CPSC 304
Introduction to Database Systems

Schema Refinement and Normal Forms

Textbook Reference
Database Management Systems
Reading: Chapter 19. Concentrate on sections19.1-19.6 (except 19.5.2)

Databases – the continuing saga

- We've learned that databases are wonderful things
- We've learned how to create a *conceptual design* using ER diagrams
- We've learned how to create a *logical design* by turning the ER diagrams into a relational schema including minimizing the data and relations created
- Now we need to *refine* that schema to reduce duplication of information

Learning Goals

- Debate the pros and cons of redundancy in a database.
- Provide examples of update, insertion, and deletion anomalies.
- Given a set of tables and a set of functional dependencies over them, determine all the keys for the tables.
- Show that a table is/isn't in 3NF or BCNF.
- Prove/disprove that a given table decomposition is a lossless join decomposition. Justify why lossless join decompositions are preferred decompositions.
- Decompose a table into a set of tables that are in 3NF, or BCNF.

Imagine that we've created a perfectly good entity for mailing addresses at UBC:

Instance of the schema on the previous slide

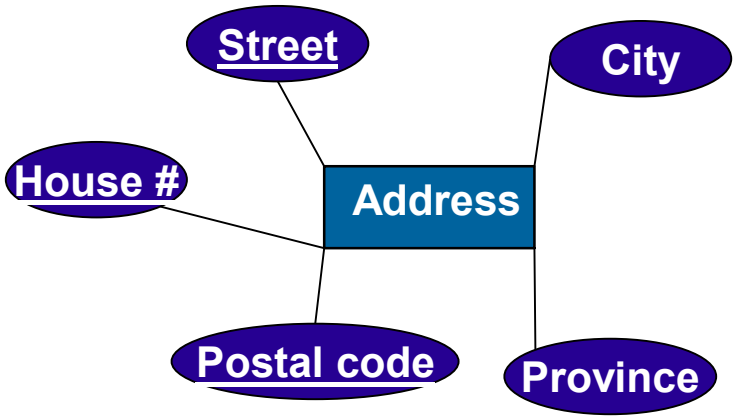
Name	Department	Mailing Location
Jessica Wong	Computer Science	201-2366 Main Mall
Rachel Pottinger	Computer Science	201-2366 Main Mall
Joel Friedman	Computer Science	201-2366 Main Mall
Joel Friedman	Math	121-1984 Mathematics Rd
Mark MacLean	Math	121-1984 Mathematics Rd

What are some problems that might happen with this design?
Based solely on intuition, what might be a better design?

Okay, that's bad. But how do I know for sure if departments have only one address?

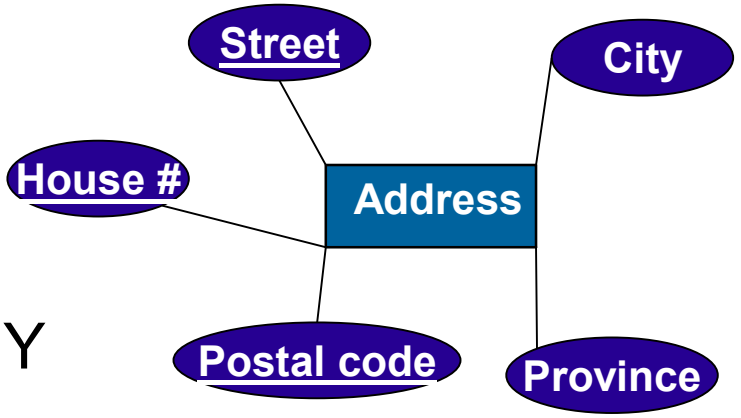
- Databases allow you to say that one attribute determines another through a *functional dependency*.
- So if Department determines Address but not Name, we say that there's a functional dependency from Department to Address. But Department is NOT a key.

- Another example:
  Address(House#, Street, City, Province, PostalCode)



Functional Dependencies (FDs) – technically speaking

- A **functional dependency** **X→Y**
  (where X & Y are sets of attributes)
  holds if for every legal instance,
  for all tuples *t1, t2* :
  if *t1*.X = *t2*.X *then* *t1*.Y = *t2*.Y



- Example:
  PostalCode → City, Province if:
  for each possible t1, t2, if t1.PostalCode = t2.PostalCode then
  t1.{City,Province} = t2.{City,Province})
- i.e., given two tuples in *r*, if the X values agree, then the Y values must also agree
- Also can be read as X *determines* Y

Let's see some more instances

House #	Street	City	Province	Postal Code
101	Main Street	Vancouver	BC	V6A 2S5
103	Main Street	Vancouver	BC	V6A 2S5
101	Cambie Street	Vancouver	BC	V6B 4R3
103	Cambie Street	Vancouver	BC	V6B 4R3
101	Main Street	Delta	BC	V4C 2N1
103	Main Street	Delta	BC	V4C 2N1
800	Benvenuto Ave	Victoria	BC	V8M 1J8
165	Duckworth Street	Victoria	NF	A1B 4R5

Which functional dependencies do we care about?

- A FD is a statement about *all* allowable instances.
  - Must be identified by application semantics
  - Given some instance *r1* of R, we can check if r1 violates some FD *f*, but we cannot tell if *f* holds over R!
- There are boring, *trivial* cases:
  - e.g. PostalCode, House# → PostalCode
- We'll concentrate on the non-boring ones and on cases where there's a single attribute on the RHS[1]: (e.g., PostalCode → Province and PostalCode → City)

1. RHS = Right Hand Side. LHS = Left Hand Side

Naming the Evils of Redundancy

Let's consider Postal Code → City, Province

House #	Street	City	Province	Postal Code
101	Main Street	Vancouver	BC	V6A 2S5
103	Main Street	Vancouver	BC	V6A 2S5
101	Cambie Street	Vancouver	BC	V6B 4R3
103	Cambie Street	Vancouver	BC	V6B 4R3
101	Main Street	Delta	BC	V4C 2N1
103	Main Street	Delta	BC	V4C 2N1

- Update anomaly: Need to change more than 1 thing to stay consistent. Can we change Delta's province?
- Insertion anomaly: attributes cannot be inserted without the presence of other attributes. What if we want to insert that V6T 1Z4 is in Vancouver?
- Deletion anomaly: attributes are lost because of deletion of other attributes. If we delete all addresses with V6A 2S5, we lose that V6A 2S5 is in Vancouver!

Once more, with feeling

House #	Street	Postal Code
101	Main Street	V6A 2S5
103	Main Street	V6A 2S5
101	Cambie Street	V6B 4R3
103	Cambie Street	V6B 4R3
101	Main Street	V4C 2N1
103	Main Street	V4C 2N1

City	Province	Postal Code
Vancouver	BC	V6A 2S5
Vancouver	BC	V6B 4R3
Delta	BC	V4C 2N1

- Did we lose anything?
- Are our problems fixed?

As it turns out, this is a pretty optimal way to split the table up, though there's still a little redundancy (e.g., Vancouver being in BC is in the table twice). Let's see why…

What do we need to know to split apart addresses without losing information?

- FDs tell us when we're storing redundant information
- Reducing redundancy helps eliminate anomalies and save storage space
- We'd like to split apart tables without losing information
- But first, we need to know both what FDs are *explicit* (given) and what FDs are *implicit* (can be derived)
- Among other things, this can help us derive additional keys from the given keys (spare keys are handy in databases, just like in real life – we'll see why shortly)

The Key is the Thing

- As a reminder, a key is a *minimal* set of attributes that uniquely identify a relation
  - i.e., a key is a minimal set of attributes that *functionally determines* all the attributes
  - e.g., House#, Street, PostalCode is a key for Address
- A superkey for a relation uniquely identifies the relation, but does not have to be minimal
  - i.e.,: key $\subseteq$ superkey
  - E.g.,:
    - House#, Street, PostalCode is a key and a superkey
    - House#, Street, PostalCode, city is a superkey, but not a key

Let's summarize:

- In a functional dependency, a set of attributes determines other attributes, e.g., AB→C, means A and B together determine C
- A trivial FD determines what you already have, eg., AB→B
- A key is a minimal set of attributes determining the rest of the attributes of a relation
  For example, R(House #, Street, City, Province, Postal Code)
- A superkey is a set of attributes determining the rest of the attributes in the relation, but does NOT have to be minimal (e.g., the key above, or adding in either of City and Province)
- Given a set of (explicit) functional dependencies, we can determine others…

Deriving Additional FDs: the basics

- Given some FDs, we can often infer additional FDs:
  - *studentid* → *city*, *city* → *acode* implies *studentid* → *acode*
- An FD *fd* is ***implied by*** a set of FDs *F* if *fd* holds whenever all FDs in *F* hold.
  - ***closure of F*** : the set of all FDs implied by *F*.
- Armstrong's Axioms (X, Y, Z are sets of attributes):
  - *Reflexivity*: If $Y \subseteq X$, then $X \rightarrow Y$
    e.g., city,major→city
  - *Augmentation*: If $X \rightarrow Y$, then $XZ \rightarrow YZ$ for any Z
    e.g., if sid→city, then sid,major → city,major
  - *Transitivity*: If $X \rightarrow Y$ and $Y \rightarrow Z$, then $X \rightarrow Z$
    sid → *city*, *city* → *areacode* implies *sid* → *areacode*
- These are *sound* and *complete* inference rules for FDs.

Deriving Additional FDs: the extended dance remix

- A couple of additional rules (that follow from axioms):
  - ***Union***: If X→Y and X→Z, then X→Y Z
    e.g., if sid→acode and sid→city, then sid→acode,city
  - ***Decomposition***: If X→Y Z, then X→Y and X→Z
    e.g., if sid→acode,city then sid→acode, and sid→city

Example: Derive union rule from axioms (Reflexivity, Augmentation, & Transitivity)

Union: If X→Y and X→Z, then X→Y Z
e.g., if sid→acode and sid→city, then sid→acode,city

1. X→Y Given
2. X→Z Given
3. X→XY 1, augmentation
4. XY→ZY 2, augmentation
5. X→ZY 3, 4, transitivity

Ex: Derive Decomposition from axioms (Reflexivity, Augmentation, & Transitivity)

Decomposition: If X→Y Z, then X→Y and X→Z
e.g., if sid→acode,city then sid→acode, and sid→city

1. X→YZ Given
2. YZ→Y Reflexivity
3. YZ→Z Reflexivity
4. X→Y 1, 2, transitivity
5. X→Z 1, 3, transitivity

All 5 Rules

- ***Reflexivity***: If $Y \subseteq X$, then $X \rightarrow Y$
  e.g., city,major→city
- ***Augmentation***: If $X \rightarrow Y$, then $XZ \rightarrow YZ$ for any Z
  e.g., if sid→city, then sid,major → city,major
- ***Transitivity***: If $X \rightarrow Y$ and $Y \rightarrow Z$, then $X \rightarrow Z$
  sid → city, *city → areacode* implies *sid → areacode*
- ***Union***: If X→Y and X→Z, then X→Y Z
  e.g., if sid→acode and sid→city, then sid→acode,city
- ***Decomposition***: If X→Y Z, then X→Y and X→Z
  e.g., if sid→acode,city then sid→acode, and sid→city

Example: Supplier-Part DB

- Suppliers supply parts to projects.
  - SupplierPart(sname,city,status,p#,pname,qty)
    - supplier attributes: sname, city, status
    - part attributes: p#, pname
    - supplier-part attributes: qty
- Functional dependencies:
  - fd1: sname → city
  - fd2: city → status
  - fd3: p# → pname
  - fd4: sname, p# → qty
- How can we show that (sname, p#) is a key?

Supplier-Part Key: Part 1: Determining all attributes

Show that (sname, p#) is a superkey of SupplierPart(sname,city,status,p#,pname,qty)
Proof has two parts:

a. Show: sname, p# is a (super)key

1.	sname, p# → sname, p#	reflex
2.	sname → city	fd1
3.	sname → status	2, fd2, trans
4.	sname,p# → city, p#	2, aug
5.	sname,p# → status, p#	3, aug
6.	sname,p# → sname, p#, status	1, 5, union
7.	sname,p# → sname, p#, status, city	4, 6, union
8.	sname,p# → sname, p#, status, city, qty	7, fd4, union
9.	sname,p# → sname, p#, status, city, qty, pname	8, fd3, union

fd1: sname → city
fd2: city → status
fd3: p# → pname
fd4: sname, p# → qty

Supplier-Part Key: Part 2: Minimalness

b. Show: (sname, p#) is a *minimal* key of SupplierPart(sname,city,status, p#,pname,qty)

1. p# does not appear on the RHS of any FD therefore except for p# itself, nothing else determines p#
3. specifically, sname → p# does not hold
4. therefore, sname is not a key
5. similarly, p# is not a key

fd1:	sname → city
fd2:	city → status
fd3:	p# → pname
fd4:	sname, p# → qty

Specifically, for combinations of sname and p#:

- sname,p# →sname, p#, city, status, pname, qty
- sname → sname, city, status
- p# → p#, pname

Do you, by any chance, have anything less painful?

- Scared you're going to mess up? *Closure* is a fool-proof method of checking FDs and finding implicit FDs.
- *Closure of* a set of attributes *X* is denoted X^+
- X^+ includes all attributes of the relation IFF X is a (super)key
- Algorithm for finding Closure of X:
  Let Closure = X
  Until Closure doesn't change do
  if $a_1, \ldots, a_n \rightarrow C$ is a FD and $\{a_1, \ldots, a_n\} \in$ Closure
  Then add C to Closure

fd1:	sname → city
fd2:	city → status
fd3:	p# → pname
fd4:	sname, p# → qty

SupplierPart(sname,city,status,p#,pname,qty)

Ex: $sname,p\#^+$ =

$sname^+$ =

$p\#^+$ =

So seeing if a set of attributes is a key means checking to see if it's closure is all the attributes – pretty simple

Let's take a minute to appreciate what closures give us

- Consider relation R(A,B,C,D) with FDs
  AB→C
  C→D
- Closures give us:
  AB^+ = {A,B,C,D}
  C^+ = {C,D}
- This tells us all the FDs, both explicit and implied:
  AB→A
  AB→B
  AB→C
  AB→D
  C→C
  C→D
- But how can we be sure that we've found all the keys (I promise you we will need this later) without resorting to Armstrong's axioms?

Finding All the Minimal Keys

Find all the minimal keys for R(ABCD) where
AB → C and C → BD.

Step 1: List all the attributes that **only** appear on the RHS of the FDs (i.e., things that can only be derived). Put on right:

Left	Middle	Right
		D

Step 2: List all the attributes that **only** ever appear on the LHS of a FD (or do not appear in a FD at all (i.e., things that have to be part of any key). Put on left:

Left	Middle	Right
A		D

Finding All the Minimal Keys

Find all the minimal keys for R(ABCD) where
AB → C and C → BD.

Step 3: List all the attributes that appear on the LHS and RHS of the FDs (i.e., things that are not obviously required and may help you derive new things). Put in middle.

Left	Middle	Right
A	BC	D

Step 4: Take the closure of the attributes in the left column.

$\{A\}^+ = \{A\}$

Are all of the attributes there? If so, you have found a minimal key. If not, start adding in attributes from the middle column to see if you can determine all the attributes of the relation.

Finding All the Minimal Keys

Find all the minimal keys for R(ABCD) where
AB → C and C → BD.

Left	Middle	Right
A	BC	D
Need	Maybe need	Don't need (can derive)

$\{AB\}^+ = \{ABCD\}$ ← minimal key

$\{AC\}^+ = \{ACBD\}$ ← minimal key

- What if the relation schema had attribute E? R(ABCDE)

Exercise: Find all Keys

Find all the minimal keys of the Relation R(ABCDE) with FD's:

$D \rightarrow C$

$CE \rightarrow A$,

$D \rightarrow A$

$AE \rightarrow D$.

Popping back up to our original question…

- Is this really a good design for a table?

Name	Department	Mailing Location
Jessica Wong	Computer Science	201-2366 Main Mall
Rachel Pottinger	Computer Science	201-2366 Main Mall
Laks Lakshmanan	Computer Science	201-2366 Main Mall
Joel Friedman	Computer Science	201-2366 Main Mall
Joel Friedman	Math	121-1984 Mathematics Rd
Mark MacLean	Math	121-1984 Mathematics Rd

- Wouldn't it be nice if there was some rule that said if the amount of redundancy that we had was good?

Approaching Normality

- Role of FDs in detecting redundancy:
  - Consider a relation R with 3 attributes, A B C.
    - No FDs hold: There is no redundancy here.
    - Given A → B: Several tuples could have the same A value, and if so, they'll all have the same B value!
- ***Normalization***: the process of removing redundancy from data

Normal Forms: Why have one rule when you can have four?

- Provide guidance for table refinement/reducing redundancy.
- Four important normal forms:
  - ***First normal form(1NF)***
  - ***Second normal form (2NF)***
  - ***Boyce-Codd Normal Form (BCNF)***
  - ***Third normal form (3NF)***
- If a relation is in a certain *normal form*, certain problems are avoided/minimized.
- Normal forms can help decide whether decomposition (i.e., splitting tables) will help.

1NF

- Each attribute in a tuple has only one value
  - E.g., for “postal code” you can’t have both V6T 1Z4 and V6S 1W6
- Why do we need 1NF?
  - because Codd’s original vision of the relational model allowed multi-valued attributes…

1NF

Client ID	Postal Code
1	V6T 1Z4, V6S 1W6

Can't have multiple values in a single attribute

Normalize to 1NF

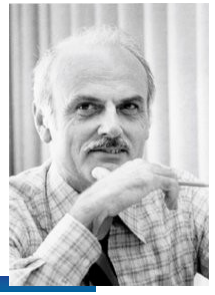
Client ID	Postal Code
1	V6T 1Z4
1	V6S 1W6

2NF

- No partial key dependency
- A relation is in 2NF, if it is in 1NF and for every FD X→Y where X is a (minimal) key and Y is a non-key attribute, then no proper subset of X determines Y
- e.g., the address relation is not in 2NF:
  - House#, street, postal_code is a (minimal) key
  - House#, street, postal_code → Province (X → Y)
  - Postal_code → Province (Subset of X → Y)

Not in 2NF

Boyce-Codd Normal Form (BCNF)



A relation R is in BCNF if:

If X → b is a non-trivial dependency in R,
then X is a superkey for R
(Must be true for every such dependency)

Recall: A dependency is trivial if the LHS contains the RHS, e.g., City, Province→ City is a trivial dependency

In English (though a bit vague):

Whenever a set of attributes of *R* determine another attribute, it should determine ***all*** the attributes of *R*.

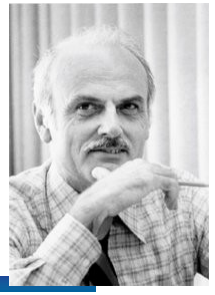
What do we want? Guaranteed freedom from redundancy!

- A relation may be BCNF already!
- Helpful fact: all two attribute relations are in BCNF. Consider relation S(A,B). There are four cases:
  - A→B and B→A
  - Only A→B
  - Only B→A
  - No FDs
- Lets look at each one

Yet Another Closure Example

- Given relationship R(A,B,C,D,E) and FDs:
  AB→C
  CD→E
  BE→A
  Take the closures of the given functional dependencies

Reminder
Boyce-Codd Normal Form (BCNF)



A relation R is in BCNF if:

If X → b is a non-trivial dependency in R,
then X is a superkey for R
(Must be true for every such dependency)

Recall: A dependency is trivial if the LHS contains the RHS, e.g., City, Province→ City is a trivial dependency

In English (though a bit vague):

Whenever a set of attributes of *R* determine another attribute, it should determine ***all*** the attributes of *R*.

What if a relation is not in BCNF? Decomposing a Relation

"Shhhhh!... the Maestro is decomposing!"

- A *decomposition* of R replaces R by two or more relations s.t.:
  - Each new relation contains a subset of the attributes of R (and no attributes not appearing in R), and
  - Every attribute of R appears in at least one new relation.
- Intuitively, decomposing R means storing instances of the relations produced by the decomposition, instead of instances of R.
- E.g., Address(House#,Street,City,Province,Postal Code)
  - Address(House#,Street#,PostalCode),
  - PC(City, Province, PostalCode)

How can we decompose correctly?

A sneak preview: The join

- Definition: $R_1 \bowtie R_2$ is the (natural) join of the two relations; i.e., each tuple of R_1 is concatenated with every tuple in R_2 having the same values on the common attributes.

R_1

A	B
1	2
4	5
7	2

R_2

B	C
2	3
5	6
2	8

$R_1 \bowtie R_2$

A	B	C
1	2	3
1	2	8
4	5	6
7	2	3
7	2	8

Lossless-Join Decompositions: Definition

Informally: If we break a relation, R, into bits, when we put the bits back together, we should get exactly R back again

Formally: Decomposition of R into X and Y is lossless-join w.r.t. a set of FDs F if, for every instance r that satisfies F:

- If we JOIN the X-part of r with the Y-part of r the result is exactly r

- It is *always* true that r is a subset of the JOIN of its X-part and Y-part, even if the join isn't lossless
- In general, the other direction does not hold – you may get back additional information! If it does hold, the decomposition is a lossless-join.
- Note: The word loss in lossless refers to loss of information, not to loss of tuples. In fact, for "loss of information" a better way to refer to it might be "addition of spurious information".

Example Lossy-Join Decomposition

A	B	C
1	2	3
4	5	6
7	2	8

decompose

A	B
1	2
4	5
7	2

B	C
2	3
5	6
2	8

join

A	B	C
1	2	3
4	5	6
7	2	8
1	2	8
7	2	3

Last two rows are not in the original.

All decompositions used to resolve redundancy *must* be lossless!

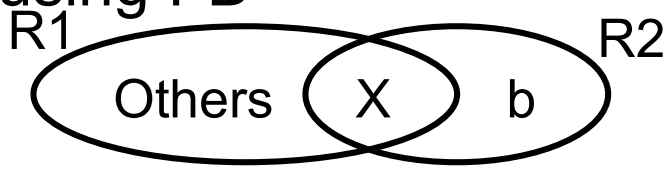
How do we decompose into BCNF losslessly?

- Let R be a relation with attributes A, and FD be a set of FDs on R s.t. all FDs determine a single attribute

1. Pick any f $\in$ FD that violates BCNF of the form X→b
2. Decompose R into two relations: R_1(A-b) & R_2(X $\cup$ b)

- Recurse on R_1 and R_2 using FD

Pictorally:



Note: answer may vary depending on order you choose. That's okay

BCNF Example (Let's do the first one together)

Remember BCNF def: For all non-trivial functional dependencies X→b, X must be a superkey for a relation to be in BCNF

Relation: R(ABCD) FD: B→C, D→A

Keys?

B^+ = {B,C} [B → B, B → C] ← What the closure tells us

D^+ = {A,D} [D → A, D → D] ← What the closure tells us

BD^+ = {B,D,C,A}

BD is the only key

BCNF Example (Let's do the first one together)

Remember BCNF def: For all non-trivial functional dependencies X→b, X must be a superkey for a relation to be in BCNF

Relation: R(ABCD) FD: B→C, D→A

Keys?

$A^+ = \{A\}$

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

$C^+ = \{C\}$

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

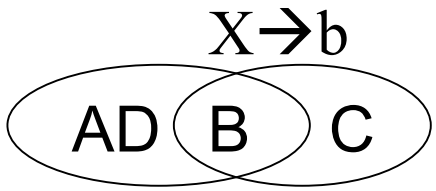
$BD^+ = \{B,D,C,A\}$

BD is the only key

Look at FD B→ C. Is B a superkey?

No. Decompose

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



BCNF Example (Let's do the first one together)

Remember BCNF def: For all non-trivial functional dependencies X→b, X must be a superkey for a relation to be in BCNF

Relation: R(ABCD) FD: B→C, D→A

Keys?

$A^+ = \{A\}$

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

$C^+ = \{C\}$

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

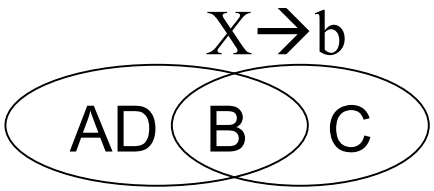
$BD^+ = \{B,D,C,A\}$

BD is the only key

Look at FD B→ C. Is B a superkey?

No. Decompose

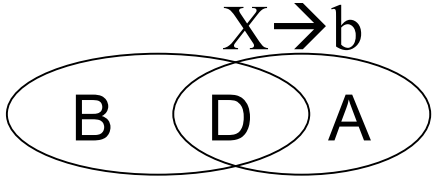
R1(B,C), R2(A,B,D)



Look at FD D→ A. Is D a superkey for R2?

No. Decompose

R3(D,A), R4(D,B)



Final answer: R1(B,C), R3(D,A), R4(D,B)

What does this answer mean?

A dependency for decomposing the second time

- When you first decide to decompose, it's easy to see which FDs apply: all of them. Returning to the previous example:
  R(ABCDE) FD: AB→C, D→E
  Clearly both FDs apply to R.
- But when you decompose to R_1(ABC), R_2(ABDE), how do we know which FDs apply to R_2?

Determining relevant FDs after decomposing

- Take the closure of the attributes using *all* FDs
- For an FD X→ b that holds in the original relation, if the decomposed relation S contains {X U b}, then the FD holds for S:
- For example. Consider the relation R(A, B, C, D, E) with functional dependencies AB→C, CD→E, C→D. Assume that you have decomposed to T(A,B,C) and S(A,B,D,E)
- Does AB→E hold on S?
  - First check if A, B and E are all in S? They are
  - Find AB^+= ABCDE ← This includes E
  - Then yes AB→ E does hold in S.
- Does AB→C hold? No ← S does not include C

Keys after a decomposition

Now that we know what FDs hold, we can determine keys

Previous example:

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

FD: AB→C, D→E

- We decomposed on AB→ C:

  R_1(<u>A,B,</u>C), R_2(A,B,D,E) (AB+ = {A,B,C}, therefore AB is a key of R_1)

- Then we decomposed on D→E

  R_3(<u>A,B,D</u>), R_4(<u>D</u>E) (D+={DE}, so D is a key of R_4. No other interesting closures, so key of R_3 is ABD)

- Final relations: R_1(<u>A,B</u>,C), R_3,(<u>A,B,D</u>), R_4(<u>D</u>, E)

What about foreign keys?

Previous example: R(A,B,C,D,E) FD: AB→C, D→E

- Final relations were R_1(<u>A,B</u>,C), R_3,(<u>A,B,D</u>), R_4(<u>D</u>, E). What are the foreign keys?
- Basically, any time you decompose on a FD X→b, the "**others**" relation **references** X in the (X U b) relation. Assuming all attributes are ints:

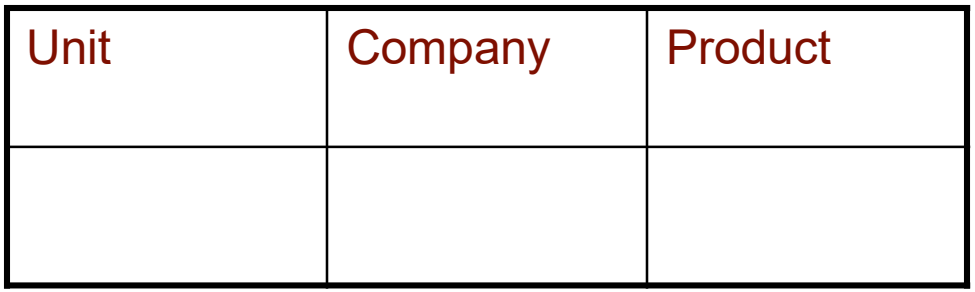
```
CREATE TABLE R3 (
    A       INT,
    B       INT,
    C       INT,
    PRIMARY KEY (A, B, D)
    FOREIGN KEY (A,B) REFERENCES R1(A,B),
    FOREIGN KEY (D) REFERENCES R4(D)
```

Note: sometimes things get hard to follow when you decompose multiple times. Just do what makes sense.

This BCNF stuff is great and easy!

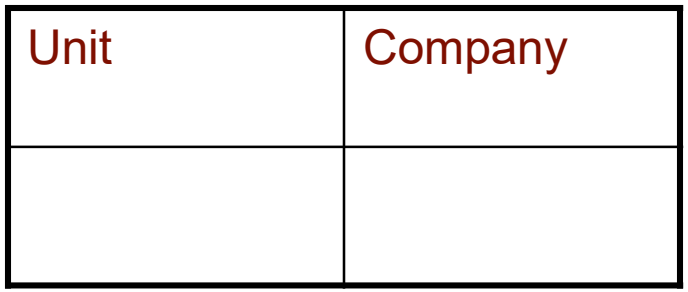
- Guaranteed that there will be no redundancy of data
- Easy to understand (just look for superkeys)
- Easy to do.
- So what is the main problem with BCNF?
  - For one thing, BCNF may not preserve all dependencies

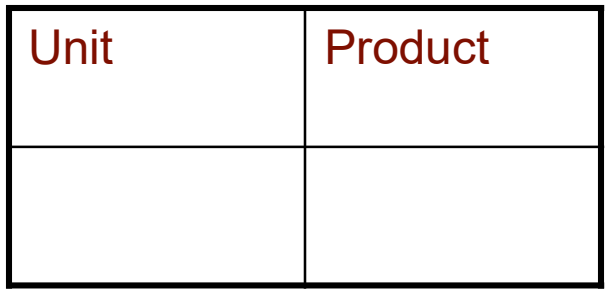
An illustrative BCNF example



Unit → Company
Company, Product → Unit

Is Unit a key?





Unit → Company

We lose the FD: Company, Product → Unit !!

So What's the Problem (1/2)?

Assume that we start with a table with a single tuple:

Unit	Company	Product
SKYWill	UBC	Databases

We decompose on Unit→Company:

Unit	Company
SKYWill	UBC

Unit	Product
SKYWill	Databases

Now someone adds a new team, Team Meat to each table:

Unit	Company
SKYWill	UBC
Team Meat	UBC

Unit	Product
SKYWill	Databases
Team Meat	Databases

So What's the Problem (2/2)?

Unit	Company
SKYWill	UBC
Team Meat	UBC

Unit	Product
SKYWill	Databases
Team Meat	Databases

Unit → Company

No problem so far. All *local* FD's are satisfied.
Let's put all the data back into a single table again:

Unit	Company	Product
SKYWill	UBC	Databases
Team Meat	UBC	Databases

Violates the FD: Company, Product → Unit

3NF to the rescue!

A relation R is in 3NF if:

If X → b is a non-trivial dependency in R,
then X is a superkey for R
or b is part of a (minimal) key.

} BCNF

(must be true for every such functional dependency)

Note: b must be part of a key *not* part of a superkey (if a key exists, *all* attributes are part of a superkey)

Example: R(Unit,Company, Product)

- Unit → Company
- Company, Product → Unit
  Keys: {Company, Product}, {Unit,Product}

So how do we decompose into 3NF?

- To do that, we need to have a way to make the set of functional dependencies as small as possible
- Otherwise, we may accidentally duplicate information (which defeats the whole point of this exercise)

To decompose to 3NF we rely on the Minimal Cover for a Set of FDs

Goal: Transform FDs to be as small as possible

- *Minimal cover* G for a set of FDs F:
  - Closure of F = closure of G (i.e., imply the same FDs)
  - Right hand side of each FD in G is a single attribute
  - If we delete an FD in G or delete attributes from an FD in G, the closure changes
- Intuitively, every FD in G is needed, and is "*as small as possible*" in order to get the same closure as F
- e.g., A→B, ABCD→E, EF→GH, ACDF→EG has the following minimal cover:
  - A→B, ACD→E, EF→G and EF→H

Finding minimal covers of FDs

1. Put FDs in standard form (have only one attribute on RHS)
2. Minimize LHS of each FD
3. Delete Redundant FDs

Example:

A→B, ABCD→E, EF→G, EF→H, ACDF → EG

- Replace last rule with
  - ACDF → E
  - ACDF → G

Finding minimal covers of FDs

1. Put FDs in standard form (have only one attribute on RHS)
2. Minimize LHS of each FD
3. Delete Redundant FDs

Example:

A→B, ABCD→E, EF→G, EF→H, ACDF → E, ACDF → G

- Can we take anything away from the LHS?
  - ACD^{+}= ABCDE, (crucially includes B) so remove B from the FD

Finding minimal covers of FDs

1. Put FDs in standard form (have only one attribute on RHS)
2. Minimize LHS of each FD
3. Delete Redundant FDs

Example:

A→B, ACD→E, EF→G, EF→H, ACDF → E, ACDF → G

- Let's find $ACDF^+$ **without** considering the highlighted FDs
  - $ACDF^+$= ACDFEBGH, so I can remove the highlighted rules

- Final answer: A→B, ACD→E, EF→G, EF→H

Make sure to minimize LHS before deleting redundant FDs

- If step 3 is done prior to step 2, the final set of FDs could still contain redundant FDs. The wrong way:
- Consider ABCD→E, E→D, A→B, AC→D
- Already has one attribute on RHS
- Let's delete redundant FDs.
  - None of the FDs are redundant.
- The right way:
- ABCD→E, E→D, A→B, AC→D
  - Now let's shorten FDs
  - ABCD→E can be replaced by AC→E
- However the current set of FDs are not minimal
  - AC→E, E→D, A→B, AC→D
  - The highlighted FD can be deleted

1. One attribute on RHS
2. Delete Redundant FDs
3. Minimize LHS of each FD

Does not work

Now we're ready to decompose into 3NF

- We'll cover two methods
- Both methods
  - Result in relations that do not violate 3NF
  - Are lossless (you don't get any additional tuples)
  - Preserve all functional dependencies
- The first one starts by ensuring that the decomposition is lossless and then preserves all functional dependencies
- The second one starts by preserving all functional dependencies and then ensures that the decomposition is lossless

There are other methods that do not ensure both of these properties. Do not use them!

Decomposition into 3NF method #1: lossless join method

Decomposition into 3NF using the lossless join method:

- Given the FDs F, compute F': the *minimal cover for F*
- Decompose using F' *if violating 3NF* similar to how it was done for BCNF – decompose *all the way down to BCNF*
- After each decomposition identify the set of dependencies N in F' that are not preserved by the decomposition.
- At the end, for each $X \rightarrow b$ in N create a relation $R_n(X \cup b)$ and add it to the decomposition

Intuition: first remove redundancy using lossless joins to ensure all results are valid. Then ensure that we maintain all functional dependencies.

3NF example (already a minimal cover)

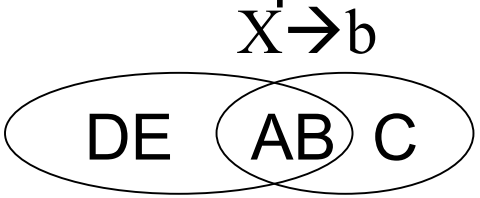
- Example: R(ABCDE) FD: AB→C, C→D

$ABE^+ = ABCDE$ only key

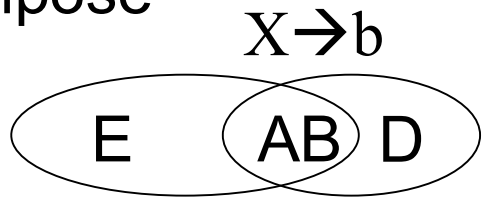
$AB^+ = ABCD$
$C^+ = CD$

AB → A
AB → B
AB → C
AB → D

- AB→C violates 3NF, so not in 3NF. Decompose to BCNF
  - $R_1(ABC)$, $R_2(ABDE)$



- AB→D violates BCNF, so decompose
  - $R_3(ABD)$, $R_4(ABE)$

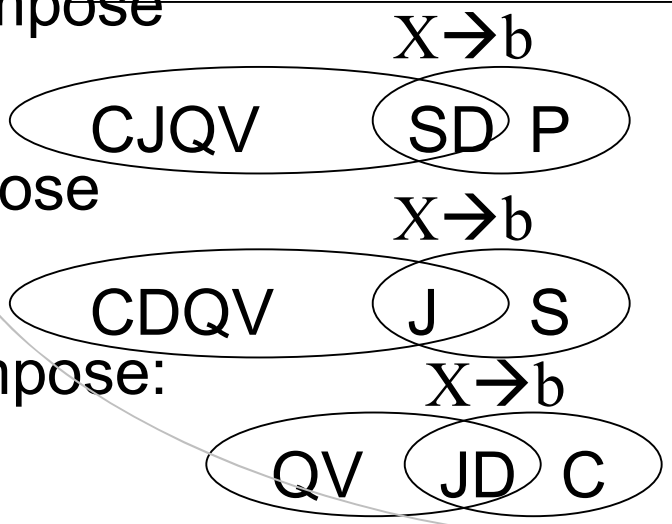


- $R_1(ABC)$, R3(ABD), $R_4(ABE)$ are now in BCNF
- Are all FDs preserved? We lost C→D so add $R_5(CD)$

Another 3NF example

- Let R(CSJDPQV) be a relation with the following FDs
  - SD→P
  - JP→C
  - J→S
  - SD→P violates 3NF in R, so decompose
  - R_1(SDP) , R_2(CSJDQV)
- J→S violates BCNF in R_2, so decompose
  - R_3(JS), R_4(CJDQV)
- JD→C violates BCNF in R_4, so decompose:
  - R_5(JDC), R_6(JDQV)
- No more violations! Are all FDs preserved? No. Add: R_7(JPC)
- Final answer: R_1(SDP), R_3(JS), R_5(JDC), R_6(JDQV), R_7(JPC)

$JP^+ = JPSC$
$SD^+ = SDP$
$J^+ = JS$
$JD^+ = JDSPC^*$ (next slide)
$JDQV^+ = CSJDPQV$ Key



On the previous slide, we took the closure of JD

- The given FDs were:
  - SD→P
  - JP→C
  - J→S
- How did we know to do that?
- Most of it is practice
- Looking at the explanation of what has to be in a key can help:

Left	Middle	Right
JDQV	SP	C

- Q&V don't appear in any FDs, so no interesting closures to take there
- Only thing left that's interesting to try: JD

Decomposition into 3NF Using a Minimal Cover and 3NF Synthesis

- This is an alternative approach to finding a lossless-join, dependency-preserving decomposition into 3NF.
- Consider relation R and its FDs F. To use synthesis to decompose to 3NF:
  1. Find a minimal cover F'.
  2. For each FD X → b in F'
     Add relation Xb to the decomposition for R.
  3. If there are no relations in the decomposition that contain all of the attributes of a key, add in a relation that contains all the attributes of a key. This preserves lossless joins.

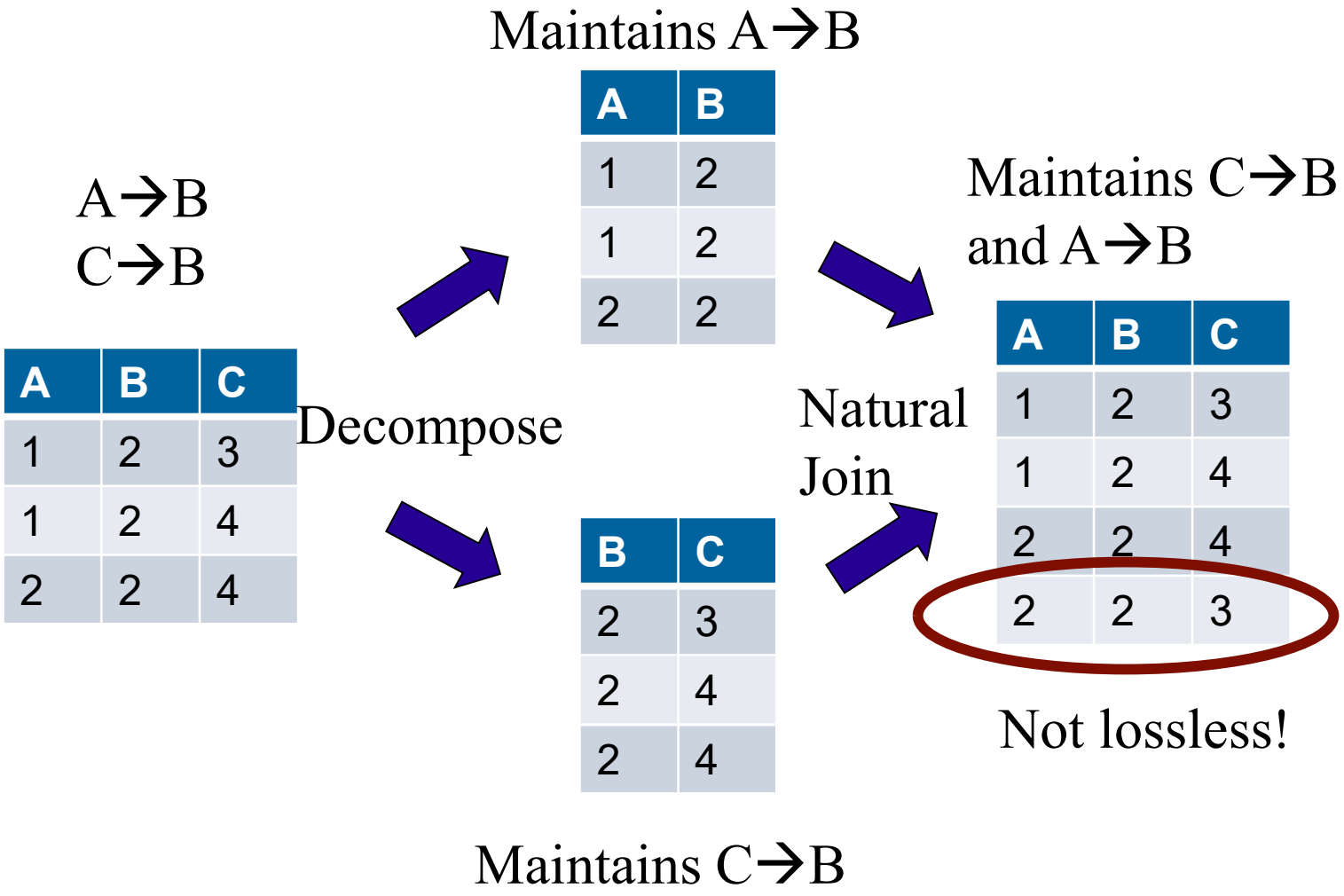
Intuition: first maintain all functional dependencies, then assure that joins are lossless by adding a key.

Example: Decomposition into 3NF Using a Minimal Cover and 3NF Synthesis

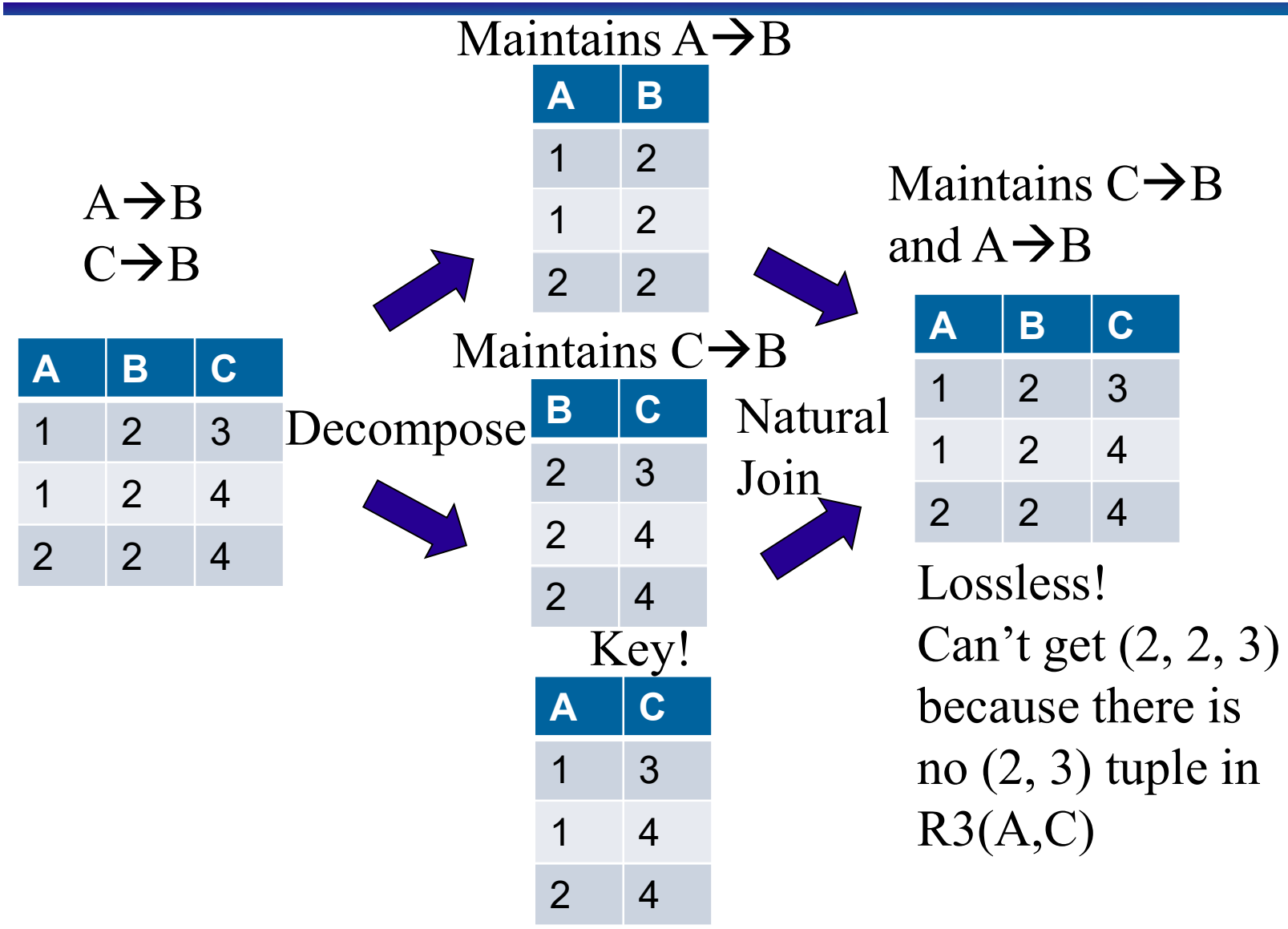
- Suppose we have R(A,B,C) with FDs: A → B, C → B. This is not in 3NF. The only key is AC.
  1. Find a minimal cover F'. Already done.
  2. For each FD X→b, add relation Xb to the decomposition for R.

     Result: R1(A,B) and R2(B,C). Are we done? No.
  3. Does it contain a key? What are the keys of R? AC. Add R3(A,C).
- Let's see why we need step #3

Consider the following set of tuples with the previous relation and FDs



Take two!


A→B
C→B
A B C
1 2 3
1 2 4
2 2 4
Decompose
Maintains A→B
A B
1 2
1 2
2 2
Maintains C→B
B C
2 3
2 4
2 4
Key!
A C
1 3
1 4
2 4
Natural Join
Maintains C→B and A→B
A B C
1 2 3
1 2 4
2 2 4
Lossless!
Can't get (2, 2, 3) because there is no (2, 3) tuple in R3(A,C)

In decompositions, you can often make some adjustments to make a "better" decomposition

- In this case, R1(ABC), R2(CD), **R3(EFG)**, **R4(EF)**, R5(ABEG) have redundant relations – you don't need R4 because all information is contained in R3
- You can make the same optimizations in decompositions for BCNF
- Other optimizations exist
- For the purposes of this course, we ask you to remove redundancy when decomposing to either BCNF or 3NF
- We define redundancy as: If all of the attributes of a relation R are a subset of the attributes of relation S, then R is redundant.
  - E.g., in this example EF is a subset of EFG, so R4 is redundant and should be removed.

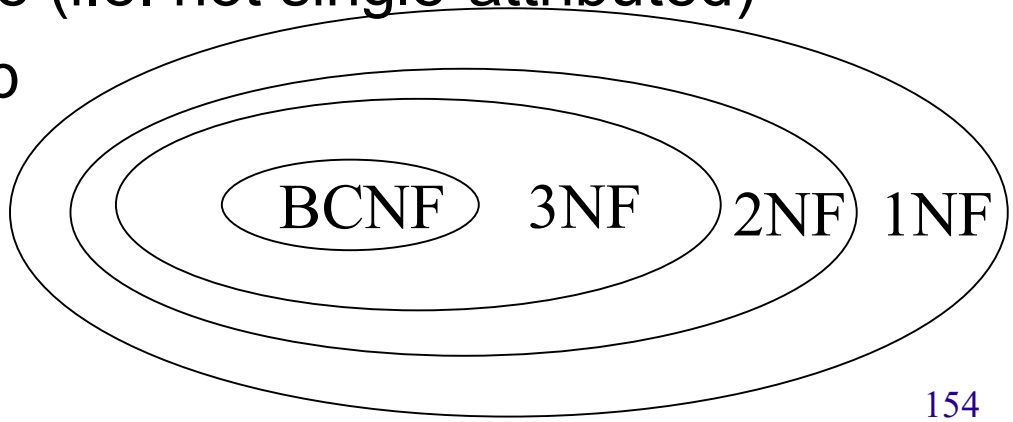
Comparing BCNF & 3NF

- BCNF guarantees removal of all anomalies
- 3NF has some anomalies, but preserves all dependencies
- If a relation R is in BCNF it is in 3NF.
- A 3NF relation R may not be in BCNF if all 3 of the following conditions are true:

  a. R has multiple keys

  b. Keys are composite (i.e. not single-attributed)

  c. These keys overlap



Normalization and Design

- Most organizations go to 3NF or better
- If a relation has only 2 attributes, it is automatically in 3NF and BCNF
- Our goal is to use lossless-join for all decompositions and preserve dependencies
- BCNF decomposition is always lossless, but may not preserve dependencies
- Good heuristic:
  - Try to ensure that all relations are in at least 3NF
  - Check for dependency preservation

On the other hand… Denormalization

- Process of intentionally violating a normal form to gain performance improvements
- Performance improvements:
  - Fewer joins
  - Reduces number of foreign keys
    - Since FD's are often indexed, the number of indexes many be reduced
- Useful if certain queries often require (joined) results, and the queries are frequent enough

Learning Goals Revisited

- Debate the pros and cons of redundancy in a database.
- Provide examples of update, insertion, and deletion anomalies.
- Given a set of tables and a set of functional dependencies over them, determine all the keys for the tables.
- Show that a table is/isn't in 3NF or BCNF.
- Prove/disprove that a given table decomposition is a lossless join decomposition. Justify why lossless join decompositions are preferred decompositions.
- Decompose a table into a set of tables that are in 3NF, or BCNF.