Schema Refinement:
Dependencies and Normal Forms

Grant Weddell

Cheriton School of Computer Science
University of Waterloo

CS 348
Introduction to Database Management
Winter 2017
Outline

1. Introduction
   - Design Principles
   - Problems due to Poor Designs

2. Functional Dependencies
   - Logical Implication of FDs
   - Attribute Closure

3. Schema Decomposition
   - Lossless-Join Decompositions
   - Dependency Preservation

4. Normal Forms based on FDs
   - Boyce-Codd Normal Form
   - Third Normal Form
Step 1 – ER-to-relational mapping: obtaining an initial design
Design Process – Where are we?

Step 1 – ER-to-relational mapping: obtaining an initial design
Step 2 – Normalization: diagnosing and improving a design
Relational Design Principles

- Relations should have semantic unity
- Information repetition should be avoided
  - Anomalies: insertion, deletion, modification
- Avoid null values as much as possible
  - Certainly avoid excessive null values
- Avoid spurious joins
Description of a parts/suppliers database:

- Each type of part has a name and an identifying number, and may be supplied by zero or more suppliers. Each supplier may offer the part at a different price.
- Each supplier has an identifying number, a name, and a contact location for ordering parts.
An instance of the parts/suppliers database.

### Suppliers

<table>
<thead>
<tr>
<th>Sno</th>
<th>Sname</th>
<th>City</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>Magna</td>
<td>Ajax</td>
</tr>
<tr>
<td>S2</td>
<td>Budd</td>
<td>Hull</td>
</tr>
</tbody>
</table>

### Parts

<table>
<thead>
<tr>
<th>Pno</th>
<th>Pname</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>Bolt</td>
</tr>
<tr>
<td>P2</td>
<td>Nut</td>
</tr>
<tr>
<td>P3</td>
<td>Screw</td>
</tr>
</tbody>
</table>

### Supplies

<table>
<thead>
<tr>
<th>Sno</th>
<th>Pno</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>P1</td>
<td>0.50</td>
</tr>
<tr>
<td>S1</td>
<td>P2</td>
<td>0.25</td>
</tr>
<tr>
<td>S1</td>
<td>P3</td>
<td>0.30</td>
</tr>
<tr>
<td>S2</td>
<td>P3</td>
<td>0.40</td>
</tr>
</tbody>
</table>
An alternative E-R model for the parts/suppliers database.
### Supplied_Items

<table>
<thead>
<tr>
<th>Sno</th>
<th>Sname</th>
<th>City</th>
<th>Pno</th>
<th>Pname</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>Magna</td>
<td>Ajax</td>
<td>P1</td>
<td>Bolt</td>
<td>0.50</td>
</tr>
<tr>
<td>S1</td>
<td>Magna</td>
<td>Ajax</td>
<td>P2</td>
<td>Nut</td>
<td>0.25</td>
</tr>
<tr>
<td>S1</td>
<td>Magna</td>
<td>Ajax</td>
<td>P3</td>
<td>Screw</td>
<td>0.30</td>
</tr>
<tr>
<td>S2</td>
<td>Budd</td>
<td>Hull</td>
<td>P3</td>
<td>Screw</td>
<td>0.40</td>
</tr>
</tbody>
</table>

A database instance corresponding to the alternative E-R model.
Consider

- Is one schema better than the other?
- What does it mean for a schema to be good?

The single-table schema suffers from several kinds of problems:
- Update problems (e.g. changing name of supplier)
- Insert problems (e.g. add a new item)
- Delete problems (e.g. Budd no longer supplies screws)

The multi-table schema does not have these problems.
Change Anomalies

Consider

- Is one schema better than the other?
- What does it mean for a schema to be good?

- The single-table schema suffers from several kinds of problems:
  - Update problems (e.g. changing name of supplier)
  - Insert problems (e.g. add a new item)
  - Delete problems (e.g. Budd no longer supplies screws)
  - Likely increase in space requirements

- The multi-table schema does not have these problems.
Is more tables always better?

<table>
<thead>
<tr>
<th>Snos</th>
<th>Snames</th>
<th>Cities</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sno</td>
<td>Sname</td>
<td>City</td>
</tr>
<tr>
<td>S1</td>
<td>Magna</td>
<td>Ajax</td>
</tr>
<tr>
<td>S2</td>
<td>Budd</td>
<td>Hull</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Inums</th>
<th>Inames</th>
<th>Prices</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inum</td>
<td>Iname</td>
<td>Price</td>
</tr>
<tr>
<td>I1</td>
<td>Bolt</td>
<td>0.50</td>
</tr>
<tr>
<td>I2</td>
<td>Nut</td>
<td>0.25</td>
</tr>
<tr>
<td>I3</td>
<td>Screw</td>
<td>0.30</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.40</td>
</tr>
</tbody>
</table>
Is more tables always better?

<table>
<thead>
<tr>
<th>Snos</th>
<th>Snames</th>
<th>Cities</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sno</td>
<td>Sname</td>
<td>City</td>
</tr>
<tr>
<td>S1</td>
<td>Magna</td>
<td>Ajax</td>
</tr>
<tr>
<td>S2</td>
<td>Budd</td>
<td>Hull</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Inums</th>
<th>Inames</th>
<th>Prices</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inum</td>
<td>Iname</td>
<td>Price</td>
</tr>
<tr>
<td>I1</td>
<td>Bolt</td>
<td>0.50</td>
</tr>
<tr>
<td>I2</td>
<td>Nut</td>
<td>0.25</td>
</tr>
<tr>
<td>I3</td>
<td>Screw</td>
<td>0.30</td>
</tr>
</tbody>
</table>

Information about relationships is lost!
Designing Good Databases

Goals

- A methodology for evaluating schemas (detecting anomalies).
- A methodology for transforming bad schemas into good schemas (repairing anomalies).

How do we know an anomaly exists?

- Certain types of integrity constraints reveal regularities in database instances that lead to anomalies.

What should we do if an anomaly exists?

- Certain schema decompositions can avoid anomalies while retaining all information in the instances.
Designing Good Databases

Goals

- A methodology for evaluating schemas (detecting anomalies).
- A methodology for transforming bad schemas into good schemas (repairing anomalies).

- How do we know an anomaly exists?
  - Certain types of *integrity constraints* reveal regularities in database instances that lead to anomalies.
Designing Good Databases

Goals

• A methodology for evaluating schemas (detecting anomalies).
• A methodology for transforming bad schemas into good schemas (repairing anomalies).

• How do we know an anomaly exists?
  • Certain types of \textit{integrity constraints} reveal regularities in database instances that lead to anomalies.

• What should we do if an anomaly exists?
  • Certain \textit{schema decompositions} can avoid anomalies while retaining all information in the instances.
Functional Dependencies (FDs)

**Idea:** Express the fact that in a relation schema (values of) a set of attributes uniquely determine (values of) another set of attributes.
Functional Dependencies (FDs)

Idea: Express the fact that in a relation schema (values of) a set of attributes uniquely determine (values of) another set of attributes.

**Definition (Functional Dependency)**

Let $R$ be a relation schema, and $X, Y \subseteq R$ sets of attributes. The functional dependency

$$X \rightarrow Y$$

holds on $R$ if whenever an instance of $R$ contains two tuples $t$ and $u$ such that $t[X] = u[X]$ then it is also true that $t[Y] = u[Y]$.

We say that $X$ functionally determines $Y$ (in $R$).

Notation: $t[A_1, \ldots, A_k]$ means projection of tuple $t$ onto the attributes $A_1, \ldots, A_k$. In other words, $(t.A_1, \ldots, t.A_k)$. 
Consider the following relation schema:

```
| SIN | PNum | Hours | EName | PName | PLoc | Allowance |
```

- SIN determines employee name
  \[ \text{SIN} \rightarrow \text{EName} \]
- project number determines project name and location
  \[ \text{PNum} \rightarrow \text{PName}, \text{PLoc} \]
- allowances are always the same for the same number of hours at the same location
  \[ \text{PLoc}, \text{Hours} \rightarrow \text{Allowance} \]
Functional Dependencies and Keys

- Keys (as defined previously):
  - A superkey is a set of attributes such that no two tuples (in an instance) agree on their values for those attributes.
  - A candidate key is a minimal superkey.
  - A primary key is a candidate key chosen by the DBA.
### Functional Dependencies and Keys

- **Keys (as defined previously):**
  - A **superkey** is a set of attributes such that no two tuples (in an instance) agree on their values for those attributes.
  - A **candidate key** is a *minimal* superkey.
  - A **primary key** is a candidate key chosen by the DBA.

- **Relating keys and FDs:**
  - If $K \subseteq R$ is a superkey for relation schema $R$, then dependency $K \rightarrow R$ holds on $R$.
  - If dependency $K \rightarrow R$ holds on $R$ and we assume that $R$ does not contain duplicate tuples (i.e. relational model) then $K \subseteq R$ is a superkey for relation schema $R$. 
How do we know what additional FDs hold in a schema?

- The closure of the set of functional dependencies \( F \) (denoted \( F^+ \)) is the set of all functional dependencies that are satisfied by every relational instance that satisfies \( F \).

- Informally, \( F^+ \) includes all of the dependencies in \( F \), plus any dependencies they imply.
Logical implications can be derived by using inference rules called Armstrong’s axioms

- (reflexivity) \( Y \subseteq X \Rightarrow X \rightarrow Y \)
- (augmentation) \( X \rightarrow Y \Rightarrow XZ \rightarrow YZ \)
- (transitivity) \( X \rightarrow Y, Y \rightarrow Z \Rightarrow X \rightarrow Z \)

The axioms are

- sound (anything derived from \( F \) is in \( F^+ \))
- complete (anything in \( F^+ \) can be derived)
Reasoning About FDs

Logical implications can be derived by using inference rules called Armstrong’s axioms

- (reflexivity) \( Y \subseteq X \Rightarrow X \rightarrow Y \)
- (augmentation) \( X \rightarrow Y \Rightarrow XZ \rightarrow YZ \)
- (transitivity) \( X \rightarrow Y, Y \rightarrow Z \Rightarrow X \rightarrow Z \)

The axioms are

- sound (anything derived from \( F \) is in \( F^+ \))
- complete (anything in \( F^+ \) can be derived)

Additional rules can be derived

- (union) \( X \rightarrow Y, X \rightarrow Z \Rightarrow X \rightarrow YZ \)
- (decomposition) \( X \rightarrow YZ \Rightarrow X \rightarrow Y \)
Reasoning About FDs (example)

Example: \( F = \{ \)
\begin{align*}
&SIN, \ PNum \rightarrow \ Hours \\
&SIN \rightarrow \ EName \\
PNum \rightarrow \ PName, \ PLoc \\
&PLoc, \ Hours \rightarrow \ Allowance \}
\end{align*}

A derivation of \( SIN, \ PNum \rightarrow \ Allowance \):

1. \( SIN, \ PNum \rightarrow \ Hours \ (\in F) \)
2. \( PNum \rightarrow \ PName, \ PLoc \ (\in F) \)
3. \( PLoc, \ Hours \rightarrow \ Allowance \ (\in F) \)
4. \( SIN, \ PNum \rightarrow \ PNum \) (reflexivity)
5. \( SIN, \ PNum \rightarrow \ PName, \ PLoc \) (transitivity, 4 and 2)
6. \( SIN, \ PNum \rightarrow \ PLoc \) (decomposition, 5)
7. \( SIN, \ PNum \rightarrow \ PLoc, \ Hours \) (union, 6, 1)
8. \( SIN, \ PNum \rightarrow \ Allowance \) (transitivity, 7 and 3)
Computing Attribute Closures

- There is a more efficient way of using Armstrong’s axioms, if we only want to derive the maximal set of attributes functionally determined by some $X$ (called the attribute closure of $X$).

function $ComputeX^+(X, F)$$begin$
  $X^+ := X;$
  while true do$
    if there exists $(Y \rightarrow Z) \in F$ such that$
      (1) Y \subseteq X^+$, and$
      (2) Z \not\subseteq X^+$
    then $X^+ := X^+ \cup Z$
    else exit;
  return $X^+$;$$end$
Let $R$ be a relational schema and $F$ a set of functional dependencies on $R$. Then

**Theorem:** $X$ is a superkey of $R$ if and only if

$$\text{Compute}X^+(X, F) = R$$

**Theorem:** $X \rightarrow Y \in F^+$ if and only if

$$Y \subseteq \text{Compute}X^+(X, F)$$
Attribute Closure Example

Example:  \( F = \{ \text{SIN} \rightarrow \text{EName}, \text{PNum} \rightarrow \text{PName}, \text{PLoc}, \text{PLoc}, \text{Hours} \rightarrow \text{Allowance} \} \)

Compute \( X^+ (\{\text{Pnum}, \text{Hours}\}, F) \):

<table>
<thead>
<tr>
<th>FD</th>
<th>( X^+ )</th>
</tr>
</thead>
<tbody>
<tr>
<td>initial</td>
<td>Pnum,Hours</td>
</tr>
<tr>
<td>Pnum \rightarrow Pname, Ploc</td>
<td>Pnum,Hours,Pname, Ploc</td>
</tr>
<tr>
<td>PLoc, Hours \rightarrow Allowance</td>
<td>Pnum,Hours,Pname, Ploc, Allowance</td>
</tr>
</tbody>
</table>
Definition (Schema Decomposition)

Let $R$ be a relation schema (= set of attributes). The collection $\{R_1, \ldots, R_n\}$ of relation schemas is a decomposition of $R$ if

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

A good decomposition does not

- lose information
- complicate checking of constraints
- contain anomalies (or at least contains fewer anomalies)
We should be able to construct the instance of the original table from the instances of the tables in the decomposition.

**Example:** Consider replacing

<table>
<thead>
<tr>
<th>Marks</th>
<th>Student</th>
<th>Assignment</th>
<th>Group</th>
<th>Mark</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ann</td>
<td>A1</td>
<td>G1</td>
<td>80</td>
<td></td>
</tr>
<tr>
<td>Ann</td>
<td>A2</td>
<td>G3</td>
<td>60</td>
<td></td>
</tr>
<tr>
<td>Bob</td>
<td>A1</td>
<td>G2</td>
<td>60</td>
<td></td>
</tr>
</tbody>
</table>

by decomposing (i.e. projecting) into two tables.

<table>
<thead>
<tr>
<th>SGM</th>
<th>Student</th>
<th>Group</th>
<th>Mark</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ann</td>
<td>G1</td>
<td>80</td>
<td></td>
</tr>
<tr>
<td>Ann</td>
<td>G3</td>
<td>60</td>
<td></td>
</tr>
<tr>
<td>Bob</td>
<td>G2</td>
<td>60</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>AM</th>
<th>Assignment</th>
<th>Mark</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1</td>
<td>80</td>
<td></td>
</tr>
<tr>
<td>A2</td>
<td>60</td>
<td></td>
</tr>
<tr>
<td>A1</td>
<td>60</td>
<td></td>
</tr>
</tbody>
</table>
But computing the natural join of SGM and AM produces

<table>
<thead>
<tr>
<th>Student</th>
<th>Assignment</th>
<th>Group</th>
<th>Mark</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ann</td>
<td>A1</td>
<td>G1</td>
<td>80</td>
</tr>
<tr>
<td>Ann</td>
<td>A2</td>
<td>G3</td>
<td>60</td>
</tr>
<tr>
<td>Ann</td>
<td>A1</td>
<td>G3</td>
<td>60</td>
</tr>
<tr>
<td>Bob</td>
<td>A2</td>
<td>G2</td>
<td>60</td>
</tr>
<tr>
<td>Bob</td>
<td>A1</td>
<td>G2</td>
<td>60</td>
</tr>
</tbody>
</table>

...and we get extra data (**spurious tuples**). We would therefore lose information if we were to replace Marks by SGM and AM.
But computing the natural join of SGM and AM produces

<table>
<thead>
<tr>
<th>Student</th>
<th>Assignment</th>
<th>Group</th>
<th>Mark</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ann</td>
<td>A1</td>
<td>G1</td>
<td>80</td>
</tr>
<tr>
<td>Ann</td>
<td>A2</td>
<td>G3</td>
<td>60</td>
</tr>
<tr>
<td>Ann</td>
<td>A1</td>
<td>G3</td>
<td>60</td>
</tr>
<tr>
<td>Bob</td>
<td>A2</td>
<td>G2</td>
<td>60</td>
</tr>
<tr>
<td>Bob</td>
<td>A1</td>
<td>G2</td>
<td>60</td>
</tr>
</tbody>
</table>

...and we get extra data (**spurious tuples**). We would therefore lose information if we were to replace Marks by SGM and AM.

If re-joining SGM and AM would **always** produce exactly the tuples in Marks, then we call SGM and AM a **lossless-join decomposition**.
A decomposition \( \{R_1, R_2\} \) of \( R \) is lossless if and only if the common attributes of \( R_1 \) and \( R_2 \) form a superkey for either schema, that is

\[
R_1 \cap R_2 \rightarrow R_1 \quad \text{or} \quad R_1 \cap R_2 \rightarrow R_2
\]
A decomposition \( \{R_1, R_2\} \) of \( R \) is lossless if and only if the common attributes of \( R_1 \) and \( R_2 \) form a superkey for either schema, that is

\[
R_1 \cap R_2 \rightarrow R_1 \quad \text{or} \quad R_1 \cap R_2 \rightarrow R_2
\]

**Example:** In the previous example we had

\[
R = \{ \text{Student, Assignment, Group, Mark} \} ,
F = \{(\text{Student, Assignment} \rightarrow \text{Group, Mark})\} ,
\]

\[
R_1 = \{ \text{Student, Group, Mark} \} ,
R_2 = \{ \text{Assignment, Mark} \}
\]

Decomposition \( \{R_1, R_2\} \) is lossy because \( R_1 \cap R_2 \) (\( = \{\text{Mark}\} \)) is not a superkey of either \( \{\text{Student, Group, Mark}\} \) or \( \{\text{Assignment, Mark}\} \)
Dependency Preservation

How do we test/enforce constraints on the decomposed schema?

Example:
A table for a company database could be

\[\text{FD1: } \text{Proj} \rightarrow \text{Dept}, \text{FD2: Dept} \rightarrow \text{Div}, \text{and FD3: Proj} \rightarrow \text{Div}\]

and two decompositions

\[D_1 = \{R_1[\text{Proj, Dept}], R_2[\text{Dept, Div}]\}\]

\[D_2 = \{R_1[\text{Proj, Dept}], R_3[\text{Proj, Div}]\}\]

Both are lossless. (Why?)
Dependency Preservation

How do we test/enforce constraints on the decomposed schema?

Example: A table for a company database could be

\[
\begin{array}{ccc}
R \\
\text{Proj} & \text{Dept} & \text{Div}
\end{array}
\]

FD1: Proj \rightarrow Dept,
FD2: Dept \rightarrow Div, and
FD3: Proj \rightarrow Div

and two decompositions

\[
D_1 = \{R1[Proj, Dept], R2[Dept, Div]\}
\]
\[
D_2 = \{R1[Proj, Dept], R3[Proj, Div]\}
\]

Both are lossless. (Why?)
Dependency Preservation (cont.)

Which decomposition is better?

- Decomposition $D_1$ lets us test FD1 on table R1 and FD2 on table R2; if they are both satisfied, FD3 is automatically satisfied.

- In decomposition $D_2$ we can test FD1 on table R1 and FD3 on table R3. Dependency FD2 is an interrelational constraint: testing it requires joining tables R1 and R3.
Dependency Preservation (cont.)

Which decomposition is *better*?

- Decomposition $D_1$ lets us test FD1 on table R1 and FD2 on table R2; if they are both satisfied, FD3 is automatically satisfied.

- In decomposition $D_2$ we can test FD1 on table R1 and FD3 on table R3. Dependency FD2 is an *interrelational constraint*: testing it requires joining tables R1 and R3.

$\Rightarrow D_1$ is better!
Which decomposition is better?

- Decomposition $D_1$ lets us test FD1 on table R1 and FD2 on table R2; if they are both satisfied, FD3 is automatically satisfied.

- In decomposition $D_2$ we can test FD1 on table R1 and FD3 on table R3. Dependency FD2 is an interrelational constraint: testing it requires joining tables R1 and R3.

$\Rightarrow D_1$ is better!

Given a schema $R$ and a set of functional dependencies $F$, decomposition $D = \{R_1, \ldots, R_n\}$ of $R$ is dependency preserving if there is an equivalent set of functional dependencies $F'$, none of which is interrelational in $D$. 
What is a “good” relational database schema?

Rule of thumb: Independent facts in separate tables:

“Each relation schema should consist of a primary key and a set of mutually independent attributes”

This is achieved by transforming a schema into a normal form.
What is a “good” relational database schema?

Rule of thumb: Independent facts in separate tables:

“Each relation schema should consist of a primary key and a set of mutually independent attributes”

This is achieved by transforming a schema into a normal form.

Goals:

- Intuitive and straightforward transformation
- Anomaly-free/Nonredundant representation of data
What is a “good” relational database schema?

Rule of thumb: Independent facts in separate tables:

“Each relation schema should consist of a primary key and a set of mutually independent attributes”

This is achieved by transforming a schema into a normal form.

Goals:

- Intuitive and straightforward transformation
- Anomaly-free/Nonredundant representation of data

Normal Forms based on Functional Dependencies:

- Boyce-Codd Normal Form (BCNF)
- Third Normal Form (3NF)
• BCNF formalizes the goal that in a good database schema, independent relationships are stored in separate tables.

• Given a database schema and a set of functional dependencies for the attributes in the schema, we can determine whether the schema is in BCNF. A database schema is in BCNF if each of its relation schemas is in BCNF.

• Informally, a relation schema is in BCNF if and only if any group of its attributes that functionally determines any others of its attributes functionally determines all others, i.e., that group of attributes is a superkey of the relation.
Formal Definition of BCNF

Let \( R \) be a relation schema and \( F \) a set of functional dependencies.

Schema \( R \) is in BCNF (w.r.t. \( F' \)) if and only if whenever \((X \rightarrow Y) \in F^+\) and \(XY \subseteq R\), then either

- \((X \rightarrow Y)\) is trivial (i.e., \(Y \subseteq X\)), or
- \(X\) is a superkey of \(R\)

A database schema \(\{R_1, \ldots, R_n\}\) is in BCNF if each relation schema \(R_i\) is in BCNF.
Why does BCNF avoid redundancy? Consider:

<table>
<thead>
<tr>
<th>Sno</th>
<th>Sname</th>
<th>City</th>
<th>Pno</th>
<th>Pname</th>
<th>Price</th>
</tr>
</thead>
</table>

- The following functional dependency holds:
  
  \[ \text{Sno} \rightarrow \text{Sname, City} \]

- Therefore, supplier name “Magna” and city “Ajax” must be repeated for each item supplied by supplier S1.
Why does BCNF avoid redundancy? Consider:

<table>
<thead>
<tr>
<th>Sno</th>
<th>Sname</th>
<th>City</th>
<th>Pno</th>
<th>Pname</th>
<th>Price</th>
</tr>
</thead>
</table>

The following functional dependency holds:

\[ \text{Sno} \rightarrow \text{Sname, City} \]

Therefore, supplier name “Magna” and city “Ajax” must be repeated for each item supplied by supplier S1.

Assume the above FD holds over a schema \( R \) that is in BCNF. This implies that:

- Sno is a superkey for \( R \)
- each Sno value appears on one row only
- no need to repeat Sname and City values
function DecomposeBCNF(R, F)
begin
    Result := \{R\};
    while some $R_i \in Result$ and $(X \rightarrow Y) \in F^+$
        violate the BCNF condition do begin
        Replace $R_i$ by $R_i - (Y - X)$;
        Add $\{X, Y\}$ to Result;
    end;
    return Result;
end
Lossless-Join BCNF Decomposition

- No *efficient* procedure to do this exists.
- Results depend on sequence of FDs used to decompose the relations.
- It is possible that no lossless join dependency preserving BCNF decomposition exists.
  - Consider $R = \{A, B, C\}$ and $F = \{AB \rightarrow C, C \rightarrow B\}$. 
BCNF Decomposition - An Example

- $R = \{\text{Sno}, \text{Sname}, \text{City}, \text{Pno}, \text{Pname}, \text{Price}\}$
- functional dependencies:
  \(\text{Sno} \rightarrow \text{Sname}, \text{City}\)
  \(\text{Pno} \rightarrow \text{Pname}\)
  \(\text{Sno, Pno} \rightarrow \text{Price}\)
- This schema is not in BCNF because, for example, Sno determines Sname and City, but is not a superkey of $R$. 
The complete schema is now

\[ R_1 = \{ \text{Sno}, \text{Sname}, \text{City} \} \]
\[ R_2 = \{ \text{Sno}, \text{Pno}, \text{Price} \} \]
\[ R_3 = \{ \text{Pno}, \text{Pname} \} \]

This schema is a lossless-join, BCNF decomposition of the original schema \( R \).
Third Normal Form (3NF)

Schema $R$ is in 3NF (w.r.t. $F$) if and only if whenever $(X \rightarrow Y) \in F^+$ and $XY \subseteq R$, then either

- $(X \rightarrow Y)$ is trivial, or
- $X$ is a superkey of $R$, or
- each attribute in $Y - X$ is contained in a candidate key of $R$

A database schema $\{R_1, \ldots, R_n\}$ is in 3NF if each relation schema $R_i$ is in 3NF.
Schema \( R \) is in **3NF** (w.r.t. \( F \)) if and only if whenever \((X \rightarrow Y) \in F^+\) and \(XY \subseteq R\), then either
- \((X \rightarrow Y)\) is trivial, or
- \(X\) is a superkey of \(R\), or
- each attribute in \(Y - X\) is contained in a candidate key of \(R\).

A database schema \( \{R_1, \ldots, R_n\} \) is in 3NF if each relation schema \( R_i \) is in 3NF.

- 3NF is looser than BCNF
  - allows more redundancy
  - e.g. \( R = \{A, B, C\} \) and \( F = \{AB \rightarrow C, C \rightarrow B\} \).
Third Normal Form (3NF)

Schema $R$ is in 3NF (w.r.t. $F$) if and only if whenever $(X \rightarrow Y) \in F^+$ and $XY \subseteq R$, then either

- $(X \rightarrow Y)$ is trivial, or
- $X$ is a superkey of $R$, or
- each attribute in $Y - X$ is contained in a candidate key of $R$

A database schema $\{R_1, \ldots, R_n\}$ is in 3NF if each relation schema $R_i$ is in 3NF.

- 3NF is looser than BCNF
  - allows more redundancy
  - e.g. $R = \{A, B, C\}$ and $F = \{AB \rightarrow C, C \rightarrow B\}$.
- lossless-join, dependency-preserving decomposition into 3NF relation schemas always exists.
Definition: Two sets of dependencies $F$ and $G$ are equivalent iff $F^+ = G^+$.

There are different sets of functional dependencies that have the same logical implications. Simple sets are desirable.
Definition: Two sets of dependencies $F$ and $G$ are equivalent iff $F^+ = G^+$.

There are different sets of functional dependencies that have the same logical implications. Simple sets are desirable.

Definition: A set of dependencies $G$ is minimal if

1. every right-hand side of an dependency in $F$ is a single attribute.
2. for no $X \rightarrow A$ is the set $F - \{X \rightarrow A\}$ equivalent to $F$.
3. for no $X \rightarrow A$ and $Z$ a proper subset of $X$ is the set $F - \{X \rightarrow A\} \cup \{Z \rightarrow A\}$ equivalent to $F$. 

Theorem: For every set of dependencies $F$ there is an equivalent minimal set of dependencies (minimal cover).
Definition: Two sets of dependencies $F$ and $G$ are equivalent iff $F^+ = G^+$.

There are different sets of functional dependencies that have the same logical implications. Simple sets are desirable.

Definition: A set of dependencies $G$ is minimal if

1. every right-hand side of an dependency in $F$ is a single attribute.
2. for no $X \rightarrow A$ is the set $F - \{X \rightarrow A\}$ equivalent to $F$.
3. for no $X \rightarrow A$ and $Z$ a proper subset of $X$ is the set $F - \{X \rightarrow A\} \cup \{Z \rightarrow A\}$ equivalent to $F$.

Theorem: For every set of dependencies $F$ there is an equivalent minimal set of dependencies (minimal cover).
A minimal cover for $F$ can be computed in three steps. Note that each step must be repeated until it no longer succeeds in updating $F$.

**Step 1.**
Replace $X \rightarrow YZ$ with the pair $X \rightarrow Y$ and $X \rightarrow Z$.

**Step 2.**
Remove $A$ from the left-hand-side of $X \rightarrow B$ in $F$ if $B$ is in $ComputeX^+(X - \{A\}, F)$.

**Step 3.**
Remove $X \rightarrow A$ from $F$ if $A \in ComputeX^+(X, F - \{X \rightarrow A\})$. 
Dependency-Preserving 3NF Decomposition

Idea: Decompose into 3NF relations and then “repair”

function Decompose3NF(R, F)
begin
    Result := {R};
    while some $R_i \in Result$ and $(X \rightarrow Y) \in F^+$ violate the 3NF condition do begin
        Replace $R_i$ by $R_i - (Y - X)$;
        Add $\{X, Y\}$ to Result;
    end;
    $N := (a \ minimal \ cover \ for \ F) - (\bigcup_i F_i)^+$
    for each $(X \rightarrow Y) \in N$ do
        Add $\{X, Y\}$ to Result;
    end;
    return Result;
end
Dep-Preserving 3NF Decomposition - An Example

- $R = \{\text{Sno, Sname, City, Pno, Pname, Price}\}$
- Functional dependencies:
  - $\text{Sno} \rightarrow \text{Sname, City}$
  - $\text{Pno} \rightarrow \text{Pname}$
  - $\text{Sno, Pno} \rightarrow \text{Price}$
  - $\text{Sno, Pname} \rightarrow \text{Price}$
- Minimal cover:
  - $\text{Sno} \rightarrow \text{Sname}$
  - $\text{Pno} \rightarrow \text{Pname}$
  - $\text{Sno} \rightarrow \text{City}$
  - $\text{Sno, Pname} \rightarrow \text{Price}$

- Add relation to preserve missing dependency
  - $R_4 = \{\text{Sno, Pname, Price}\}$
- $R = \{\text{Sno, Sname, City, Pno, Pname, Price}\}$
- Functional dependencies:
  - $\text{Sno} \rightarrow \text{Sname, City}$
  - $\text{Pno} \rightarrow \text{Pname}$
  - $\text{Sno, Pno} \rightarrow \text{Price}$
  - $\text{Sno, Pname} \rightarrow \text{Price}$
- Following same decomposition tree as BCNF example:
  - $R_1 = \{\text{Sno, Sname, City}\}$
  - $R_2 = \{\text{Sno, Pno, Price}\}$
  - $R_3 = \{\text{Pno, Pname}\}$
Dep-Preserving 3NF Decomposition - An Example

- \( R = \{\text{Sno, Sname, City, Pno, Pname, Price}\} \)
- Functional dependencies:
  - \( \text{Sno} \rightarrow \text{Sname, City} \)
  - \( \text{Pno} \rightarrow \text{Pname} \)
  - \( \text{Sno, Pno} \rightarrow \text{Price} \)
  - \( \text{Sno, Pname} \rightarrow \text{Price} \)
- Following same decomposition tree as BCNF example:
  - \( R_1 = \{\text{Sno, Sname, City}\} \)
  - \( R_2 = \{\text{Sno, Pno, Price}\} \)
  - \( R_3 = \{\text{Pno, Pname}\} \)
- Minimal cover:
  - \( \text{Sno} \rightarrow \text{Sname} \)
  - \( \text{Pno} \rightarrow \text{Pname} \)
  - \( \text{Sno} \rightarrow \text{City} \)
  - \( \text{Sno, Pname} \rightarrow \text{Price} \)
Dep-Preserving 3NF Decomposition - An Example

- \( R = \{\text{Sno, Sname, City, Pno, Pname, Price}\} \)
- Functional dependencies:
  \( \text{Sno} \rightarrow \text{Sname, City} \)
  \( \text{Pno} \rightarrow \text{Pname} \)
  \( \text{Sno, Pno} \rightarrow \text{Price} \)
  \( \text{Sno, Pname} \rightarrow \text{Price} \)
- Following same decomposition tree as BCNF example:
  \( R_1 = \{\text{Sno, Sname, City}\} \)
  \( R_2 = \{\text{Sno, Pno, Price}\} \)
  \( R_3 = \{\text{Pno, Pname}\} \)
- Minimal cover:
  \( \text{Sno} \rightarrow \text{Sname} \)
  \( \text{Pno} \rightarrow \text{Pname} \)
  \( \text{Sno} \rightarrow \text{City} \)
  \( \text{Sno, Pname} \rightarrow \text{Price} \)
- Add relation to preserve missing dependency
  \( R_4 = \{\text{Sno, Pname, Price}\} \)
A lossless-join 3NF decomposition that is dependency preserving can be efficiently computed.

```
function Synthesize3NF(R, F)
begin
    Result := ∅;
    F' := a minimal cover for F;
    for each (X → Y) ∈ F' do
        Result := Result ∪ {XY};
    if there is no Ri ∈ Result such that
        Ri contains a candidate key for R then begin
            compute a candidate key K for R;
            Result := Result ∪ {K};
        end;
    return Result;
end
```
3NF Synthesis - An Example

- \( R = \{\text{Sno, Sname, City, Pno, Pname, Price}\} \)
- Functional dependencies:
  - \( \text{Sno} \rightarrow \text{Sname, City} \)  
  - \( \text{Pno} \rightarrow \text{Pname} \)  
  - \( \text{Sno, Pno} \rightarrow \text{Price} \)  
  - \( \text{Sno, Pname} \rightarrow \text{Price} \)  

- Add relation for candidate key
  - \( R_5 = \{\text{Sno, Pno}\} \)

- Optimization: combine relations \( R_1 \) and \( R_2 \) (same key)
3NF Synthesis - An Example

- \( R = \{\text{Sno}, \text{Sname}, \text{City}, \text{Pno}, \text{Pname}, \text{Price}\} \)

- Functional dependencies:
  
  \[
  \begin{align*}
  \text{Sno} & \rightarrow \text{Sname}, \text{City} \\
  \text{Sno}, \text{Pno} & \rightarrow \text{Price} \\
  \text{Pno} & \rightarrow \text{Pname} \\
  \text{Sno}, \text{Pname} & \rightarrow \text{Price}
  \end{align*}
  \]

- Minimal cover:

  \[
  \begin{align*}
  &\text{Sno} \rightarrow \text{Sname} & R_1 = \{\text{Sno, Sname}\} \\
  &\text{Sno} \rightarrow \text{City} & R_2 = \{\text{Sno, City}\} \\
  &\text{Pno} \rightarrow \text{Pname} & R_3 = \{\text{Pno, Pname}\} \\
  &\text{Sno}, \text{Pname} \rightarrow \text{Price} & R_4 = \{\text{Sno, Pname, Price}\}
  \end{align*}
  \]
3NF Synthesis - An Example

- \( R = \{\text{Sno, Sname, City, Pno, Pname, Price}\} \)
- Functional dependencies:
  - \( \text{Sno} \rightarrow \text{Sname, City} \)
  - \( \text{Pno} \rightarrow \text{Pname} \)
  - \( \text{Sno, Pno} \rightarrow \text{Price} \)
  - \( \text{Sno, Pname} \rightarrow \text{Price} \)
- Minimal cover:
  - \( \text{Sno} \rightarrow \text{Sname} \)
  - \( \text{Sno} \rightarrow \text{City} \)
  - \( \text{Pno} \rightarrow \text{Pname} \)
  - \( \text{Sno, Pname} \rightarrow \text{Price} \)
  - \( R_1 = \{\text{Sno}, \text{Sname}\} \)
  - \( R_2 = \{\text{Sno}, \text{City}\} \)
  - \( R_3 = \{\text{Pno}, \text{Pname}\} \)
  - \( R_4 = \{\text{Sno, Pname, Price}\} \)
- Add relation for candidate key \( R_5 = \{\text{Sno, Pno}\} \)
3NF Synthesis - An Example

- \( R = \{\text{Sno, Sname, City, Pno, Pname, Price}\} \)

- Functional dependencies:
  - \( \text{Sno} \rightarrow \text{Sname}, \text{City} \)
  - \( \text{Pno} \rightarrow \text{Pname} \)
  - \( \text{Sno, Pno} \rightarrow \text{Price} \)
  - \( \text{Sno, Pname} \rightarrow \text{Price} \)

- Minimal cover:
  - \( \text{Sno} \rightarrow \text{Sname} \)
  - \( \text{Sno} \rightarrow \text{City} \)
  - \( \text{Pno} \rightarrow \text{Pname} \)
  - \( \text{Sno, Pname} \rightarrow \text{Price} \)
  - \( R_1 = \{\text{Sno, Sname}\} \)
  - \( R_2 = \{\text{Sno, City}\} \)
  - \( R_3 = \{\text{Pno, Pname}\} \)
  - \( R_4 = \{\text{Sno, Pname, Price}\} \)

- Add relation for candidate key \( R_5 = \{\text{Sno, Pno}\} \)

- Optimization: combine relations \( R_1 \) and \( R_2 \) (same key)
Functional dependencies provide clues towards elimination of (some) redundancies in a relational schema.

Goals: to decompose relational schemas in such a way that the decomposition is

(1) lossless-join
(2) dependency preserving
(3) BCNF (and if we fail here, at least 3NF)