Schema Refinement and Normal Forms

Chapter 19
1. The Problem

- Given an application and a RDBMS find the set of tables that best describe that application.

- Schema design
1.1. Example of Bad Database Design

Assume we want to store information about

- suppliers, their names and city they leave, departments they supply, and
- parts currently supplied by each supplier,

<table>
<thead>
<tr>
<th>s</th>
<th>name</th>
<th>city</th>
<th>p</th>
<th>name</th>
<th>qty</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>smith</td>
<td>London</td>
<td>301</td>
<td>bolt</td>
<td>20</td>
<td>84</td>
</tr>
<tr>
<td>5</td>
<td>nick</td>
<td>NY</td>
<td>500</td>
<td>screw</td>
<td>50</td>
<td>50</td>
</tr>
<tr>
<td>2</td>
<td>steve</td>
<td>Boston</td>
<td>20</td>
<td>nut</td>
<td>10</td>
<td>20</td>
</tr>
<tr>
<td>5</td>
<td>nick</td>
<td>NY</td>
<td>400</td>
<td>tire</td>
<td>40</td>
<td>85</td>
</tr>
<tr>
<td>5</td>
<td>nick</td>
<td>NY</td>
<td>301</td>
<td>bolt</td>
<td>10</td>
<td>86</td>
</tr>
</tbody>
</table>

relation parts-suppliers

**Problems:**

- **Redundancy**
  suppliers name and city are repeated once for each item supplied (it leads to update anomalies).

- **Insertion Anomalies** *(inability to represent certain information)*
  We cannot insert the name and city of a supplier that does not currently supply at least one part.

- **Deletion Anomalies** *(possibility of losing data by deleting other data)*
  If a part is not currently supplied by any supplier, we lose info about parts.

**UPDATE Anomalies**
1.2. Careless Decomposition

Assume this decomposition:

\[ \text{part}(p, \text{name}, \text{qty}, d) = \pi_{p, \text{name}, \text{qty}, d}(\text{part-supplier}) \]

and

\[ \text{supplier}(s, \text{name}, \text{city}, \text{qty}) = \pi_{s, \text{name}, \text{city}, \text{qty}}(\text{part-supplier}) \]

Those two relations are as follows:

<table>
<thead>
<tr>
<th>p</th>
<th>name</th>
<th>qty</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>301</td>
<td>bolt</td>
<td>20</td>
<td>84</td>
</tr>
<tr>
<td>500</td>
<td>screw</td>
<td>50</td>
<td>50</td>
</tr>
<tr>
<td>20</td>
<td>nut</td>
<td>10</td>
<td>20</td>
</tr>
<tr>
<td>400</td>
<td>tire</td>
<td>40</td>
<td>85</td>
</tr>
<tr>
<td>301</td>
<td>bolt</td>
<td>10</td>
<td>86</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>s</th>
<th>name</th>
<th>city</th>
<th>qty</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>smith</td>
<td>London</td>
<td>20</td>
</tr>
<tr>
<td>5</td>
<td>nick</td>
<td>NY</td>
<td>50</td>
</tr>
<tr>
<td>5</td>
<td>nick</td>
<td>NY</td>
<td>40</td>
</tr>
<tr>
<td>5</td>
<td>nick</td>
<td>NY</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>steve</td>
<td>Boston</td>
<td>10</td>
</tr>
</tbody>
</table>

Suppliers...
Parts...
## Supplier & part

<table>
<thead>
<tr>
<th>S</th>
<th>Name</th>
<th>City</th>
<th>P</th>
<th>Product</th>
<th>Qty</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>Smith</td>
<td>London</td>
<td>301</td>
<td>bolt</td>
<td>20</td>
<td>84</td>
</tr>
<tr>
<td>5</td>
<td>Nick</td>
<td>NY</td>
<td>300</td>
<td>screw</td>
<td>50</td>
<td>50</td>
</tr>
<tr>
<td>5</td>
<td>Nick</td>
<td>NY</td>
<td>20</td>
<td>nut</td>
<td>10</td>
<td>20</td>
</tr>
<tr>
<td>2</td>
<td>Steve</td>
<td>Boston</td>
<td>20</td>
<td>nut</td>
<td>10</td>
<td>20</td>
</tr>
<tr>
<td>5</td>
<td>Nick</td>
<td>NY</td>
<td>400</td>
<td>hinge</td>
<td>40</td>
<td>85</td>
</tr>
<tr>
<td>5</td>
<td>Nick</td>
<td>NY</td>
<td>301</td>
<td>bolt</td>
<td>10</td>
<td>86</td>
</tr>
<tr>
<td>2</td>
<td>Steve</td>
<td>Boston</td>
<td>301</td>
<td>bolt</td>
<td>10</td>
<td>86</td>
</tr>
</tbody>
</table>

*NOT EXISTING IN ORIGINAL RELATION (SPURIOUS TUPLES)*

**LOSSY JOIN**
1.3. Lossy Decomposition

Assume we want to find out who supplies what. It appears that

\[ part \times supplier \neq part \times supplier \] \hspace{1cm} \text{Wrong}

It is wrong, because we have tuples that do not exist in the original relation.

\[ \Rightarrow \text{although we have more tuples, we actually have less information, since we can no longer represent the original relation.} \]

We say that this decomposition is a \textit{lossy} decomposition. (\textit{lossy JOIN})

\[ \square \text{A decomposition that is not lossy is referred as \textit{lossless} decomposition.} \]

(\textit{lossless JOIN})
Relational DB design is a decomposition problem.

Start from a universal relation that contains all attributes and decompose into relations so that the resulting relations obey some desirable properties (i.e., no update anomalies, lossless join etc.).

How to formalize this procedure?

Use real-world constraints of data (Example: independent facts should go into separate relations (then no update anomalies)).

One way to represent these constraints are functional dependencies (FDs).
The Evils of Redundancy

- *Redundancy* is at the root of several problems associated with relational schemas:
  - redundant storage, insert/delete/update anomalies

- Integrity constraints, in particular *functional dependencies*, can be used to identify schemas with such problems and to suggest refinements.

- Main refinement technique: *decomposition* (replacing ABCD with, say, AB and BCD, or ACD and ABD).

- Decomposition should be used judiciously:
  - Is there reason to decompose a relation?
  - What problems (if any) does the decomposition cause?
Using FD's we will introduce normal forms (2NF, 3NF, BCNF, etc.)

i.e. formal ways to say if a given database design is good or bad.

So first we discuss what an FD is and properties of FD's.

Then we will discuss Normal Forms.
Formally, if \( X, Y \) are subsets of \( R \) and \( X \rightarrow Y \), then,

\[
\forall t_1, t_2 \in R : \quad t_1[x] = t_2[x] \Rightarrow t_1[y] = t_2[y]
\]

Thus an FD \( X \rightarrow Y \) is a constraint that says: if 2 tuples agree on their \( X \) values they should also agree on their \( Y \) values.

Or equivalently:

Knowing the value of \( X \) implies the value of \( Y \) is known.

Another example:

\( \text{child} \rightarrow \text{mother} \)

but \( \text{mother} \not\rightarrow \text{child} \)
2.1. Important Observations

- Functional dependencies are assertions about the real world; they cannot be proved.
- Functional dependencies are statements about all possible instances of a relation schema.

Example:
Consider, \( R = \{A,B,C\} \) and \( r(R) \):

\[
\begin{array}{ccc}
| A | B | C | \\
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>
\end{array}
\]

Can we deduce that \( A \rightarrow C \) holds?
Can we deduce that \( A \rightarrow B \) does not hold?
Hence FDs are given to us from the application.

Example: \( R(A,B,C,D) \)

\[ AB \rightarrow D \] (given)

\[
\begin{array}{c|c|c|c}
  r(R): & A & B & C & D \\
  \hline
  x_1 & b_1 & c_1 & d_1 \\
  x_2 & b_2 & c_1 & d_2 \\
  x_2 & b_2 & c_2 & d_2 \\
  x_2 & b_3 & c_2 & d_3 \\
  x_3 & b_3 & c_2 & d_4 \\
\end{array}
\]

\textbf{NOTE:} If \( X \) is a key for \( R \) then \( X \rightarrow R \)

Why? \( X \) is a key: \( R(t_1) \neq R(t_2) \Rightarrow X(t_1) \neq X(t_2) \)

or equivalently:

\[ X(t_1) = X(t_2) \Rightarrow R(t_1) = R(t_2) \]
2. Functional Dependencies

Let $R = \{ R_1, R_2, \cdots, R_n \}$ be a relation schema and $X, Y$ subsets of $R$.

- We say $X \rightarrow Y$

(read: $X$ functionally determines $Y$ or $Y$ functionally depends on $X$)

if for any instance $r(R)$ it is not possible to have two tuples that agree in the components of all attributes in $X$ and disagree in one or more components of $Y$.

- A functional dependency $X \rightarrow Y$ is called trivial if it is true for any $X$ and $Y$ of any relation, regardless of $X, Y$ semantics.

Example: $A \rightarrow A$ is satisfied for any attribute of any relation.

In general,

if $Y \subseteq X$ then $X \rightarrow Y$ is trivial.
Example: Constraints on Entity Set

- Consider relation obtained from Hourly_Emps:
  - Hourly_Emps (ssn, name, lot, rating, hrly_wages, hrs_worked)

- **Notation**: We will denote this relation schema by listing the attributes: SNLRWH
  - This is really the set of attributes \{S,N,L,R,W,H\}.
  - Sometimes, we will refer to all attributes of a relation by using the relation name. (e.g., Hourly_Emps for SNLRWH)

- Some FDs on Hourly_Emps:
  - ssn is the key: S → SNLRWH
  - rating determines hrly_wages: R → W
Example (Contd.)

- Problems due to $R \rightarrow W$:
  - **Update anomaly**: Can we change $W$ in just the 1st tuple of SNLRWH?
  - **Insertion anomaly**: What if we want to insert an employee and don't know the hourly wage for his rating?
  - **Deletion anomaly**: If we delete all employees with rating 5, we lose the information about the wage for rating 5!

Will 2 smaller tables be better?

---

Database Management Systems, 3rd ed, R. Ramakrishnan and J. Gehrke
2.3. Inference Rules for FDs

In the following

- \( R = \{ A_1, A_2, \ldots, A_n \} \) is a relation schema
- \( X, Y, W, Z \) subsets of \( R \)
- \( XY \) is a shorthand for \( X \cup Y \)

1. Reflexivity Rule
   If \( Y \subseteq X \) than \( X \rightarrow Y \). This rule gives trivial dependencies.

2. Augmentation Rule
   If \( X \rightarrow Y \) then \( XZ \rightarrow YZ \)

3. Transitivity Rule
   If \( X \rightarrow Y \) and \( Y \rightarrow Z \) then \( X \rightarrow Z \).

4. Union Rule
   If \( X \rightarrow Y \) and \( X \rightarrow Z \) then \( X \rightarrow YZ \).

5. Pseudotransitivity Rule
   If \( X \rightarrow Y \) and \( WY \rightarrow Z \) then \( XW \rightarrow Z \).

6. Decomposition Rule
   If \( X \rightarrow Y \) and \( Z \subseteq Y \) then \( X \rightarrow Z \)
How to get all facts out of a set of FDs $F$, (i.e. how to get $F^+$)?

Use Armstrong's rules

- Reflexivity
- Augmentation
- Transitivity

Basic rules. They are **SOUND** (i.e. do not produce any incorrect FD) and **COMPLETE** (i.e. they produce all FD's in $F^+$ from $F$)

In addition & for ease of use we have:
- Union
- Pseudotransitivity
- Decomposition
2.2. Equivalent Sets of Functional Dependencies

- The *closure* of a set of functional dependencies $F$, denoted by $F^+$, is the set of all functional dependencies implied by $F$.

- Two sets $F, G$ of functional dependencies are equivalent iff $F^+ = G^+$.

  - To test whether $F \equiv G$
    - each member of $F$ must be a member of $G^+$ and each member of $G$ must also be a member of $F^+$.
    - If there is a dependency $X \rightarrow Y$ in $F$ which does not appear in $G^+$ or vice versa, then $F^+ \neq G^+$.

A trivial equivalence is the one between $F$ and $F^+$,

**Definition:** $F$ covers $G$

if $G^+ \subseteq F^+$

(i.e. every FD of $G$ can be inferred from $F$)

$F \equiv G \iff F$ covers $G$ and $G$ covers $F$
In general, $F^+$'s size is exponential to $F$'s size.

Example: $R = \{AB, BC\}$

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

Some FD's from $F^+$:

$A \rightarrow BC$

$(A \rightarrow B, B \rightarrow C) \Rightarrow A \rightarrow C \Rightarrow A \rightarrow BC$

$A \rightarrow B$

$AB \rightarrow BC$

$ABC \Rightarrow AB$ (trivial)

$C \rightarrow C$ (trivial)

... (many more)
Example

\[ R = \{ A, B, C \} \]
\[ F = \{ AB \rightarrow C, C \rightarrow B \} \]

Prove that: \( AC \rightarrow B \)

Proof: from \( C \rightarrow B \) & augmentation rule
we get: \( AC \rightarrow AB \) (1)

from \( B \in \{ B, A \} \) we get: \( AB \rightarrow B \) (2)

(1), (2) & transitivity rule
we get: \( AC \rightarrow B \)
Reasoning About FDs (Contd.)

- Couple of additional rules (that follow from AA):
  - *Union*: If $X \rightarrow Y$ and $X \rightarrow Z$, then $X \rightarrow YZ$
  - *Decomposition*: If $X \rightarrow YZ$, then $X \rightarrow Y$ and $X \rightarrow Z$

- Example: Contracts($cid, sid, jid, did, pid, qty, value$), and:
  - C is the key: $C \rightarrow CSJDPQV$
  - Project purchases each part using single contract: $JP \rightarrow C$
  - Dept purchases at most one part from a supplier: $SD \rightarrow P$

- $JP \rightarrow C, C \rightarrow CSJDPQV$ imply $JP \rightarrow CSJDPQV$
- $SD \rightarrow P$ implies $SDJ \rightarrow JP$
- $SDJ \rightarrow JP, JP \rightarrow CSJDPQV$ imply $SDJ \rightarrow CSJDPQV$
Why do we need to compute the closure of an attribute $X^+$?

Two reasons:

1. Indirect but systematic way to compute $F^+$ (compute $X^+$ for all subsets $X$)

2. Verify if $X$ is a superkey (if $X^+ = R$)

Note that computing $X^+$ involves only transitivity (neither reflexivity nor augmentation can add any attributes to $X^+$)
Figure 15.6 Computing the Attribute Closure of Attribute Set $X$

**Assume** $F = \{ A \rightarrow B, B \rightarrow C, CD \rightarrow E \}$ i.e. the relation contains $ABCD$ attributes

**Question:** $A \rightarrow E$?

We can answer it by first computing $A^+$ using the above algorithm.

- **Initially:** $\text{closure} = \{A\}$
  - using $A \rightarrow B$ we set $\text{closure} = \{A, B\}$
  - using $B \rightarrow C$ "" closure $\{A, B, C\}$
  - $CD \rightarrow E$ does not increase closure set

Hence $A^+ = \{A, B, C\}$ Since $E \notin A^+$ then $A \not\rightarrow E$

**Important:** Can use the above alg. to find whether a set of attributes $X$ is a key for a relation.

**Why?**

Ex. - is $A$ a key for the relation $R = \{ABCDE\}$? **N**
- is $AD$ a key for "" "" "" ? **Y** Why?
- is $AC$ "" "" "" ? **N**
Example

Let $R = \{A, B, C, D, E\}$ with the following set of dependencies.

\[
\begin{align*}
C & \rightarrow A \quad (1) \\
AB & \rightarrow C \quad (2) \\
BC & \rightarrow D \quad (3) \\
A & \rightarrow BE \quad (4) \\
B & \rightarrow E \quad (5)
\end{align*}
\]

Compute the closures of all attributes of $R$.

\[
\begin{align*}
A^+ &= A, \quad A^+ = 4ABE, \quad A^+ = 2ABCE, \quad A^+ = 3ABCDE \\
B^+ &= B, \quad B^+ = 5BE \\
C^+ &= C, \quad C^+ = 1AC, \quad C^+ = 4ABCE, \quad C^+ = 3ABCDE \\
D^+ &= D \\
E^+ &= E
\end{align*}
\]

and

\((A \text{ and any other combination of attributes})^+ = ABCDE\)

\((C \text{ and any other combination of attributes})^+ = ABCDE\)

\((BD)^+ = BDE\)

\text{etc.}\)
Reasoning About FDs (Contd.)

- Computing the closure of a set of FDs can be expensive. (Size of closure is exponential in # attrs!)
- Typically, we just want to check if a given FD $X \rightarrow Y$ is in the closure of a set of FDs $F$. An efficient check:
  - Compute attribute closure of $X$ (denoted $X^+$) wrt $F$:
    - Set of all attributes $A$ such that $X \rightarrow A$ is in $F^+$
    - There is a linear time algorithm to compute this.
  - Check if $Y$ is in $X^+$
- Does $F = \{ A \rightarrow B, \ B \rightarrow C, \ C \ D \rightarrow E \}$ imply $A \rightarrow E$?
  - i.e., is $A \rightarrow E$ in the closure $F^+$? Equivalently, is $E$ in $A^+$?
Decomposition of a Relation Scheme

- Suppose that relation R contains attributes A1 ... An. A decomposition of R consists of replacing R by two or more relations such that:
  - Each new relation scheme contains a subset of the attributes of R (and no attributes that do not appear in R), and
  - Every attribute of R appears as an attribute of one of the new relations.
- Intuitively, decomposing R means we will store instances of the relation schemes produced by the decomposition, instead of instances of R.
- E.g., Can decompose SNLRWH into SNLRH and RW.
Example Decomposition

- Decompositions should be used only when needed.
  - SNLRWH has FDs S → SNLRWH and R → W
  - Second FD causes violation of 3NF; W values repeatedly associated with R values. Easiest way to fix this is to create a relation RW to store these associations, and to remove W from the main schema:
    - i.e., we decompose SNLRWH into SNLRH and RW

- The information to be stored consists of SNLRWH tuples. If we just store the projections of these tuples onto SNLRH and RW, are there any potential problems that we should be aware of?
**Problems with Decompositions**

- There are three potential problems to consider:
  - Some queries become more expensive.
    - e.g., How much did sailor Joe earn? (salary = W*H)
  - Given instances of the decomposed relations, we may not be able to reconstruct the corresponding instance of the original relation!
    - Fortunately, not in the SNLRWH example.
  - Checking some dependencies may require joining the instances of the decomposed relations.
    - Fortunately, not in the SNLRWH example.

- *Tradeoff*: Must consider these issues vs. redundancy.
Lossless Join Decompositions

- Decomposition of R into X and Y is **lossless-join** w.r.t. a set of FDs F if, for every instance r that satisfies F:
  - \( \pi_X(r) \bowtie \pi_Y(r) = r \)

- It is always true that \( r \subseteq \pi_X(r) \bowtie \pi_Y(r) \)
  - In general, the other direction does not hold! If it does, the decomposition is lossless-join.

- Definition extended to decomposition into 3 or more relations in a straightforward way.

- It is essential that all decompositions used to deal with redundancy be lossless! (Avoids Problem (2).)
More on Lossless Join

- The decomposition of R into X and Y is lossless-join wrt F if and only if the closure of F contains:
  - $X \cap Y \rightarrow X$, or
  - $X \cap Y \rightarrow Y$

- In particular, the decomposition of R into UV and R - V is lossless-join if $U \rightarrow V$ holds over R.
3.1. Testing for Lossless-Join Decomposition

Let

\[ D = \{ R_1, R_2 \} \] a decomposition of relational schema \( R \)

\( F \) a set of functional dependencies of \( R \).

\( \square \) \( D \) is a lossless-join decomposition if at least one of the following functional dependencies are in \( F^+ \).

\[ R_1 \cap R_2 \rightarrow R_1 \]

\[ R_1 \cap R_2 \rightarrow R_2 \]

**Example:** Suppose

\[ R = \{ A, B, C \} \]

\[ F = \{ A \rightarrow B \} \]

\[ D = \{ R_1, R_2 \} \], where

\[ R_1 = \{ A, B \} \] and \( R_2 = \{ A, C \} \)

Test if \( D \) is lossless join decomposition.

- Notice that \( A \rightarrow B \) is the only non trivial dependency in \( F^+ \).
- \( R_1 \cap R_2 = \{ A \} \)
- we have to show that at least one of the dependencies \( A \rightarrow R_1 \) and \( A \rightarrow R_2 \) are in \( F^+ \).
- Since \( A \rightarrow B \), then \( A \rightarrow AB \) is in \( F^+ \).
Example

Assume

\[ R = \{ \text{numstr}, \text{city}, \text{zip} \} \]
\[ F = \{ \{\text{numstr, city}\} \rightarrow \text{zip}, \text{zip} \rightarrow \text{city} \} \]
\[ D = \{ R_1, R_2 \}, \text{ where} \]
\[ R_1 = \{ \text{numstr, zip} \} \text{ and } R_2 = \{ \text{city, zip} \}. \]

Test if \( D \) is lossless join.

- \( R_1 \cap R_2 = \{ \text{zip} \} \)
- We have to test whether \( \text{zip} \rightarrow \{ \text{numstr, zip} \} \) or \( \text{zip} \rightarrow \{ \text{city, zip} \} \).
- Removing the trivial dependencies, I should check whether \( \text{zip} \rightarrow \text{numstr} \) or \( \text{zip} \rightarrow \text{city} \).
- The functional dependency \( \text{zip} \rightarrow \text{city} \) is already in \( F \) and therefore we do not even have to find the closure of \( F \).
- Thus, \( D \) is lossless join.
Example

Suppose

\[ R = \{ A, B, C \} \]
\[ F = \{ A \rightarrow B \} \]
\[ D = \{ R_1, R_2 \}, \text{ where} \]
\[ R_1 = \{ A, B \} \text{ and } R_2 = \{ B, C \} \]

Test if \( D \) is lossless join decomposition.

- \( R_1 \cap R_2 = \{ B \} \)

- We have to test whether at least one of the dependencies \( B \rightarrow R_1 \) and \( B \rightarrow R_2 \) are in \( F^+ \).

- None is in \( F^+ \) and therefore this decomposition is not lossless-join.
General case for decomposition into many relations.

Algorithm for testing lossless join:

Build matrix, with attributes of all subrelations as columns and subrelations as rows.

1. If attribute \( a \) is in subrelation \( R \), place symbol ‘o’ in appropriate column \( a \), row \( R \).

2. For (each FD \( X \rightarrow Y \)) do
   - For (every pair of rows having ‘o’ in \( X \) columns) do
     - If (either row has ‘o’ in some \( Y \) column)
       - Then place ‘o’ in \( Y \) column of both rows

When finished, if some row is all ‘o’s, then join is lossless; else it’s lossy.

Example:

\[
R_1 = \{ \text{SSN}, \text{Name} \} \\
R_2 = \{ \text{PNumber}, \text{Name}, \text{PLocation} \} \\
R_3 = \{ \text{SSN}, \text{PNumber}, \text{Hours} \}
\]

\[
F = \{ \text{SSN} \rightarrow \text{Name}, \ \text{PNumber} \rightarrow \text{Name, PLocation}, \ \text{SSN, PNumber} \rightarrow \text{Hours} \}
\]

<table>
<thead>
<tr>
<th></th>
<th>SSN</th>
<th>Name</th>
<th>PNumber</th>
<th>PName</th>
<th>PLocation</th>
<th>Hours</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>0</td>
<td></td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R2</td>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>R3</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Step 1 adds more "o"

This algorithm runs in polynomial time in size of \( F \)
3. Desirable Properties of Decomposition

1. Lossless-join Decomposition

☐ A decomposition of \( r(R) \) into \( \{R_1, R_2, \cdots, R_n\} \) is a lossless decomposition if:

\[
r(R) = r_1(R_1) \Join r_2(R_2) \Join \cdots \Join r_n(R_n)
\]

2. Dependency Preserving Decomposition

Let \( D = \{R_1, R_2, \cdots, R_n\} \) be a decomposition of relational schema \( R \), and \( F \) a set of functional dependencies of \( R \).

or restriction

☐ The projection of \( F \) onto \( R_i \), written \( F_i \), is the set of all functional dependencies in \( F^+ \) that include attributes only from \( R_i \).

☐ A decomposition \( D \) preserves a set of dependencies \( F \) if the union of all \( F_i \) logically implies \( F \), i.e.

\[
\bigcup_{\text{all } i} F_i \equiv F
\]

\[\equiv \text{equivalent}\]

or

\[
\left[ \bigcup_{\text{all } i} F_i \right]^+ = F^+
\]
Why do we need the dependency preservation property?

In a design, FDs (constraints from real world) are represented into relations.

To prevent update anomalies whenever an update is made, we want to check quickly whether the update violates any FD.

Let F be the set of FDs.

To check F against an update it's easy & fast to check only one relation & NOT join relations.

If \( \left( \bigcup_{i \in I} F_i \right)^+ = F^+ \)

we know that to check F we only need to check the F_i's, i.e. only individual relations.
Dependency Preserving Decomposition

- Consider CSJDPQV, C is key, JP → C and SD → P.
  - BCNF decomposition: CSJDQV and SDP
  - Problem: Checking JP → C requires a join!

- Dependency preserving decomposition (Intuitive):
  - If R is decomposed into X, Y and Z, and we enforce the FDs that hold on X, on Y and on Z, then all FDs that were given to hold on R must also hold. (Avoids Problem (3).)

- **Projection of set of FDs F**: If R is decomposed into X, ... projection of F onto X (denoted F_X) is the set of FDs U → V in F^+ (closure of F) such that U, V are in X.
Dependency Preserving Decompositions
(Contd.)

- Decomposition of R into X and Y is dependency preserving if \((F_X \cup F_Y)^+ = F^+\)
  - i.e., if we consider only dependencies in the closure \(F^+\) that can be checked in X without considering Y, and in Y without considering X, these imply all dependencies in \(F^+\).

- Important to consider \(F^+\), not \(F\), in this definition:
  - ABC, A → B, B → C, C → A, decomposed into AB and BC.
  - Is this dependency preserving? Is C → A preserved?

- Dependency preserving does not imply lossless join:
  - ABC, A → B, decomposed into AB and BC.

- And vice-versa! (Example?)
3.2. Testing for Dependency Preservation

Algorithm

\[ G = \{ \} ; \]
for each \( R_i \) in decomposition \( D \) do
\[ G = G \cup F_i ; \]
If \( G \equiv F \) then
\[ \text{return true} \]
else
\[ \text{return false;} \]

Example: Suppose

\[ R = \{ A, B, C \} \]
\[ F = \{ A \rightarrow B, B \rightarrow C, C \rightarrow A \} \]
\[ D = \{ R_1, R_2 \}, \text{ where} \]
\[ R_1 = \{ A, B \} \text{ and } R_2 = \{ B, C \} \]

Under this scheme

- \( F^+ = \{ A \rightarrow B, B \rightarrow C, C \rightarrow A, C \rightarrow B, A \rightarrow C, \ldots \} = \{ A \rightarrow R, B \rightarrow R, C \rightarrow R, \ldots \} \)
- \( F_{12} = \{ A \rightarrow B, B \rightarrow A \} \) and \( F_{22} = \{ B \rightarrow C, C \rightarrow B \} \)
- Clearly, \( (F_{12} \cup F_{22})^+ = F^+ \).

\[ \text{Yes because} \]

\[ F_{12} \text{ includes } B \rightarrow A \]
\[ F_{22} \text{ includes } C \rightarrow B \]
\[ \Rightarrow C \rightarrow A \]
\[ (F_{12} \cup F_{22})^+ \]
Example

\[ R = \{ \text{numstr}, \text{city}, \text{zip} \} \]

\[ F = \{ \{ \text{numstr}, \text{city} \} \rightarrow \text{zip}, \ \text{zip} \rightarrow \text{city} \} \]

BCNF: \( R_1 = \{ \text{numstr}, \text{zip} \} \) and \( R_2 = \{ \text{city}, \text{zip} \} \).

Test if this BCNF decomposition preserves dependencies

- The non-trivial dependencies in \( F^+ \) are:
  \[
  \begin{align*}
  \text{numstr}, \text{city} & \rightarrow \text{zip} \\
  \text{numstr}, \text{zip} & \rightarrow \text{city} \\
  \text{zip} & \rightarrow \text{city}
  \end{align*}
  \]

- \( F_1 \) contains trivial dependencies

- \( F_2 \) includes the non-trivial dependency \( \text{zip} \rightarrow \text{city} \).

- Thus the non-trivial dependencies of \( F_1 \cup F_2 \) are \( \text{zip} \rightarrow \text{city} \).

- It is clear that we can not deduce \( \{ \text{numstr}, \text{city} \} \rightarrow \text{zip} \) from \( \{ \text{zip} \rightarrow \text{city} \} \) and therefore \( F \) is not equivalent to \( F_1 \cup F_2 \).

\[
\begin{array}{ccc}
\text{numstr} & \text{zip} \\
2 \text{ Mass} & 02215 \\
2 \text{ Mass} & 02200
\end{array}
\]

\[
\begin{array}{ccc}
\text{city} & \text{zip} \\
\text{Boston} & 02215 \\
\text{Boston} & 02200
\end{array}
\]

\[
\begin{array}{ccc}
\text{numstr} & \text{city} & \text{zip} \\
2 \text{ Mass} & \text{Boston} & '02215 \\
2 \text{ Mass} & \text{Boston} & 02200
\end{array}
\]
Normal Forms

- Returning to the issue of schema refinement, the first question to ask is whether any refinement is needed!
- If a relation is in a certain normal form (BCNF, 3NF etc.), it is known that certain kinds of problems are avoided/minimized. This can be used to help us decide whether decomposing the relation will help.
- Role of FDs in detecting redundancy:
  - Consider a relation R with 3 attributes, ABC.
    - No FDs hold: There is no redundancy here.
    - Given \( A \rightarrow B \): Several tuples could have the same A value, and if so, they'll all have the same B value!
Boyce-Codd Normal Form (BCNF)

- Reln R with FDs F is in BCNF if, for all $X \rightarrow A$ in $F^+$
  - $A \in X$ (called a trivial FD), or
  - $X$ contains a key for R.

- In other words, R is in BCNF if the only non-trivial FDs that hold over R are key constraints.
  - No dependency in R that can be predicted using FDs alone.
  - If we are shown two tuples that agree upon the X value, we cannot infer the A value in one tuple from the A value in the other.
  - If example relation is in BCNF, the 2 tuples must be identical (since X is a key).

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>y1</td>
<td>a</td>
</tr>
<tr>
<td>x</td>
<td>y2</td>
<td>?</td>
</tr>
</tbody>
</table>
Third Normal Form (3NF)

- Reln R with FDs F is in 3NF if, for all $X \rightarrow A$ in $F^+$
  - $A \in X$ (called a trivial FD), or
  - $X$ contains a key for R, or
  - $A$ is part of some key for R.

- Minimality of a key is crucial in third condition above!

- If R is in BCNF, obviously in 3NF.

- If R is in 3NF, some redundancy is possible. It is a compromise, used when BCNF not achievable (e.g., no "good" decomp, or performance considerations).
  - Lossless-join, dependency-preserving decomposition of R into a collection of 3NF relations always possible.
BCNF and Dependency Preservation

- In general, there may not be a dependency preserving decomposition into BCNF.
  - e.g., CSZ, CS → Z, Z → C
  - Can't decompose while preserving 1st FD; not in BCNF.
- Similarly, decomposition of CSJDQV into SDP, JS and CJDQV is not dependency preserving (w.r.t. the FDs JP → C, SD → P and J → S).
  - However, it is a lossless join decomposition.
  - In this case, adding JPC to the collection of relations gives us a dependency preserving decomposition.
    - JPC tuples stored only for checking FD! (Redundancy!)
BCNF $\Rightarrow$ 3NF $\Rightarrow$ 2NF $\Rightarrow$ 1NF

There exists a decomposition algorithm that can always give:

BCNF + lossless join + No redundancy

But MAY NOT have depend. preservation

There exists an algorithm that produces

3NF, lossless join + depend. present

But MAY allow redundancy

In practice: Try to get BCNF if the decomp. does NOT preserve dependencies, then USE 3NF
Algorithm for BCNF decomposition:

compute $F^+$
subrelations := $R$
while (relation $S$ in subrelations isn't BCNF) do begin
    choose nontrivial FD `$X \rightarrow a$' at $X$ not superkey
    replace $S$ with `$Xa$' and `$S-a$'
end

Note: each step preserves lossless join

Why?

$R_1 = Xa$

$R_2 = S-a$

then $R_1 \cap R_2 \rightarrow R_4$ is in $F^+$

($X \rightarrow Xa$)

Example: $R = (\text{Branch, Assets, Bcity, loan#, Customer, Amount})$

$F = \{ \text{Branch} \rightarrow \text{assets, Branch} \rightarrow \text{Bcity, loan#} \rightarrow \text{amount, loan#} \rightarrow \text{Branch} \}$

Key: $(\text{loan#, Customer})$
\( R \) is not in BCNF

Branch \( \rightarrow \) assets holds AND not key

\[ \Rightarrow R_1 = (\text{Branch, assets}) \checkmark \]

\[ R_2 = (\text{Branch, B city, loan #, cust, Amount}) \]

\( R_2 \) not in BCNF

Branch \( \rightarrow \) B city

\[ \Rightarrow R_3 = (\text{Branch, B city}) \checkmark \]

\[ R_4 = (\text{Branch, loan #, cust, amount}) \]

\( R_4 \) Not in BCNF

\[ \Rightarrow R_5 = (\text{loan #, amount}) \checkmark \]

\[ R_6 = (\text{branch, loan #, customer}) \]

\( R_6 \) not BCNF

\[ \Rightarrow R_7 = (\text{loan #, branch}) \checkmark \]

\[ R_8 = (\text{loan #, customer}) \checkmark \]
Decomposition into 3NF

- Obviously, the algorithm for lossless join decom into BCNF can be used to obtain a lossless join decom into 3NF (typically, can stop earlier).

- To ensure dependency preservation, one idea:
  - If \(X \rightarrow Y\) is not preserved, add relation \(XY\).
  - Problem is that \(XY\) may violate 3NF! e.g., consider the addition of \(CJP\) to `preserve` \(JP \rightarrow C\). What if we also have \(J \rightarrow C\)?

- Refinement: Instead of the given set of FDs \(F\), use a minimal cover for \(F\).
Minimal Cover for a Set of FDs

- **Minimal cover** G for a set of FDs F:
  - Closure of F = closure of G.
  - Right hand side of each FD in G is a single attribute.
  - If we modify G by deleting an FD or by deleting attributes from an FD in G, the closure changes.

- Intuitively, every FD in G is needed, and "as small as possible" in order to get the same closure as F.

- e.g., \( A \rightarrow B, \ ABCD \rightarrow E, \ EF \rightarrow GH, \ ACDF \rightarrow EG \) has the following minimal cover:
  - \( A \rightarrow B, \ ACD \rightarrow E, \ EF \rightarrow G \) and \( EF \rightarrow H \)

- M.C. \( \rightarrow \) Lossless-Join, Dep. Pres. Decomp!!! (in book)

Database Management Systems, 3ed, R. Ramakrishnan and J. Gehrke
Given a relation $R$ with a set of FDs $F$, let $F_c$ denote a minimal (or canonical) cover of $F$.

3NF Synthesis Algorithm:

- First, find a $F_c$.
- Then:

```plaintext
subrelations := {}  
/
* break into tiny subrelations */
for (each FD $X \rightarrow a$ in $F_c$) do
    subrelations := ``Xa`
/
* combine tiny subrelations sharing same key */
while (combinations still possible) do
    if Xa and Xb are subrelations and $X \rightarrow a$ and $X \rightarrow b$,
    then subrelations := Xab - Xa - Xb
end
/
* if necessary, make one relation containing a key */
* for the original relation */
if (no subrelation contains key K for original relation)
    subrelations := K
```

3NF avoids transitive FD's which may also cause update anomalies.
Ex 1

Let \( R : ABC \)

\( F = \{ A \rightarrow B, C \rightarrow B \} \)

Clearly \( F \) is in \( F_c \) form

\( A \rightarrow B \) implies subrelation \( AB \)

\( C \rightarrow B \) implies \( CB \)

But this is not a lossless-join decomp.

(Why?)

Need to add a subrelation with the key.

(from \( F \) you can see that \( AC \rightarrow ABC \))

Hence add: subrelation \( AC \)

Thus \( ABC \) is decomposed to

\( AB, CB, AC \), which are 3NF + depend. preserving.

(This is also lossless join. Why?)
Ex. 2  Consider \( R = \text{CSIDPOQV} \)

\[ F = \{ c \rightarrow \text{CSIDPOQV}, \; \text{JP} \rightarrow c, \; \text{SD} \rightarrow p, \; J \rightarrow s \} \]

First find a \( F_c \).

\( c \rightarrow \text{CSIDPOQV} \) is replaced by

\( c \rightarrow s, \; c \rightarrow j, \; c \rightarrow d, \; c \rightarrow p, \; c \rightarrow q, \; c \rightarrow v \)

Note that: \( c \rightarrow p \) can be inferred by

\( c \rightarrow s, \; c \rightarrow d, \; \text{SD} \rightarrow p \)

Hence \( c \rightarrow p \) not in \( F_c \).

Similarly \( c \rightarrow s \) is implied by \( c \rightarrow j, \; j \rightarrow s \)

Hence \( F_c = \{ c \rightarrow j, \; c \rightarrow d, \; c \rightarrow q, \; c \rightarrow v, \; \text{JP} \rightarrow c, \; \text{SD} \rightarrow p, \; J \rightarrow s \} \)

3NF synthesis alg:
creates \( \text{CJ, CD, CQ, CV, JP, SD, JS} \)

replace by \( \text{CIDPOQV} \)

The key \( c \) is contained \( \Rightarrow \) 3NF, lossless join
& depend. pres.
Refining an ER Diagram

- 1st diagram translated:
  Workers(S,N,L,D,S)
  Departments(D,M,B)
  - Lots associated with workers.

- Suppose all workers in a dept are assigned the same lot: D → L

- Redundancy; fixed by:
  Workers2(S,N,D,S)
  Dept_Lots(D,L)

- Can fine-tune this:
  Workers2(S,N,D,S)
  Departments(D,M,B,L)