

Chapter 6

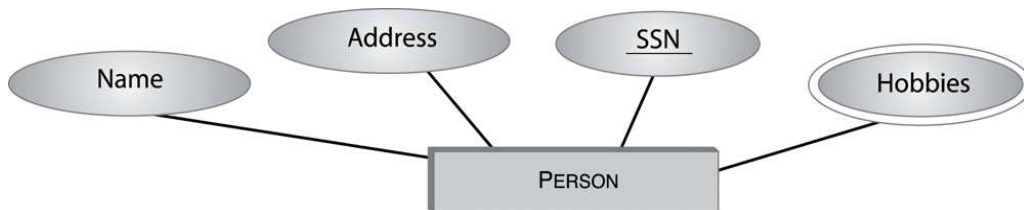
Normalization

Conceptual modeling techniques such as E/R modeling will only get things started, but they do not provide a way to evaluate the quality of the data models. In this chapter, we will present the *database normalization theory*, which will help us to evaluate, and improve, the tables that we have obtained in the conceptual modeling stage.

The main tool we will use is the *functional dependency*, which generalizes the key dependency concept that we used in E/R modeling. This concept will help us to spot unnatural situation where *attributes of unrelated relations are placed in one relation*. We can then break apart such a relation into a bunch of smaller and/or simpler relations which leads to a more natural database.

An example

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



```
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. This will cause a few problems.

What's wrong?

Given the following table

ID	Name	Address	Hobby
1111	John Doe	123 Main St.	Stamps
1111	John Doe	123 Main St.	Coins
2222	Mary Doe	7 Lake Dr.	Acting

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

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

A reason for this abnormality is that the address information is redundantly kept: we have to repeat it for every combination of key value. But, where you live only depends on SSN, but not what you enjoy doing.

Another abnormality

Assume that we want to add Homer Simpson into the table, but he does not have any hobby yet. One way to do it is to place `null` into the hobby field for him, but since `Hobby` is part of a primary key, we can't do that. (Insertion abnormality)

The reason for this issue is again we somehow mix two things together: What you like has nothing to do with where you live.

This table talks about two things: where you live, which depends only on your `SSN`; and what you like to do, also should only depend on your `SSN`. When we mix them up, because `Hobby` is multi-valued, we have to put it into the key, which causes much trouble.

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, 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 can also get away with this issue, when Homer no longer likes acting, and 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. Thus, we should only replace acting with `NULL`. But, this can't be done. (Deletion abnormality)

Question: Can you figure out what is going on and why yourself?

A clear sign for all such issues is that we keep too much information in one table.

Question: What to do?

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)

It is easy to see that the original relation is just the join of the two derived relations. Thus, we say that 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 both relations. It is easy to check that all the redundancy related issues are now gone.

What would it look like?

Below shows an instance of this revised schema.

SSN	Name	Address
111111111	John Doe	123 Main St.
555666777	Mary Doe	7 Lake Dr.
987654321	Bart Simpson	Fox 5 TV

(a) PERSON1

SSN	Hobby
111111111	stamps
111111111	hiking
111111111	coins
555666777	hiking
555666777	skating
987654321	acting

(b) HOBBY

Question: How far *could* we go?

Below shows an ultimate, but useless, decomposition. Notice that they are no longer *related*.

SSN
111111111
555666777
987654321

Name
John Doe
Mary Doe
Bart Simpson

Address
123 Main St.
7 Lake Dr.
Fox 5 TV

Hobby
stamps
hiking
coins
skating
acting

What have we learned?

1. Decomposition can serve as a useful tool that compliments 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 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 to develop various *normal forms*.

Functional dependency

We use upper cases of the beginning letters, such as A, B, C, \dots , to represent individual attribute, and use upper cases of the letters near the end of the alphabet with bars over them, such as $\bar{P}, \bar{V}, \bar{Z}, \dots$ 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{XYZ} to represent the union of these sets, i.e., $\bar{X} \cup \bar{Y} \cup \bar{Z}$.

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 R .

We say that r , an instance of R , satisfies this dependency iff 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.

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.

A	B	$A \rightarrow B$
F	F	T
F	T	T
T	F	F
T	T	T

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.

For example, when I say “If I have \$5, the lunch is on me.” the only case you can say I lie is when I do have \$5, but I don’t buy you lunch.

FD examples

Given the following value of a table SCP, telling us some suppliers with a home base supply a certain number of parts.

<u>S#</u>	CITY	P#	QTY
S1	London	P1	100
S1	London	P2	100
S2	Paris	P1	200
S2	Paris	P2	200
S3	Paris	P2	300
S4	London	P2	400
S4	London	P4	400
S4	London	P5	400

There are some natural relations between the attributes. For example, it seems that every S# lives in exactly one city. Thus, $\{S\# \} \rightarrow \{CITY\}$.

Moreover, SCP also satisfies several other FDs:

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. $\{QTY \} \rightarrow \{S\# \}$

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 $QTY \rightarrow CITY$ holds for the above instance, but need not for another one.

What we are really interested is the general case: A set of FDs not only hold for some instance of a table, but for *all the possible instances* that table could have.

For example, $S\# \rightarrow CITY$ holds for all possible values of SCP , since at any given time, a given supplier has precisely one corresponding city, thus, any two tuples appearing in SCP at the same time with the same supplier number must necessarily have the same city as well.

From now on, by FD, we mean this more demanding, time-independent sense. Below are some examples:

$$\begin{aligned} \{S\#, P\# \} &\rightarrow \{QTY\} \\ \{S\# \} &\rightarrow \{CITY\} \end{aligned}$$

The nature of FD

A functional dependency defined on a relational schema is really an integrity constraint imposed on all its instances, in the sense that all the instances of that schema has to satisfy such a dependency.

We already mentioned that some of the instances of a table might satisfy certain relationship which is not intended as a FD for the schema. For example, we notice that in the SCP table, every S# determines a unique quantity, on the other hand, but $\{S\# \} \rightarrow \{QTY\}$ is certainly not a natural FD for SCP.

Given a schema $R = (\bar{R}, C)$, where \bar{R} is the attributes of R , and C is a collection of FDs, then a *legal instance of R* is a relation (table) with the attributes \bar{R} that satisfies every FD in C .

Key is a special FD

Note that the key constraint is a special case of the functional dependency: Everything agrees on the key attribute must agree on the other stuff by definition.

A bit more formally, if $key(\overline{K})$ is a key constraint of R and r is an instance with R . By definition, r satisfy $key(\overline{K})$ iff there is no pair of distinct tuples $t, s \in r$, such that they agree on \overline{K} . In other words, if they agree on \overline{K} they have to agree on everything else.

Thus, we can represent this property of \overline{K} as follows:

$$\overline{K} \rightarrow \overline{A},$$

where \overline{A} is any collection of attributes of R .

Homework: 6.3, 6.4, 6.6.

FD implications

In some cases, one FD necessarily implies, or entails, another. Thus if we guarantee the first, the second will be there for free.

For example, FD $\{S\#,P\# \} \rightarrow \{CITY, QTY\}$ *implies* the following two FDs: $\{S\#,P\# \} \rightarrow CITY$ and $\{S\#,P\# \} \rightarrow QTY$. (?)

More generally, let f_1 be $X \rightarrow \{ Y, Z\}$. Then, by definition, for any X values, x_1, x_2 ; Y values, y_1, y_2 ; and Z values, z_1, z_2 ; because of f_1 , that $x_1 = x_2 \implies (y_1, z_1) = (y_2, z_2) \Leftrightarrow y_1 = y_2$ and $z_1 = z_2$.

In other words, $x_1 = x_2 \implies y_1 = y_2$ and $x_1 = x_2 \implies z_1 = z_2$.

Thus, f_1 implies the following two FDs: $X \rightarrow Y$, and $X \rightarrow Z$.

Trivial dependencies

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

In practice, we are only interested in those non-trivial ones, since they correspond to “genuine” integrity constraints.

We can use such ideas to come up with the *closure* of an FD set, i.e., all FDs implied by this FD set.

This idea of implication is very useful. For example, it can be used to identify a candidate key of a relation.

Candidate key identification

Let $R = (\overline{R}, \mathcal{F})$ be a database schema. Assume that we know how to check if \mathcal{F} implies any arbitrary FD (Cf. pp. 204 of the textbook), then we can find a candidate key for \overline{R} as follow:

Pick up any attribute $A \in \overline{R}$, and check if \mathcal{F} implies the FD $(\overline{R} - A) \rightarrow \overline{R}$.

If it is, it means that $\overline{R} - A$ still has the uniqueness property, thus, a super key. We will then repeat the above process, replacing R with $R - A$, until the above property fails for every attribute A in \overline{R} .

We now know that \overline{R} has the uniqueness property, but no subset of \overline{R} does. Thus, \overline{R} is the minimal set of attributes satisfying the uniqueness property, i.e., a candidate key.

Identify the FDs...

Consider the table Person

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

Besides the assumed dependency: $SSN, Hobby \rightarrow SSN, Name, Address, Hobby$, it could have other dependencies among its attributes.

For example, each person has at most one address and a name, i.e., $SSN \rightarrow Name, Address$.

Also, a person's hobby has nothing to do with where he lives, thus, $SSN \rightarrow Hobby$.

...and decompose the table

Once we have identified such FD's, we can decompose the table by projecting it with the FD's.

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

`SSN → Name, Address` and `SSN → Hobby`;

we can decompose the `Person` table to the following two smaller tables:

`Person(SSN, Name, Address)`

`Hobby(SSN, Hobby)`

Those two tables are lossless decomposition of the original one, and moreover, they contain no redundancy.

Another example

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  
)
```

Every account can be assigned to only one office, thus

$\text{AccountNumber} \rightarrow \text{OfficeId}$

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

Moreover, multiple clients might jointly share the same account, leading to the same office, via the above FD. Thus, `OfficeId` itself is not a key, either.

A headache

This table has a key consisting of two attributes:

$\text{ClientId, OfficeId} \rightarrow \text{AccountNumber}$.

AcctNumber	ClientId	OfficeId
A057	1111	BS32
A908	1234	MN08
B123	1111	SB01
B123	2222	SB01
B321	3333	SB01

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.

Thus, we have to locate all the rows showing different owners of this account, and change the associated office ID. In this case, we have a dependency whose determinant, `AccountNumber`, is also a dependent of another dependency.

Normal forms

To eliminate redundancy and potential update 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.

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 one, but Boyce and Codd came up with the *BCNF* to cut away more redundancy.

What does that mean?

Every RDB is in 1NF. The rest is up in the air. For example, given the table `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, with `SSN` being the key,

`Person(SSN, Name, Address)`

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

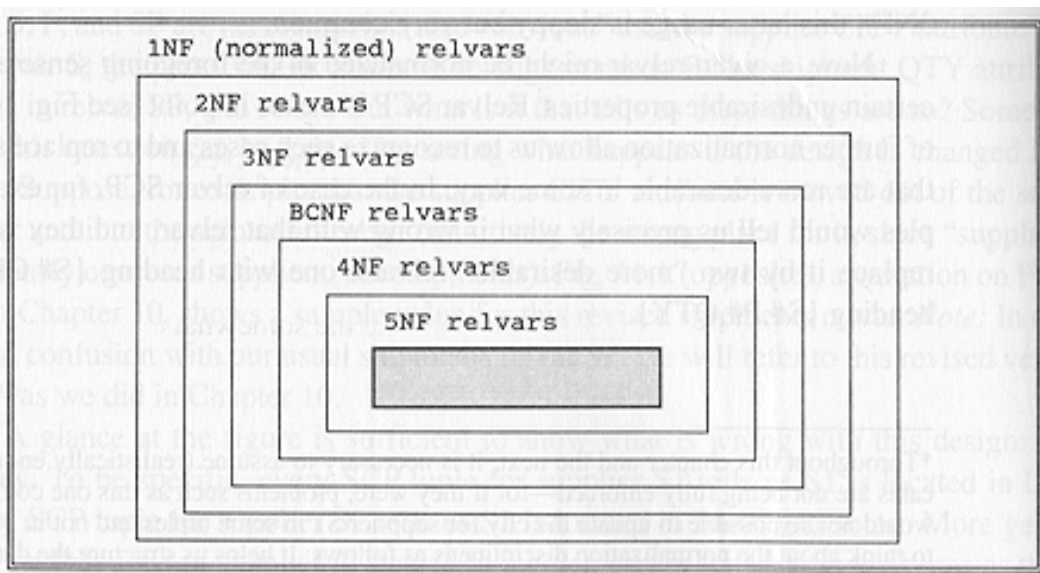
`Hobby(SSN, Hobby)`

with FD $SSN \rightarrow Hobby$, then they both belong to the 2NF, indeed 3NF and BCNF.

Normal forms

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:



The BCNF

A relation schema $S = (\overline{R}, \mathcal{F})$ is in a Boyce-Codd normal form if for every FD $\overline{X} \rightarrow \overline{Y} \in \mathcal{F}$, either of the following is true 1) $\overline{Y} \subseteq \overline{X}$; or \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 (Person1(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 superkey for that table. The schema (Hobby(SSN, Hobby), \emptyset) is also in BCNF, since there is nothing to check out.

If we consider the original table Person(SSN, Name, Address, Hobby), then one of the FDs $\text{SSN} \rightarrow \text{Name, Address}$ violates the BCNF condition, since SSN is not a superkey of this schema.

A couple of points

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

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

For example, since $SSN \rightarrow Name$, the information on $Name$ is redundant in the following structure.

SSN	Name	Address	Hobby
1111	John Doe	123 Main St.	Stamps
1111	John Doe	123 Main St.	Coins
2222	Mary Doe	7 Lake Dr.	Acting

BCNF states that if this is the case, then X has to be a key, which requires for each value of X , only one value of A will be kept. For example, in a decomposed table, we will have only one copy of the address for each distinct value of SSN.

SSN	Name	Address
1111	John Doe	123 Main St.
2222	Mary Doe	7 Lake Dr.

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

A	B	C	D
1	1	3	4
2	1	3	4

It seems that it contains redundancy since the two tuples only differs on A . But, since there is no FD over the attribute set of B, C and D , there can't be any redundancy *from a technical perspective*.

The fact that the two tuples contain duplicate data is just a coincidence.

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

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. This won't happen in BCNF tables since for any such FD, X must be a super key, thus occurring only once in the whole table, together with any related data.

Deletion abnormality happens in a table when in an FD $X \rightarrow Y$, X is part of the key. Thus, when the other part of the key is deleted, so is X and all the associated data. This won't happen in BCNF tables either, since if there is such an FD, X must be the whole thing.

5. BCNF relations with more than one keys still can have insertion problems. For example, if we add in two more associations to the above table to derive the following

A	B	C	D
1	1	3	4
2	1	3	4
3	4	Null	5
3	Null	2	5

where we add in two `nulls` since we don't know their real values at the time of entry.

Question: What happens if the C value of the third becomes 5, while the B value of the fourth becomes 4? (Recall that AB is the key.)

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 $S = (\overline{R}, \mathcal{F})$ is in the third normal form if it is in 2NF and, for every FD $\overline{X} \rightarrow \overline{Y} \in \mathcal{F}$, either of the following is true 1) $\overline{Y} \subseteq \overline{X}$; or \overline{X} is a superkey of R ; 2) each attribute in $A \in \overline{Y} - \overline{X}$ belongs to some candidate key \overline{K} of 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 `HasAccount(AccountNumber, ClientID, OfficeID)`, with its primary key being `{ClientID, OfficeID}`, and the FD `AccountNumber → OfficeID` is not in BCNF, since `AccountNumber` is not a superkey.

However, since `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. Now, it is clear that this is caused by the FD

`AccountNumber` \rightarrow `OfficeID`,

which is not implied by a key constraint. Thus, in an instance of this table, it could be the case that for the same value of `AccountNumber`, and the same value of `OfficeID` multiple values of `ClientID` occur.

Why is it the case?

The latter could happen since `OfficeID` itself is only part of the key, thus the same value of `OfficeID` occurs with different values of the other part of the key, i.e., `ClientID`.

AcctNumber	ClientId	OfficeId
A057	1111	BS32
A908	1234	MN08
B123	1111	SB01
B123	2222	SB01
B123	3333	SB01

Then, the relationship between `AcctNumber` and `OfficeID` has to be repeated, which is redundant.

Another example

The schema `Person(SSN, Name, Address, Hobby)`, with FDs `SSN → Name`, is not in the BCNF, since `SSN` is not a superkey.

This schema also violates the 3NF requirements since `Name` is not part of a candidate key.

It is not even in 2NF since `SSN` is a proper subset of the key attribute set.

It is in 1NF since none of its attributes has a set as their values.

A plan

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.

This essentially says that if we join together the decomposed tables, we will get the original table back.

Decomposition

A *decomposition* of a schema $S = (\overline{R}, \mathcal{F})$ is a collection of schemas such

$$S_1 = (\overline{R}_1, \mathcal{F}_1), \dots, S_n = (\overline{R}_n, \mathcal{F}_n)$$

satisfying the following conditions: 1) for all $i \neq j$, $S_i \neq S_j$; 2) $\overline{R} = \cup_{i=1}^n \overline{R}_i$; and 3) for all $i \in [1, n]$, \mathcal{F} implies \mathcal{F}_i , i.e., \mathcal{F} implies all the \mathcal{F}_i 's.

This condition essentially requires that no attributes be either introduced or dropped during the process; and no FDs be added, but maybe dropped.

The decomposition of a schema naturally leads to the decomposition of all the instances of this type, such that for all $i \in [1, n]$,

$$r_i = \pi_{R_i}(r).$$

An example

Given the schema $\text{Person}(\text{SSN}, \text{Name}, \text{Address}, \text{Hobby})$, with two FDs: $\text{SSN} \rightarrow \text{Name}, \text{Address}$; and $\text{SSN} \rightarrow \text{Hobby}$, we can decompose it to $\text{Person1}(\text{SSN}, \text{Name}, \text{Address})$ with its FD being $\text{SSN} \rightarrow \text{Name}, \text{Address}$; and $\text{Hobby}(\text{SSN}, \text{Hobby})$, with FD $\text{SSN} \rightarrow \text{Hobby}$.

Another, though not interesting, is $\text{SSN}(\text{SSN}), \text{Name}(\text{Name}), \text{Address}(\text{Address}), \text{Hobby}(\text{Hobby})$. In this case, all the FD's are dropped.

Both of these two decompositions satisfy all the three conditions, thus, we clearly need more stuff to make a decomposition interesting and useful.

Lossless and lossy

Consider a relation instance r and its decomposition r_1, \dots, r_n . Since after this decomposition, the database no longer stores r , but r_i 's, it is natural to require that the database should be able to reconstruct r based on r_i 's. We usually use *natural join* as this reconstruction operator.

We thus have the following definition: A decomposition of $S = (\overline{R}, \mathcal{F})$ into a collection of $S_1 = (\overline{R}_1, \mathcal{F}_1), \dots, S_n = (\overline{R}_n, \mathcal{F}_n)$ is *lossless* if, for every instance r of S ,

$$r = r_1 \bowtie r_2 \bowtie \dots \bowtie r_n,$$

where for all $i \in [1, n]$,

$$r_i = \pi_{R_i}(r).$$

Homework: 6.18, and 6.22.

An example

Given the schema $\text{Person}(\text{SSN}, \text{Name}, \text{Address}, \text{Hobby})$, with two FDs: $\text{SSN} \rightarrow \text{Name}, \text{Address}$; and $\text{SSN} \rightarrow \text{Hobby}$, the following decomposition $\text{Person}_1(\text{SSN}, \text{Name}, \text{Address})$ with its FD being $\text{SSN} \rightarrow \text{Name}, \text{Address}$; and $\text{Hobby}(\text{SSN}, \text{Hobby})$, with FD $\text{SSN} \rightarrow \text{Hobby}$ is lossless since the join of these two tables gets back to the original one.

The other decomposition, i.e., $\{\text{SSN}(\text{SSN}), \text{Name}(\text{Name}), \text{Address}(\text{Address}), \text{Hobby}(\text{Hobby})\}$ is not.

How do I know...

There exists a simple 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 $S = (\overline{R}, \mathcal{F})$ be a relation schema and let $S_1 = (\overline{R}_1, \mathcal{F}_1), S_2 = (\overline{R}_2, \mathcal{F}_2)$ be a binary decomposition of S . It is lossless iff either i) $\overline{R}_1 \cap \overline{R}_2 \rightarrow \overline{R}_1 \in \mathcal{F}^+$; or ii) $\overline{R}_1 \cap \overline{R}_2 \rightarrow \overline{R}_2 \in \mathcal{F}^+$;

Here, \mathcal{F}^+ , the *closure* of \mathcal{F} is the collection of all the FDs that \mathcal{F} implies.

In other words, the above binary decomposition is lossless iff either $\overline{R}_1 \cap \overline{R}_2 \rightarrow \overline{R}_1$ or $\overline{R}_1 \cap \overline{R}_2 \rightarrow \overline{R}_2$ follows from \mathcal{F} .

An example

Given Person(SSN, Name, Address, Hobby) and FDs $SSN \rightarrow Name, Address$; and FD: $SSN \rightarrow Hobby$.

With the following decomposition: Person1(SSN, Name, Address) and Hobby(SSN, Hobby); we notice that $SSN \rightarrow Hobby$ is one of the original FD's; and we trivially have $SSN \rightarrow SSN$. Hence, $SSN \rightarrow SSN, Hobby$ is implied by the original FD set.

Finally, SSN is the intersection of the attributes of the two decomposed tables, and $\{SSN, Hobby\}$ is the collection of all the attributes of the second table, i.e., $\overline{R_1} \cap \overline{R_2} \rightarrow \overline{R_2}$.

Thus, by the trick, this decomposition is lossless.

Another example

Consider the following schema:

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

ClientId,OfficeId \rightarrow AccountNumber

AccountNumber \rightarrow OfficeId

Then the following decomposition is also lossless:

AcctOffice: (AccountNumber,OfficeId), with FD
AccountNumber \rightarrow OfficeId

and AcctClient: (AccountNumber, ClientId) with
no FD.

Because AccountNumber, the intersection of the two sets, is the key of the AcctOffice table, thus necessarily determines OfficeId.

What is missing?

Although the above decomposition is lossless, something is wrong: the following FD is homeless.

`ClientId,OfficeId → AccountNumber`

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 a join of two tables.

Thus, this decomposition leads to an increased cost of maintaining an integrity constraint that corresponding to the above FD.

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

An example

The decomposition of HasAccount to AcctOffice and AccClient does not preserve the dependency: Here is the original table

AcctNumber	ClientId	OfficeId
B123	1111	SB01
A908	1234	MN08

the one for AcctOffice

AcctNumber	OfficeId
B123	SB01
A908	MN08

and the one for AcctClient

AcctNumber	ClientId
B123	1111
A908	1234

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:

AcctNumber	OfficeId
B123	SB01
B567	SB01
A908	MN08

and the one for AcctClient

AcctNumber	ClientId
B123	1111
B567	1111
A908	1234

But, they actually violate the FD $ClientId, OfficeId \rightarrow AccountNumber, \dots$

A bit too late

..., which we will only be able to find out this violation after joining these two tables.

AcctNumber	ClientId	OfficeId
B123	1111	SB01
B567	1111	SB01
A908	1234	MN08

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.

Dependency preservation

Let $S = (\overline{R}, \mathcal{F})$ be a relation schema and $S_1 = (\overline{R}_1, \mathcal{F}_1), \dots, S_n = (\overline{R}_n, \mathcal{F}_n)$ be a decomposition of S . We say that this is a *dependency-preserving* decomposition iff \mathcal{F} and $\cup_{i=1}^n \mathcal{F}_i$ are *equivalent*.

In the lossless case, we only require $\cup_{i=1}^n \mathcal{F}_i$ follows from \mathcal{F} , sort of half way through.

Consider $S(\text{SSN}, \text{EmpId}, \text{DeptId})$ with FDs $\{f_1 : \text{SSN} \rightarrow \text{EmpId}; f_2 : \text{EmpId} \rightarrow \text{SSN}; f_3 : \text{SSN} \rightarrow \text{DeptId}\}$. Its decomposition into $S_1(\text{SSN}, \text{EmpId}; \{f_1, f_2\})$ and $S_2(\text{EmpId}, \text{DeptId}; f_4 : \{\text{EmpId} \rightarrow \text{DeptId}\})$ is dependency preserving.

Notice that $f_3 \notin \mathcal{F}_1 \cup \mathcal{F}_2$, but it can be derived from f_1 and f_4 ; and f_4 can be derived from f_2 and f_3 . Thus, $\mathcal{F}^+ = (\mathcal{F}_1 \cup \mathcal{F}_2)^+$.

How hard is it?

If we have already had a decomposition complete with FD sets attached to the component schemas, dependency preservation can be relatively checked out: We just make sure that each FD in the original FD is entailed by the union of the local FDs.

It is more difficult in general, since we have to make the decomposition then figure out the respective dependencies and attach them to the local attributes.

Which one do you like better?

We have so far discussed two important features of decomposition: losslessness and dependency preservation. We have also seen a BCNF decomposition that is lossless but not dependency preserving.

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

Question: Which property is more important?

Answer: The former is mandatory and the latter is optional.

Now we are ready to get into the decomposition business.

BCNF Decomposition

Let $S_0 = (\overline{R}, \mathcal{F})$ be a relation schema, the following algorithm constructs a decomposition of S by repeatedly splitting \overline{R} into smaller subschemas and 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. $D = \{S_0\}$
2. While $S = (\overline{S}; \mathcal{F}') \in D$ is not in BCNF do
3. Let $\overline{X} \rightarrow \overline{Y} \in \mathcal{F}'^+$ such that $\overline{X\overline{Y}} \subseteq \overline{S}$
 and it violates BCNF in S
4. Replace S with $S_1 = (\overline{X\overline{Y}}; \mathcal{F}'_1)$ and
 $S_2 = ((S - \overline{Y}) \cup \overline{X}; \mathcal{F}'_2)$, where
 $\mathcal{F}'_1 = \pi_{\overline{X\overline{Y}}}(\mathcal{F}')$, $\mathcal{F}'_2 = \pi_{(S - \overline{Y}) \cup \overline{X}}(\mathcal{F}')$.

An example

Consider the HasAccount(AccountNumber, ClientId, OfficeId) and the FD: { ClientId, OfficeId \rightarrow AccountNumber, AccountNumber \rightarrow 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:

$S_1 = (\{\text{AccountNumber, OfficeId}\}; \{\text{AccountNumber} \rightarrow \text{OfficeId}\})$ and $S_2 = (\{\text{ClientId, AccountNumber}\}; \{\text{ClientID} \rightarrow \text{AccountNumber}\})$.

Question: Is the last one in BCNF?

Some properties

1. The algorithm always comes up with a lossless decomposition in every step.

The respective attributes sets of the two decompositions are \overline{XY} and $(S - \overline{Y}) \cup \overline{X}$. Since $\overline{XY} \cap ((S - \overline{Y}) \cup \overline{X}) = \overline{X}$, and $\overline{X} \rightarrow \overline{Y}$, we have that $\overline{XY} \cap ((S - \overline{Y}) \cup \overline{X}) \rightarrow XY$, by *the augmentation axiom* (Cf. p. 221). Thus, by the lossless test condition, such a decomposition must be lossless.

Thus, the final result of this process is a lossless decomposition.

2. As we have already seen, the final result does not always preserve data dependency.
3. Since there could be multiple rules violating the condition, this process is also non-deterministic.

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 already 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 is likely to 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. There also exist 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 multivalued redundancy as we saw that might exist in BCNF schemas.

To decompose or not?

It is really hard to say how much decomposition should be done. We have to notice the following:

1. Decomposition usually makes answering complexity question less efficient since decomposition leads to more join which takes time.
2. It can make answering simple questions more efficient since such query usually uses smaller number of attributes belonging to the same relation. Decomposition leads to less projection and selection operation.

3. It makes simple update more efficient but complex update less efficient since the latter might need more attributes spread over multiple tables.

4. It will lower the space requirement since it often cuts out redundancy, but it may lead to more space if the redundancy is already low.

Homework: Self study §6.12 through the end of the first paragraph of p.243.