CHAPTER 8

Normalization

We are doing pretty well at designing a database. So far, you have learned how use cases and a data model can help you understand many of the complexities of the problem you are trying to represent. In the previous chapter, you saw how to represent the main parts of the data model in a relational database. To recap:

- Each class is represented by a table.
- Each attribute is represented by a field with a datatype and possible constraints.
- Each object becomes a row in a table.
- For each table, we determine a primary key, which is a field(s) that uniquely identifies each row.
- We use the primary key field(s) to represent relationships between classes by way of foreign keys.

At this stage, everything could be absolutely fine, but then again there may be some classes in our model (or tables in our database) that might still cause us problems. Normalization is a formal way of checking the fields to ensure they are in the right table or to see if perhaps we might need restructured or additional tables to help keep our data accurate. The initial idea of normalization was first proposed by E. F. Codd [1] in 1970 and has been a cornerstone of relational database design since then. Some readers of this book may be throwing up their hands in horror that I have left this important topic until Chapter 8. However, we have actually been normalizing our database right from Example 1-1 when we saw that two classes were needed to keep information about plants and their uses.

In this chapter, we will first look at why it is critical that all the attributes are in the right table and how normalization helps us make sure they are.

Update Anomalies

Let's have a look at a simple example where having the attributes in the wrong table can cause a number of problems in maintaining data. Let's say we have a database for maintaining information about many different aspects of a company. There may be several tables for maintaining customers, products, orders, suppliers, and so on, and there are also two tables as shown in Figure 8-1 about employees and some small projects to which they have been assigned. Can you see a problem lurking in the `Assignment` table?

[1] Edgar F. Codd, (June 1970). "A Relational Model of Data for Large Shared Data Banks." *Communications of the ACM*: 13(6): pp. 377–387.

empID	last_name	first_name
1001	Smith	John
1005	Jones	Susan
1029	Li	Jane

Employee

emp	project_num	project_name	contact	hours
1005	1	JenningsLtd	325-1234	8
1001	3	ABCPromo	142-3456	8
1005	3	ABCPromo	142-3456	14
1001	6	Smith&Co	365-8765	20

Assignment

Figure 8-1. *Tables with potential update problems*

A problem with the `Assignment` table is one that we encountered way back in Example 1-3, "Insect Data." We have repeated information about a project. The number, name, and contact can be repeated several times in this table if there is more than one employee working on the project. This will almost inevitably lead to some rows (for, say, project number 3) having inconsistent names or contact numbers at some stage. This is relatively easy to spot for the data in Figure 8-1, but often it can be less easy to see. If we hadn't had data for two employees working on project 3, we might not have even realized this was a possibility. Normalization gives us a formal way of checking for such situations before we get into trouble.

As well as the possibility of inconsistent data, there are other problems that the design of the `Assignment` table can cause. These are often collectively referred to as *update anomalies*. We will look at some of these other problems now.

Insertion Problems

You will recall that it is necessary to have a primary key for every table in our database. This is so we can uniquely identify each row and have a mechanism for relating rows in different tables. What is a possible primary key for the `Assignment` table in Figure 8-1? Just looking at the data in the table, we can see that there is no single field that is a potential primary key field. Every column has duplicated values. We need to look for a concatenated key, and the pair `emp` and `project_num` is possible. We need to confirm that each employee is associated with a project just once, and if that is the case, the pair of values for `emp` and `proj_num` is a suitable primary key.

However, we have a problem. If we want to keep information about a particular project but there is no employee yet working on it, we have no value for `emp`, which is one of the fields making up our primary key. If a field is essential to uniquely determine a particular row in our table, it makes no sense that it can be empty. As you may recall from the previous chapter, one of the constraints imposed by putting a primary key on a table is that the fields involved must always have a value. We cannot enter a record for which the value of `emp`, being part of the primary key, is empty. Therefore we have no way of recording information about any project before someone is working on it.

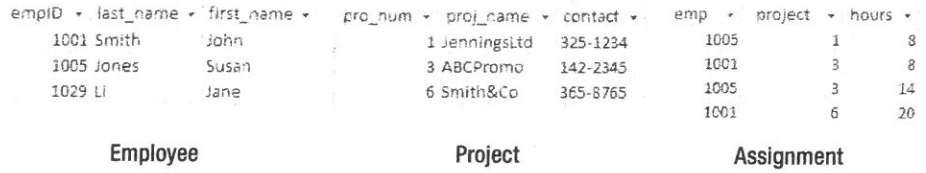
Deletion Problems

Here is another situation that might occur. Employee 1001 may finish working on the Smith&Co project. If this happens, we will remove that row from the `Assignment` table. What is a possible side effect of deleting this row? Well, if employee 1001 was the only person working on the project, every reference to Smith&Co will have gone, and we will have lost the project's contact number. By deleting information about employee 1001's involvement in a project, we have inappropriately lost information about the project.

Dealing With Update Anomalies

We have seen three different updating problems with the `Assignment` table in Figure 8-1: possible inconsistent data when repeated information is modified, problems inserting new records because part of the primary key may be empty, and accidental loss of information as a by-product of a deletion.

I'm sure you have spotted the solution to these problems ages ago. What we need is another table to record information about projects, as in Figure 8-2. With this design, we don't have a project's contact number recorded more than once, we can add a new project in the `Project` table even if no one is working on it, and we can delete an assignment (employee 1001 working on project 6) without accidentally losing information about the project.



empID	last_name	first_name
1001	Smith	John
1005	Jones	Susan
1029	Li	Jane

Employee

pro_num	proj_name	contact
1	JenningsLtd	325-1234
3	ABCPromo	142-2345
6	Smith&Co	365-8765

Project

emp	project	hours
1005	1	8
1001	3	8
1005	3	14
1001	6	20

Assignment

Figure 8-2. *Tables with update anomalies removed*

Chances are that the `Project` table would have surfaced in your original analysis of use cases and the data model. But how can you be sure you haven't missed anything? This is where the formal definition of a normalized table helps.

Functional Dependencies

Normalization helps us to determine whether our tables are structured in such a way as to avoid the update anomalies described in the previous section. Central to the definition of normalization is the idea of a *functional dependency*. Functional dependencies are a way of describing the interdependence of attributes or fields in our tables. With a definition of functional dependencies, we can provide a more formal definition of a primary key, explain what is meant by a normalized table, and discuss the different forms of normalization.

Definition of a Functional Dependency

A functional dependency is a statement that essentially says, "If I know the value for this attribute(s), I can uniquely tell you the value of some other attribute(s)." For example, we can say,

> If I know the value of an employee's ID number, I can tell you his last name with certainty.

Or equivalently,

> Employee's ID number functionally determines employee's last name.

Or in symbols,

`empID` → `last_name`

For the situation depicted in Figure 8-2, if I know an employee's ID is 1001, I can tell you that his last name is Smith. Does it work the other way round? If I know an employee's last name, can I uniquely tell you his employee ID number? From the data displayed in the tables, you might say, "Yes, you can." However, for a functional

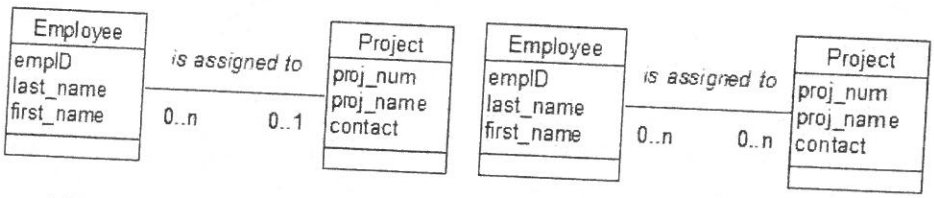
dependency to hold, it must be true for any data that can ever be put in our tables. We know that in the long term it is possible we might have several employees called Smith, so that knowing the last name does not uniquely determine the ID. Or more formally, `last_name` does not functionally determine `empID`.

Let's try another example. For the database tables in Figure 8-2, do we have a functional dependency between an employee's ID number and a project to which he is assigned? If I know the employee's ID is 1001, I cannot tell you a *unique* project number. It could be project 3 or project 6, and so an employee's ID number does not functionally determine (or uniquely determine) a project number.

Determining the functional dependencies requires us to understand the intricacies of the specific situation. For the case of employees and projects we need to know whether an employee can be assigned to only one project or whether he can be assigned to many different projects. Does this sound familiar? Determining whether attributes functionally determine each other involves the same sort of questions we went through when trying to understand the data model in Chapter 4.

In terms of a data model with an `Employee` class and a `Project` class, we would ask, "Can an employee ever be associated with more than one project?"

If the answer is "No," we have a 1-Many relationship between employees and projects as in Figure 8-3a; otherwise, we have a Many-Many relationship as in Figure 8-3b.



a) Employee is assigned to at most one project

b) Employee could be assigned to many projects

Figure 8-3. *Different relationships between Employee and Project*

In terms of functional dependencies, we have an analogous question: "If I know an employee's ID number, can I tell you a unique project number?"

If the answer is "Yes," `empID` →`proj_num`; otherwise, employee ID does not functionally determine the project number. Understanding the functional dependencies and understanding classes and their relationships are two different approaches to figuring out the intricacies of the problem we are trying to model.

Functional Dependencies and Primary Keys

Now that you know about functional dependencies, we have another way of thinking about what we mean by a primary key. If we know the values of the key fields of a table, we can find a unique row in the table. Once we have that row, then we know the value of all the other fields in that row. For example, if I know `empID`, I can find a unique row in the `Employee` table and so be able to determine the `last_name` and `first_name`. Or, in terms of functional dependencies:

```
empID → last_name, first_name
```

This leads us to a more formal way of defining a key:

The key fields functionally determine all the other fields in the table.

If I know the value of the key, I guarantee I can tell you the value of every other field in the row. This is why `last_name` cannot be a key field for our `Employee` table. If I know the last name of an employee is Smith, I cannot guarantee that I can find a single row and tell you the value for `empID`.

You have probably noticed that I've been using the term *key* rather than *primary key* in the last couple of paragraphs. There is a distinction between the two. Think about this. Is the pair of attributes (`empID`, `last_name`) a possible key for our `Employee` table? Our definition of a key is that if we know the value of the key fields, we can find a unique row. That is certainly the case if we know `empID` and `last_name`. However, I'm sure you can see that `last_name` is redundant. The pair of attributes (`empID`, `last_name`) is a key because `empID` is a key. If we know `empID`, we can find the row regardless of what additional information we have; we don't need to know `last_name` as well.

This idea of having fields in our key that are superfluous is the distinction between a key and a candidate for a primary key. To be considered as a primary key, there must be no unnecessary fields. More formally:

A primary key has no subset of the fields that is also a key.

Why is this important? Say each of our projects has one manager as shown in the snippet of the data model in Figure 8-4.

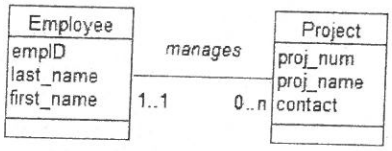


Figure 8-4. *A 1-Many relationship*

Remember how we represent a 1-Many relationship in our database. We take the primary key field(s) from the table at the 1 end (`Employee`) and put those field(s) as a foreign key in the `Project` table. If we had mistakenly used the pair (`empID`, `last_name`) as a primary key for the `Employee` table, we would get a `Project` table as shown in Figure 8-5. I'm sure you can see the information redundancy and potential for problems there.

foreign key fields

pro_num	proj_name	contact	manager_num	manager_name
1	JenningsLtd	325-1234	1005	Jones
3	ABCPromo	142-2345	1001	Smith
6	Smith&Co	365-8765	1001	Smith

Figure 8-5. *Redundancy problems caused by not having a suitable primary key*

Now that you have an idea of what a functional dependency is and a more formal definition of a primary key, we can look at how normalization can help ensure that we have a good design for our tables.

Normal Forms

Tables that are "normalized" will generally avoid the updating problems we examined earlier in the chapter. There are several levels of normalization called *normal forms*, each addressing additional situations where problems may occur. In this section we will look at the normal forms that are defined using functional dependencies.

First Normal Form

First normal form is the most important, and essentially says that we should not try to cram several pieces of data into a single field. Our very first example of what can go wrong, Example 1-1, "The Plant Database," was a situation where this was a problem. In the plant database, we were keeping information about different plant species and the different uses for which they were suited. Some possible (but not recommended!) ways of keeping several uses for each plant are shown in Figure 8-6.

plantID	genus	species	common_name	uses
1	Dodonaea	viscosa	Akeake	soil stability, hedging, shelter
2	Cedrus	atlantica	Atlas cedar	shelter
3	Alnus	glutinosa	Black alder	firewood, soil stability, shelter
4	Eucalyptus	nicholii	Black peppermint gum	shelter, coppicing, bird food

plantID	genus	species	common_name	use1	use2	use3
1	Dodonaea	viscosa	Akeake	shelter	hedging	soil stability
2	Cedrus	atlantica	Atlas cedar	shelter		
3	Alnus	glutinosa	Black alder	soil stability	shelter	firewood
4	Eucalyptus	nicholii	Black peppermint gum	shelter	coppicing	bird food

Figure 8-6. *Nonrecommended ways of keeping information about multiple uses*

We saw in Example 1-1 the problems that eventuate from keeping the plant data in tables like those in Figure 8-6. For example, it can be difficult to find all the plants with particular uses (e.g., all the shelter plants).

Thinking back to our new definition of a primary key, let's reconsider the primary keys of the two tables in Figure 8-6. `plantID` is a primary key of both tables in the sense that it is different in every row. Does it functionally determine all the other attributes? If I know the value of `plantID` (e.g., `plantID = 1`), can I tell you a unique use? Well, in the top table I can tell you the character string in the `uses` field, and in the second table I can tell you what is in any particular one of the three columns, so in a very formal sense, yes, I can. However, if we are thinking about the meanings behind these fields, I can't give you any information about a unique use just by knowing the plant's ID. I can only tell you about a collection of uses for each plant.

The two tables are not in first normal form (except in a technical sense). They are both trying in a roundabout way to keep multiple values of use.

A table is not in first normal form if it is keeping multiple values for a piece of information.

Normalization has given us a formal way of determining that there is something wrong with the design of the tables in Figure 8-6. It also gives us a method for solving the problem.

If a table is not in first normal form, remove the multivalued information from the table. Create a new table with that information and the primary key of the original table.

For our plant database example, this means setting up two tables, as in Figure 8-7.

plantID	genus	species	common_name
1	Dodonaea	viscosa	Akeake
2	Cedrus	atlantica	Atlas cedar
3	Alnus	glutinosa	Black alder
4	Eucalyptus	nicholii	Black peppermint gum
5	Juglans	nigra	Black walnut

Plant

plant	use
1	soil stability
1	hedging
1	shelter
2	shelter
3	firewood
3	soil stability
3	shelter

PlantUse

Figure 8-7. *Removing the multivalued field from unnormalized table to create an additional table*

When we considered this problem by way of a data model, we decided that we actually had two classes, `Plant` and `Use`, with a Many-Many relationship between them. In Chapter 7, you saw that to represent a Many-Many relationship, we needed to add an intermediate table. If you go back and take a look at Figures 7-17 to 7-19, you will see that the new table is the same as the `PlantUse` table in Figure 8-7. We arrived at the same normalized solution, but via two different routes. As discussed in Example 1-1, normalized tables such as those in Figure 8-7 avoid the many problems associated with the original, unnormalized tables of Figure 8-6.

Second Normal Form

It is possible for a table in first normal form to still have updating problems. The `Assignment` table in Figure 8-8, which we discussed at the beginning of this chapter, is an example. It has the information about the names and contacts of projects repeated several times, with the result that eventually the information might become inconsistent. We also saw that there could be problems with inserting new records and losing information as a by-product of deleting certain records.

empID	project_num	project_name	contact	hours
1005	1	JenningsLtd	325-1234	8
1001	3	ABCPromo	142-3456	8
1005	3	ABCPromo	142-3456	14
1001	6	Smith&Co	365-8765	20

Figure 8-8. *Assignment table with update anomalies*

The definition of both first and second normal form requires us to know the primary key of the table we are assessing. The primary key of the `Assignment` table is the combination of the `empID` and `proj_num` fields. Is the table in first normal form? If I tell you an employee ID and a project number (e.g., 1005 and 1), can you tell me unique values for all the other non-key fields? Yes. The project name is Jennings Ltd, the contact is 325-1234, and the hours are 8. There are no multivalued fields in this table. We are not trying to squeeze several bits of information into one field anywhere. But there is still a problem with update anomalies.

The problem here is that while I can figure out the value of all the non-key fields by knowing the primary key, I don't actually need both fields of the primary key to do that. If I want to know the number of hours, I need to know the values of both `empID` and `proj_num`. However, if I want to know the contact number or the project name, I only need to know the value of the `proj_num`. Here is where our problem arises, and it leads us to the definition of second normal form.

> *A table is in second normal form if it is in first normal form AND we need ALL the fields in the key to determine the values of the non-key fields.*

We also have a way of fixing a table that is not in second normal form.

> *If a table is not in second normal form, remove those non-key fields that are not dependent on the whole of the primary key. Create another table with these fields and the part of the primary key on which they do depend.*

This means that we remove the non-key fields `proj_name` and `contact` from the `Assignment` table and put them in a new table with `proj_num` (the part of the key on which they do depend). This splitting up of an unnormalized table is often referred to as *decomposition*. So we could say the original `Assignment` table is decomposed into the two tables in second normal form, as shown in Figure 8-9.

empID	project_num	hours
1005	1	8
1001	3	8
1005	3	14
1001	6	20

Assignment

pro_num	proj_name	contact
1	JenningsLtd	325-1234
3	ABCPromo	142-2345
6	Smith&Co	365-8765

Project

Figure 8-9. *Assignment table decomposed into two tables*

Had we approached this from a data modeling perspective, we would have said we have two classes, `Employee` and `Project`, with a Many-Many relationship between them as in Figure 8-3b. As discussed in Chapter 7, to represent this relationship we need to add an intermediary table, and we would have come up with exactly the same tables (along with an `Employee` table), as in Figure 8-9.

Once again, we have arrived at the same solution via two routes: thinking about the classes and their relationships, or considering the functional dependencies and normalization.

Third Normal Form

You guessed it. Tables in second normal form can still cause us problems. This time, consider our `Employee` table with some added information about the department for which an employee works. Take a look at the table in Figure 8-10.

empID	last_name	first_name	dept_num	dep_name
1001	Smith	John	2	Marketing
1005	Jones	Susan	2	Marketing
1029	Li	Jane	1	Sales

Figure 8-10. *Employee table with updating problems*

What is the primary key for the Employee table in Figure 8-10? If an employee works for only one department, it is enough to know just the empID to find a particular row. Is the table in first normal form? Yes. If I know the value of empID (e.g., 1029), I can tell you a unique value for each of the other fields. Is the table in second normal form? Yes, the primary key is only one field now, so nothing can depend on "part" of the key.
Are there still problems? Yes. The information about the department name is repeated on several rows and is liable to become inconsistent.

The situation in this table is that the name of the department is determined by more than one field. If I know that the value of the primary key field empID is 1001, I can tell you that the department name is Marketing. However, if I know that the value of dept_num is 2, I can also tell you that the department name is Marketing. There are two different fields determining what the value of the department name is. This is where the problem arises this time, and it leads to a definition for third normal form.

> *A table is in third normal form if it is in second normal form AND no non-key fields depend on a field(s) that is not the primary key.*

As in the other normal forms, we also have a simple method for correcting a table that is not in third normal form.

> *If a table is not in third normal form, remove the non-key fields that are dependent on a field(s) that is not the primary key. Create another table with this field(s) and the field on which it does depend.*

For the Employee table in Figure 8-10, this would mean removing the field dept_name from the original Employee table and putting it in a new table along with the field on which it depends (dept_num),v as shown in Figure 8-11. The field dept_num will be the primary key of our new table and will also remain in the Employee table as a foreign key.

empID	last_name	first_name	dept_num
1001	Smith	John	2
1005	Jones	Susan	2
1029	Li	Jane	1

Employee

dept_num	dep_name
1	Sales
2	Marketing
3	Research

Department

Figure 8-11. *Employee table decomposed to two tables*

Boyce-Codd Normal Form

This is the last normal form that involves functional dependencies. For most tables, it is equivalent to third normal form, but it is a slightly stronger statement for some tables where there is more than one possible combination of fields that could be used as a primary key. We are not going to consider those here. However, Boyce-Codd normal form is quite an elegant statement that encapsulates the first three normal forms.

A table is in Boyce-Codd normal form if every determinant could be a primary key.

Let's see how this works. Say I know that the value of a particular field (e.g., `proj_num`) determines the value of another field (e.g., `proj_name`). We say that `proj_num` is a *determinant* (it determines the value of something else). In any table where this is the case, then `proj_num` must be able to be the primary key.

Consider the `Assignment` table in Figure 8-8. `proj_num` determines `proj_name`, but `proj_num` is not able to be a primary key (there can be several rows with the same value of `proj_num`). In this case, Boyce-Codd normal form is a more general statement of second normal form—`proj_num` is a determinant, but it is not the whole key. In the `Employee` table in Figure 8-10, `dept_num` is a determinant, but it cannot be a primary key because it is not different in every row. In this case, Boyce-Codd normal form is a statement that includes third normal form.

One of the sweetest ways to sum up the normal forms we have discussed is from Bill Kent.[2] He summarizes the normal forms this way:

A table is based on
the key,
the whole key,
and nothing but the key (so help me Codd)

Just remembering this simple quotation can help you ensure all your tables are normalized to third normal form.

Data Models or Functional Dependencies?

In our discussions of the normal forms, based on functional dependencies, you have seen that in most of the examples we have arrived at the same set of tables as we did, in previous chapters, by considering classes and their relationships. What are the differences between the two approaches? In general, how should we go about our database design?

Essentially, we have two tools at our dipsosal, and we should use either or both when we find them helpful. This will depend on particular people and the particular problem we are trying to model. Whichever tool we use, the most essential thing is to understand the scope of the problem and the intricacies of the relationships between pieces of data. A detailed understanding requires us to ask very specific questions about the project. We can represent the answers with either part of a data model or by writing down a functional dependency. Sometimes one way just feels more natural than the other. Let's look at some examples.

For a particular problem, I may know I am going to require data about employees and projects, and I need to know more about the relationships between them.

From a data modeling perspective, I might ask, "Can an employee be associated with more than one project?" I can use the answer to decide whether the relationship between the employee class and project class is 1-Many or Many-Many. To me, this feels like a natural way to think about and discuss the issue. From a functional dependency perspective, I would ask something like, "If I know the employee's ID number, can I know a unique project with which she is associated?"

[2] William Kent, "A Simple Guide to Five Normal Forms in Relational Database Theory," Communications of the ACM 26(2), Feb. 1983, 120-125.

If the answer is "Yes," I would represent this as the functional dependency

```
empID → project
```

To me, this latter way of describing this aspect of the problem doesn't feel natural. Other people might think quite differently. The two questions and the two ways of representing them (class diagram or functional dependency) contain pretty much the same information about the relationship between employees and projects.

Let's try another example. What about the relationship between salary and tax rate? From a functional dependency perspective, I would ask, "If I know the salary, can I uniquely determine the tax rate?"

To me, that feels like a good way to think about this aspect of the problem. If the answer is "Yes," I can represent it as the functional dependency

```
salary → tax_rate
```

From a data-modeling perspective, I'm not sure what question I would ask. I probably don't have a salary class or a tax rate class, so thinking about relationships between classes is not such a natural way to come to terms with this intricacy.

What would happen if we tried to do our whole database design in terms of functional dependencies and normalization? We could start out with one huge table with a field to hold every piece of information. This is sometimes referred to as the *universal relation*. We could make a list of all the functional dependencies that apply between all the different fields and then apply our normalization rules to gradually decompose our big table into a set of normalized tables. There are in fact algorithms that allow you to do exactly that automatically. However, putting all the rules about our database in terms of functional dependencies and treating all the pieces of information as independent fields of one big table is rarely a practical way to start.

When we first start thinking about a problem, it is natural to think in quite general terms. For example, we might know we have to keep data about people, and information about projects, and we mustn't forget the buildings. We might not have a clear idea at the start what data we want to keep about each of these things, so trying to capture this original information with functional dependencies is not going to be helpful. However, the very basic ideas about the project fall quite naturally into classes. A data model or class diagram will show us that we need classes for buildings, projects, and people, and will allow us to start thinking about the relationships. Do people work in a particular building? Do people work on more than one project? Do people have other relationships with projects (e.g., might they manage them as well as work on them)? Do people manage each other?

All these broad initial ideas about a project are easily captured by the data model. The data model also helps us to find out more detailed information as we question the cardinalities and optionalities of the relationships, or look for fan traps, or check to see whether some relationships are redundant.

Once we are satisfied that our class diagram captures the information correctly, we can then represent the diagram as a set of tables and primary and foreign keys as described in Chapter 7. At this point, it is then a good idea to look at each table and see whether it is normalized. We might have an `Employee` table with fields `empID`, `last_name`, `first_name`, `salary`, and `tax_rate`, with `empID` as the primary key. Now we might ask about the functional dependencies between `salary` and `tax_rate`. If there is a functional dependency, our table is not in third normal form (`tax_rate` depends on something other than the primary key) and it should not be in this table.

The data model is great for the big picture, and normalization is great for the finer details. Use both these tools to ensure that you get the best structure for your database.

Additional Considerations

We have looked at functional dependencies and the normal forms that are defined using them: first, second, third, and Boyce-Codd normal form. There are other dependencies that can exist between pieces of data and additional normal forms that protect against some problems that may occur. I am not going to describe these in any very formal way, but I will point out what aspects of data models they relate to.

Fourth and fifth normal forms deal with tables for which there are multi-valued dependencies. We have already seen a case where this can occur. Let's reconsider the sports team example from Chapter 5. We have players, matches, and teams. If I name a team, there are multiple values of match associated with that team, and similarly multiple associated players. Let's say we are particularly interested in matches—who plays in them and what teams are involved. We need to consider whether we should have the intermediate table (Appearance) and/or the other relationships in Figure 8-12.

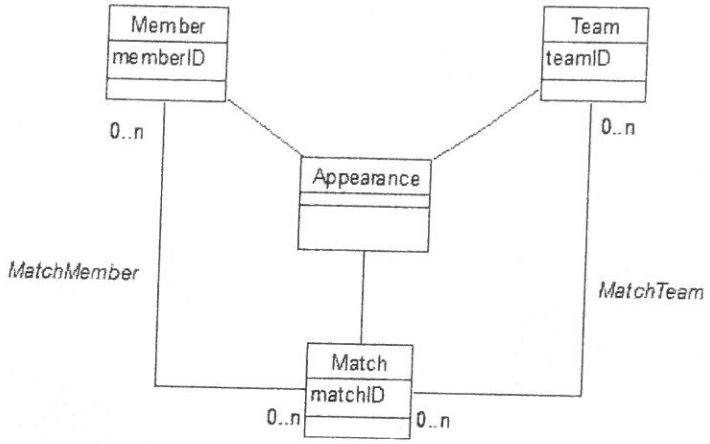


Figure 8-12. *What relationships are needed between Member, Team, and Match?*

If we represent the model in Figure 8-12 in a relational database, we would need tables for each of the classes `Member`, `Team`, `Match`, and `Appearance`. We would also need two additional tables to represent the Many-Many relationships between `Match` and `Team`, and between `Match` and `Member`. Figure 8-13 shows some data that could be in the tables.

Match	Member
MatchA	Jim
MatchA	Sue
MatchA	Hal
MatchA	Li

MatchMember

Match	Team
MatchA	Team1
MatchA	Team2
MatchB	Team1
MatchB	Team3

MatchTeam

Match	Member	Team
MatchA	Jim	Team1
MatchA	Sue	Team1
MatchA	Hal	Team1
MatchA	Li	Team2

Appearance

Figure 8-13. *Sample data representing the relationships in Figure 8-12*

For each of the tables in Figure 8-13, the primary key is made up of all the fields. There are no non-key fields, and there are no functional dependencies. They are all therefore in Boyce-Codd normal form because there are no determinants that are not possible keys (there are no determinants!). The question is, "Do we need all three tables?" There is clearly some repeated information with the data as it stands. For example, the fact that Jim is involved in MatchA can be seen from both the `MatchMember` and the `Appearance` tables. When information is stored twice, there is always the danger of it becoming inconsistent. So what (if anything) do we need to get rid of?

A match has many members involved in it and many (two) teams taking part. The question we need to answer is, "Are these two sets of information independent for our problem?" If they are, we don't need (and

shouldn't have) the Appearance table. However, as we discussed in Chapter 5, it is likely that we will need to know which member played for which team in a particular match. We cannot work that out with just the data in the other two tables (nor even if we included a MemberTeam table). So for this situation where we need to know "who played for which team in which match," the Appearance table is necessary.

What about the other two tables in Figure 8-13? If we have the Appearance table, do we need these other two as well? Recapping the discussion in Chapter 5, the questions we need to ask are, "Do we want to know about matches and teams independent of the members involved?" and, "Do we want to know about members and matches independent of the teams?" Let's think about the first question. What happens when the original draw for the competition is determined? We will probably need to record in our database that Team1 and Team2 are scheduled to play in MatchA. If we only have the Appearance table, we cannot insert appropriate records. Why? Because as all the fields are part of the primary key, none can be empty, and we have nothing to put in the Member field. We want to record the fact that this match is scheduled, and we need to do that independently of the members involved. We may also have additional information to record about matches and teams that is independent of members. For example, we will probably need to record a score. Without a MatchTeam table, where would we store that? Which row in the Appearance table would it go in? Many of them. So yes, we do need the MatchTeam table if we want to store all this information. You can go through a similar thought process to decide whether the table MatchMember is also necessary.

These sorts of questions arise every time we have three (or more) classes that are interrelated in any way. Are there situations when we need to know about combinations of objects from all three classes? Do we have information about combinations of objects from two of the classes independent of the third? If we figure out the answers to these questions correctly, we can be confident that the final tables will adequately represent the problem.

Summary

If we have poorly structured tables in a database, we run the risk of having problems with updating data. These include:

- **Modification problems**: If information is repeated, it will become inconsistent if not updated everywhere.
- **Insertion problems**: If we don't have information for each of the primary key fields, we will not be able to enter a record
- **Deletion problems**: If we delete a record to remove a piece of information, we might as a consequence lose some additional information.

By understanding the concepts of functional dependencies, primary keys, and normalization, we can ensure that our tables are structured in such a way as to avoid the update problems described previously.

- A functional dependency exists between two sets of fields in a table: If field A functionally determines field B, this means that if I know the value for A, I can uniquely tell you a value for B.
- A primary key is a (minimal) set of field(s) that functionally determines all the other fields in the table.
- The first three normal forms can be summed up as

  A table is based on
  the key,
  the whole key,
  and nothing but the key

- A table in Boyce-Codd normal form is one in which every determinant could be a primary key.
- Where you have three or more interrelated classes, ask questions about what information, if any, you need to know that involves all three classes and what information involves two classes independent of the third.

When designing a relational database

- Create original use cases and a data model.
- Ask questions about the data model to improve understanding of the problem.
- Represent the data model with tables, primary keys, and foreign keys.
- Check that each table is suitably normalized.

TESTING YOUR UNDERSTANDING

Exercise 8-1.

Example 1-3 back in Chapter 1 is a good real-life example of unnormalized data. To recap: Farms are visited and a number of samples are taken from different fields. The number of each species (just Springtail and Beetle for now) in each sample is recorded. A version of the data is shown in Figure 8-14.

FarmID	FarmName	Field	Date	Visit	SampleID	Insect	Count
1	HighGate	F2	12-Mar-11	14	3	Beetle	2
1	HighGate	F2	09-Feb-11	14	2	Beetle	4
1	HighGate	F1	09-Feb-11	14	1	Beetle	4
1	HighGate	F1	18-Mar-11	15	1	Springtail	5
1	HighGate	F2	09-Feb-11	14	3	Springtail	3
1	HighGate	F2	09-Feb-11	14	2	Springtail	5
1	HighGate	F1	09-Feb-11	14	1	Springtail	6
1	High-Gate	F1	18-Mar-11	15	1	Beetle	7
2	Greyton	F2	09-Feb-11	16	1	Beetle	2
2	Greyton	F1	09-Feb-11	16	2	Beetle	4
2	Greyton	F1	09-Feb-11	16	2	Springtail	5
2	Greyton	F2	09-Feb-11	16	1	Springtail	3

Figure 8-14. *Unnormalized version of insect data*

Consider the following questions:

a) What are some of the updating problems that could occur with the table in Figure 8-14?

b) Which of the following functional dependencies hold for the insect data?

- FarmID → FarmName?
- FarmID → Visit?

- Visit → Date?
- Date → Visit?
- Visit → FarmID?
- Sample → Field?
- (Sample, VisitID) → Field?
- (Sample, Insect) → Count?

c) (VisitID, Sample, Insect) → Count? (VisitID, Sample, Insect) is suggested as an appropriate primary key. Can you determine all the other values from knowing the values of these three fields? Would it be a suitable primary key?

d) Using the fields in Part C as a primary key use the normalization rules to decompose the table in Figure 8-14 into a set of tables in third normal form.