COP 4710: Database Systems
Spring 2004

-Day 9 – February 2, 2004 –
Introduction to Functional Dependencies

Instructor :	Mark Llewellyn
	markl@cs.ucf.edu
	CC1 211, 823-2790
	http://www.cs.ucf.edu/courses/cop4710/spr2004

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.

Update 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).

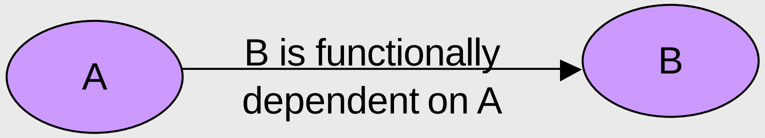
How can $AB \rightarrow C$ be enforced without joining the two relation schemas in γ? Answer, it can't, therefore the fds are not preserved in γ.

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.)

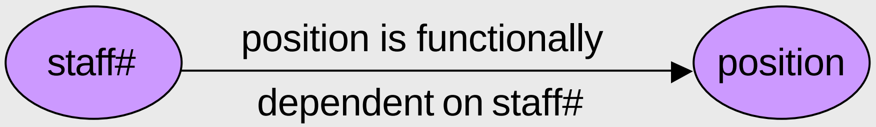
- Consider a relation with attributes A and B, where attribute B is functionally dependent on attribute A. If we know the value of A and we examine the relation that holds this dependency, we will find only one value of B in all of the tuples that have a given value of A, at any moment in time. Note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

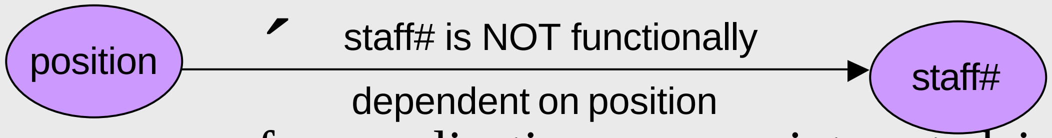


- The determinant of a functional dependency is the attribute or group of attributes on the left-hand side of the arrow in the functional dependency. The consequent of a fd is the attribute or group of attributes on the right-hand side of the arrow.
  – In the figure above, A is the determinant of B 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 – 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.





- 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#

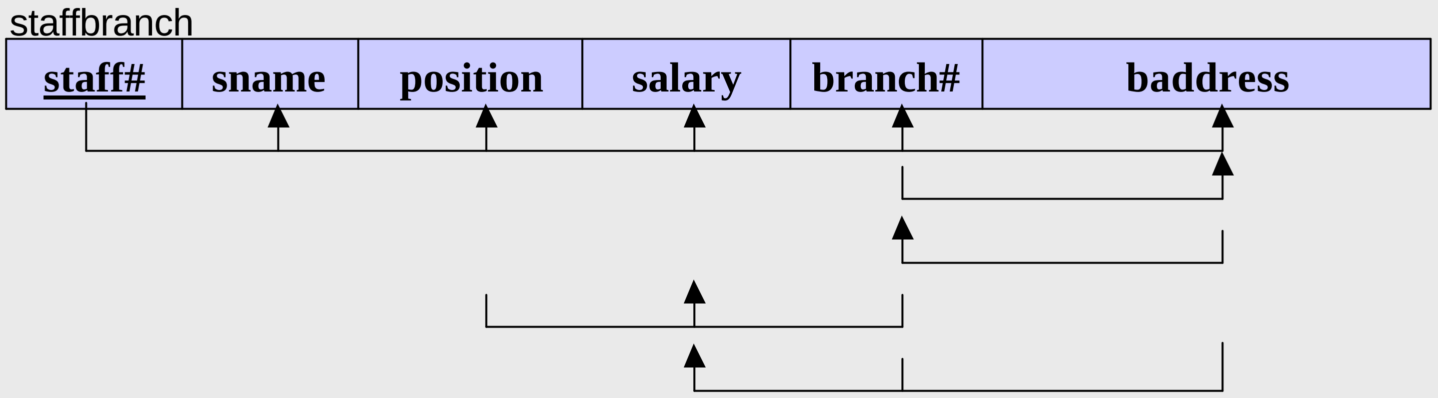
branch#, position $\rightarrow$ salary

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



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:

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

    $\{$ staff#, sname$\} \rightarrow$ 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 between attribute(s) in the determinant and 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 ? $X \rightarrow Y$ denotes that the functional dependency $X \rightarrow Y$ is implied by the set of fds F.
- Formally, $F^+ \equiv \{X \rightarrow Y \mid F$? $X \rightarrow Y\}$
- 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?

Proof

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

Practice Problem

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

Answer: in next set of notes

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 ? 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)
{
    $X^+ \leftarrow X$;
    repeat
        $oldX^+ \leftarrow X^+$;
        for every fd $W \rightarrow Z$ in F do
            if $W \subseteq X^+$ then $X^+ \leftarrow X^+ \cup Z$;
    until ($oldX^+ = X^+$);
}
```

Example Using Algorithm Closure

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

<u>pass 1</u>

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

<u>pass 2</u>

$X^+ = \{A, E, D, C\}$
using $A \to D$, yes, but no changes
using $AB \to E$, no
using $BI \to E$, no
using $CD \to I$, $CD \subseteq X^+$, so add I to X^+, $X^+ = \{A, E, D, C, I\}$
using $E \to C$, 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 $X \rightarrow Y$
output: true if F ? $X \rightarrow Y$, false otherwise
Member (F, $X \rightarrow Y$)
{
    if $Y \subseteq$ 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 ≡ 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 ← G;
    for each fd X → Y ∈ G do
        if Member(F – {X → Y}, X → Y)
            then F ← F – {X → 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 ® B, B ® A, B ® 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 ? 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 $X \to Y \in F$, then attribute A is extraneous in $X \to Y$ wrt F if:

1. $X = AZ$, $X \neq Z$ and $\{F - \{X \to Y\}\} \cup \{Z \to Y\} \equiv F$, or
2. $Y = AW$, $Y \neq W$ and $\{F - \{X \to Y\}\} \cup \{X \to W\} \equiv F$

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

Extraneous Attributes (cont.)

Example:

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

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

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

Left and Right Reduced Sets of FDs

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

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

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

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

  $X \to Y$ 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 ← G;
    for each fd X → Y in G do
        for each attribute A in Y do
            if Member(F – {X → Y} ∪ {X → (Y - A)}, X → A)
                then remove A from Y in X → 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.

<u>Example:</u>

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.

<u>Example:</u>

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

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.

<u>Example:</u>

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

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)
{
    $G \leftarrow F$;
    replace each fd $X \rightarrow A_1A_2 \ldots A_n$ in G by n fds $X \rightarrow A_1, X \rightarrow A_2, \ldots, X \rightarrow A_n$
    for each fd $X \rightarrow A$ in G do
        if Member( G? {$X \rightarrow A$}, $X \rightarrow A$ )
            then $G \leftarrow G - \{X \rightarrow A\}$
    endfor
    for each remaining fd in G, $X \rightarrow A$ do
        for each attribute $B \in X$ do
            if Member( [{G? {$X \rightarrow A$}} $\cup$ {(X?B) $\rightarrow$ A}], (X?B) $\rightarrow$ A)
                then $G \leftarrow$ {G? {$X \rightarrow A$}} $\cup$ {(X?B) $\rightarrow$ A}
    endfor
    return(G);
}
```