LOGICAL DESIGN OF RELATIONAL DATABASES

The process of logical design of a relational database deals with the task of designing the logical structure of a relational database which contains information about some particular enterprise. This process consists of two phases:

1. Decide what is to be stored in the database. That is, gather information which is deemed to be important for the enterprise and thus needs to be stored in the database, and the semantic constraints among such information; identify such information with attributes, and identify the constraints.

2. Decide the relational scheme, i.e., decide how the database is to be logically structured. Basically, this is to decide
   (i) Which relations are needed to represent all the information desired (which is contained in the attributes and the constraints), and
   (ii) Which attributes should be grouped together to form a relation.

In our discussion, we are interested only in phase (2) of the design process, i.e., given a set of attributes \(\{A_1, \ldots, A_n\}\), and the set of associated constraints, we are to determine a relational scheme \(S\) which consists of several relations \(R_1, \ldots, R_m\) and \(S\) possesses the desirable properties as will be described later.

In general, the goal of relational database logical design is to generate a relational scheme \(S\) that allows us to store information without unnecessary redundancy, yet allows us to retrieve data easily. The design criteria, or the desired properties of the relational scheme are the following:

1. Reduce redundancy.
2. Achieve representation power, i.e., the relations in the resulting scheme should contain all the information conveyed by the attributes and constraints.
3. Avoid loss of information.

One approach, called the normalization approach, to the relational database logical design task is to design schemes which are in certain normal form. Relations in certain normal forms possess desired properties for our goal of logical design. We will take the normalization approach in our discussion of logical database design. In taking this approach, we represent the constraints among data as functional dependencies and use these functional dependencies to obtain normal form relations. The theory of functional dependency will be introduced and the first, second, third normal forms, and the Boyce/Codd normal form will be discussed. We will learn the technique of applying functional dependency theory to obtain a good logical relational database design.
DESIR ED PROPERTIES OF A GOOD LOGICAL DESIGN

As we said in the beginning, the desired properties of a good logical design should be the reduction of information redundancy, the ability to represent desired information of the enterprise and the ability to avoid loss of information.

Example 1. (Redundancy of information.) Suppose we have a database describing suppliers, parts, and the "shipment" of parts by suppliers (with some quantities). Assume that each supplier is identified by S#, and each has a descriptive attribute STATUS. We also assume that each part is identified by P#, and each part has a descriptive attribute COLOR. The "shipment" relationship between supplier and part is "many-to-many": A supplier can supply many parts (with possibly different quantities), and each part can be supplied by many suppliers.

Clearly, the best way to model this suppliers-parts database is with 3 relations:

\[
S' = (S#, \text{STATUS}) \text{ describing suppliers} \\
P' = (P#, \text{COLOR}) \text{ describing parts} \\
SP = (S#, P#, \text{QTY}) \text{ describing shipments}
\]

Now consider the relational schemas which form a POOR design of the database:

\[
SP' = (S#, P#, \text{QTY, STATUS}), \text{ describing the suppliers and shipments} \\
P' = (P#, \text{COLOR}) \text{ describing parts}
\]

Now the relation SP' is the following:

<table>
<thead>
<tr>
<th>S#</th>
<th>P#</th>
<th>QTY</th>
<th>STATUS</th>
</tr>
</thead>
<tbody>
<tr>
<td>s1</td>
<td>p1</td>
<td>100</td>
<td>30</td>
</tr>
<tr>
<td>s1</td>
<td>p2</td>
<td>200</td>
<td>30</td>
</tr>
<tr>
<td>s1</td>
<td>p3</td>
<td>150</td>
<td>30</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Obviously, the decision to use SP' to represent supplier status and his shipments in one relation is not a good choice because we have lots of repetition of information in the relation: the status value for each supplier is repeated as many times as his shipments. Now if we want to modify the status of s1, we have to go through the table to update every tuple which describes a shipment from s1. If s1 supplies 1000 items, the cost of such an update will be 1000 times updates! However, if we use the first design, namely, if we split the relation SP' into two smaller relations S'(S#, STATUS) and SP(S#, P#, QTY), such problem is completely eliminated. ♣

Example 2. (Bad design results in the inability to represent certain information.) Still consider the relation SP' (S#, P#, QTY, STATUS). The information "supplier s2 has status value 50" is a desired information irrespective whether s2 supplies some parts or not. But if s2 does not supply any parts, we can not represent the above information in the SP' relation, because the primary key of SP' = (S#, P#) and the attribute P# can not accept null-values. ♣

Example 3. (Bad design results in loss of information.) Examples 1 and 2 suggest that sometimes we need to decompose a relation into several relations to reduce redundancy and to achieve the desired representation power. However, a bad decomposition will cause loss of information as shown by this example. Assume that the SP' relation is the following:
we decompose SP’ into SP’’ = (S#, P#, QTY) and S-Q = (QTY, STATUS):

\[
\begin{array}{cccc}
S# & P# & QTY & STATUS \\
\hline
s1 & p1 & 100 & 30 \\
s1 & p2 & 200 & 30 \\
s2 & p3 & 100 & 50 \\
\end{array}
\]

\[
\begin{array}{cccc}
\text{Relation SP’} & \text{Relation S-Q} \\
S# & P# & QTY & QTY & STATUS \\
\hline
s1 & p1 & 100 & 100 & 30 \\
s1 & p1 & 100 & 200 & 30 \\
s1 & p2 & 200 & 200 & 30 \\
s2 & p3 & 100 & 100 & 50 \\
\end{array}
\]

Now suppose that we want to get the information about the status of s1, using a natural join of SP’’ and S-Q. The resulting relation of the natural join becomes the following:

\[
\begin{array}{cccc}
\text{The relation obtained by joining SP’’ and S-Q} \\
S# & P# & QTY & STATUS \\
\hline
s1 & p1 & 100 & 30 \\
s1 & p1 & 100 & 50 (*) \\
s1 & p2 & 200 & 30 \\
s2 & p3 & 100 & 30 (*) \\
s2 & p3 & 100 & 50 \\
\end{array}
\]

Note that we get 5 tuples instead of the original 3 in the relation SP’. Although we have more tuples, we have less information, because we can no longer decide the status of s1 or s2 from the above relation, simply due to the two spurious tuples (marked by (*)). So the desired information is lost as a result of bad design.

From the above examples, we can see that bad logical design can result in a database with lots of redundancy and the scheme fails to represent the desired information. The solution to these problems is to use functional dependency as constraints and to apply the normalization technique in order to form the normalized relations which can avoid the problems of redundancy and loss of information.
FUNCTIONAL DEPENDENCY

Given a relation schema $R(A_1, ..., A_n)$, and $X \subseteq R$, $Y \subseteq R$, we say $R$ satisfies the functional dependency (fd) $X \rightarrow Y$, i.e., $X$ functionally determines $Y$, or $Y$ is functionally determined by $X$, or $Y$ is functionally dependent on $X$, if for each legal relation $r(R)$ (i.e., $r$ is a specific relation with attributes $A_1, ..., A_n$), $r$ satisfies the functional dependency $X \rightarrow Y$.

A specific relation $r$ satisfies $X \rightarrow Y$ if each $X$-value in $r$ is associated with a unique $Y$-value in $r$. In other words, $X$ functionally determines $Y$ in a relation $r$ if

\[ t_1[X] = t_2[X] \Rightarrow t_1[Y] = t_2[Y]. \]

As an example of functional dependency, consider a relation schema $R$ which has four attributes MOTHER, CHILD, GIFT, DATE, describing the mother’s gift-giving to her child on certain date. It is easy to see that each child has precisely one (biological) mother, while one mother can have several children. That means if two tuples in $r(R)$ have the same CHILD value (denoting the same child), they must have the same MOTHER value (denoting the child’s mother). So we say that CHILD functionally determines MOTHER or MOTHER is functionally determined by CHILD. Namely, $\text{CHILD} \rightarrow \text{MOTHER}$. If we also assume that each child gets at most one gift on a fixed date, then we will also have the functional dependency (CHILD, DATE) $\rightarrow$ GIFT.

(Think: What are the candidate keys of this relation schema $R$?)

An attribute $Y$ of relation schema $R$ is said to be fully functionally dependent on attribute $X$ of $R$, if $Y$ is functionally dependent on $X$ but $Y$ is not functionally dependent on any proper subset of $X$.

In the relation $SP'(S\#, \ P\#, \ QTY, \ STATUS)$, the attribute STATUS is functionally dependent on the attribute $(S\#, \ P\#)$, but it is not fully functionally dependent on $(S\#, \ P\#)$ because STATUS is functionally dependent on $S\#$ which is a proper subset of the attribute $(S\#, \ P\#)$. Think: in the above relation schema $R(\text{MOTHER}, \text{CHILD}, \text{GIFT}, \text{DATE})$ with functional dependencies $F = \{\text{CHILD} \rightarrow \text{MOTHER}, \ (\text{CHILD}, \text{DATE}) \rightarrow \text{GIFT}\}$, is the attribute MOTHER fully functionally dependent on (CHILD, DATE)!

Consider the following sample data (instance) for the relation $r(R)$:

<table>
<thead>
<tr>
<th>MOTHER</th>
<th>CHILD</th>
<th>GIFT</th>
<th>DATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linda</td>
<td>Jim</td>
<td>Bike</td>
<td>1-20-2010</td>
</tr>
<tr>
<td>Linda</td>
<td>Jim</td>
<td>Sweater</td>
<td>12-25-2008</td>
</tr>
<tr>
<td>Linda</td>
<td>Jim</td>
<td>CD</td>
<td>3-8-2007</td>
</tr>
<tr>
<td>Linda</td>
<td>Mary</td>
<td>Game</td>
<td>12-25-2008</td>
</tr>
<tr>
<td>Linda</td>
<td>Mary</td>
<td>Book</td>
<td>2-20-2010</td>
</tr>
<tr>
<td>Susan</td>
<td>David</td>
<td>CD</td>
<td>7-20-2009</td>
</tr>
<tr>
<td>Susan</td>
<td>David</td>
<td>Shoes</td>
<td>8-30-2009</td>
</tr>
</tbody>
</table>

Do you see any problems (redundancy, etc.) from the above sample data?
EXAMPLE OF FUNCTIONAL DEPENDENCY

The two most important things to remember about functional dependency (fd) are:

(1) Fd’s are determined by the meaning of the attributes and their role in the "real world" which is being modeled by the database.

(2) Fd’s are in turn used to group the attributes together to form the normalized relations of the database.

Example. Consider the attributes

<table>
<thead>
<tr>
<th>C</th>
<th>Class (of a course)</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>Time for the class</td>
</tr>
<tr>
<td>R</td>
<td>Room for the class</td>
</tr>
<tr>
<td>I</td>
<td>Instructor of the class</td>
</tr>
</tbody>
</table>

which describe the arrangement of room and time for classes taught by the instructors. These attributes are used to model part of the "real world" in which we have classes at certain time in certain room lectured by certain instructors. We have certain constraints about the objects (such as class, time, etc.) in the "real world" and such constraints are in turn represented in terms of functional dependencies. The constraints are:

(1) No two instructors teach the same course.

(2) At any time and in a given room, there is at most one class being taught there.

(3) No class can be taught at one given time in two rooms.

(4) No instructor can teach two classes at one given time.
The functional dependencies are:

(a) \( C \rightarrow I \)
(b) \( TR \rightarrow C \)
(c) \( CT \rightarrow R \)
(d) \( IT \rightarrow C \)

Given below is a database which has some tuples violating the above functional dependencies.

<table>
<thead>
<tr>
<th>Class</th>
<th>Instructor</th>
<th>Time</th>
<th>Room</th>
</tr>
</thead>
<tbody>
<tr>
<td>Csc4402</td>
<td>Chen</td>
<td>1:30/M</td>
<td>101</td>
</tr>
<tr>
<td>His7700</td>
<td>Brown</td>
<td>3:00/T</td>
<td>205</td>
</tr>
<tr>
<td>Csc4402</td>
<td>Smith</td>
<td>2:00/F</td>
<td>310 (a)</td>
</tr>
<tr>
<td>Art2450</td>
<td>John</td>
<td>3:00/T</td>
<td>205 (b)</td>
</tr>
<tr>
<td>His7700</td>
<td>Brown</td>
<td>3:00/T</td>
<td>208 (c)</td>
</tr>
<tr>
<td>Csc7420</td>
<td>Chen</td>
<td>1:30/M</td>
<td>202 (d)</td>
</tr>
</tbody>
</table>

It is easy to see that the tuples marked (a), (b), (c), (d) violate the functional dependencies (a), (b), (c), (d) respectively.
AXIOMS FOR FUNCTIONAL DEPENDENCIES

Logical implications of dependencies. Let $R(ABC)$ be a relation schema and $F = \{ A \rightarrow B, B \rightarrow C \}$ be the set of functional dependencies that hold on $R$. It is easy to see that the fd $A \rightarrow C$ also holds in $R$.

In general, let $F$ be a set of functional dependencies and let $X \rightarrow Y$ be a functional dependency. We say $F$ logically implies $X \rightarrow Y$, written $F \models X \rightarrow Y$, if every relation instance $r$ that satisfies $F$ also satisfies $X \rightarrow Y$.

The closure of $F$

The closure of $F$, denoted as $F^+$, is defined as the set of all the functional dependencies that are logically implied by $F$. That is, $F^+ = \{ X \rightarrow Y | F \models X \rightarrow Y \}$.

Axioms for Functional Dependencies

To determine candidate keys, and to understand logical implications among functional dependencies, we need to compute $F^+$, or at least, to tell, for given $F$ and fd $X \rightarrow Y$, whether $X \rightarrow Y$ is in $F^+$. So we need inference rules to tell us how one or more fd’s imply other fd’s. The set of rules is called Armstrong’s axioms, or Armstrong’s rules. These rules are both sound and complete.

1. The soundness of the rules refers to the fact that using the rules, we can not deduce from $F$ any fd’s which are not in $F^+$.

2. The completeness of the Armstrong’s rules refers to the fact that given a set of functional dependencies $F$, we can deduce all the dependencies in $F^+$ from $F$ using these rules.

Let $F^* = F \cup \{ X \rightarrow Y | X \rightarrow Y$ is deduced from $F$ using the Armstrong’s rules$\}$. Here $F^*$ is the set of functional dependencies derivable from $F$ using the Armstrong’s rules. The soundness of the Armstrong’s rules tells us $F^* \subseteq F^+$ and the completeness of the rules indicates $F^* \subseteq F^+$. Together, we have $F^* = F^+$. In other words, an fd $X \rightarrow Y$ belongs to $F^+$ if and only if $X \rightarrow Y$ can be deduced from $F$ by using the Armstrong’s rules.

Now we can define $F^+$ in terms of $F$ and Armstrong’s rules. The closure $F^+$ of $F$ is the set of all fd’s which can be derived using the rules (1) - (3) (in next page) starting with the fd’s in $F$, and including those in $F$. 


ARMSTRONG’S RULES

In the example relation R(ABC) given earlier, we noticed that from the following functional dependencies \( \{ A \rightarrow B, B \rightarrow C \} \), we can deduce \( A \rightarrow C \) by transitivity, and deduce \( AB \rightarrow C \) by a simple application of the definition of functional dependency. In fact, using the following Armstrong’s rules, we can deduce \( AB \rightarrow C \) from either \( A \rightarrow C \) or \( B \rightarrow C \).

Armstrong’s rules:

(1) \( X \rightarrow X' \) for all subsets \( X' \subseteq X \) (projection rule)
(2) if \( X \rightarrow Y \) then \( XZ \rightarrowYZ \) (augmentation rule)
(3) if \( X \rightarrow Y \) and \( Y \rightarrow Z \) then \( X \rightarrow Z \) (transitive rule)

We can also derive many other rules by repeated application of the rules (1) - (3). For example, we can obtain the union rule:

if \( X \rightarrow Y \) and \( X \rightarrow Z \) then \( X \rightarrowYZ \).

\[
\begin{align*}
X \rightarrow Y & \text{ gives } X \rightarrow XY \text{ by (2)} \\
X \rightarrow Z & \text{ gives } XY \rightarrowYZ \text{ by (2)} \\
X \rightarrow XY \text{ and } XY \rightarrowYZ & \text{ gives } X \rightarrowYZ \text{ by (3)}
\end{align*}
\]

Also, we can derive the decomposition rule:

if \( X \rightarrowYZ \) then \( X \rightarrow Y \) and \( X \rightarrow Z \).

\[
\begin{align*}
YZ \rightarrow Y & \text{ by (1)} \\
YZ \rightarrow Z & \text{ by (1)} \\
X \rightarrowYZ \text{ and } YZ \rightarrow Y & \text{ gives } X \rightarrow Y \text{ by (3)} \\
X \rightarrowYZ \text{ and } YZ \rightarrow Z & \text{ gives } X \rightarrow Z \text{ by (3)}
\end{align*}
\]

The importance of the Armstrong’s rules lies in that they allow manipulation of the fd’s purely in a syntactic way without looking at the tuples of a relation.
DETERMINING DEPENDENT ATTRIBUTES

Let \( X \) be a set of attributes and \( F \) be a set of functional dependencies. Let \( X^+ \) be the set of all attributes that depend on a subset of \( X \) with respect to \( F \), i.e., \( X^+ \) is the set of attributes \( Z \), such that \( X \rightarrow Z \in F^+ \). \( X^+ \) is called the closure of \( X \) (w.r.t. \( F \)). By definition, \( X \subseteq X^+ \).

The algorithm for computing \( X^+ \) (w.r.t. \( F \)):

Input: A relation schema \( R \), a set of functional dependencies \( F \) and a set of attributes \( X \).
Output: \( X^+ \).

1. \( X^+ \leftarrow X \).
2. Repeat
   - found \leftarrow false;
   - for (each fd \( Y \rightarrow Z \in F \)) do
     - if (\( Y \subseteq X^+ \)) then begin
       - found \leftarrow true;
       - \( X^+ \leftarrow X^+ \cup Z \);
       - remove \( Y \rightarrow Z \) from further consideration;
     - end;
   - until (found = false) or (\( X^+ = \) all attributes).

Example. The computation of \((AC)^+ = ABCDE\) w.r.t. \( F = \{AB \rightarrow E, AD \rightarrow B, B \rightarrow C, C \rightarrow D\} \) is shown below.

<table>
<thead>
<tr>
<th>Fd’s</th>
<th>Value of ((AC)^+) after each iteration of repeat-loop</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>#1</td>
</tr>
<tr>
<td>( AB \rightarrow E ) ((Y = AB))</td>
<td>-</td>
</tr>
<tr>
<td>( AD \rightarrow B ) ((Y = AD))</td>
<td>-</td>
</tr>
<tr>
<td>( B \rightarrow C ) ((Y = B))</td>
<td>-</td>
</tr>
<tr>
<td>( C \rightarrow D ) ((Y = C))</td>
<td>ACD</td>
</tr>
<tr>
<td>found</td>
<td>true</td>
</tr>
</tbody>
</table>

"*" indicates that the fd will not be considered further.

Note that the order in which the fd’s are used affects the details of the progress of computation, but not the final \( X^+ \).
Example - Compact Representation of $F^+$. If $F = \{AB \rightarrow C, C \rightarrow B, C \rightarrow A\}$, then we can use the following functional dependencies as a shorthand to represent $F^+$: (The right hand side of each fd is simply the $X^+$ where $X = \text{the left hand side};$ the $X$'s are all the non-empty subsets of the attributes (ABC)).

\[
\begin{align*}
A & \rightarrow A \\
B & \rightarrow B \\
C & \rightarrow ABC \\
AB & \rightarrow ABC \\
AC & \rightarrow ABC \\
BC & \rightarrow ABC \\
ABC & \rightarrow ABC
\end{align*}
\]

Note that we do not mean that the $F^+$ consists of only the above 7 functional dependencies. In fact, each $X \rightarrow Y$ in the above represents the set of all the fd's in the form of $X \rightarrow Z$, where $Z$ is a subset of $Y$. For example, the fd $AC \rightarrow ABC$ represents the set of fd's $\{AC \rightarrow A, AC \rightarrow B, AC \rightarrow C, AC \rightarrow AB, AC \rightarrow AC, AC \rightarrow BC, AC \rightarrow ABC\}$. The reason that we represent $F^+$ only with the above 7 functional dependencies is for the sake of clarity and conciseness.
EQUIVALENCE AND REDUNDANCY OF FD'S

Two sets of fd's $F_1$ and $F_2$ are said to be equivalent, $F_1 \equiv F_2$, if they have the same closure:

$$F_1^+ = F_2^+,$$

or equivalently,

$$F_1 \subseteq F_2^+ \text{ and } F_2 \subseteq F_1^+.$$

Note that to test $F_1 \subseteq F_2^+$, we need not compute all of $F_2^+$. We need only to verify that for each $X \rightarrow Y \in F_1$, we have $X \rightarrow Y \in F_2^+$, i.e., the closure $X^+$ computed using $F_2$ contains $Y$.

**Example.** Let $F_1 = \{A \rightarrow B, B \rightarrow C, C \rightarrow A\}$ and $F_2 = \{A \rightarrow C, C \rightarrow B, B \rightarrow A\}$. Then $F_1 \equiv F_2$.

This is true because $A^+$ using $F_2$ equals ABC which contains B. Thus $A \rightarrow B \in F_2^+$. Similarly, $B \rightarrow C$ and $C \rightarrow A$ are in $F_2^+$, etc.

**Redundant fd.** An fd $f \in F$ is redundant if $F \equiv F - \{f\}$.

**Example.** $A \rightarrow C$ is redundant in $\{A \rightarrow B, B \rightarrow C, A \rightarrow C\}$.

**Theorem.** $F_1 \equiv F_2$ if and only if for each subset of attributes $X$, $X_1^+ = X_2^+$, where $X_i^+$ is the closure of $X$ computed using the fd's in $F_i$.

**Theorem.** An fd $X \rightarrow Y$ in $F$ is redundant if and only if $Y \subseteq X^+$, where $X^+$ is computed without using the fd $X \rightarrow Y$.

A set of fd's is said to be *reduced* if it contains no redundant fd's.

**Example.** The set $F = \{A \rightarrow B, B \rightarrow C, C \rightarrow A, A \rightarrow C, C \rightarrow B, B \rightarrow A\}$ can be reduced in two ways by successively removing the redundant fd's.

1. $\{A \rightarrow B, B \rightarrow C, C \rightarrow A\}$
2. $\{A \rightarrow C, C \rightarrow B, B \rightarrow A\}$

Note that removing a redundant fd from a set $F$ does not affect the closure $F^+$. There are some other ways to reduce the set $F$, you may try to find them.
MINIMAL COVERS

Let $F_1$ and $F_2$ be two sets of functional dependencies. If $F_1 \equiv F_2$, then we say the $F_1$ is a cover of $F_2$ and $F_2$ is a cover of $F_1$. We also say that $F_1$ covers $F_2$ and vice versa. It is easy to show that every set of functional dependencies $F$ is covered by a set of functional dependencies $G$, in which the right hand side of each fd has only one attribute.

We say a set of dependencies $F$ is minimal if:

1. Every right hand side of each fd in $F$ is a single attribute.

2. The left hand side of each fd does not have any redundant attribute, i.e., for every fd $X \rightarrow A$ in $F$ where $X$ is a composite attribute, and for any proper subset $Z$ of $X$, the functional dependency $Z \rightarrow A \notin F^+$.

3. $F$ is reduced (without redundant fd’s). This means that for every $X \rightarrow A$ in $F$, the set $F \setminus \{X \rightarrow A\}$ is NOT equivalent to $F$.

Minimal Covers of $F$. It is easy to see that for each set $F$ of functional dependencies, there exists a set of functional dependencies $F'$ such that $F \equiv F'$ and $F'$ is minimal. We call such $F'$ a minimal cover of $F$.

The algorithm to compute $F'$, a minimal cover of $F$.

Input: $F$, a set of fd’s.
output: $F'$, a minimal cover of $F$.

1. Let $F' = \{X \rightarrow A \mid X \rightarrow A \in F \text{ and } A \text{ is a single attribute}\}$. For each fd $X \rightarrow A_1A_2 \ldots A_n \in F$ ($n > 1$), put the fd’s $X \rightarrow A_1$, $X \rightarrow A_2$, ..., $X \rightarrow A_n$ into $F'$, where $A_i$ is a single attribute.

2. While there is an fd $X \rightarrow A \in F'$ such that $X$ is a composite attribute and $Z \subseteq X$ is a proper subset of $X$ and $Z \rightarrow A \in (F')^+$, do
   replace $X \rightarrow A$ with $Z \rightarrow A$.

3. For each fd $X \rightarrow A \in F'$, check if it is redundant, if it is, eliminate it. ♦

It is important to note that for the above algorithm, the ordering between step (2) and step (3) is critical: If you first perform step (3) and then perform step (2) of the algorithm, the resulting set of fd’s may still have redundant functional dependencies.

It should be pointed out that for a set of functional dependencies $F$, there may be more than one minimal covers of $F$. 
Example. (Computing a minimal cover.)

Let \( R = R(ABCDEGH) \) and \( F = \{ CD \rightarrow AB, C \rightarrow D, D \rightarrow EH, AE \rightarrow C, A \rightarrow C, B \rightarrow D \} \). The process of computing a minimal cover of \( F \) is as follows:

1. **Break down the right hand side of each fd’s.** After performing step (1) in the algorithm, we get \( F' = \{ CD \rightarrow A, CD \rightarrow B, C \rightarrow D, D \rightarrow E, D \rightarrow H, AE \rightarrow C, A \rightarrow C, B \rightarrow D \} \).

2. **Eliminate redundancy in the left hand side.** The fd \( CD \rightarrow A \) is replaced by \( C \rightarrow A \). This is because \( C \rightarrow D \in (F')^+ \), hence \( C \rightarrow CD \in (F')^+ \); from \( C \rightarrow CD \in (F')^+ \) and \( CD \rightarrow A \in F' \), by transitivity, we have \( C \rightarrow A \in (F')^+ \) and hence \( CD \rightarrow A \) should be replaced by \( C \rightarrow A \). Similarly, \( CD \rightarrow B \) is replaced by \( C \rightarrow B \), \( AE \rightarrow C \) is replaced by \( A \rightarrow C \). \( F' = \{ C \rightarrow A, C \rightarrow B, C \rightarrow D, D \rightarrow E, D \rightarrow H, A \rightarrow C, B \rightarrow D \} \) after step (2).

3. **Remove redundant fd’s.** The fd \( C \rightarrow D \) is eliminated because it can be derived from \( C \rightarrow B \) and \( B \rightarrow D \) and hence it is redundant. The \( F' \) now becomes \( \{ C \rightarrow A, C \rightarrow B, D \rightarrow E, D \rightarrow H, A \rightarrow C, B \rightarrow D \} \), which is the only minimal cover of \( F \). ♣

Example. (Computing a minimal cover.)

Let the relation \( R \) be \( R(ABCDE) \) and the set of functional dependencies be \( F = \{ AB \rightarrow C, ABC \rightarrow D, AE \rightarrow BC, BC \rightarrow AE \} \).

We compute a minimal cover of \( F \) in the following steps:

1. **Break down the right hand side of each fd’s.** \( F' = \{ AB \rightarrow C, ABC \rightarrow D, AE \rightarrow B, AE \rightarrow C, BC \rightarrow A, BC \rightarrow E \} \).

2. **Eliminate redundancy in the left hand side.** The fd \( ABC \rightarrow D \) is replaced by \( AB \rightarrow D \) because \( AB^+ = ABCDE \), and hence the attribute \( C \) in \( fd ABC \rightarrow D \) is redundant. Note that we could also replace \( ABC \rightarrow D \) by \( BC \rightarrow D \), because \( BC^+ = BCDE \). But we do NOT need to include both \( AB \rightarrow D \) and \( BC \rightarrow D \) in \( F' \); one of the two is sufficient. No other composite left hand side of fd’s can be reduced further, and thus we get \( F'' = \{ AB \rightarrow C, AB \rightarrow D, AE \rightarrow B, AE \rightarrow C, BC \rightarrow A, BC \rightarrow E \} \).

3. **Remove redundant fd’s.** The fd \( AE \rightarrow C \) is redundant because we can derive it from \( AE \rightarrow B \) and \( AB \rightarrow C \). Thus \( AE \rightarrow C \) is removed. No other fd is redundant. The final minimal cover of \( F \) is \( F_{\text{min}} = \{ AB \rightarrow C, AB \rightarrow D, AE \rightarrow B, BC \rightarrow A, BC \rightarrow E \} \).

Note: If we choose to replace \( ABC \rightarrow D \) by \( BC \rightarrow D \) in step (2) above, we would get an alternative minimal cover of \( F \). ♣
CANDIDATE KEYS

A candidate key of a relation schema R is a subset X of the attributes of R with the following two properties:

1. Every attribute is functionally dependent on X, i.e., \( X^+ = \text{all attributes of } R \) (also denoted as \( X^+ = R \)).
2. No proper subset of X has the property (1), i.e., X is minimal with respect to the property (1).

There may be more than one candidate key in a relation schema R. A primary key is a candidate key which has been specially designated (perhaps for the purpose of indexing). A sub-key of R is a subset of a candidate key; a super-key is a set of attributes containing a candidate key. Sometimes, we may simply use the term "key" instead of "candidate key" when no confusion arises. We also use the abbreviation CK to denote "candidate key".

Let R(ABCDE) be a relation schema and consider the following functional dependencies \( F = \{ AB \rightarrow E, AD \rightarrow B, B \rightarrow C, C \rightarrow D \} \). Since \( (AC)^+ = ABCDE \), \( A^+ = A \), and \( C^+ = CD \), we know that AC is a candidate key, both A and C are sub-keys, and ABC is a super-key. The only other candidate keys are AB and AD. Note that since nothing determines A, A is in every candidate key.

It is not always true that every candidate key of a relation has the same size (= 2 in the above example). For example, if we have a relation R(ABC) and the fd's \( \{ A \rightarrow BC, BC \rightarrow A \} \) (this would be the case for a small community of people with A being SSN, B the last-name and C the first-name), then A is a candidate key and so is BC. Here the two candidate keys have different number of attributes.

Since a candidate key is often used for indexing the file which stores the tuples of a relation, it is important to find a candidate key which has the smallest size. Unfortunately, computing the minimum size candidate key is a very difficult problem, in the sense that it may take very large amount of computation time if there are many attributes in R and many functional dependencies.

Given a relation R(A₁, A₂, ..., Aₙ) and a set of functional dependencies F which are true on R, how can we compute all candidate keys of R?

Since each candidate key must be a minimal subset of {A₁, ..., Aₙ} which determines all attributes of R, the most straightforward (and brute-force) way to compute all candidate keys of R is the following:

1) Construct a list L consisting of all non-empty subsets of {A₁, ..., Aₙ} (there are 2ⁿ − 1 of them). These subsets are arranged in L in ascending order of the size of the subset: We get L = (Z₁, Z₂, ..., Z₂ⁿ⁻¹), such that |Zᵢ| ≤ |Zᵢ₊₁|. Here |Zᵢ| denotes the number of elements in the set Zᵢ.

2) Initialize the set K = {} (K will contain all CK’s of R). While L is not empty, remove the first element Zᵢ from L, and compute Zᵢ⁺. If Zᵢ⁺ = R, then
   (a) Add Zᵢ to K
   (b) Remove any element Zⱼ from L if Zᵢ ⊂ Zⱼ (because Zⱼ is too big, it can not be a CK).

3) Output K as the final result.

**Analysis:** The method in previous page is correct but not efficient: the list L is too big (exponential in the number of attributes in R). Thus we need to utilize some "clever" ideas to cut down the number of \( Z_i \)'s that the algorithm has to check. The general idea of the CK computing algorithm is to focus on only attributes that will definitely appear in ANY candidate key of R, and ignore any attribute that will NEVER be part of a CK. The following notions are useful for this purpose:

**Necessary attributes:**
An attribute A is said to be a necessary attribute if

(a) A occurs only in the L.H.S. (left hand side) of the fd’s in \( F \); or

(b) A is an attribute in relation R, but A does not occur in either L.H.S. or R.H.S. of any fd in \( F \).

In other words, necessary attributes NEVER occur in the R.H.S. of any fd in \( F \).

**Useless attributes:**
An attribute A is a useless attribute if A occurs ONLY in the R.H.S. of fd’s in \( F \).

**Middle-ground attributes:**
An attribute A in relation R is a middle-ground attribute if A is neither necessary nor useless.

**Example.**
Consider the relation R(ABCDEG) with set of fd's \( F = \{ AB \rightarrow C, C \rightarrow D, AD \rightarrow E \} \)

All attributes of R are partitioned as follows:

<table>
<thead>
<tr>
<th>Necessary</th>
<th>Useless</th>
<th>Middle-ground</th>
</tr>
</thead>
<tbody>
<tr>
<td>A, B, G</td>
<td>E</td>
<td>C, D</td>
</tr>
</tbody>
</table>

An important observation about necessary attribute is: a necessary attribute will appear in every CK of R, and thus ALL necessary attributes must appear in every CK of R. Thus, if X is the collection of ALL necessary attributes (and thus X can be seen as a composite attribute), and \( X^+ \) = All attributes of R, then X must be the ONLY candidate key of R. (Think: Why is that?)

Thus we should first check the necessary attribute closure \( X^+ \), and terminate the CK computing algorithm when \( X^+ = R \). On the other hand, we also notice that useless attributes can never be part of any CK. Therefore, in case \( X^+ \neq R \), and we have to enumerate subsets \( Z_i \) of R to find CK’s, the \( Z_i \)'s should NOT contain any useless attributes, and each \( Z_i \) should contain all the necessary attributes. In fact, the list L of subsets to test should be constructed from all non-empty subsets of the middle-ground attributes, with each subset expanded to include all necessary attributes.
The algorithm for computing all candidate keys of R.

**Input:** A relation R={ A₁, A₂, ..., Aₙ }, and F, a set of functional dependencies.

**Output:** K={ K₁, ..., Kₜ }, the set of all candidate keys of R.

**Step 1.**
Set F’ to a minimal cover of F (This is needed because otherwise we may not detect all useless attributes).

**Step 2.**
Partition all attributes in R into necessary, useless and middle-ground attribute sets according to F’. Let X={ C₁, ..., Cₗ } be the necessary attribute set, Y = { B₁, ..., Bₖ } be the useless attribute set, and M = { A₁, ..., Aₙ } − (X ∪ Y) be the middle-ground attribute set.
If X={} , then go to step 4.

**Step 3.**
Compute X + . If X + =R, then set K= {X}, terminate.

**Step 4.**
Let L = (Z₁, Z₂, ..., Zₘ) be the list of all non-empty subsets of M (the middle-ground attributes) such that L is arranged in ascending order of the size of Zᵢ.
Add all attributes in X (necessary attributes) to each Zᵢ in L.

Set K = { }.
i ← 0.
WHILE L ≠ empty do
    BEGIN
        i ← i + 1.
        Remove the first element Z from L.
        Compute Z + .
        If Z + = R,
        then begin
            set K ← K ∪ {Z};
            for any Z_j ∈ L, if Z ⊂ Z_j
            then L ← L − {Z_j}.
        end
    END
Example. (Computing all candidate keys of \( R \).)

Let \( R = R(ABCDEG) \) and \( F = \{ AB \rightarrow CD, A \rightarrow B, B \rightarrow C, C \rightarrow E, BD \rightarrow A \} \). The process to compute all candidate keys of \( R \) is as follows:

1. The minimal cover of \( F \) is \( \{ A \rightarrow B, A \rightarrow D, B \rightarrow C, C \rightarrow E, BD \rightarrow A \} \).

2. Since attribute \( G \) never appears in any \( f.d. \)'s in the set of functional dependencies, \( G \) must be included in a candidate key of \( R \). The attribute \( E \) appears only in the right hand side of \( f.d. \)'s and hence \( E \) is not in any key of \( R \). No attribute of \( R \) appears only in the left hand side of the set of \( f.d. \)'s. Therefore \( X = G \) at the end of step 2.

3. Compute \( G^+ = G \), so \( G \) is not a candidate key.

4. The following table shows the \( L \), \( K \), \( Z \) and \( Z^+ \) at the very beginning of each iteration in the \textbf{WHILE} statement.

<table>
<thead>
<tr>
<th>( i )</th>
<th>( Z )</th>
<th>( Z^+ )</th>
<th>( L )</th>
<th>( K )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>–</td>
<td>–</td>
<td>{AG, BG, CG, DG, ABG, ACG, ADG, BCG, BDG, CDG, ABCG, ABDG, ACDG, BCDG, ABCDG}</td>
<td>{}</td>
</tr>
<tr>
<td>1</td>
<td>AG</td>
<td>ABCDEG = ( R )</td>
<td>{BG, CG, DG, BCG, BDG, CDG, BCDG}</td>
<td>{AG}</td>
</tr>
<tr>
<td>2</td>
<td>BG</td>
<td>BCEG ( \neq R )</td>
<td>{CG, DG, BCG, BDG, CDG, BCDG}</td>
<td>{AG}</td>
</tr>
<tr>
<td>3</td>
<td>CG</td>
<td>CEG ( \neq R )</td>
<td>{DG, BCG, BDG, CDG, BCDG}</td>
<td>{AG}</td>
</tr>
<tr>
<td>4</td>
<td>DG</td>
<td>DG ( \neq R )</td>
<td>{BCG, BDG, CDG, BCDG}</td>
<td>{AG}</td>
</tr>
<tr>
<td>5</td>
<td>BCG</td>
<td>BCEG ( \neq R )</td>
<td>{BDG, CDG, BCDG}</td>
<td>{AG}</td>
</tr>
<tr>
<td>6</td>
<td>BDG</td>
<td>ABCDEG = ( R )</td>
<td>{CDG}</td>
<td>{AG, BDG}</td>
</tr>
<tr>
<td>7</td>
<td>CDG</td>
<td>CEDG ( \neq R )</td>
<td>{}</td>
<td>{AG, BDG}</td>
</tr>
</tbody>
</table>
FIRST, SECOND NORMAL FORM

First normal form. A relation R is said to be in first normal form if all the underlying domains of R contain only atomic values. It is easy to see that if R is in first normal form, then R does not contain any repeating groups. We write 1NF as a short hand for first normal form.

Non-key attribute. An attribute A in relation R is said to be a non-key attribute if it is not a subkey, i.e., A is not a component of any candidate key K of R.

Second normal form. A relation R is said to be in second normal form if R is in 1NF and every non-key attribute is fully dependent on each candidate key K of R. In Example 1, the relation SP’ has only one candidate key (S#, P#). The attribute STATUS is a non-key attribute of SP’ and it is not fully dependent on (S#, P#), therefore SP’ is not in 2NF.

Although relations in 2NF have better properties than those in 1NF, there are still problems with them. These problems motivate the search for more "ideal" form of relations and hence the 3NF and BCNF normal forms are identified. In general, in the logical design process, we may start with a relation which is in 1NF (or even it is not in 1NF so we have to first transform it into 1NF), then use the functional dependencies as constraints to decompose R into several relations R₁, ..., Rₖ such that each Rᵢ is in 3NF (or even better, in BCNF) and the natural join of R₁, ..., Rₖ gives the same information as contained in R.
LOSS-LESS-JOIN DECOMPOSITION

The decomposition

\[(S\#, P\#, QTY, STATUS) = (S\#, P\#, QTY) \otimes (S\#, STATUS)\]

is called a *loss-less-join* (LLJ) decomposition, or we say that the decomposition is loss-less, because the join of the component relations \((S\#, P\#, QTY)\) and \((S\#, STATUS)\) gives back the original relation \((S\#, P\#, QTY, STATUS)\).

Notice that the resulting relations \(R_1 = (S\#, P\#, QTY)\) and \(R_2 = (S\#, STATUS)\) have better properties than the original relation \(SP' = (S\#, P\#, QTY, STATUS)\). For example, assume that we have only one shipment from supplier s3 which is "s3, p4, 100, 60" in \(SP'\). Now we delete this tuple from \(SP'\) because s3 stops supplying p4. Then the information "the status of s3 is 60" will be gone from relation \(SP'\), while in the case of \(R_1\) and \(R_2\), we delete the tuple "s3, p4, 100" from \(R_1\) but the tuple "s3, 60" remains to be in \(R_2\).

We must be very careful in performing the decomposition otherwise the loss-less property may not be maintained. As we have seen in Example 3, if we decompose \(SP'\) into \((S\#, P\#, QTY)\) and \((QTY, STATUS)\), the loss-less property is gone because the join of the latter two relations create some spurious tuples. Intuitively, the decomposition of \(SP' = R_1 \otimes R_2\) is loss-less because the attribute "S#" which is mutual to \(R_1\) and \(R_2\) can uniquely determine the attribute "STATUS", but the mutual attribute "QTY" in the second decomposition can not determine "STATUS". Theoretically, we have the following theorem which tells us the sufficient condition for a decomposition to be loss-less.

**Heath Theorem.** For a relation \(R\) with attributes A, B, C and functional dependency \(A \rightarrow B\), the decomposition

\[R(A, B, C) = R_1(A, B) \otimes R_2(A, C)\]

is always loss-less.

The two most desirable properties of any decomposition of a relation \(R\) are:

1. loss-less, and
2. fd-preserving.

Let \(R\) be a relation schema with functional dependencies \(F\). Assume that we have a decomposition of \(R\) into a set of relations \(R_1, R_2, ..., R_k\), and the \(F\) is subsequently decomposed into \(F_1, F_2, ..., F_k\). As we already discussed before, we want the decomposition to be "loss-less" in the sense that the natural join of the \(R_1, R_2, ..., R_k\) is exactly \(R\), i.e., \(R = R_1 \otimes R_2 \otimes ... \otimes R_k\). Another desirable property of a decomposition is functional dependency-preserving (fd-preserving), i.e., we want to maintain all the functional dependencies by requiring \(F_1 \cup F_2 \cup ... \cup F_k \equiv F\).
THIRD NORMAL FORM AND 3NF DECOMPOSITION

A relation schema $R$ with a set of fd’s $F$ is said to be in third normal form (3NF) if for each fd $X \rightarrow A \in F$ (here $A$ is a single attribute), we have

either X is a superkey

or A is a prime.

An attribute $A$ is called a prime if $A$ is a subkey, i.e., $A$ is a component of a key.

Example. The relation (ADB) with $F = \{AD \rightarrow B, B \rightarrow D\}$ is in 3NF, because AD and AB are its keys, and thus although B in $B \rightarrow D$ is not a superkey, D is a prime.

As we can see clearly, the 3NF requirement is weaker than the BCNF. As a result, we can always find a 3NF decomposition of a relation $R$ such that the decomposition is both LLJ and fd-preserving.

Bernstein’s Theorem. Every relation has a loss-less, fd-preserving 3NF decomposition.
BERNSTEIN’S ALGORITHM

Input: A set of attributes in R and a set F of functional dependencies in R (assume F is already minimal).

Output: A set of 3NF relations that form a loss-less, fd-preserving decomposition of R.

Algorithm:

1. Group together all fd’s which have the same L.H.S. If X → Y₁, X → Y₂, ..., X → Yₖ are all the fd’s with the L.H.S. X, then replace all of them by the single fd X → Y₁Y₂...Yₖ.

2. For each fd X → Y in F, form the relation (XY) in the decomposition.

3. If X’Y’ ⊂ XY, then remove the relation (X’Y’) from the decomposition.

4. If none of the relations in the decomposition contains a candidate key of the original relation, then find a candidate key K of R and add the relation (K) to the decomposition.

Example. This example shows the need for step 1 and 3. Let attributes = ABC and F = {A → B, A → C, C → A}.

Step 1. {A → BC, C → A}.
Step 2. (ABC), (CA)
Step 3. (ABC)
Step 4. (ABC); the key A is contained in ABC.

The decomposition (ABC) = (AB) ⊗ (AC) is not preferred because it causes unnecessary duplication of values for the attribute A. The relation R(ABC) is already in 3NF.
BOYCE-CODD NORMAL FORM (BCNF)

A relation schema $R$ is in BCNF, if for every nontrivial functional dependency $X \rightarrow Y$ that holds on $R$, the attribute $X$ is a superkey.

We often want to decompose a relation $R$ which is not in BCNF into a set of smaller relations $R_1, R_2, ..., R_n$ such that each $R_i$ is in BCNF. Such a decomposition of $R$ is called a BCNF decomposition of the relation $R$.

Consider the relation $R(ABCDE)$ and the fd's $F = \{AB \rightarrow E, AD \rightarrow B, B \rightarrow C, C \rightarrow D\}$. It is not difficult to see that the relation $R$ has 3 candidate keys $AB, AC, AD$. Obviously the relation $R$ is not in BCNF, because we have the fd $B \rightarrow C$ and $C \rightarrow D$ in $R$, but $B$ is not a superkey of $R$, neither is $C$. By using the Heart Theorem, we can obtain an LLJ decomposition of $R$ into three BCNF relations as follows:

\[ R(ABCDE) = (CD) \otimes (BC) \otimes (ABE) \]

The decomposition is given by $R(ABCDE) = (CD) \otimes (BC) \otimes (ABE)$.

In the following, we will give an algorithm to find a loss-less BCNF decomposition of a relation $R$. This tells us that every relation has a loss-less BCNF decomposition.
Algorithm to decompose a relation \( R \) into a set of BCNF relations

Input: Relation \( R \) and set of functional dependencies \( F \) (\( F \) is minimal)
Output: Result=\{ \( R_1 \), \( R_2 \), ..., \( R_n \) \} such that
(A) Each \( R_i \) is in BCNF
(B) The decomposition is loss-less-join decomposition

1. Group together all fd’s which have the same L.H.S. If \( X \to Y_1, X \to Y_2, \ldots, X \to Y_k \) are in \( F \), then replace all of them by a single fd \( X \to Y \), where \( Y=Y_1 Y_2 \cdots Y_k \).
   \[ \text{result} := \{ R \}; \]
   done := FALSE;
   Compute \( F^+ \); // Using the compact representation discussed in page 10

2. \textbf{WHILE} (NOT \ done) \ DO
   \textbf{IF} (there is a scheme \( R_i \) in \ result that is not in BCNF)
   \textbf{THEN BEGIN}
   Let \( X \to Y \) be a nontrivial functional dependency in \( F^+ \) that holds on \( R_i \)
   such that \( X \to R_i \) is not in \( F^+ \);
   \( \text{result} := (\text{result} - \{ R_i \}) \cup \{ (R_i - Y), (XY) \}; \)
   (this means to break \( R_i \) into two relations \( R_i - Y \) and \( XY \))
   \textbf{END}
   \textbf{ELSE done := TRUE ;}

Example. (LLJ decomposition of \( R \) into a set of BCNF relations.) Let \( R = R(ABCDEGH) \) and \( F = \{ B \to E, B \to H, E \to A, E \to D, AH \to C \} \). The decomposition according to the given algorithm is as follows:

(1) After grouping the R.H.S of the fd’s together, we get
   \[ \{ B \to EH, E \to AD, AH \to C \}. \]

(2) The only candidate key of \( R \) is \( BG \). We have the following LLJ decomposition:
It is easy to verify that the four resulting relations are all in BCNF, and

\[ R(ABCDEHG) = (AHC) \otimes (EAD) \otimes (BEH) \otimes (BG). \]

Also we can see that all the functional dependencies in F are preserved.