Relational Database Design

- Goals of Relational Database Design
- Functional Dependencies
- Loss-less Joins
- Dependency Preservation
- Normal Forms (1st, 2nd, 3rd, BCNF)

Goals of Relational Database Design

- Traditional Design Goals:
  - Avoid redundant data – generally considered enemy #1.
  - Ensure that relationships among attributes are represented.
  - Facilitate the checking of updates for violation of integrity constraints.
- We will formalize these goals in several steps.
- What about performance, reliability and security?

Florida Institute of Technology

The Database Design Process

- Database design is driven by *normalization*.

- If relational scheme R is not sufficiently normalized, *decompose* it into a set of relational schemes $\{R_1, R_2, ..., R_n\}$ such that:
  - Each relational scheme is sufficiently *normalized*.
  - The decomposition has a *lossless-join*.
  - All functional dependencies are *preserved*.

- So what are normalization, lossless-join, functional dependencies, and what does preserving them mean?

First Normal Form

- A domain is ***atomic*** if its elements are *treated* as indivisible units.
  - Examples of atomic domains:
    - Number of pets
    - Gender
  - Examples of non-atomic domains:
    - Person names
    - List of dependent first names
    - Identification numbers like CS101 that can be broken into parts
- A relational schema R is in ***first normal form*** if all attributes of R are atomic (or rather, are treated atomically).

The Problem with Redundancy

- So why is redundancy considered “enemy #1?”

- Consider the relation 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

- Note the redundancy in *branch-name, branch-city,* and *assets*.
  - Wastes space.
  - Creates insertion, deletion, and update anomalies.

Update, Insertion and Deletion Anomalies

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

- Insertion Anomalies:
  - Cannot store information about a branch if no loans exist without using null values; this is particularly bad since loan-number is part of the primary key.
  - Subsequent insertion of a loan for that same branch would require the first tuple to be deleted.
- Deletion Anomalies:
  - Deleting L-17 and L-14 might result in all Downtown branch information being deleted.
- Update Anomalies:
  - Modify the asset value for the branch of loan L-17.
  - Add $100 to the balance of all loans at a Brooklyn branch.

Florida Institute of Technology

Decomposition

- Solution - decompose *Lending-schema* into:

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

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

- For any decomposition:
  - All attributes of an original schema must appear in the decomposition*:*

    $$R = R_1 \cup R_2$$

  - The decomposition must have a *lossless-join*, i.e., for all possible relations *r* on schema *R:*

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

Example of Lossy-Join Decomposition

- Decomposition of $R = (A, B)$ into $R_1 = (A)$ and $R_2 = (B)$

r

A	*B*
α	1
α	2
β	1

$\Pi_A(r)$

A
α
β

$\Pi_B(r)$

B
1
2

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

A	*B*
α	1
α	2
β	1
β	2

Functional Dependencies

- Informally, a *Functional Dependency (FD)* is a constraint on the contents of a relation.

- An FD specifies that the values for one set of attributes determines the values for another set of attributes.

- The notion of an FD is a generalization of the notion of a *key* (super, candidate, primary or unique).
  - *In fact, in a "good" design, most FDs are realized as keys.*

Functional Dependencies – Example #1

- Consider the following schema:

Loan-info-schema = (<u>customer-name</u>, <u>loan-number</u>, branch-name, amount)

- Applicable FDs:

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

- Non-applicable FDs:

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

  customer-name $\rightarrow$ *amount*

Functional Dependencies – Example #2

- Consider the following schema:

  Grade-schema = (<u>student-id</u>, name, <u>course-id</u>, grade)

- Applicable FDs:

  student-id $\rightarrow$ *name*

  student-id, course-id $\rightarrow$ *grade*

- Non-applicable FDs:

  student-id $\rightarrow$ *grade*

  grade $\rightarrow$ *course-id*

- Exercise – list out all possible FDs for the above relational scheme, and determine which ones hold and which ones don't (same for the one on the previous page).

Functional Dependency - Formal Definition

- Let *R* be a relation schema where $\alpha \subseteq R$ *and* $\beta \subseteq R$.

- The functional dependency

  $$\alpha \rightarrow \beta$$

  is said to hold 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, i.e.,

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

- Alternatively, if $\alpha \rightarrow \beta$ then the relation *r* can never contain two tuples that agree on α and disagree on β.

Use of Functional Dependencies

- Let R be a relational scheme and let F be an associated set of functional dependencies.
- *F* holds on *R* if all legal relations on *R* satisfy the set of functional dependencies *F*
  - *F* is *imposed* or *enforced* on *R*.
- If a relation *r* is legal for a set *F* of functional dependencies, we say that *r* satisfies *F*
  - *F* is currently satisfied but may or may not be *imposed* or *enforced* on r.
- Note that the difference between the two is important!
  - If *F* holds on relation *R*, then every relation (i.e., a set of tuples) must satisfy *F*.
  - If a relation satisfies *F*, it may or may not be the case that *F* holds on *R*.

Florida Institute of Technology

Example – FD Formal Definition

- An analogy - assume for the moment that all drivers actually follow speed limits...
- Thus we say that the speed limit established for a road holds on that road.
- You will never see a car exceed whatever the speed limit happens to be.

Example – FD Formal Definition

- Suppose you are watching cars drive by on a road where you don't know what the speed limit is.

- A some point in time, there might be 3 cars on the road, one going 45, another going 30, and another going 42.
  - These do not *satisfy* a speed limit of 25, 10, etc.
  - We can conclude, therefore, that the speed limit is not 25.
  - They do, however, satisfy a speed limit of 55, 60, 45, etc.
  - We cannot conclude however, that, for example, 55 is the speed limit, just by looking at the cards.
  - Speed limit could be 45, 46, 47, 90, etc.

- If a particular speed limit <u>holds</u> on a road, then the speed of all cars on that road <u>satisfy</u> the speed limit.
  - Cars are like rows in a table
  - FDs that hold are like speed limits

Example – FD Formal Definition

- Consider the following relation:

A	B
1	4
1	5
3	7

- For this set of tuples:
  - $A \rightarrow B$ is **NOT** satisfied
  - $A \rightarrow B$ therefore does **NOT** hold
  - $B \rightarrow A$ **IS** currently satisfied
  - but does $B \rightarrow A$ hold?

Florida Institute of Technology

Example – FD Formal Definition

- By simply looking at the tuples in a relation, one can determine if an FD is currently satisfied or not.
- Similarly, by looking at the tuples you can determine that an FD doesn't hold, but you can never be certain that an FD does hold (for that you need to look at the set of FDs).
- Similarly by simply looking at the cars on a road, one can determine if a speed limit is currently satisfied or not.
- Similarly, by looking at the cars you can determine that a speed limit doesn't hold, but you can never be certain that a speed limit does hold (for that you need to look at the speed limit sign).

FDs – Holding vs. Satisfying

- One more time - a specific relation may satisfy an FD even if the FD does not hold on all legal instances of that relation.
- Example #1: A specific instance of *Loan-info-schema* may satisfy:

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

- Example #2: A specific instance of *Grade-schema* may satisfy:

  course-id $\rightarrow$ *grade*

- Although either of the above might satisfy the specified FD, in neither case does the FD hold.
- Example #3: Suppose an instance of *Loan-info-schema* (or *Grade-schema*) is empty. What FDs does it satisfy?

Defining Keys in Terms of FDs

- The notions of a superkey and a candidate key can be defined in terms of functional dependencies.
- K is a superkey for relation schema R if and only if $K \rightarrow R$
- K is a candidate key for R if and only if
  - K is a superkey for R, and
  - There is no set $\alpha \subset K$ such that α is a superkey.
- Note how declaring K as the primary key of the table effectively enforces the functional dependency $K \rightarrow R$

Functional Dependencies (Cont.)

- A functional dependency $\alpha \rightarrow \beta$ is said to be *trivial* if $\beta \subseteq \alpha$
- Examples:

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

  customer-name $\rightarrow$ *customer-name*

- Trivial functional dependencies are always satisfied (by every instance of a relation).

Armstrong’s Axioms

- Given a set F of FDs, there are other FDs that are logically implied by F.
- For example, if $A \rightarrow B$ and $B \rightarrow C$, then $A \rightarrow C$.
- Example:

  ID# $\rightarrow$ *Date-of-Birth*

  Date-of-Birth $\rightarrow$ *Zodiac-Sign*

  $\therefore$ *ID#* $\rightarrow$ *Zodiac-Sign*

- But there are other rules…

Armstrong's Axioms

- 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)**

- Armstrong's axioms are *sound, complete* and *minimal*:
  - Sound – generate only functional dependencies that actually hold.
  - Complete – generate all functional dependencies that hold.
  - Minimal – no proper subset of the Axioms is complete.

Closure of a Set of FDs

- The set of all FDs logically implied by *F* is called the *closure* of *F*.
- The closure of *F* is denoted by F^+.
- Given a set *F*, we can find all FDs in F^+ by applying Armstrong's Axioms

Closure Example

- Consider the following:

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

- Some members of F^+
  - $A \rightarrow H$

    Transitivity from $A \rightarrow B$ *and* $B \rightarrow H$
  - $AG \rightarrow I$

    Augmentation of $A \rightarrow C$ with G, to get $AG \rightarrow CG$,
    then transitivity with $CG \rightarrow I$
  - $CG \rightarrow HI$

    Augmentation of $CG \rightarrow I$ to get $CG \rightarrow CGI$,
    augmentation of $CG \rightarrow H$ to get $CGI \rightarrow HI$,
    and then transitivity

Florida Institute of Technology

Closure Example

- Note that a formal derivation (proof) can be given for each FD in F^+.
- Example: Show that $CG \rightarrow HI$ *is in* $F^{+:}$

1.	$CG \rightarrow I$	*Given*
2.	$CG \rightarrow CGI$	*Augmentation of (1) with CG*
3.	$CG \rightarrow H$	*Given*
4.	$CGI \rightarrow HI$	*Augmentation of (3) with I*
5.	$CG \rightarrow HI$	*Transitivity with (2) and (4)*

- Exercises:
  - Suppose $A \rightarrow B$ *and* $A \rightarrow C$*. Show* $A \rightarrow BC$.
  - Suppose $A \rightarrow BC$ *then* $A \rightarrow B$ *and* $A \rightarrow C$.
- By the way, what is the difference between $CG \rightarrow I$, $GC \rightarrow I$ *and* $CGC \rightarrow I$?

Procedure for Computing F^+

- To compute the closure of a set *F* of FDs (modified from the book):

```
F+ = F;
add all trivial functional dependencies to F+;
repeat
      for each functional dependency f in F+
         apply augmentation rules on f
         add the resulting functional dependencies to F+
      for each pair of functional dependencies f1 and f2 in F+
         if f1 and f2 can be combined using transitivity
            then add the resulting functional dependency to F+
until F+ does not change any further;
```

Worst case time is exponential!
Consider F = {A→*B1*, A→*B2*,…,A→*Bn*}

- We will see an alternative procedure for this task later.

Florida Institute of Technology

Additional FD Rules

- The following additional rules will occasionally be helpful:
  - $\alpha \rightarrow \beta$ and $\alpha \rightarrow \gamma$ implies $\alpha \rightarrow \beta\gamma$ **(union)**
  - $\alpha \rightarrow \beta\gamma$ implies $\alpha \rightarrow \beta$ and $\alpha \rightarrow \gamma$ **(decomposition)**
  - $\alpha \rightarrow \beta$ and $\gamma\beta \rightarrow \delta$ implies $\alpha\gamma \rightarrow \delta$ **(pseudotransitivity)**
- Notes:
  - The above rules are **NOT** Armstrong's axioms.
  - The above rules can be proven using Armstrong's axioms.

Proving the Decomposition Rule

- Example - Proving the decomposition rule.
- Suppose $\alpha \rightarrow \beta\gamma$. Show that $\alpha \rightarrow \beta$ and $\alpha \rightarrow \gamma$.

1. $\alpha \rightarrow \beta\gamma$ *Given*
2. $\beta\gamma \rightarrow \beta$ *Reflexivity*
3. $\alpha \rightarrow \beta$ *Transitivity with (1) and (2)*
4. $\beta\gamma \rightarrow \gamma$ *Reflexivity*
5. $\alpha \rightarrow \gamma$ *Transitivity with (1) and (4)*

- Exercise: prove the union rule and the pseudo-transitivity rule.

Closure of Attribute Sets

- Let α be a set of attributes, and let *F* be a set of functional dependencies.
- The *closure* of α *under* *F* (denoted by α^+) is the set of attributes that are functionally determined by α under *F.*
- Closure of a set of attributes α^+ is NOT the same as the closure of a set of FDs F^+.

Example of Attribute Set Closure

- Consider the following:

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

- Compute $\{AG\}^+$

AG	
ABG	$A \rightarrow B$
$ABCG$	$A \rightarrow C$
$ABCGH$	$CG \rightarrow H$
$ABCGHI$	$CG \rightarrow I$

Closure of Attribute Sets

- Algorithm to compute α^+

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

Uses of Attribute Closure

There are several uses of the attribute closure algorithm:

- Testing if a functional dependency $\alpha \rightarrow \beta$ holds, i.e., is it in F^+ :
  - Check if $\beta \subseteq \alpha^+$
  - Is AG $\rightarrow$ I in F^+ for the previous example?
- Testing if a set of attributes α is a superkey:
  - Check if $\alpha^+ = R$
- Testing if a set of attributes α is a candidate key:
  - Check if α^+ is a superkey (using the above)
  - Check if has a subset $\alpha' \subset \alpha$ that is a superkey (using the above)

Uses of Attribute Closure

- Computing closure of a set *F* of functional dependencies:
  - for each $\gamma \subseteq R$, we find the closure γ^+, and then
  - for each $S \subseteq \gamma^+$, we output a functional dependency $\gamma \rightarrow S$

- How helpful is that?

Example of Attribute Set Closure

- Is *AG* a candidate key for the preceding relational scheme?

  1. Is AG a super key?
     - Does $AG \rightarrow R$, i.e., is $R \subseteq \{AG\}^+$

  2. Is any subset of AG a super key?
     - Does $A \rightarrow R$, i.e., is $R \subseteq \{A\}^+$
     - Does $G \rightarrow R$, i.e., is $R \subseteq \{G\}^+$

- IS *CG* a candidate key?

Equivalent Sets of FDs

- Let F_1 and F_2 be two sets of functional dependencies.

- F_1 and F_2 are said to be *equal* (or identical), denoted $F_1 = F_2$, if:
  - $F_1 \subseteq F_2$ and
  - $F_2 \subseteq F_1$

- The above definition is not particularly helpful; it merely states the obvious…

Equivalent Sets of FDs

- F_2 is said to ***imply*** F_1 if $F_1 \subseteq F_2^+$

- F_1 and F_2 are said to be ***equivalent*** , denoted $F_1 \approx F_2$, if F_1 implies F_2 and F_2 implies F_1, i.e.,
  - $F_2 \subseteq F_1^+$
  - $F_1 \subseteq F_2^+$

- What does the above definition suggest?

Florida Institute of Technology

Equivalent Sets of FDs

- Consider the following sets of FDs:

  $F_1 = \{A \rightarrow B, B \rightarrow C, AB \rightarrow C\}$

  $F_2 = \{A \rightarrow B, B \rightarrow C, A \rightarrow C\}$

- Clearly, F_1 and F_2 are not equal.
- However, F_1 is implied by F_2 since $F_1 \subseteq F_2^+$
- And F_2 is implied by F_1 since $F_2 \subseteq F_1^+$
- Hence, F_1 and F_2 are <u>equivalent</u>, i.e., $F_1 \approx F_2$.

Equivalent Sets of FDs

- Consider the following sets of FDs:

  $F_1 = \{A \rightarrow B, CG \rightarrow I, B \rightarrow H, A \rightarrow H\}$

  $F_2 = \{A \rightarrow B, CG \rightarrow H, A \rightarrow C, CG \rightarrow I, B \rightarrow H\}$

- Clearly, F_1 and F_2 are not equal.

- However, F_1 is implied by F_2 since $F_1 \subseteq F_2^+$

- But, F_2 is not implied by F_1 since $F_2 \not\subseteq F_1^+$

- Hence, F_1 and F_2 are <u>not</u> equivalent.

Canonical Cover

- A set of FDs may contain redundancies.
- Sometimes an entire FD is redundant:

  $A \rightarrow C$ is redundant in $\{A \rightarrow B, \quad B \rightarrow C, \quad A \rightarrow C\}$

- How can we test if an FD is redundant?

Canonical Cover

- Other times, an attribute in an FD may be redundant:

  $\{A \rightarrow B,\ \ B \rightarrow C,\ \ A \rightarrow CD\}$ can be simplified to

  $\{A \rightarrow B,\ \ B \rightarrow C,\ \ A \rightarrow D\}$

  $\{A \rightarrow B,\ \ B \rightarrow C,\ \ AC \rightarrow D\}$ can be simplified to

  $\{A \rightarrow B,\ \ B \rightarrow C,\ \ A \rightarrow D\}$

- How can we test if an attribute in an FD is redundant?

Extraneous Attributes

- Let F be a set of FDs and suppose that $\alpha \rightarrow \beta$ is in F.
  - Attribute A is *extraneous* in α if $A \in \alpha$ and F logically implies $(F - \{\alpha \rightarrow \beta\}) \cup \{(\alpha - A) \rightarrow \beta\}$.
  - 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.

- Note that implication in the opposite direction is trivial in each of the above cases.

Examples of Extraneous Attributes

- Example #1:

  $F = \{A \rightarrow C, AB \rightarrow C\}$

  Is B is extraneous in $AB \rightarrow C$?

  Is A is extraneous in $AB \rightarrow C$?

- Example #2:

  $F = \{A \rightarrow C, AB \rightarrow CD\}$

  Is C is extraneous in $AB \rightarrow CD$?

  How about A, B or D?

Canonical Cover – Formal Definition

- Intuitively, a canonical cover for *F* is a "minimal" set that is equivalent to *F*, i.e., having no redundant FDs, or FDs with redundant attributes.

- More formally, a *canonical cover* for *F* is a set of dependencies F_c such that:
  - $F \approx F_c$
  - No functional dependency in F_c contains an extraneous attribute.
  - Each left side of a functional dependency in F_c is unique.

Computing a Canonical Cover

- Given a set *F* of FDs, a canonical cover for *F* can be computed as follows:

repeat
    Replace any dependencies of the form $\alpha_1 \rightarrow \beta_1$ and $\alpha_1 \rightarrow \beta_2$ with $\alpha_1 \rightarrow \beta_1 \beta_2$; // union rule
    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 that the 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

$R = (A, B, C)$
$F = \{A \rightarrow BC$
$B \rightarrow C$
$A \rightarrow B$
$AB \rightarrow C\}$

- Combining $A \rightarrow BC$ and $A \rightarrow B$ gives $\{A \rightarrow BC, B \rightarrow C, AB \rightarrow C\}$
- A is extraneous in $AB \rightarrow C$ gives $\{A \rightarrow BC, B \rightarrow C\}$
- C is extraneous in $A \rightarrow BC$ gives $\{A \rightarrow B, B \rightarrow C\}$

Review – The Goals of Normalization

Recall:

- Given a relational scheme *R* and an associated set *F* of FDs, first determine whether or not *R* is sufficiently normalized.

- If *R* is not sufficiently normalized, decompose it into a set of relations $\{R_1, R_2, \ldots, R_n\}$ such that
  - Each relation is sufficiently normalized
  - The decomposition is a lossless-join decomposition
  - All dependencies are preserved

- All of the above requirements will be based on functional dependencies.

Decomposition

- Previously, we decomposed the *Lending-schema* into:

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

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

- The decomposition must have a lossless-join, i.e., for all possible relations *r* on *R:*

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

- Having defined FDs, we can now define the conditions under which a decomposition has a loss-less join…

Decomposition

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

- In other words:
  - R_1 *and* R_2 must have at least one attribute in common, and
  - The common attributes must be a super-key for either R_1 or R_2.

Example

- $R = (A, B, C)$
  $F = \{A \rightarrow B, B \rightarrow C)$
  - Can be decomposed in three different ways (with a common attribute).
- $R_1 = (A, B), \quad R_2 = (B, C)$
  - Has a lossless-join:

    $R_1 \cap R_2 = \{B\}$ and $B \rightarrow BC$
- $R_1 = (A, B), \quad R_2 = (A, C)$
  - Has a lossless-join:

    $R_1 \cap R_2 = \{A\}$ and $A \rightarrow AB$
- $R_1 = (A, C), \quad R_2 = (B, C)$
  - Does not have a lossless-join.

Preservation of Functional Dependencies

- Suppose that:
  - R is a relational scheme
  - F is an associated set of functional dependencies
  - $\{R_1, R_2, ..., R_n\}$ is a decomposition of R
  - Let F_i be the set of dependencies F^+ that include only attributes in R_i.
- The decomposition $\{R_1, R_2, ..., R_n\}$ is said to be *dependency preserving* if

  $(F_1 \cup F_2 \cup \ldots \cup F_n)^+ = F^+$
- Why is it important for a decomposition to preserve dependencies?
  - The goal is to replace R by $R_1, R_2, ..., R_n$
  - Enforcing $F_1, F_2, \ldots, F_n$ on $R_1, R_2, ..., R_n$ must be equivalent to enforcing F on R.

Preservation of Functional Dependencies

- Food for thought - what is the difference between each of the following?

$(F_1 \cup F_2 \cup \ldots \cup F_n)^+ = F^+$

$F \subseteq (F_1 \cup F_2 \cup \ldots \cup F_n)^+$ *technically, this is all we need!*

$F_1 \cup F_2 \cup \ldots \cup F_n = F$ *very strict definition of preservation*

$(F_1 \cup F_2 \cup \ldots \cup F_n)^+ = F$ *gets the job done, but unrealistic*

$F_1 \cup F_2 \cup \ldots \cup F_n = F^+$ *gets the job done, but also unrealistic*

- Any of the above would work, but the first is the most flexible and realistic.
- All of the last three imply the first.
- Technically, we will subscribed to the first (but <u>informally</u>, we will use the second).

Florida Institute of Technology

Example

- $R = (A, B, C)$
  $F = \{A \rightarrow B, B \rightarrow C)$
  - Can be decomposed in three different ways.
- $R_1 = (A, B), \quad R_2 = (B, C)$
  - Lossless-join decomposition (as noted previously)
  - Dependency preserving
- $R_1 = (A, B), \quad R_2 = (A, C)$
  - Lossless-join decomposition (as noted previously)
  - Not dependency preserving; $B \rightarrow C$ is not preserved
- $R_1 = (A, C), \quad R_2 = (B, C)$
  - Does not have a lossless-join (as noted previously)
  - Not dependency preserving; $A \rightarrow B$ is not preserved

Boyce-Codd Normal Form

A relational scheme *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*

Testing for BCNF

- To determine if a relational scheme is in BCNF:

  Calculate F^+

  For each non-trivial functional dependency $\alpha \rightarrow \beta$ in F^+

  1. compute α^+ (the attribute closure of α)
  2. verify that α^+ includes all attributes of *R*, i.e., that it is a superkey for *R*

=> If a functional dependency $\alpha \rightarrow \beta$ in F^+ is identified that (1) is non-trivial and (2) where α is *not* a superkey, then R is not in BCNF.

Example

- $R = (A, B, C)$
  $F = \{A \rightarrow B, B \rightarrow C\}$
  Candidate Key = $\{A\}$

- R is not in BCNF *(why not?)*

- Decompose R into $R_1 = (A, B)$ and $R_2 = (B, C)$
  - R_1 is in BCNF
  - R_2 is in BCNF
  - The decomposition has a lossless-join (noted previously)
  - The decomposition preserves dependencies (noted previously)

Testing for BCNF

- It turns out to be only necessary to check the dependencies in *F* (and not F^+).
- This leads to the following simpler definition for BCNF.

Let *R* be a relational scheme and let *F* be a set of functional dependences. Then *R* is said to be in BCNF with respect to *F* if, for each $\alpha \rightarrow \beta$ in *F,* either $\alpha \rightarrow \beta$ is trivial or α is a superkey for *R.*

Why the authors don't define it this way is…anybodies' guess…

Testing for BCNF

- Note that when testing a relation R_i in a decomposition for BCNF, however, make sure you consider ALL dependencies in F_i.

- For example, consider $R = (A, B, C, D)$, with $F = \{A \rightarrow B, B \rightarrow C\}$
  - Decompose R into $R_1(A,B)$ and $R_2(A,C,D)$
  - One might think F_2 is empty, and hence R_2 satisfies BCNF.
  - In fact, $A \rightarrow C$ is in F^+, and hence in F_2, which shows R_2 is not in BCNF.

BCNF Decomposition Algorithm

- Let *R* be a relational scheme, let *F* be an associated set of functional dependencies, and suppose that *R* is not in BCNF.

- The following will give a decomposition of R into R_1, R_2, ..., R_n such that each R_i is in BCNF, and such that the decomposition has a lossless-join.

```
result := {R};
compute F+;
while (there is a schema Ri in result that is not in BCNF) do
   let α → β be a nontrivial functional dependency that
   holds on Ri such that α → Ri is not in F+, and α ∩ β  = ∅;
   result := (result – Ri ) ∪ (Ri – β) ∪ (α, β );
end;
```

Example of BCNF Decomposition

- Consider the following Relational Scheme, which is not in BCNF (why?):

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

F = {branch-name → assets, branch-city
loan-number → amount, branch-name}

Candidate Key = {loan-number, customer-name}

- Decomposition:

~~*R = (branch-name, branch-city, assets, customer-name, loan-number, amount)*~~

R1 = (branch-name, branch-city, assets)

~~*R2 = (branch-name, customer-name, loan-number, amount)*~~

R3 = (branch-name, loan-number, amount)

R4 = (customer-name, loan-number)

Keys Created by the BCNF Algorithm

- What are the primary keys for the resulting relations?
- Ideally, each R_i represents one functional dependency, where the LHS will be the primary key, i.e., α ; thus the primary key constraint enforces the FD.
- Although this enforces the majority of FDs, it does not enforce all FDs, in general.
- In such cases the other FDs can frequently be enforced by a secondary key; in the worst case, code must be written to repeatedly check for FD violations.

Keys Created by the BCNF Algorithm

- Example:

$R = (A,B,C)$

$F = \{ A \rightarrow C,$
$B \rightarrow C,$
$A \rightarrow B,$
$B \rightarrow A\}$

Two Candidate Keys = $\{A\}$ $\{B\}$

Primary Key - A

Secondary (unique) Key - B

BCNF and Dependency Preservation

- As noted, the algorithm produces a set of BCNF relational schemes that have a lossless join, but what about preserving dependencies?

- It is not always possible to get a BCNF decomposition that is dependency preserving:

  $R = (J, K, L)$
  $F = \{JK \rightarrow L, L \rightarrow K\}$
  Two candidate keys = *JK* and *JL*

- In terms of the banking enterprise:
  Banker-schema = (<u>branch-name</u>, <u>customer-name</u>, banker-name)
  banker-name $\rightarrow$ *branch name*
  customer-name, branch name $\rightarrow$ *banker-name*

- *R* is not in BCNF *(why?)*

BCNF and Dependency Preservation

- However, any decomposition of R will fail to preserve $JK \rightarrow L$.

  $R = (J, K, L)$
  $F = \{JK \rightarrow L, L \rightarrow K\}$
  Two candidate keys = JK and JL

- Decompositions:

JK	*KL*	*J*	*KL*
JK	*JL*	*K*	*JL*
JL	*KL*	*L*	*JK*

- In every case $JK \rightarrow L$ is lost.

Third Normal Form Motivation

- It follows that there is *no* algorithm for decomposing a relational scheme that guarantees both, i.e., BCNF and preservation of dependencies.

- Solution - Define a weaker normal form, called *Third Normal Form*.
  - Allows some redundancy (with resultant problems; as we shall see)

- *Given any relational scheme, there is always a lossless-join, dependency-preserving decomposition into 3NF relational schemes.*

- This is why 3NF is industry standard.

Third Normal Form

- A relation schema *R* is in third normal form (3NF) 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., $\beta \in \alpha$)
  - α is a superkey for *R*
  - Each attribute *A* in $\beta - \alpha$ is contained in a candidate key for *R*.
- For the last condition, each attribute may be in a different candidate key.
- The third condition is a minimal relaxation of BCNF that will ensure dependency preservation.
- If a relation is in BCNF it is in 3NF *(why?)*

Testing for 3NF

- As with BCNF, the definition can be simplified to only consider FD's in F.
- The 3NF test is a slight modification of the BCNF test.
- If $\alpha \rightarrow \beta$ is not trivial, and if α is not a superkey, we have to verify if each attribute in β is contained in a candidate key of *R.*
  - Expensive - requires finding all candidate keys.
  - Testing for 3NF has been shown to be NP-hard, i.e., likely requires exponential time.
  - Ironically, decomposition into third normal form (described shortly) can be done in polynomial time.

BCNF vs. 3NF

- Note that our previous "problematic" scheme is in 3NF but not BCNF:

  $R = (J, K, L)$
  $F = \{JK \rightarrow L, L \rightarrow K\}$

  Two candidate keys = *JK* and *JL*

3NF Decomposition Algorithm

- 3NF Decomposition Algorithm:

```
Let Fc be a canonical cover for F;
i := 0;
for (each  functional dependency α → β in Fc) loop
    if (none of the schemas Rj, 1 ≤ j  ≤ i contains α and β) then
        i := i  + 1;
        Ri  := (α,β);
    end if;
end loop;
if (none of the schemas Rj, 1≤j≤i contains a candidate key for R) then
    i := i  + 1;
    Ri := any candidate key for R;
end if;
return (R1, R2, ..., Ri);
```

- Each resulting R_i is in 3NF, the decomposition has a lossless-join, and all dependencies are preserved.

- Each resulting R_i represents one or more functional dependencies, *one* of which will be enforced by a primary key.

Example

- Relation schema *R*:

  Banker-schema = (branch-name, customer-name,banker-name, office-number)

- Functional dependencies *F*:

  banker-name $\rightarrow$ *branch-name, office-number*
  customer-name, branch-name $\rightarrow$ *banker-name*

- Candidate keys:

  {customer-name, branch-name}

  {customer-name, banker-name}

- *R* is not in 3NF *(why?)*

- The algorithm creates the following schemas (*F* is already a canonical cover):

  Banker-office-schema = (<u>banker-name</u>, branch-name, office-number)
  Banker-schema = (<u>customer-name</u>, <u>branch-name</u>, banker-name)

Summary: Comparison of BCNF and 3NF

- In summary…

- It is always possible to decompose a relational scheme into a set of relational schemes such that:
  - All resulting relational schemes are in 3NF
  - The decomposition has a lossless join
  - All dependencies are preserved

- It is always possible to decompose a relational scheme into a set of relational schemes such that:
  - All resulting relational schemes are in BCNF
  - The decomposition has a lossless join

=> The decomposition, however, is not guaranteed to preserve dependencies.

Summary: Comparison of BCNF and 3NF

- Now for some final notes…

3NF (Cont.)

Note #1:

- So how does 3NF help us with our "problem" schema?

  $R = (J, K, L)$
  $F = \{JK \rightarrow L, L \rightarrow K\}$

  Two candidate keys: *JK* and *JL*

- Although R is not in BCNF, it is in 3NF:

  $JK \rightarrow L$ — *JK* is a superkey
  $L \rightarrow K$ — *K* is contained in a candidate key

- In other words, if 3NF is our desired level of normalization, then the new algorithm leaves it as is.

Summary: Comparison of BCNF and 3NF, Cont.

- But there is a "cost" to accepting this schema as is…

- Redundancy in 3NF:

  $R = (J, K, L)$
  $F = \{JK \rightarrow L, L \rightarrow K\}$

J	*L*	*K*
j_1	l_1	k_1
j_2	l_1	k_1
j_3	l_1	k_1
null	l_2	k_2

Design Goals

Note #2:

- It is relatively easy to prove that if a relational scheme is in 3NF but not in BCNF; such a relational scheme must have multiple distinct overlapping candidate keys (left as an exercise).

  $R = (J, K, L)$
  $F = \{JK \rightarrow L, L \rightarrow K\}$
  Two candidate keys = *JK* and *JL*

- Thus, if a relational scheme does not have multiple distinct overlapping candidate keys, and if it is in 3NF, then it is also in BCNF.

- Another reason why 3NF is industry standard.

Design Goals

Note #3:

- SQL does not provide a direct way of specifying functional dependencies other than as primary or secondary keys.
- So how are the FD's in the following enforced (in particular, the second)?

  $R = (J, K, L)$
  $F = \{JK \rightarrow L, L \rightarrow K\}$

- FDs can be specified using assertions but they are expensive to test.
- FDs can also be checked in program code, but that has drawbacks.
- In general, using SQL there is no efficient way to test a functional dependency whose left hand side is not a key.

End of Chapter