assume the position determines the salary:
position → salary

first_name	last_name	address	department	position	salary
Dewi	Srijaya	12a Jln Lempeng	Toys	**clerk**	**2000**
Izabel	Leong	10 Outram Park	Sports	trainee	1200
John	Smith	107 Clementi Rd	Toys	**clerk**	**2000**
Axel	Bayer	55 Cuscaden Rd	Sports	trainee	1200
Winny	Lee	10 West Coast Rd	Sports	**manager**	**2500**
Sylvia	Tok	22 East Coast Lane	Toys	**manager**	**2600**
Eric	Wei	100 Jurong drive	Toys	assistant manager	**2200**
?	?	?	?	security guard	1500

16. Decomposition Example:

Decomposition Example

T2

first_name	last_name	address	department	position
Dewi	Srijaya	12a Jln lempeng	Toys	clerk
Izabel	Leong	10 Outram Park	Sports	trainee
John	Smith	107 Clementi Rd	Toys	clerk
Axel	Bayer	55 Cuscaden Rd	Sports	trainee
Winny	Lee	10 West Coast Rd	Sports	manager
Sylvia	Tok	22 East Coast Lane	Toys	manager
Eric	Wei	100 Jurong drive	Toys	assistant manager

T3

position	salary
clerk	2000
trainee	1200
manager	2500
assistant manager	2200
security guard	1500

- No Redundant storage
- No Update anomaly
- No Deletion anomaly
- No Insertion anomaly