Chapter 7: Relational Database Design

Database System Concepts, 5th Ed.
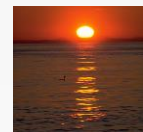
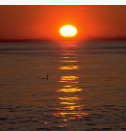# Chapter 7: Relational Database Design

- Features of Good Relational Design
- Atomic Domains and First Normal Form
- Decomposition Using Functional Dependencies
- Functional Dependency Theory
- Algorithms for Functional Dependencies
- Decomposition Using Multivalued Dependencies
- More Normal Form
- Database-Design Process
- Modeling Temporal Data

The Banking Schema

- *branch* = (*branch_name*, *branch_city*, *assets*)
- *customer* = (*customer_id*, *customer_name*, *customer_street*, *customer_city*)
- *loan* = (*loan_number*, *amount*)
- *account* = (*account_number*, *balance*)
- *employee* = (*employee_id*. *employee_name*, *telephone_number*, *start_date*)
- *dependent_name* = (*employee_id, dname*)
- *account_branch* = (*account_number*, *branch_name*)
- *loan_branch* = (*loan_number*, *branch_name*)
- *borrower* = (*customer_id, loan_number*)
- *depositor* = (*customer_id, account_number*)
- *cust_banker* = (*customer_id, employee_id*, *type*)
- *works_for* = (*worker_employee_id*, *manager_employee_id*)
- *payment* = (*loan_number, payment_number*, *payment_date*, *payment_amount*)
- *savings_account* = (*account_number*, *interest_rate*)
- *checking_account* = (*account_number*, *overdraft_amount*)

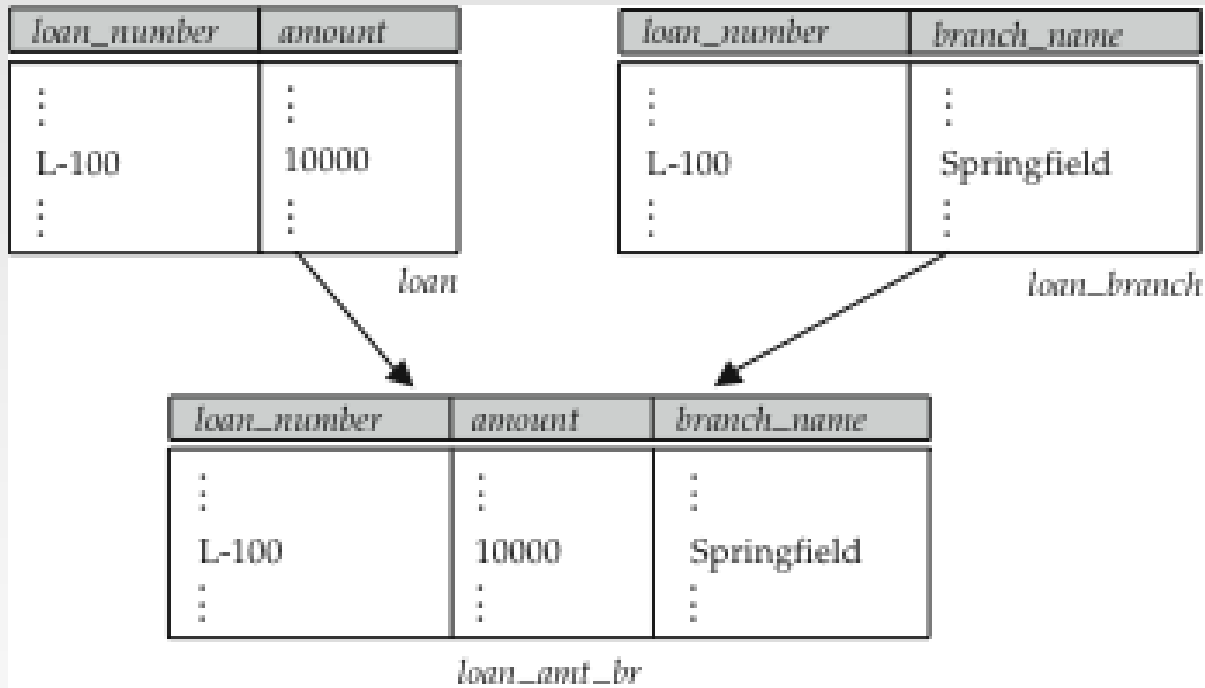# Combine Schemas?

- Suppose we combine *borrower* and *loan* to get

  bor_loan = (*customer_id*, *loan_number*, *amount*)

- Result is possible repetition of information (L-100 in example below)
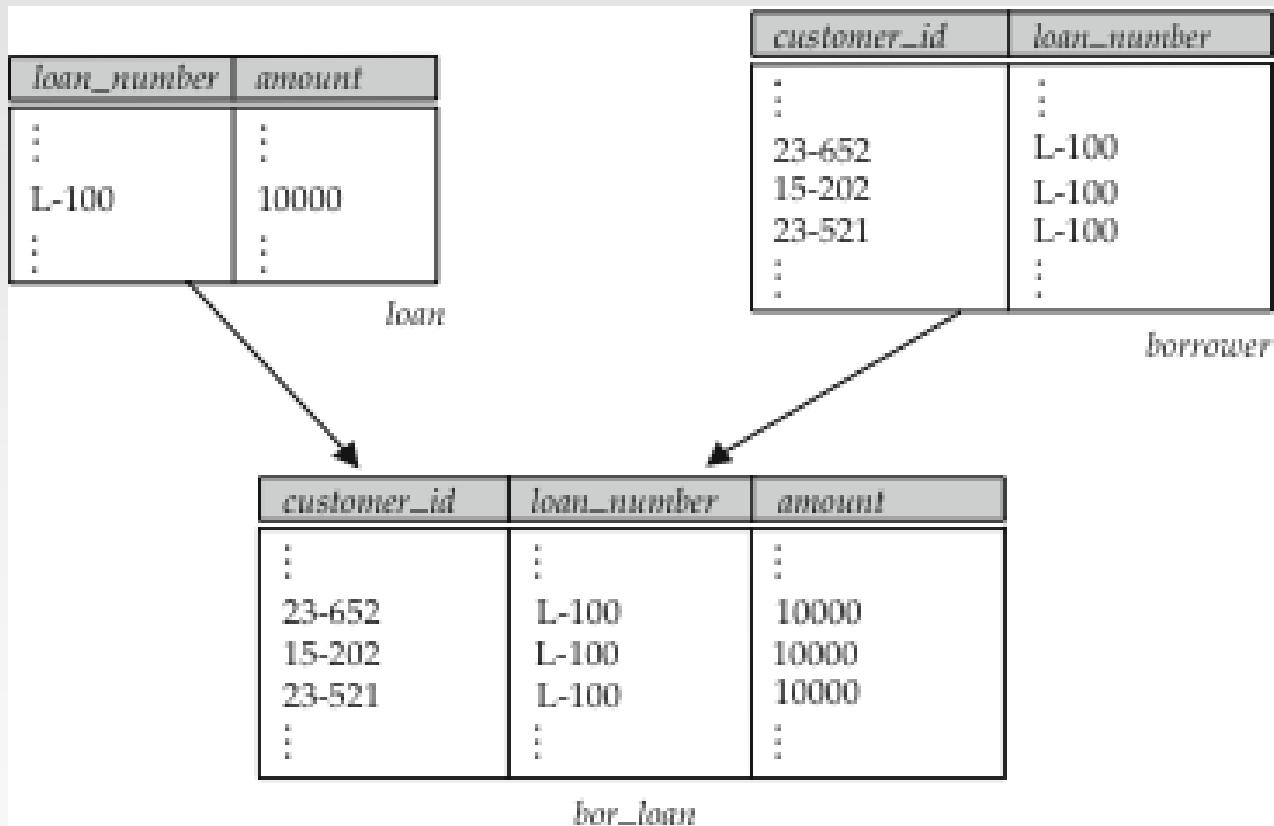
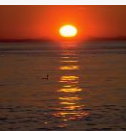# A Combined Schema Without Repetition

□ Consider combining *loan_branch* and *loan*

   loan_amt_br = (*loan_number*, *amount*, *branch_name*)

□ No repetition (as suggested by example below)

What About Smaller Schemas?

- Suppose we had started with *bor_loan.* How would we know to split up (**decompose**) it into *borrower* and *loan*?

- Write a rule "if there were a schema (*loan_number, amount*), then *loan_number* would be a candidate key"

- Denote as a **functional dependency**:

    $$loan\_number \rightarrow amount$$

- In *bor_loan*, because *loan_number* is not a candidate key, the amount of a loan may have to be repeated. This indicates the need to decompose *bor_loan*.

- Not all decompositions are good. Suppose we decompose *employee* into

    employee1 = (*employee_id*, *employee_name*)

    employee2 = (*employee_name*, *telephone_number*, *start_date*)

- The next slide shows how we lose information -- we cannot reconstruct the original *employee* relation -- and so, this is a lossy decomposition.

A Lossy Decomposition
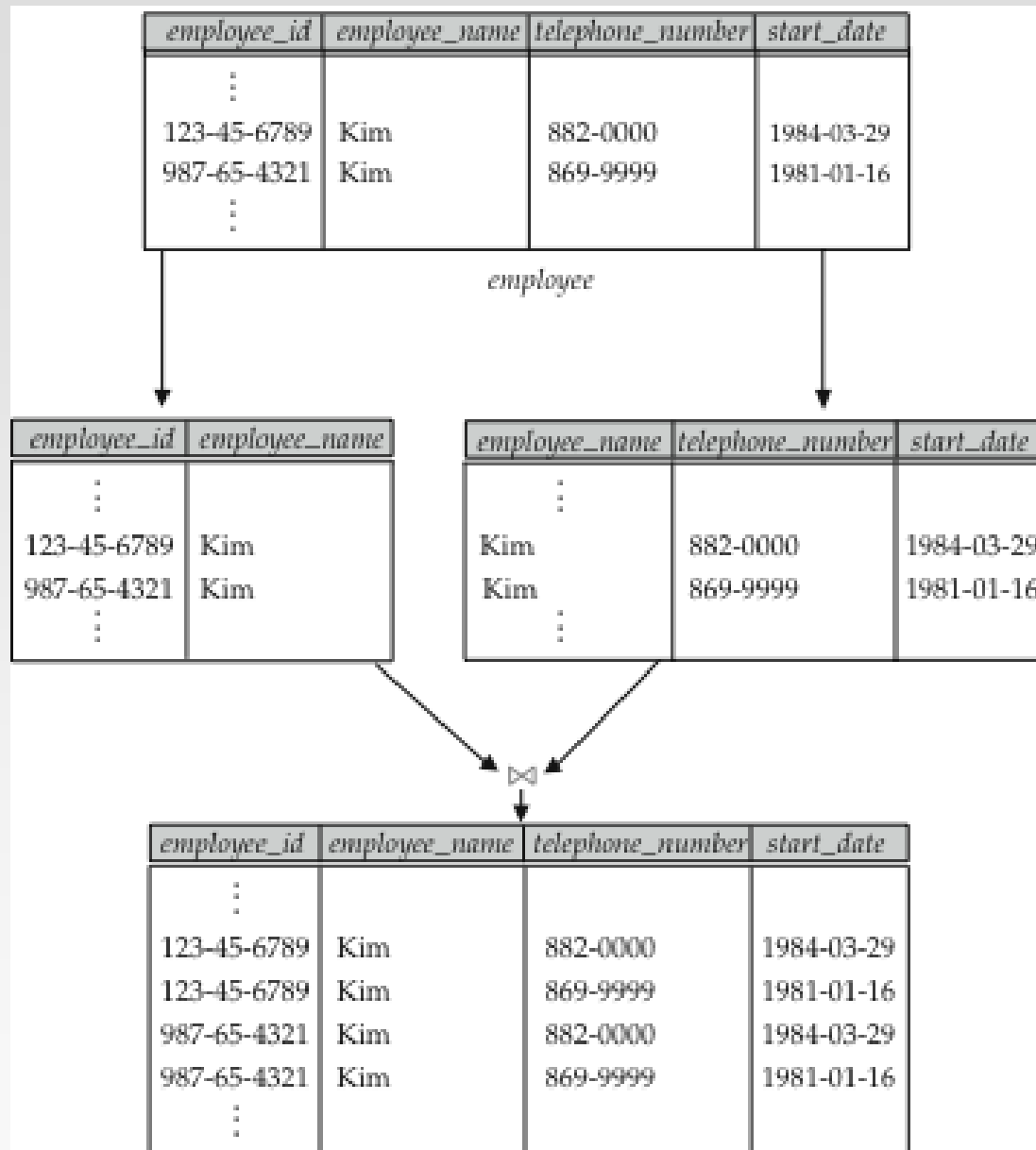
First Normal Form

- 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

- 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

  - Example: Set of accounts stored with each customer, and set of owners stored with each account

  - We assume all relations are in first normal form (and revisit this in Chapter 9)

First Normal Form (Cont'd)

- Atomicity is actually a property of how the elements of the domain are used.

    - Example: Strings would normally be considered indivisible

    - Suppose that students are given roll numbers which are strings of the form *CS0012* or *EE1127*

    - If the first two characters are extracted to find the department, the domain of roll numbers is not atomic.

    - Doing so is a bad idea: leads to encoding of information in application program rather than in the database.

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.*

Functional Dependencies (Cont.)

□ Let R be a relation schema

$$\alpha \subseteq R \ \text{and} \ \beta \subseteq R$$

□ 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,

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

□ Example: Consider $r(A,B)$ with the following instance of r.

1	4
1	5
3	7

□ On this instance, $A \rightarrow B$ does **NOT** hold, but $B \rightarrow A$ does hold.

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 if 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:

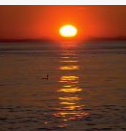 *bor_loan = (customer_id, loan_number, amount).*

We expect this functional dependency to hold:

  loan_number $\rightarrow$ amount

but would not expect the following to hold:

  amount $\rightarrow$ customer_name

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.
- 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* may, by chance, satisfy *amount* $\rightarrow$ *customer_name.*

Functional Dependencies (Cont.)

- *A* functional dependency is trivial if it is satisfied by all instances of a relation

  - Example*:*

    - *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* of functional dependencies, there are certain other functional dependencies that are logically implied by *F*.

  - For example: 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⁺.

- F⁺ is a superset of *F*.

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

$$\alpha \rightarrow \beta$$
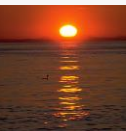
where $\alpha \subseteq R$ and $\beta \subseteq R$, at least one of the following holds:

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

Example schema *not* in BCNF:

$$bor\_loan = (\ customer\_id,\ loan\_number,\ amount\ )$$

because *loan_number* $\rightarrow$ *amount* holds on *bor_loan* but *loan_number* is not a superkey

Decomposing a Schema into BCNF

☐ Suppose we have a schema R and a non-trivial dependency $\alpha \rightarrow \beta$ causes a violation of BCNF.
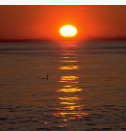
We decompose R into:

- $(\alpha \cup \beta)$
- $(R - (\beta - \alpha))$

☐ In our example,

- ☐ α = *loan_number*
- ☐ β = *amount*

and *bor_loan* is replaced by

- ☐ $(\alpha \cup \beta)$ = (*loan_number, amount*)
- ☐ $(R - (\beta - \alpha))$ = (*customer_id, loan_number*)

BCNF and Dependency Preservation

- Constraints, including functional dependencies, are costly to check in practice unless they pertain to only one relation

- If it is sufficient to test only those dependencies on each individual relation of a decomposition in order to ensure that *all* functional dependencies hold, then that decomposition is *dependency preserving.*

- Because it is not always possible to achieve both BCNF and dependency preservation, we consider a weaker normal form, known as *third normal form.*

Third Normal Form

- A relation schema R is in third normal form (3NF) if for all:

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

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.

  (**NOTE**: each attribute may be in a different candidate key)

- If a relation is in BCNF it is in 3NF (since in BCNF one of the first two conditions above must hold).

- Third condition is a minimal relaxation of BCNF to ensure dependency preservation (will see why later).

Goals of Normalization

- Let R be a relation scheme with a set F of functional dependencies.

- Decide whether a relation scheme R is in "good" form.

- In the case that a relation scheme R is not in "good" form, decompose it into a set of relation scheme $\{R_1, R_2, ..., R_n\}$ such that

  - each relation scheme is in good form

  - the decomposition is a lossless-join decomposition

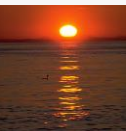 - Preferably, the decomposition should be dependency preserving.

How good is BCNF?

□ There are database schemas in BCNF that do not seem to be sufficiently normalized

□ Consider a database

  classes (*course, teacher, book*)

  such that $(c, t, b) \in$ *classes* means that *t* is qualified to teach *c,* and *b* is a required textbook for *c*

□ The database is supposed to list for each course the set of teachers any one of which can be the course's instructor, and the set of books, all of which are required for the course (no matter who teaches it).

How good is BCNF? (Cont.)

course	teacher	book
database	Avi	DB Concepts
database	Avi	Ullman
database	Hank	DB Concepts
database	Hank	Ullman
database	Sudarshan	DB Concepts
database	Sudarshan	Ullman
operating systems	Avi	OS Concepts
operating systems	Avi	Stallings
operating systems	Pete	OS Concepts
operating systems	Pete	Stallings

classes

- There are no non-trivial functional dependencies and therefore the relation is in BCNF

- Insertion anomalies – i.e., if Marilyn is a new teacher that can teach database, two tuples need to be inserted

> (database, Marilyn, DB Concepts)
> (database, Marilyn, Ullman)

How good is BCNF? (Cont.)
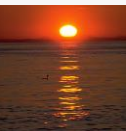
☐ Therefore, it is better to decompose *classes* into:

course	teacher
database	Avi
database	Hank
database	Sudarshan
operating systems	Avi
operating systems	Jim

teaches

course	book
database	DB Concepts
database	Ullman
operating systems	OS Concepts
operating systems	Shaw

text

This suggests the need for higher normal forms, such as Fourth Normal Form (4NF), which we shall see later.

Functional-Dependency Theory

- We now consider the formal theory that tells us which functional dependencies are implied logically by a given set of functional dependencies.

- We then develop algorithms to generate lossless decompositions into BCNF and 3NF

- We then develop algorithms to test if a decomposition is dependency-preserving

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*.

  - For example: 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*[+].

- 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 that hold).

Example

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

- some members of F^+

  - $A \rightarrow H$

    - by transitivity from $A \rightarrow B$ and $B \rightarrow H$

  - $AG \rightarrow I$

    - by augmenting $A \rightarrow C$ with G, to get $AG \rightarrow CG$
      and then transitivity with $CG \rightarrow I$

  - $CG \rightarrow HI$

    - by augmenting $CG \rightarrow I$ to infer $CG \rightarrow CGI$,

      and augmenting of $CG \rightarrow H$ to infer $CGI \rightarrow HI$,

      and then transitivity

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
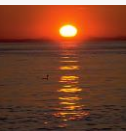   add the resulting functional dependencies to F^+
    for each pair of functional dependencies f_1 and f_2 in F^+
        if f_1 and f_2 can be combined using transitivity
            then add the resulting functional dependency to F^+
until F^+ does not change any further

**NOTE**: We shall 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 and $\alpha \rightarrow \gamma$ 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 and $\gamma\beta \rightarrow \delta$ holds, then $\alpha\gamma \rightarrow \delta$ holds **(pseudotransitivity)**

    The above rules can be inferred from Armstrong's axioms.
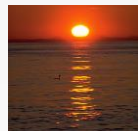
Closure of Attribute Sets

- 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*

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

  result := α;
  while (changes to *result*) **do**
      for each $\beta \rightarrow \gamma$ **in** *F* **do**
        begin
          if $\beta \subseteq$ *result* **then** *result* := *result* $\cup$ γ
        end

Example of Attribute Set Closure

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

  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 $(G)^+ \supseteq R$

Uses of Attribute Closure
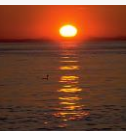
There are several uses of the attribute closure algorithm:

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

- 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

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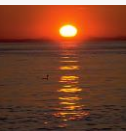
Canonical Cover

- Sets of functional dependencies may have redundant dependencies that can be inferred from the others

    - For example: $A \rightarrow C$ is redundant in: $\{A \rightarrow B, \quad B \rightarrow C\}$

    - 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
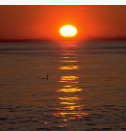
Extraneous Attributes

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

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

- Example: Given $F = \{A \rightarrow C, AB \rightarrow C\}$

  - B is extraneous in $AB \rightarrow C$ because $\{A \rightarrow C, AB \rightarrow C\}$ logically implies $A \rightarrow C$ (I.e. the result of dropping B from $AB \rightarrow C$).

- 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
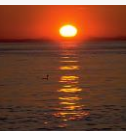
Testing if an Attribute is Extraneous

- 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 β; if it does, *A* is extraneous in α

- 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 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.

- To compute a canonical cover for F:
  repeat

  Use the union rule to replace any dependencies in F

  $$\alpha_1 \rightarrow \beta_1 \text{ and } \alpha_1 \rightarrow \beta_2 \text{ with } \alpha_1 \rightarrow \beta_1 \ \beta_2$$

  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

Computing a Canonical Cover

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

- Combine $A \rightarrow BC$ and $A \rightarrow B$ into $A \rightarrow BC$

  - Set is now $\{A \rightarrow BC, B \rightarrow C, AB \rightarrow C\}$

- 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!

  - 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: $\quad\quad A \rightarrow B$
  $\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad B \rightarrow C$

Lossless-join Decomposition

- For the case of $R = (R_1, R_2)$, we require that for all possible relations r on schema R

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

- 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^+:

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

Example

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

  - Can be decomposed in two different ways

- $R_1 = (A, B),\quad R_2 = (B, C)$

  - Lossless-join decomposition:

  $$R_1 \cap R_2 = \{B\} \text{ and } B \rightarrow BC$$

  - Dependency preserving

- $R_1 = (A, B),\quad R_2 = (A, C)$

  - Lossless-join decomposition:

  $$R_1 \cap R_2 = \{A\} \text{ and } A \rightarrow AB$$

  - Not dependency preserving
    (cannot check $B \rightarrow C$ without computing $R_1 \bowtie R_2$)
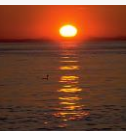
Dependency Preservation

☐ Let F_i be the set of dependencies F^+ that include only attributes in R_i.

▸ A decomposition is dependency preserving, if

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

▸ If it is not, then checking updates for violation of functional dependencies may require computing joins, which is expensive.

Testing for Dependency Preservation

- To check if a dependency $\alpha \rightarrow \beta$ is preserved in a decomposition of R into $R_1, R_2, \ldots, R_n$ we apply the following test (with attribute closure done with respect to F)

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

  - If $result$ contains all attributes in β, then the functional dependency $\alpha \rightarrow \beta$ is preserved.

- We apply the test on all dependencies in F to check if a decomposition is dependency preserving

- This procedure takes polynomial time, instead of the exponential time required to compute F^+ and $(F_1 \cup F_2 \cup \ldots \cup F_n)^+$
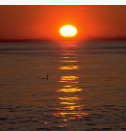
Example

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

- R is not in BCNF

- Decomposition $R_1 = (A, B),\ R_2 = (B, C)$

  - R_1 and R_2 in BCNF

  - Lossless-join decomposition
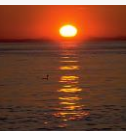
  - Dependency preserving

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.

- However, using only F is incorrect when testing a relation in a decomposition of R

  - Consider $R = (A, B, C, D, E)$, with $F = \{ A \rightarrow B, BC \rightarrow D\}$

    - Decompose R into $R_1 = (A,B)$ and $R_2 = (A,C,D, E)$

    - Neither of the dependencies in F contain only attributes from (A,C,D,E) so we might be mislead into thinking R_2 satisfies BCNF.

    - In fact, dependency $AC \rightarrow D$ in F^+ shows R_2 is not in BCNF.

Testing Decomposition for BCNF

☐ To check if a relation R_i in a decomposition of R is in BCNF,

    ☐ Either test R_i for BCNF with respect to the restriction of F to R_i (that is, all FDs in F^+ that contain only attributes from R_i)

    ☐ or use the original set of dependencies F that hold on R, but with the following test:

        – for every set of attributes $\alpha \subseteq R_i$, check that α^+ (the attribute closure of α) either includes no attribute of R_i- α, or includes all attributes of R_i.

        ▸ If the condition is violated by some $\alpha \to \beta$ in F, the dependency
$$\alpha \to (\alpha^+ - \alpha) \cap R_i$$
can be shown to hold on R_i, and R_i violates BCNF.

        ▸ We use above dependency to decompose R_i

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 $\alpha \cap \beta = \varnothing$;
      $result := (result - R_i) \cup (R_i - \beta) \cup (\alpha, \beta)$;
    end
  else $done :=$ **true;**

Note: each R_i is in BCNF, and decomposition is lossless-join.

Example of BCNF Decomposition

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

- R is not in BCNF ($B \rightarrow C$ but B is not superkey)

- Decomposition

  - $R_1 = (B, C)$

  - $R_2 = (A, B)$

Example of BCNF Decomposition

- Original relation *R* and functional dependency *F*

   R = (*branch_name, branch_city, assets,*

        customer_name, loan_number, amount)

   F = {*branch_name* $\rightarrow$ *assets branch_city*

      loan_number $\rightarrow$ *amount branch_name* }

   Key = {*loan_number, customer_name*}

- Decomposition

   - R_1 = (*branch_name, branch_city, assets*)

   - R_2 = (*branch_name, customer_name, loan_number, amount*)

   - R_3 = (*branch_name, loan_number, amount*)

   - R_4 = (*customer_name, loan_number*)

- Final decomposition

   $$R_1, R_3, R_4$$
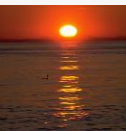
BCNF and Dependency Preservation

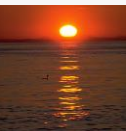It is not always possible to get a BCNF decomposition that is dependency preserving

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

☐ R is not in BCNF

☐ Any decomposition of R will fail to preserve
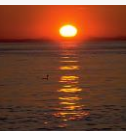
$$JK \rightarrow L$$

This implies that testing for $JK \rightarrow L$ requires a join

Third Normal Form: Motivation

- There are some situations where

    - BCNF is not dependency preserving, and

    - efficient checking for FD violation on updates is important

- Solution: define a weaker normal form, called Third Normal Form (3NF)

    - Allows some redundancy (with resultant problems; we will see examples later)

    - But functional dependencies can be checked on individual relations without computing a join.

    - There is always a lossless-join, dependency-preserving decomposition into 3NF.

3NF Example

- Relation R:

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

  - Two candidate keys: *JK* and *JL*

  - *R* is in 3NF

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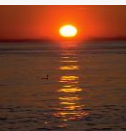 $L \rightarrow K$ *K* is contained in a candidate key

Redundancy in 3NF

- There is some redundancy in this schema

- Example of problems due to redundancy in 3NF

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

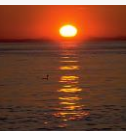| J	L	K
j_1	l_1	k_1
j_2	l_1	k_1
j_3	l_1	k_1
null	l_2	k_2

- repetition of information (e.g., the relationship l_1, k_1)

- need to use null values (e.g., to represent the relationship l_2, k_2 where there is no corresponding value for J).

Testing for 3NF

- Optimization: Need to check only FDs in *F*, need not check all FDs in F^+.

- Use attribute closure to check for each dependency $\alpha \rightarrow \beta$, if α is a superkey.

- If α is not a superkey, we have to verify if each attribute in β is contained in a candidate key of *R*

  - this test is rather more expensive, since it involve finding candidate keys

  - testing for 3NF has been shown to be NP-hard

  - Interestingly, decomposition into third normal form (described shortly) can be done in polynomial time

3NF Decomposition Algorithm

Let F_c be a canonical cover for F;
$i := 0$;
for each functional dependency $\alpha \to \beta$ in F_c **do**
  if none of the schemas R_j, $1 \le j \le i$ contains $\alpha\ \beta$
        then begin
            $i := i + 1$;
            $R_i := \alpha\ \beta$
        end
if none of the schemas R_j, $1 \le j \le i$ contains a candidate key for R
  then begin
            $i := i + 1$;
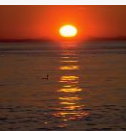 $R_i :=$ any candidate key for R;
        end
return $(R_1, R_2, ..., R_i)$

3NF Decomposition Algorithm (Cont.)

☐ Above algorithm ensures:

☐ each relation schema R_i is in 3NF

☐ decomposition is dependency preserving and lossless-join

☐ Proof of correctness is at end of this presentation (<u>click here</u>)

3NF Decomposition: An Example

- Relation schema:

  cust_banker_branch = (<u>customer_id, employee_id</u>, branch_name, type)

- The functional dependencies for this relation schema are:

  1. *customer_id, employee_id $\rightarrow$ branch_name, type*

  2. *employee_id $\rightarrow$ branch_name*

  3. *customer_id, branch_name $\rightarrow$ employee_id*

- We first compute a canonical cover

  - *branch_name* is extraneous in the r.h.s. of the 1st dependency

  - No other attribute is extraneous, so we get F_C =

    customer_id, employee_id $\rightarrow$ type
    employee_id $\rightarrow$ branch_name
    customer_id, branch_name $\rightarrow$ employee_id

3NF Decompsition Example (Cont.)

☐ The **for** loop generates following 3NF schema:

   (*customer_id, employee_id, type*)

   (*employee_id, branch_name*)

   (*customer_id, branch_name, employee_id*)

   ☐ Observe that (*customer_id, employee_id, type*) contains a candidate key of the original schema, so no further relation schema needs be added

☐ If the FDs were considered in a different order, with the 2nd one considered after the 3rd,

   (*employee_id, branch_name*)

would not be included in the decomposition because it is a subset of

   (*customer_id, branch_name, employee_id*)

☐ Minor extension of the 3NF decomposition algorithm: at end of for loop, detect and delete schemas, such as (*employee_id, branch_name*), which are subsets of other schemas

   ☐ result will not depend on the order in which FDs are considered

☐ The resultant simplified 3NF schema is:

   (*customer_id, employee_id, type*)

   (*customer_id, branch_name, employee_id*)

Comparison of BCNF and 3NF

- It is always possible to decompose a relation into a set of relations that are in 3NF such that:

  - the decomposition is lossless

  - the dependencies are preserved

- It is always possible to decompose a relation into a set of relations that are in BCNF such that:

  - the decomposition is lossless

  - it may not be possible to preserve dependencies.

Design Goals

- Goal for a relational database design is:
    - BCNF.
    - Lossless join.
    - Dependency preservation.
- If we cannot achieve this, we accept one of
    - Lack of dependency preservation
    - Redundancy due to use of 3NF
- Interestingly, SQL does not provide a direct way of specifying functional dependencies other than superkeys.

    Can specify FDs using assertions, but they are expensive to test
- Even if we had a dependency preserving decomposition, using SQL we would not be able to efficiently test a functional dependency whose left hand side is not a key.
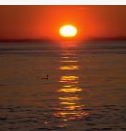
Multivalued Dependencies (MVDs)

▢ Let R be a relation schema and let $\alpha \subseteq R$ and $\beta \subseteq R.$ The *multivalued dependency*

$$\alpha \rightarrow\rightarrow \beta$$

holds on R if in any legal relation $r(R),$ for all pairs for tuples t_1 and t_2 in r such that $t_1[\alpha] = t_2[\alpha]$, there exist tuples t_3 and t_4 in r such that:

$$t_1[\alpha] = t_2[\alpha] = t_3[\alpha] = t_4[\alpha]$$
$$t_3[\beta] \quad\quad = \quad t_1[\beta]$$
$$t_3[R - \beta] = \quad t_2[R - \beta]$$
$$t_4[\beta] \quad\quad = \quad t_2[\beta]$$
$$t_4[R - \beta] = \quad t_1[R - \beta]$$

MVD (Cont.)

- Tabular representation of $\alpha \rightarrow\rightarrow \beta$

	α	β	$R - \alpha - \beta$
t_1	$a_1 \ldots a_i$	$a_{i+1} \ldots a_j$	$a_{j+1} \ldots a_n$
t_2	$a_1 \ldots a_i$	$b_{i+1} \ldots b_j$	$b_{j+1} \ldots b_n$
t_3	$a_1 \ldots a_i$	$a_{i+1} \ldots a_j$	$b_{j+1} \ldots b_n$
t_4	$a_1 \ldots a_i$	$b_{i+1} \ldots b_j$	$a_{j+1} \ldots a_n$

Example

- Let R be a relation schema with a set of attributes that are partitioned into 3 nonempty subsets.

$$Y, Z, W$$

- We say that $Y \rightarrow\rightarrow Z$ (Y multidetermines Z) if and only if for all possible relations $r(R)$

$$< y_1, z_1, w_1 > \in r \text{ and } < y_1, z_2, w_2 > \in r$$

then

$$< y_1, z_1, w_2 > \in r \text{ and } < y_1, z_2, w_1 > \in r$$

- Note that since the behavior of Z and W are identical it follows that

$$Y \rightarrow\rightarrow Z \text{ if } Y \rightarrow\rightarrow W$$
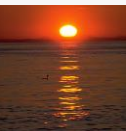
Example (Cont.)

- In our example:

$$course \rightarrow\rightarrow teacher$$
$$course \rightarrow\rightarrow book$$

- The above formal definition is supposed to formalize the notion that given a particular value of Y (*course*) it has associated with it a set of values of Z *(teacher)* and a set of values of W *(book)*, and these two sets are in some sense independent of each other.

- Note:

    - If $Y \rightarrow Z$ then $Y \rightarrow\rightarrow Z$

    - Indeed we have (in above notation) $Z_1 = Z_2$
      The claim follows.

Use of Multivalued Dependencies

- We use multivalued dependencies in two ways:

  1. To test relations to determine whether they are legal under a given set of functional and multivalued dependencies

  2. To specify constraints on the set of legal relations. We shall thus concern ourselves *only* with relations that satisfy a given set of functional and multivalued dependencies.

- If a relation *r* fails to satisfy a given multivalued dependency, we can construct a relations *r′* that does satisfy the multivalued dependency by adding tuples to *r.*

# Theory of MVDs

- From the definition of multivalued dependency, we can derive the following rule:

    - If $\alpha \rightarrow \beta$, then $\alpha \rightarrow\rightarrow \beta$

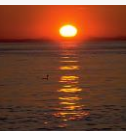   That is, every functional dependency is also a multivalued dependency

- The **closure** D^+ of D is the set of all functional and multivalued dependencies logically implied by D.

    - We can compute D^+ from D, using the formal definitions of functional dependencies and multivalued dependencies.

    - We can manage with such reasoning for very simple multivalued dependencies, which seem to be most common in practice

    - For complex dependencies, it is better to reason about sets of dependencies using a system of inference rules (see Appendix C).
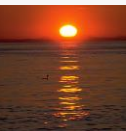
Fourth Normal Form

- A relation schema R is in 4NF with respect to a set D of functional and multivalued dependencies if for all multivalued dependencies in D^+ of the form $\alpha \rightarrow\rightarrow \beta$, where $\alpha \subseteq R$ and $\beta \subseteq R$, at least one of the following hold:

    - $\alpha \rightarrow\rightarrow \beta$ is trivial (i.e., $\beta \subseteq \alpha$ or $\alpha \cup \beta = R$)

    - α is a superkey for schema R

- If a relation is in 4NF it is in BCNF

Restriction of Multivalued Dependencies

□ The restriction of D to R_i is the set D_i consisting of

   □ All functional dependencies in D^+ that include only attributes of R_i

   □ All multivalued dependencies of the form

   $$\alpha \rightarrow\rightarrow (\beta \cap R_i)$$

   where $\alpha \subseteq R_i$ and $\alpha \rightarrow\rightarrow \beta$ is in D^+

4NF Decomposition Algorithm

result: = {*R*};
done := false;
compute D⁺;
Let D$_i$ denote the restriction of D⁺ to R$_i$

while (**not** *done*)
   if (there is a schema **R**$_i$ in *result* that is not in 4NF) **then**
     begin

       let $\alpha \rightarrow\rightarrow \beta$ be a nontrivial multivalued dependency that holds
         on R_i such that $\alpha \rightarrow R_i$ is not in D_i, and $\alpha \cap \beta = \phi$;
       result := (*result* - R_i) $\cup$ (R_i - β) $\cup$ (α, β);
     end
   else *done*:= true;

Note: each R_i is in 4NF, and decomposition is lossless-join

Example

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

  $F = \{ A \rightarrow\rightarrow B$

  $\qquad B \rightarrow\rightarrow HI$

  $\qquad CG \rightarrow\rightarrow H \}$

- R is not in 4NF since $A \rightarrow\rightarrow B$ and A is not a superkey for R

- Decomposition

  a) $R_1 = (A, B)$ (R_1 is in 4NF)

  b) $R_2 = (A, C, G, H, I)$ (R_2 is not in 4NF)

  c) $R_3 = (C, G, H)$ (R_3 is in 4NF)

  d) $R_4 = (A, C, G, I)$ (R_4 is not in 4NF)

- Since $A \rightarrow\rightarrow B$ and $B \rightarrow\rightarrow HI$, $A \rightarrow\rightarrow HI$, $A \rightarrow\rightarrow I$

  e) $R_5 = (A, I)$ (R_5 is in 4NF)

  f) $R_6 = (A, C, G)$ (R_6 is in 4NF)
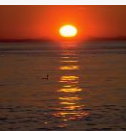
Further Normal Forms

- **Join dependencies** generalize multivalued dependencies
    - lead to **project-join normal form (**PJNF) (also called **fifth normal form**)
- A class of even more general constraints, leads to a normal form called **domain-key normal form**.
- Problem with these generalized constraints: are hard to reason with, and no set of sound and complete set of inference rules exists.
- Hence rarely used

Overall Database Design Process

- We have assumed schema *R* is given

  - *R* could have been generated when converting E-R diagram to a set of tables.

  - *R* could have been a single relation containing *all* attributes that are of interest (called **universal relation**).

  - Normalization breaks *R* into smaller relations.

  - *R* could have been the result of some ad hoc design of relations, which we then test/convert to normal form.

ER Model and Normalization

- When an E-R diagram is carefully designed, identifying all entities correctly, the tables generated from the E-R diagram should not need further normalization.

- However, in a real (imperfect) design, there can be functional dependencies from non-key attributes of an entity to other attributes of the entity

  - Example: an *employee* entity with attributes *department_number* and *department_address*, and a functional dependency *department_number* $\rightarrow$ *department_address*

  - Good design would have made department an entity

- Functional dependencies from non-key attributes of a relationship set possible, but rare --- most relationships are binary

Denormalization for Performance

- May want to use non-normalized schema for performance

- For example, displaying *customer_name* along with *account_number* and *balance* requires join of *account* with *depositor*

- Alternative 1: Use denormalized relation containing attributes of *account* as well as *depositor* with all above attributes

  - faster lookup

  - extra space and extra execution time for updates

  - extra coding work for programmer and possibility of error in extra code

- Alternative 2: use a materialized view defined as
  account ⋈ depositor

  - Benefits and drawbacks same as above, except no extra coding work for programmer and avoids possible errors

Other Design Issues

- Some aspects of database design are not caught by normalization
- Examples of bad database design, to be avoided:

Instead of *earnings* (*company_id, year, amount*), use

- *earnings_2004, earnings_2005, earnings_2006*, etc., all on the schema (*company_id, earnings*).
  - Above are in BCNF, but make querying across years difficult and needs new table each year
- *company_year*(*company_id, earnings_2004, earnings_2005, earnings_2006*)
  - Also in BCNF, but also makes querying across years difficult and requires new attribute each year.
  - Is an example of a **crosstab**, where values for one attribute become column names
  - Used in spreadsheets, and in data analysis tools

Modeling Temporal Data

- *Temporal data* have an association time interval during which the data are *valid.*

- A *snapshot* is the value of the data at a particular point in time

- Several proposals to extend ER model by adding valid time to

  - attributes, e.g. address of a customer at different points in time

  - entities, e.g. time duration when an account exists

  - relationships, e.g. time during which a customer owned an account

- But no accepted standard

- Adding a temporal component results in functional dependencies like

  $$customer\_id \rightarrow customer\_street, customer\_city$$

  not to hold, because the address varies over time

- A *temporal functional dependency* $X \rightarrow Y$ holds on schema *R* if the functional dependency $X \rightarrow Y$ holds on all snapshots for all legal instances r (*R*)

Modeling Temporal Data (Cont.)

- In practice, database designers may add start and end time attributes to relations

    - E.g. *course*(*course_id, course_title*) →

        course(*course_id, course_title, start, end*)

        - Constraint: no two tuples can have overlapping valid times
            - Hard to enforce efficiently

- Foreign key references may be to current version of data, or to data at a point in time

    - E.g. student transcript should refer to course information at the time the course was taken

End of Chapter

Database System Concepts, 5th Ed.
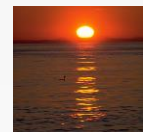
Proof of Correctness of 3NF Decomposition Algorithm
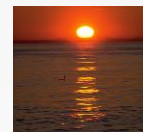
Correctness of 3NF Decomposition Algorithm

☐ 3NF decomposition algorithm is dependency preserving (since there is a relation for every FD in F_c)

☐ Decomposition is lossless

  ☐ A candidate key (C) is in one of the relations R_i in decomposition

  ☐ Closure of candidate key under F_c must contain all attributes in R.

  ☐ Follow the steps of attribute closure algorithm to show there is only one tuple in the join result for each tuple in R_i
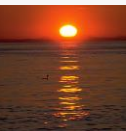
Correctness of 3NF Decomposition Algorithm (Cont'd.)

Claim: if a relation R_i is in the decomposition generated by the

above algorithm, then R_i satisfies 3NF.

□ Let R_i be generated from the dependency $\alpha \rightarrow \beta$

□ Let $\gamma \rightarrow B$ be any non-trivial functional dependency on R_i. (We need only consider FDs whose right-hand side is a single attribute.)

□ Now, B can be in either β or α but not in both. Consider each case separately.

Correctness of 3NF Decomposition (Cont'd.)

- Case 1: If B in β:

  - If γ is a superkey, the 2nd condition of 3NF is satisfied

  - Otherwise α must contain some attribute not in γ

  - Since $\gamma \rightarrow B$ is in F^+ it must be derivable from F_c, by using attribute closure on γ.

  - Attribute closure not have used $\alpha \rightarrow \beta$. If it had been used, α must be contained in the attribute closure of γ, which is not possible, since we assumed γ is not a superkey.

  - Now, using $\alpha \rightarrow (\beta - \{B\})$ and $\gamma \rightarrow B$, we can derive $\alpha \rightarrow B$

    (since $\gamma \subseteq \alpha \beta$, and B $\notin \gamma$ since $\gamma \rightarrow B$ is non-trivial)

  - Then, B is extraneous in the right-hand side of $\alpha \rightarrow \beta$; which is not possible since $\alpha \rightarrow \beta$ is in F_c.

  - Thus, if B is in β then γ must be a superkey, and the second condition of 3NF must be satisfied.

Correctness of 3NF Decomposition (Cont'd.)

- Case 2: *B* is in α.

  - Since α is a candidate key, the third alternative in the definition of 3NF is trivially satisfied.

  - In fact, we cannot show that γ is a superkey.

  - This shows exactly why the third alternative is present in the definition of 3NF.

Q.E.D.

Figure 7.5: Sample Relation r

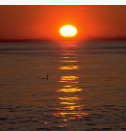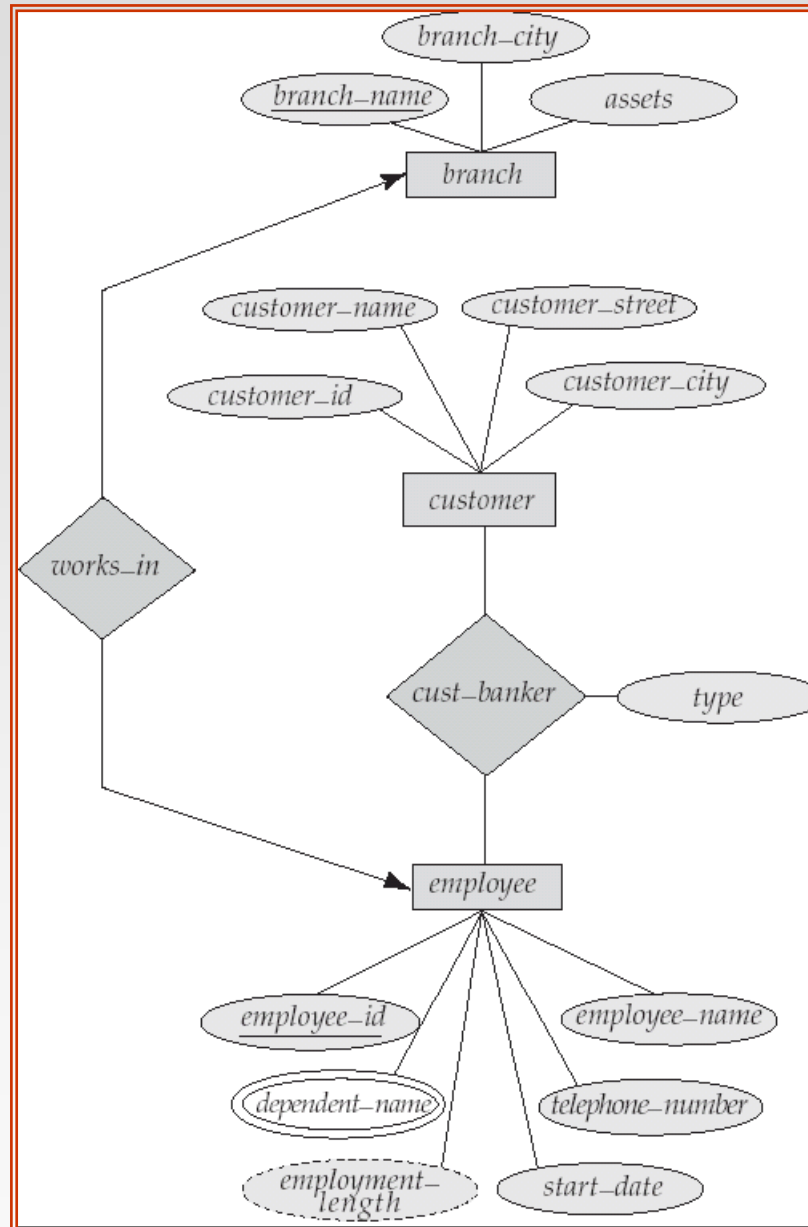| A	B	C	D
a_1	b_1	c_1	d_1
a_1	b_2	c_1	d_2
a_2	b_2	c_2	d_2
a_2	b_3	c_2	d_3
a_3	b_3	c_2	d_4
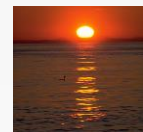
Figure 7.6

Figure 7.7
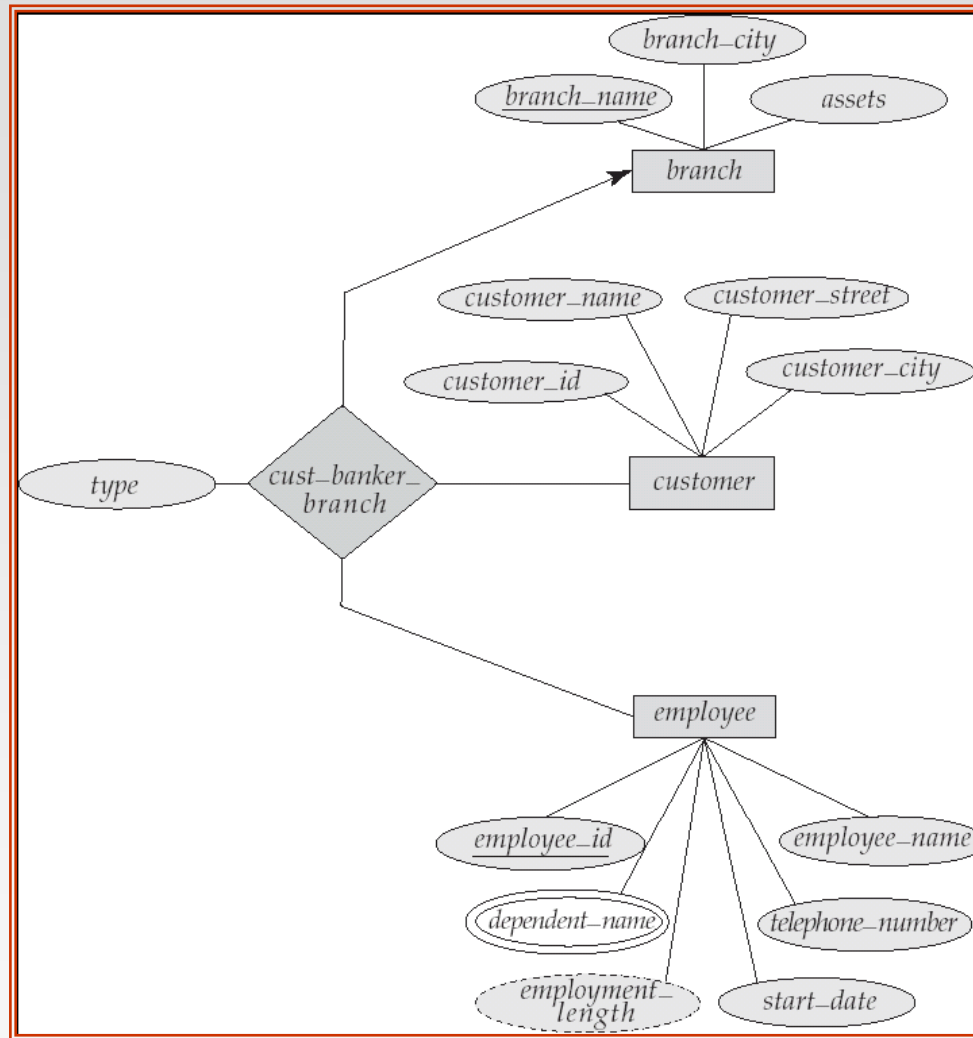
Figure 7.15: An Example of Redundancy in a BCNF Relation

loan_number	customer_id	customer_street	customer_city
L-23	99-123	North	Rye
L-23	99-123	Main	Manchester
L-93	15-106	Lake	Horseneck

Figure 7.16: An Illegal R_2 Relation

loan_number	customer_id	customer_street	customer_city
L-23	99-123	North	Rye
L-27	99-123	Main	Manchester
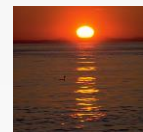
Figure 7.18: Relation of Practice Exercise 7.2

A	B	C
a_1	b_1	c_1
a_1	b_1	c_2
a_2	b_1	c_1
a_2	b_1	c_3