Database Design & Normalization: Part 2

COM 3563: Database Implementation

Avraham Leff

Yeshiva University

avraham.leff@yu.edu

COM3563: Fall 2020

Today's Lecture: Overview

1. Warmup
2. Functional Dependencies, Implications & Closures
3. Boyce-Codd Normal Form & BCNF Decompositions
4. Lossless Joins & Preserving FD Information
5. Algorithms Related to Functional Dependency
6. Bonus Material

Some figures on functional dependency material taken from lectures created by John Greiner, Rice University:

used with permission

Warmup

Introduction

- We're in the middle of the topic of relational database design
- Previous E-R & database design discussions involved topics such as
  - How to choose attributes & types appropriately
  - How to pick entity and relationship sets appropriately
- We want to know: can we get an objective evaluation of our design?
- Last lecture introduced the concept of database normalization in an effort to answer this question in the affirmative

Today's Lecture

Review:

- Database normalization is a process in which we place constraints on table design to limit redundancy and other anomalies
- An initial set of tables are decomposed to eliminate various classes of data anomalies
- There is a hierarchy of normal forms that guarantee different properties about the database, involving tradeoffs with respect to the amount of required effort
- Last lecture took an ad-hoc approach to this topic
- Today's lecture: another, more rigorous and algorithmic, look at the topic

ID	*name*	*salary*	*dept_name*	*building*	*budget*
22222	Einstein	95000	Physics	Watson	70000
12121	Wu	90000	Finance	Painter	120000

- Some well-meaning person joins the *instructor* and *department* relations (from the textbook) to improve performance
- Intuitively, this *instructor_department* relation will incorporate lots of redundancy
  - Example: every tuple that includes an instructor must also include information about her department's budget
  - And there are many instructors in a given department!
- This intuition can be made rigorous by writing a business rule
  - *"if there were a schema (dept_name, building, budget), then dept_name would be a candidate key"*
- Alternatively: write a functional dependency (FD)
  - $dept\_name \rightarrow building, budget$

How Do FDs Help?

- The *dept_name* $\rightarrow$ *building*, *budget* FD enables us to algorithmically detect that *instructor_department* has a problem
- *dept_name* is not a candidate key for instructor_department
  - Because departments are associated with multiple instructors
- Therefore: we know that we have to decompose *instructor_department* into a set of tables that will allow *dept_name* to be a candidate key
  - We could call those tables *instructor* and *department* ☺

Note:

- Initial set of FDs cannot be derived simply by "examining all instances $\in R$"
- Such FDs must come from requirements analysis

Functional Dependencies, Implications & Closures

> *"Undergraduates frequently find this chapter difficult. It is acceptable to cover only Sections 8.1 and 8.3 for classes that find the material particularly difficult."*
>
> *(Source: previous edition of textbook)*

I am confident that you can do this ☺
That said: we'll only be covering (and you're only responsible for) a subset of the more advanced material

- Functional-dependency theory is the formal theory that tells us *"which functional dependencies are implied logically by a given set of functional dependencies?"*
- Enables us to develop algorithms to generate lossless decompositions into BCNF and 3NF
- Enables us to develop algorithms to test if a decomposition preserves dependencies

Generalize Notion of a Key: I

Review:

- A attribute subset K of relation R is a superkey of R if, for any legal instance of R ...
  - For all pairs of tuples t_1 and t_2
  - $t_1 \neq t_2 \implies t_1[K] \neq t_2[K]$
- In English: "no two tuples in any legal instance of R can have the same value on an attribute set K"
- Note: Superkeys uniquely identify entire tuples
- K is a candidate key for R *iff*
  - $K \to R$
  - There is no $\alpha \subset K$ such that $\alpha \to R$
- In other words, a candidate key K is *"a superkey with the additional property that removal of any attribute from K will cause K to not be a superkey"*

FDs Generalize Notion of a Key: II

FDs generalize the superkey idea:

- FDs express constraints that uniquely identify values of certain attributes
- Consider two sets of attributes $\alpha \subseteq R$ and $\beta \subseteq R$
- Then: an instance r(R) satisfies the functional dependency $\alpha \rightarrow \beta$ *iff*
  - For all pairs of tuples t_1 and t_2 in the instance
  - $t_1[\alpha] = t_2[\alpha] \implies t_1[\beta] = t_2[\beta]$
- We say that the functional dependency $\alpha \rightarrow \beta$ holds on schema r(R) if every legal instance of r(R) satisfies the functional dependency
- (Note: K is a superkey of R if the FD $K \rightarrow R$ holds on r(R))
- (We term an FD $\alpha \rightarrow \beta$ "trivial" if $\beta \subseteq \alpha$)

FD example

emp_id	name	phone	position
E0045	Smith	1234	Clerk
E3542	Mike	9876	Salesrep
E1111	Smith	9876	Salesrep
E9999	Mary	1234	Lawyer

{position} → {phone}

emp_id	name	phone	position
E0045	Smith	1234	Clerk
E3542	Mike	9876	Salesrep
E1111	Smith	9876	Salesrep
E9999	Mary	1234	Lawyer

Not: {phone} → {position}

Logical Implications Of FDs

name	color	category	dept	price
Gizmo	Green	Gadget	Toys	49
Widget	Black	Gadget	Toys	59
Gizmo	Green	Whatsit	Garden	99

Given:
{name} → {color}
{category} → {dept}
{color, category} → {price}

Logically implies:
{name, category} → {price}

- Given a set of FDs, we can often infer the existence of additional FDs and add them to the set
- This Figure shows an example of such a chain of inferences
- We'll examine a set of "canned inference rules" in the next several slides

Armstrong's Axioms

- Let F be a set of FDs
- We define the closure of F, F^+ as the set of all FDs logically implied by F
- (Note: $\alpha\gamma$ is standard notation for *"the union of the α and γ FDs"*

> Armstrong's axioms (1974) are a set of inference rules used to infer all the functional dependencies on a relational database. By repeatedly apply the axioms, we can find F^+

- Reflexivity: $\beta \subseteq \alpha \implies \alpha \to \beta$
- Augmentation: If $\alpha \to \beta$ then $\gamma\alpha \to \gamma\beta$
- Transitivity: If $\alpha \to \beta$ and $\beta \to \gamma$ then $\alpha \to \gamma$
- These rules are sound
  - Generate only correct FDs
- And complete
  - Generate all FDs

Additional Rules That Can Be Inferred From Armstrong's Axioms

- Union: $\alpha \to \beta$ and $\alpha \to \gamma \implies \alpha \to \beta\gamma$
- Decomposition: $\alpha \to \beta\gamma \implies \alpha \to \beta$ and $\alpha \to \gamma$
- Pseudotransitivity: $\alpha \to \beta$ and $\gamma\beta \to \delta \implies \alpha\gamma \to \delta$

We can use these rules to compute F^+ with an $O(2^{2n})$ algorithm. The idea to apply the rules successively to each of the 2^n (subsets of R) FDs on either side of an FD relationship.
See Textbook, Figure 7.7 (and Figure 7.8 for a quadratic algorithm)

Examples of Functional Dependency Closures

Given R = (A, B, C, G, H, I)
and given $F = \{ A \rightarrow B$
$A \rightarrow C$
$CG \rightarrow H$
$CG \rightarrow I$
$B \rightarrow H\}$

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 $CG \rightarrow I$
  - Alternatively: apply pseudotransitivity
- $CG \rightarrow HI$ by augmenting $CG \rightarrow I$ to infer $CG \rightarrow CG$ and augmenting $CG \rightarrow H$ to infer $CGI \rightarrow HI$, and then apply transitivity
  - Alternatively: Apply union

Closure of Attribute Sets (I)

- Let F be a set of FDs, and let α be a set of attributes
- We've previously defined the closure of F (F^+) as the "*set of all FDs logically implied by F*"
- We can similarly define the closure of α (or α^+) as "the set of all attributes β such that $\alpha \to \beta$"
- Key point: All "$\alpha \to \beta$" rules are specified by an FD
  - In other words, α^+ is the "set of attributes that are functionally determined by α under F"

The *closure* of A (A^+) is the set of all attributes B such that $A \rightarrow B$.

Closure of attribute sets

Given:

{name} $\rightarrow$ {color}
{category} $\rightarrow$ {dept}
{color, category} $\rightarrow$ {price}

Some closures:

$\{name\}^+$ = {name, color}
$\{name, category\}^+$ = {name, category, color, dept, price}
$\{color\}^+$ = {color}

Closure of Attribute Sets (II)

- Conceptually trivial to calculate the attribute closure of α:
  1. Compute F^+
  2. Determine the set of all FDs with α on the LHS
  3. Take the union of the RHS of these FDs
- But . . . as noted before: the F^+ algorithm is expensive ☹

Algorithm That Computes Closure of Attribute Sets: I

Note: this algorithm depends on F, not on F^+

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

- This algorithm is correct because attributes are added only when a FD implies (through reflexivity and transitivity) that $\alpha \rightarrow \gamma$ is part of the closure
- The union rule implies that $\alpha \rightarrow result \cup \gamma$
- Any attributes added to *result* are thus part of α^+

Algorithm That Computes Closure of Attribute Sets: II

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

- This algorithm computes α^+ because
  - If there is an attribute in α^+ that is not yet in *result* at any point in the algorithm
  - Then there must be an FD $\beta \rightarrow \gamma$ for which $\beta \subseteq$ *result* and at least one attribute γ which is not yet in *result*
- When the algorithm terminates (nothing can be added to *result*), all such FDs have been processed
  - All attributes "γ" have been added to *result*
  - All attributes α^+ are in *result*
- This algorithm is quadratic in the size of *F*

Example of Attribute Set Closure

Given R = (A, B, C, G, H, I)
and given $F = \{ A \rightarrow B$
$A \rightarrow C$
$CG \rightarrow H$
$CG \rightarrow I$
$B \rightarrow H\}$

Calculate (AG)+

1. result = AG
2. result = $ABCG$ ($A \rightarrow C$ and $A \rightarrow B$)
   - More explicitly: $A \rightarrow B$, $A \subseteq result = AG$, so $result = result \cup B$
3. result = $ABCGH$ ($CG \rightarrow H$ and $CG \subseteq AGBC$)
4. result = $ABCGHI$ ($CG \rightarrow I$ and $CG \subseteq AGBCH$)

Benefits of Attribute Closure Algorithm: I

Test superkey property

To test whether α is a superkey: compute α^+ and check whether α^+ contains all the attributes in R

- In previous example, we showed that $AG \rightarrow ABCGHI$
- Then to answer whether $AG \rightarrow R$, we simply check whether $R \supseteq (AG)^+$
- To answer whether any subset of AG is a a superkey, we check whether
  - $R \supseteq (A)^+$
  - $R \supseteq (G)^+$
- More generally: check for each subset of the superkey of size $n - 1$

Benefits of Attribute Closure Algorithm: II

Test whether an $\alpha \rightarrow \beta$ FD is in F^+

1. Compute α^+ using attribute closure algorithm
2. Check whether α^+ contains β

Benefits of Attribute Closure Algorithm: III

Better algorithm for computing F^+

1. For each $\gamma \subseteq R$, compute γ^+
2. Each $S \subseteq \gamma^+$, gives us a FD $\gamma \rightarrow S$

Previous algorithm was exponential in n: this is quadratic in n

- Instead of pseudo-code stating:
  - $\forall$ "provable" $\alpha \rightarrow \beta$
  - Add $\alpha \rightarrow \beta$ to F^+
- This algorithms states
  - $\forall \beta \subseteq \alpha^+$
  - Add $\alpha \rightarrow \beta$ to F^+

Functional Dependency Set Closure: Worked Example (I)

Given:

$\{a, b\} \rightarrow \{c\}$
$\{a, d\} \rightarrow \{b\}$
$\{b\} \rightarrow \{d\}$

Compute attribute set closures:

$\{a\}^+ = \{a\}$
$\{b\}^+ = \{b, d\}$
$\{c\}^+ = \{c\}$
$\{d\}^+ = \{d\}$
$\{a, b\}^+ = \{a, b, c, d\}$
$\{a, c\}^+ = \{a, c\}$
$\{a, d\}^+ = \{a, b, c, d\}$
$\{b, c\}^+ = \{b, c, d\}$
$\{b, d\}^+ = \{b, d\}$
$\{c, d\}^+ = \{c, d\}$
$\{a, b, c\}^+ = \{a, b, c, d\}$
$\{a, b, d\}^+ = \{a, b, c, d\}$
$\{a, c, d\}^+ = \{a, b, c, d\}$
$\{b, c, d\}^+ = \{b, c, d\}$
$\{a, b, c, d\}^+ = \{a, b, c, d\}$

Compute FDs:

$\{a\} \rightarrow \{a\}$
$\{b\} \rightarrow \{b\}, \ldots \rightarrow \{d\}, \ldots \rightarrow \{b, d\}$
$\{c\} \rightarrow \{c\}$
$\{d\} \rightarrow \{d\}$
$\{a, b\} \rightarrow \{a, b, c, d\}$
$\{a, c\} \rightarrow \{a, c\}$
$\{a, d\} \rightarrow \{a, b, c, d\}$
$\{b, c\} \rightarrow \{b, c, d\}$
$\{b, d\} \rightarrow \{b, d\}$
$\{c, d\} \rightarrow \{c, d\}$
$\{a, b, c\} \rightarrow \{a, b, c, d\}$
$\{a, b, d\} \rightarrow \{a, b, c, d\}$
$\{a, c, d\} \rightarrow \{a, b, c, d\}$
$\{b, c, d\} \rightarrow \{b, c, d\}$
$\{a, b, c, d\} \rightarrow \{a, b, c, d\}$

If we take $A \rightarrow B$ to be shorthand for $A \rightarrow B'$, where $B' \subseteq B$.

Functional Dependency Set Closure: Worked Example (II)

Given:	…	**Compute FDs:**	**Shorthand version:**
$\{a, b\} \rightarrow \{c\}$		$\{a\} \rightarrow \{a\}$	
$\{a, d\} \rightarrow \{b\}$		$\{b\} \rightarrow \{b\}$, … $\rightarrow \{d\}$, … $\rightarrow \{b, d\}$	$\{b\} \rightarrow \{d\}$
$\{b\} \rightarrow \{d\}$		$\{c\} \rightarrow \{c\}$	
		$\{d\} \rightarrow \{d\}$	
		$\{a, b\} \rightarrow \{a, b, c, d\}$	$\{a, b\} \rightarrow \{c, d\}$
		$\{a, c\} \rightarrow \{a, c\}$	
		$\{a, d\} \rightarrow \{a, b, c, d\}$	$\{a, d\} \rightarrow \{b, c\}$
		$\{b, c\} \rightarrow \{b, c, d\}$	$\{b, c\} \rightarrow \{d\}$
		$\{b, d\} \rightarrow \{b, d\}$	
		$\{c, d\} \rightarrow \{c, d\}$	
		$\{a, b, c\} \rightarrow \{a, b, c, d\}$	$\{a, b, c\} \rightarrow \{d\}$
		$\{a, b, d\} \rightarrow \{a, b, c, d\}$	$\{a, b, d\} \rightarrow \{c\}$
		$\{a, c, d\} \rightarrow \{a, b, c, d\}$	$\{a, c, d\} \rightarrow \{b\}$
		$\{b, c, d\} \rightarrow \{b, c, d\}$	
		$\{a, b, c, d\} \rightarrow \{a, b, c, d\}$	

Eliminating *trivial* FDs $A \rightarrow B$, where $B \subseteq A$.

Replacing FDs $A \rightarrow AB$ with $A \rightarrow B$.

Boyce-Codd Normal Form & BCNF Decompositions

Straightforward algorithm ☺

- While there are "bad" FDs ...
  - Decompose a table with "bad" FDs into sub-tables
- Q: OK, but what is a "bad" FD?
- A: let's look at the informal definition of BCNF (from Chris Date)

  BCNF: "Each attribute must represent a fact about the key, the whole key, and nothing but the key"
- $\alpha \to R$: this is a "good" FD, because α is a (super)key
- $\alpha \to \beta, \beta \subset R$: this is a "bad" FD because α is not a (super)key and yet functionally determines some of the attributes
  - Intuitively: this FD causes redundancy

BCNF Versus 3NF

- 3NF:

  Each non-key attribute "must provide a fact about the key, the whole key, and nothing but the key"

- BCNF is an enhancement of 3NF

  Each attribute ...

  - The BCNF requirement is concerned with every attribute in the relation, in contrast to only being concerned with non-key attributes
- The difference between the two normalization forms applies only when
  - A relation has multiple compound candidate keys
  - And: an attribute within one candidate key has a dependency on a part of another candidate key

Boyce-Codd Normal Form

- BCNF is "good" because it eliminates all redundancies from a schema
  - More precisely: redundancies that can be inferred from functional dependencies
- A relation schema R is in BCNF with respect to a set F of FDs if for all functional dependencies in F^+ of the form
  - $\alpha \to \beta$ where both α and $\beta \subseteq R$
- Either of the following are true
  - $\alpha \to \beta$ is a trivial dependency
  - α is a superkey for schema R
- Alternatively: if you can find an FD in which α is not a superkey ...
  - Then R is not in BCNF

Decomposing a Schema into BCNF

- The definition of BCNF suggests an algorithm for converting a non-BCNF schema into a new schema that is BCNF
- We have at least one non-trivial $\alpha \rightarrow \beta$ FD such that α is not a superkey
- Replace R with two schemas
  1. $\alpha \cup \beta$
  2. $R - (\beta - \alpha)$

ID	*name*	*salary*	*dept_name*	*building*	*budget*
22222	Einstein	95000	Physics	Watson	70000
12121	Wu	90000	Finance	Painter	120000

- α = *dept_name*, $\beta = \{building, budget\}$
- Replace *instructor_department* with
  1. $\alpha \cup \beta$ = (*dept_name*, *building*, *budget*)
  2. $R - (\beta - \alpha)$ = (*ID*, *name*, *dept_name*, *salary*)

Find a "Bad" Functional Dependency

name	ssn	phone	city
Mary	123-45-6789	713-555-1234	Houston
Mary	123-45-6789	713-555-6543	Houston
Joe	987-65-4321	512-555-2121	Austin
Joe	987-65-4321	512-555-1234	Austin

$$\{ssn\} \rightarrow \{name, city\}$$

- But: *ssn* doesn't determine *phone*
- So: *ssn* isn't a superkey

How to fix?

Fix the “Bad” Functional Dependency

Convert to BCNF using decomposition

name	ssn	city
Mary	123-45-6789	Houston
Joe	987-65-4321	Austin

Now a “good” FD.

{ssn} → {name, city}

ssn	phone
123-45-6789	713-555-1234
123-45-6789	713-555-6543
987-65-4321	512-555-2121
987-65-4321	512-555-1234

BCNF Decomposition: Doing This Systematically

- We've talked in an *ad-hoc* way about how to decompose a relational schema to satisfy BCNF
- We have the concepts to devise a *general-purpose* BCNF decomposition algorithm

Some version of this material is presented in the Textbook (Chapter 7.5). I am substituting a version that (in my opinion) is easier to understand. (As always, feedback welcome)

BCNF Decomposition Algorithm

BCNF_decomp(R) =

Find attribute set X s.t. $X^+ \neq X$ (not trivial) and $X^+ \neq R$ (not superkey).

If no such X, then return R.

Let D = X^+ - X. (attributes functionally determined by X)

Let N = R - X^+. (attributes not functionally determined by X)

Decompose R into $R_1(X \cup D)$ and $R_2(X \cup N)$.

Return BCNF_decomp(R_1), BCNF_decomp(R_2).

1. If no "bad" FDs found, return *R*: it's already in BCNF!
2. Else: split *R* into R_1 containing *X* plus attributes that *X* determines (*D*) and R_2 containing *X* plus attributes it does not determine (*N*)
3. Proceed recursively until no more "bad" FDs ...

BCNF Decomposition Algorithm: Example (I)

BCNF_decomp(R) =

Find attribute set X s.t. $X^+ \neq X$ and $X^+ \neq R$.

If no such X, then return R.

Let $D = X^+ - X$.

Let $N = R - X^+$.

Decompose R into $R_1(X \cup D)$ and $R_2(X \cup N)$.

Return BCNF_decomp(R_1), BCNF_decomp(R_2).

R(a,b,c,d,e)
$\{a\} \rightarrow \{b, c\}$
$\{c\} \rightarrow \{d\}$

$X = \{a\}$
$X^+ = \{a,b,c,d\}$

$D = \{b,c,d\}$
$N = \{e\}$

$R_1(a,b,c,d)$
$R_2(a,e)$

BCNF Decomposition Algorithm: Example (II)

BCNF_decomp(R) =

Find attribute set X s.t. $X^+ \neq X$ and $X^+ \neq R$.

If no such X, then return R.

Let $D = X^+ - X$.

Let $N = R - X^+$.

Decompose R into $R_1(X \cup D)$ and $R_2(X \cup N)$.

Return BCNF_decomp(R_1), BCNF_decomp(R_2).

$R_1(a,b,c,d)$
$\{a\} \rightarrow \{b, c\}$
$\{c\} \rightarrow \{d\}$

$X = \{c\}$
$X^+ = \{c,d\}$

$D = \{d\}$
$N = \{a,b\}$

$R_{11}(c,d)$
$R_{12}(a,b,c)$

BCNF Decomposition Algorithm: Example (III)

BCNF_decomp(R) =

Find attribute set X s.t. $X^+ \neq X$ and $X^+ \neq R$.

If no such X, then return R.

Let D = X^+ - X.

Let N = R - X^+.

Decompose R into $R_1(X \cup D)$ and $R_2(X \cup N)$.

Return BCNF_decomp(R_1), BCNF_decomp(R_2).

R_{11}(c,d)
{a} → {b, c}
{c} → {d}

No such X.

BCNF Decomposition Algorithm: Example (IV)

BCNF_decomp(R) =

Find attribute set X s.t. $X^+ \neq X$ and $X^+ \neq R$.

If no such X, then return R.

Let D = X^+ - X.

Let N = R - X^+.

Decompose R into $R_1(X \cup D)$ and $R_2(X \cup N)$.

Return BCNF_decomp(R_1), BCNF_decomp(R_2).

R_{12}(a,b,c)
{a} → {b, c}
{c} → {d}

No such X.

BCNF Decomposition Algorithm: Example (V)

BCNF_decomp(R) =

Find attribute set X s.t. $X^+ \neq X$ and $X^+ \neq R$.

If no such X, then return R.

Let D = X^+ - X.

Let N = R - X^+.

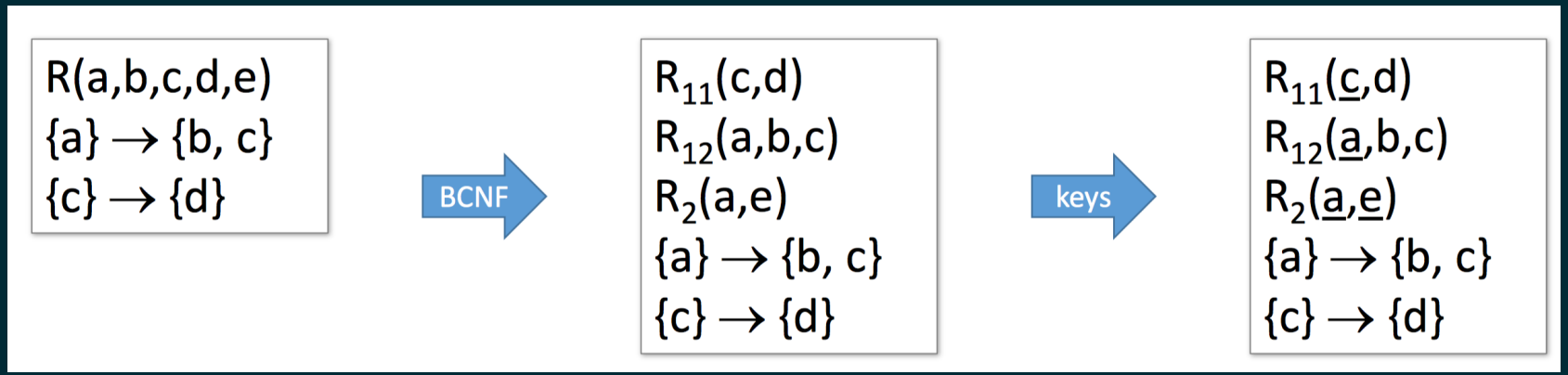
Decompose R into $R_1(X \cup D)$ and $R_2(X \cup N)$.

Return BCNF_decomp(R_1), BCNF_decomp(R_2).

R_2(a,e)
{a} → {b, c}
{c} → {d}

No such X.

BCNF Decomposition & Keys

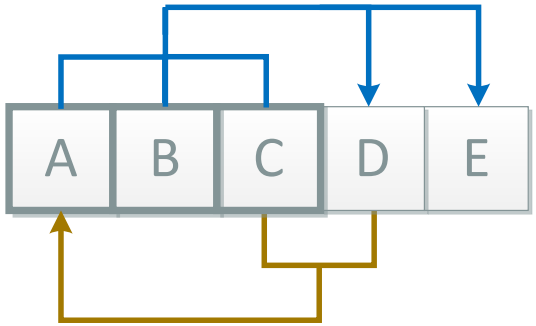


Lossless Joins & Preserving FD Information

Which FDs Are Allowed For Some Normal Forms
Consider X → A (*X* a set, *A* a single attribute)

BCNF	3NF	2NF
X → A is trivial (A is inside X)	X → A is trivial (A is inside X)	X → A is trivial (A is inside X)
X contains a key	X contains a key	X contains a key
	A is inside some key (informally: X → A into a key, but X can overlap a key)	A is inside some key (informally: X → A into a key, but X can overlap a key
		X not a proper subset of any/some key

Example: Relation in 3NF And Not in BCNF



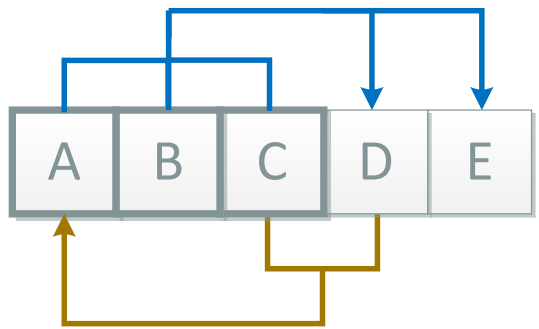
- Given functional dependencies: ABC → DE and CD → A
- ABC is a key and designated as primary
- This relation is not in BCNF as we have CD → A and CD does not contain a key as is easily seen
- But CD → A is of the form: (something not containing a key) → (attribute in a key) and this is permitted by 3NF
- Note there is another key that could have been the primary key: BCD
- Originally people were confused as they considered only one key and did not realize that in general 3NF ≠ BCNF

If Only One Key Then 3NF ⇒ BCNF

- Proof by contradiction (using example, but really general)
- Assume that a relation is in 3NF but not in BCNF and there is only one key
- Then we have a functional dependency that is permitted by 3NF but not permitted by BCNF, that is of the form

  (something not containing a key) → (attribute in a key)
- Example

  ABC is a key
  and CD → A holds

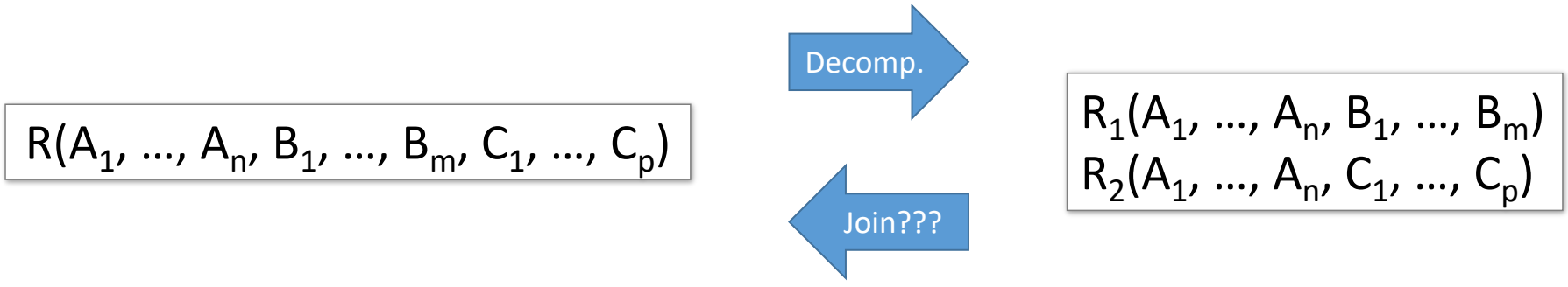


- Then we see that BCD is a key also, so we have more than one key
- So we proved: ***if*** 3NF and only one key ***then*** BCNF

BCNF Versus 3NF

- Q: Tempting to go for the best normalization: why bother with 3NF?
- A: Because a BCNF normalization can result in a design that is not dependency preserving
  - Meaning: after we've decomposed into smaller tables, we've lost one of the original FDs
  - Sure: we can write the information down somewhere, but the table itself cannot enforce the FD in terms of the FD "key"
- FD information is so important that it may be worth "staying with" 3NF to preserve dependency information
  - See Textbook discussion Chapter 7.3.3
- Remember: any BCNF relation is automatically also in 3NF
- 3NF can be viewed as a "relaxation" of BCNF that preserves dependencies

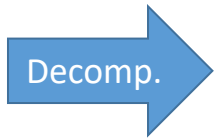
Decompositions & joins



Decomposition should be *lossless*, so that joining restores the original relation.

Decomposition & join – lossless example

name	price	category
Gizmo	19.99	Gadget
OneClick	24.99	Camera
Gizmo	19.99	Camera



name	price
Gizmo	19.99
OneClick	24.99
Gizmo	19.99

name	category
Gizmo	Gadget
OneClick	Camera
Gizmo	Camera

Decomposition & join – lossy example

name	price	category
Gizmo	19.99	Gadget
OneClick	24.99	Camera
Gizmo	19.99	Camera

Decomp.

price	category
19.99	Gadget
24.99	Camera
19.99	Camera

Join

name	price	category
Gizmo	19.99	Gadget
OneClick	24.99	Camera
Gizmo	24.99	Camera
OneClick	19.99	Camera
Gizmo	19.99	Camera

name	category
Gizmo	Gadget
OneClick	Camera
Gizmo	Camera

Loses the association between **name** and **price**.

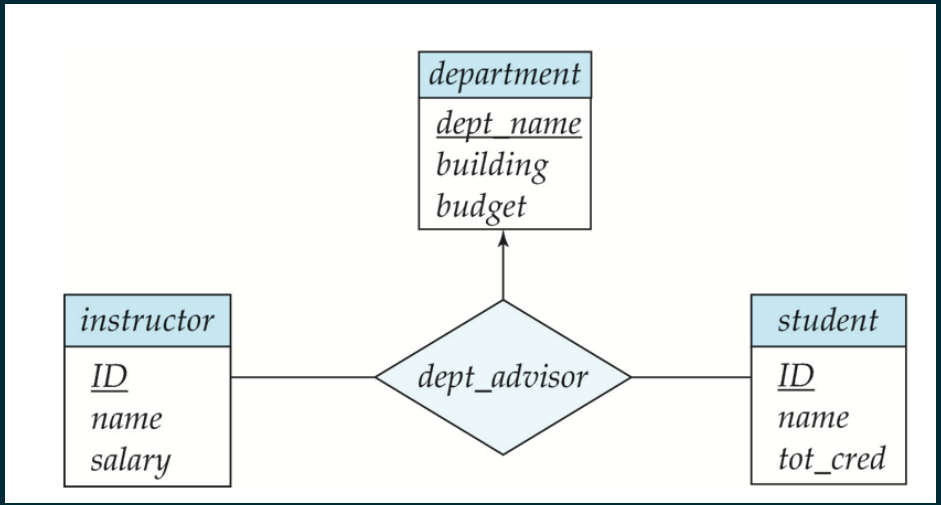
Lossless-Join Decomposition: I

- We can use functional dependencies to determine whether a decomposition of $r(R)$ into R_1, R_2 is "lossy"
- A decomposition is lossless if $\Pi_{R_1(r)} \bowtie \Pi_{R_2(r)} = r$
  - That is: "we projected r onto R_1 and R_2, computed the natural join, and got back our original relation R"
- Example: will this decomposition of $employee(ID, name, street, city, salary)$ into
  - $employee1(ID, name)$
  - $employee2(name, street, city, salary)$
  - be a lossy decomposition?
- Intuitively: this is a lossy decomposition because we cannot reconstruct the original *employee* relation since "name is not unique"
  - We've lost information about which employees are associated with a given address and salary

Lossless-Join Decomposition: II

- What about a decomposition of $instructor\_department(ID, name, dept\_name, salary)$ into
  - $instructor(ID, name, dept\_name, salary)$
  - $department(dept\_name, building, budget)$
- Why is this decomposition "lossless", but the previous one is "lossy"?
- The rule: R_1 and R_2 are a lossless decomposition of R if either of the following FDs are in F^+
  - $R_1 \cap R_2 \rightarrow R_1$
  - $R_1 \cap R_2 \rightarrow R_2$
- In other words: if $R_1 \cap R_2$ is a superkey of either R_1 or R_2 then we have a lossless decomposition
- Recall: attribute closure algorithm allows us to easily test whether the closure of given attribute(s) form a superkey

Dependency Preservation: I



- Just because a schema design is derived from a lossless decomposition (e.g., is BCNF) does not imply that the design preserves dependencies
- This Figure is an E-R diagram which specifies this constraint: a student may have more than one advisor, but at most one advisor from a given department
- This results in a schema that includes $dept\_advisor(student\_id, instructor\_id, dept\_name)$

Dependency Preservation: II

- So far so good . . . Now we add another constraint: an instructor can act as an advisor for only a single department
- We now have two FDs on *dept_advisor*
  - $instructor\_id \rightarrow dept\_name$
    - FD follows from: "an instructor can act as an advisor for only one department"
  - $student\_id, dept\_name \rightarrow instructor\_id$
    - FD follows from: "a student may have at most one advisor from a given department"
- Q: Is *dept_advisor* in BCNF?
- A: No, because *instructor_id* is not a superkey!
  - This design forces us to repeat the department name once every time an instructor participates in a dept_advisor relationship

Dependency Preservation: III

- To keep *dept_advisor* in BCNF, we decompose and get
  - $(\mathit{student\_id}, \mathit{instructor\_id})$
  - $(\mathit{instructor\_id}, \mathit{dept\_name})$
- Good news: the schema are now BCNF (trivially so) ☺
- Bad news: what happened to the FD $\mathit{student\_id}, \mathit{dept\_name} \rightarrow \mathit{instructor\_id}$?
  - Very hard to enforce a "non-local FD" ☹
  - We say that this schema design is not dependency preserving
- I've already mentioned that the ability to preserve dependencies is one of the reasons that 3NF is still used
  - There is always a lossless-join, dependency-preserving decomposition into 3NF
  - So: may have tradeoff between "dependency-preserving" *versus* "some redundancy" (and resulting anomalies)

Another Example

Given $R = (A, B, C)$ and $F = \{A \to B, B \to C\}$, there are two ways to decompose R

1. $R_1 = (A, B), R_2 = (B, C)$
   - Lossless-join decomposition: $R_1 \cap R_2 = \{B\}$ and $B \to BC$
   - Dependency preserving by inspection
2. $R_1 = (A, B), R_2 = (A, C)$
   - Lossless-join decomposition: $R_1 \cap R_2 = \{A\}$ and $A \to AB$
   - Not dependency preserving: cannot verify that $B \to C$ without computing a natural join of $R_1 \bowtie R_2$

BCNF decomposition can lose FD information

R(name, company, category)
{category, company} → {name}
{name} → {company}

Keys: {category, company}, {category, name}

"Bad" FD

Decomp.

R_1(<u>name</u>, <u>category</u>)
R_2(<u>name</u>, company)
{category, company} → {name}
{name} → {company}

Can't enforce "nonlocal" FD.

name	company	category
Gizmo	GizmoWorks	Gadget
GizmoPlus	GizmoWorks	Gadget

Join

name	category
Gizmo	Gadget
GizmoPlus	Gadget

name	company
Gizmo	GizmoWorks
GizmoPlus	GizmoWorks

3NF avoids losing FD information

R(name, company, category)
{category, company} → {name}
{name} → {company}

BCNF: "Bad"
3NF: "Good" because **company** part of some key.

Keys: {category, company}, {category, name}

3NF allows some redundancies/anomalies

name	company	category
Gizmo	GizmoWorks	Gadget
Gizmo	GizmoWorks	Camera
GizmoPlus	GizmoWorks	Gadget
GizmoPlus	GizmoWorks	Camera
NewThing	NULL	Gadget

Redundancy. Repeating product's company.

Insertion anomaly. Product not yet made by any company.

Summary: I

- In an ideal world, your goal for a relational database design should be
  - Schemas are in BCNF
  - Lossless
  - Schema decomposition preserved all functional dependencies
- Unfortunately, we seen (lecture & textbook) that we can't always achieve these all three goals ☹
- We're sometimes forced to accept one of
  - Losing dependency information due to use of BCNF
  - Redundancy due to use of 3NF

Summary: II

- Textbook says: “generally preferable to opt for BCNF”
- The reasoning basically comes down to:
  - The big advantage of 3NF is “dependency preservation”
  - In practice, we may not be able to exploit those FDs!
- SQL does not provide a direct way of specifying functional dependencies other than superkeys
  - Via the **primary key** or **unique** constraints
- Can specify FDs using assertions, but they are expensive to test
  - Textbook claims: “currently not supported by any of the widely used databases!”
- Therefore: even if we had a dependency preserving decomposition, cannot efficiently test (using SQL)
  - Unless the LHS of the FD is a key

Algorithms Related to Functional Dependency

Note To Self & Students

- This material should become part of a third normalization lecture
  - In current form, is a third-to-a-half of a lecture
- Currently no time in syllabus to do this ☹
- You are responsible for this material (between these slides and the textbook) despite the fact that we probably won't cover this in lecture

Canonical Cover: Concept

- Motivation: Given a set of FDs, DBMS must "roll back" a database change if applying the change will violate any of the functional dependencies
  - Remember: just because the FDs were satisfied at a given moment in time, we cannot assume that the FDs will continue to be satisfied!
- If we can extract a "minimal set" of functional dependencies equivalent to F from F, DBMS can test that smaller set instead of the larger set
- We use the term canonical cover to refer to this "minimal set"
  - Contains no redundant dependencies or redundant parts of dependencies

Canonical Cover: Intuition

From earlier example

Given:

{a, b} → {c}
{a, d} → {b}
{b} → {d}

=

Compute FDs:

{a} → {a}
{b} → {b, d}
{c} → {c}
{d} → {d}
{a, b} → {a, b, c, d}
{a, c} → {a, c}
{a, d} → {a, b, c, d}
{b, c} → {b, c, d}
{b, d} → {b, d}
{c, d} → {c, d}
{a, b, c} → {a, b, c, d}
{a, b, d} → {a, b, c, d}
{a, c, d} → {a, b, c, d}
{b, c, d} → {b, c, d}
{a, b, c, d} → {a, b, c, d}

=

Shorthand version:

{b} → {d}

{a, b} → {c, d}

{a, d} → {b, c}
{b, c} → {d}

{a, b, c} → {d}
{a, b, d} → {c}
{a, c, d} → {b}

F covers *G* *iff* *G* can be inferred from *F*
That is: $G^+ \subseteq F^+$
G and *F* are equivalent *iff* $G^+ = F^+$

Canonical Cover: Examples of Redundancies

- Example: $A \to C$ is redundant in: $\{A \to B, B \to C, A \to C\}$
- Example: simplify parts of redundancy on RHS
  - $\{A \to B, B \to C, A \to CD\}$ can be simplified to $\{A \to B, B \to C, A \to D\}$
- Example: simplify parts of redundancy on LHS
  - $\{A \to B, B \to C, AC \to D\}$ can be simplified to $\{A \to B, B \to C, A \to D\}$

Extraneous Attributes

- We want an algorithm that will compute the canonical cover. This can be done using the concept of extraneous attributes
- Consider a set F of functional dependencies and the functional dependency $\alpha \rightarrow \beta$ in F.
  - We say that attribute *A* is extraneous in α if $A \in \alpha$ and *F* logically implies $(F - \{\alpha \rightarrow \beta\}) \cup \{(\alpha - A) \rightarrow \beta\}$
  - We say that attribute *A* is extraneous in β if $A \in B$ and the set of functional dependencies $(F - \{\alpha \rightarrow \beta\}) \cup \{\alpha \rightarrow (\beta - A)\}$ logically implies *F*

Canonical Cover & Extraneous Attributes

- 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.
- See Texbook Figure 7.9 for an algorithm that implements this definition
- Algorithm requires an algorithm for computing extraneous attributes ☹
  - Next slide …

Attribute Closure Can Compute Extraneous Attributes

- Consider a set F of functional dependencies and the functional dependency $\alpha \rightarrow \beta$ in F
- To test if attribute $A \in \alpha$ is extraneous in α
  1. Compute $(\{\alpha\} - A)^+$ using the dependencies in F
  2. Does $(\{\alpha\} - A)^+$ contain β?
  3. If it does, then A is extraneous in α
- To test whether $A \in \beta$ is extraneous in β
  1. Compute α^+ using only the dependencies in $F' = (F - \{\alpha \rightarrow \beta\}) \cup \{\alpha \rightarrow (\beta - A)\}$
  2. Does α^+ contain A?
  3. If it does, then A is extraneous in β

Bonus Material

Activity – using FD implications

name	color	category	dept	price
Gizmo	Green	Gadget	Toys	49
Widget	Black	Gadget	Toys	59
Gizmo	Green	Whatsit	Garden	99

Reflexivity: If $B \subseteq A$, then $A \rightarrow B$.
Augmentation: If $A \rightarrow B$, then $AC \rightarrow BC$.
Transitivity: If $A \rightarrow B$ and $B \rightarrow C$, then $A \rightarrow C$
Union: If $A \rightarrow B$ and $A \rightarrow C$, then $A \rightarrow BC$.
Decomposition: If $A \rightarrow BC$, then $A \rightarrow B$ and $A \rightarrow C$.
Pseudo-transitivity: If $A \rightarrow B$ and $BC \rightarrow D$, then $AC \rightarrow D$.

Given:

{name} → {color}
{category} → {dept}
{color, category} → {price}

Logical implication:

{name, category} → {price}
{name, category} → {name}
{name, category} → {color}
{name, category} → {category}
{name, category} → {color, category}

Rule(s)?

Today's Lecture: Wrapping it Up

Warmup

Functional Dependencies, Implications & Closures

Boyce-Codd Normal Form & BCNF Decompositions

Lossless Joins & Preserving FD Information

Algorithms Related to Functional Dependency

Bonus Material

Readings

- Responsible for
  - Textbook Chapter 7 through Chapter 7.3 (in detail)
- Also responsible for Chapter 7.4-7.5 (at least to the point of being able to run the algorithms "by hand")
- Not responsible for material on *Multivalued dependencies* (which are a motivation for 4NF)
  - Textbook: Chapter 7.6
- Skim the rest of Chapter 7