Unit 4 Schema Refinement and Normal Forms

Readings :

3rd edition: Chapter 19, sections 19.1-19.6 (except 19.5.2), or

2nd edition: Chapter 15 sections 15.1-15.7

In Databases so far …

- What's great about databases?
- How to create a *conceptual design* using ER diagrams
- How to create a *logical design* by turning the ER diagrams into a relational schema including "minimizing" the data and relations created
- Now showing …
  - Are we done (with the logical design)?
  - How to *refine* that schema to reduce duplication of information

Learning Goals

- Discuss 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.

Consider the following entity set for mailing addresses at UBC:

Meets all the criteria that we have for an entity
There is nothing wrong with this entity

What would an instance look like?

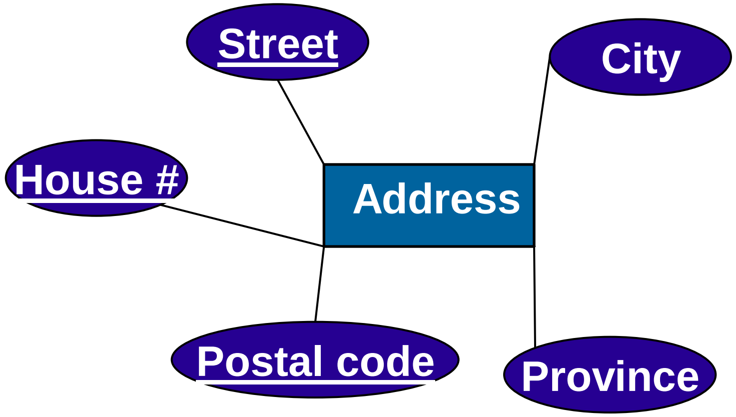
Name	Department	Mailing Location
Ed Knorr	Computer Science	201-2366 Main Mall
Raymond Ng	Computer Science	201-2366 Main Mall
Laks V.S. Lakshmanan	Computer Science	201-2366 Main Mall
Meghan Allan	Computer Science	201-2366 Main Mall
Joel Friedman	Computer Science	201-2366 Main Mall
Joel Friedman	Math	121-1984 Mathematics Rd
Brian Marcus	Math	121-1984 Mathematics Rd

Problems? 1. space. 2. typos 3. changes (e.g., departments move, or change names)

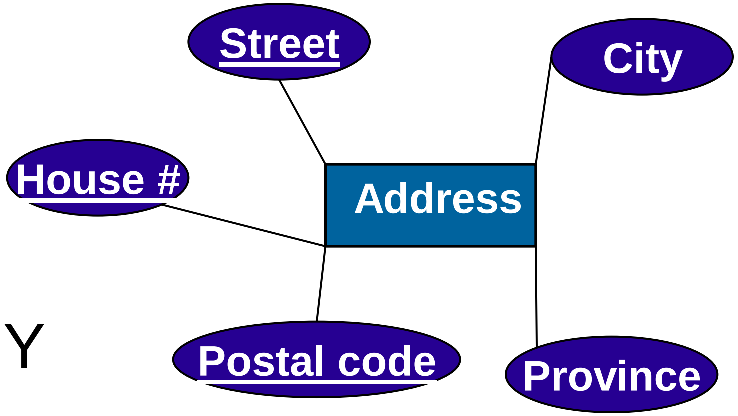
0,1

Okay, that's bad. But how do I *know* if a department has just one address?

- Databases allow you to say that one attribute determines another through a *functional dependency (FD)*.
- So if Department determines MailingLocation but not Name, we say that there's a functional dependency from Department to MailingLocation. But Department is NOT a key.
- Another example: Address(House#, Street, City, Province, PostalCode).
- PostalCode determines City, and Province, but is NOT a key either.



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

FDs made precise

- You've already seen a special case of FDs – Key Constraints.
- The FD Department → MailingLocation is supposed to hold for mailingAddress(Name, Department, MailingLocation).
- In Datalog notation, this means $mailingAddress(\_, D, A), mailingAddress(\_, D, A') \rightarrow A = A'$.

1.5

- The FD PostalCode → {City, Province} is supposed to hold for address(House#,Street,City,Province,PostalCode). In Datalog notation, $address(\_,\_,C,\_,PC), address(\_,\_,C',\_,PC) \rightarrow C = C'$. and
- $address(\_,\_,\_,P,PC), address(\_,\_,\_,P',PC) \rightarrow P = P'$.

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

Note: Key House#, Street, Postal Code
FD:
It looks like maybe City→Province, but there’s a Victoria in BC, Newfoundland, and Ontario & a Delta in Ontario:
Moral: can’t tell from instances

Which functional dependencies. again?

- A FD is a statement about *all* **allowable** instances.
  - Must be identified by application semantics and at design time.

Recall, r denotes instance and R denotes schema.

  - Given some instance *r1* of R, we can check if r1 violates some FD *f*, but we cannot tell if *f* holds over R!

Postal code → street? Department → mailingLocation?

- We'll concentrate on cases where there's a single attribute on the RHS: (e.g., PostalCode → Province)
- There are boring, *trivial* cases:
  - e.g. PostalCode, House# → PostalCode
- Our focus: the non-boring ones

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: Can we change Delta's province? Nunavut
- Insertion anomaly: What if we want to insert that V6T 1Z4 is in Vancouver? Can't do now without full address
- Deletion anomaly: If we delete all addresses with V6A 2S5, we lose that V6A 2S5 is in Vancouver!

Can we do better?

Once more try

- What if we tried…

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?

Okay, that worked pretty well.
Would be nice to understand why it worked!
Would be even better to understand when it would work.

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

Suppose a schema R(A,B,C,D) is not known to satisfy any FDs.
Can we split R in a lossless way?

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

Suppose a schema R(A,B,C,D) does satisfy some FDs.
Will any split of R be a lossless split?

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:
  - 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's the key!

- As a reminder, a key is a *minimal* set of attributes that uniquely identify tuples in 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
- 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 super key
    - House#, Street, PostalCode, Province is a superkey, but not a key

2

Deriving Additional FDs: the basics

William W. Armstrong. Canadian, eh?

- Given some FDs, we can often infer additional FDs:
  - *sid* → *city*, *city* → *acode* implies *sid* → *acode.*
- An FD *f* is *implied by* a set of FDs *F* if *f* holds whenever all FDs in *F* hold.
  - ***(Consequence) 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 → Y
    e.g., city,major→city
  - *Augmentation*: If X → Y, then X Z → Y Z for any Z
    e.g., if sid→city, then sid,major → city,major
  - *Transitivity*: If X → Y and Y → Z, then X → Z
    sid → *city*, *city* → *acode* implies *sid* → *acode*
- These are *sound* and *complete* inference rules for FDs.

Deriving Additional FDs

- 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
- Examples:
  - Derive union rule from axioms (Reflexivity, Augmentation, and Transitivity)
  - Drive Decomposition rule from Reflex and Trans.

Corollary: Given any set of FDs F, can convert F into an *equivalent* set of FDs F', s.t. every FD in F' is of the form X→A, where X is a set of attributes and A is a single attribute.

Why do we care? Greatly simplifies analysis!

Example: Supplier-Part DB

- Suppliers supply parts to projects.
  - supplier attributes: sname, city, status
  - part attributes: p#, pname
  - supplier-part attributes: qty:
    SupplierPart(sname,city,status,p#,pname,qty)
- Functional dependencies:
  - fd1: sname → city
  - fd2: city → status
  - fd3: p# → pname
  - fd4: sname, p# → qty

Supplier-Part Key: Part 1: Determining all attributes

Exercise: Show that (sname, p#) is a key of
SupplierPart(sname,city,status,p#,pname,qty)

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

Supplier-Part Key: Part 1: Determining all attributes

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

Exercise: Show that (sname, p#) is a key 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,pname	fd3, aug.
10.	sname,p# → sname, p#, status, city, qty, pname	8, 9, union

Supplier-Part Key: Part 2: Minimality

b. Show: (sname, p#) is a *minimal* superkey 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

Functional dependencies & keys

- 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, e.g.,

  R(House #, Street, City, Province, Postal Code)
- A super key 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 derive others. We'd covered how to do so using Armstrong's axioms
- **Theorem:** R satisfying FDs F, decomposed into R1 and R2. It is lossless join (LLJ) iff one of these FDs is implied by F:
  - R1 ∩ R2 → R1 OR
  - R1 ∩ R2 → R2.

Note the Key connection!

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

- Scared you're going to mess up? *Closure* for a set of attributes is a fool-proof method of checking FDs.
- Closure for 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:

  Set Closure = X

  Until Closure doesn't change do

  if $A_1, \ldots, A_n \rightarrow B$ is a FD **and** $\{A_1, \ldots, A_n\} \subseteq$ Closure

  then add B to Closure

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

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

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

$\{sname\}^+$ =

$\{p\#\}^+$ =

Here's a painless method

- Let R be a relation schema, i.e., R = a set of attributes.
- So to check if a set of attributes $X \subseteq R$ is a superkey, just check to see if its closure = all the attributes, i.e., check if $X^+ = R$ – this is pretty simple!
- Additionally, if you want to check if X is a key, just check that for every subset Y of X, $Y^+ \neq R$.
  - Do we need to do this check for *every* subset of X?
  - If a subset Y of X is not a superkey, does any subset of Y have a chance?
  - So, $\forall A \in X$, just check that $(X - \{A\})^+ \neq R$.
  - MUCH simpler!

4

Flash back – our original question was …

- Is this a good design?

Name	Department	Mailing Location
Ed Knorr	Computer Science	201-2366 Main Mall
Raymond Ng	Computer Science	201-2366 Main Mall
Laks V.S. Lakshmanan	Computer Science	201-2366 Main Mall
Meghan Allan	Computer Science	201-2366 Main Mall
Joel Friedman	Computer Science	201-2366 Main Mall
Joel Friedman	Math	121-1984 Mathematics Rd
Brian Marcus	Math	121-1984 Mathematics Rd

- Is there a rule that says if the amount of redundancy that we have is good?

Time we achieved some normalcy! ☺

- 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)***
  - ***Third normal form (3NF)***
  - ***Boyce-Codd Normal Form (BCNF)***
- If a relation is in a certain *normal form*, certain problems (aka anomalies!) are avoided/minimized.
- Normal forms can help decide whether decomposition (i.e., splitting tables) will help.

1NF

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

2NF

- No partial key dependency
- A relation is in 2NF, if for every FD X→A where X is a (not necessarily primary) key and A is a non-key attribute, then no proper subset of X determines A. Here, A is a non-key attribute.
- e.g., the relation address(house#,street,city,province,postal_code) relation is not in 2NF:
  - house#, street, postal_code is a key
  - postal_code → province ➔ 2NF-violating FD
  - Other examples of 2NF violation?

3NF

A relation R is in 3NF if:

If X → A is a non-trivial dependency in R, then X is a superkey for R **or A is part of a key.**

i.e., whenever X determines a non-key attr, X better be a super key.

Note: Being part of a super key doesn't count! Why? Super Key could contain "junk".

Example: address(Street, City, PostalCode), abbreviated to: address(S,C,P).

FDs: SC→P.

P→C.

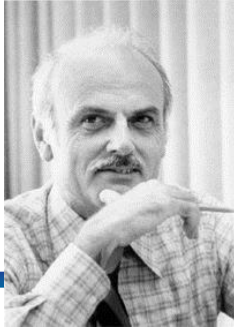
Keys: SC, SP.

Does it satisfy 3NF? What about 2NF?

We will return to 3NF a little later.

Raymond Boyce & Ted Codd

Boyce-Codd Normal Form (BCNF)



A relation R is in BCNF if:

If X → A is a non-trivial FD in R,

then X is a superkey for R

(Must be true for every such FD)

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

In English:

Only (super)keys should determine other attributes.

Ex: Address(<u>House#, Street, </u>City, Province, <u>PostalCode</u>)

FD: PostalCode → City

Is it in BCNF? Why (not)?

What do we want? Guaranteed freedom from redundancy!

How do we get there?

A relation may be in BCNF already!

Interesting fact: all two attribute relations are in BCNF!

Hint: What are the only possible non-trivial FDs in a 2-attribute relation schema?

If not, decomposition is the answer!

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)

How can we decompose without losing information?

How can we decompose a relation w/o losing information?

Address(House#,Street,City,Province,Postal Code).

Address(House#,Street#,PostalCode)

PC(City, Province, PostalCode)

Does the above decomposition lose information?
What does it mean to lose information?
How can we tell if we lose?
We need to know how the JOIN operation in Relational Algebra works, for this purpose.

A sneak preview: the join

- Definition: $R_1 \bowtie R_2$ is the 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
4	5	6
7	2	8
1	2	8
7	2	3

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
- In general, the other direction does not hold! If it does, the decomposition is a lossless-join.

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

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

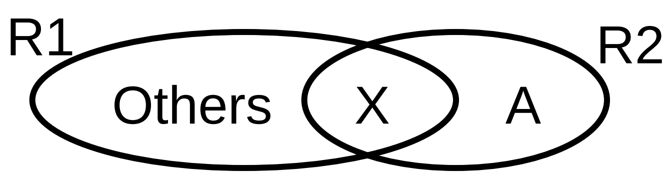
Note tuples 1 2 8 and 7 2 3 not in original.

So what did we lose?

How do we decompose into BCNF losslessly?

- Let r be a relation with attributes R, and F be a set of FDs on R s.t. all FDs have a single attribute on the RHS.
- Pick any f $\in$ FD of the form X→A that violates BCNF
- Decompose R into two relations: R_1(R-A) & R_2(XA)
- Recurse on R_1 and R_2 using FDs

Pictorially:



Note: answer may vary depending on order you choose. That’s okay -- All final answers guaranteed to be in BCNF.

BCNF Example

Recall def. of BCNF: For **all** non-trivial FDs X→A, X must be a superkey .

E.g.: Relation: R(ABCD)   FD: B→C, D→A

Keys?

A+ = A; B+ = BC; C+ = C; D+ = AD; BD+ = BDCA

BD is the only key

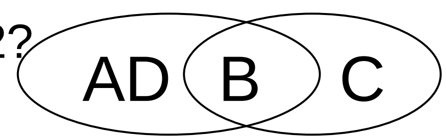
Process R(ABCD).

Look at FD B→ C. Is B a superkey?

No. Decompose R into R1(B,C), R2(A,B,D)

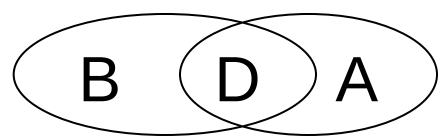
B→C is the only FD that applies to R1. R1 is in BCNF. Process R2(ABD).

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



No. Decompose R2 into

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



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

{R1, R3, R4} is a LLJ decomposition of R.

R1, R3, R4 are each in BCNF.

Another BCNF Example

- R(ABCDE)
- FD: AB→C, D→E.
- Generate the BCNF (lossless-join) decomposition of R. IOW, split up R into smaller relation schemas s.t. each of them is in BCNF and together they are LLJ.

After you decompose, how do you know which FDs apply?

- Take the closure of the attributes using *all* FDs
- For an FD X→A, if the decomposed relation S contains XA (i.e., X U {A}), then the FD holds for S:
- E.g., Consider relation R(A,B,C,D,E) with FDs $AB \rightarrow C$, $BC \rightarrow D$, $CD \rightarrow E$, and $DE \rightarrow A$. Is CD a super key? $(CD)^+ = CDEA \neq R$.
- Split R into R1(CDE) and R2(ABCD).
- Does CD→A hold for R2? Yes. Closure
- We need this knowledge in successfully completing a LLJ BCNF decomp. of R.

Yet Another BCNF Example:

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

FD =

$A \rightarrow B$

$DE \rightarrow F$,

$B \rightarrow C$

Is it in BCNF? If so, why. If not, decompose into BCNF

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 why are there more normal forms?
  - For one thing, BCNF may not “preserve all dependencies”…
    - What does that mean?

An illustrative BCNF example

Unit	Company	Product

Unit → Company
Company, Product → Unit

Company, Product
Unit, Product

Key(s)?

Unit	Company

Unit	Product

Unit → Company

BCNF:
A must be superkey

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

So What's the Problem?

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:

How could the dbms check if an update would violate the FD Company,Product →Unit?

3NF to the rescue!

Recall: A relation R is in 3NF if:

If X → A is a non-trivial dependency in R, then X is a superkey for R } BCNF

or A is part of a key.

(must be true for every such functional dependency)

Note: A 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)

FDs: Unit → Company BCNF, no. Company part of a key so 3nf

Company, Product → Unit Company, Product = superkey

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

Is it in BCNF? 3NF?

To decompose into 3NF we rely on the *minimal cover*

Minimal Cover for a Set of FDs

Goal: Transform FDs to be as compact as possible

- *Minimal cover* G for a set of FDs F:
  - Closure of F = closure of G (i.e., imply the same FDs)
  - RHS 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 slim 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

We'll see how to derive this on the next slide

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→GH, ACDF → EG

1. Need ACDF→E, ACDF→G ?
2. ABCD→E goes to ACD→E (closure)
3. Redundant: ACDF→E, ACDF→G
(take closure of ACDF w/o rule ACDF→E)
In the end: A→B, ACD→E, EF→G, EF→H

Another minimal cover example

- Consider the relation R(CSJDPQV) with FDs
  $C \rightarrow SJDPQV$, $JP \rightarrow C$, $SD \rightarrow P$, $J \rightarrow S$

Find a minimal cover

Decomposition into 3NF using Minimal Cover

- Decomposition into 3NF:

  1. Given the FDs F, compute F': the *minimal cover for F*

  2. Obtain a BCNF decomposition of R, say $R_1, \ldots, R_k$.

  3. Clearly, this is also an LLJ 3NF decomp. of R but may not preserve some FDs. So:

     3a. Project F onto each $R_i \Rightarrow F_i$.

     3b. $F \setminus (\cup_{1 \le i \le k} F_i)^+$ is the set of FDs that are not preserved.

     3c. ∀such FD $X \rightarrow A$, add a scheme XA to the decomp. above. All FDs are obviously preserved now.

     Need an efficient algorithm for step 3b.

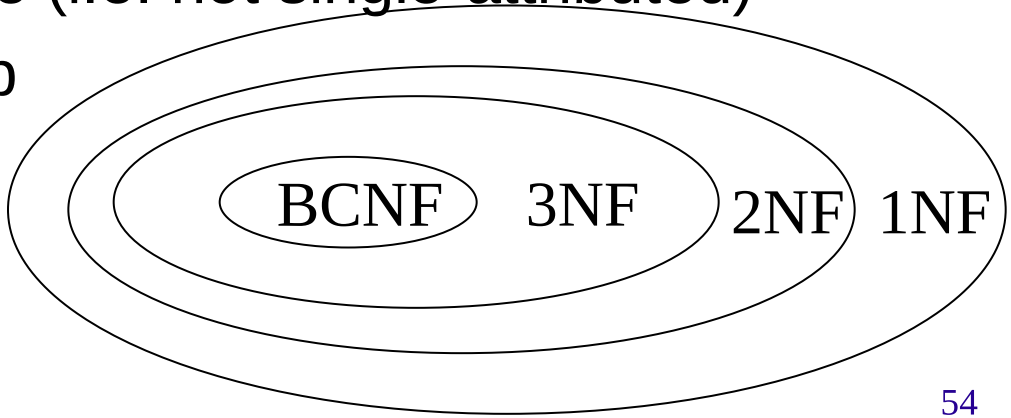
Example: R(ABCDE) FD: AB→C, C→D

Synthesis of 3NF from scratch

- Conceptually simpler.
- Given a set of FDs F, obtain a minimal cover F'.
- $\forall$FD $X \rightarrow A \in F'$, create a scheme XA.
- Resulting decomp. is guaranteed to preserve all FDs (trivially) and each scheme is in 3NF. But no guarantee for LLJ!
- Easy fix: add any scheme that contains a key of the original relation scheme R.
- Revisit previous example: R(ABCDE) FD: AB→C, C→D.

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



On the one hand…
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 FDs are often indexed, the number of indexes may 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.
- Justify why lossless join decompositions are preferred decompositions.
- Decompose a table into a set of tables that are in 3NF, or BCNF.
- Additionally …

- Given a set of FDs, find all keys of a relation scheme and prove that we have found them all.
- Find the minimal cover for a set of FDs.
- Test if a decomp. Is LLJ.
- Test if a decomp. is dependency preserving, i.e., preserves all FDs.