Chapter 6
Normalization

Conceptual modeling techniques such as E/R chart will only get things started, but they do not provide a way to evaluate the quality of the data models.

We now come back to fill the final missing link: How could we put in place a well structured database?

Question: Which database structure is better?

Answer: The one with less redundancy.

The database normalization theory will help us to evaluate, and improve if needed, the preliminary table structure that we have obtained with, e.g., the ER charts.
The basic approach

The main tool we will use is the functional dependency (FD), which generalizes the key dependency concept.

I heard this stuff, very much related to logic, is being taught in MA 2250 this semester. 😊

The gist is still what you learned in Java: One class talks about one thing only. By the same token, one table holds one relation only.

If a relation is too complicated, we will see how to use FD to decompose such a relation, i.e., break it apart, into a bunch of smaller and/or simpler relations.

Such a process leads to a more natural, i.e., normalized, database.
Remember this one?

Consider the following `Person` table, which is obtained via a direct translation from an E/R diagram, implementing a relationship.

Create Table Person (  
    SSN    Integer,  
    Name   Char(20),  
    Address Char(50),  
    Hobby  Char(10),  
    Primary key (SSN, Hobby)
)

We notice that SSN itself is not a key, a combination of SSN and Hobby is, since the latter potentially could have a multiset as its value.
What’s wrong with it?

Given the following table instance

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>Address</th>
<th>Hobby</th>
</tr>
</thead>
<tbody>
<tr>
<td>1111</td>
<td>John Doe</td>
<td>123 Main St.</td>
<td>Stamps</td>
</tr>
<tr>
<td>1111</td>
<td>John Doe</td>
<td>123 Main St.</td>
<td>Coins</td>
</tr>
<tr>
<td>2222</td>
<td>Mary Doe</td>
<td>7 Lake Dr.</td>
<td>Acting</td>
</tr>
</tbody>
</table>

If John Doe moves to 1 Russell Street, we have to change the address information in both tuples for John. Imagine what could happen if John were to have 123 hobbies.

We either have to update this information 123 times, and if we miss even one, the information will no longer consistent. (Update abnormally)

A reason for this abnormality is that the address information is redundant: we have to repeat it for every combination of key value that shares the same SSN. 😞
Another abnormality

To continue with the story, assume that we want to add Homer Simpson into the table, but he does not have any hobby yet. 😞

What to do? Well, we could place NULL into his hobby field. But since Hobby is part of a primary key, we can’t do that. 😞 (Insertion abnormality)

What’s behind such abnormalities is that this table mixes up two things, what someone likes and where she lives. Because Hobby is multivalued, we have to put it into the key, which causes this much trouble.

On the other hand, where you live has nothing to do with what you enjoy doing. Why do you mix them up?
Yet one more

Just assume we can get away with that. Later on, Homer develops a hobby, say acting.

When we add in this row with acting as his hobby, should the original one with NULL be replaced? We might think so, but the computer has no reason to take it out.

Assume that we delete the original row. Later on, when Homer no longer likes acting, thus we delete this row. We not only delete the acting part, but also all the other information, such as his address, since Hobby is part of the key.

We should only replace acting with NULL. But, a deletion cannot selectively delete part of a tuple. (Deletion abnormality)

**Question:** What to do?

**Answer:** Break it apart so that one table contains one thing.
Decomposition

The solution to the above problem is easy: We decompose the original table into two smaller ones:

Person(SSN, Name, Address)
Hobby(SSN, Hobby)

Since the original relation is just the join of the two derived relations, this decomposition is lossless. In other words, we don’t lose any information by carrying out such a decomposition.

As a result, Hobby, which might be missing, is no longer part of a key; while SSN, which is never missing, is the key of the first relation.

For the second relation, the key consists of both attributes (?). If you don’t have a hobby, you are not in....
What would it look like?

Below shows an instance of this revised schema.

<table>
<thead>
<tr>
<th>SSN</th>
<th>Name</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>111111111</td>
<td>John Doe</td>
<td>123 Main St.</td>
</tr>
<tr>
<td>555666777</td>
<td>Mary Doe</td>
<td>7 Lake Dr.</td>
</tr>
<tr>
<td>987654321</td>
<td>Bart Simpson</td>
<td>Fox 5 TV</td>
</tr>
</tbody>
</table>

(a) PERSON

<table>
<thead>
<tr>
<th>SSN</th>
<th>Hobby</th>
</tr>
</thead>
<tbody>
<tr>
<td>111111111</td>
<td>stamps</td>
</tr>
<tr>
<td>111111111</td>
<td>hiking</td>
</tr>
<tr>
<td>111111111</td>
<td>coins</td>
</tr>
<tr>
<td>555666777</td>
<td>hiking</td>
</tr>
<tr>
<td>555666777</td>
<td>skating</td>
</tr>
<tr>
<td>987654321</td>
<td>acting</td>
</tr>
</tbody>
</table>

(b) HOBBY

Question: How far *could* we go?

Below shows an ultimate, but useless, decomposition. Notice that they are no longer related, thus a lossy decomposition.
What have we learned?

1. Decomposition can serve as a useful tool that augments conceptual modeling by cutting out some of the redundancy.

2. The criteria of choosing the right decomposition is not straightforward, especially when we have to deal with a complicated relation, which is where such a guidance is needed the most.

Thus, we will develop various techniques in this part, based on the idea of functional dependency, that will guide us through the process where we develop various normalized databases.
Functional dependency

We use upper cases of the beginning letters, such as $A, B, C, \ldots$, to represent individual attribute, and use upper cases of the letters near the end of the alphabet with bars over them, such as $\overline{P}, \overline{V}, \overline{Z}, \ldots$ to represent a set of attributes.

We also use, e.g., $ABCD$ to represent set of attributes $\{A, B, C, D\}$; and use, e.g., $\overline{X\overline{Y}Z}$ to represent the union of these sets, i.e., $\overline{X\overline{U}Y\overline{U}Z}$. 
Technically, ...

..., a *functional dependency* (FD) on a relation schema $R$ is a constraint of the form $\overline{X} \rightarrow \overline{Y}$, where both $\overline{X}$ and $\overline{Y}$ are attributes of a relation $R$.

We say that $r$, an instance of $R$, satisfies this dependency if and only if, for every pair of tuples $t$ and $s$ in $r$, if they agree on all attributes in $\overline{X}$, then they also agree on $\overline{Y}$.

Intuitively, given an FD, $\overline{X} \rightarrow \overline{Y}$, to satisfy this FD, for any two rows in a table, $r_1, r_2$, if they have the same values in $\overline{X}$, they must have the same value in $\overline{Y}$.

This is where the word *function* comes from: *Same input leads to the same output*. Again, the same stuff as you learned in Java.

**Homework:** 6.8.
A bit review

The essential requirement imposed by an FD is an implication, $A \rightarrow B$. Conceptually, implication comes with the following truth table.

<table>
<thead>
<tr>
<th>$A$</th>
<th>$B$</th>
<th>$A \rightarrow B$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$F$</td>
<td>$F$</td>
<td>$T$</td>
</tr>
<tr>
<td>$F$</td>
<td>$T$</td>
<td>$T$</td>
</tr>
<tr>
<td>$T$</td>
<td>$F$</td>
<td>$F$</td>
</tr>
<tr>
<td>$T$</td>
<td>$T$</td>
<td>$T$</td>
</tr>
</tbody>
</table>

Thus, the only case an implication fails is when $A$ holds, while $B$ does not. In other words, if $A$ holds, then to ensure the implication constraint holds, $B$ must hold.

Have you heard of this example? “If I have $5, the lunch is on me.”

The only case you can say I lie is when I do have $5, but don’t buy you lunch. 😊
FD examples

The following instance of a table SCP tells us some suppliers, with a home base, supply a certain number of parts.

<table>
<thead>
<tr>
<th>S#</th>
<th>CITY</th>
<th>P#</th>
<th>QTY</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>London</td>
<td>P1</td>
<td>100</td>
</tr>
<tr>
<td>S1</td>
<td>London</td>
<td>P2</td>
<td>100</td>
</tr>
<tr>
<td>S2</td>
<td>Paris</td>
<td>P1</td>
<td>200</td>
</tr>
<tr>
<td>S2</td>
<td>Paris</td>
<td>P2</td>
<td>200</td>
</tr>
<tr>
<td>S3</td>
<td>Paris</td>
<td>P2</td>
<td>300</td>
</tr>
<tr>
<td>S4</td>
<td>London</td>
<td>P2</td>
<td>400</td>
</tr>
<tr>
<td>S4</td>
<td>London</td>
<td>P4</td>
<td>400</td>
</tr>
<tr>
<td>S4</td>
<td>London</td>
<td>P5</td>
<td>400</td>
</tr>
</tbody>
</table>
What could we get?

There are some natural relationships between the attributes. For example, it seems that every $S#$ lives in exactly one city. Thus,

$$S# \rightarrow CITY.$$ 

Moreover, we also have the following FD’s: 1. 
{ $S#,$P# } $\rightarrow$ QTY 
2. { $S#,$P# } $\rightarrow$ CITY 
3. { $S#,$P# } $\rightarrow$ CITY, QTY 
4. { $S#,$P# } $\rightarrow$ S# 
5. { $S#,$P# } $\rightarrow$ { $S#,$P#,CITY,QTY$} 
6. $S# \rightarrow$ QTY 
7. $QTY \rightarrow$ S#

**Question:** What does each FD mean?

The left- and right-hand sides of an FD are sometimes called the *determinant* and the *dependent*, respectively. When such a set consists of just one attribute, we often drop the set braces, e.g., $S# \rightarrow$ CITY.
What do we really want?

We notice that $\text{QTY} \rightarrow \text{CITY}$ holds for the above instance, but it does not have to for another one (?). Remember the difference between data base structure and its instances?

An FD holds for the structure, thus it has to hold for *all of its possible instances*.

For example, a given supplier has to stay in just one place at one time. Thus, any two tuples appearing in a SCP instance with the same supplier number must necessarily have the same city as well. This is characterized with the FD: $\text{S#} \rightarrow \text{CITY}$.

Similarly, a supplier supplies a certain number of a certain part, i.e., $\{\text{S#,P#}\} \rightarrow \text{QTY}$.

From now on, by FD, we mean this more demanding, time-independent sense.
The nature of FD

A functional dependency defined on a relational schema is really an integrity constraint, IC (still remember this stuff?), imposed on a database structure, in the sense that all the instances of that structure have to satisfy such a dependency.

Technically speaking, given a schema \( R(\overline{R}, C) \), where \( \overline{R} \) refers to the attributes, and \( C \) is a collection of FDs, then a legal instance of \( R \) is a relation (table) with the attributes \( \overline{R} \) that satisfies every FD in \( C \).

Let \( r_1, r_2 \) be any two tuples in \( \overline{R} \),

\[
X \rightarrow Y \equiv (\forall r_1, r_2, r_1[X] = r_2[X] \rightarrow r_1[Y] = r_2[Y]).
\]

For example, the previous SCP instance is a legal one with the set of two FDs: \( S\# \rightarrow CITY \) and \( \{S\#, P\# \} \rightarrow QTY \).
Key is a special FD

Recall that $\overline{K}$ is a super key of a relation $R$ means that there cannot be two tuples in $R$ that agree on $\overline{K}$ but disagree on something else. In other words, everything agrees on $\overline{K}$ must agree on the other stuff.

Thus, we can represent this property of $\overline{K}$ as the following FD: $\overline{K} \rightarrow \overline{A}$, where $\overline{A}$ is any collection of attributes of $R$.

Technically, let $r_1, r_2$ be any two tuples in a relation $\overline{R}$, then

$$r_1[\overline{K}] = r_2[\overline{K}] \rightarrow r_1[\overline{A}] = r_2[\overline{A}] .$$

In other words, if they agree on $\overline{K}$ they have to agree on everything else.

**Homework:** 6.3, 6.4, 6.6.
We obviously (?) want to deal with as few FD's as possible. Some of them are just trivial in the sense that they always hold. For example, \( \{S\#,P\#\} \rightarrow \{S\#\} \) always holds (?) In general, an FD is trivial if and only if the right-hand side is a subset of the left-hand side.

In some cases, one FD necessarily implies, or entails, another. Thus if we have the first, the second will be there for free. For example, FD \( \{S\#,P\#\} \rightarrow \{\text{CITY, QTY}\} \) implies the following two FDs: \( \{S\#,P\#\} \rightarrow \text{CITY} \) and \( \{S\#,P\#\} \rightarrow \text{QTY} \). (?) Vice versa

In practice, we are only interested in those non-trivial FDs which are not implied by the others, since they correspond to “genuine” FDs.

For more details, check out §6.4 of the textbook.
How to identify the FDs?

Consider the Person table

Create Table Person (  
    SSN Integer,  
    Name Char(20),  
    Address Char(50),  
    Hobby Char(10),  
    Primary key (SSN, Hobby))

Besides the assumed key dependency:

    SSN, Hobby → SSN, Name, Address, Hobby,

it could have other dependencies among its attributes: Each person has at most one address and a name at a time, i.e.,

    SSN → Name, Address

Also, each person has some hobbies at a given time, thus,

    SSN → Hobby
...and decompose the table by...

... projecting the table in terms of the attributes involved with the FD's.

For example, given the Person table with the following FDs:

\[ \text{SSN} \rightarrow \text{Name, Address} \] and \[ \text{SSN} \rightarrow \text{Hobby} \]

we can decompose the Person table via a projection to the attributes as associated with these two FDs, which leads to the following two smaller and simpler tables:

Person(SSN, Name, Address)

Hobby(SSN, Hobby)

Those two tables are lossless decomposition of the original one, and moreover, they contain no redundancy. We will see why a bit later.

**Question:** Is it always this easy?

**Answer:** No!
The bad guy....

Remember this piece? We saw it in the conceptual modeling chapter.

This example will give us something to think about...
The first draft

Consider the following HasAccount table,

```
Create Table HasAccount ( 
    AccountNumber Integer Not Null, 
    ClientId Char(20), 
    OfficeId Integer, 
    Primary key (ClientId, OfficeId), 
    Foreign key (OfficeId) references Office, 
    Foreign key (ClientId) references Client 
 )
```

One integrity constraint is that every account can be assigned to only one office, thus

```
AccountNumber → OfficeId
```

Since a client can have a few accounts, thus, ClientId is not a key by itself.

Moreover, OfficeId itself is not a key, either, since multiple accounts might be held there.
A headache

Indeed, the primary key of this table consists of two attributes, ClientId, OfficeId, *reflecting the constraint that one client has only one account in any office.*

\[
\text{ClientId, OfficeId} \rightarrow \text{AccountNumber}
\]

<table>
<thead>
<tr>
<th>AcctNumber</th>
<th>ClientId</th>
<th>OfficeId</th>
</tr>
</thead>
<tbody>
<tr>
<td>A057</td>
<td>1111</td>
<td>BS32</td>
</tr>
<tr>
<td>A908</td>
<td>1234</td>
<td>MN08</td>
</tr>
<tr>
<td>B123</td>
<td>1111</td>
<td>SB01</td>
</tr>
<tr>
<td>B123</td>
<td>2222</td>
<td>SB01</td>
</tr>
<tr>
<td>B321</td>
<td>3333</td>
<td>SB01</td>
</tr>
</tbody>
</table>

Assume we have to change the office affiliation of an account, e.g., B123, since we have a joint key, we can expect that multiple rows with the same OfficeId, SB01 in this case, but different ClientId, 1111 and 2222.

**Question:** What happens if an account is jointly owned by 155 people?
What do we have to do?

We have to locate all the rows showing different owners of this account, and change the associated office ID.

In this case, we have two FDs, where the determinant of one FD, \( \text{AccountNumber} \),

\[ \text{AccountNumber} \rightarrow \text{OfficeId}, \]

is also a dependent of in another FD associated with the same schema:

\[ \text{ClientId}, \text{OfficeId} \rightarrow \text{AccountNumber}. \]

Thus, the same account number, e.g., B123, leads to the same OfficeId, e.g., SB01 which could be associated with multiple ClientId values, e.g., 1111 and 2222.

**Question**: What have we got?

**Answer**: Redundancy. 😊
Normal forms

To eliminate redundancy and potential update related abnormalities, we have identified several normal forms for relational schemas such that if a schema is in one of these forms, it has certain predictable properties.

The first normal form (1NF) is equivalent to the definition of the relational data model, saying that the value of an attribute must be atomic (Every RDB table is a set.)

The second normal form (2NF) states that a schema must not have an FD $X \rightarrow Y$ such that $X$ is a strict subset of the key of that schema.

The third normal form (3NF) is thought to be ultimate, but Boyce and Codd came up with the BCNF to cut off more redundancy.
Why is it a good guy?

Every RDB is in 1NF. The rest is up in the air.

Given Person(SSN, Name, Address, Hobby), it is not in 2NF, since it has an FD,

$$SSN \rightarrow Hobby,$$

where SSN is part of its key: \{SSN, Hobby\}.

On the other hand, once we split it into two tables,

Person(SSN, Name, Address)

with FD: SSN $\rightarrow$ Name, Address, and,

Hobby(SSN, Hobby)

with empty FD, then they both belong to the 2NF, indeed 3NF and BCNF, as we will see.
One step at a time

BCNF is more desirable, but not always achievable without paying a price. More normal forms have been defined such as fourth normal form and fifth normal form, which further cut down more redundancy. They essentially put more and more restrictions over the previous layers.

The following picture shows their relationship:

Sky is the limit....
The BCNF

A relation schema \( S(\overline{R}, \mathcal{F}) \) is in a Boyce-Codd normal form (BCNF) if, for every FD \( \overline{X} \rightarrow \overline{Y} \in \mathcal{F} \), either 1) it is trivial, i.e., \( \overline{Y} \subseteq \overline{X} \); or 2) \( \overline{X} \) is a superkey of \( R \).

In other words, the only nontrivial FDs are those in which a key functionally determines one or more attributes.

The schema \((\text{Person1}(\text{SSN, Name, Address}), \text{SSN } \rightarrow \text{Name, Address})\) is in BCNF since SSN, the left-hand side in the only FD is indeed a key, thus a superkey for that table. The schema \((\text{Hobby}(\text{SSN, Hobby}), \emptyset)\) is also in BCNF, since there is nothing for us to check.

But \( \text{Person}(\text{SSN, Name, Address, Hobby}) \) is not in BCNF, since one of the FDs, \( \text{SSN } \rightarrow \text{Name, Address} \), violates the BCNF condition.
A couple of points

1. A RDB can have more than one candidate keys. For example, $R(ABCD, \mathcal{F})$, where $\mathcal{F} = \{AB \rightarrow CD, AC \rightarrow BD\}$, has two candidate keys, $AB$ and $AC$. Then, $R$ is still in BCNF, since the determinant of both FDs are different superkeys.

2. Redundancy arises when the values of some non-key attribute(s) $X$ necessarily implies the values of other attribute(s) $A$, because of a FD $X \rightarrow A$.

For two tuples in a table instance, if they agree on $X$, they must agree on $A$. Since no duplicates exist in the table, when $X$ is not a super key, there could be multiple tuples where the same value of $X$, and $A$, occurs multiple times. This clearly leads to redundancy.

Thus, A table not in BCNF is “bad”.
An example

With \( SSN \rightarrow Name \), any two tuples that agree on \( SSN \) must agree on the \( Name \) part. For example, the first and the second tuple agree on 1111 and also share the same names, and addresses. On the other hand, they differ in \( Hobby \).

<table>
<thead>
<tr>
<th>SSN</th>
<th>Name</th>
<th>Address</th>
<th>Hobby</th>
</tr>
</thead>
<tbody>
<tr>
<td>1111</td>
<td>John Doe</td>
<td>123 Main St.</td>
<td>Stamps</td>
</tr>
<tr>
<td>1111</td>
<td>John Doe</td>
<td>123 Main St.</td>
<td>Coins</td>
</tr>
<tr>
<td>2222</td>
<td>Mary Doe</td>
<td>7 Lake Dr.</td>
<td>Acting</td>
</tr>
</tbody>
</table>

Notice that \( SSN \) is not a super key so it can occur multiple times.

In this case, the first two tuples must be different, while the same \( SSN \), leading to the same \( Name \), guarantee redundancy.
What does BCNF do?

BCNF states that, for any FD $X \rightarrow Y$, $X$ has to be a superkey.

Recall that any key value could occur at most once in a table instance, which requires for each value of $X$ that shows up in a table instance, only one value of $A$ be kept. Thus, this BCNF requirement cuts out redundancy.

For example, in a decomposed table, we will have only one copy of Name for each distinct value of SSN, since the latter is a key.

<table>
<thead>
<tr>
<th>SSN</th>
<th>Name</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>1111</td>
<td>John Doe</td>
<td>123 Main St.</td>
</tr>
<tr>
<td>2222</td>
<td>Mary Doe</td>
<td>7 Lake Dr.</td>
</tr>
</tbody>
</table>


3. Duplicated data does not always mean redundancy. Given the following value of the aforementioned table $R$ and an instance

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

It seems that it contains redundancy since the two tuples agree on $B, C$ and $D$.

This schema would contain redundancy if $\{A, B\}$ is a key, and $B \rightarrow C, D$ is a FD. Then, it is not in BCNF (?). It is not even in 2NF. (?)

On the other hand, if we don’t put a FD there, it is in BCNF since there is nothing to check.

Thus, redundancy depends on both factors of a schema: data and the associated FD set.

That’s why more normal forms are there beyond BCNF.
4. A relation instance that is in a BCNF schema does not store redundant information. As a result, deletion and update abnormalities won’t happen in BCNF tables.

Update abnormality happens when we have redundant information about certain relationship as expressed by an FD, $X \rightarrow Y$, where for the same value of $X$, multiple values of $Y$ occur.

Thus, we either have to update all these identical values of $Y$ or run the risk of data inconsistency.

This won’t happen in BCNF tables since there will be only one copy of the same $Y$ value.
Deletion abnormality happens in a table when in an FD $x \rightarrow y$, $x$ is part of the key. When the other part of the key is deleted, so are $x$ and all the associated data. 😞

For example, in the following table, if we delete the hobby of Mary, the whole tuple has to be deleted, since $\text{Hobby}$ is part of its key.

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>Address</th>
<th>Hobby</th>
</tr>
</thead>
<tbody>
<tr>
<td>1111</td>
<td>John Doe</td>
<td>123 Main St.</td>
<td>Stamps</td>
</tr>
<tr>
<td>1111</td>
<td>John Doe</td>
<td>123 Main St.</td>
<td>Coins</td>
</tr>
<tr>
<td>2222</td>
<td>Mary Doe</td>
<td>7 Lake Dr.</td>
<td>Acting</td>
</tr>
</tbody>
</table>

With a BCNF table, it won’t be an issue. If we delete, e.g., ID, the whole tuple is gone.

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>1111</td>
<td>John Doe</td>
<td>123 Main St.</td>
</tr>
<tr>
<td>2222</td>
<td>Mary Doe</td>
<td>7 Lake Dr.</td>
</tr>
</tbody>
</table>
5. BCNF relations with more than one keys still can have insertion problems. For example, in the following instance of $R = (ABCD, \mathcal{F})$, where $\mathcal{F} = \{AB \rightarrow CD, AC \rightarrow BD\}$, and both $AB$ and $AC$ are candidate keys.

$$
\begin{array}{|c|c|c|c|}
\hline
A & B & C & D \\
\hline
1 & 1 & 3 & 4 \\
2 & 1 & 3 & 4 \\
3 & 4 & \text{Null} & 5 \\
3 & \text{Null} & 2 & 5 \\
\hline
\end{array}
$$

where we add in two nulls since we don’t know their real values at the time of entry, what happens if the $C$ value of the third becomes 5, while the $B$ value of the fourth becomes 4?

In practice, such a situation will not happen since we will designate one of them as the primary key, which can’t have a null value.
Third normal form

A relation schema \( S = (R, F) \) is in the third normal form if it is in 2NF and, for every FD \( X \rightarrow Y \in F \), either 1) it is trivial, i.e., \( Y \subseteq X \); or 2) \( X \) is a superkey of a table \( R \); or 3) each attribute in \( A \in Y - X \) belongs to some candidate key \( K \) of a table \( R \).

It is clear that BCNF is a special case of 3NF (°). Thus, every schema in BCNF is automatically in 3NF, while the other direction does not need to be true.

The schema \( \text{HasAccount(AccountNumber, ClientId, OfficeId)} \), with its primary key being \( \{\text{ClientId, OfficeId}\} \), and the FD \( \text{AccountNumber} \rightarrow \text{OfficeId} \) is not in BCNF, since \( \text{AccountNumber} \) is not a superkey.

However, since \( \text{OfficeId} \) is part of the key, this schema is in 3NF.

**Homework:** 6.2
3NF could be redundant

We saw earlier that HasAccount could contain redundant information, which is caused by the FD AccountNumber → OfficeId.

In an instance of this table, it could be the case that for the same AccountNumber value, which requires the same OfficeId value (?), multiple ClientId values occur in that instance.

<table>
<thead>
<tr>
<th>AcctNumber</th>
<th>ClientId</th>
<th>OfficeId</th>
</tr>
</thead>
<tbody>
<tr>
<td>A057</td>
<td>1111</td>
<td>BS32</td>
</tr>
<tr>
<td>A908</td>
<td>1234</td>
<td>MN08</td>
</tr>
<tr>
<td>B123</td>
<td>1111</td>
<td>SB01</td>
</tr>
<tr>
<td>B123</td>
<td>2222</td>
<td>SB01</td>
</tr>
<tr>
<td>B123</td>
<td>3333</td>
<td>SB01</td>
</tr>
</tbody>
</table>

Definitely a tissue moment.... 😔

*I have a tissue for your issue.* 😊
What to do?

Since there is no redundancy for schemas in BCNF and less redundancy for schema in 3NF, we are interested in decomposing a schema into a bunch of schemas such that each of which is in one of these normal forms.

We first try to look at some of the properties of such decomposition process. We are mainly interested in lossless decomposition, which does not lose information during this process.

Recall that a decomposition is lossless if we join together the decomposed tables, we will get the original table back (Cf. Page 20 of this set of notes).
How do I know...?

There exists a simple and effective way to test whether a binary decomposition is lossless, which can be generalized if we get a decomposition via a sequence of binary decomposition.

Let \( S(R, F) \) be a relation schema and let \( S_1(R_1, F_1), S_2(R_2, F_2) \) be a binary decomposition of \( S \). It is lossless if and only if either i) \( R_1 \cap R_2 \rightarrow R_1 \in F^+ \); or ii) \( R_1 \cap R_2 \rightarrow R_2 \in F^+ \).

Here, \( F^+ \), the closure of \( F \) is the collection of all the FDs implied by \( F \). (See §6.4 of the textbook)

In other words, the above binary decomposition is lossless if and only if either \( R_1 \cap R_2 \rightarrow R_1 \) or \( R_1 \cap R_2 \rightarrow R_2 \) follows from \( F \).
An example

Consider the following schema:

HasAccount(AccountNumber, ClientId, OfficeId) with the following FDs:

ClientId,OfficeId → AccountNumber
AccountNumber → OfficeId

Then the following decomposition is lossless and also in BCNF (?):

AcctOffice: (AccountNumber, OfficeId), with FD AccountNumber → OfficeId

AcctClient: (AccountNumber, ClientId) with no FD.

Since AccountNumber is the common attribute of the two decomposed schemas, and the following is one of the original FDs.

AccountNumber → OfficeId
What is missing?

Although the above decomposition is lossless, something is wrong: the FD ClientId,OfficeId → AccountNumber is homeless.

*Keep in your mind that, no matter how you implement your database, you have to stick to all the FDs.*

Notice that the FD AccountNumber → OfficeId can be checked locally via one of the schemas, verification of the missing FD has to be done via an expensive join of two tables. (Yuk!)

Thus, this decomposition leads to an increased cost of maintaining an integrity constraint as characterized by the above FD.

In such a case, we say that the decomposition, although lossless, does not *preserve the dependency in the original schema.*
An example

Here is the original instance of HasAccount:

<table>
<thead>
<tr>
<th>AcctNumber</th>
<th>ClientId</th>
<th>OfficeId</th>
</tr>
</thead>
<tbody>
<tr>
<td>B123</td>
<td>1111</td>
<td>SB01</td>
</tr>
<tr>
<td>A908</td>
<td>1234</td>
<td>MN08</td>
</tr>
</tbody>
</table>

After the decomposition, here is the AcctOffice instance:

<table>
<thead>
<tr>
<th>AcctNumber</th>
<th>OfficeId</th>
</tr>
</thead>
<tbody>
<tr>
<td>B123</td>
<td>SB01</td>
</tr>
<tr>
<td>A908</td>
<td>MN08</td>
</tr>
</tbody>
</table>

and the AcctClient instance:

<table>
<thead>
<tr>
<th>AcctNumber</th>
<th>ClientId</th>
</tr>
</thead>
<tbody>
<tr>
<td>B123</td>
<td>1111</td>
</tr>
<tr>
<td>A908</td>
<td>1234</td>
</tr>
</tbody>
</table>
What could go wrong?

Now, we add in \((B567, SB01)\) and \((B567, 1111)\) to the tables, respectively.

Those newly added tuples still satisfy all the local dependencies associated with this decomposition:

<table>
<thead>
<tr>
<th>AcctNumber</th>
<th>OfficeId</th>
</tr>
</thead>
<tbody>
<tr>
<td>B123</td>
<td>SB01</td>
</tr>
<tr>
<td>B567</td>
<td>SB01</td>
</tr>
<tr>
<td>A908</td>
<td>MN08</td>
</tr>
</tbody>
</table>

and the one for AcctClient

<table>
<thead>
<tr>
<th>AcctNumber</th>
<th>ClientId</th>
</tr>
</thead>
<tbody>
<tr>
<td>B123</td>
<td>1111</td>
</tr>
<tr>
<td>B567</td>
<td>1111</td>
</tr>
<tr>
<td>A908</td>
<td>1234</td>
</tr>
</tbody>
</table>

But, they actually violate the FD \(ClientId, OfficeId \rightarrow AccountNumber,\ldots\) 😞
A bit too late

..., which, in real life, we will only be able to find out after joining back these two tables.

<table>
<thead>
<tr>
<th>AcctNumber</th>
<th>ClientId</th>
<th>OfficeId</th>
</tr>
</thead>
<tbody>
<tr>
<td>B123</td>
<td>1111</td>
<td>SB01</td>
</tr>
<tr>
<td>B567</td>
<td>1111</td>
<td>SB01</td>
</tr>
<tr>
<td>A908</td>
<td>1234</td>
<td>MN08</td>
</tr>
</tbody>
</table>

This is what we meant by saying we have to make extra effort to maintain the integrity of desired constraints when a decomposition does not preserve some of the original dependencies.

**Question:** What to do, besides pulling out tissues?
What do you want?

We have so far discussed two important features of decomposition: losslessness and dependency preservation, as well as the goal of such a decomposition: get rid of redundancy.

We have also seen a BCNF decomposition that is lossless, redundancy free, but not dependency preserving.

Unfortunately, there is no BCNF for this example that honors both properties.

**Question:** Which property is more important?

**Answer:** losslessness is mandatory and dependency preserving is optional.

Now we are ready to get into the decomposition business.
BCNF Decomposition

Let $S_0 = (R, F)$ be a relation schema, the following algorithm constructs a decomposition of $S$ by repeatedly splitting $R$ into smaller subschemas. At each step the new database schema has strictly fewer FDs that violate BCNF than the one in the previous iteration. When the algorithm terminates, all schemas in the result are in BCNF.

1. $D = (S_0, F_0)$
2. While $S = (\overline{S}; F') \in D$ is not in BCNF do
3. Let $X \rightarrow Y \in F' +$ such that $XY \subseteq S$ and it violates BCNF in $S$
4. Replace $S$ with $S_1 = (\overline{X}Y; F'_1)$ and $S_2 = ((S - \overline{Y}) \cup \overline{X}; F'_2)$, where $F'_1 = \pi_{\overline{XY}}(F')$, $F'_2 = \pi_{(S - \overline{Y}) \cup \overline{X}}(F')$.

Both $F'_1$ and $F'_2$ must be implied by $F' +$.

It can be shown this decomposition is always lossless.
An example

Consider the HasAccount(AccountNumber, ClientId, OfficeId) and the FD: \{ ClientId, OfficeId → AccountNumber, AccountNumber → OfficeId.\}

It is not in BCNF since AccountNumber in the second FD is not a super key of the relation. We can thus use this FD as a guidance to split the relation.

Letting X be AccountNumber, and Y be OfficeId, the While loop will come up with two sub-schemas:

\[ S_1 = (\{ \text{AccountNumber, OfficeId} \}; \{ \text{AccountNumber → OfficeId} \}) \text{ and } S_2 = (\{ \text{ClientId, AccountNumber} \}); \emptyset). \]

Notice that, as we discussed earlier, this decomposition is lossless, in BCNF, but one of the FDs is not preserved.
Via 3NF

If we settle for 3NF, it is always possible to come up with a decomposition that preserves dependency, although as we already saw, 3NF could contain redundancy.

A detailed process for generating a 3NF table is given in §6.8, which leads to decomposition both lossless and dependency preserving.

If the result of this procedure is also in BCNF, we are done. Otherwise, we can further apply the BCNF procedure to it until all of the components are in BCNF.

The point is that if there is a lossless and dependency preserving decomposition, the 3NF procedure will find it. On the other hand, if some components are not in BCNF after the 3NF process, loss of some FD will be inevitable.
A bit summary

1. 3NF schemas might have redundancy. There exists an algorithm that generates 3NF schemas that are both lossless and dependency preserving.

2. BCNF schemas do not have redundancy if we only consider the FDs. We went through an algorithm that generates BCNF schemas that are lossless but might not be dependency preserving.

3. There are other normal forms, such as 4NF, which lead to even less redundancy, such as multi-valued redundancy as we saw that might exist in BCNF schemas (Cf. Page 35).

4. There are lot more to this regard, 5NF and beyond in the box. Check out the book if you want to dig more out, while waiting for the smelling Turkey in the oven. 😊