Relational Database Design

Relational Database Design

- First Normal Form
- Pitfalls in Relational Database Design
- Functional Dependencies
- Decomposition
- Boyce-Codd Normal Form
- Third Normal Form
- Multivalued Dependencies and Fourth Normal Form
- Overall Database Design Process

First Normal Form

- Domain is atomic if its elements are considered to be indivisible units
  - Examples of non-atomic domains:
    - Set of names, composite attributes
    - Identification numbers like CS101 that can be broken up into parts
    - Ii requires extra programming
    - If such identification numbers are used as primary keys
      - When an employee changes department
      - The employee's identification number must be changed

First Normal Form

- A relational schema R is in **first normal form** if the domains of all attributes **of R are atomic**
- Non-atomic values complicate storage and encourage redundant (repeated) storage of data
  - E.g. Set of accounts stored with each customer, and set of owners stored with each account
  - We assume all relations are in first normal form

In a onference paper by Edgar Codd, 1971

- A character string would seem not to be atomic, as the RDBMS typically provides operators to decompose it into substrings.
- A fixed-point number would seem not to be atomic, as the RDBMS typically provides operators to decompose it into integer and fractional components.
- An ISBN would seem not to be atomic, as it includes language and publisher identifier.

First Normal Form

Customer			
Customer ID	**First Name**	**Surname**	**Telephone Number**
123	Pooja	Singh	555-861-2025, 192-122-1111
456	Zhang	San	(555) 403-1659 Ext. 53; 182-929-2929
789	John	Doe	555-808-9633

Customer			
Customer ID	**First Name**	**Surname**	**Telephone Number**
123	Pooja	Singh	555-861-2025
123	Pooja	Singh	192-122-1111
456	Zhang	San	182-929-2929
456	Zhang	San	(555) 403-1659 Ext. 53
789	John	Doe	555-808-9633

First Normal Form

Customer Name		
Customer ID	**First Name**	**Surname**
123	Pooja	Singh
456	Zhang	San
789	John	Doe

Customer Telephone Number	
Customer ID	**Telephone Number**
123	555-861-2025
123	192-122-1111
456	(555) 403-1659 Ext. 53
456	182-929-2929
789	555-808-9633

When we design a relational database:
We first list those functional dependencies that must always hold.

In the banking example
List of dependencies includes:

Pitfalls in Relational Database Design

- Relational database design requires that we find a "good" **collection of relation schemas**. A **bad** design may lead to
  - **Repetition of Information**.
  - Inability to represent **certain information.**
- Design Goals:
  - **Avoid redundant data**
  - Ensure that **relationships among attributes** are represented
  - Facilitate the checking of updates for violation of database **integrity constraints**.

branch-name	*branch-city*	*assets*	*customer-name*	*loan-number*	*amount*
Downtown	Brooklyn	9000000	Jones	L-17	1000
Redwood	Palo Alto	2100000	Smith	L-23	2000
Perryridge	Horseneck	1700000	Hayes	L-15	1500
Downtown	Brooklyn	9000000	Jackson	L-14	1500
Mianus	Horseneck	400000	Jones	L-93	500
Round Hill	Horseneck	8000000	Turner	L-11	900
Pownal	Bennington	300000	Williams	L-29	1200
North Town	Rye	3700000	Hayes	L-16	1300
Downtown	Brooklyn	9000000	Johnson	L-18	2000
Perryridge	Horseneck	1700000	Glenn	L-25	2500
Brighton	Brooklyn	7100000	Brooks	L-10	2200

Figure 7.1 Sample *lending* relation.

Update:
(**Perryridge**, **Horseneck,** 1700000, Adams, **L-31, 1500**)

- **t[assets]** is the asset figure for the branch named t[branch-name].
- **t[branch-city]** is the city in which the branch named t[branch-name] is located.
- **t[loan-number]** is the number assigned to a loan made by the branch named **t[branch-name]** to the customer named t[customer-name].
- **t[amount]** is the amount of the loan whose number is t[loan-number].

Example

- Consider the relation schema:

  Lending-schema = (*branch-name, branch-city, assets, customer-name, loan-number, amount)*

branch-name	*branch-city*	*assets*	*customer-name*	*loan-number*	*amount*
Downtown	Brooklyn	9000000	Jones	L-17	1000
Redwood	Palo Alto	2100000	Smith	L-23	2000
Perryridge	Horseneck	1700000	Hayes	L-15	1500
Downtown	Brooklyn	9000000	Jackson	L-14	1500

- Redundancy:
  - Data for ***branch-name, branch-city,*** **assets** are repeated for each loan that a branch makes
  - **Wastes space**
  - **Complicates updating**, introducing possibility of inconsistency of *assets* value
- Null values
  - **Cannot store information about a branch** if no loans exist
  - Can use **null values**, but they are difficult to handle.

We say that the functional dependency **branch-name → assets** holds on Lending-schema

branch-name	*branch-city*	*assets*	*customer-name*	*loan-number*	*amount*
Downtown	Brooklyn	9000000	Jones	L-17	1000
Redwood	Palo Alto	2100000	Smith	L-23	2000
Perryridge	Horseneck	1700000	Hayes	L-15	1500
Downtown	Brooklyn	9000000	Jackson	L-14	1500
Mianus	Horseneck	400000	Jones	L-93	500
Round Hill	Horseneck	8000000	Turner	L-11	900
Pownal	Bennington	300000	Williams	L-29	1200
North Town	Rye	3700000	Hayes	L-16	1300
Downtown	Brooklyn	9000000	Johnson	L-18	2000
Perryridge	Horseneck	1700000	Glenn	L-25	2500
Brighton	Brooklyn	7100000	Brooks	L-10	2200

We do not expect the functional dependency **branchname → loan-number** to hold.

branch-name	*branch-city*	*assets*	*customer-name*	*loan-number*	*amount*
Downtown	Brooklyn	9000000	Jones	L-17	1000
Redwood	Palo Alto	2100000	Smith	L-23	2000
Perryridge	Horseneck	1700000	Hayes	L-15	1500
Downtown	Brooklyn	9000000	Jackson	L-14	1500
Mianus	Horseneck	400000	Jones	L-93	500
Round Hill	Horseneck	8000000	Turner	L-11	900
Pownal	Bennington	300000	Williams	L-29	1200
North Town	Rye	3700000	Hayes	L-16	1300
Downtown	Brooklyn	9000000	Johnson	L-18	2000
Perryridge	Horseneck	1700000	Glenn	L-25	2500
Brighton	Brooklyn	7100000	Brooks	L-10	2200

We cannot represent directly
the information concerning a branch (branch-name, branch-city, assets

branch-name	*branch-city*	*assets*	*customer-name*	*loan-number*	*amount*
Downtown	Brooklyn	9000000	Jones	L-17	1000
Redwood	Palo Alto	2100000	Smith	L-23	2000
Perryridge	Horseneck	1700000	Hayes	L-15	1500
Downtown	Brooklyn	9000000	Jackson	L-14	1500
Mianus	Horseneck	400000	Jones	L-93	500
Round Hill	Horseneck	8000000	Turner	L-11	900
Pownal	Bennington	300000	Williams	L-29	1200
North Town	Rye	3700000	Hayes	L-16	1300
Downtown	Brooklyn	9000000	Johnson	L-18	2000
Perryridge	Horseneck	1700000	Glenn	L-25	2500
Brighton	Brooklyn	7100000	Brooks	L-10	2200

Repetition of information

Inability to represent certain information

Decomposition

- Decompose the relation schema *Lending-schema* into:

Branch-schema = (branch-name, branch-city,assets)

Loan-info-schema = (customer-name, loan-number, branch-name, amount)

- **All attributes** of an **original schema (*R*)** must appear in the decomposition **(*R*$_1$, *R*$_2$*):***

$$R = R_1 \cup R_2$$

- **Lossless-join** decomposition.
  For all possible relations *r* on **schema *R***

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

Example of Non Lossless-Join Decomposition

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

  $R_2 = (A)$ $R_2 = (B)$

A	B
α	1
α	2
β	1

r

A
α
β

$\Pi_A(r)$

B
1
2

$\Pi_{B(r)}$

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

A	B
α	1
α	2
β	1
β	2

Goal — Devise a Theory for the Following

- Decide whether a particular relation *R* is in “good” form.
- In the case that a relation *R* is not in “good” form, decompose it into a set of relations {R_1, R_2, ..., R_n} such that
  - each relation is in good form
  - the decomposition is a **lossless-join** decomposition
- Our theory is based on:
  - **functional dependencies**
  - **multivalued dependencies**

Functional Dependencies

- **Constraints** on the set of **legal relations.**
- Require that the value for a **certain set of attributes determines uniquely** the value **for another set of attributes**.
- A functional dependency is a generalization of the notion of a ***key***.

Example:

branch-name → assets

Functional Dependencies

A subset *K* of *R* is a **superkey** of *R* if, in any legal relation *r*(*R*),

for all pairs

$$t_1 \neq t_2, \text{ then } t_1[K] \neq t_2[K].$$

That is, no two tuples in any legal relation *r*(*R*) may have the same **value on attribute set *K***.

The *loan* relation

loan_number	*branch_name*	*amount*
L-11	Round Hill	900
L-14	Downtown	1500
L-15	Perryridge	1500
L-16	Perryridge	1300
L-17	Downtown	1000
L-23	Redwood	2000
L-93	Mianus	500

L-15	Perryridge	1500

Using the functional-dependency notation,
we say that K is a superkey of R
if $\mathbf{K \rightarrow R.}$
That is, K is a superkey

If, whenever **t1[K] = t2[K],**
it is also the case that **t1[R] = t2[R]**

(that is, t1 = t2).

Functional Dependencies (Cont.)

- Let *R* be a relation schema

$$\alpha \subseteq R \quad and \quad \beta \subseteq R$$

- The functional dependency

$$\alpha \rightarrow \beta$$

holds on *R* if and only **if for any legal relations *r*(R)**, whenever **any two tuples** t_1 and t_2 of *r* agree on the attributes α, they also agree on the attributes β. That is,

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

Functional Dependencies (Cont.)

A	*B*	*C*	*D*
a1	b1	C1	d1
a1	B2	C1	d2
a2	B2	C2	d2
a2	B3	C2	d3
a3	B3	C2	D4

A → C is satisfied

The functional dependency *C → A is not* satisfied,
consider the tuples *t1 = (a2, b3, c2, d3)*
t2 = (a3, b3, c2, d4).

r satisfies AB → D.
There is no pair of distinct tuples *t1 and t2 such that t1[AB] = t2[AB].*
Therefore, if t1[AB] = t2[AB], it must be that t1 = t2
and, thus, *t1[D] = t2[D]. So, r satisfies AB → D.*

Functional Dependencies (Cont.)

Some functional dependencies are said to be **trivial** because they are satisfied by all relations.

For example, $A \rightarrow A$ is satisfied by all relations involving attribute A.

For all tuples t_1, and t_2 such that $t_1[A] = t_2[A]$, it is the case that $t_1[A] = t_2[A]$.

Similarly, $AB \rightarrow A$, is satisfied by all relations involving attribute A.

In general, a functional dependency of the form $\alpha \rightarrow \beta$ is **trivial** if $\beta \subseteq \alpha$.

Functional Dependencies (Cont.)

- ***K* is a superkey** for relation schema *R* if and only if $K \rightarrow R$
- ***K* is a candidate** key for *R* i**f and only if**
  - $K \rightarrow R$, and
  - for no $\alpha \subset K$, $\alpha \rightarrow R$
- Functional dependencies allow us to express **constraints** that cannot be expressed using **superkeys**. Consider the schema:

  Loan-info-schema = (customer-name, loan-number, branch-name, amount).

  We **expect this set** of functional dependencies to hold:

  loan-number $\rightarrow$ *amount*
  loan-number $\rightarrow$ *branch-name*

  but would **not expect** the following to hold:

  loan-number $\rightarrow$ *customer-name*

branch-name	*branch-city*	*assets*	*customer-name*	*loan-number*	*amount*
Downtown	Brooklyn	9000000	Jones	L-17	1000
Redwood	Palo Alto	2100000	Smith	L-23	2000
Perryridge	Horseneck	1700000	Hayes	L-15	1500
Downtown	Brooklyn	9000000	Jackson	L-14	1500
Mianus	Horseneck	400000	Jones	L-93	500
Round Hill	Horseneck	8000000	Turner	L-11	900
Pownal	Bennington	300000	Williams	L-29	1200
North Town	Rye	3700000	Hayes	L-16	1300
Downtown	Brooklyn	9000000	Johnson	L-18	2000
Perryridge	Horseneck	1700000	Glenn	L-25	2500
Brighton	Brooklyn	7100000	Brooks	L-10	2200

Figure 7.1 Sample *lending* relation.

We say that the functional dependency **branch-name → assets** holds on Lending-schema

branch-name	*branch-city*	*assets*	*customer-name*	*loan-number*	*amount*
Downtown	Brooklyn	9000000	Jones	L-17	1000
Redwood	Palo Alto	2100000	Smith	L-23	2000
Perryridge	Horseneck	1700000	Hayes	L-15	1500
Downtown	Brooklyn	9000000	Jackson	L-14	1500
Mianus	Horseneck	400000	Jones	L-93	500
Round Hill	Horseneck	8000000	Turner	L-11	900
Pownal	Bennington	300000	Williams	L-29	1200
North Town	Rye	3700000	Hayes	L-16	1300
Downtown	Brooklyn	9000000	Johnson	L-18	2000
Perryridge	Horseneck	1700000	Glenn	L-25	2500
Brighton	Brooklyn	7100000	Brooks	L-10	2200

The *depositor* Relation

customer-name	*account-number*
Hayes	A-102
Johnson	A-101
Johnson	A-201
Jones	A-217
Lindsay	A-222
Smith	A-215
Turner	A-305

The *customer* Relation

customer-name	*customer-street*	*customer-city*
Adams	Spring	Pittsfield
Brooks	Senator	Brooklyn
Curry	North	Rye
Glenn	Sand Hill	Woodside
Green	Walnut	Stamford
Hayes	Main	Harrison
Johnson	Alma	Palo Alto
Jones	Main	Harrison
Lindsay	Park	Pittsfield
Smith	North	Rye
Turner	Putnam	Stamford
Williams	Nassau	Princeton

On Depositor-schema:
No functional dependencies

On Customer-schema:
customer-name→ customer-city
customer-name→ customer-street

The *borrower* relation

customer_name	*loan_number*
Adams	L-16
Curry	L-93
Hayes	L-15
Jackson	L-14
Jones	L-17
Smith	L-11
Smith	L-23
Williams	L-17

The *loan* relation

loan_number	*branch_name*	*amount*
L-11	Round Hill	900
L-14	Downtown	1500
L-15	Perryridge	1500
L-16	Perryridge	1300
L-17	Downtown	1000
L-23	Redwood	2000
L-93	Mianus	500

On Borrower-schema:
No functional dependencies

- On Loan-schema:

loan-number → amount
loan-number → branch-name

The *branch* relation

branch_name	*branch_city*	*assets*
Brighton	Brooklyn	7100000
Downtown	Brooklyn	9000000
Mianus	Horseneck	400000
North Town	Rye	3700000
Perryridge	Horseneck	1700000
Pownal	Bennington	300000
Redwood	Palo Alto	2100000
Round Hill	Horseneck	8000000

The *account* relation

account-number	*branch-name*	*balance*
A-101	Downtown	500
A-102	Perryridge	400
A-201	Brighton	900
A-215	Mianus	700
A-217	Brighton	750
A-222	Redwood	700
A-305	Round Hill	350

On Branch-schema:
branch-name →branch-city
branch-name →assets

On Account-schema:
account-number → branch-name
account-number → balance

Use of Functional Dependencies

- We use functional dependencies to:
  - **test relations** to see if they **are legal** under a given **set of functional dependencies.**
    - If a **relation *r* is legal** under a set *F* of functional dependencies, we say that ***r* satisfies *F.***
  - **specify constraints** on the set of legal relations
    - We say that ***F* holds on *R*** if **all legal relations on *R* satisfy** the set of functional dependencies *F.*

Use of Functional Dependencies

- Note: A **specific instance** of a relation schema may **satisfy a functional dependency** even if the functional dependency does not hold on **all legal** instances.
  - For example, a specific instance of *Loan-schema* may, by chance, satisfy ***loan-number → customer-name***.

Functional Dependencies (Cont.)

- ***A* functional** dependency is **trivial** if it is satisfied by **all instances of a relation**
  - *E.g.*
    - *customer-name, loan-number* $\rightarrow$ *customer-name*
    - *customer-name* $\rightarrow$ *customer-name*
  - In general, $\alpha \rightarrow \beta$ is **trivial** **if** $\beta \subseteq \alpha$

Closure of a Set of Functional Dependencies

- Given a set *F* set of functional dependencies, there are certain **other functional dependencies** that **are logically implied** by *F*.
  - E.g. If $A \rightarrow B$ and $B \rightarrow C$, then we can infer that $A \rightarrow C$
- The set of all functional dependencies logically implied by *F* is the *closure* of *F*.
- We denote the ***closure* of *F* by F^+.**

Functional Dependencies (Cont.)

$A \rightarrow B$

$A \rightarrow C$

$CG \rightarrow H$

$CG \rightarrow I$

$B \rightarrow H$

The functional dependency

$A \rightarrow H$

Suppose that, ***t*1 and *t*2** are tuples such that

$t_1[A] = t_2[A]$

Since we are given that $A \rightarrow B$,

$t_1[B] = t_2[B]$

Then, since we are given that $B \rightarrow H$

$t_1[H] = t_2[H]$

Closure of a Set of Functional Dependencies

- We can find all of F^+ by applying Armstrong's Axioms:
  - if $\beta \subseteq \alpha$, then $\alpha \rightarrow \beta$ **(reflexivity)**
  - if $\alpha \rightarrow \beta$, then $\gamma\ \alpha \rightarrow \gamma\ \beta$ **(augmentation)**
  - if $\alpha \rightarrow \beta$, and $\beta \rightarrow \gamma$, then $\alpha \rightarrow \gamma$ **(transitivity)**
- These rules are
  - **sound** (generate only functional dependencies that **actually hold**) and
  - **complete** (generate **all functional** dependencies tha hold).

Example

- $R = (A, B, C, G, H, I)$

$$F = \{ \; A \rightarrow B$$
$$A \rightarrow C$$
$$CG \rightarrow H$$
$$CG \rightarrow I$$
$$B \rightarrow H\}$$

Example

some members of F^+

- $CG \rightarrow HI$
  - from ***$CG \rightarrow H$ and $CG \rightarrow I$*** *:* "union rule" can be inferred from
    - definition of functional dependencies, or
    - Augmentation of $CG \rightarrow I$
      - » to infer $CG \rightarrow CGI$ ***{that is, $CGCG \rightarrow CGI$}***
    - Augmentation of $CG \rightarrow H$
      - » to infer ***$CGI \rightarrow HI$*** *{****that is, $CGI \rightarrow CGI$****}*
    - And transitivity
      - » $CG \rightarrow HI$

- $R = (A, B, C, G, H, I)$

$$F = \{ A \rightarrow B,\ A \rightarrow C,\ CG \rightarrow H,\ CG \rightarrow I,\ B \rightarrow H\}$$

Procedure for Computing F^+

- To compute the closure of a set of functional dependencies F:

$F^+ = F$
repeat
    for each functional dependency f in F^+
        apply **reflexivity and augmentation** rules on f
        add the resulting functional dependencies to F^+
    for each pair of functional dependencies f_1 and f_2 in F^+
        if f_1 and f_2 can be combined using **transitivity**
            then add the resulting functional dependency to F^+
until F^+ does not change any further

NOTE: We will see an alternative procedure for this task later

Closure of Functional Dependencies (Cont.)

- We can further simplify manual computation of F^+ by using the following additional rules.
  - If $\alpha \rightarrow \beta$ holds *a*nd $\alpha \rightarrow \gamma$ holds, then $\alpha \rightarrow \beta\gamma$ holds **(union)**
  - If $\alpha \rightarrow \beta\gamma$ holds, then $\alpha \rightarrow \beta$ holds and $\alpha \rightarrow \gamma$ holds **(decomposition)**
  - If $\alpha \rightarrow \beta$ holds *a*nd $\gamma\beta \rightarrow \delta$ holds, then $\alpha\gamma \rightarrow \delta$ holds **(pseudotransitivity)**

  The above rules can be inferred from **Armstrong's axioms**.

Closure of Attribute Sets

- Given a set of attributes α, define the ***closure*** **of** α **under** ***F*** (denoted by α^+) as the **set of attributes** that are functionally **determined by** α **under** ***F:***

$$\alpha \rightarrow \beta \text{ is in } F^+ \Leftrightarrow \beta \subseteq \alpha^+$$

- Algorithm to compute α^+, the closure of α under *F*

result := α;

while (changes to *result*) **do**

for each f1$\rightarrow$ f2 **in** *F* **do**

begin

if f1 $\subseteq$ *result* **then** *result* := *result* $\cup$ f2

end

Example of Attribute Set Closure

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

Example of Attribute Set Closure

- $(AG)^+$
  1. $result = AG$
  2. $result = ABCG$ $(A \rightarrow C \text{ and } A \rightarrow B)$
  3. $result = ABCGH$ $(CG \rightarrow H \text{ and } CG \subseteq AGBC)$
  4. $result = ABCGHI$ $(CG \rightarrow I \text{ and } CG \subseteq AGBCH)$

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

Closure of Functional Dependencies (Cont.)

- Given the following FDs
- A→ BC , CD→ E, E→ C
- D→ AEH, ABH → BD, DH→ BCA
- Find out, whether BCD → H or not from the following Functional Dependencies?
- **Solution :**Compute the closure of
- $BCD = BCD^+$
- $BCD^+ = BCD$

  = ABCDEH {as D → AEH}

  Hence BCD → H is true as the closure contains H.

Example of Attribute Set Closure

- Suppose we are given relation R with attributes A,B,C,D and

FDs A→BC B→CD

- (B^+) = {BCD}
- (C^+) = {C}
- (D^+) = {D}
- (AB^+) = {ABCD}
- (AC^+) = {ABCD}
- (AD^+) = {ABCD}
- (BC^+) = {BCD}
- (BD^+) = {BCD}
- (CD^+) = {CD

Example of Attribute Set Closure

- relation schema R = {A, B, C, D, E}.
  - A → BC
  - CD → E
  - B → D
  - E → A

Example of Attribute Set Closure

- $(A)^+ = \{ABCDE\}$
- $(AB)^+ = \{ABCDE\}$
- $(AC)^+ = \{ABCDE\}$
- $(BC)^+ = \{ABCDE\}$
- $(BD)^+ = \{BD\}$
- $(BE)^+ = \{ABCDE\}$
- $(CD)^+ = \{ABCDE\}$
- $(CE)^+ = \{ABCDE\}$
- $(DE)^+ = \{ABCDE\}$
- $(ABC)^+ = \{ABCDE\}$

- relation schema R = {A, B, C, D, E}.
  - $A \rightarrow BC$
  - $CD \rightarrow E$
  - $B \rightarrow D$
  - $E \rightarrow A$

Uses of Attribute Closure

There are several uses of the attribute closure algorithm:

- Testing for superkey:
  - To test if α is a superkey, we compute α^+, and check if α^+ **contains all attributes of *R*.**

Example of Attribute Set Closure

- $(AG)^+$
  1. *result* = *AG*
  2. *result* = *ABCG* $(A \rightarrow C$ and $A \rightarrow B)$
  3. *result* = *ABCGH* $(CG \rightarrow H$ and $CG \subseteq AGBC)$
  4. *result* = *ABCGHI* $(CG \rightarrow I$ and $CG \subseteq AGBCH)$
- Is *AG* a candidate key?
  1. Is AG a super key?
     1. Does $AG \rightarrow R$? == Is $(AG)^+ \supseteq$ R
  2. Is any subset of AG a superkey?
     1. Does $A \rightarrow R$? == Is $(A)^+ \supseteq$ R
     2. Does $G \rightarrow R$? == Is $(G)^+ \supseteq$ R

Closure of Functional Dependencies (Cont.)

- relational scheme R with attributes A,B,C,D,F and the FDs
- $A \rightarrow BC, \quad B \rightarrow E, \quad CD \rightarrow EF$
- Prove that functional dependency $AD \rightarrow F$ holds in R.
- Compute the closure of AD
- **$(AD)^+ = ADBCEF$**
  - Hence $AD \rightarrow E$ is true as the closure contains E.

Uses of Attribute Closure

- Testing functional dependencies
  - To check if a functional dependency $\alpha \rightarrow \beta$ **holds** (or, in other words, is in F^+), just check **if** $\beta \subseteq \alpha^+$**.**
  - That is, we compute α^+ by using attribute closure, and then check if it contains β.
  - Is a simple and cheap test, and very useful

Uses of Attribute Closure

- Computing closure of F
  - For each $\gamma \subseteq \boldsymbol{R}$,
  - we find the closure γ^{+},
    - **and for each $S \subseteq \gamma^{+}$,**
    - we output a functional dependency $\gamma \rightarrow \boldsymbol{S}$.

Canonical Cover

- Sets of functional dependencies may have **redundant dependencies** that can be inferred from the others
  - Eg: $A \rightarrow C$ is redundant in: $\{A \rightarrow B, \quad B \rightarrow C, \quad A \rightarrow C\}$
  - Parts of a functional dependency may be redundant
    - E.g. on RHS: $\{A \rightarrow B, \quad B \rightarrow C, \quad A \rightarrow CD\}$ can be simplified to

      $\{A \rightarrow B, \quad B \rightarrow C, \quad A \rightarrow D\}$
    - E.g. on LHS: $\{A \rightarrow B, \quad B \rightarrow C, \quad AC \rightarrow D\}$ can be simplified to

      $\{A \rightarrow B, \quad B \rightarrow C, \quad A \rightarrow D\}$
- Intuitively, a canonical cover of F is a "**minimal" set of functional** dependencies equivalent to F, having no redundant dependencies or redundant parts of dependencies

Extraneous Attributes

An attribute of a functional dependency is said to be **extraneous**
If we can **remove it**
without **changing the closure of the set of functional dependencies**.

Extraneous Attributes

- Consider a set *F* of functional dependencies and the functional dependency $\alpha \rightarrow \beta$ in *F*.
  - Attribute A is extraneous in α if $A \in \alpha$
    and ***F* logically implies $(F - \{\alpha \rightarrow \beta\}) \cup \{(\alpha - A) \rightarrow \beta\}$.**

- **Example**: Given $F = \{A \rightarrow C, AB \rightarrow C\}$
  - ***B*** is extraneous in $\boldsymbol{AB \rightarrow C}$

  Because $\{A \rightarrow C, AB \rightarrow C\}$ logically implies $A \rightarrow C$

  (I.e. the result of dropping *B* from $AB \rightarrow C$).

  $\boldsymbol{A \rightarrow C}$

  $\boldsymbol{AB \rightarrow CB}$

  $\boldsymbol{AB \rightarrow C}$ **,** $\boldsymbol{AB \rightarrow B}$

Extraneous Attributes

- Consider a set *F* of functional dependencies and the functional dependency $\alpha \rightarrow \beta$ in *F*.
  - Attribute *A* is extraneous in β if $A \in \beta$
    and the set of functional dependencies
    $(F - \{\alpha \rightarrow \beta\}) \cup \{\alpha \rightarrow (\beta - A)\}$ logically implies *F.*
- Example: Given $F = \{A \rightarrow C, AB \rightarrow CD\}$
  - ***C*** is extraneous in $AB \rightarrow CD$

  Since AB $\rightarrow C$ can be inferred even after deleting *C*

AB → CD

AB → D, AB → C

A → C

AB → BC

AB → C , AB → B

Extraneous Attributes

- *Note:* implication in the opposite direction is **trivial** in each of the cases above, since a “stronger” functional dependency always implies a weaker one

Testing if an Attribute is Extraneous

- 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. check that $(\{\alpha\} - A)^+$ **contains A; if it does,** *A* is extraneous
- To test if attribute $A \in \beta$ is extraneous in β
  1. compute α^+ using only the dependencies in
     $F' = (F - \{\alpha \rightarrow \beta\}) \cup \{\alpha \rightarrow (\beta - A)\}$,
  2. check that α^+ contains *A;* if it does, *A* is extraneous

Testing if an Attribute is Extraneous

Let *F* be the given set of functional dependencies that hold on *R*.

Consider an attribute *A* in a **dependency $\alpha \rightarrow \beta$.**

- If $A \in \alpha$, to check if *A* is extraneous, **let $\gamma = \alpha - \{A\}$, and check if $\gamma \rightarrow \beta$** can be inferred from *F*.
- To do so, compute γ^+ (the closure of γ) under *F*; if γ^+ ncludes all **attributes in β,** then *A* is extraneous in α.

Testing if an Attribute is Extraneous

Let *F* be the given set of functional dependencies that hold on *R*.

Consider an attribute ***A* in a dependency $\alpha \rightarrow \beta$.**

- If $A \in \beta$, to check if *A* is extraneous consider the set

$F = (F - \{\alpha \rightarrow \beta\}) \cup \{\alpha \rightarrow (\beta - A)\}$

- Check if $\alpha \rightarrow A$ can be inferred from *F*.
- To do so, compute α^+ (the closure of α) under *F*;
- If α^+ **includes *A***, then *A* is extraneous in β.

Canonical Cover

- A *canonical cover* for *F* is a set of dependencies F_c such that
  - *F* logically implies all dependencies in F_c, and
  - F_c logically implies all dependencies in *F*, and
  - **No** functional dependency in F_c contains an **extraneous** attribute, and
  - Each left side of functional dependency in F_c **is unique**.

Canonical Cover

- To compute a canonical cover for *F*:
  repeat
  Use the union rule to replace any dependencies in *F*
  $\alpha_1 \rightarrow \beta_1$ and $\alpha_1 \rightarrow \beta_2$ with $\alpha_1 \rightarrow \beta_1 \beta_2$
  Find a functional dependency $\alpha \rightarrow \beta$ with an **extraneous** attribute either in α or in β
  If an extraneous attribute is found, **delete** it from $\alpha \rightarrow \beta$
  until *F* does not change
- Note: Union rule may become **applicable after some extraneous attributes have been deleted, so it has to be re-applied**

Example of Computing a Canonical Cover

- $R = (A, B, C)$
  $F = \{A \rightarrow BC$
  $B \rightarrow C$
  $A \rightarrow B$
  $AB \rightarrow C\}$
- Combine $A \rightarrow BC$ and $A \rightarrow B$ into $A \rightarrow BC$
  - Set is now $\{A \rightarrow BC, B \rightarrow C, AB \rightarrow C\}$

Example of Computing a Canonical Cover

- ***A*** is extraneous **in** $AB \rightarrow C$
  - Check if the result of deleting **A** from $AB \rightarrow C$ is implied by the other dependencies
    - Yes: in fact, $B \rightarrow C$ is already present!

    ***F* logically implies (*F* – {*AB* → *C*}) ∪ {*B* → *C*}.**
  - Set is now $\{A \rightarrow BC, B \rightarrow C\}$
- ***C*** is extraneous **in** $A \rightarrow BC$
  - Check if $A \rightarrow C$ is logically implied by $A \rightarrow B$ and the other dependencies
    - Yes*:* using transitivity on $A \rightarrow B$ ***and*** $B \rightarrow$ **C.**
      - Can use attribute closure of *A* in more complex cases
- The canonical cover is: $A \rightarrow B$

Canonical Cover - 1

- Given a relation R with a set of FDs F
- **4 steps to find a canonical cover for F:**
- **Step 1: Decompose all FDs in standard form**
- Replace each FD **X → A1A2…Ak** in **F** with **X→A1, X→A2, …, X→Ak**

Canonical Cover - 2

- **Step 2: Eliminate unnecessary attributes from LHS**
- For every FD **X** $\rightarrow$ **A** in F, **check if the closure of a subset of X determines A**. If so, remove redundant attribute(s) from X

Canonical Cover - 3

- **Step 3: Remove redundant FD(s)**
- For every FD **X** $\rightarrow$ **A** in **G**
  - Remove **X** $\rightarrow$ **A** from **G**, and call the result **G'**
  - Compute **X+** under **G'**
  - If **A** $\in$ **X+,** then **X** $\rightarrow$ **A** is redundant. Hence, we remove the FD **X** $\rightarrow$ **A** from **G** **(**That is, we rename **G'** to **G)**

Canonical Cover - 4

- **Step 4: Make LHS of FDs unique**
- Replace **X→A1, X→A2, …, X→Ak** with **X → A1A2…Ak**

Example of Computing a Canonical Cover

F= functional dependencies :

$AB \rightarrow CD$

$BC \rightarrow D$

Step 1 : Union Simplification :

$AB \rightarrow C$ ..(1)

$AB \rightarrow D$ ..(2)

$BC \rightarrow D$ ..(3)

Example of Computing a Canonical Cover

Step 2 : Removal of Redundant set of FDs -

(i) Find the closure of LHS of (1),

i.e. Compute (AB)+ from (2) and (3) by hiding (1).

(AB)+ = {ABD}, AB → C is non redundant

(II) Similarly for (2),

Compute (AB)+ from (1) and (3) by hiding (2).

(AB)+ = {ABCD}, AB → D is redundant

Remaining FDs = {AB→C, BC→D}

Example of Computing a Canonical Cover

For (3), Compute (BC)+ from (1) by hiding (3).

(BC)+ = {BC}, BC → D is also non redundant

Example of Computing a Canonical Cover

LHS Simplification or Removal of Extraneous Attributes -

FDs remained after step 2

$AB \rightarrow C$ .. (3)

$BC \rightarrow D$ .. (4)

Example of Computing a Canonical Cover

Remove **A from AB** → **C**, and

` find B+ from (4).

If the attribute A exists in the closure, then A is said to be redundant.

B+ = {B} ⇒ , A is Non Redundant

Remove B from AB → C and

find A+ from (4).

A+ = {A} ⇒ , B is Non Redundant.

Example of Computing a Canonical Cover

Remove B from BC → D, and
find C+ from (3).
C+ = {C} ⇒ B is Non Redundant

Remove C from BC → D and
find B+ from (3).
B+ = {B} ⇒ C is Non Redundant

Example of Computing a Canonical Cover

F= functional dependencies :

ABCD $\rightarrow$ E .. (1)

E $\rightarrow$ D .. (2)

AC $\rightarrow$ D .. (3)

A $\rightarrow$ B .. (4)

Removal of redundant set of FDs -

(i) For (1),

Compute (ABCD)+ from (2) (3) and (4) by hiding (1).

(ABCD)+ = {ABCD}

$\Rightarrow$ ABCD $\rightarrow$ E is non redundant.

Example of Computing a Canonical Cover

(ii) For (2),
Compute (E)+ from (1) (3) and (4) by hiding (2).

$(E)^+ = \{E\}$

$\Rightarrow E \rightarrow D$ is non redundant.

(iii) For (3)
Compute (AC)+ from (1) (2) and (4) by hiding (3).

$(AC)^+ = \{ACB\}$

$\Rightarrow AC \rightarrow D$ is non redundant.

(iV) For (4),
Compute (A)+ from (1) (2) and (3) by hiding (4).

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

$\Rightarrow A \rightarrow B$ is non redundant.

Example of Computing a Canonical Cover

Step 3 : LHS Simplification or Removal of Extraneous Attributes -

(i) Remove **A** from **ABCD → E**

(BCD)+ = {BCD} ⇒ **A is Non Redundant**

(II) Remove **B** from ABCD → E

(ACD)+ = {ACDB} ⇒ **B is Redundant.**

ACD → E

(III) Remove C from ACD → E

(AD)+ = {ADB} ⇒ **C is Non Redundant.**

(iV) Remove D from ACD → E,

AC)+ = {ACDB} ⇒ D is Redundant.

Example of Computing a Canonical Cover

Remove A from **AC** $\rightarrow$ **D**

(C)+ = {C} $\Rightarrow$ A is non redundant.

Remove C from AC $\rightarrow$ D

A)+ = {AB} $\Rightarrow$ C is non redundant.

Example of Computing a Canonical Cover

FDs remained after step 3 are :

$AC \rightarrow E$ .. (5)

$E \rightarrow D$ .. (6)

$AC \rightarrow D$ .. (7)

$A \rightarrow B$ .. (8)

Example of Computing a Canonical Cover

Applying step 2 again to find redundant FDs -

(i) For (5), Compute (AC)+).

(AC)+ = {ACDB} , ⇒ AC → E is non redundant.

(ii) For (6), Compute (E)+

(E)+ = {E}, ⇒ E → D is non redundant.

(iii) For (7), Compute (AC)+ .

(AC)+ = {ACEDB}, **⇒ AC → D is redundant.**

(iv) For (8), Compute (A)+

(A)+ = {AB}, ⇒ A → B is non redundant.

Example of Computing a Canonical Cover

Finally Canonical Cover will be

$AC \rightarrow E$

$E \rightarrow D$

$A \rightarrow B$

Example of Computing a Canonical Cover

A canonical cover is "allowed" to have more than one attribute on the right hand side.

A minimal cover cannot.
As an example,

the canonical cover may be "A -> BC"

the minimal cover would be "A -> B, A -> C

Goals of Normalization

- Decide whether a particular relation *R* is in “good” form.
- In the case that a relation *R* is not in “good” form, decompose it into a set of relations {R_1, R_2, ..., R_n} such that
  - each relation is in good form
  - the decomposition is a lossless-join decomposition
- Our theory is based on:
  - **functional dependencies**
  - **multivalued dependencies**

Example of Lossy-Join Decomposition

- Lossy-join decompositions result in information loss.
- Example: Decomposition of $R = (A, B)$

$R_2 = (A)$ $R_2 = (B)$

A	B
α	1
α	2
β	1

r

A
α
β

$\Pi_A(r)$

B
1
2

$\Pi_{B(r)}$

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

A	B
α	1
α	2
β	1
β	2

Lending-schema = (branch-name, branch-city, assets, customer-name, loan-number, amount)

branch-name	*branch-city*	*assets*	*customer-name*	*loan-number*	*amount*
Downtown	Brooklyn	9000000	Jones	L-17	1000
Redwood	Palo Alto	2100000	Smith	L-23	2000
Perryridge	Horseneck	1700000	Hayes	L-15	1500
Downtown	Brooklyn	9000000	Jackson	L-14	1500
Mianus	Horseneck	400000	Jones	L-93	500
Round Hill	Horseneck	8000000	Turner	L-11	900
Pownal	Bennington	300000	Williams	L-29	1200
North Town	Rye	3700000	Hayes	L-16	1300
Downtown	Brooklyn	9000000	Johnson	L-18	2000
Perryridge	Horseneck	1700000	Glenn	L-25	2500
Brighton	Brooklyn	7100000	Brooks	L-10	2200

Figure 7.1 Sample *lending* relation.

Branch-customer-schema =
(branch-name, branch-city, assets, customer-name)

branch-name	*branch-city*	*assets*	*customer-name*	*loan-number*	*amount*
Downtown	Brooklyn	9000000	Jones	L-17	1000
Redwood	Palo Alto	2100000	Smith	L-23	2000
Perryridge	Horseneck	1700000	Hayes	L-15	1500
Downtown	Brooklyn	9000000	Jackson	L-14	1500
Mianus	Horseneck	400000	Jones	L-93	500
Round Hill	Horseneck	8000000	Turner	L-11	900
Pownal	Bennington	300000	Williams	L-29	1200
North Town	Rye	3700000	Hayes	L-16	1300
Downtown	Brooklyn	9000000	Johnson	L-18	2000
Perryridge	Horseneck	1700000	Glenn	L-25	2500
Brighton	Brooklyn	7100000	Brooks	L-10	2200

branch-name	*branch-city*	*assets*	*customer-name*	*loan-number*	*amount*
Downtown	Brooklyn	9000000	Jones	L-17	1000
Redwood	Palo Alto	2100000	Smith	L-23	2000
Perryridge	Horseneck	1700000	Hayes	L-15	1500
Downtown	Brooklyn	9000000	Jackson	L-14	1500
Mianus	Horseneck	400000	Jones	L-93	500
Round Hill	Horseneck	8000000	Turner	L-11	900
Pownal	Bennington	300000	Williams	L-29	1200
North Town	Rye	3700000	Hayes	L-16	1300
Downtown	Brooklyn	9000000	Johnson	L-18	2000
Perryridge	Horseneck	1700000	Glenn	L-25	2500
Brighton	Brooklyn	7100000	Brooks	L-10	2200

Customer-loan-schema =
(customer-name, loan-number, amount)

$$branch\text{-}customer \bowtie customer\text{-}loan$$

branch-name	*branch-city*	*assets*	*customer-name*	*loan-number*	*amount*
Downtown	Brooklyn	9000000	Jones	L-17	1000
Downtown	Brooklyn	9000000	Jones	L-93	500
Redwood	Palo Alto	2100000	Smith	L-23	2000
Perryridge	Horseneck	1700000	Hayes	L-15	1500
Perryridge	Horseneck	1700000	Hayes	L-16	1300
Downtown	Brooklyn	9000000	Jackson	L-14	1500
Mianus	Horseneck	400000	Jones	L-17	1000
Mianus	Horseneck	400000	Jones	L-93	500
Round Hill	Horseneck	8000000	Turner	L-11	900
Pownal	Bennington	300000	Williams	L-29	1200
North Town	Rye	3700000	Hayes	L-15	1500
North Town	Rye	3700000	Hayes	L-16	1300
Downtown	Brooklyn	9000000	Johnson	L-18	2000
Perryridge	Horseneck	1700000	Glenn	L-25	2500
Brighton	Brooklyn	7100000	Brooks	L-10	2200

$$branch\text{-}customer \bowtie customer\text{-}loan$$

branch-name	*branch-city*	*assets*	*customer-name*	*loan-number*	*amount*
Downtown	Brooklyn	9000000	Jones	L-17	1000
Downtown	Brooklyn	9000000	Jones	L-93	500
Redwood	Palo Alto	2100000	Smith	L-23	2000
Perryridge	Horseneck	1700000	Hayes	L-15	1500
Perryridge	Horseneck	1700000	Hayes	L-16	1300
Downtown	Brooklyn	9000000	Jackson	L-14	1500
Mianus	Horseneck	400000	Jones	L-17	1000
Mianus	Horseneck	400000	Jones	L-93	500
Round Hill	Horseneck	8000000	Turner	L-11	900
Pownal	Bennington	300000	Williams	L-29	1200
North Town	Rye	3700000	Hayes	L-15	1500
North Town	Rye	3700000	Hayes	L-16	1300
Downtown	Brooklyn	9000000	Johnson	L-18	2000
Perryridge	Horseneck	1700000	Glenn	L-25	2500
Brighton	Brooklyn	7100000	Brooks	L-10	2200

branch-name	*branch-city*	*assets*	*customer-name*	*loan-number*	*amount*
Downtown	Brooklyn	9000000	Jones	L-17	1000
Redwood	Palo Alto	2100000	Smith	L-23	2000
Perryridge	Horseneck	1700000	Hayes	L-15	1500
Downtown	Brooklyn	9000000	Jackson	L-14	1500
Mianus	Horseneck	400000	Jones	L-93	500
Round Hill	Horseneck	8000000	Turner	L-11	900
Pownal	Bennington	300000	Williams	L-29	1200
North Town	Rye	3700000	Hayes	L-16	1300
Downtown	Brooklyn	9000000	Johnson	L-18	2000
Perryridge	Horseneck	1700000	Glenn	L-25	2500
Brighton	Brooklyn	7100000	Brooks	L-10	2200

$$\Pi_{branch\text{-}name}\left(\sigma_{amount < 1000}\left(branch\text{-}customer \bowtie customer\text{-}loan\right)\right)$$

branch-name	*branch-city*	*assets*	*customer-name*	*loan-number*	*amount*
Downtown	Brooklyn	9000000	Jones	L-17	1000
Downtown	Brooklyn	9000000	Jones	L-93	500
Redwood	Palo Alto	2100000	Smith	L-23	2000
Perryridge	Horseneck	1700000	Hayes	L-15	1500
Perryridge	Horseneck	1700000	Hayes	L-16	1300
Downtown	Brooklyn	9000000	Jackson	L-14	1500
Mianus	Horseneck	400000	Jones	L-17	1000
Mianus	Horseneck	400000	Jones	L-93	500
Round Hill	Horseneck	8000000	Turner	L-11	900
Pownal	Bennington	300000	Williams	L-29	1200
North Town	Rye	3700000	Hayes	L-15	1500
North Town	Rye	3700000	Hayes	L-16	1300
Downtown	Brooklyn	9000000	Johnson	L-18	2000
Perryridge	Horseneck	1700000	Glenn	L-25	2500
Brighton	Brooklyn	7100000	Brooks	L-10	2200

branch-name	*branch-city*	*assets*	*customer-name*	*loan-number*	*amount*
Downtown	Brooklyn	9000000	Jones	L-17	1000
Redwood	Palo Alto	2100000	Smith	L-23	2000
Perryridge	Horseneck	1700000	Hayes	L-15	1500
Downtown	Brooklyn	9000000	Jackson	L-14	1500
Mianus	Horseneck	400000	Jones	L-93	500
Round Hill	Horseneck	8000000	Turner	L-11	900
Pownal	Bennington	300000	Williams	L-29	1200
North Town	Rye	3700000	Hayes	L-16	1300
Downtown	Brooklyn	9000000	Johnson	L-18	2000
Perryridge	Horseneck	1700000	Glenn	L-25	2500
Brighton	Brooklyn	7100000	Brooks	L-10	2200

Problem: If a customer happens to have several loans from different branches, we cannot tell which loan belongs to which branch.
Thus, when we join branch-customer and customer-loan, we obtain not only the tuples we had originally in lending, but also several additional tuples. Although

branch-name	*branch-city*	*assets*	*customer-name*
Downtown	Brooklyn	9000000	Jones
Redwood	Palo Alto	2100000	Smith
Perryridge	Horseneck	1700000	Hayes
Downtown	Brooklyn	9000000	Jackson
Mianus	Horseneck	400000	Jones
Round Hill	Horseneck	8000000	Turner
Pownal	Bennington	300000	Williams
North Town	Rye	3700000	Hayes
Downtown	Brooklyn	9000000	Johnson
Perryridge	Horseneck	1700000	Glenn
Brighton	Brooklyn	7100000	Brooks

customer-name	*loan-number*	*amount*
Jones	L-17	1000
Smith	L-23	2000
Hayes	L-15	1500
Jackson	L-14	1500
Jones	L-93	500
Turner	L-11	900
Williams	L-29	1200
Hayes	L-16	1300
Johnson	L-18	2000
Glenn	L-25	2500
Brooks	L-10	2200

(Downtown, Brooklyn, 9000000, Jones, L-93, 500)
(Perryridge, Horseneck, 1700000, Hayes, L-16, 1300)
(Mianus, Horseneck, 400000, Jones, L-17, 1000)
(North Town, Rye, 3700000, Hayes, L-15, 1500)

Lending-schema = (branch-name, branch-city, assets, customer-name, loan-number, amount)

branch-name	*branch-city*	*assets*	*customer-name*	*loan-number*	*amount*
Downtown	Brooklyn	9000000	Jones	L-17	1000
Redwood	Palo Alto	2100000	Smith	L-23	2000
Perryridge	Horseneck	1700000	Hayes	L-15	1500
Downtown	Brooklyn	9000000	Jackson	L-14	1500
Mianus	Horseneck	400000	Jones	L-93	500
Round Hill	Horseneck	8000000	Turner	L-11	900
Pownal	Bennington	300000	Williams	L-29	1200
North Town	Rye	3700000	Hayes	L-16	1300
Downtown	Brooklyn	9000000	Johnson	L-18	2000
Perryridge	Horseneck	1700000	Glenn	L-25	2500
Brighton	Brooklyn	7100000	Brooks	L-10	2200

Figure 7.1 Sample *lending* relation.

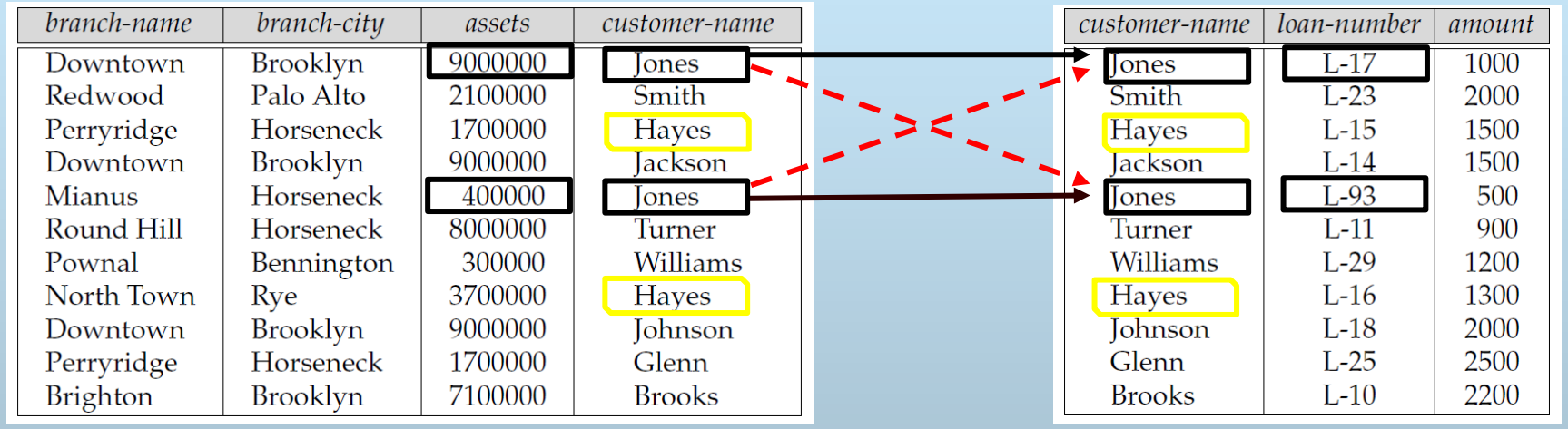
Problem: If a customer happens **to have several loans from different branches**, we cannot tell which loan belongs to which branch.
Thus, when we join branch-customer and customer-loan, we obtain not only the tuples we had originally in lending, but also several additional tuples. Although

branch-name	*branch-city*	*assets*	*customer-name*	*loan-number*	*amount*
Downtown	Brooklyn	9000000	Jones	L-17	1000
Downtown	Brooklyn	9000000	Jones	L-93	500
Redwood	Palo Alto	2100000	Smith	L-23	2000
Perryridge	Horseneck	1700000	Hayes	L-15	1500
Perryridge	Horseneck	1700000	Hayes	L-16	1300
Downtown	Brooklyn	9000000	Jackson	L-14	1500
Mianus	Horseneck	400000	Jones	L-17	1000
Mianus	Horseneck	400000	Jones	L-93	500
Round Hill	Horseneck	8000000	Turner	L-11	900
Pownal	Bennington	300000	Williams	L-29	1200
North Town	Rye	3700000	Hayes	L-15	1500
North Town	Rye	3700000	Hayes	L-16	1300
Downtown	Brooklyn	9000000	Johnson	L-18	2000
Perryridge	Horseneck	1700000	Glenn	L-25	2500
Brighton	Brooklyn	7100000	Brooks	L-10	2200

(Downtown, Brooklyn, 9000000, Jones, L-93, 500)
(Perryridge, Horseneck, 1700000, Hayes, L-16, 1300)
(Mianus, Horseneck, 400000, Jones, L-17, 1000)
(North Town, Rye, 3700000, Hayes, L-15, 1500)

$branch\text{-}customer = \Pi_{branch\text{-}name,\ branch\text{-}city,\ assets,\ customer\text{-}name}\ (lending)$

$customer\text{-}loan = \Pi_{customer\text{-}name,\ loan\text{-}number,\ amount}\ (lending)$



branch-name	branch-city	assets	customer-name
Downtown	Brooklyn	9000000	Jones
Redwood	Palo Alto	2100000	Smith
Perryridge	Horseneck	1700000	Hayes
Downtown	Brooklyn	9000000	Jackson
Mianus	Horseneck	400000	Jones
Round Hill	Horseneck	8000000	Turner
Pownal	Bennington	300000	Williams
North Town	Rye	3700000	Hayes
Downtown	Brooklyn	9000000	Johnson
Perryridge	Horseneck	1700000	Glenn
Brighton	Brooklyn	7100000	Brooks

customer-name	loan-number	amount
Jones	L-17	1000
Smith	L-23	2000
Hayes	L-15	1500
Jackson	L-14	1500
Jones	L-93	500
Turner	L-11	900
Williams	L-29	1200
Hayes	L-16	1300
Johnson	L-18	2000
Glenn	L-25	2500
Brooks	L-10	2200

Branch-customer-schema = (branch-name, branch-city, assets, **customer-name)**
Customer-loan-schema = **(customer-name**, loan-number, amount)

$$branch\text{-}customer \bowtie customer\text{-}loan$$

branch-name	*branch-city*	*assets*	*customer-name*	*loan-number*	*amount*
Downtown	Brooklyn	9000000	Jones	L-17	1000
Downtown	Brooklyn	9000000	Jones	L-93	500
Redwood	Palo Alto	2100000	Smith	L-23	2000
Perryridge	Horseneck	1700000	Hayes	L-15	1500
Perryridge	Horseneck	1700000	Hayes	L-16	1300
Downtown	Brooklyn	9000000	Jackson	L-14	1500
Mianus	Horseneck	400000	Jones	L-17	1000
Mianus	Horseneck	400000	Jones	L-93	500
Round Hill	Horseneck	8000000	Turner	L-11	900
Pownal	Bennington	300000	Williams	L-29	1200
North Town	Rye	3700000	Hayes	L-15	1500
North Town	Rye	3700000	Hayes	L-16	1300
Downtown	Brooklyn	9000000	Johnson	L-18	2000
Perryridge	Horseneck	1700000	Glenn	L-25	2500
Brighton	Brooklyn	7100000	Brooks	L-10	2200

branch-name	*branch-city*	*assets*	*customer-name*	*loan-number*	*amount*
Downtown	Brooklyn	9000000	Jones	L-17	1000
Redwood	Palo Alto	2100000	Smith	L-23	2000
Perryridge	Horseneck	1700000	Hayes	L-15	1500
Downtown	Brooklyn	9000000	Jackson	L-14	1500
Mianus	Horseneck	400000	Jones	L-93	500
Round Hill	Horseneck	8000000	Turner	L-11	900
Pownal	Bennington	300000	Williams	L-29	1200
North Town	Rye	3700000	Hayes	L-16	1300
Downtown	Brooklyn	9000000	Johnson	L-18	2000
Perryridge	Horseneck	1700000	Glenn	L-25	2500
Brighton	Brooklyn	7100000	Brooks	L-10	2200

Why is the decomposition lossy

There is one attribute in common between
Branchc-ustomer-schema and **Customer-loan-schema:**

Branch-customer-schema ∩ Customer-loan-schema = {customer-name}

The only way that we can **represent a relationship** between:
Example

loan-number and branch-name is through customer-name.

This representation is not adequate because

- A **customer may have several loans**, yet
- These loans are not necessarily obtained from the **same branch**

Alternative Design?

Branch-schema = **(branch-name**, branch-city, assets)
Loan-info-schema = **(branch-name**, customer-name, loan-number, amount)

Branch-loan-schema* ∩ *Customer-loan-schema* = *{branch-name}

For a given ***branch-name,***
there is **exactly one assets** value and **exactly one branch-city**
branch-name → assets, branch-city

In general:

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

Let r be a relation on schema R, and let $r_i = \Pi_{R_i}(r)$ for $i = 1, 2, \ldots, n$.

$$r \subseteq r_1 \bowtie r_2 \bowtie \cdots \bowtie r_n$$

In general, $r \neq r_1 \bowtie r_2 \bowtie \ldots \bowtie$

Lossless-join decomposition

A relation is **legal** if it satisfies **all rules, or constraints,** that we impose on our database.

A **lossless-join decomposition** if, for **all relations *r* on schema *R*** that are legal **under *C, set of constraints***

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

Desirable Properties of Decomposition

Lending-schema =
(*branch-name, branch-city, assets, customer-name, loan-number, amount*)

branch-name → branch-city assets
loan-number → amount branch-name

?????

Branch-schema = **(branch-name**, branch-city, assets)
Loan-schema = (**loan-number**, branch-name, amount)
Borrower-schema = (customer-name, loan-number)

Lossless-join decomposition

Let
R be a relation schema,
F be a set of functional dependencies on R.

Let
R1 and R2 form a decomposition of R.
This decomposition is a lossless-join decomposition of R
if at least one of the following functional dependencies is **in F+:**

$$R_1 \cap R_2 \rightarrow R_1$$
$$R_1 \cap R_2 \rightarrow R_2$$

Decomposition

- Decompose the relation schema *Lending-schema* into:

Branch-schema = (branch-name, branch-city,assets)

Loan-info-schema = (customer-name, loan-number, branch-name, amount)

- **All attributes** of an original schema (*R)* must appear in the decomposition (R_1, R_2)*:*

$$R = R_1 \cup R_2$$

- **Lossless-join** decomposition.
  For all possible relations *r* on schema *R*

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

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

Desirable Properties of Decomposition

Lending-schema =
(*branch-name, branch-city, assets, customer-name, loan-number, amount*)

Branch-schema = (***branch-name***, *branch-city, assets)*
Loan-info-schema = (***branch-name***, *customer-name, loan-number, amount)*

branch-name → branch-name branch-city assets

Branch-schema ∩ Loan-info-schema = {branch-name}

Desirable Properties of Decomposition

Branch-schema = (***branch-name***, *branch-city, assets)*

Loan-info-schema = (***branch-name***, *customer-name, loan-number, amount)*

Loan-schema = (***loan-numbe***r, *branch-name, amount)*
Borrower-schema = (*customer-name,* ***loan-number****)*

loan-number → loan-number amount branch-name.

Loan-schema ∩ Borrower-schema = {Loan-number}

Desirable Properties of Decomposition

Lending-schema =
(*branch-name, branch-city, assets, customer-name, loan-number, amount*)

Branch-schema = (***branch-name***, *branch-city, assets)*
Loan-info-schema = (***branch-name***, *customer-name, loan-number, amount)*

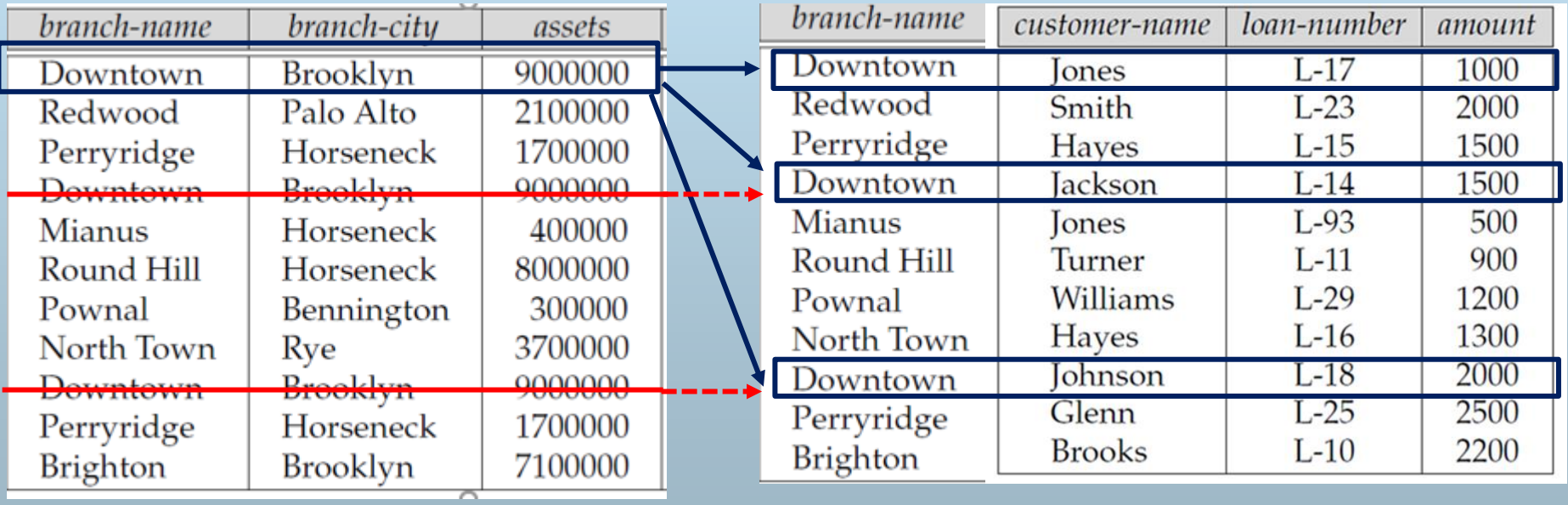
branch-name → branch-name branch-city assets

Branch-schema ∩ Loan-info-schema = {branch-name}

Branch-schema = (***branch-name***, *branch-city, assets*)
Loan-info-schema = (***branch-name***, *customer-name, loan-number, amount*)

branch-name → branch-name branch-city assets

Branch-schema ∩ Loan-info-schema = {branch-name}



branch-name	*branch-city*	*assets*
Downtown	Brooklyn	9000000
Redwood	Palo Alto	2100000
Perryridge	Horseneck	1700000
~~Downtown~~	~~Brooklyn~~	~~9000000~~
Mianus	Horseneck	400000
Round Hill	Horseneck	8000000
Pownal	Bennington	300000
North Town	Rye	3700000
~~Downtown~~	~~Brooklyn~~	~~9000000~~
Perryridge	Horseneck	1700000
Brighton	Brooklyn	7100000

branch-name	*customer-name*	*loan-number*	*amount*
Downtown	Jones	L-17	1000
Redwood	Smith	L-23	2000
Perryridge	Hayes	L-15	1500
Downtown	Jackson	L-14	1500
Mianus	Jones	L-93	500
Round Hill	Turner	L-11	900
Pownal	Williams	L-29	1200
North Town	Hayes	L-16	1300
Downtown	Johnson	L-18	2000
Perryridge	Glenn	L-25	2500
Brighton	Brooks	L-10	2200

Dependency Preservation

F = {A → B, B → C}

Let decomposed into {AC} and {AB}.

The **restriction** of F to **{AC} is then A → C**,
Since A → C is in F^+
Even though it is not in F.

Dependency Preservation

The input is a set

$D = \{R_1, R_2, \ldots, R_n\}$

decomposed relation schemas

The set of restrictions

$F_1, F_2, \ldots, F_n$

is the set of dependencies that can be checked efficiently.

Let

$F' = F_1 \cup F_2 \cup \cdots \cup F_n$.

F is a set of functional dependencies on schema R

But, **in general, $F'^{+} = F^{+}$**

(When Decomposition is Dependency Preserving)

Dependency Preservation

The input is a set D = {R_1, R_2, . . .,R_n}
Decomposed relation schemas

compute F^+;
for each schema R_i in D **do**
    begin
        F_i := the restriction of F^+ to R_i;
    end
$F' := \emptyset$
for each restriction F_i **do**
    begin
        $F' = F' \cup F_i$
    end
compute F'^+;
if ($F'^+ = F^+$) **then return** (true)
                else return (false);

Dependency Preservation

Apply the following procedure to each α → β in F.

$result = \alpha$
while (changes to *result*) **do**
  for each R_i in the decomposition
    $t = (result \cap R_i)^+ \cap R_i$
    $result = result \cup t$

- The attribute closure is with **respect to the functional dependencies in F**.
- If result **contains all attributes in β**
  Then the **functional dependency α → β is preserved**

- The **decomposition is dependency preserving**
  if and only **if all the dependencies in F are preserved.**

Dependency Preservation

Example:

F= A → B, B → C

Decomposed into R1={AB}, R2={AC}

For R2= {AC} to check dependency preservation A → C ??

α = {A}

result = {A}

result ∩ R2= {A}

{result ∩ R2}+ = {A}+

= {ACB}

{result ∩ R2}+ ∩ R2

= {ACB} ∩ {AC}

= {AC}

t= {AC}

result = result ∪ t

= {A} ∪ {AC}

= {AC}

= β

Apply the procedure to each α → β in F.

Example:

F= A → B, B → C

Decomposed into R1={AB}, R2={AC}

For R2= {AC} to check dependency preservation A → C ??

α = {A}

result = {A}

result ∩ R2= {A}

{result ∩ R2}+ = {A}+

= {ACB}

{result ∩ R2}+ ∩ R2

= {ACB} ∩ {AC}

= {AC}

t= {AC}

result = result ∪ t

= {A} ∪ {AC}

= {AC}

= β

Example: Dependency Preservation in Decomposition

Let a relation R(A,B,C,D) and set a FDs
F = { A -> B , A -> C , C -> D} are given.

A relation R is decomposed into -

R_1 = (A, B, C) with FDs F_1 = {A -> B, A -> C}
R_2 = (C, D) with FDs F_2 = {C -> D}.

F' = $F_1 \cup F_2$ = {A -> B, A -> C, C -> D}
so, F' = F.
And, $F'^{+} = F^{+}$.

Functional Dependency

Functional dependency $\alpha \rightarrow \beta$ means,
If we know the value of α,

- we can determine β
- corresponding value of β
- value of β in the that record
- Never Compute the value of β as function

For $\alpha \rightarrow \beta$ to be true

If $t1.\alpha = t2.\alpha$ Then $t1.\beta = t2.\beta$

For R={A,B,C,D}

If t1.A=t2.A Then t1.B=t2.B for **A → B**
If t1.B=t2.B Then t1.C=t2.C for **B → C**
If t1.A=t2.A Then t1.C=t2.C for **A → C**

We first check the first part, if it TRUE then we will go for 2nd part

Functional Dependency

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

If t1.A=t2.A Then t1.B=t2.B
If t1.A=t2.A Then t1.C=t2.C

There is no **duplicate** value for A,
That's means, **for each value of A,**
we can determine corresponding value of B and C

That means:
A → B
and
A → C
That means: A → BC

Functional Dependency

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

If t1.BC=t2.BC Then t1.A=t2.A

There is no **duplicate** value for BC,
That's means, for each value of BC,
we can determine corresponding value of A

That means:
BC→ A

Functional Dependency

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

If t1.B=t2.B Then t1.C=t2.C

Lets check first two records:
1=1 (for t1.B=t2.B) but **1 ≠ 2** (t1.C ≠ t2.C)

That means,
If we ask, what will **corresponding value of C for A=1**,
There are TWO corresponding **values for C, 1 and 2,** which will be returned ?

Thank means
B→ C becomes False
Same way: **B→ A, C→ B, becomes False**

Functional Dependency Preservation

A	B
1	1
2	1
3	2
4	2

R1 = [AB}
A → B True
B → A False

B	C
1	1
1	2
2	1
2	2

R1 = [BC}
B → C False
C → B False

Functional Dependency Preservation

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

R={ABC}

F= {BC→ A, A → BC}

A	B
1	1
2	1
3	2
4	2

R1 = [AB}

F1= {A → B}

B	C
1	1
1	2
2	1
2	2

R1 = [BC}

F2= { }

F´= F1 ∪ F2

= {A → B}

Therefore, F´ ≅ F Becomes False

NOT dependency preserving

Functional Dependency Preservation

R = {ABCDE}

F = {A → B, B → C, C → D, D → A}

R1 = {ABC} R2 = {CDE}

Functional Dependency Preservation

R = {ABCDE}
F = {A → B, B → C, C → D, D → A}

R1 = {ABC}
A^+ = ABCD <*delete A as Trivial, D is not in R1*>
=BC Therefore, **A → BC ……(1)**
B^+ = BCDA <*delete B as Trivial, D is not in R1*>
=CA Therefore, **B → CA ……(2)**
C^+ = CDAB <*delete C as Trivial, D is not in R1*>
=AB Therefore, **C → AB ……(3)**
AB^+ = ABCD <*delete AB as Trivial, D is not in R1*>
=C Therefore, **AB → C ……(4)**
redundant since from (1), A → B, A → C
Same way, BC^+ and CA^+ are redundant **BC → A, CA → B**
ABC^+ is Trivial
F1 = {**A → BC, B → CA, C → AB }**

R = {ABCDE}

F = {A → B, B → C, C → D, D → A}

R2 = {CDE}

C^+ = CDAB <*delete C as Trivial, AB is not in R2*>

=D Therefore, **C → D …...(5)**

D^+ = DBAC <*delete D as Trivial, AB is not in R2*>

=C Therefore, **D → C ……(6)**

E^+ = E <*delete E as Trivial*>

={ }

CD^+ = CD <*delete CD as Trivial, AB is not in R2*>

={ }

DE^+ = DEABC <*delete DE as Trivial, AB is not in R2*>

Therefore, **DE → C ……(4)**

redundant since from (6), D → C

Same way, CE^+ is redundant CE → D

CDE^+ is Trivial

F1 = {**C → D , D → C** }

Functional Dependency Preservation

R = {ABCDE} F = {**A → B, B → C, C → D, D → A**}

R1 = {ABC} F1 = {**A → BC, B → CA, C → AB** }

R2 = {CDE} F1 = {**C → D , D → C** }

F´= F1 ∪ F2
= {A → BC, B → CA, C → AB } ∪ {C → D , D → C }
= {A → B, A → C, B → C, B → A, C → A, C → B, C → D , D → C }
= {A → B, A → C, B → C, B → A, C → A, C → B, C → D , D → C }
= {A → B, A → C, B → C, B → A, C → A, C → B, C → D , D → C, D → A}

Since D+ = {DCAB} so, D → A

Therefore, F → F´ and F´ → F
As a result, the decomposition is dependency preserving

Functional Dependency

Roll	Name	Marks	Dept	Course
1	a	78	CSE	C1
2	b	60	EEE	C1
3	a	78	CSE	C2
4	b	60	EEE	C3
5	c	80	ICE	C3
6	d	80	MSE	C2

Example:

If t1.Roll=t2.Roll Then t1.Name=t2.Name for **Roll → Name**

If t1.Name=t2.Name Then t1.Marks=t2.Marks for **Name → Marks**

Roll → Name <True>

Name → Roll <False>

Roll → Marks <True>

Name, Marks → Dept. <True>

Dept → Course <False>

Roll, Name → Marks <True, but Name is Redundant>

Course → Dept <False>

3NF and BCNF

3NF

It is unable to handle overlapped candidate keys:
In such cases, it may arise redundancy.

Example cases:
R = {ABCD}
Let's consider candidate key:
A**B**, **BC**, D**C**

BCNF
Its not 4NF,
but it eliminates the above limitation of 3NF

Boyce-Codd Normal Form

A relation schema *R* is in BCNF with respect to a set *F* of functional dependencies if for **all functional dependencies** in F^+ of the form $\alpha \rightarrow \beta$, **where** $\alpha \subseteq R$ **and** $\beta \subseteq R$, **at least** one of the following holds:

- $\alpha \rightarrow \beta$ **is trivial (i.e.,** $\beta \subseteq \alpha$**)**
- α **is a superkey for** ***R***

Example

- $R = (A, B, C)$
  $F = \{A \rightarrow B$
  $B \rightarrow C\}$
  Key = $\{A\}$
- R is not in BCNF < **? all functional dependencies ? >**
- Decomposition $R_1 = (A, B)$, $R_2 = (B, C)$
  - R_1 and R_2 in BCNF
  - Lossless-join decomposition
  - Dependency preserving

Testing for BCNF

- To check if a non-trivial dependency $\alpha \rightarrow \beta$ causes a violation of BCNF
  1. **compute** α^+ (the attribute closure of α), and
  2. verify that it includes all attributes of ***R,*** that is, it is a superkey of *R*.
- Simplified test: To check if a relation schema *R* is in BCNF, it suffices to check only the dependencies in the given set *F* for violation of BCNF, rather than checking all dependencies in F^+.
  - If none of the dependencies in *F* causes a violation of BCNF, then none of the dependencies in F^+ will cause a violation of BCNF either.

Testing for BCNF

- However, using only **F is incorrect** when testing a relation in a decomposition of R
  - E.g. Consider R (A, B, C, D), with F = { $A \rightarrow B$, $B \rightarrow C$}
    - Decompose R into $R_1(A,B)$ and $R_2(A,C,D)$
    - Neither of the dependencies in F contain only attributes from (A,C,D) so we might be mislead into thinking R_2 satisfies BCNF.
    - In fact, dependency $\boldsymbol{A \rightarrow C}$ **in** $\boldsymbol{F^+}$ **shows** $\boldsymbol{R_2}$ is not in BCNF.

BCNF

R = {ABC} F = {**A → B, B → C, C → A**}

A^+ = {ABC}
B^+ = {ABC}
C^+ = {ABC}

So, A,B,C **All are** the **Super keys**
Since, there is no **proper subset** of those keys,
So all those are **Candidate keys** as well

BCNF

R = {ABCDE} F = {**A → BCDE, BC → ACE, D → E}**

A^+ = {ABCDE}

So, A is **Super keys** and **Candidate** keys

BC → ACE

BC → A, BC → C, BC → E

BC is super key (since RHS A is a PA)

OR

BC^+ = {ABCED} since A → D

Proper subset of BC, {B}, {C}

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

BC is a Candidate key too

D^+ = {DE}, D is NOT Super keys

BCNF

R = {ABCDE} F = {**AB → CDE, D → A**}

AB+ = {ABCDE}

AB is super key

Proper subset of AB, {A}, {B}

A^+ = {A }, B^+ = {B }

AB is a Candidate key too

D+ = {DA}

D is NOT a super key

To have lossless-join decompositions

Needs to replace a schema R_i with $(R_i - \beta)$ and (α, β),
the dependency $\alpha \rightarrow \beta$ holds,
and $(R_i - \beta) \cap (\alpha, \beta) = \alpha$.

BCNF Decomposition Algorithm

result := {*R*};
done := false;
compute F^+;
while (not *done)* **do**
  if (there is a schema R_i in *result* that is not in BCNF)
    then begin
      let $\alpha \rightarrow \beta$ be a nontrivial functional
        dependency that holds on R_i
        such that $\alpha \rightarrow R_i$ is not in F^+,
        and $\alpha \cap \beta = \varnothing$;
      result := (*result* – R_i) $\cup$ (R_i – β) $\cup$ (α, β);
    end
  else *done* := **true;**

Example of BCNF Decomposition

Customer-schema =
(customer-name, customer-street, customer-city)
customer-name → customer-street customer-city

A candidate key for the schema is customer-name.

Branch-schema =
{branch-name, assets, branch-city}
branch-name → assets branch-city

Example of BCNF Decomposition

Loan-info-schema =
(branch-name, **customer-name**, loan-number, amount)
loan-number → amount branch-name
! is not in BCNF !

(Downtown, **John Bell**, L-44, 1000)
(Downtown, **Jane Bell**, L-44, 1000)

Loan-schema =
(loan-number, branch-name, amount)
Borrower-schema =
(customer-name, loan-number)

The *borrower* relation

customer_name	*loan_number*
Adams	L-16
Curry	L-93
Hayes	L-15
Jackson	L-14
Jones	L-17
Smith	L-11
Smith	L-23
Williams	L-17

The *loan* relation

loan_number	*branch_name*	*amount*
L-11	Round Hill	900
L-14	Downtown	1500
L-15	Perryridge	1500
L-16	Perryridge	1300
L-17	Downtown	1000
L-23	Redwood	2000
L-93	Mianus	500

Is it ossless-join decomposition ???

Example of BCNF Decomposition

Loan-schema = {loan-number, branch-name, amount}
Borrower-schema = {customer-name, loan-number}

The *borrower* relation

customer_name	*loan_number*
Adams	L-16
Curry	L-93
Hayes	L-15
Jackson	L-14
Jones	L-17
Smith	L-11
Smith	L-23
Williams	L-17

The *loan* relation

loan_number	*branch_name*	*amount*
L-11	Round Hill	900
L-14	Downtown	1500
L-15	Perryridge	1500
L-16	Perryridge	1300
L-17	Downtown	1000
L-23	Redwood	2000
L-93	Mianus	500

Only trivial functional dependencies apply to

$\alpha \rightarrow \beta$ **is trivial (i.e.,** $\beta \subseteq \alpha$**)**

loan-number $\rightarrow$ amount
loan-number $\rightarrow$ branch-name

Example of BCNF Decomposition

Often testing of a relation to see
if it satisfies BCNF can be **simplified:**

To check if a **nontrivial** dependency
$\alpha \rightarrow \beta$ causes a violation of BCNF,

Compute α^+ and verify that it includes all attributes of R;
That is, it is a superkey of R.

Example of BCNF Decomposition

Often testing of a relation to see if
it satisfies BCNF can be **simplified:**

• To check if a relation schema R is in BCNF,
it suffices to check only the dependencies
in the given set F for violation of BCNF,

Rather than check all dependencies in F+.

Example of BCNF Decomposition

Lending-schema =
(branch-name, branch-city, assets, customer-name, loan-number, amount)

branch-name → assets branch-city
loan-number → amount branch-name

candidate key
{loan-number, customer-name}.

Normalization Using Functional Dependencies

- When we decompose a relation schema R with a set of functional dependencies F into R_1, R_2,.., R_n we want
  - **Lossless-join decomposition:** Otherwise decomposition would result in information loss.
  - **No redundancy**: The relations R_i preferably should be in either Boyce-Codd Normal Form or Third Normal Form.
  - **Dependency preservation**: Let F_i be the set of dependencies F^+ that include only attributes in R_i.
    - Preferably the decomposition should be dependency preserving, that is, $(F_1 \cup F_2 \cup \ldots \cup F_n)^+ = F^+$
    - Otherwise, checking updates for violation of functional dependencies may require computing joins, which is expensive.

Example

- $R = (A, B, C)$
  $F = \{A \rightarrow B, B \rightarrow C\}$
  - Can be decomposed in two different ways
- $R_1 = (A, B)$, $R_2 = (B, C)$
  - **Lossless-join** decomposition:

    $R_1 \cap R_2 = \{B\}$ and $B \rightarrow BC$
  - Dependency preserving
- $R_1 = (A, B)$, $R_2 = (A, C)$
  - Lossless-join decomposition:

    $R_1 \cap R_2 = \{A\}$ and $A \rightarrow AB$
  - **Not dependency preserving**
    (cannot **check** $B \rightarrow C$ without computing $R_1 \bowtie R_2$)