# CHAPTER 5

# Database Design (DBD) Toolkit

This chapter introduces a toolkit written in SWI-Prolog (a freely available Prolog system for the Unix as well as the Windows operating systems) that allows the student to learn and test out the various concepts, definitions, and algorithms associated with relational database design using functional dependency theory.

The DBD system is available as SWI-Prolog source code in a single file, `dbd.pl`. This file needs to be consulted (read into the Prolog interpreter's memory) before invoking any of the DBD predicates.

## 5.1 Coding Relational Schemas and Functional Dependencies

The DBD system uses lists to code the relational schemas and functional dependencies. Variables in Prolog begin with an upper-case letter and hence attributes of relational schemes are coded as Prolog constants – terms that begin with a lower-case letter. Consider the following relational schema and associated set of functional dependencies:

```
R(A,B,C,D,E)
F = { A → B, AC → D, E → AB, AD → E, ABC → D }
```

These will be coded as the following Prolog predicates in DBD:

```
schema([a,b,c,d,e]).
fds([ [[a],[b]], [[a,c],[d]], [[e],[a,b]],
      [[a,d],[e]], [[a,b,c],[d]] ]).
```

## 5.2 Invoking the SWI-Prolog Interpreter

Once the SWI-Prolog system is installed in your server or local computer, the interpreter is invoked using the following terminal command:

$ **pl**

Here `$` is the command prompt. In the rest of the chapter, all user inputs are shown in **bold** font and system displays in regular font.

After printing a welcome message, the interpreter responds with the following prompt:

```
?-
```

At this prompt, the user may enter a Prolog query to invoke a predicate. In order to invoke one of the predicates of the DBD system, the user must first load Prolog interpreter's memory with the DBD program using the following command/query:

```
?- ['dbd.pl'].
```

Note that all commands/queries to Prolog end with a period (".") symbol. To exit the Prolog system, the halt command/query is entered at the Prolog prompt as follows:

```
?- halt.
```

To solve a particular problem using the DBD system, the student will usually create a file containing the code to represent the relational schema and the functional dependencies. In addition, one or more Prolog rules may be included to execute predicates to solve the problem at hand. For example, consider the problem of finding one or more candidate keys for the relational schema and functional dependencies mentioned earlier in the section. The student will create a text file with the following code:

```
schema([a,b,c,d,e]).
fds([ [[a],[b]], [[a,c],[d]], [[e],[a,b]],
      [[a,d],[e]], [[a,b,c],[d]] ]).
answer(K) :- schema(R), fds(F), candkey(R,F,K).
```

The first two predicates (`schema` and `fds`) code the relational schema and functional dependencies. The third rule invokes the `candkey` predicate of the DBD system to find the candidate keys for `R` and `F`. The answer is captured in the `answer` predicate with one argument. Let us assume that the above code is present in a file called `example.pl`. The following interaction with the SWI-Prolog system shows the steps necessary to find the answer(s) to the problem:

```
$ pl
Welcome to SWI-Prolog (Multi-threaded, Version 5.4.7)
Copyright (c) 1990-2003 University of Amsterdam.
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software,
and you are welcome to redistribute it under certain conditions.
Please visit http://www.swi-prolog.org for details.

For help, use ?- help(Topic). or ?- apropos(Word).

?- ['dbd.pl'].
% dbd.pl compiled 0.02 sec, 30,740 bytes

Yes
?- ['example.pl'].
% example.pl compiled 0.00 sec, 1,116 bytes

Yes
?- answer(K).

K = [c, e] ;
```

```
K = [a, c] ;

No
?- answer(K).

K = [c, e]

Yes
?- halt.
$
```

The user inputs are shown in **bold** font. After invoking the interpreter, the user first loads the DBD system code (`dbd.pl`). The user then loads the code to solve the problem at hand (`example.pl`). Finally, the user submits the query, **answer(K)**, to determine the candidate keys. By default, every Prolog system responds with the first answer to the query and waits for user input. At this point the user has two choices: press the `ENTER` key to stop looking at further answers to the query or press the semi-colon key to request Prolog to display the next answer to the query. In the above screen capture, both options are shown.

## 5.3 DBD System Predicates

The DBD system predicates are discussed in this section. Almost all of the predicates take as input the relation schema R and a set of functional dependencies F. R and F must be coded properly as indicated in earlier sections. In particular, the attribute names must begin with a lower-case letter. The predicate names themselves also must begin with a lower-case letter.

### 5.3.1 xplus(R,F,X,Xplus)

The `xplus` predicate takes as input a relation schema `R`, a set of functional dependencies `F`, and a subset `X` of `R` and produces as output the attribute-closure of `X` under `F`, i.e. the set of attributes that are functionally dependent on `X`. The following shows a sample interaction with the Prolog system to compute the attribute closure of `{A, C}` for the relational schema and functional dependencies mentioned earlier in this chapter (the file `e2.pl` is assumed to contain the `schema` and `fds` predicates):

```
?- ['dbd.pl'].
% dbd.pl compiled 0.02 sec, 30,740 bytes

Yes
?- ['e2.pl'].
% e2.pl compiled 0.00 sec, 1,096 bytes

Yes
?- schema(R),fds(F),xplus(R,F,[a,c],Xplus).

R = [a, b, c, d, e]
F = [[[a], [b]], [[a, c], [d]], [[e], [a, b]], [[a, d], [e]], [[a, b, c], [d]]]
Xplus = [a, b, c, d, e]
```

```
Yes
?- halt.
$
```

### 5.3.2 finfplus(R,F,[X,Y])

The `finfplus` predicate takes as input a relation schema `R`, a set of functional dependencies `F`, and a functional dependency `X → Y` (in list coded form `[X,Y]`) and returns "yes" if `X → Y` is in the closure of `F` and "no" otherwise. An example using the same `R` and `F` as before follows:

```
?- ['dbd.pl'].
% dbd.pl compiled 0.02 sec, 30,740 bytes

Yes
?- ['e2.pl'].
% e2.pl compiled 0.00 sec, 1,096 bytes

Yes
 ?- schema(R), fds(F), finfplus(R,F,[[a,b],[c]]).

No
?- schema(R), fds(F), finfplus(R,F,[[a,d],[b]]).

R = [a, b, c, d, e]
F = [[[a], [b]], [[a, c], [d]], [[e], [a, b]], [[a, d], [e]], [[a, b, c], [d]]]

Yes
```

In the above screen capture we observe that the functional dependency `AB → C` is not in the closure of `F` whereas the functional dependency `AD → B` is in the closure of `F`.

### 5.3.3 fplus(R,F,Fplus)

The `fplus` predicate takes as input a relational schema `R` and a set of functional dependencies `F` and returns as output the closure of `F` in the third parameter. Typically the number of functional dependencies in the closure of `F` is large and in the worst case can be exponential in the number of attributes of `R`.

Consider the following Prolog code in file `e3.pl`:

```
schema([a,b,c,d,e]).
fds([ [[a],[b]], [[a,c],[d]], [[e],[a,b]],
      [[a,d],[e]], [[a,b,c],[d]] ]).
displayFDs([]) :- nl.
displayFDs([[X,Y]|Rest]) :-
  write(X),write('-->'), write(Y),nl,displayFDs(Rest).
go :- schema(R), fds(F), fplus(R,F,Fplus), displayFDs(Fplus).
```

This program adds the display rules to the `e2.pl` file and invokes the `displayFDs` predicate on the closure of `F`. All functional dependencies that are implied by `F` are displayed by the following invocation:

```
?- ['dbd.pl'].
% dbd.pl compiled 0.02 sec, 30,740 bytes

Yes
?- ['e3.pl'].
% e3.pl compiled 0.01 sec, 1,700 bytes

Yes
?- go.
[a]-->[a, b]
[a]-->[b]
...
...
```

**Note**: The `fplus` predicate does not generate any trivial functional dependencies (dependencies whose RHS attributes are also present in the LHS attributes).

### 5.3.4 implies(R,F1,F2) and equiv(R,F1,F2)

The `implies` predicate takes as input a relational schema `R` and two sets of functional dependencies `F1` and `F2`. It returns "yes" if `F1` implies `F2`, i.e. if each functional dependency in `F2` is in the closure of `F1`; returns "no" otherwise.

The `equiv` predicate takes as input a relational schema `R` and two sets of functional dependencies `F1` and `F2`. It returns "yes" if `F1` is equivalent to `F2`, i.e. if each functional dependency in `F2` is in the closure of `F1` and if each functional dependency in `F1` is in the closure of `F2`; returns "no" otherwise.

Consider the relational schema `R(A,B,C)` and the two sets of functional dependencies:

```
F1 = {A →BC, B →A}
F2 = {A →B, A →C}
```

The above schema and functional dependencies can be coded into a file `e4.pl` as follows:

```
schema([a,b,c]).
fds1([[[a],[b,c]], [[b],[a]]]).
fds2([[[a],[b]], [[a],[c]]]).
```

The following screen capture illustrates the `implies` and `equiv` predicates:

```
?- ['dbd.pl'].
% dbd.pl compiled 0.02 sec, 30,740 bytes
```

```
Yes
?- ['e4.pl'].
% e4.pl compiled 0.00 sec, 1,120 bytes

Yes
?- schema(R), fds1(F1), fds2(F2), implies(R,F1,F2).

R = [a, b, c]
F1 = [[[a], [b, c]], [[b], [a]]]
F2 = [[[a], [b]], [[a], [c]]]

Yes
?- schema(R), fds1(F1), fds2(F2), implies(R,F2,F1).

No
?- schema(R), fds1(F1), fds2(F2), equiv(R,F1,F2).

No
?-
```

### 5.3.5 superkey(R,F,K) and candkey(R,F,K)

The `superkey` and `candkey` predicates both take a relation schema `R` and a set of functional dependencies as input in the first two parameters. The third parameter may be a constant (i.e. an instantiated list of attributes of `R`) or a variable.

In case the third parameter is instantiated to a list of attributes the `superkey` (`candkey`) predicate with return "yes" if the instantiated third parameter is a super key (candidate key) of R; "no" otherwise.

In case the third parameter is a variable, the predicate `superkey` (`candkey`) will generate all super keys (candidate keys) one at a time.

Consider the following `e5.pl` file containing a relation schema and associated functional dependencies:

```
schema([a,b,c]).
fds([[[a],[b,c]], [[b],[a]]]).
```

The following screen capture illustrates usage of the `superkey` and `candkey` predicates:

```
?- ['dbd.pl'].
% dbd.pl compiled 0.01 sec, 30,428 bytes

Yes
?- ['e5.pl'].
% e5.pl compiled 0.00 sec, 1,124 bytes

Yes

?- schema(R), fds(F), candkey(R,F,K).
```

```
R = [a, b, c]
F = [[[a], [b, c]], [[b], [a]]]
K = [b] ;

R = [a, b, c]
F = [[[a], [b, c]], [[b], [a]]]
K = [a] ;

No
?- schema(R), fds(F), superkey(R,F,K).

R = [a, b, c]
F = [[[a], [b, c]], [[b], [a]]]
K = [a, b, c] ;

R = [a, b, c]
F = [[[a], [b, c]], [[b], [a]]]
K = [b, c] ;

R = [a, b, c]
F = [[[a], [b, c]], [[b], [a]]]
K = [b] ;

R = [a, b, c]
F = [[[a], [b, c]], [[b], [a]]]
K = [a, c] ;

R = [a, b, c]
F = [[[a], [b, c]], [[b], [a]]]
K = [a] ;

R = [a, b, c]
F = [[[a], [b, c]], [[b], [a]]]
K = [a, b] ;

No
?- schema(R), fds(F), candkey(R,F,[a,b]).

No
?- schema(R), fds(F), superkey(R,F,[a,b]).

R = [a, b, c]
F = [[[a], [b, c]], [[b], [a]]]

Yes
?-
```

## 5.3.6 mincover(R,F,FC)

The `mincover` predicate takes as input a relation schema `R` and a set of functional dependencies `F` and returns as output in the third parameter the minimal cover for `F`.

Consider the following relation schema and functional dependencies in file `e6.pl`:

```
schema([a,b,c,d,e]).
fds([[[a],[b,c]], [[b],[a]], [[a,b],[d]], [[a],[d]], [[b,c],[a]]]).
```

The functional dependencies coded in the file are:

F = {A → BC, B → A, AB → D, A → D, BC → A}

The following screen capture illustrates the `mincover` predicate usage:

```
?- ['dbd.pl'].
% dbd.pl compiled 0.02 sec, 30,740 bytes

Yes
?- ['e6.pl'].
% e6.pl compiled 0.00 sec, 1,072 bytes

Yes
?- schema(R),fds(F),mincover(R,F,MinF).

R = [a, b, c, d, e]
F = [[[a], [b, c]], [[b], [a]], [[a, b], [d]], [[a], [d]], [[b, c], [a]]]
MinF = [[[a], [b, c, d]], [[b], [a]]]

Yes
?-
```

The minimum cover for the set of functional dependencies is

MinF = {A → BCD, B → A}.

## 5.3.7 ljd(R,F,R1,R2), ljd(R,F,D), and fpd(R,F,D)

The predicates discussed in this sub-section are related to decompositions of relation schemas. We represent decompositions using lists of relational schemas. For example, the decomposition D = {ABC, ABD, BDE} of R = ABCDE is represented as the following Prolog term:

```
D = [[a,b,c],[a,b,d],[b,d,e]]
```

**Testing for lossless join decomposition**

There are two versions of the `ljd` predicate.

The first version of the `ljd` predicate takes as input a relation scheme `R`, a set of functional dependencies `F`, and two relation schemas `R1` and `R2` that are decompositions of `R`. The predicate returns "yes" if `(R1, R2)` is a lossless join decomposition of `R`; "no" otherwise.

As an example, consider the R = ABC and F = {A → B}. Consider the decomposition of R into two sub-schemas R1 = AB and R2 = AC. The following screen capture illustrates the use of the first version of ljd predicate:

```
?- ['dbd.pl'].
% dbd.pl compiled 0.02 sec, 30,740 bytes

Yes
?- ljd([a,b,c],[[[a],[b]]],[a,b],[a,c]).

Yes
?-
```

The second version of the ljd predicate takes as input a relation scheme R, a set of functional dependencies F, and a decomposition D into two or more sub-schemas. D is represented as a list of relation schemas where each schema is itself a list of attributes. The predicate returns "yes" if D is a lossless join decomposition of R; "no" otherwise. This version implements the "matrix" method (Algorithm 11.1 of the Elmasri/Navathe text) to test for lossless join and prints the final matrix.

As an example, consider the following file (named e7a.pl) containing a relational schema, a set of functional dependencies and a decomposition into more than two sub-schemas:

```
schema([a,b,c,d,e]).
fds([[[a],[c]], [[b],[c]], [[c],[d]], [[d,e],[c]], [[c,e],[a]]]).
decomp([[a,d],[a,b],[b,e],[c,d,e],[a,e]]).
```

The following screen capture illustrates the usage of the more general version of the ljd predicate:

```
?- ['dbd.pl'].
% dbd.pl compiled 0.02 sec, 30,740 bytes

Yes
?- ['e7a.pl'].
% e7a.pl compiled 0.01 sec, 1,440 bytes

Yes
?- schema(R), fds(F), decomp(D), ljd(R,F,D).
[[a, 1], [b, 1, 2], [a, 3], [a, 4], [b, 1, 5]]
[[a, 1], [a, 2], [a, 3], [a, 4], [b, 2, 5]]
[[a, 1], [a, 2], [a, 3], [a, 4], [a, 5]]
[[a, 1], [b, 4, 2], [a, 3], [a, 4], [a, 5]]
[[a, 1], [b, 5, 2], [a, 3], [a, 4], [a, 5]]


R = [a, b, c, d, e]
F = [[[a], [c]], [[b], [c]], [[c], [d]], [[d, e], [c]], [[c, e], [a]]]
D = [[a, d], [a, b], [b, e], [c, d, e], [a, e]]

Yes
?-
```

Note that the `ljd` predicate execution prints the final matrix to the screen; In this case it can be observed that the third row is turned into an "all-a" row.

**Testing for FD-preserving decomposition**

The `fpd` predicate takes as input a relation scheme `R`, a set of functional dependencies `F`, and a decomposition `D` into two or more sub-schemas. `D` is represented as a list of relation schemas where each schema is itself a list of attributes. The predicate returns "yes" if `D` is a functional dependency preserving decomposition of `R`; "no" otherwise.

As an example, consider the following file (named `e7b.pl`) containing a relational schema, a set of functional dependencies and a decomposition into sub-schemas:

```
schema([a,b,c,d]).
fds([[[a],[b]], [[b],[c]], [[c],[d]], [[d],[a]]]).
decomp([[a,b],[b,c],[c,d]]).
```

The following screen capture illustrates the usage of the `fpd` predicate:

```
?- ['dbd.pl'].
% dbd.pl compiled 0.03 sec, 30,740 bytes

Yes
?- ['e7b.pl'].
% e7b.pl compiled 0.00 sec, 1,260 bytes

Yes
?- schema(R), fds(F), decomp(D), fpd(R,F,D).
Considering [a]-->[b]
Xplus=[a, b, c, d]
Considering [b]-->[c]
Xplus=[a, b, c, d]
Considering [c]-->[d]
Xplus=[a, b, c, d]
Considering [d]-->[a]
Xplus=[a, b, c, d]

R = [a, b, c, d]
F = [[[a], [b]], [[b], [c]], [[c], [d]], [[d], [a]]]
D = [[a, b], [b, c], [c, d]]

Yes
?-
```

Note that the `fpd` predicate execution prints each functional dependency in `F` and displays the attribute closure of the LHS of the functional dependency with respect to the "projected" functional dependencies. The algorithm implemented in this predicate is from "*Jeffrey D. Ullman, Principles of Database Systems, Second Edition, Computer Science Press, 1982*".

### 5.3.8 is3NF(R,F) and threenf(R,F,D)

The `is3NF` predicate takes as input a relation schema `R` and a set of associated functional dependencies and returns "yes" if the relation schema is in 3NF; "no" otherwise.

The `threenf` predicate takes as input a relation schema `R` and a set of associated functional dependencies and returns in the third parameter a decomposition of `R` into 3NF. The `threenf` predicate implements Algorithm 11.4 of the Elmasri/Navathe text.

As an example, consider the following file (named `e8.pl`) containing a relational schema and a set of functional dependencies:

```
schema([a,b,c,d]).
fds([[[a,b],[c]],[[b],[d]],[[b,c],[a]]]).
```

The following screen capture illustrates the 3NF test as well as a 3NF decomposition of the relation schema.

```
?- ['dbd.pl'].
% dbd.pl compiled 0.02 sec, 30,740 bytes

Yes
?- ['e8.pl'].
% e8a.pl compiled 0.01 sec, 932 bytes

Yes
?- schema(R), fds(F), is3NF(R,F).

No
?- schema(R), fds(F), threenf(R,F,R3NF).

R = [a, b, c, d]
F = [[[a, b], [c]], [[b], [d]], [[b, c], [a]]]
R3NF = [[a, b, c], [b, d]]

Yes
?-
```

As can be seen in the above interaction with the DBD system, the schema `R = ABCD` is not in 3NF and is decomposed into 3NF sub-schemas `ABC` and `BD`.

### 5.3.9 isBCNF(R,F) and bcnf(R,F,D)

The `isBCNF` predicate takes as input a relation schema `R` and a set of associated functional dependencies and returns "yes" if the relation schema is in BCNF; "no" otherwise.

The `bcnf` predicate takes as input a relation schema `R` and a set of associated functional dependencies and returns in the third parameter a decomposition of `R` into BCNF. The `bcnf` predicate implements Algorithm 11.3 of the Elmasri/Navathe text.

As an example, consider the same `e8.pl` file used in the previous sub-section.

The following screen capture illustrates the BCNF test as well as a BCNF decomposition of the relation schema.

```
?- ['dbd.pl'].
% dbd.pl compiled 0.02 sec, 30,740 bytes

Yes
?- ['e8.pl'].
% e8.pl compiled 0.00 sec, 928 bytes

Yes
?- schema(R), fds(F), isBCNF(R,F).

No
?- schema(R), fds(F), bcnf(R,F,D).
Scheme to decompose = [a, b, c, d] Offending FD: [b]-->[d]
Final Result is:
[a, b, c]
[b, d]

R = [a, b, c, d]
F = [[[a, b], [c]], [[b], [d]], [[b, c], [a]]]
D = [[a, b, c], [b, d]];
```

As can be seen in the above interaction with the DBD system, the schema `R = ABCD` is not in BCNF and is decomposed into BCNF sub-schemas `ABC` and `BD`. Note that the `bcnf` predicate implementation also prints out the schema that is being decomposed along with the "offending" functional dependency.

# Exercises

For all the problems in this set of exercises, use the predicates in the DBD system to answer the questions.

**Chapter 10 (Elmasri/Navathe Text) Problems:**

1. Consider the following two sets of functional dependencies:

   $F = \{A \rightarrow C,\ AC \rightarrow D,\ E \rightarrow AD,\ E \rightarrow H\}$
   $G = \{A \rightarrow CD,\ E \rightarrow AH\}$

   Check whether they are equivalent.

2. Consider the relation schema

   `EMP_DEPT(ename,ssn,bdate,address,dnumber,dname,dmgrssn)`

and the following set G of functional dependencies on EMP_DEPT:

```
ssn → ename, bdate, address, dnumber
dnumber → dname, dmgrssn
```

Calculate the closures {ssn}$^+$ and {dnumber}$^+$.

3. Is the set of functional dependencies G in Exercise 2 minimal? If not, find a minimal set of functional dependencies that is equivalent to G.

4. Consider the universal relation R = { A,B,C,D,E,F,G,H,I} and the set of functional dependencies:

   ```
   F = {AB → C, A → DE, B → F, F → GH, D → IJ}
   ```

   What is the key for R? Decompose R into 3NF relations.

5. Repeat Exercise 4 for the same R but a different set of functional dependencies

   ```
   G = {AB → C, BD → EF, B → F, F → GH, D → IJ}
   ```

6. Consider a relation R(A,B,C,D,E) with the following functional dependencies:

   ```
   F = {AB → C, CD → E, DE → B}
   ```

   Is AB a candidate key of this relation? If not, is ABD?

7. Consider the relation R, which has attributes that hold schedules of courses and sections at a university;

   ```
   R = {courseno,secno,offeringdept,credithours,courselevel,
        instructorssn,semester,year, Dayshours,roomno,
        noofstudents}
   ```

   Suppose the following functional dependencies hold on R:

   ```
   courseno → offeringdept,credithours,courselevel

   courseno,secno,semester,year →
             dayshours,roomno,noofstudents,instructorssn

   roomno,dayshours,semester,year →
             instructorssn,courseno,secno
   ```

   Find the candidate keys for R and decompose the relation into 3NF relations.

**Chapter 11 (Elmasri/Navathe Text) Problems:**

8. Consider the relation schema

   LOTS(propertyid,countyname,lotno,area,price,taxrate)

   and associated set of functional dependencies:

   propertyid → countyname,lotno,area,price,taxrate
   countyname,lotno → propertyid,area,price,taxrate
   countyname → taxrate
   area → price

   Determine if the decomposition of LOTS into

   LOTS1AX(propertyis,area,lotno)
   LOTS1AY(area,countyname)
   LOTS1B(area,price)
   LOTS2(countyname,taxrate)

   is a lossless join decomposition?

9. Consider the universal relation R = { A,B,C,D,E,F,G,H,I} and the set of functional dependencies:

   F = {AB → C, A → DE, B → F, F → GH, D → IJ}

   Determine the candidate keys for R, find a minimal cover for F, and decompose R into 3NF relations.

10. Consider the universal relation R = { A,B,C,D,E,F,G,H,I} and the set of functional dependencies:

    G = {AB → C, BD → EF, B → F, F → GH, D → IJ}

    Determine the candidate keys for R, find a minimal cover for F, and decompose R into 3NF relations.

11. Consider the universal relation R = { A,B,C,D,E,F,G,H,I} and the set of functional dependencies:

    F = {AB → C, A → DE, B → F, F → GH, D → IJ}

    Decompose R into BCNF relations. Repeat the exercise for

    G = {AB → C, BD → EF, B → F, F → GH, D → IJ}

12. Consider a relation R(A,B,C,D,E) with the following functional dependencies:

    F = {AB → C, CD → E, DE → B}

    Determine the candidate keys for R and decompose R into 3NF relations and into BCNF relations.

    Repeat the exercise for

    R = {courseno,secno,offeringdept,credithours,courselevel,
         instructorssn,semester,year, Dayshours,roomno,
         noofstudents}

    and F containing the following functional dependencies:

    courseno → offeringdept,credithours,courselevel,

    courseno,secno,semester,year →
              dayshours,roomno,noofstudents,instructorssn

    roomno,dayshours,semester,year →
              instructorssn,courseno,secno

13. Consider the following decompositions for the relation schema

        R(a,b,c,d,e,f,g,h,i)

    and the set of associated functional dependencies:

        F = {ab → c, a → de, b → f, f → gh, d → ij}.

    Determine whether each of the following decompositions has the (a) dependency preservation property, and (b) the lossless join property with respect to F:

i. D1 = {R1,R2,R3,R4,R5};
   R1 = {a,b,c}, R2 = {a,d,e}, R3 = {b,f}, R4 = {f,g,h},
   R5 = {d,i,j}.

ii. D2 = {R1,R2,R3};
    R1 = {a,b,c,d,e}, R2 = {b,f,g,h}, R3 = d,i,j}

iii. D3 = {R1,R2,R3,R4,R5};
     R1 = {a,b,c,d}, R2 = {d,e}, R3 = {b,f}, R4 = {f,g,h},
     R5 = {d,i,j}.

14. Consider the relation REFRIG(modelno,year,price,manufplant,color) which is abbreviated as REFRIG(m,y,p,mp,c) and the set of functional dependencies

    F = {m → mp, {m,y} → p, mp → c}

iv. Evaluate each of the following as candidate keys: {m}, {m,y}, {m,c}.
v. Test if REFRIG is in 3NF? In BCNF?
vi. Test if the decomposition D = {R1(m,y,p), R2(m,mp,c)} is lossless.