Relational Normalization Theory
Chapter 8

Limitations of E-R Designs

- Provides a set of guidelines, does not result in a unique database schema
- Does not provide a way of evaluating alternative schemas
- Normalization theory provides a mechanism for analyzing and refining the schema produced by an E-R design

Redundancy

- Dependencies between attributes cause redundancy
  - Ex. All addresses in the same town have the same zip code

<table>
<thead>
<tr>
<th>SSN</th>
<th>Name</th>
<th>Town</th>
<th>Zip</th>
</tr>
</thead>
<tbody>
<tr>
<td>1234</td>
<td>Joe</td>
<td>Stony Brook</td>
<td>11790</td>
</tr>
<tr>
<td>4321</td>
<td>Mary</td>
<td>Stony Brook</td>
<td>11790</td>
</tr>
<tr>
<td>5454</td>
<td>Tom</td>
<td>Stony Brook</td>
<td>11790</td>
</tr>
</tbody>
</table>

Redundancy and Other Problems

- Set valued attributes in the E-R diagram result in multiple rows in corresponding table
- Example: Person (SSN, Name, Address, Hobbies)
  - A person entity with multiple hobbies yields multiple rows in table Person
    - Hence, the association between Name and Address for the same person is stored redundantly
  - SSN is key of entity set, but (SSN, Hobby) is key of corresponding relation
    - The relation Person can’t describe people without hobbies

Example

<table>
<thead>
<tr>
<th>SSN</th>
<th>Name</th>
<th>Address</th>
<th>Hobby</th>
</tr>
</thead>
<tbody>
<tr>
<td>1111</td>
<td>Joe</td>
<td>123 Main</td>
<td>biking, hiking</td>
</tr>
</tbody>
</table>

ER Model

<table>
<thead>
<tr>
<th>SSN</th>
<th>Name</th>
<th>Address</th>
<th>Hobby</th>
</tr>
</thead>
<tbody>
<tr>
<td>1111</td>
<td>Joe</td>
<td>123 Main</td>
<td>biking</td>
</tr>
</tbody>
</table>

Relational Model

<table>
<thead>
<tr>
<th>SSN</th>
<th>Name</th>
<th>Address</th>
<th>Hobby</th>
</tr>
</thead>
<tbody>
<tr>
<td>1111</td>
<td>Joe</td>
<td>123 Main</td>
<td>biking</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SSN</th>
<th>Name</th>
<th>Address</th>
<th>Hobby</th>
</tr>
</thead>
<tbody>
<tr>
<td>1111</td>
<td>Joe</td>
<td>123 Main</td>
<td>hiking</td>
</tr>
</tbody>
</table>

Anomalies

- Redundancy leads to anomalies:
  - Update anomaly: A change in Address must be made in several places
  - Deletion anomaly: Suppose a person gives up all hobbies. Do we:
    - Set Hobby attribute to null? No, since Hobby is part of key
    - Delete the entire row? No, since we lose other information in the row
  - Insertion anomaly: Hobby value must be supplied for any inserted row since Hobby is part of key
Decomposition

- **Solution:** use two relations to store Person information
  - Person1 (SSN, Name, Address)
  - Hobbies (SSN, Hobby)
- The decomposition is more general: people with hobbies can now be described
- No update anomalies:
  - Name and address stored once
  - A hobby can be separately supplied or deleted

Normalization Theory

- Result of E-R analysis need further refinement
- Appropriate decomposition can solve problems
- The underlying theory is referred to as normalization theory (and other kinds, like multivalued dependencies)

Functional Dependencies

- **Definition:** A functional dependency (FD) on a relation schema \( R \) is a constraint \( X \rightarrow Y \), where \( X \) and \( Y \) are subsets of attributes of \( R \).
- **Definition:** An FD \( X \rightarrow Y \) is satisfied in an instance \( r \) of \( R \) if for every pair of tuples, \( t \) and \( s \): if \( t \) and \( s \) agree on all attributes in \( X \) then they must agree on all attributes in \( Y \)
  - Key constraint is a special kind of functional dependency: all attributes of relation occur on the right-hand side of the FD:
    - \( SSN \rightarrow SSN, Name, Address \)

Entailment, Closure, Equivalence

- **Definition:** If \( F \) is a set of FDs on schema \( R \) and \( f \) is another FD on \( R \), then \( F \) entails \( f \) if every instance \( r \) of \( R \) that satisfies every FD in \( F \) also satisfies \( f \)
  - Ex: \( F = \{A \rightarrow B, B \rightarrow C\} \) and \( f \) is \( A \rightarrow C \)
  - If \( StreetAdr \rightarrow Town \rightarrow Zip \) then \( StreetAdr \rightarrow Zip \)
- **Definition:** The closure of \( F \), denoted \( F^+ \), is the set of all FDs entailed by \( F \)
- **Definition:** \( F \) and \( G \) are equivalent if \( F \) entails \( G \) and \( G \) entails \( F \)
Entailment (cont’d)

- Satisfaction, entailment, and equivalence are *semantic* concepts – defined in terms of the actual relations in the “real world.”
  - They define what *these notions are*, not how to compute them
- How to check if $F$ entails $f$ or if $F$ and $G$ are equivalent?
  - *Bad idea:* might be infinite number for infinite domains
  - *Solution:* find algorithmic, *syntactic* ways to compute these notions
    - Important: The syntactic solution must be “correct” with respect to the *semantic* definitions
    - Correctness has two aspects: *soundness* and *completeness* – see later

Armstrong’s Axioms for FDs

- This is the *syntactic* way of computing/testing the various properties of FDs

  - ** Reflexivity:** If $Y \subseteq X$ then $X \rightarrow Y$ (trivial FD)
    - Name, Address $\rightarrow$ Name
  - **Augmentation:** If $X \rightarrow Y$ then $XZ \rightarrow YZ$
    - If $Town \rightarrow Zip$ then $Town, Name \rightarrow Zip, Name$
  - **Transitivity:** If $X \rightarrow Y$ and $Y \rightarrow Z$ then $X \rightarrow Z$

Soundness

- Axioms are *sound*: If an FD $f: X \rightarrow Y$ can be derived from a set of FDs $F$ using the axioms, then $f$ holds in every relation that satisfies every FD in $F$.
- Example: Given $X \rightarrow Y$ and $X \rightarrow Z$ then

  - $X \rightarrow XY$ Augmentation by $X$
  - $YX \rightarrow YZ$ Augmentation by $Y$
  - $X \rightarrow YZ$ Transitivity
  - Thus, $X \rightarrow YZ$ is satisfied in every relation where both $X \rightarrow Y$ and $X \rightarrow Y$ are satisfied
  - Therefore, we have derived the union rule for FDs: we can take the union of the RHSs of FDs that have the same LHS

Correctness

- The notions of *soundness* and *completeness* link the syntax (Armstrong’s axioms) with semantics (the definitions in terms of relational instances)
- This is a precise way of saying that the algorithm for entailment based on the axioms is “correct” with respect to the definitions

Completeness

- Axioms are *complete*: If $F$ entails $f$, then $f$ can be derived from $F$ using the axioms
- A consequence of completeness is the following (naïve) algorithm to determining if $F$ entails $f$:
  
  - Algorithm: Use the axioms in all possible ways to generate $F^+$ (the set of possible FD’s is finite so this can be done) and see if $f$ is in $F^+$

Generating $F^+$

Thus, $AB \rightarrow BD$, $AB \rightarrow BCD$, $AB \rightarrow BCDE$, and $AB \rightarrow CDE$ are all elements of $F^+$
Attribute Closure

- Calculating attribute closure leads to a more efficient way of checking entailment.
- The attribute closure of a set of attributes, $X$, with respect to a set of functional dependencies, $F$, (denoted $X^+_F$) is the set of all attributes, $A$, such that $X \rightarrow A$
  - $X^+_F$ is not necessarily the same as $X^+_F$ if $F_1 \neq F_2$
- Attribute closure and entailment:
  - Algorithm: Given a set of FDs, $F$, then $X \rightarrow Y$ if and only if $X^+_F \supseteq Y$
  - Example - Computing Attribute Closure

Example - Computing Attribute Closure

<table>
<thead>
<tr>
<th>$X$</th>
<th>$X^*_F$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$AB \rightarrow C$</td>
<td>$A$</td>
</tr>
<tr>
<td>$A \rightarrow D$</td>
<td>$AB$</td>
</tr>
<tr>
<td>$D \rightarrow E$</td>
<td>(Hence $AB$ is a key)</td>
</tr>
<tr>
<td>$AC \rightarrow B$</td>
<td>$B$</td>
</tr>
<tr>
<td>$D \rightarrow E$</td>
<td>${D, E}$</td>
</tr>
</tbody>
</table>

Is $AB \rightarrow E$ entailed by $F$? Yes
Is $D \rightarrow C$ entailed by $F$? No
Result: $X^*_F$ allows us to determine FDs of the form $X \rightarrow Y$ entailed by $F$

Computation of Attribute Closure $X^+_F$

```
closure := X;  // since X <= X^+_F
repeat
  old := closure;
  if there is an FD $Z \rightarrow V$ in $F$ such that $Z \subseteq closure$ and $V \subseteq closure$
  then closure := closure $\cup$ $V$
  until old = closure
  if $T \subseteq closure$ then $X \rightarrow T$ is entailed by $F$
```

Example: Computation of Attribute Closure

Problem: Compute the attribute closure of $AB$ with respect to the set of FDs:
- $AB \rightarrow C$ (a)
- $A \rightarrow D$ (b)
- $D \rightarrow E$ (c)
- $AC \rightarrow B$ (d)

Solution:
Initially closure = $\{AB\}$
Using (a) closure = $\{ABC\}$
Using (b) closure = $\{ABCD\}$
Using (c) closure = $\{ABCDE\}$

Normal Forms

- Each normal form is a set of conditions on a schema that guarantees certain properties (relating to redundancy and update anomalies)
- First normal form (1NF) is the same as the definition of relational model (relations = sets of tuples; each tuple = sequence of atomic values)
- Second normal form (2NF) – a research lab accident; has no practical or theoretical value – won’t discuss
- The two commonly used normal forms are third normal form (3NF) and Boyce-Codd normal form (BCNF)

BCNF

- Definition: A relation schema $R$ is in BCNF if for every FD $X \rightarrow Y$ associated with $R$ either
  - $Y \subseteq X$ (i.e., the FD is trivial) or
  - $X$ is a superkey of $R$
- Example: Person1(SSN, Name, Address)
  - The only FD is $SSN \rightarrow Name, Address$
  - Since SSN is a key, Person1 is in BCNF
(non) BCNF Examples

- **Person (SSN, Name, Address, Hobby)**
  - The FD $SSN \rightarrow Name, Address$ does not satisfy requirements of BCNF
    - since the key is $(SSN, Hobby)$
- **HasAccount (AccountNumber, ClientId, OfficeId)**
  - The FD $AccountNumber \rightarrow OfficeId$ does not satisfy BCNF requirements
    - since keys are $(ClientId, OfficeId)$ and $(AccountNumber, ClientId)$

Redundancy

- Suppose $R$ has a FD $A \rightarrow B$. If an instance has 2 rows with same value in $A$, they must also have same value in $B$ (⇒ redundancy, if the $A$-value repeats twice)

<table>
<thead>
<tr>
<th>SSN</th>
<th>Name</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>1111</td>
<td>Joe</td>
<td>123 Main</td>
</tr>
<tr>
<td>1111</td>
<td>Joe</td>
<td>123 Main</td>
</tr>
</tbody>
</table>

- If $A$ is a superkey, there cannot be two rows with same value of $A$
  - Hence, BCNF eliminates redundancy

Third Normal Form

- A relational schema $R$ is in 3NF if for every FD $X \rightarrow Y$ associated with $R$ either:
  - $Y \subseteq X$ (i.e., the FD is trivial); or
  - $X$ is a superkey of $R$; or
  - Every $A \in Y$ is part of some key of $R$
- 3NF is weaker than BCNF (every schema that is in BCNF is also in 3NF)

3NF Example

- **HasAccount (AccountNumber, ClientId, OfficeId)**
  - $ClientId, OfficeId \rightarrow AccountNumber$
    - OK since LHS contains a key
  - $AccountNumber \rightarrow OfficeId$
    - OK since RHS is part of a key
  - $HasAccount$ is in 3NF but it might still contain redundant information due to $AccountNumber \rightarrow OfficeId$
    - (which is not allowed by BCNF)

3NF (Non) Example

- **Person (SSN, Name, Address, Hobby)**
  - $(SSN, Hobby)$ is the only key.
  - $SSN \rightarrow Name$ violates 3NF conditions since $Name$ is not part of a key and $SSN$ is not a superkey
Decompositions

- **Goal**: Eliminate redundancy by decomposing a relation into several relations in a higher normal form.
- Decomposition must be *lossless*: it must be possible to reconstruct the original relation from the relations in the decomposition.
  - We will see why.

---

Example Decomposition

Schema (R, F) where
R = \{SSN, Name, Address, Hobby\}
F = \{SSN → Name, Address\}
can be decomposed into
\[ R_1 = \{\text{SSN, Name, Address}\} \]
and
\[ R_2 = \{\text{SSN, Hobby}\} \]
\[ F_1 = \{\text{SSN} → \text{Name, Address}\} \]
\[ F_2 = \{\} \]

---

Lossless Schema Decomposition

- A decomposition should not lose information.
- A decomposition \((R_1, \ldots, R_n)\) of a schema, \(R\), is *lossless* if every valid instance, \(r\), of \(R\) can be reconstructed from its components:
  \[ r = r_1 \Join r_2 \Join \ldots \Join r_n \]
- where each \(r_i = \pi_{R_i}(r)\)

---

Lossy Decomposition

The following is always the case (Think why?):
\[ r \subseteq r_1 \Join r_2 \Join \ldots \Join r_n \]

But the following is not always true:
\[ r \supseteq r_1 \Join r_2 \Join \ldots \Join r_n \]

Example:
<table>
<thead>
<tr>
<th></th>
<th>r</th>
<th>r_1</th>
<th>r_2</th>
</tr>
</thead>
<tbody>
<tr>
<td>SSN</td>
<td>1111 Joe</td>
<td>1111 Joe</td>
<td>1111 Joe</td>
</tr>
<tr>
<td>Name</td>
<td>1 Pine</td>
<td>Joe</td>
<td>1 Pine</td>
</tr>
<tr>
<td>Address</td>
<td>2 Oak</td>
<td>2222 Alice</td>
<td>2 Oak</td>
</tr>
<tr>
<td>Address</td>
<td>3 Pine</td>
<td>3333 Alice</td>
<td>3 Pine</td>
</tr>
</tbody>
</table>

The tuples (2222, Alice, 3 Pine) and (3333, Alice, 2 Oak) are in the join, but not in the original.

---

Lossy Decompositions: What is Actually Lost?

- In the previous example, the tuples (2222, Alice, 3 Pine) and (3333, Alice, 2 Oak) were gained, not lost!
  - Why do we say that the decomposition was lossy?
- What was lost is information:
  - That 2222 lives at 2 Oak: In the decomposition, 2222 can live at either 2 Oak or 3 Pine.
  - That 3333 lives at 3 Pine: In the decomposition, 3333 can live at either 2 Oak or 3 Pine.
Testing for Losslessness

- A (binary) decomposition of $R = (R, F)$ into $R_1 = (R_1, F_1)$ and $R_2 = (R_2, F_2)$ is lossless if and only if:
  - either the FD $(R_1 \cap R_2) \rightarrow R_i$ is in $F^*$
  - or the FD $(R_1 \cap R_2) \rightarrow R_i$ is in $F^*$

Intuition Behind the Test for Losslessness

- Suppose $R_1 \cap R_2 \rightarrow R_i$. Then a row of $r_i$ can combine with exactly one row of $r_j$ in the natural join (since in $r_j$, a particular set of values for the attributes in $R_1 \cap R_2$ defines a unique row)

\[
\begin{array}{c|c|c}
R_1 \cap R_2 & R_i \\
\hline
\vdots & \vdots & \vdots \\
a & b & c \\
\vdots & \vdots & \vdots \\
r_1 & r_2
\end{array}
\]

Proof of Lossless Condition

- $r \subseteq r_1 \Join r_2$ — this is true for any decomposition

- $r \supseteq r_1 \Join r_2$

If $R_1 \cap R_2 \rightarrow R_i$ then

$\text{card}(r_1 \Join r_2) \subseteq \text{card}(r)$

(since each row of $r_i$ joins with exactly one row of $r_j$)

But $\text{card}(r) \geq \text{card}(r_1)$ (since $r_1$ is a projection of $r$) and therefore $\text{card}(r) \geq \text{card}(r_1 \Join r_2)$

Hence $r \supseteq r_1 \Join r_2$

Dependency Preservation

- Consider a decomposition of $R = (R, F)$ into $R_1 = (R_1, F_1)$ and $R_2 = (R_2, F_2)$

  - An FD $X \rightarrow Y$ of $F$ is in $F_i$ iff $X \cup Y \subseteq R_i$
  - An FD, $f \in F$ may be in neither $F_1$, nor $F_2$, nor even $(F_1 \cup F_2)^*$
    - Checking that $f$ is true in $r_i$ or $r_2$ is (relatively) easy
    - Checking $f$ in $r_i \Join r_2$ is harder — requires a join
    - Ideally: want to check FDs locally, in $r_i$ and $r_2$, and have a guarantee that every $f \in F$ holds in $r_i \Join r_2$
  - The decomposition is dependency preserving iff the sets $F$ and $F_1 \cup F_2$ are equivalent: $F^* = (F_1 \cup F_2)^*$

  - Then checking all FDs in $F$, as $r_1$ and $r_2$ are updated, can be done by checking $F_1$ in $r_1$ and $F_2$ in $r_2$.

Dependency Preservation

- If $f$ is an FD in $F$, but $f$ is not in $F_1 \cup F_2$, there are two possibilities:
  - $f \in (F_1 \cup F_2)^*$
    - If the constraints in $F_1$ and $F_2$ are maintained, $f$ will be maintained automatically.
  - $f \notin (F_1 \cup F_2)^*$
    - $f$ can be checked only by first taking the join of $r_1$ and $r_2$. This is costly.
Example

Schema \((R, F)\) where
- \(R = \{\text{SSN, Name, Address, Hobby}\}\)
- \(F = \{\text{SSN} \rightarrow \text{Name, Address}\}\)
can be decomposed into
- \(R_1 = \{\text{SSN, Name, Address}\}\)
- \(R_2 = \{\text{SSN} \rightarrow \text{Name, Address}\}\)
and
- \(R_2 = \{\text{SSN, Hobby}\}\)
- \(F_2 = \{\}\)
Since \(F = F_1 \cup F_2\) the decomposition is dependency preserving.

Example

- HaveAccount (AccountNumber, ClientId, OfficeId)
  - \(f_1: \text{AccountNumber} \rightarrow \text{OfficeId}\)
  - \(f_2: \text{ClientId, OfficeId} \rightarrow \text{AccountNumber}\)
- Decomposition:
  - Given: \(R\)
  - Not Decomposition:
  - Hence: \(BCNF + \text{lossless+dependency preserving}\)
- \(\text{HasAccount}\)
- \(\text{AccountNumber}\)
- \(\text{ClientNumber} = (\text{ClientId} \rightarrow \text{ClientName})\)
- \(\text{Address}\)
- \(\text{Hobby}\)
- \(\text{Depression F1 = (F1 \cup F2)^+}\)
- Note 1: FDs \(T = \{ABH\}\)
  - Hence, decomposition is \(R_1\)
- Note 2: Decomposition is \(R_2\)
  - Note 3: FDs \(T = \{\text{AccountNumber}, \text{OfficeId}\}\)
  - \(\text{HasAccount}\)
  - \(\text{AccountNumber}\)
  - \(\text{ClientNumber} = (\text{ClientId} \rightarrow \text{ClientName})\)
  - \(\text{Address}\)
  - \(\text{Hobby}\)
- \(\text{Depression F1 = (F1 \cup F2)^+}\)
  - So \(F^* = (F_1 \cup F_2)^*\) and thus the decompositions is still dependency preserving.

Example

- Schema: \((ABC; F)\), \(F = \{A \rightarrow B, B \rightarrow C, C \rightarrow B\}\)
- Decomposition:
  - \((AC, F_1), F_1 = \{A \rightarrow C\}\)
    - Note: \(A \rightarrow C \notin F\) but in \(F^*\)
    - \((BC, F_2), F_2 = \{B \rightarrow C, C \rightarrow B\}\)
- \(A \rightarrow B \in (F_1 \cup F_2)\), but \(A \rightarrow B \notin (F_1 \cup F_2)^*\).
  - So \(F^* = (F_1 \cup F_2)^*\) and thus the decompositions is still dependency preserving.

BCNF Decomposition Algorithm

Input: \(R = (R; F)\)

Decomp := \(R\)

while there is \(S = (S; F') \in \text{Decomp}\) and \(S\) not in \(\text{BCNF}\) do
- Find \(X \rightarrow Y \in F'\) that violates \(\text{BCNF}\) \(// X\) isn’t a superkey in \(S\)
- Replace \(S\) in \(\text{Decomp}\) with \(S_1 = (XY; F_1), S_2 = (S \setminus (Y \cdot X); F_2)\)
- \(F = \text{all FDs of } F'\) involving only attributes of \(Y\)
- \(F_2 = \text{all FDs of } F'\) involving only attributes of \(S \setminus (Y \cdot X)\)
end
return Decompr

Example

- Given: \(R = (R; T)\) where \(R = ABCDEFGH\) and
  - \(T = \{ABH \rightarrow C, A \rightarrow DE, BGH \rightarrow F, F \rightarrow ADH, BH \rightarrow GE\}\)
- step 1: Find a FD that violates BCNF
  - Not \(ABH \rightarrow C\) since \((ABH)^+\) includes all attributes \(\text{(BH is a key)}\)
  - \(A \rightarrow DE\) violates BCNF since \(A\) is not a superkey \(\text{(A* = ADE)}\)
- step 2: Split \(R\) into:
  - \(R_1 = (ADE, (A \rightarrow DE))\)
  - \(R_2 = (ABCFGH; (ABH \rightarrow C, BGH \rightarrow F, F \rightarrow ADH, BH \rightarrow GE))\)
Note 1: \(R_1\) is in BCNF
- Note 2: Decomposition is \(\text{lossless}\) since \(A\) is a key of \(R_1\)
- Note 3: FDs \(F \rightarrow D\) and \(BH \rightarrow E\) are not in \(T_1\) or \(T_2\)
- Both can be derived from \(T_1 \cup T_2\)
  - \(E \rightarrow F\) and \(A \rightarrow D\) implies \(F \rightarrow D\)
Hence, decomposition \(\text{is dependency preserving}.

Properties of BCNF Decomposition Algorithm

Let \(X \rightarrow Y\) violate BCNF in \(R_1\) and \(R_2\) than there were in \(R\)
- There are fewer violations of BCNF in \(R_1\) and \(R_2\) than there were in \(R\)
  - \(X \rightarrow Y\) implies \(X\) is a key of \(R_1\)
  - Hence \(X \rightarrow Y\) is not a key of \(R_2\)
  - Suppose \(f \in F\) does not violate BCNF in \(R_2\) and, since \(X \rightarrow Y\) is not a key of \(R_2\), does not violate BCNF in \(R_2\) either
  - \(f \in F\) doesn’t violate BCNF in \(R_1\)
  - \(R_1\) is in BCNF
- The decomposition \(\text{is lossless}\)
  - \(f \in F\) and \(F\) is in BCNF in \(R_1\) or \(R_2\) either since \(X\) is a superkey of \(R_1\) and hence also of \(R_3\) and \(R_4\).
Example (con’t)

Given: \( R_2 = (ABC\{GH\}; \{ ABH, BGH \rightarrow F, F \rightarrow AH, BH \rightarrow G \}) \)

step 1: Find a FD that violates BCNF.  
Not \( ABH \rightarrow C \) or \( BGH \rightarrow F \), since \( BH \) is a key of \( R_2 \).  
\( F \rightarrow AH \) violates BCNF since \( F \) is not a superkey (\( F^+ = AH \)).

step 2: Split \( R_2 \) into:  
\( R_{21} = (FAH, \{ F \rightarrow AH \}) \)  
\( R_{22} = (BCFG, \{ \}) \)

Note 1: Both \( R_{21} \) and \( R_{22} \) are in BCNF.

Note 2: The decomposition is lossless (since \( F \) is a key of \( R_{21} \)).

Note 3: FDs \( ABH \rightarrow C \), \( BGH \rightarrow F \), \( BH \rightarrow G \) are not in \( T_{21} \) or \( T_{22} \), and they can’t be derived from \( T_1 \cup T_{21} \cup T_{22} \).  
Hence the decomposition is not dependency-preserving.

Properties of BCNF Decomposition Algorithm

- A BCNF decomposition is not necessarily dependency preserving.
- But always lossless.
- BCNF+lossless+dependency preserving is sometimes unachievable (recall HasAccount).

Minimal Cover

- A minimal cover of a set of dependencies, \( T \), is a set of dependencies, \( U \), such that:  
  - \( U \) is equivalent to \( T \) (\( T^+ = U^+ \))
  - All FDs in \( U \) have the form \( X \rightarrow A \) where \( A \) is a single attribute.
  - It is not possible to make \( U \) smaller (while preserving equivalence) by:  
    - Deleting an FD.
    - Deleting an attribute from an FD (either from LHS or RHS).
  - FDs and attributes that can be deleted in this way are called redundant.

Computing Minimal Cover

- Example: \( T = \{ ABH \rightarrow CK, A \rightarrow D, C \rightarrow E, 
  BGH \rightarrow F, F \rightarrow AD, E \rightarrow F, BH \rightarrow E \} \)
- step 1: Make RHS of each FD into a single attribute.  
  - Algorithm: Use the decomposition inference rule for FDs.  
    - Example: \( F \rightarrow AD \) replaced by \( F \rightarrow A, F \rightarrow D \); \( AB \rightarrow CK \) by \( AB \rightarrow C, ABH \rightarrow K \).
- step 2: Eliminate redundant attributes from LHS.  
  - Algorithm: If FD \( XB \rightarrow A \in T \) (where \( B \) is a single attribute) and \( X \rightarrow A \) is entailed by \( T \), then \( B \) was unnecessary.
  - Example: Can an attribute be deleted from \( ABH \rightarrow C \) ?
    - Compute \( ABH \), \( ABH^* \), \( BHF \)
    - Since \( C \in BHF \), \( BHF \rightarrow C \) is entailed by \( T \) and \( A \) is redundant in \( ABH \rightarrow C \).

Computing Minimal Cover (con’t)

- step 3: Delete redundant FDs from \( T \)
  - Algorithm: If \( T - f \) entails \( f \), then \( f \) is redundant.
    - If \( f \) is \( X \rightarrow A \) then check if \( A \in X^+ \).
  - Example: \( BGH \rightarrow F \) is entailed by \( E \rightarrow F, BH \rightarrow E \), so it is redundant.

- Note: The order of steps 2 and 3 cannot be interchanged!! See the textbook for a counterexample.
Synthesizing a 3NF Schema

Starting with a schema \( R = (R, T) \)

- **step 1**: Compute a minimal cover, \( U \), of \( T \). The decomposition is based on \( U \), but since \( U^+ = T^+ \) the same functional dependencies will hold
  - A minimal cover for \( U \)
  
  \[ T = \{ ABH \rightarrow CK, A \rightarrow D, C \rightarrow E, BGH \rightarrow F, F \rightarrow AD, E \rightarrow F, BH \rightarrow E \} \]
  
  is
  
  \[ U = \{ BH \rightarrow C, BH \rightarrow K, A \rightarrow D, C \rightarrow E, F \rightarrow A, E \rightarrow F \} \]

- **step 2**: Partition \( U \) into sets \( U_1, U_2, \ldots, U_n \) such that the LHS of all elements of \( U_i \) are the same
  
  \[ U_1 = \{ BH \rightarrow C, BH \rightarrow K \}, U_2 = \{ A \rightarrow D \}, U_3 = \{ C \rightarrow E \}, U_4 = \{ F \rightarrow A \}, U_5 = \{ E \rightarrow F \} \]

- **step 3**: For each \( U_i \) form schema \( R_i = (R_i, U_i) \), where \( R_i \) is the set of all attributes mentioned in \( U_i \)
  
  - Each FD of \( U \) will be in some \( R_i \). Hence the decomposition is dependency preserving
  
  \[ R_1 = (BHC; BH \rightarrow C, BH \rightarrow K), R_2 = (AD; A \rightarrow D), R_3 = (CE; C \rightarrow E), R_4 = (FA; F \rightarrow A), R_5 = (EF; E \rightarrow F) \]

Synthesizing a 3NF schema (con’t)

- **step 4**: If no \( R_i \) is a superkey of \( R \), add schema \( R_0 = (R_0, \{\}) \) where \( R_0 \) is a key of \( R \).
  
  \[ R_0 = (BGH, \{\}) \]
  
  - \( R_0 \) might be needed when not all attributes are necessarily contained in \( R_1, R_2, \ldots, R_n \)
  
  - A missing attribute, \( A \), must be part of all keys (since it’s not in any FD of \( U \), deriving a key constraint from \( U \) involves the augmentation axiom)
  
  - \( R_0 \) might be needed even if all attributes are accounted for in \( R_1, R_2, \ldots, R_n \)
  
  - Example: \( (ABCD; A \rightarrow B, C \rightarrow D) \). Step 3 decomposition:
    
    \[ R_1 = (AB; A \rightarrow B), R_2 = (CD; C \rightarrow D) \]. Lossy! Need to add \( (AC; \{\}) \) for losslessness
  
  - Step 4 guarantees lossless decomposition.

Synthesizing a 3NF schema (con’t)

- **step 5**: If no \( R_i \) is a superkey of \( R \), add schema \( R_0 = (R_0, \{\}) \) where \( R_0 \) is a key of \( R \).
  
  \[ R_0 = (BGH, \{\}) \]
  
  - \( R_0 \) might be needed when not all attributes are necessarily contained in \( R_1, R_2, \ldots, R_n \)
  
  - A missing attribute, \( A \), must be part of all keys (since it’s not in any FD of \( U \), deriving a key constraint from \( U \) involves the augmentation axiom)
  
  - \( R_0 \) might be needed even if all attributes are accounted for in \( R_1, R_2, \ldots, R_n \)
  
  - Example: \( (ABCD; A \rightarrow B, C \rightarrow D) \). Step 3 decomposition:
    
    \[ R_1 = (AB; A \rightarrow B), R_2 = (CD; C \rightarrow D) \]. Lossy! Need to add \( (AC; \{\}) \) for losslessness
  
  - Step 4 guarantees lossless decomposition.

BCNF Design Strategy

- The resulting decomposition, \( R_0, R_1, \ldots, R_n \), is
  
  - Dependency preserving (since every FD in \( U \) is a FD of some schema)
  
  - Lossless (although this is not obvious)
  
  - In 3NF (although this is not obvious)

- Strategy for decomposing a relation
  
  - Use 3NF decomposition first to get lossless, dependency preserving decomposition
  
  - If any resulting schema is not in BCNF, split it using the BCNF algorithm (but this may yield a non-dependency preserving result)

Normalization Drawbacks

- By limiting redundancy, normalization helps maintain consistency and saves space

- But performance of querying can suffer because related information that was stored in a single relation is now distributed among several

- Example: A join is required to get the names and grades of all students taking CS305 in S2002.

```
SELECT S.Name, T.Grade
FROM Student S, Transcript T
WHERE S.Id = T.StudId AND T.CrsCode = 'CS305' AND T.Semester = 'S2002'
```
Denormalization

- **Tradeoff:** Judiciously introduce redundancy to improve performance of certain queries
- **Example:** Add attribute `Name` to `Transcript`

```
SELECT T.Name, T.Grade
FROM Transcript T
WHERE T.CrsCode = 'CS305' AND T.Semester = 'S2002'
```

- Join is avoided
- If queries are asked more frequently than `Transcript` is modified, added redundancy might improve average performance
- But, `Transcript` is no longer in BCNF since key is `SuidId, CrsCode, Semester` and `SuidId` → `Name`

---

**Multi-Valued Dependency**

- **Problem:** multi-valued (or binary join) dependency
  - **Definition:** If every instance of schema `R` can be (losslessly) decomposed using attribute sets `(X, Y)` such that:

    `r = π_X(r) △ Y π_Y(r)`

    then a multi-valued dependency

    `R = X △ Y`

    holds in `r`

- **Ex:** `SSN` does not uniquely determine `PhoneN` or `ChildSSN`, thus `Person` is not in 4NF.

---

**Fourth Normal Form**

- A schema is in **fourth normal form** (4NF) if for every non-trivial multi-valued dependency:

  `R = X △ Y`

  either:
  - `X ⊆ Y` or `Y ⊆ X` (trivial case); or
  - `X ∩ Y` is a superkey of `R` (i.e., `X ∩ Y → R`)

---

**4NF Implies BCNF**

- Suppose `R` is in 4NF and `X → Y` is an FD.
  - `R1 = XY`, `R2 = R → Y` is a lossless decomposition of `R`
  - Thus `R` has the multi-valued dependency:

    `R = X △ Y`

  - Since `R` is in 4NF, one of the following must hold:
    - `X ⊆ Y` or `Y ⊆ X` (impossibility)
    - `R → Y ≤ XY` (i.e., `R = XY` and `X` is a superkey)
    - `XY ∩ R → Y` (= `X`) is a superkey
  - Hence `X → Y` satisfies BCNF condition