COP 4710: Database Systems
Spring 2007

Chapter 19 – Normalization

Instructor :   Mark Llewellyn
                 markl@cs.ucf.edu
                 ENG3 236, 823-2790
                 http://www.cs.ucf.edu/courses/cop4710/spr2007

School of Electrical Engineering and Computer Science
University of Central Florida

Normalization

- Normalization is a technique for producing a set of relations with desirable properties, given the data requirements of the enterprise being modeled.

- The process of normalization was first developed by Codd in 1972.

- Normalization is often performed as a series of tests on a relation to determine whether it satisfies or violates the requirements of a given normal form.

- Codd initially defined three normal forms called first (1NF), second (2NF), and third (3NF). Boyce and Codd together introduced a stronger definition of 3NF called Boyce-Codd Normal Form (BCNF) in 1974.

Normalization (cont.)

- All four of these normal forms are based on functional dependencies among the attributes of a relation.

- A functional dependency describes the relationship between attributes in a relation.

  - For example, if A and B are attributes or sets of attributes of relation R, B is functionally dependent on A (denoted A $\rightarrow$ B), if each value of A is associated with exactly one value of B.

- In 1977 and 1979, a fourth (4NF) and fifth (5NF) normal form were introduced which go beyond BCNF. However, they deal with situations which are quite rare. Other higher normal forms have been subsequently introduced, but all of them are based on dependencies more involved than functional dependencies.

Normalization (cont.)

- A relational schema consists of a number of attributes, and a relational database schema consists of a number of relations.

- Attributes may grouped together to form a relational schema based largely on the common sense of the database designer, or by mapping the relational schema from an ER model.

- Whatever approach is taken, a formal method is often required to help the database designer identify the optimal grouping of attributes for each relation in the database schema.

- The process of normalization is a formal method that identifies relations based on their primary or candidate keys and the functional dependencies among their attributes.

- Normalization supports database designers through a series of tests, which can be applied to individual relations so that a relational schema can be normalized to a specific form to prevent the possible occurrence of update anomalies.

Data Redundancy and Update Anomalies

- The major aim of relational database design is to group attributes into relations to minimize data redundancy and thereby reduce the file storage space required by the implemented base relations.

- Consider the following relation schema:

staffbranch

staff#	sname	position	salary	branch#	baddress
SL21	Kristy	manager	30000	B005	22 Deer Road
SG37	Debi	assistant	12000	B003	162 Main Street
SG14	Alan	supervisor	18000	B003	163 Main Street
SA9	Traci	assistant	12000	B007	375 Fox Avenue
SG5	David	manager	24000	B003	163 Main Street
SL41	Anna	assistant	10000	B005	22 Deer Road

Data Redundancy and Update Anomalies (cont.)

- The **staffbranch** relation on the previous page contains redundant data. The details of a branch are repeated for every member of the staff located at that branch. Contrast this with the relation schemas shown below.

- In this case, branch details appear only once for each branch.

staff

staff#	sname	position	salary	branch#
SL21	Kristy	manager	30000	B005
SG37	Debi	assistant	12000	B003
SG14	Alan	supervisor	18000	B003
SA9	Traci	assistant	12000	B007
SG5	David	manager	24000	B003
SL41	Anna	assistant	10000	B005

branch

branch#	baddress
B005	22 Deer Road
B003	163 Main Street
B007	375 Fox Avenue

Data Redundancy and Update Anomalies (cont.)

- Relations which contain redundant data may have problems called update anomalies, which can be classified as insertion, deletion, or modification (update) anomalies.

Insertion Anomalies

1. To insert the details of new staff members into the staffbranch relation, we must include the details of the branch at which the new staff member is to be located.

    - For example, if the new staff member is to be located at branch B007, we must enter the correct address so that it matches existing tuples in the relation. The database schema with staff and branch does not suffer this problem.

2. To insert the details of a new branch that currently has no staff members, we'll need to insert null values for the attributes of the staff such as staff number. However, since staff number is a primary key, this would violate key integrity and is not allowed. Thus we cannot enter information for a new branch with no staff members!

Data Redundancy and Update Anomalies (cont.)

Deletion Anomalies

- If we delete a tuple from the staffbranch relation that represents the last member of the staff located at that branch, the details about that branch will also be lost from the database.

    - For example, if we delete staff member Traci from the staffbranch relation then the information about branch B007 will also be lost. This however, is not the case with the database schema (staff, branch) because details about the staff are maintained separately from details about the various branches.

Data Redundancy and Update Anomalies (cont.)

Modification Anomalies

- If we want to change the value of one of the attributes of a particular branch in the staffbranch relation, for example, the address for branch number B003, we'll need to update the tuples for every staff member located at that branch.

- If this modification is not carried out on all the appropriate tuples of the staffbranch relation, the database will become inconsistent, e.g., branch B003 will appear to have different addresses for different staff members.

Data Redundancy and Update Anomalies (cont.)

- The examples of three types of update anomalies suffered by the staffbranch relation demonstrate that its decomposition into the staff and branch relations avoids such anomalies.

- There are two important properties associated with the decomposition of a larger relation into a set of smaller relations.

1. The lossless-join property ensures that any instance of the original relation can be identified from corresponding instances of the smaller relations.

2. The dependency preservation property ensures that a constraint on the original relation can be maintained by simply enforcing some constraint on each of the smaller relations. In other words, the smaller relations do not need to be joined together to check if a constraint on the original relation is violated.

The Lossless Join Property

- Consider the following relation schema SP and its decomposition into two schemas S1 and S2.

SP

s#	p#	qty
S1	P1	10
S2	P2	50
S3	P3	10

S1

s#	qty
S1	10
S2	50
S3	10

S2

p#	qty
P1	10
P2	50
P3	10

S1 ▷◁ S2

s#	p#	qty
S1	P1	10
S1	P3	10
S2	P2	10
S3	P1	10
S3	P3	10

These are extraneous tuples which did not appear in the original relation. However, now we can't tell which are valid and which aren't. Once the decomposition occurs the original SP relation is lost.

Preservation of the Functional Dependencies

Example

$R = (A, B, C)$

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

$\gamma = \{(B, C), (A, C)\}$

Clearly $C \rightarrow A$ can be enforced on schema (A, C).

How can $AB \rightarrow C$ be enforced without joining the two relation schemas in γ? Answer, it can't, therefore the fds are not preserved in γ.
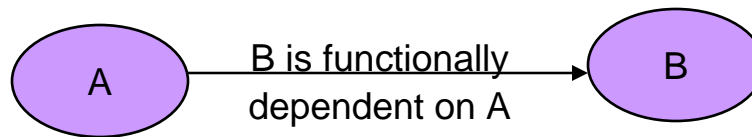
Functional Dependencies

- For our discussion on functional dependencies (fds), assume that a relational schema has attributes (A, B, C, ..., Z) and that the whole database is described by a single universal relation called R = (A, B, C, ..., Z). This assumption means that every attribute in the database has a unique name.

- A functional dependency is a property of the semantics of the attributes in a relation. The semantics indicate how attributes relate to one another, and specify the functional dependencies between attributes.

- When a functional dependency is present, the dependency is specified as a constraint between the attributes.

Functional Dependencies (cont.)

- Consider a relation with attributes A and B, where attribute B is functionally dependent on attribute A. If we know the value of A and we examine the relation that holds this dependency, we will find only one value of B in all of the tuples that have a given value of A, at any moment in time. Note however, that for a given value of B there may be several different values of A.
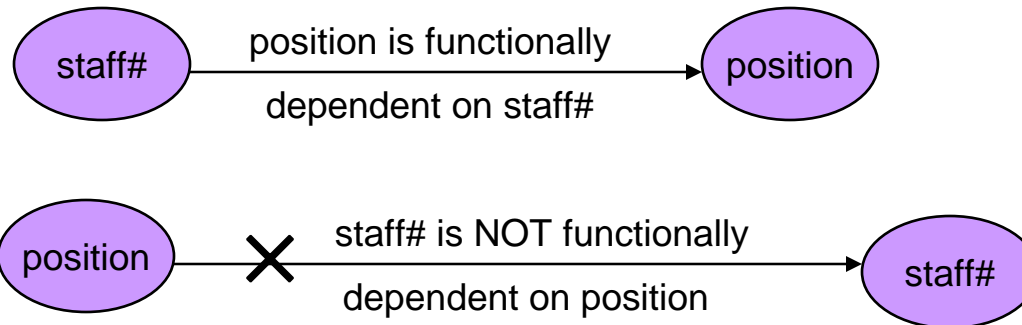
A — B is functionally dependent on A → B

- The determinant of a functional dependency is the attribute or group of attributes on the left-hand side of the arrow in the functional dependency. The consequent of a fd is the attribute or group of attributes on the right-hand side of the arrow.

  – In the figure above, A is the determinant of B and B is the consequent of A.

Identifying Functional Dependencies

- Look back at the staff relation on page 6. The functional dependency staff# → position clearly holds on this relation instance. However, the reverse functional dependency position → staff# clearly does not hold.

  – The relationship between staff# and position is 1:1 (from staff to position) – for each staff member there is only one position. On the other hand, the relationship between position and staff# is 1:M – there are several staff numbers associated with a given position.

  position is functionally
  staff# ──────────────────────→ position
  dependent on staff#

  staff# is NOT functionally
  position ──×──────────────────→ staff#
  dependent on position

- For the purposes of normalization we are interested in identifying functional dependencies between attributes of a relation that have a 1:1 relationship.

Identifying Functional Dependencies (cont.)

- When identifying fds between attributes in a relation it is important to distinguish clearly between the values held by an attribute at a given point in time and the *set of all possible values* that an attributes may hold at different times.

- In other words, a functional dependency is a property of a relational schema (its intension) and not a property of a particular instance of the schema (extension).

- The reason that we need to identify fds that hold for all possible values for attributes of a relation is that these represent the types of integrity constraints that we need to identify. Such constraints indicate the limitations on the values that a relation can legitimately assume. In other words, they identify the legal instances which are possible.

Identifying Functional Dependencies (cont.)

- Let's identify the functional dependencies that hold using the relation schema **staffbranch** shown on page 5 as an example.

- In order to identify the time invariant fds, we need to clearly understand the semantics of the various attributes in each of the relation schemas in question.

   – For example, if we know that a staff member's position and the branch at which they are located determines their salary. There is no way of knowing this constraint unless you are familiar with the enterprise, but this is what the requirements analysis phase and the conceptual design phase are all about!

   staff# $\rightarrow$ sname, position, salary, branch#, baddress

   branch# $\rightarrow$ baddress

   baddress $\rightarrow$ branch#

   branch#, position $\rightarrow$ salary

   baddress, position $\rightarrow$ salary

Identifying Functional Dependencies (cont.)

- It is common in many textbooks to use diagrammatic notation for displaying functional dependencies (this is how your textbook does it). An example of this is shown below using the relation schema staffbranch shown on page 5 for the fds we just identified as holding on the relational schema.
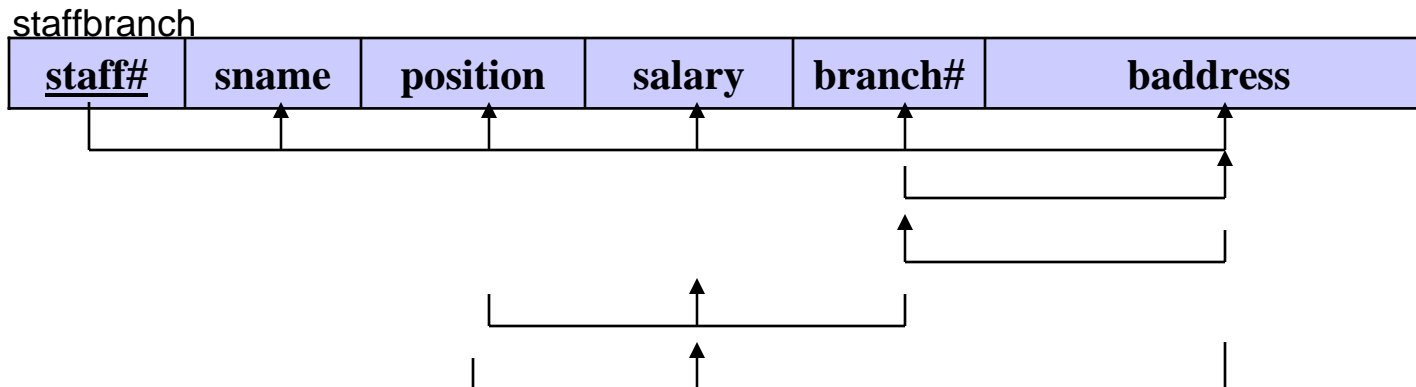
staff# → sname, position, salary, branch#, baddress
branch# → baddress
baddress → branch#
branch#, position → salary
baddress, position → salary

staffbranch

staff#	sname	position	salary	branch#	baddress

Trivial Functional Dependencies

- As well as identifying fds which hold for all possible values of the attributes involved in the fd, we also want to ignore trivial functional dependencies.

- A functional dependency is trivial iff, the consequent is a subset of the determinant. In other words, it is impossible for it *not* to be satisfied.

  - Example: Using the relation instances on page 6, the trivial dependencies include:

    $$\{ \text{staff\#, sname}\} \rightarrow \text{sname}$$

    $$\{ \text{staff\#, sname}\} \rightarrow \text{staff\#}$$

- Although trivial fds are valid, they offer no additional information about integrity constraints for the relation. As far as normalization is concerned, trivial fds are ignored.

Summary of FD Characteristics

- In summary, the main characteristics of functional dependencies that are useful in normalization are:

    1. There exists a 1:1 relationship from the attribute(s) in the determinant to the attribute(s) in the consequent.

    2. The functional dependency is time invariant, i.e., it holds in all possible instances of the relation.

    3. The functional dependencies are nontrivial. Trivial fds are ignored.

Inference Rules for Functional Dependencies

- We'll denote as F, the set of functional dependencies that are specified on a relational schema R.

- Typically, the schema designer specifies the fds that are *semantically obvious*; usually however, numerous other fds hold in all legal relation instances that satisfy the dependencies in F.

- These additional fds that hold are those fds which can be *inferred* or *deduced* from the fds in F.

- The set of all functional dependencies implied by a set of functional dependencies F is called the closure of F and is denoted F^{+}.

Inference Rules (cont.)

- The notation: $F \models X \rightarrow Y$ denotes that the functional dependency $X \rightarrow Y$ is implied by the set of fds F.

- Formally, $F^+ \equiv \{X \rightarrow Y \mid F \models X \rightarrow Y\}$

- A set of inference rules is required to infer the set of fds in F^+.

  - For example, if I tell you that Kristi is older than Debi and that Debi is older than Traci, you are able to infer that Kristi is older than Traci. How did you make this inference? Without thinking about it or maybe knowing about it, you utilized a transitivity rule to allow you to make this inference.

- The next page illustrates a set of six well-known inference rules that apply to functional dependencies.

Inference Rules (cont.)

IR1: reflexive rule – if $X \supseteq Y$, then $X \rightarrow Y$

IR2: augmentation rule – if $X \rightarrow Y$, then $XZ \rightarrow YZ$

IR3: transitive rule – if $X \rightarrow Y$ and $Y \rightarrow Z$, then $X \rightarrow Z$

IR4: projection rule – if $X \rightarrow YZ$, then $X \rightarrow Y$ and $X \rightarrow Z$

IR5: additive rule – if $X \rightarrow Y$ and $X \rightarrow Z$, then $X \rightarrow YZ$

IR6: pseudotransitive rule – if $X \rightarrow Y$ and $YZ \rightarrow W$, then $XZ \rightarrow W$

- The first three of these rules (IR1-IR3) are known as Armstrong's Axioms and constitute a necessary and sufficient set of inference rules for generating the closure of a set of functional dependencies.

Example Proof Using Inference Rules

- Given R = (A,B,C,D,E,F,G,H, I, J) and
  F = {AB → E, AG → J, BE → I, E → G, GI → H}
  does F ⊨ AB → GH?

<u>Proof</u>

1.   AB → E, given in F
2.   AB → AB, reflexive rule IR1
3.   AB → B, projective rule IR4 from step 2
4.   AB → BE, additive rule IR5 from steps 1 and 3
5.   BE → I, given in F
6.   AB → I, transitive rule IR3 from steps 4 and 5
7.   E → G, given in F
8.   AB → G, transitive rule IR3 from steps 1 and 7
9.   AB → GI, additive rule IR5 from steps 6 and 8
10. GI → H, given in F
11. AB → H, transitive rule IR3 from steps 9 and 10
12. AB → GH, additive rule IR5 from steps 8 and 11 - proven

<u>Practice Problem</u>

Using the same set F, prove that F ⊨ BE → H

Answer: on next page/

Proof For Practice Problem

- Given R = (A,B,C,D,E,F,G,H, I, J) and
  F = {AB → E, AG → J, BE → I, E → G, GI → H}
  does F ⊨ BE → H?

Proof

1.   BE → I, given in F
2.   BE → BE, reflexive rule IR1
3.   BE → E, projective rule IR4 from step 2
4.   E → G, given
5.   BE → G, transitive rule IR3 from steps 3 and 4
6.   BE → GI, additive rule IR5 from steps 1 and 5
7.   GI → H, given in F
8.   BE → H, transitive rule IR3 from steps 6 and 7 - proven

Determining Closures

- Another way of looking at the closure of a set of fds F is: F^+ is the smallest set containing F such that Armstrong's Axioms cannot be applied to the set to yield an fd not in the set.

- F^+ is finite, but exponential in size in terms of the number of attributes of R.

    - For example, given R=(A,B,C) and F = {AB $\rightarrow$ C, C $\rightarrow$ B}, F^+ will contain 29 fds (including trivial fds).

- Thus, to determine if a fd X $\rightarrow$ Y holds on a relation schema R given F, what we really need to determine is does F $\vDash$ X $\rightarrow$ Y, or more correctly is X$\rightarrow$Y in F^+? However, we want to do this without generating all of F^+ and checking to see if X$\rightarrow$Y is in that set.

Determining Closures (cont.)

- The technique for this is to generate not F^+ but rather X^+, where X is any determinant from a fd in F. An algorithm for generating X^+ is shown below.

- X^+ is called the closure of X under F (or with respect to F).

Algorithm Closure

```
Algorithm Closure  {returns X+ under F}
input:  set of attributes X, and a set of fds F
output: X+ under F
Closure (X, F)
{
    X+ ← X;
    repeat
        oldX+ ← X+;
        for every fd W→ Z in F do
            if W ⊆ X+ then X+ ← X+ ∪ Z;
    until (oldX+ = X+);
}
```

Example Using Algorithm Closure

$$\text{Given } F = \{A \rightarrow D, AB \rightarrow E, BI \rightarrow E, CD \rightarrow I, E \rightarrow C\},$$
$$\text{Find } (AE)^+$$

pass 1

$X^+ = \{A, E\}$
using $A \rightarrow D$, $A \subseteq X^+$, so add D to X^+, $X^+ = \{A, E, D\}$
using $AB \rightarrow E$, no
using $BI \rightarrow E$, no
using $CD \rightarrow I$, no
using $E \rightarrow C$, $E \subseteq X^+$, so add C to X^+, $X^+ = \{A, E, D, C\}$
changes occurred to X^+ so another pass is required

pass 2

$X^+ = \{A, E, D, C\}$
using $A \rightarrow D$, yes, but no changes
using $AB \rightarrow E$, no
using $BI \rightarrow E$, no
using $CD \rightarrow I$, $CD \subseteq X^+$, so add I to X^+, $X^+ = \{A, E, D, C, I\}$
using $E \rightarrow C$, yes, but no changes
changes occurred to X^+ so another pass is required

Example Using Algorithm Closure Continues

pass 3

$X^+ = \{A, E, D, C, I\}$
using $A \rightarrow D$, yes, but no changes
using $AB \rightarrow E$, no
using $BI \rightarrow E$, no
using $CD \rightarrow I$, yes, but no changes
using $E \rightarrow C$, yes, but no changes
no changes occurred to X^+ so algorithm terminates

$(AE)^+ = \{A, E, C, D, I\}$

This means that the following fds are in F^+: $AE \rightarrow AECDI$

Algorithm Member

- Once the closure of a set of attributes X has been generated, it becomes a simple test to tell whether or not a certain functional dependency with a determinant of X is included in F^+.

- The algorithm shown below will determine if a given set of fds implies a specific fd.

Algorithm Member

```
Algorithm Member  {determines membership in F+}
input:  a set of fds F, and a single fd X → Y
output: true if F ⊨ X → Y, false otherwise
Member (F, X → Y)
{
    if Y ⊆ Closure(X,F)
        then return true;
        else return false;
}
```

Covers and Equivalence of Sets of FDs

- A set of fds F is covered by a set of fds F (alternatively stated as G covers F) if every fd in G is also in F^+.

    - That is to say, F is covered if every fd in F can be inferred from G.

- Two sets of fds F and G are equivalent if $F^+ = G^+$.

    - That is to say, every fd in G can be inferred from F and every fd in F can be inferred from G.

    - Thus $F \equiv G$ if F covers G and G covers F.

- To determine if G covers F, calculate X^+ wrt G for each $X \rightarrow Y$ in F. If $Y \subseteq X^+$ for each X, then G covers F.

Why Covers?

- Algorithm Member has a run time which is dependent on the size of the set of fds used as input to the algorithm. Thus, the smaller the set of fds used, the faster the execution of the algorithm.

- Fewer fds require less storage space and thus a corresponding lower overhead for maintenance whenever database updates occur.

- There are many different types of covers ranging from non-redundant covers to optimal covers. We won't look at all of them.

- Essentially the idea is to ultimately produce a set of fds G which is equivalent to the original set F, yet has as few total fds (symbols in the extreme case) as possible.

Non-redundant Covers

- A set of fds is non-redundant if there is no proper subset G of F with $G \equiv F$. If such a G exists, F is redundant.

- F is a non-redundant cover for G if F is a cover for G and F is non-redundant.

Algorithm Nonredundant

Algorithm Nonredundant {produces a non-redundant cover}
input: a set of fds G
output: a nonredundant cover for G
Nonredundant (G)
{
    $F \leftarrow G$;
    for each fd $X \rightarrow Y \in G$ do
        if Member(F $-$ {$X \rightarrow Y$}, $X \rightarrow Y$)
            then $F \leftarrow F - \{X \rightarrow Y\}$;
      return (F);
}

Example: Producing a Non-redundant Cover

Let G = {A → B, B → A, B → C, A → C}, find a non-redundant cover for G.

F ← G

Member({B → A, B → C, A → C}, A → B)

    Closure(A, {B → A, B → C, A → C})

       A^+ = {A, C}, therefore A → B is not redundant

Member({A → B, B → C, A → C}, B → A)

    Closure(B, {A → B, B → C, A → C})

       B^+ = {B, C}, therefore B → A is not redundant

Member({A → B, B → A, A → C}, B → C)

    Closure(B, {A → B, B → A, A → C})

       B^+ = {B, A, C}, therefore B → C is redundant F = F − {B → C}

Member({A → B, B → A}, A → C)

    Closure(A, {A → B, B → A})

       A^+ = {A, B}, therefore A → C is not redundant

Return F = {A → B, B → A, A → C}

Example 2: Producing a Non-redundant Cover

If $G = \{A \rightarrow B, A \rightarrow C, B \rightarrow A, B \rightarrow C\}$, the same set as before but given in a different order. A different cover will be produced!

$F \leftarrow G$

Member($\{A \rightarrow C, B \rightarrow A, B \rightarrow C\}, A \rightarrow B$)

    Closure($A, \{A \rightarrow C, B \rightarrow A, B \rightarrow C\}$)

        $A^+ = \{A, C\}$, therefore $A \rightarrow B$ is not redundant

Member($\{A \rightarrow B, B \rightarrow A, B \rightarrow C\}, A \rightarrow C$)

    Closure($A, \{A \rightarrow B, B \rightarrow A, B \rightarrow C\}$)

        $A^+ = \{A, B, C\}$, therefore $A \rightarrow C$ is redundant $F = F - \{A \rightarrow C\}$

Member($\{A \rightarrow B, B \rightarrow C\}, B \rightarrow A$)

    Closure($B, \{A \rightarrow B, B \rightarrow C\}$)

        $B^+ = \{B, C\}$, therefore $B \rightarrow A$ is not redundant

Member($\{A \rightarrow B, B \rightarrow A\}, B \rightarrow C$)

    Closure($B, \{A \rightarrow B, B \rightarrow A\}$)

        $B^+ = \{B, A\}$, therefore $B \rightarrow C$ is not redundant

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

Non-redundant Covers (cont.)

- The previous example illustrates that a given set of functional dependencies can contain more than one non-redundant cover.

- It is also possible that there can be non-redundant covers for a set of fds G that are not contained in G.

  - For example, if

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

    then $F = \{A \rightarrow B, B \rightarrow A, AB \rightarrow C\}$ is a non-redundant cover for G

    however, F contains fds that are not in G.

Extraneous Attributes

- If F is a non-redundant set of fds, this means that there are no "extra" fds in F and thus F cannot be made smaller by removing fds. If fds are removed from F then a set G would be produced where $G \not\equiv F$.

- However, it may still be possible to reduce the overall size of F by removing attributes from fds in F.

- If F is a set of fds over relation schema R and $X \rightarrow Y \in F$, then attribute A is extraneous in $X \rightarrow Y$ wrt F if:

  1. $X = AZ, X \neq Z$ and $\{F - \{X \rightarrow Y\}\} \cup \{Z \rightarrow Y\} \equiv F$, or

  2. $Y = AW, Y \neq W$ and $\{F - \{X \rightarrow Y\}\} \cup \{X \rightarrow W\} \equiv F$

- In other words, an attribute A is extraneous in $X \rightarrow Y$ if A can be removed from either the determinant or consequent without changing F^+.

Extraneous Attributes (cont.)

Example:

let $F = \{A \to BC, B \to C, AB \to D\}$

attribute C is extraneous in the consequent of $A \to BC$ since $A^+ = \{A, B, C, D\}$ when $F = F - \{A \to C\}$

similarly, B is extraneous in the determinant of $AB \to D$ since $AB^+ = \{A, B, C, D\}$ when $F = F - \{AB \to D\}$

Left and Right Reduced Sets of FDs

- Let F be a set of fds over schema R and let $X \rightarrow Y \in F$.

  $X \rightarrow Y$ is left-reduced if X contains no extraneous attribute A.

  - A left-reduced functional dependency is also called a full functional dependency.

  $X \rightarrow Y$ is right-reduced if Y contains no extraneous attribute A.

  $X \rightarrow Y$ is reduced if it is left-reduced, right-reduced, and Y is not empty.

Algorithm Left-Reduce

- The algorithm below produces a left-reduced set of functional dependencies.

Algorithm Left-Reduce

Algorithm Left-Reduce {returns left-reduced version of F}
input: set of fds G
output: a left-reduced cover for G
Left-Reduce (G)
{
    F ← G;
    for each fd X→ Y in G do
        for each attribute A in X do
            if Member(F, (X-A) → Y)
                then remove A from X in X→ Y in F
    return(F);
}

Algorithm Right-Reduce

- The algorithm below produces a right-reduced set of functional dependencies.

Algorithm Right-Reduce

Algorithm Right-Reduce {returns right-reduced version of F}
input: set of fds G
output: a right-reduced cover for G
Right-Reduce (G)
{
    $F \leftarrow G$;
    for each fd $X \rightarrow Y$ in G do
        for each attribute A in Y do
            if Member($F - \{X \rightarrow Y\} \cup \{X \rightarrow (Y - A)\}, X \rightarrow A$)
                then remove A from Y in $X \rightarrow Y$ in F
    return(F);
}

Algorithm Reduce

- The algorithm below produces a reduced set of functional dependencies.

Algorithm Reduce

```
Algorithm Reduce  {returns reduced version of F}
input:  set of fds G
output: a reduced cover for G
Reduce (G)
{
    F ← Right-Reduce( Left-Reduce(G));
    remove all fds of the form X→ null from F
    return(F);
}
```

If G contained a redundant fd, X→ Y, every attribute in Y would be extraneous and thus reduce to X → null, so these need to be removed.

Algorithm Reduce (cont.)

- The order in which the reduction is done by algorithm Reduce is important. The set of fds must be left-reduced first and then right-reduced. The example below illustrates what may happen if this order is violated.

Example:

Let $G = \{B \rightarrow A , D \rightarrow A , BA \rightarrow D\}$

G is right-reduced but not left-reduced. If we left-reduce

G to produce $F = \{B \rightarrow A , D \rightarrow A , B \rightarrow D\}$

We have F is left-reduced but not right-reduced!

$B \rightarrow A$ is extraneous on right side since $B \rightarrow D \rightarrow A$

Canonical Cover

- A set of functional dependencies F is canonical if every fd in F is of the form $X \rightarrow A$ and F is left-reduced and non-redundant.

Example:

$G = \{A \rightarrow BCE, \ AB \rightarrow DE, \ BI \rightarrow J\}$

a canonical cover for G is:

$F = \{A \rightarrow B, A \rightarrow C, A \rightarrow D, A \rightarrow E, BI \rightarrow J\}$

Minimum Cover

- A set of functional dependencies F is minimal if

    1. Every fd has a single attribute for its consequent.

    2. F is non-redundant.

    3. No fd $X \rightarrow A$ can be replaced with one of the form $Y \rightarrow A$ where $Y \subseteq X$ and still be an equivalent set, i.e., F is left-reduced.

Example:

$$G = \{A \rightarrow BCE, \ AB \rightarrow DE, \ BI \rightarrow J\}$$

a minimal cover for G is:

$$F = \{A \rightarrow B, A \rightarrow C, A \rightarrow D, A \rightarrow E, BI \rightarrow J\}$$

Algorithm MinCover

- The algorithm below produces a minimal cover for a set of functional dependencies.

Algorithm MinCover

```
Algorithm MinCover  {returns minimum cover for F}
input:  set of fds F
output: a minimum cover for F
MinCover (F)
{
    G ← F;
    replace each fd X → A₁A₂...Aₙ in G by n fds X → A₁, X → A₂,..., X → Aₙ
    for each fd X → A in G do
        if Member( G– {X → A}, X → A )
            then G ← G – {X → A}
    endfor
    for each remaining fd in G, X → A do
        for each attribute B ∈ X do
            if Member( [{G– {X → A}} ∪ {(X–B) → A}], (X–B) → A)
                then G ← {G– {X → A}} ∪ {(X–B) → A}
     endfor
     return(G);
}
```

Determining the Keys of a Relation Schema

- If R is a relational schema with attributes $A_1, A_2, ..., A_n$ and a set of functional dependencies F where $X \subseteq \{A_1, A_2, ..., A_n\}$ then X is a key of R if:

  1. $X \rightarrow A_1 A_2 ... A_n \in F^+$, and

  2. no proper subset $Y \subseteq X$ gives $Y \rightarrow A_1 A_2 ... A_n \in F^+$.

- Basically, this definition means that you must attempt to generate the closure of all possible subsets of the schema of R and determine which sets produce all of the attributes in the schema.

Determining Keys - Example

Let $r = (C, T, H, R, S, G)$ with

$F = \{C \rightarrow T, HR \rightarrow C, HT \rightarrow R, CS \rightarrow G, HS \rightarrow R\}$

Step 1: Generate $(A_i)^+$ for $1 \leq i \leq n$

$\binom{6}{1} = \frac{6!}{1! \times (6-1)!} = \frac{720}{120} = 6$

$C^+ = \{CT\}, \quad T^+ = \{T\}, \quad H^+ = \{H\}$

$R^+ = \{R\}, \qquad S^+ = \{S\}, \quad G^+ = \{G\}$

no single attribute is a key for R

$\binom{6}{2} = \frac{6!}{2! \times (6-2)!} = \frac{720}{48} = 15$

Step 2: Generate $(A_iA_j)^+$ for $1 \leq i \leq n, 1 \leq j \leq n$

$(CT)^+ = \{C,T\}, \quad (CH)^+ = \{CHTR\}, \quad (CR)^+ = \{CRT\}$

$(CS)^+ = \{CSGT\}, \quad (CG)^+ = \{CGT\}, \quad (TH)^+ = \{THRC\}$

$(TR)^+ = \{TR\}, \quad (TS)^+ = \{TS\}, (TG)^+ = \{TG\}$

$(HR)^+ = \{HRCT\}, \quad (HS)^+ = \{HSRCTG\}, \quad (HG)^+ = \{HG\}$

$(RS)^+ = \{RS\}, \quad (RG)^+ = \{RG\}, \quad (SG)^+ = \{SG\}$

The attribute set (HS) is a key for R

Determining Keys - Example

Step 3: Generate $(A_i A_j A_k)^+$ for $1 \le i \le n$, $1 \le j \le n$, $1 \le k \le n$

$(CTH)^+ = \{CTHR\}$, $(CTR)^+ = \{CTR\}$

$(CTS)^+ = \{CTSG\}$, $(CTG)^+ = \{CTG\}$

$$\binom{6}{3} = \frac{6!}{3! \times (6-3)!} = \frac{720}{36} = 20$$

$(CHR)^+ = \{CHRT\}$, $(CHS)^+ = \{CHSTRG\}$

$(CHG)^+ = \{CHGTR\}$, $(CRS)^+ = \{CRSTG\}$

$(CRG)^+ = \{CRGT\}$, $(CSG)^+ = \{CSGT\}$

$(THR)^+ = \{THRC\}$, $(THS)^+ = \{THSRCG\}$

$(THG)^+ = \{THGRC\}$, $(TRS)^+ = \{TRS\}$

$(TRG)^+ = \{TRG\}$, $(TSG)^+ = \{TSG\}$

$(HRS)^+ = \{HRSCTG\}$, $(HRG)^+ = \{HRGCT\}$

$(HSG)^+ = \{HSGRCT\}$, $(RSG)^+ = \{RSG\}$

Superkeys are shown in red.

Determining Keys - Example

Step 4: Generate $(A_iA_jA_kA_r)^+$ for $1 \leq i \leq n$, $1 \leq j \leq n$, $1 \leq k \leq n$, $1 \leq r \leq n$

$$\binom{6}{4} = \frac{6!}{4! \times (6-4)!} = \frac{720}{48} = 15$$

$(CTHR)^+ = \{CTHR\}$, $(CTHS)^+ = \{CTHSRG\}$

$(CTHG)^+ = \{CTHGR\}$, $(CHRS)^+ = \{CHRSTG\}$

$(CHRG)^+ = \{CHRGT\}$, $(CRSG)^+ = \{CRSGT\}$

$(THRS)^+ = \{THRSCG\}$, $(THRG)^+ = \{THRGC\}$

$(TRSG)^+ = \{TRSG\}$, $(HRSG)^+ = \{HRSGCT\}$

$(CTRS)^+ = \{CTRS\}$, $(CTSG)^+ = \{CTSG\}$

$(CSHG)^+ = \{CSHGTR\}$, $(THSG)^+ = \{THSGRC\}$

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

Superkeys are shown in red.

Determining Keys - Example

Step 5: Generate $(A_iA_jA_kA_rA_s)^+$ for $1 \leq i \leq n$, $1 \leq j \leq n$, $1 \leq k \leq n$, $1 \leq r \leq n$, $1 \leq s \leq n$

$$\binom{6}{5} = \frac{6!}{5! \times (6-5)!} = \frac{720}{120} = 6$$

$(CTHRS)^+ = \{CTHSRG\}$
$(CTHRG)^+ = \{CTHGR\}$
$(CTHSG)^+ = \{CTHSGR\}$
$(CHRSG)^+ = \{CHRSGT\}$
$(CTRSG)^+ = \{CTRSG\}$
$(THRSG)^+ = \{THRSGC\}$

Superkeys are shown in red.

Determining Keys - Example

Step 6: Generate $(A_iA_jA_kA_rA_sA_t)^+$ for $1 \le i \le n$, $1 \le j \le n$, $1 \le k \le n$, $1 \le r \le n$, $1 \le s \le n$, $1 \le t \le n$

$$\binom{6}{6} = \frac{6!}{6! \times (6-6)!} = \frac{720}{720} = 1$$

$(CTHRSG)^+ = \{CTHSRG\}$

Superkeys are shown in red.

- In general, for 6 attributes we have:

$$\binom{6}{1} + \binom{6}{2} + \binom{6}{3} + \binom{6}{4} + \binom{6}{5} + \binom{6}{6} = 6 + 15 + 20 + 15 + 1 = 63 \text{ cases}$$

Practice Problem: Find all the keys of R = (A,B,C,D) given F = {A$\rightarrow$B, B$\rightarrow$C}

Normalization Based on the Primary Key

- Normalization is a formal technique for analyzing relations based on the primary key (or candidate key attributes and functional dependencies.

- The technique involves a series of rules that can be used to test individual relations so that a database can be normalized to any degree..

- When a requirement is not met, the relation violating the requirement is decomposed into a set of relations that individually meet the requirements of normalization.

- Normalization is often executed as a series of steps. Each step corresponds to a specific normal form that has known properties.

Relationship Between Normal Forms

N1NF

1NF

2NF

3NF

BCNF

4NF

5NF

Higher Normal Forms

The Process Of Normalization

Table with multi-valued attributes N1NF

Remove multi-valued attributes

1NF

Remove partial dependencies

2NF

Remove transitive dependencies

3NF

Remove remaining anomalies from FDs

BCNF

Remove multi-valued dependencies

4NF

Remove remaining anomalies from MVDs

5NF
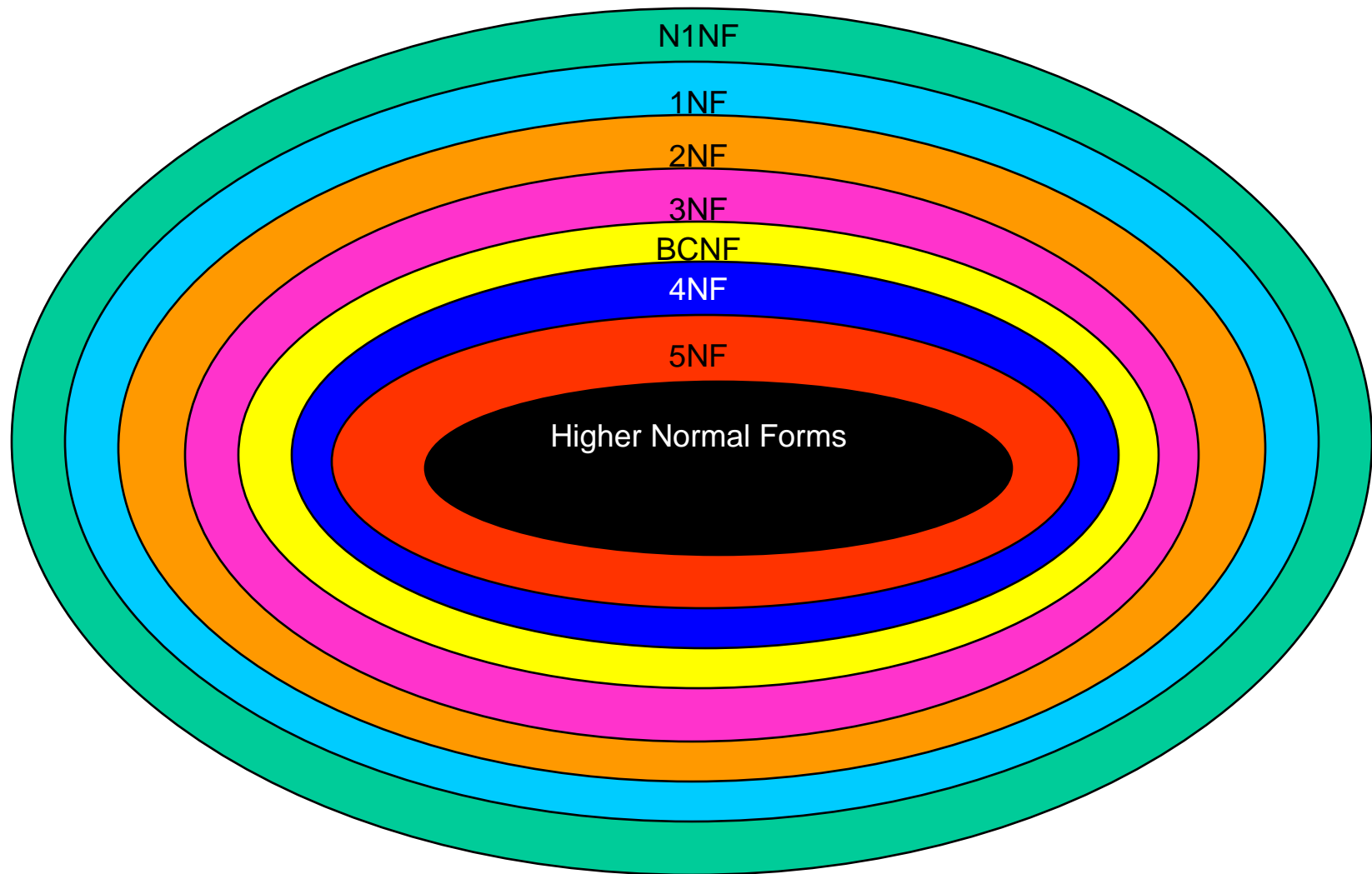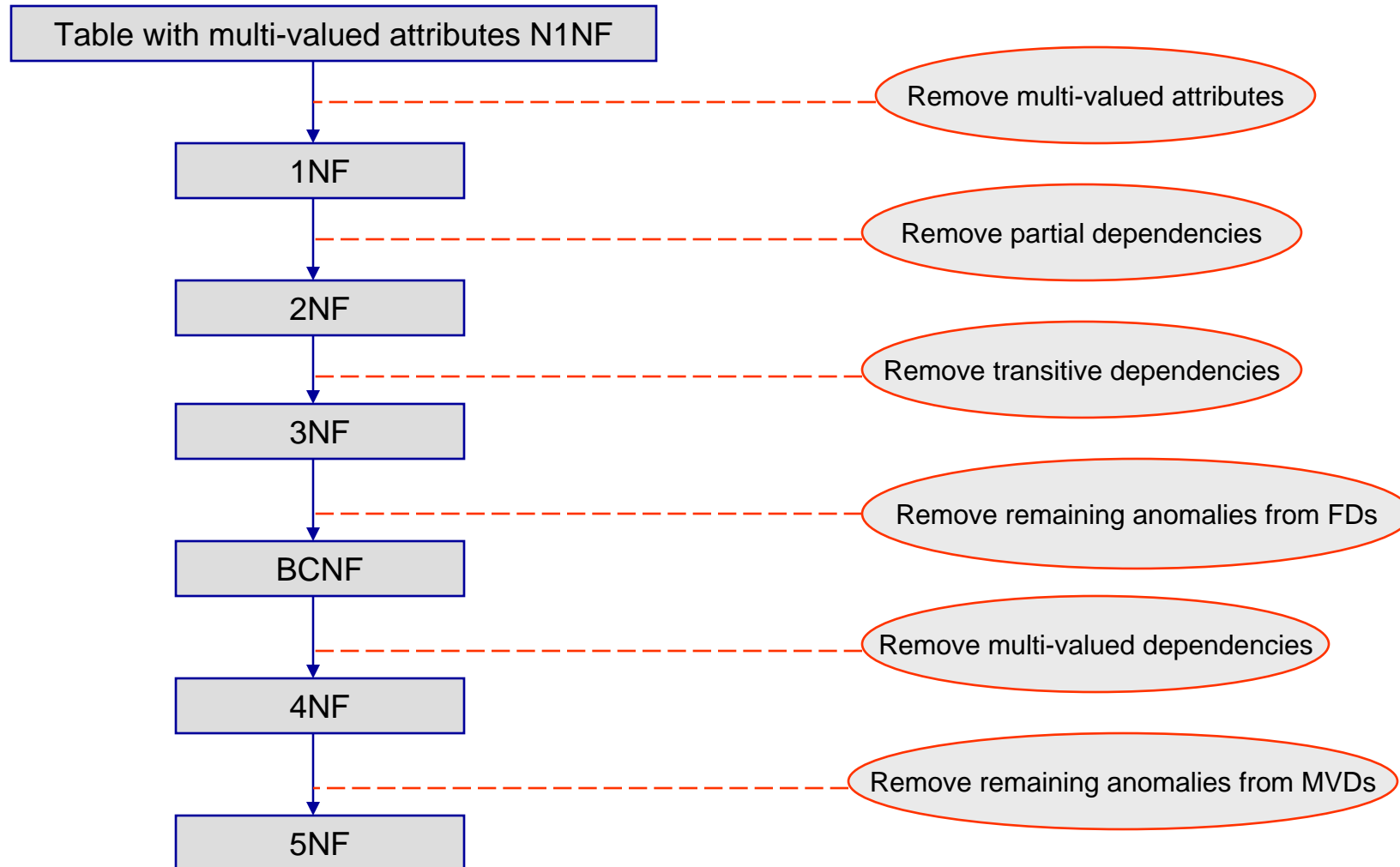
Normalization Requirements

- For the relational model it is important to recognize that it is only first normal form (1NF) that is critical in creating relations. All the subsequent normal forms are optional.

- However, to avoid the update anomalies that we discussed earlier, it is normally recommended that the database designer proceed to at least 3NF.

- As the figure on the previous page illustrates, some 1NF relations are also in 2NF and some 2NF relations are also in 3NF, and so on.

- As we proceed, we'll look at the requirements for each normal form and a decomposition technique to achieve relation schemas in that normal form.

Non-First Normal Form (N1NF)

- Non-first normal form relation are those relations in which one or more of the attributes are non-atomic. In other words, within a relation and within a single tuple there is a multi-valued attribute.

- There are several important extensions to the relational model in which N1NF relations are utilized. For the most part these go beyond the scope of this course and we will not discuss them in any significant detail. Temporal relational databases and certain categories of spatial databases fall into the N1NF category.

First Normal Form (1NF)

- A relation in which every attribute value is atomic is in 1NF.

- We have only considered 1NF relations for the most part in this course.

- When dealing with multi-valued attributes at the conceptual level, recall that in the conversion into the relational model created a separate table for the multi-valued attribute. (See Chapter 3 Notes, Pages 19-21)

Some Additional Terminology

- A key is a superkey with the additional property that the removal of any attribute from the key will cause it to no longer be a superkey. In other words, the key is minimal in the number of attributes.

- The candidate key for a relation a set of minimal keys of the relation schema.

- The primary key for a relation is a selected candidate key. All of the remaining candidate keys (if any) become secondary keys.

- A prime attribute is any attribute of the schema of a relation R that is a member of any candidate key of R.

- A non-prime attribute is any attribute of R which is not a member of any candidate key.

Second Normal Form (2NF)

- Second normal form (2NF) is based on the concept of a full functional dependency.

- A functional dependency $X \rightarrow Y$ is a full functional dependency if the removal of any attribute A from X causes the fd to no longer hold.

    for any attribute $A \in X$, $X - \{A\} \not\rightarrow Y$

- A functional dependency $X \rightarrow Y$ is a partial functional dependency if some attribute A can be removed from X and the fd still holds.

    for any attribute $A \in X$, $X - \{A\} \rightarrow Y$

Definition of Second Normal Form (2NF)

- A relation scheme R is in 2NF with respect to a set of functional dependencies F if every non-prime attribute is fully dependent on every key of R.

- Another way of stating this is: there does not exist a non-prime attribute which is partially dependent on any key of R. In other words, no non-prime attribute is dependent on only a portion of the key of R.

Example of Second Normal Form (2NF)

Given $R = (A, D, P, G)$, $F = \{AD \rightarrow PG, A \rightarrow G\}$ and

$$K = \{AD\}$$

Then R is not in 2NF because G is partially dependent on the key AD since $AD \rightarrow G$ yet $A \rightarrow G$.

Decompose R into:

$R1 = (A, D, P)$ $R2 = (A, G)$

$K1 = \{AD\}$ $K2 = \{A\}$

$F1 = \{AD \rightarrow P\}$ $F2 = \{A \rightarrow G\}$

Third Normal Form (3NF)

- Third Normal Form (3NF) is based on the concept of a transitive dependency.

- Given a relation scheme R with a set of functional dependencies F and subset $X \subseteq R$ and an attribute $A \in R$. A is said to be transitively dependent on X if there exists $Y \subseteq R$ with $X \to Y$, $Y\ X \not\to X$ and $Y \to A$ and $A \notin X \cup Y$.

- An alternative definition for a transitive dependency is: a functional dependency $X \to Y$ in a relation scheme R is a transitive dependency if there is a set of attributes $Z \subseteq R$ where Z is not a subset of any key of R and yet both $X \to Z$ and $Z \to Y$ hold in F.

Third Normal Form (3NF) (cont.)

- A relation scheme R is in 3NF with respect to a set of functional dependencies F, if whenever $X \rightarrow A$ holds either: (1) X is a superkey of R or (2) A is a prime attribute.

- Alternative definition: A relation scheme R is in 3NF with respect to a set of functional dependencies F if no non-prime attribute is transitively dependent on any key of R.

Example: Let R = (A, B, C, D)

$$K = \{AB\}, F = \{AB \rightarrow CD, \; C \rightarrow D, \; D \rightarrow C\}$$

then R is not in 3NF since $C \rightarrow D$ holds and C is not a superkey of R.

Alternatively, R is not in 3NF since $AB \rightarrow C$ and $C \rightarrow D$ and thus D is a non-prime attribute which is transitively dependent on the key AB.

Why Third Normal Form?

- What does 3NF do for us? Consider the following database:

  assign(flight, day, pilot-id, pilot-name)
  K = {flight day}
  F = {pilot-id $\rightarrow$ pilot-name, pilot-name $\rightarrow$ pilot-id}

flight	day	pilot-id	pilot-name
112	Feb.11	317	Mark
112	Feb. 12	246	Kristi
114	Feb.13	317	Mark

Why Third Normal Form? (cont.)

flight	day	pilot-id	pilot-name
112	Feb.11	317	Mark
112	Feb. 12	246	Kristi
114	Feb.13	317	Mark
112	Feb. 11	319	Mark

Since {flight day} is key, clearly {flight day} $\rightarrow$ pilot-name.
But in F we also know that pilot-name $\rightarrow$ pilot-id, and
we have that {flight day} $\rightarrow$ pilot-id.

Now suppose the highlighted tuple is added to this instance.
is added. The fd pilot-name $\rightarrow$ pilot-id is violated by this
insertion. A transitive dependency exists since: pilot-id $\rightarrow$
pilot-name holds and pilot-id is not a superkey.

Boyce-Codd Normal Form (BCNF)

- Boyce-Codd Normal Form (BCNF) is a more stringent form of 3NF.

- A relation scheme R is in Boyce-Codd Normal Form with respect to a set of functional dependencies F if whenever $X \rightarrow A$ hold and $A \not\subseteq X$, then X is a superkey of R.

Example: Let $R = (A, B, C)$

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

$$K = \{AB\}$$

R is not in BCNF since $C \rightarrow A$ holds and C is not a superkey of R.

Boyce-Codd Normal Form (BCNF) (cont.)

- Notice that the only difference in the definitions of 3NF and BCNF is that BCNF drops the allowance for A in X $\rightarrow$ A to be prime.

- An interesting side note to BCNF is that Boyce and Codd originally intended this normal form to be a simpler form of 3NF. In other words, it was supposed to be between 2NF and 3NF. However, it was quickly proven to be a more strict definition of 3NF and thus it wound up being between 3NF and 4NF.

- In practice, most relational schemes that are in 3NF are also in BCNF. Only if X $\rightarrow$ A holds in the schema where X is not a superkey and A is prime, will the schema be in 3NF but not in BCNF.

Moving Towards Relational Decomposition

- The basic goal of relational database design should be to ensure that every relation in the database is either in 3NF or BCNF.

- 1NF and 2NF do not remove a sufficient number of the update anomalies to make a significant difference, whereas 3NF and BCNF eliminate most of the update anomalies.

- As we've mentioned before, in addition to ensuring the relation schemas are in either 3NF or BCNF, the designer must also ensure that the decomposition of the original database schema into the 3NF or BCNF schemas guarantees that the decomposition have (1) the lossless join property (also called a non-additive join property) and (2) the functional dependencies are preserved across the decomposition.

Moving Towards Relational Decomposition (cont.)

- There are decomposition algorithms that will guarantee a 3NF decomposition which ensures both the lossless join property and preservation of the functional dependencies.

- However, there is no algorithm which will guarantee a BCNF decomposition which ensures both the lossless join property and preserves the functional dependencies. There is an algorithm that will guarantee BCNF and the lossless join property, but this algorithm cannot guarantee that the dependencies will be preserved.

- It is for this reason that many times, 3NF is as strong a normal form as will be possible for a certain set of schemas, since an attempt to force BCNF may result in the non-preservation of the dependencies.

- In the next few pages we'll look at these two properties more closely.

Preservation of the Functional Dependencies

- Whenever an update is made to the database, the DBMS must be able to verify that the update will not result in an illegal instance with respect to the functional dependencies in F^+.

- To check updates in an efficient manner the database must be designed with a set of schemas which allows for this verification to occur without necessitating join operations.

- If an fd is "lost", the only way to enforce the constraint would be to effect a join of two or more relations in the decomposition to get a "relation" that includes all of the determinant and consequent attributes of the lost fd into a single table, then verify that the dependency still holds after the update occurs. Obviously, this requires too much effort to be practical or efficient.

Preservation of the Functional Dependencies (cont.)

- Informally, the preservation of the dependencies means that if $X \rightarrow Y$ from F appears either explicitly in one of the relational schemas in the decomposition scheme or can be inferred from the dependencies that appear in some relational schema within the decomposition scheme, then the original set of dependencies would be preserved on the decomposition scheme.

- It is important to note that what is required to preserve the dependencies is not that every fd in F be explicitly present in some relation schema in the decomposition, but rather the union of all the dependencies that hold on all of the individual relation schemas in the decomposition be equivalent to F (recall what equivalency means in this context).

Preservation of the Functional Dependencies (cont.)

- The projection of a set of functional dependencies onto a set of attributes Z, denoted F[Z] (also sometime as $\pi_Z(F)$), is the set of functional dependencies $X \rightarrow Y$ in F^+ such that $X \cup Y \subseteq Z$.

- A decomposition scheme $\gamma = \{R_1, R_2, \ldots, R_m\}$ is dependency preserving with respect to a set of fds F if the union of the projection of F onto each R_i ($1 \leq i \leq m$) in γ is equivalent to F.

  $$(F[R_1] \cup F[R_2] \cup \ldots \cup F[R_m])^+ = F^+$$

Preservation of the Functional Dependencies (cont.)

- It is **always** possible to find a dependency preserving decomposition scheme D with respect to a set of fds F such that each relation schema in D is in 3NF.

- In a few pages, we will see an algorithm that guarantees a 3NF decomposition in which the dependencies are preserved.

Algorithm for Testing the Preservation of Dependencies

Algorithm Preserve

// input: a decomposition D= $(R_1, R_2, \ldots, R_k)$, a set of fds F, an fd $X \rightarrow Y$

// output: true if D preserves F, false otherwise

Preserve $(D, F, X \rightarrow Y)$

  $Z = X$;

  while (changes to Z occur) do

      for i = 1 to k do // there are k schemas in D

         $Z = Z \cup ((Z \cap R_i)^+ \cap R_i)$

      endfor;

   endwhile;

  if $Y \subseteq Z$

     then return true; // $Z \vDash X \rightarrow Y$

     else return false;

end.

How Algorithm Preserves Works

- The set Z which is computed is basically the following: $G = \bigcup_{i=1}^{k} F[R_i]$

- Note that G is not actually computed but merely tested to see if G covers F. To test if G covers F we need to consider each fd $X \to Y$ in F and determine if X_G^+ contains Y.

- Thus, the technique is to compute X_G^+ without having G available by repeatedly considering the effect of closing F with respect to the projections of F onto the various R_i.

A Hugmongously Big Example

Let R = (A, B, C, D)

$\quad$ F = {A→B, B→C, C→D, D→A}

$\quad\quad$ D = {(AB), (BC), (CD)}

G = F[AB] $\cup$ F[BC] $\cup$ F[CD] $\qquad$ Z = Z $\cup$ ((Z $\cap$ R$_i$)$^+$ $\cap$ R$_i$)

Test for each fd in F.

Test for A→B

$\quad\quad$ Z = A,

$\quad\quad\quad$ = {A} $\cup$ ((A $\cap$ AB)$^+$ $\cap$ AB)

$\quad\quad\quad$ = {A} $\cup$ ((A)$^+$ $\cap$ AB)

$\quad\quad\quad$ = {A} $\cup$ (ABCD $\cap$ AB)

$\quad\quad\quad$ = {A} $\cup$ {AB}

$\quad\quad\quad$ = **{AB}**

A Hugmongously Big Example (cont.)

$Z = \{AB\}$
$= \{AB\} \cup ((AB \cap BC)^+ \cap BC)$
$= \{AB\} \cup ((B)^+ \cap BC)$
$= \{AB\} \cup (BCDA \cap BC)$
$= \{AB\} \cup \{BC\}$
$= \textbf{\{ABC\}}$

$Z = \{ABC\}$
$= \{ABC\} \cup ((ABC \cap CD)^+ \cap CD)$
$= \{ABC\} \cup ((C)^+ \cap CD)$
$= \{ABC\} \cup (CDAB \cap CD)$
$= \{ABC\} \cup \{CD\}$
$= \textbf{\{ABCD\}}$

G covers A $\rightarrow$ B

A Hugmongously Big Example (cont.)

Test for B→C

$Z = B$,

$= \{B\} \cup ((B \cap AB)^+ \cap AB)$

$= \{B\} \cup ((B)^+ \cap AB)$

$= \{B\} \cup (BCDA \cap AB)$

$= \{B\} \cup \{AB\}$

$= \mathbf{\{AB\}}$

$Z = \{AB\}$

$= \{AB\} \cup ((AB \cap BC)^+ \cap BC)$

$= \{AB\} \cup ((B)^+ \cap BC)$

$= \{AB\} \cup (BCDA \cap BC)$

$= \{AB\} \cup \{BC\}$

$= \mathbf{\{ABC\}}$

$Z = \{ABC\}$

$= \{ABC\} \cup ((ABC \cap CD)^+ \cap CD)$

$= \{ABC\} \cup ((C)^+ \cap CD)$

$= \{ABC\} \cup (CDAB \cap CD)$

$= \{ABC\} \cup \{CD\}$

$= \mathbf{\{ABC\}}$ **So G covers B →C**

A Hugmongously Big Example (cont.)

Test for C→D

$$Z = C,$$
$$= \{C\} \cup ((C \cap AB)^+ \cap AB)$$
$$= \{C\} \cup ((\varnothing)^+ \cap AB)$$
$$= \{C\} \cup (\varnothing)$$
$$= \mathbf{\{C\}}$$

$$Z = \{C\}$$
$$= \{C\} \cup ((C \cap BC)^+ \cap BC)$$
$$= \{C\} \cup ((C)^+ \cap BC)$$
$$= \{C\} \cup (CDAB \cap BC)$$
$$= \{C\} \cup \{BC\}$$
$$= \mathbf{\{BC\}}$$

$$Z = \{BC\}$$
$$= \{BC\} \cup ((BC \cap CD)^+ \cap CD)$$
$$= \{BC\} \cup ((C)^+ \cap CD)$$
$$= \{BC\} \cup (CDAB \cap CD)$$
$$= \{BC\} \cup \{CD\}$$
$$= \mathbf{\{BCD\}} \qquad\qquad \textbf{So G covers C →D}$$

A Hugmongously Big Example (cont.)

Test for D→A

$$Z = D,$$
$$= \{D\} \cup ((D \cap AB)^+ \cap AB)$$
$$= \{D\} \cup ((\emptyset)^+ \cap AB)$$
$$= \{D\} \cup (\emptyset)$$
$$= \mathbf{\{D\}}$$

$$Z = \{D\}$$
$$= \{D\} \cup ((D \cap BC)^+ \cap BC)$$
$$= \{D\} \cup ((\emptyset)^+ \cap BC)$$
$$= \{D\} \cup (\emptyset)$$
$$= \mathbf{\{D\}}$$

$$Z = \{D\}$$
$$= \{D\} \cup ((D \cap CD)^+ \cap CD)$$
$$= \{D\} \cup ((D)^+ \cap CD)$$
$$= \{D\} \cup (DABC \cap CD)$$
$$= \{D\} \cup \{CD\}$$
$$= \mathbf{\{DC\}} \qquad \textbf{Changes made to G so continue.}$$

A Hugmongously Big Example (cont.)

Test for D$\rightarrow$A continues on a second pass through D.

$Z = DC$,
$\quad = \{DC\} \cup ((DC \cap AB)^+ \cap AB)$
$\quad = \{DC\} \cup ((\emptyset)^+ \cap AB)$
$\quad = \{DC\} \cup (\emptyset)$
$\quad = \mathbf{\{DC\}}$

$Z = \{DC\}$
$\quad = \{DC\} \cup ((DC \cap BC)^+ \cap BC)$
$\quad = \{DC\} \cup ((C)^+ \cap BC)$
$\quad = \{D\} \cup (CDAB \cap BC)$
$\quad = \{D\} \cup (BC)$
$\quad = \mathbf{\{DBC\}}$

$Z = \{DBC\}$
$\quad = \{DBC\} \cup ((DBC \cap CD)^+ \cap CD)$
$\quad = \{DBC\} \cup ((CD)^+ \cap CD)$
$\quad = \{DBC\} \cup (CDAB \cap CD)$
$\quad = \{DBC\} \cup \{CD\}$
$\quad = \mathbf{\{DBC\}}$ **Again changes made to G so continue.**

A Hugmongously Big Example (cont.)

Test for D→A continues on a third pass through D.

$Z = DBC,$

$= \{DBC\} \cup ((DBC \cap AB)^+ \cap AB)$

$= \{DBC\} \cup ((B)^+ \cap AB)$

$= \{DBC\} \cup (BCDA \cap AB)$

$= \{DBC\} \cup (AB)$

$= \mathbf{\{DBCA\}}$

Finally, we've included every attribute in R.

Thus, G covers D →A.

Thus, D preserves the functional dependencies in F.

Practice Problem: Determine if D preserves the dependencies in F given:

R = (C, S, Z)

F = {CS →Z, Z→C}

D = {(SZ), (CZ)}   **Solution on next page.**

Practice Problem Solution

Let $\quad$ R = (C, S, Z)
$\qquad$ F = {CS $\rightarrow$Z, Z$\rightarrow$C}
$\qquad$ D = {(SZ), (CZ)}

$$G = F[SZ] \cup F[CZ] \qquad Z = Z \cup ((Z \cap R_i)^+ \cap R_i)$$

Test for each fd in F.
Test for CS$\rightarrow$Z
$\qquad$ Z = CS,
$\qquad\quad = \{CS\} \cup ((CS \cap SZ)^+ \cap SZ)$
$\qquad\quad = \{CS\} \cup ((S)^+ \cap SZ)$
$\qquad\quad = \{CS\} \cup (S)$
$\qquad\quad = \{CS\}$
$\qquad\quad = \{CS\} \cup ((CS \cap CZ)^+ \cap CZ)$
$\qquad\quad = \{CS\} \cup ((C)^+ \cap CZ)$
$\qquad\quad = \{CS\} \cup (C \cap CZ)$
$\qquad\quad = \{CS\} \cup (C)$
$\qquad\quad = \{CS\}$ $\quad$ thus, CS $\rightarrow$Z is not preserved.

Algorithm for Testing for the Lossless Join Property

Algorithm Lossless
// input: a relation schema R= (A_1, A_2, …, A_n),   a set of fds F, a decomposition
//   scheme D = {R_1, R_2, ..., R_k)
// output: true if D has the lossless join property, false otherwise

Lossless (R, F, D)
   Create a matrix of *n* columns and *k* rows where column *y* corresponds to attribute
   A_y ($1 \leq y \leq n$) and row *x* corresponds to relation schema R_x ($1 \leq x \leq k$). Call this matrix T.

   Fill the matrix according to: in T_{xy} put the symbol a_y if A_y is in R_x and the symbol b_{xy} if not.

   Repeatedly "consider" each fd $X \rightarrow Y$ in F until no more changes can be made to T.
        Each time an fd is considered, look for rows in T which agree on all of the columns
        corresponding to the attributes in X. Equate all of the rows which agree in the X
        value on the Y values according to: If any of the Y symbols is a_y make them all a_y,
         if none of them are a_y equate them arbitrarily to one of the b_{xy} values.

    If after making all possible changes to T one of the rows has become $a_1 a_2 ... a_n$
       then return yes, otherwise return no.
end.

Testing for a Lossless Join - Example

Let R = (A, B, C, D, E)

F = {A→C, B→C, C→D, DE→C, CE→A}

D = {(AD), (AB), (BE), (CDE), (AE)}

initial matrix T:

	A	B	C	D	E
(AD)	a_1	b_{12}	b_{13}	a_4	b_{15}
(AB)	a_1	a_2	b_{23}	b_{24}	b_{25}
(BE)	b_{31}	a_2	b_{33}	b_{34}	a_5
(CDE)	b_{41}	b_{42}	a_3	a_4	a_5
(AE)	a_1	b_{52}	b_{53}	b_{54}	a_5

Testing for a Lossless Join – Example (cont.)

Consider each fd in F repeatedly until no changes are made to the matrix:

$A \rightarrow C$: equates b_{13}, b_{23}, b_{53}. Arbitrarily we'll set them all to b_{13} as shown.

	A	B	C	D	E
(AD)	a_1	b_{12}	b_{13}	a_4	b_{15}
(AB)	a_1	a_2	b_{13}	b_{24}	b_{25}
(BE)	b_{31}	a_2	b_{33}	b_{34}	a_5
(CDE)	b_{41}	b_{42}	a_3	a_4	a_5
(AE)	a_1	b_{52}	b_{13}	b_{54}	a_5

Testing for a Lossless Join – Example (cont.)

Consider each fd in F repeatedly until no changes are made to the matrix:

B→C: equates b_{13}, b_{33}. We'll set them all to b_{13} as shown.

	A	B	C	D	E
(AD)	a_1	b_{12}	b_{13}	a_4	b_{15}
(AB)	a_1	a_2	b_{13}	b_{24}	b_{25}
(BE)	b_{31}	a_2	b_{13}	b_{34}	a_5
(CDE)	b_{41}	b_{42}	a_3	a_4	a_5
(AE)	a_1	b_{52}	b_{13}	b_{54}	a_5

Testing for a Lossless Join – Example (cont.)

Consider each fd in F repeatedly until no changes are made to the matrix:

$C \rightarrow D$: equates a_4, b_{24}, b_{34}, $b_{54.}$. We set them all to a_4 as shown.

	A	B	C	D	E
(AD)	a_1	b_{12}	b_{13}	a_4	b_{15}
(AB)	a_1	a_2	b_{13}	a_4	b_{25}
(BE)	b_{31}	a_2	b_{13}	a_4	a_5
(CDE)	b_{41}	b_{42}	a_3	a_4	a_5
(AE)	a_1	b_{52}	b_{13}	a_4	a_5

Testing for a Lossless Join – Example (cont.)

Consider each fd in F repeatedly until no changes are made to the matrix:

DE$\rightarrow$C: equates a_3, b_{13}. We set them both to a_3 as shown.

	A	B	C	D	E
(AD)	a_1	b_{12}	b_{13}	a_4	b_{15}
(AB)	a_1	a_2	b_{13}	a_4	b_{25}
(BE)	b_{31}	a_2	a_3	a_4	a_5
(CDE)	b_{41}	b_{42}	a_3	a_4	a_5
(AE)	a_1	b_{52}	a_3	a_4	a_5

Testing for a Lossless Join – Example (cont.)

Consider each fd in F repeatedly until no changes are made to the matrix:

$CE \rightarrow A$: equates b_{31}, b_{41}, a_1. We set them all to a_1 as shown.

	A	B	C	D	E
(AD)	a_1	b_{12}	b_{13}	a_4	b_{15}
(AB)	a_1	a_2	b_{13}	a_4	b_{25}
(BE)	a_1	a_2	a_3	a_4	a_5
(CDE)	a_1	b_{42}	a_3	a_4	a_5
(AE)	a_1	b_{52}	a_3	a_4	a_5

Testing for a Lossless Join – Example (cont.)

First pass through F is now complete. However row (BE) has become all
a_is, so stop and return true, this decomposition has the lossless join
property.

	A	B	C	D	E
(AD)	a_1	b_{12}	b_{13}	a_4	b_{15}
(AB)	a_1	a_2	b_{13}	a_4	b_{25}
(BE)	a_1	a_2	a_3	a_4	a_5
(CDE)	a_1	b_{42}	a_3	a_4	a_5
(AE)	a_1	b_{52}	a_3	a_4	a_5

Algorithm #1 for Producing a 3NF Decomposition

Algorithm 3NF.1
// input: a relation schema R= $(A_1, A_2, …, A_n)$,   a set of fds F, a set of candidate keys K.
// output: a 3NF decomposition of R, called D, which has the lossless join property and the
//   functional dependencies are preserved.

3NF.1 (R, F, K)
   a = 0;
   for each fd $X \rightarrow Y$ in F do
        a = a +1;
        $R_a = XY$;
   endfor
   if [none of the schemes R_b $(1 \leq b \leq a)$ contains a candidate key of R] then
        a = a + 1;
        R_a = any candidate key of R
   endif
   if [$\bigcup_{b=1}^{a} R_b \neq R$] then   //there are missing attributes

        $R_{a+1} = R - \bigcup_{b=1}^{a} R_b$
   return D = $\{R_1, R_2, ..., R_{a+1}\}$
end.

Example – Using Algorithm 3NF.1

Let R = (A, B, C, D, E)

    K = {AB, AC}

    F = {AB→CDE, AC→BDE, B→C, C→B, C→D, B→E}

Step 1: D = {(ABCDE), (ACBDE), (BC), (CB), (CD), (BE)}

      Reduce to: D = {(ABCDE), (BC), (CD), (BE)}

Step 2: Does D contain a candidate key for R?
    Yes, in (ABCDE)

Step 3: Are all the attributes of R contained in D?
    Yes.

Return D as: **{(ABCDE), (BC), (CD), (BE)}**

Algorithm #2 for Producing a 3NF Decomposition

Algorithm 3NF.2
// input: a relation schema R= $(A_1, A_2, ..., A_n)$,   a set of fds F, a set of candidate keys K.
// output: a 3NF decomposition of R, called D, which is not guaranteed to have either the
//   lossless join property or to preserve the functional dependencies in F.
// This algorithm is based on the removal of transitive dependencies.

3NF.2 (R, F, K)
   do
        if [K $\rightarrow$ Y $\rightarrow$ A where A is non-prime and not an element of either K or Y] then
            decompose R into: $R_1 = \{R - A\}$ with $K_1 = \{K\}$ and $R_2 = \{YA\}$ with $K_2 = \{Y\}$.
   repeat until no transitive dependencies exist in any schema
   D = union of all 3NF schemas produced above.
   test for lossless join
   test for preservation of the functional dependencies
end.

Example – Using Algorithm 3NF.2

Let $R = (A, B, C, D, E)$
$K = \{AB, AC\}$
$F = \{AB \rightarrow CDE, AC \rightarrow BDE, B \rightarrow C, C \rightarrow B, C \rightarrow D, B \rightarrow E\}$

Step 1: R not in 3NF since $AB \rightarrow C \rightarrow D$
Decompose to: $R_1 = (A, B, C, E)$ with $K_1 = K = \{AB, AC\}$
$R_2 = (C, D)$ with $K_2 = \{C\}$

Step 2: R_2 in 3NF. R_1 not in 3NF since $AB \rightarrow B \rightarrow E$
Decompose R_1 to: $R_{11} = (A, B, C)$ with $K_{11} = K_1 = K = \{AB, AC\}$
$R_{12} = (B, E)$ with $K_{12} = \{B\}$

Step 3: R_2, R_{11}, and R_{12} are all in 3NF

Step 4: Test for the lossless join property (see next page).

Step 4: Checking for a Lossless Join in the Decomposition

AB→CDE: (1st time: equates nothing)

AC→BDE: (1st time: equates nothing)

B→C: (1st time: equates a_3 & b_{33})

C→B: (1st time: equates a_2 & b_{12})

C→D: (1st time: equates b_{14}, b_{24}, b_{34}) – stop second row becomes all a's

B→E: (1st time: equates a_5, b_{15}, b_{25})

Decomposition has the lossless join property.

	A	B	C	D	E
(CD)	b_{11}	a_2	a_3	a_4	b_{15}
(ABC)	a_1	a_2	a_3	a_4	b_{15}
(BE)	b_{31}	a_2	a_3	a_4	a_5

Step 5: Testing the Preservation of the Functional Dependencies

Let
$$R = (A, B, C, D, E)$$
$$F = \{AB \rightarrow CDE, AC \rightarrow BDE, B \rightarrow C, C \rightarrow B, C \rightarrow D, B \rightarrow E\}\}$$
$$D = \{(CD), (ABC), (BE)\}$$

$$G = F[CD] \cup F[ABC] \cup F[BE] \qquad\qquad Z = Z \cup ((Z \cap R_i)^+ \cap R_i)$$

Test for AB→CDE

$$Z = AB,$$
$$= \{AB\} \cup ((AB \cap CD)^+ \cap CD)$$
$$= \{AB\} \cup ((\varnothing)^+ \cap CD)$$
$$= \{AB\} \cup (\varnothing \cap CD)$$
$$= \{AB\} \cup (\varnothing)$$
$$= \mathbf{\{AB\}}$$
$$= \{AB\} \cup ((AB \cap ABC)^+ \cap ABC)$$
$$= \{AB\} \cup ((AB)^+ \cap ABC)$$
$$= \{AB\} \cup (ABCDE \cap ABC)$$
$$= \{AB\} \cup (ABC)$$
$$= \mathbf{\{ABC\}}$$
$$= \{ABC\} \cup ((ABC \cap BE)^+ \cap BE)$$
$$= \{ABC\} \cup ((B)^+ \cap BE)$$
$$= \{ABC\} \cup (BCDE \cap BE)$$
$$= \{ABC\} \cup (BE)$$
$$= \mathbf{\{ABCE\}}$$

Step 5: Testing the Preservation of the Functional Dependencies
(cont.)

Test for AB→CDE continues

$$Z = \{ABCE\} \cup ((ABCE \cap CD)^+ \cap CD)$$
$$= \{ABCE\} \cup ((C)^+ \cap CD)$$
$$= \{ABCE\} \cup (CBDE \cap CD)$$
$$= \{ABCE\} \cup (CD)$$
$$= \textbf{\{ABCDE\} thus, AB→CDE is preserved}$$

Test for AC→BDE

$$Z = AC$$
$$= \{AC\} \cup ((AC \cap CD)^+ \cap CD)$$
$$= \{AC\} \cup ((C)^+ \cap CD)$$
$$= \{AC\} \cup (CBDE \cap CD)$$
$$= \{AC\} \cup (CD)$$
$$= \textbf{\{ACD\}}$$
$$= \{ACD\} \cup ((ACD \cap ABC)^+ \cap ABC)$$
$$= \{ACD\} \cup ((AC)^+ \cap ABC)$$
$$= \{ACD\} \cup (ACBDE \cap ABC)$$
$$= \{ACD\} \cup (ABC)$$
$$= \textbf{\{ABCD\}}$$

Step 5: Testing the Preservation of the Functional Dependencies (cont.)

Test for AC→BDE continues

$Z = \{ABCD\} \cup ((ABCD \cap BE)^+ \cap BE)$
$\quad = \{ABCD\} \cup ((B)^+ \cap BE)$
$\quad = \{ABCD\} \cup (BCDE \cap BE)$
$\quad = \{ABCD\} \cup (BE)$
$\quad = \{ABCDE\}$ **thus, AC→BDE is preserved**

Test for B→C

$Z = B$
$\quad = \{B\} \cup ((B \cap CD)^+ \cap CD)$
$\quad = \{B\} \cup ((C)^+ \cap CD)$
$\quad = \{B\} \cup (CBDE \cap CD)$
$\quad = \{B\} \cup (CD)$
$\quad = \{BCD\}$ **thus B→C is preserved**

Test for C→B

$Z = C$
$\quad = \{C\} \cup ((C \cap CD)^+ \cap CD)$
$\quad = \{C\} \cup ((C)^+ \cap CD)$
$\quad = \{C\} \cup (CBDE \cap CD)$
$\quad = \{C\} \cup (CD)$
$\quad = \{CD\}$

Step 5: Testing the Preservation of the Functional Dependencies (cont.)

Test for C→B continues

$$Z = \{CD\} \cup ((CD \cap ABC)^+ \cap ABC)$$
$$= \{CD\} \cup ((C)^+ \cap ABC)$$
$$= \{CD\} \cup (CBDE \cap ABC)$$
$$= \{CD\} \cup (BC)$$
$$= \{BCD\} \text{ thus, } C \to B \text{ is preserved}$$

Test for C→D

$$Z = C$$
$$= \{C\} \cup ((C \cap CD)^+ \cap CD)$$
$$= \{C\} \cup ((C)^+ \cap CD)$$
$$= \{C\} \cup (CBDE \cap CD)$$
$$= \{C\} \cup (CD)$$
$$= \{CD\} \text{ thus } C \to D \text{ is preserved}$$

Test for B→E

$$Z = B$$
$$= \{B\} \cup ((B \cap CD)^+ \cap CD)$$
$$= \{B\} \cup ((\varnothing)^+ \cap CD)$$
$$= \{B\} \cup (\varnothing)$$
$$= \{B\}$$

Step 5: Testing the Preservation of the Functional Dependencies (cont.)

Test for B→E continues

$Z = \{B\} \cup ((B \cap ABC)^+ \cap ABC)$

$\quad = \{B\} \cup ((B)^+ \cap ABC)$

$\quad = \{B\} \cup (BCDE \cap ABC)$

$\quad = \{BC\} \cup (BC)$

$\quad = \mathbf{\{BC\}}$

$Z = \{BC\}$

$\quad = \{BC\} \cup ((BC \cap ABC)^+ \cap ABC)$

$\quad = \{BC\} \cup ((C)^+ \cap ABC)$

$\quad = \{BC\} \cup (CBDE \cap ABC)$

$\quad = \{BC\} \cup (BC)$

$\quad = \mathbf{\{BC\}}$

$Z = \{BC\}$

$\quad = \{BC\} \cup ((BC \cap BE)^+ \cap BE)$

$\quad = \{BC\} \cup ((B)^+ \cap BE)$

$\quad = \{BC\} \cup (BCDE \cap BE)$

$\quad = \{BC\} \cup (BE)$

$\quad = \mathbf{\{BCE\}}$ **thus, B →E is preserved.**

Why Use 3NF.2 Rather Than 3NF.1

- Why would you use algorithm 3NF.2 rather than algorithm 3NF.1 when you know that algorithm 3NF.1 will guarantee that both the lossless join property and the preservation of the functional dependencies?

- The answer is that algorithm 3NF.2 will typically produce fewer relational schemas than will algorithm 3NF.1. Although both the lossless join and dependency preservation properties must be independently tested when using algorithm 3NF.2.

Algorithm #3 for Producing a 3NF Decomposition

Algorithm 3NF.3
// input: a relation schema R= $(A_1, A_2, ..., A_n)$, a set of fds F.
// output: a 3NF decomposition of R, called D, which is guaranteed to have both the
//   lossless join property and to preserve the functional dependencies in F.
// This algorithm is based on the minimal cover for F (see page 46).

3NF.3 (R, F)
    find a minimal cover for F, call this cover G (see page 46 for algorithm)
    for each determinant X that appears in G do
        create a relation schema { $X \cup A_1 \cup A_2 \cup ... \cup A_m$} where A_i ($1 \leq i \leq m$) represents
            all the consequents of fds in G with determinant X.
    place all remaining attributes, if any, in a single schema.
    if none of the schemas contains a key for R, create an additional schema which
        contains any candidate key for R.
end.

Algorithm 3NF.3

- Algorithm 3NF.3 is very similar to algorithm 3NF.1, differing only in how the schemas of the decomposition scheme are created.

    – In algorithm 3NF.1, the schemas are created directly from F.

    – In algorithm 3NF.3, the schemas are created from a minimal cover for F.

- In general, algorithm 3NF.3 should generate fewer relation schemas than algorithm 3NF.1.

Another Technique for Testing the Preservation of Dependencies

- The algorithm given on page 75 for testing the preservation of a set of functional dependencies on a decomposition scheme is fairly efficient for computation, but somewhat tedious to do by hand.

- On the next page is an example solving the same problem that we did in the example on page 77, utilizing a different technique which is based on the concept of covers.

- Given D, R, and F, if $D = \{R_1, R_2, ..., R_n)$ then

  $G = F[R_1] \cup F[R_2] \cup F[R_3] \cup ... \cup F[R_n]$ and if every

  functional dependency in F is implied by G, then G covers F.

- The technique is to generate that portion of G^+ that allows us to know if G covers F.

A Hugmongously Big Example Using Different Technique

Let $R = (A, B, C, D)$

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

$\quad\quad D = \{(AB), (BC), (CD)\}$

$G = F[AB] \cup F[BC] \cup F[CD]$

Projection onto schema (AB)

$\quad\quad\quad F[AB] = A^+ \cup B^+ \cup (AB)^+$

$\quad\quad\quad\quad\quad = \{ABCD\} \cup \{ABCD\} \cup \{ABCD\}$

apply projection: $= \{AB\} \cup \{AB\} \cup \{AB\} = \mathbf{\{AB\}}$, **A→B is covered**

Projection onto schema (BC)

$\quad\quad\quad F[BC] = B^+ \cup C^+ \cup (BC)^+$

$\quad\quad\quad\quad\quad = \{BCDA\} \cup \{CDAB\} \cup \{BCDA\}$

apply projection: $= \{BC\} \cup \{BC\} \cup \{BC\} = \mathbf{\{BC\}}$, **C→C is covered**

A Hugmongously Big Example Using Different Technique
(cont.)

Projection onto schema (CD)

$$F[CD] = C^+ \cup D^+ \cup (CD)^+$$
$$= \{CDAB\} \cup \{DABC\} \cup \{CDAB\}$$

apply projection: $= \{CD\} \cup \{CD\} \cup \{CD\} = \mathbf{\{CD\}}$, **C→D is covered**

- Thus, the projections have covered every functional dependency in F except $D \rightarrow A$. So, now the question becomes does G logically imply $D \rightarrow A$?

- Generate D^+(with respect to G) and if A is in this closure the answer is yes.

$$D_G^+ = \{D, C, B, A\} \text{ Therefore, } G \vDash D \rightarrow A$$

Multi-valued Dependencies and Fourth Normal Form

- Functional dependencies are the most common and important type of constraint in relational database design theory.

- However, there are situations in which the constraints that hold on a relation cannot be expressed as a functional dependency.

- Multi-valued dependencies are related to 1NF. Recall that 1NF simply means that all attribute values in a relation are atomic, which means that a tuple cannot have a set of values for some particular attribute.

- If we have a situation in which two or more multi-valued independent attributes appear in the same relation schema, then we'll need to repeat every value of one of the attributes with every value of the other attribute to keep the relation instance consistent and to maintain the independence among the attributes involved.

- Basically, whenever two independent 1:M relationships A:B and A:C occur in the same relation, a multi-valued dependency may occur.

Multi-valued Dependencies (cont.)

- Consider the following situation of a N1NF relation.

name	classes	vehicles
Mark	COP 4710 COP 4610	Mercedes E350 Ford F350
Kristy	COP 3330 CDA 3103 COT 4810	Mercedes E500 Porsche Carrera

Multi-valued Dependencies (cont.)

- Converting the N1NF relation to a 1NF relation.

name	classes	vehicles
Mark	COP 4710	Mercedes E350
Mark	COP 4710	Ford F350
Mark	COP 4610	Mercedes E350
Mark	COP 4610	Ford F350
Kristy	COP 3330	Mercedes E500
Kristy	CDA 3103	Mercedes E500
Kristy	COT 4810	Mercedes E500
Kristy	COP 3330	Porsche Carrera
Kristy	CDA 3103	Porsche Carrera
Kristy	COT 4810	Porsche Carrera

Multi-valued Dependencies (cont.)

- Basically, a multi-valued dependency is an assertion that two attributes or sets of attributes are independent of one another.

- This is a generalization of the notion of a functional dependency, in the sense that every fd implies a corresponding multi-valued dependency.

- However, there are certain situations involving independence of attributes that cannot be explained as functional dependencies.

- There are situations in which a relational schema may be in BCNF, yet the relation exhibits a kind of redundancy that is not related to functional dependencies.

Multi-valued Dependencies (cont.)

- The most common source of redundancy in BCNF schemas is an attempt to put two or more M:M relationships in a single relation.

name	city	classes	vehicles
Mark	Orlando	COP 4710	Mercedes E350
Mark	Orlando	COP 4710	Ford F350
Mark	Orlando	COP 4610	Mercedes E350
Mark	Orlando	COP 4610	Ford F350
Kristy	Milan	COP 3502	Mercedes E500
Kristy	Milan	CDA 3103	Mercedes E500
Kristy	Milan	COT 4810	Mercedes E500
Kristy	Milan	COP 3502	Ford F350
Kristy	Milan	CDA 3103	Ford F350
Kristy	Milan	COT 4810	Ford F350

Multi-valued Dependencies (cont.)

- Focusing on the relation on the previous page, notice that there is no reason to associate a given class with a given vehicle and not another vehicle.

- To express the fact that classes and vehicles are independent properties of a person, we have each class appear with each class.

- Clearly, there is redundancy in this relation, but this relation does not violate BCNF. In fact there are no non-trivial functional dependencies at all in this schema.

- We know from our earlier discussions of normal forms based on functional dependencies that redundancies were removed, yet here is a schema in BCNF that clearly contains redundant information.

Multi-valued Dependencies (cont.)

- For example, in this relation, attribute city is not functionally determined by any of the other three attributes.

- Thus the fd: name class vehicle $\rightarrow$ city does not hold for this schema because we could have two persons with the same name, enrolled in the same class, and drive the same type of vehicle.

- You should verify that none of the four attributes in functionally determined by the other three. Which means that there are no non-trivial functional dependencies that hold on this relation schema.

- Thus, all four attributes form the only key and this means that the relation is in BCNF, yet clearly is redundant.

Multi-valued Dependencies (cont.)

- A multi-valued dependency (mvd) is a statement about some relation R that when you fix the values for one set of attributes, then the values in certain other attributes are independent of the values of all the other attributes in the relation.

- More precisely, we have the mvd

  $$A_1A_2...A_n \twoheadrightarrow B_1B_2...B_m$$

  holds for a relation R if when we restrict ourselves to the tuples of R that have particular values for each of the attributes among the A's, then the set of values we find among the B's is independent of the set of values we find among the attributes of R that are **not** among the A's or B's.
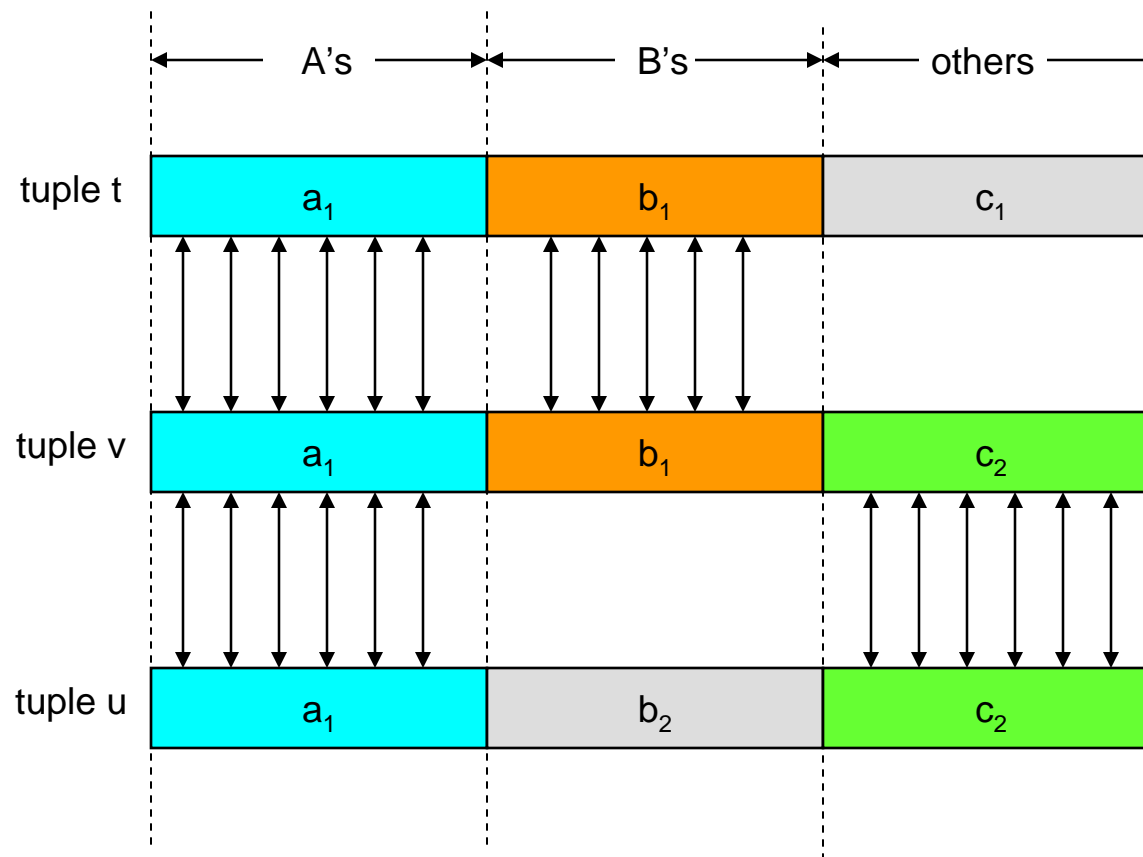
Multi-valued Dependencies (cont.)

- Even more precisely, a mvd holds if:

  For each pair of tuples t and u of relation R that agree on all the A's, we can find in R some tuple v that agrees:

  1. With both t and u on the A's

  2. With t on the B's

  3. With u on all attributes of R that are not among the A's or B's.

  – Note that we can use this rule with t and u interchanged, to infer the existence of a fourth tuple w that agrees with u on the B's and with t on the other attributes. As a consequence, for any fixed values of the A's, the associated values of the B's and the other attributes appear in all possible combinations in different tuples.

Relationship of Tuple *v* to Tuple *t* When mvd Exists

A multi-valued dependency guarantees that tuple *V* exists

Multi-valued Dependencies (cont.)

- In general, we can assume that the A's and B's (left side and right side) of a mvd are disjoint.

- As with functional dependencies, it is permissible to add some of the A's to the right side.

- Unlike, functional dependencies where a set of attributes on the right side was a short-hand notation for a set of fds with single attribute right sides, with mvds, we must deal only with sets of attributes on the right side as it is not always possible to break the right side of mvds into single attributes.

Example: Multi-valued Dependencies

- Consider the following relation instance.

name	street	city	title	year
C. Fisher	123 Maple Street	Hollywood	Star Wars	1977
C. Fisher	5 Locust Lane	Malibu	Star Wars	1977
C. Fisher	123 Maple Street	Hollywood	Empire Strikes Back	1980
C. Fisher	5 Locust Lane	Malibu	Empire Strikes Back	1980
C. Fisher	123 Maple Street	Hollywood	Return of the Jedi	1983
C. Fisher	5 Locust Lane	Malibu	Return of the Jedi	1983

- The mvd name ↠ street city holds on this relation.

  – That is, for each star's name, the set of addresses appears in conjunction with each of the star's movies.

Example: Multi-valued Dependencies (cont.)

- For an example of how the formal definition of this mvd applies, consider the first and fourth tuples from the previous relation instance.

name	street	city	title	year
C. Fisher	123 Maple Street	Hollywood	Star Wars	1977
C. Fisher	5 Locust Lane	Malibu	Empire Strikes Back	1980

- If we let the first tuple be t and the second tuple be u, then the mvd asserts that we must also find in R the tuple that has name C. Fisher, a street and city that agree with the first tuple, and other attributes (title and year) that agree with the second tuple. There is indeed such a tuple (the third tuple in the original instance).

name	street	city	title	year
C. Fisher	123 Maple Street	Hollywood	Empire Strikes Back	1980

Example: Multi-valued Dependencies (cont.)

- Similarly, we could let t be the second tuple below and u be the first tuple below (reversed from the previous page). Then the mvd tells us that there is a tuple of R that agrees with the second tuple in attributes name, street, and city with the first tuple in attributes name, title, and year.

name	street	city	title	year
C. Fisher	123 Maple Street	Hollywood	Star Wars	1977
C. Fisher	5 Locust Lane	Malibu	Empire Strikes Back	1980

- There is indeed such a tuple (the second tuple in the original instance).

name	street	city	title	year
C. Fisher	5 Locust Lane	Malibu	Star Wars	1977

Reasoning about Multi-valued Dependencies

- There are a number of inference rules that deal with mvds that are similar to the inference rules for functional dependencies.

1. Trivial multi-valued dependencies:

   If $A_1A_2...A_n \twoheadrightarrow B_1B_2...B_m$ holds for some relation, then so does $A_1A_2...A_n \twoheadrightarrow C_1C_2...C_k$ where the C's are the B's plus one or more of the A's.

   Conversely, we can also remove attributes from the B's if they are among the A's and infer the mvd $A_1A_2...A_n \twoheadrightarrow D_1D_2...D_r$ if the D's are those B's that are not among the A's.

Reasoning about Multi-valued Dependencies

2. Transitive rule for multi-valued dependencies:

   If $A_1A_2...A_n \twoheadrightarrow B_1B_2...B_m$ and $B_1B_2...B_m \twoheadrightarrow C_1C_2...C_k$ both hold for some relation, then so does $A_1A_2...A_n \twoheadrightarrow C_1C_2...C_k$. However, any C's that are also B's must be deleted from the right side.

- mvds do not obey the additivity/projectivity rules as do functional dependencies.

Reasoning about Multi-valued Dependencies

- Consider the same relation schema as before, where the mvd name ↠ street city held. If the projectivity (splitting) rule held we would expect that

   name ↠ street would also be true. This mvd states that each star's street addresses are independent of the other attributes (including city). However, that statement is false. The first two tuples in the relation instance indicate that this is not true.

name	street	city	title	year
C. Fisher	123 Maple Street	Hollywood	Star Wars	1977
C. Fisher	5 Locust Lane	Malibu	Star Wars	1977

Reasoning about Multi-valued Dependencies

- This hypothetical mvd name ↠ street, if it held would allow us to infer that the tuples with the streets interchanged would be in the relation instance. However, these tuples are not there because the home at 5 Locust Lane is in Malibu and not Hollywood.

name	street	city	title	year
C. Fisher	5 Locust Lane	Hollywood	Star Wars	1977
C. Fisher	123 Maple Street	Malibu	Star Wars	1977

invalid tuples that cannot exist

Reasoning about Multi-valued Dependencies

- There are however, several new inference rules that apply only to multi-valued dependencies.

- First, every fd is a mvd. That is, if $A_1A_2...A_n \rightarrow B_1B_2...B_m$ holds for some relation, then so does $A_1A_2...A_n \twoheadrightarrow B_1B_2...B_m$ hold.

- Second, complementation has no fd counterpart. The complementation rule states: if $A_1A_2...A_n \twoheadrightarrow B_1B_2...B_m$ is a mvd that holds on some relation R, then R also satisfies $A_1A_2...A_n \twoheadrightarrow C_1C_2...C_k$, where the C's are all attributes of R that are not included in the A's or B's.

  – Thus, if name $\twoheadrightarrow$ street city holds, the complementation rule states that name $\twoheadrightarrow$ title year also holds, because street and city are not mentioned in the first mvd. The inferred mvd intuitively means that each star has a set of movies that they appeared in, which are independent of their address.

Fourth Normal Form

- The redundancy that we've seen in the relation instances in this section of the notes are caused by the existence of multi-valued dependencies.

- As we did with functional dependencies, we can use multi-valued dependencies and a different decomposition algorithm to produce a stronger normal form which is based not on functional dependencies but the multi-valued dependencies.

- Fourth Normal Form (4NF) eliminates all non-trivial multi-valued dependencies (as are all fds that violate BCNF). The resulting decomposition scheme has neither the redundancy from fds nor redundancy from mvds.

Fourth Normal Form (cont.)

- A mvd $A_1A_2...A_n \twoheadrightarrow B_1B_2...B_m$ for a relation scheme R is non-trivial if:

  1. None of the B's is among the A's.

  2. Not all of the attributes of R are among the A's and B's.

- 4NF is essentially the BCNF condition, but applied to mvds instead of fds.

- Formally, a relation scheme R is in 4NF if whenever $A_1A_2...A_n \twoheadrightarrow B_1B_2...B_m$ is a non-trivial mvd, $\{A_1A_2...A_n\}$ is a superkey of R.

Fourth Normal Form (cont.)

- The example relation scheme that we have been dealing with is not in 4NF because name ↠ street city is a non-trivial mvd, yet name by itself is not a superkey. In fact, for this relation the only key is all the attributes.

- 4NF is truly a generalization of BCNF. Since every fd is a mvd, every BCNF violation is also a 4NF violation. In other words, every relation scheme that is in 4NF is therefore in BCNF.

- However, there are some relation that are in BCNF but not in 4NF. The relation instance we have been using in this section of notes is a case in point. It is clearly in BCNF, yet as we just illustrated, it is not in 4NF.

Decomposition into Fourth Normal Form

- The 4NF decomposition algorithm is analogous to the 3NF and BCNF decomposition algorithm:

- Find a 4NF violation, say $A_1A_2...A_n \twoheadrightarrow B_1B_2...B_m$ where $\{A_1A_2...A_n\}$ is not a superkey. Note that this mvd could be a true mvd or it could be derived from the corresponding fd $A_1A_2...A_n \rightarrow B_1B_2...B_m$, since every fd is an mvd. Then break the schema for R into two schemas where: (1) the first schema contains all the A's and B's and the second schema contains the A's and all the attributes of R that are not among the A's or B's.

Decomposition into Fourth Normal Form (cont.)

- Using our previous example relation that we now know is not in 4NF, let's decompose into a relation schema that is in 4NF.

- We know that name ↠ street city is a 4NF violation. The original schema R (5 attributes) will be replaced by one schema that contains only the three attributes from the mvd above, and a second schema that consists of the left side of the above mvd plus the attributes that do not appear in this mvd, which are the attributes title, and year.

  R1 = {name, street, city}

  R2 = {name, title, year}

Decomposition into Fourth Normal Form (cont.)

R1 = {name, street, city}   R2 = {name, title, year}

- In each of these schema there are no non-trivial mvds or fds, so they are both in 4NF. Notice that in the relation scheme R1, the mvd name ↠ street city is now trivial since it involves every attribute. Likewise, in R2, the mvd name ↠ title year is also trivial.

Summary of Normal Forms

Property	3NF	BCNF	4NF
Eliminates redundancy due to functional dependencies	most	yes	yes
Eliminates redundancy due to multi-valued dependencies	no	no	yes
Preserves functional dependencies	yes	maybe	maybe
Preserves multi-valued dependencies	maybe	maybe	maybe
Has the lossless join property	yes	yes	yes