Chapter 7

Relational Database Design

The goal of relational database design is to generate a set of schemas that allow us to

- Store information without unnecessary redundancy.
- Retrieve information easily (and accurately).

7.1 Pitfalls in Relational DB Design

A bad design may have several properties, including:

- Repetition of information.
- Inability to represent certain information.
- Loss of information.

7.1.1 Representation of Information

1. Suppose we have a schema, *Lending-schema*,

   Lending-schema = (bname, bcity, assets, cname, loan#, amount)

   and suppose an instance of the relation is Figure 7.1.

2. A tuple t in the new relation has the following attributes:

   - $t[assets]$ is the assets for $t[bname]$
   - $t[bcity]$ is the city for $t[bname]$

bname	bcity	assets	cname	loan#	amount
SFU	Burnaby	2M	Tom	L-10	10K
SFU	Burnaby	2M	Mary	L-20	15K
Downtown	Vancouver	8M	Tom	L-50	50K

Figure 7.1: Sample *lending* relation.

bname	bcity	assets	cname
SFU	Burnaby	2M	Tom
SFU	Burnaby	2M	Mary
Downtown	Vancouver	8M	Tom

cname	loan#	amount
Tom	L-10	10K
Mary	L-20	15K
Tom	L-50	50K

Figure 7.2: The decomposed *lending* relation.

- $t[loan\#]$ is the loan number made by branch $t[bname]$ to $t[cname]$.
- $t[amount]$ is the amount of the loan for $t[loan\#]$

3. If we wish to add a loan to our database, the original design would require adding a tuple to *borrow*:

   (SFU, L-31, Turner, 1K)

4. In our new design, we need a tuple with all the attributes required for *Lending-schema*. Thus we need to insert

   (SFU, Burnaby, 2M, Turner, L-31, 1K)

5. We are now repeating the assets and branch city information for every loan.

   - Repetition of information wastes space.
   - Repetition of information complicates updating.

6. Under the new design, we need to change many tuples if the branch's assets change.

7. Let's analyze this problem:

   - We know that a branch is located in exactly one city.
   - We also know that a branch may make many loans.
   - The functional dependency $bname \rightarrow bcity$ holds on *Lending-schema*.
   - The functional dependency $bname \rightarrow loan\#$ does not.
   - These two facts are best represented in separate relations.

8. Another problem is that we cannot represent the information for a branch (assets and city) unless we have a tuple for a loan at that branch.

9. Unless we use nulls, we can only have this information when there are loans, and must delete it when the last loan is paid off.

7.2 Decomposition

1. The previous example might seem to suggest that we should decompose schema as much as possible.

   Careless decomposition, however, may lead to another form of bad design.

2. Consider a design where *Lending-schema* is decomposed into two schemas

   Branch-customer-schema = (bname, bcity, assets, cname)
   Customer-loan-schema = (cname, loan#, amount)

3. We construct our new relations from *lending* by:

   $branch\text{-}customer = \Pi_{bname,bcity,assets,cname}(lending)$
   $customer\text{-}loan = \Pi_{cname,loan\#,amount}(lending)$

4. It appears that we can reconstruct the *lending* relation by performing a natural join on the two new schemas.

bname	bcity	assets	cname	loan#	amount
SFU	Burnaby	2M	Tom	L-10	10K
SFU	Burnaby	2M	Tom	L-50	50K
SFU	Burnaby	2M	Mary	L-20	15K
Downtown	Vancouver	8M	Tom	L-10	10K
Downtown	Vancouver	8M	Tom	L-50	50K

Figure 7.3: Join of the decomposed relations.

5. Figure 7.3 shows what we get by computing *branch-customer* $\bowtie$ *customer-loan*.
6. We notice that there are tuples in *branch-customer* $\bowtie$ *customer-loan* that are not in *lending*.
7. How did this happen?
   - The intersection of the two schemas is *cname*, so the natural join is made on the basis of equality in the cname.
   - If two lendings are for the same customer, there will be four tuples in the natural join.
   - Two of these tuples will be spurious - they will not appear in the original *lending* relation, and should not appear in the database.
   - Although we have **more** tuples in the join, we have **less** information.
   - Because of this, we call this a **lossy** or **lossy-join decomposition**.
   - A decomposition that is not lossy-join is called a **lossless-join decomposition**.
   - The only way we could make a connection between *branch-customer* and *customer-loan* was through *cname*.
8. When we decomposed *Lending-schema* into *Branch-schema* and *Loan-info-schema*, we will not have a similar problem. Why not?

   Branch-schema = (bname, bcity, assets)
   Branch-loan-schema = (bname, cname, loan#, amount)

   - The only way we could represent a relationship between tuples in the two relations is through *bname*.
   - This will not cause problems.
   - For a given branch name, there is exactly one assets value and branch city.
9. For a given branch name, there is exactly one assets value and exactly one bcity; whereas a similar statement associated with a loan depends on the customer, not on the amount of the loan (which is not unique).
10. We'll make a more formal definition of lossless-join:
    - Let R be a relation schema.
    - A set of relation schemas $\{R_1, R_2, \ldots, R_n\}$ is a **decomposition** of R if

      $$R = R_1 \cup R_2 \cup \ldots \cup R_n$$

    - That is, every attribute in R appears in at least one R_i for $1 \leq i \leq n$.
    - Let r be a relation on R, and let

      $$r_i = \Pi_{R_i}(r) \text{ for } 1 \leq i \leq n$$

    - That is, $\{r_1, r_2, \ldots r_n\}$ is the database that results from decomposing R into $\{R_1, R_2, \ldots R_n\}$.
    - It is always the case that:

      $$r \subseteq r_1 \bowtie r_2 \bowtie \ldots \bowtie r_n$$

- To see why this is, consider a tuple $t \in r$.
  - When we compute the relations $\{r_1, r_2, \ldots r_n\}$, the tuple t gives rise to one tuple t_i in each r_i.
  - These n tuples combine together to regenerate t when we compute the natural join of the r_i.
  - Thus every tuple in r appears in $\bowtie_{i=1}^{n} r_i$.
- However, in general,
$$r \neq r_1 \bowtie r_2 \bowtie \ldots \bowtie r_n$$
- We saw an example of this inequality in our decomposition of *lending* into *branch-customer* and *customer-loan*.
- In order to have a lossless-join decomposition, we need to impose some constraints on the set of possible relations.
- Let C represent a set of constraints on the database.
- A decomposition $\{R_1, R_2, \ldots, R_n\}$ of a relation schema R is a **lossless-join decomposition** for R if, for all relations r on schema R that are legal under C:
$$r = \Pi_{R_1}(r) \bowtie \Pi_{R_2}(r) \bowtie \ldots \bowtie \Pi_{R_n}(r)$$

11. In other words, a lossless-join decomposition is one in which, for any legal relation r, if we decompose r and then "recompose" r, we get what we started with – no more and no less.

7.3 Normalization Using Functional Dependencies

We can use functional dependencies to design a relational database in which most of the problems we have seen do not occur.

Using functional dependencies, we can define several **normal forms** which represent "good" database designs.

7.3.1 Desirable Properties of Decomposition

1. We'll take another look at the schema

   Lending-schema = (bname, assets, bcity, loan#, cname, amount)

   which we saw was a bad design.

2. The set of functional dependencies we required to hold on this schema was:

   bname $\rightarrow$ *assets bcity*
   loan# $\rightarrow$ *amount bname*

3. If we decompose it into

   Branch-schema = (bname, assets, bcity)
   Loan-info-schema = (bname, loan#, amount)
   Borrow-schema = (cname, loan#)

   we claim this decomposition has several desirable properties.

Lossless-Join Decomposition

1. We claim the above decomposition is lossless. How can we decide whether a decomposition is lossless?

   - Let R be a relation schema.
   - Let F be a set of functional dependencies on R.
   - Let R_1 and R_2 form a decomposition of R.

- The decomposition is a lossless-join decomposition of R if at least one of the following functional dependencies are in F^+:

  (a) $R_1 \cap R_2 \rightarrow R_1$

  (b) $R_1 \cap R_2 \rightarrow R_2$

  Why is this true? Simply put, it ensures that the attributes involved in the natural join ($R_1 \cap R_2$) are a candidate key for at least one of the two relations.

  This ensures that we can never get the situation where spurious tuples are generated, as for any value on the join attributes there will be a unique tuple in **one** of the relations.

2. We'll now show our decomposition is lossless-join by showing a set of steps that generate the decomposition:

   - First we decompose *Lending-schema* into
     Branch-schema = (bname, bcity, assets)
     Loan-info-schema = (bname, cname, loan#, amount)
   - Since *bname* $\rightarrow$ *assets bcity*, the augmentation rule for functional dependencies implies that
     bname $\rightarrow$ *bname assets bcity*
   - Since *Branch-schema* $\cap$ *Borrow-schema* = *bname*, our decomposition is lossless join.
   - Next we decompose *Borrow-schema* into
     Loan-schema = (bname, loan#, amount)
     Borrow-schema = (cname, loan#)
   - As *loan#* is the common attribute, and
     loan# $\rightarrow$ *amount bname*
     This is also a lossless-join decomposition.

Dependency Preservation

1. Another desirable property in database design is **dependency preservation**.

   - We would like to check easily that updates to the database do not result in illegal relations being created.
   - It would be nice if our design allowed us to check updates without having to compute natural joins.
   - To know whether joins must be computed, we need to determine what functional dependencies may be tested by checking each relation individually.
   - Let F be a set of functional dependencies on schema R.
   - Let $\{R_1, R_2, \ldots, R_n\}$ be a decomposition of R.
   - The **restriction** of F to R_i is the set of all functional dependencies in F^+ that include only attributes of R_i.
   - Functional dependencies in a restriction can be tested in one relation, as they involve attributes in one relation schema.
   - The set of restrictions $F_1, F_2, \ldots, F_n$ is the set of dependencies that can be checked efficiently.
   - We need to know whether testing only the restrictions is sufficient.
   - Let $F' = F_1, F_2, \ldots, F_n$.
   - F' is a set of functional dependencies on schema R, but in general, $F' \neq F$.
   - However, it may be that $F'^+ = F^+$.
   - If this is so, then every functional dependency in F is implied by F', and if F' is satisfied, then F must also be satisfied.
   - A decomposition having the property that $F'^+ = F^+$ is a **dependency-preserving** decomposition.

2. The algorithm for testing dependency preservation follows this method:

```
compute F+
for each schema Ri in D do
  begin
    Fi := the restriction of F+ to Ri;
  end
F' = ∅
for each restriction Fi do
  begin
    F' = F' ∪ Fi
  end
compute F'+;
if (F'+ = F+) then return (true)
          else return (false);
```

3. We can now show that our decomposition of *Lending-schema* is dependency preserving.

   - The functional dependency

     $bname \rightarrow assets\ bcity$

     can be tested in one relation on *Branch-schema*.
   - The functional dependency

     $loan\# \rightarrow amount\ bname$

     can be tested in *Loan-schema*.

4. As the above example shows, it is often easier not to apply the algorithm shown to test dependency preservation, as computing F^+ takes exponential time.

5. **An Easier Way To Test For Dependency Preservation**

   Really we only need to know whether the functional dependencies in F and not in F' are implied by those in F'.

   In other words, are the functional dependencies not easily checkable logically implied by those that are?

   Rather than compute F^+ and F'^+, and see whether they are equal, we can do this:

   - Find $F - F'$, the functional dependencies not checkable in one relation.
   - See whether this set is obtainable from F' by using Armstrong's Axioms.
   - This should take a great deal less work, as we have (usually) just a few functional dependencies to work on.

   Use this simpler method on exams and assignments (unless you have exponential time available to you).

Repetition of Information

1. Our decomposition does not suffer from the repetition of information problem.

   - Branch and loan data are separated into distinct relations.
   - Thus we do not have to repeat branch data for each loan.
   - If a single loan is made to several customers, we do not have to repeat the loan amount for each customer.
   - This lack of redundancy is obviously desirable.
   - We will see how this may be achieved through the use of **normal forms**.

7.3.2 Boyce-Codd Normal Form

1. A relation schema R is in **Boyce-Codd Normal Form (BCNF)** with respect to a set F of functional dependencies if for all functional dependencies in F^+ of the form $\alpha \rightarrow \beta$, where $\alpha \subseteq R$ and $\beta \subseteq R$, at least one of the following holds:
   - $\alpha \rightarrow \beta$ is a trivial functional dependency (i.e. $\beta \subseteq \alpha$).
   - α is a superkey for schema R.
2. A database design is in BCNF if each member of the set of relation schemas is in BCNF.
3. Let's assess our example banking design:
   Customer-schema = (cname, street, ccity)
   cname $\rightarrow$ *street ccity*
   Branch-schema = (bname, assets, bcity)
   bname $\rightarrow$ *assets bcity*
   Loan-info-schema = (bname, cname, loan#, amount)
   loan# $\rightarrow$ *amount bname*
   Customer-schema and *Branch-schema* are in BCNF.
4. Let's look at *Loan-info-schema*:
   - We have the non-trivial functional dependency *loan#* $\rightarrow$ *amount*, and
   - *loan#* is not a superkey.
   - Thus *Loan-info-schema* is not in BCNF.
   - We also have the repetition of information problem.
   - For each customer associated with a loan, we must repeat the branch name and amount of the loan.
   - We can eliminate this redundancy by decomposing into schemas that are all in BCNF.
5. If we decompose into
   Loan-schema = (bname, loan#, amount)
   Borrow-schema = (cname, loan#)
   we have a lossless-join decomposition. (Remember why?)
   To see whether these schemas are in BCNF, we need to know what functional dependencies apply to them.
   - For *Loan-schema*, we have *loan#* $\rightarrow$ *amount bname* applying.
   - Only trivial functional dependencies apply to *Borrow-schema*.
   - Thus both schemas are in BCNF.

   We also no longer have the repetition of information problem. Branch name and loan amount information are not repeated for each customer in this design.
6. Now we can give a general method to generate a collection of BCNF schemas.
   result := $\{R\}$;
   done := false;
   compute F^+;
   while (not *done*) **do**
   if (there is a schema R_i in *result* that is not in BCNF)
   then begin
   let $\alpha \rightarrow \beta$ be a nontrivial functional dependency that holds on R_i
   such that $\alpha \rightarrow R_i$ is not in F^+, and $\alpha \cap \beta = \emptyset$;
   result = $(result - R_i) \cup (R_i - \beta) \cup (\alpha, \beta)$;
   end
   else *done* = true;
7. This algorithm generates a lossless-join BCNF decomposition. Why?

- We replace a schema R_i with $(R_i - \beta)$ and (α, β).
- The dependency $\alpha \rightarrow \beta$ holds on R_i.
- $(R_i - \beta) \cap (\alpha, \beta) = \alpha$.
- So we have $R_1 \cap R_2 \rightarrow R_2$, and thus a lossless join.

8. Let's apply this algorithm to our earlier example of poor database design:

   Lending-schema = (bname, assets, bcity, loan#, cname, amount)

   The set of functional dependencies we require to hold on this schema are

   bname $\rightarrow$ *assets bcity*
   loan# $\rightarrow$ *amount bname*

   A candidate key for this schema is {*loan#, cname*}.

   We will now proceed to decompose:

   - The functional dependency

     bname $\rightarrow$ *assets bcity*

     holds on *Lending-schema*, but *bname* is not a superkey. We replace *Lending-schema* with
     Branch-schema = (bname, assets, bcity)
     Loan-info-schema = (bname, loan#, cname, amount)
   - *Branch-schema* is now in BCNF.
   - The functional dependency

     loan# $\rightarrow$ *amount bname*

     holds on *Loan-info-schema*, but *loan#* is not a superkey.

     We replace *Loan-info-schema* with

     Loan-schema = (bname, loan#, amount)
     Borrow-schema = (cname, loan#)
   - These are both now in BCNF.
   - We saw earlier that this decomposition is both lossless-join and dependency-preserving.

9. Not every decomposition is dependency-preserving.

   - Consider the relation schema

     Banker-schema = (bname, cname, banker-name)
   - The set F of functional dependencies is

     banker-name $\rightarrow$ *bname*
     cname bname $\rightarrow$ *banker-name*
   - The schema is not in BCNF as *banker-name* is not a superkey.
   - If we apply our algorithm, we may obtain the decomposition

     Banker-branch-schema = (bname, banker-name)
     Cust-banker-schema = (cname, banker-name)
   - The decomposed schemas preserve only the first (and trivial) functional dependencies.
   - The closure of this dependency does not include the second one.
   - Thus a violation of *cname bname* $\rightarrow$ *banker-name* cannot be detected unless a join is computed.

   This shows us that not every BCNF decomposition is dependency-preserving.

10. It is not always possible to satisfy all three design goals:

    - BCNF.
    - Lossless join.

- Dependency preservation.

11. We can see that any BCNF decomposition of *Banker-schema* must fail to preserve

    $cname\ bname \rightarrow banker\text{-}name$

12. **Some Things To Note About BCNF**

    - There is sometimes more than one BCNF decomposition of a given schema.
    - The algorithm given produces only one of these possible decompositions.
    - Some of the BCNF decompositions may also yield dependency preservation, while others may not.
    - Changing the order in which the functional dependencies are considered by the algorithm may change the decomposition.
    - For example, try running the BCNF algorithm on

      $R = (A, B, C, D)$

      $A \rightarrow B, C$
      $B \rightarrow D$
      $D \rightarrow B$

      Then change the order of the last two functional dependencies and run the algorithm again. Check the two decompositions for dependency preservation.

7.3.3 Third Normal Form

1. When we cannot meet all three design criteria, we abandon BCNF and accept a weaker form called **third normal form (3NF)**.

2. It is always possible to find a dependency-preserving lossless-join decomposition that is in 3NF.

3. A relation schema R is in **3NF** with respect to a set F of functional dependencies if for all functional dependencies in F^+ of the form $\alpha \rightarrow \beta$, where $\alpha \subseteq R$ and $\beta \subseteq R$, at least one of the following holds:

   - $\alpha \rightarrow \beta$ is a trivial functional dependency.
   - α is a superkey for schema R.
   - Each attribute A in $\beta - \alpha$ is contained in a candidate key for R.

4. A database design is in 3NF if each member of the set of relation schemas is in 3NF.

5. We now allow functional dependencies satisfying only the third condition. These dependencies are called **transitive dependencies**, and are not allowed in BCNF.

6. As all relation schemas in BCNF satisfy the first two conditions only, a schema in BCNF is also in 3NF.

7. BCNF is a more restrictive constraint than 3NF.

8. Our *Banker-schema* decomposition did not have a dependency-preserving lossless-join decomposition into BCNF. The schema was already in 3NF though (check it out).

9. We now present an algorithm for finding a dependency-preserving lossless-join decomposition into 3NF.

10. Note that we require the set F of functional dependencies to be in **canonical form**.

    let F_c be a **canonical cover** for F;
    $i := 0$;
    for each functional dependency $\alpha \rightarrow \beta \in F_c$ **do**
    if none of the schemas R_j, $1 \leq j \leq i$ contains $\alpha\beta$
      then begin
        $i := i + 1$;
        $R_i := \alpha\beta$
      end
    if none of the schemas R_j, $1 \leq j \leq i$ contains a candidate key for R
      then begin
        $i := i + 1$;
        $R_i :=$ any candidate key for R
      end
    return $(R_1, R_2, \ldots, R_i)$

11. Each relation schema is in 3NF. Why? (A proof is given is [Ullman 1988].)

12. The design is dependency-preserving as a schema is built for each given dependency.

    Lossless-join is guaranteed by the requirement that a candidate key for R be in at least one of the schemas.

13. To review our *Banker-schema* consider an extension to our example:

    Banker-info-schema = (bname, cname, banker-name, office#)

    The set F of functional dependencies is

    banker-name $\rightarrow$ *bname office#*
    cname bname $\rightarrow$ *banker-name*

    The **for** loop in the algorithm gives us the following decomposition:

    Banker-office-schema = (banker-name, bname, office#)
    Banker-schema = (cname, bname, banker-name)

    Since *Banker-schema* contains a candidate key for *Banker-info-schema*, the process is finished.

7.3.4 Comparison of BCNF and 3NF

1. We have seen BCNF and 3NF.
   - It is always possible to obtain a 3NF design without sacrificing lossless-join or dependency-preservation.
   - If we do not eliminate all transitive dependencies, we may need to use null values to represent some of the meaningful relationships.
   - Repetition of information occurs.

2. These problems can be illustrated with *Banker-schema*.
   - As *banker-name* $\rightarrow$ *bname* , we may want to express relationships between a banker and his or her branch.
   - Figure 7.4 shows how we must either have a corresponding value for customer name, or include a null.
   - Repetition of information also occurs.
   - Every occurrence of the banker's name must be accompanied by the branch name.

3. If we must choose between BCNF and dependency preservation, it is generally better to opt for 3NF.
   - If we cannot check for dependency preservation efficiently, we either pay a high price in system performance or risk the integrity of the data.

cname	banker-name	bname
Bill	John	SFU
Tom	John	SFU
Mary	John	SFU
null	Tim	Austin

Figure 7.4: An instance of *Banker-schema*.

- The limited amount of redundancy in 3NF is then a lesser evil.

4. To summarize, our goal for a relational database design is
   - BCNF.
   - Lossless-join.
   - Dependency-preservation.
5. If we cannot achieve this, we accept
   - 3NF
   - Lossless-join.
   - Dependency-preservation.
6. **A final point:** there is a price to pay for decomposition. When we decompose a relation, we have to use natural joins or Cartesian products to put the pieces back together. This takes computational time.

7.4 Normalization Using Multivalued Dependencies (not to be covered)

1. Suppose that in our banking example, we had an alternative design including the schema:

   BC-schema = (loan#, cname, street, ccity)

   We can see this is not BCNF, as the functional dependency

   $cname \rightarrow street\ ccity$

   holds on this schema, and *cname* is not a superkey.
2. If we have customers who have several addresses, though, then we no longer wish to enforce this functional dependency, and the schema is in BCNF.
3. However, we now have the repetition of information problem. For each address, we must repeat the loan numbers for a customer, and vice versa.

7.4.1 Multivalued Dependencies

1. **Functional dependencies** rule out certain tuples from appearing in a relation.

   If $A \rightarrow B$, then we cannot have two tuples with the same A value but different B values.
2. **Multivalued dependencies** do not rule out the existence of certain tuples.

   Instead, they **require** that other tuples of a certain form be present in the relation.
3. Let R be a relation schema, and let $\alpha \subseteq R$ and $\beta \subseteq R$.

   The **multivalued dependency**

   $$\alpha \twoheadrightarrow \beta$$

	α	β	$R-\alpha-\beta$
t_1	$a_1 \cdots a_i$	$a_{i+1} \cdots a_j$	$a_{j+1} \cdots a_n$
t_2	$a_1 \cdots a_i$	$b_{i+1} \cdots b_j$	$b_{j+1} \cdots b_n$
t_3	$a_1 \cdots a_i$	$a_{i+1} \cdots a_j$	$b_{j+1} \cdots b_n$
t_4	$a_1 \cdots a_i$	$b_{i+1} \cdots b_j$	$a_{j+1} \cdots a_n$

Figure 7.5: Tabular representation of $\alpha \twoheadrightarrow \beta$.

name	address	car
Tom	North Rd.	Toyota
Tom	Oak St.	Honda
Tom	North Rd.	Honda
Tom	Oak St.	Toyota

Figure 7.6: $(name, address, car)$ where $name \twoheadrightarrow address$ and $name \twoheadrightarrow car$.

holds on R if in any legal relation $r(R)$, for all pairs of tuples t_1 and t_2 in r such that $t_1[\alpha] = t_2[\alpha]$, there exist tuples t_3 and t_4 in r such that:

$t_1[\alpha] = t_2[\alpha] = t_3[\alpha] = t_4[\alpha]$
$t_3[\beta] = t_1[\beta]$
$t_3[R-\beta] = t_2[R-\beta]$
$t_4[\beta] = t_2[\beta]$
$t_4[R-\beta] = t_1[R-\beta]$

4. Figure 7.5 (textbook 6.10) shows a tabular representation of this. It looks horrendously complicated, but is really rather simple. A simple example is a table with the schema $(name, address, car)$, as shown in Figure 7.6.

   - Intuitively, $\alpha \twoheadrightarrow \beta$ says that the relationship between α and β is independent of the relationship between α and $R-\beta$.
   - If the multivalued dependency $\alpha \twoheadrightarrow \beta$ is satisfied by all relations on schema R, then we say it is a **trivial** multivalued dependency on schema R.
   - Thus $\alpha \twoheadrightarrow \beta$ is trivial if $\beta \subseteq \alpha$ or $\beta \cup \alpha = R$.

5. Look at the example relation *bc* relation in Figure 7.7 (textbook 6.11).

   - We must repeat the loan number once for each address a customer has.
   - We must repeat the address once for each loan the customer has.
   - This repetition is pointless, as the relationship between a customer and a loan is independent of the relationship between a customer and his or her address.

loan#	cname	street	ccity
23	Smith	North	Rye
23	Smith	Main	Manchester
93	Curry	Lake	Horseneck

Figure 7.7: Relation bc, an example of redundancy in a BCNF relation.

loan#	cname	street	ccity
23	Smith	North	Rye
27	Smith	Main	Manchester

Figure 7.8: An illegal bc relation.

- If a customer, say "Smith", has loan number 23, we want all of Smith's addresses to be associated with that loan.
- Thus the relation of Figure 7.8 (textbook 6.12) is illegal.
- If we look at our definition of multivalued dependency, we see that we want the multivalued dependency
  $$cname \twoheadrightarrow street\ ccity$$
  to hold on *BC-schema*.

6. Note that if a relation r fails to satisfy a given multivalued dependency, we can construct a relation r' that does satisfy the multivalued dependency by adding tuples to r.

7.4.2 Theory of Multivalued Dependencies

1. We will need to compute all the multivalued dependencies that are logically implied by a given set of multivalued dependencies.
   - Let D denote a set of functional and multivalued dependencies.
   - The closure D^+ of D is the set of all functional and multivalued dependencies logically implied by D.
   - We can compute D^+ from D using the formal definitions, but it is easier to use a set of inference rules.
2. The following set of inference rules is **sound** and **complete**. The first three rules are Armstrong's axioms from Chapter 5.
   (a) **Reflexivity rule:** if α is a set of attributes and $\beta \subseteq \alpha$, then $\alpha \rightarrow \beta$ holds.
   (b) **Augmentation rule:** if $\alpha \rightarrow \beta$ holds, and γ is a set of attributes, then $\gamma\alpha \rightarrow \gamma\beta$ holds.
   (c) **Transitivity rule:** if $\alpha \rightarrow \beta$ holds, and $\beta \rightarrow \gamma$ holds, then $\alpha \rightarrow \gamma$ holds.
   (d) **Complementation rule:** if $\alpha \twoheadrightarrow \beta$ holds, then $\alpha \twoheadrightarrow R - \beta - \alpha$ holds.
   (e) **Multivalued augmentation rule:** if $\alpha \twoheadrightarrow \beta$ holds, and $\gamma \subseteq R$ and $\delta \subseteq \gamma$, then $\gamma\alpha \twoheadrightarrow \delta\beta$ holds.
   (f) **Multivalued transitivity rule:** if $\alpha \twoheadrightarrow \beta$ holds, and $\beta \twoheadrightarrow \gamma$ holds, then $\alpha \twoheadrightarrow \gamma - \beta$ holds.
   (g) **Replication rule:** if $\alpha \rightarrow \beta$ holds, then $\alpha \twoheadrightarrow \beta$.
   (h) **Coalescence rule:** if $\alpha \twoheadrightarrow \beta$ holds, and $\gamma \subseteq \beta$, and there is a δ such that $\delta \subseteq R$ and $\delta \cap \beta = \emptyset$ and $\delta \rightarrow \gamma$, then $\alpha \rightarrow \gamma$ holds.
3. An example of *multivalued transitivity rule* is as follows. $loan\# \twoheadrightarrow cname$ and $cname \twoheadrightarrow \{cname, caddress\}$. Thus we have $loan\# \twoheadrightarrow caddress$, where $caddress = \{cname, caddress\} - cname$.
   An example of *coalescence rule* is as follows. If we have $student\_name \twoheadrightarrow \{bank, account\}$, and $student\_id \rightarrow bank$, then we have $student\_name \rightarrow bank$.
4. Let's do an example:
   - Let $R = (A, B, C, G, H, I)$ be a relation schema.
   - Suppose $A \twoheadrightarrow BC$ holds.

- The definition of multivalued dependencies implies that if $t_1[A] = t_2[A]$, then there exists tuples t_3 and t_4 such that:
  $t_1[A] = t_2[A] = t_3[A] = t_4[A]$
  $t_3[BC] = t_1[BC]$
  $t_3[GHI] = t_2[GHI]$
  $t_4[GHI] = t_1[GHI]$
  $t_4[BC] = t_2[BC]$
- The complementation rule states that if $A \twoheadrightarrow BC$ then $A \twoheadrightarrow GHI$.
- Tuples t_3 and t_4 satisfy $A \twoheadrightarrow GHI$ if we simply change the subscripts.

5. We can simplify calculating D^+, the closure of D by using the following rules, derivable from the previous ones:
   - **Multivalued union rule:** if $\alpha \twoheadrightarrow \beta$ holds and $\alpha \twoheadrightarrow \gamma$ holds, then $\alpha \twoheadrightarrow \beta\gamma$ holds.
   - **Intersection rule:** if $\alpha \twoheadrightarrow \beta$ holds and $\alpha \twoheadrightarrow \gamma$ holds, then $\alpha \twoheadrightarrow \beta \cap \gamma$ holds.
   - **Difference rule:** if $\alpha \twoheadrightarrow \beta$ holds and $\alpha \twoheadrightarrow \gamma$ holds, then $\alpha \twoheadrightarrow \beta - \gamma$ holds and $\alpha \twoheadrightarrow \gamma - \beta$ holds.
6. An example will help:
   Let $R = (A, B, C, G, H, I)$ with the set of dependencies:
   $A \twoheadrightarrow B$
   $B \twoheadrightarrow HI$
   $CG \rightarrow H$
   We list some members of D^+:
   - $A \twoheadrightarrow CGHI$: since $A \twoheadrightarrow B$, complementation rule implies that $A \twoheadrightarrow R-B-A$, and $R-B-A = CGHI$.
   - $A \twoheadrightarrow HI$: Since $A \twoheadrightarrow B$ and $B \twoheadrightarrow HI$, multivalued transitivity rule implies that $A \twoheadrightarrow HI - B$.
   - $B \rightarrow H$: coalescence rule can be applied. $B \twoheadrightarrow HI$ holds, $H \subseteq HI$ and $CG \twoheadrightarrow H$ and $CG \cap HI = \emptyset$, so we can satisfy the coalescence rule with α being B, β being HI, δ being CG, and γ being H. We conclude that $B \rightarrow H$.
   - $A \twoheadrightarrow CG$: now we know that $A \twoheadrightarrow CGHI$ and $A \twoheadrightarrow HI$. By the difference rule, $A \twoheadrightarrow CGHI - HI = CG$.

7.4.3 Fourth Normal Form (4NF)

1. We saw that *BC-schema* was in BCNF, but still was not an ideal design as it suffered from repetition of information. We had the multivalued dependency $cname \twoheadrightarrow street\ ccity$, but no non-trivial functional dependencies.
2. We can use the given multivalued dependencies to improve the database design by decomposing it into **fourth normal form**.
3. A relation schema R is in 4NF with respect to a set D of functional and multivalued dependencies if for all multivalued dependencies in D^+ of the form $\alpha \twoheadrightarrow \beta$, where $\alpha \subseteq R$ and $\beta \subseteq R$, at least one of the following hold:
   - $\alpha \twoheadrightarrow \beta$ is a trivial multivalued dependency.
   - α is a superkey for schema R.
4. A database design is in 4NF if each member of the set of relation schemas is in 4NF.
5. The definition of 4NF differs from the BCNF definition only in the use of multivalued dependencies.
   - Every 4NF schema is also in BCNF.
   - To see why, note that if a schema is not in BCNF, there is a non-trivial functional dependency $\alpha \rightarrow \beta$ holding on R, where α is not a superkey.

- Since $\alpha \rightarrow \beta$ implies $\alpha \twoheadrightarrow \beta$, by the replication rule, R cannot be in 4NF.

6. We have an algorithm similar to the BCNF algorithm for decomposing a schema into 4NF:

```
result := {R};
done := false;
compute D+;
while (not done) do
  if (there is a schema Ri in result
        that is not in 4NF)
    then begin
      let α ↠ β be a nontrivial multivalued
      dependency that holds on Ri such that
        α → Ri is not in D+, and
        α ∩ β = ∅;
      result = (result − Ri) ∪ (Ri − β) ∪ (α, β);
    end
  else done = true;
```

7. If we apply this algorithm to *BC-schema*:

   - *cname* $\twoheadrightarrow$ *loan#* is a nontrivial multivalued dependency and *cname* is not a superkey for the schema.
   - We then replace *BC-schema* by two schemas:
     Cust-loan-schema=(cname, loan#)
     Customer-schema=(cname, street, ccity)
   - These two schemas are in 4NF.

8. We can show that our algorithm generates only lossless-join decompositions.

   - Let R be a relation schema and D a set of functional and multivalued dependencies on R.
   - Let R_1 and R_2 form a decomposition of R.
   - This decomposition is lossless-join if and only if at least one of the following multivalued dependencies is in D^+:
     $R_1 \cap R_2 \twoheadrightarrow R_1$
     $R_1 \cap R_2 \twoheadrightarrow R_2$
   - We saw similar criteria for functional dependencies.
   - This says that for **every** lossless-join decomposition of R into two schemas R_1 and R_2, one of the two above dependencies must hold.
   - You can see, by inspecting the algorithm, that this must be the case for every decomposition.

9. Dependency preservation is not as simple to determine as with functional dependencies.

   - Let R be a relation schema.
   - Let $R_1, R_2, \ldots R_n$ be a decomposition of R.
   - Let D be the set of functional and multivalued dependencies holding on R.
   - The **restriction** of D to R_i is the set D_i consisting of:
     - All functional dependencies in D^+ that include only attributes of R_i.
     - All multivalued dependencies of the form $\alpha \twoheadrightarrow \beta \cap R_i$ where $\alpha \subseteq R_i$ and $\alpha \twoheadrightarrow \beta$ is in D^+.
   - A decomposition of schema R is dependency preserving with respect to a set D of functional and multivalued dependencies if for every set of relations $r_1(R_1), r_2(R_2), \ldots r_n(R_n)$ such that for all i, r_i satisfies D_i, there exists a relation $r(R)$ that satisfies D and for which $r_i = \Pi_{R_i}(r)$ for all i.

r_1 :

A	B
a_1	b_1
a_2	b_1

r_2 :

C	G	H
c_1	g_1	h_1
c_2	g_2	h_2

r_3 :

A	I
a_1	i_1
a_1	i_2

r_4 :

A	C	G
a_1	c_1	g_1
a_2	c_2	g_2

Figure 7.9: Projection of relation r onto a 4NF decomposition of R.

10. What does this formal statement say? It says that a decomposition is dependency preserving if for every set of relations on the decomposition schema satisfying only the restrictions on D there exists a relation r on the entire schema R that the decomposed schemas can be derived from, and that r also satisfies the functional and multivalued dependencies.

11. We'll do an example using our decomposition algorithm and check the result for dependency preservation.

    - Let $R = (A, B, C, G, H, I)$.
    - Let D be
      $A \twoheadrightarrow B$
      $B \twoheadrightarrow HI$
      $CG \rightarrow H$
    - R is not in 4NF, as we have $A \twoheadrightarrow B$ and A is not a superkey.
    - The algorithm causes us to decompose using this dependency into
      $R_1 = (A, B)$
      $R_2 = (A, C, G, H, I)$
    - R_1 is now in 4NF, but R_2 is not.
    - Applying the multivalued dependency $CG \twoheadrightarrow H$ (how did we get this?), our algorithm then decomposes R_2 into
      $R_3 = (C, G, H)$
      $R_4 = (A, C, G, I)$
    - R_3 is now in 4NF, but R_4 is not.
    - Why? As $A \twoheadrightarrow HI$ is in D^+ (why?) then the restriction of this dependency to R_4 gives us $A \twoheadrightarrow I$.
    - Applying this dependency in our algorithm finally decomposes R_4 into
      $R_5 = (A, I)$
      $R_6 = (A, C, G)$
    - The algorithm terminates, and our decomposition is R_1, R_3, R_5 and R_6.

12. Let's analyze the result.

    - This decomposition is not dependency preserving as it fails to preserve $B \twoheadrightarrow HI$.
    - Figure 7.9 (textbook 6.14) shows four relations that may result from projecting a relation onto the four schemas of our decomposition.
    - The restriction of D to (A, B) is $A \twoheadrightarrow B$ and some trivial dependencies.
    - We can see that r_1 satisfies $A \twoheadrightarrow B$ as there are no pairs with the same A value.
    - Also, r_2 satisfies all functional and multivalued dependencies since no two tuples have the same value on any attribute.
    - We can say the same for r_3 and r_4.
    - So our decomposed version satisfies all the dependencies in the restriction of D.
    - However, there is no relation r on (A, B, C, G, H, I) that satisfies D and decomposes into r_1, r_2, r_3 and r_4.
    - Figure 7.10 (textbook 6.15) shows $r = r_1 \bowtie r_2 \bowtie r_3 \bowtie r_4$.

A	B	C	G	H	I
a_1	b_1	c_1	g_1	h_1	i_1
a_2	b_1	c_2	g_2	h_2	i_2

Figure 7.10: A relation $r(R)$ that does not satisfy $B \twoheadrightarrow HI$.

- Relation r does not satisfy $B \twoheadrightarrow HI$.
- Any relation s containing r and satisfying $B \twoheadrightarrow HI$ must include the tuple $(a_2, b_1, c_2, g_2, h_1, i_1)$.
- However, $\Pi_{CGH}(s)$ includes a tuple (c_2, g_2, h_1) that is not in r_2.
- Thus our decomposition fails to detect a violation of $B \twoheadrightarrow HI$.

13. We have seen that if we are given a set of functional and multivalued dependencies, it is best to find a database design that meets the three criteria:
    - 4NF.
    - Dependency Preservation.
    - Lossless-join.
14. If we only have functional dependencies, the first criteria is just BCNF.
15. We cannot always meet all three criteria. When this occurs, we compromise on 4NF, and accept BCNF, or even 3NF if necessary, to ensure dependency preservation.

7.5 Normalization Using Join Dependencies (not to be covered)

We will omit this section.

7.6 Domain-Key Normal Form (not to be covered)

We will omit this section.

7.7 Alternative Approaches to Database Design (not to be covered)

1. We have taken the approach of starting with a single relation schema and decomposing it.
    - One goal was lossless-join decomposition.
    - For that, we decided we needed to talk about the join of all relations on the decomposed database.
    - Figure 6.20 shows a *borrow* relation decomposed in PJNF, where the loan amount is not yet determined.
    - If we compute the natural join, we find that all tuples referring to loan number 58 disappear.
    - In other words, there is no *borrow* relation corresponding to the relations of figure 6.20.
    - We call the tuples that disappear when the join is computed **dangling tuples**.
    - Formally, if $r_1(R_1), r_2(R_2), \ldots r_n(R_n)$ are a set of relations, a tuple t of relation r_i is a dangling tuple if t is not in the relation
    $$\Pi_{R_i}(r_1 \bowtie r_2 \bowtie \ldots \bowtie r_n)$$
    - Dangling tuples may occur in practical applications.
    - They represent incomplete information.

- The relation $(r_1 \bowtie r_2 \bowtie \ldots \bowtie r_n)$ is called a **universal relation** since it involves all the attributes in the universe defined by $R_1 \cup R_2 \cup \ldots \cup R_n$.
- The only way to write a universal relation for our example is include null values.
- Because of the difficulty in managing null values, it may be desirable to view the decomposed relations as representing the database rather than the universal relation.
- We still might need null values if we tried to enter a loan number without a customer name, branch name, or amount.
- In this case, a particular decomposition defines a restricted form of incomplete information that is acceptable in our database.

2. The normal forms we have defined generate good database design from the point of view of representation of incomplete information.
   - We need a loan number to represent any information in our example.
   - We do not want to store data for which the key attributes are unknown.
   - The normal forms we have defined do not allow us to do this unless we use null values.
   - Thus our normal forms allow representation of acceptable incomplete information via dangling tuples while prohibiting the storage of undesirable incomplete information.
3. Another point in our method of design is that attribute names must be unique in the universal relation.
   - We call this the **unique role assumption**.
   - If we defined the relations
     branch-loan(name, number)
     loan-customer(number, name)
     loan(number, amount)
     expressions like *branch-loan* $\bowtie$ *loan-customer* are possible but meaningless.
   - In SQL, there is no natural join operation, and so references to names are disambiguated by prefixing relation names.
   - In this case, non-uniqueness might be both convenient and allowed.
   - The unique role assumption is generally preferable, and if it is not made, special care must be taken when constructing a normalized design.