Recap:

- 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 lecture, you saw how to represent the main parts of the data model in a relational database.
  - 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.

Normalization

- In this lecture, 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.

- 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 in 1970 and has been a cornerstone of relational database design since then.

Update Problems

- 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 the next slide about employees and some small projects to which they have been assigned.

Update Problems

- 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.

Can you see a problem lurking in the Assignment table?

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

Tables with potential update problems

Update anomalies or update problems

- 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 the previous slide, 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*.

Insertion Problems

- You will recall that it is necessary to have a primary key for every table in our DB.
- 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 the previous slide?

- 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.

Concatenated key, the pair emp and project_num

- 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 lectures, 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 Problems

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

Solution: another table to record information about projects

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

Tables with update anomalies removed

Functional Dependencies

- Normalization helps us to determine whether our tables are structured in such a way as to avoid the update problems described previously.
- 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 $\longrightarrow$ last_name
- For the situation depicted in the updated table, 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."

Functional dependency requirement

- 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.

Do we have a functional dependency between an employee's ID number and a project to which he is assigned?

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

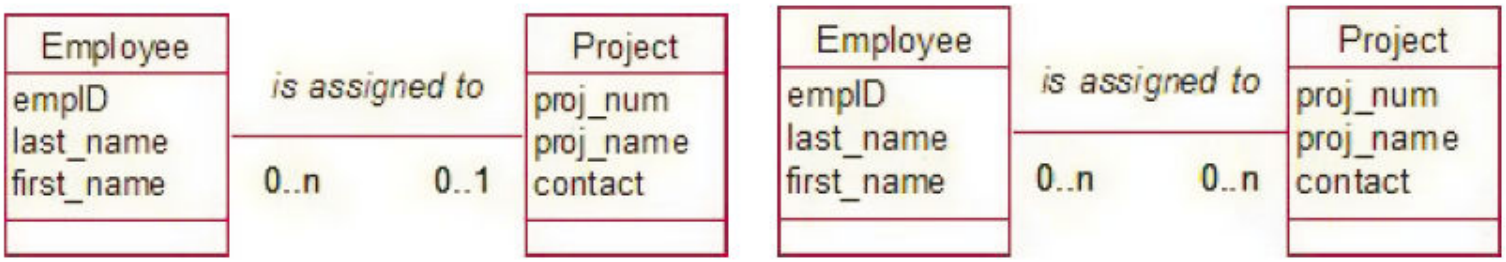
Does emp ID uniquely determine a project number?

- 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 we 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.*

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

- If I know the value of the key, 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.
- 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.

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.

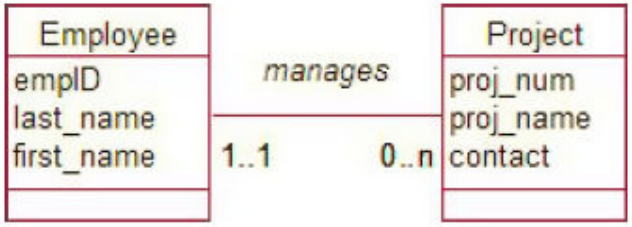


Figure 8-4. *A 1–Many relationship*

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*

Normal Forms

- Tables that are “normalized” will generally avoid the updating problems we examined earlier.
- There are several levels of normalization called normal forms, each addressing additional situations where problems may occur.
- We will look at the normal forms that are defined using functional dependencies.

- One of the concepts most important to a relational database is that of normalized data.
- Normalized data is organized into a structure that preserves the integrity of the data while minimizing redundant data.
- The goal of all normalized data is to prevent lost data and inconsistent data, while minimizing redundant data.
- A normalized database is one whose tables are structured according to the rules of normalization.
- These rules — referred to as normal forms — specify how to organize data so that it is considered normalized.
- When Codd first introduced the relational model, he included three normal forms.
- Since then, more normal forms have been introduced, but the first three still remain the most critical to the relational model.
- The degree to which a database is considered normalized depends on which normal forms can be applied.

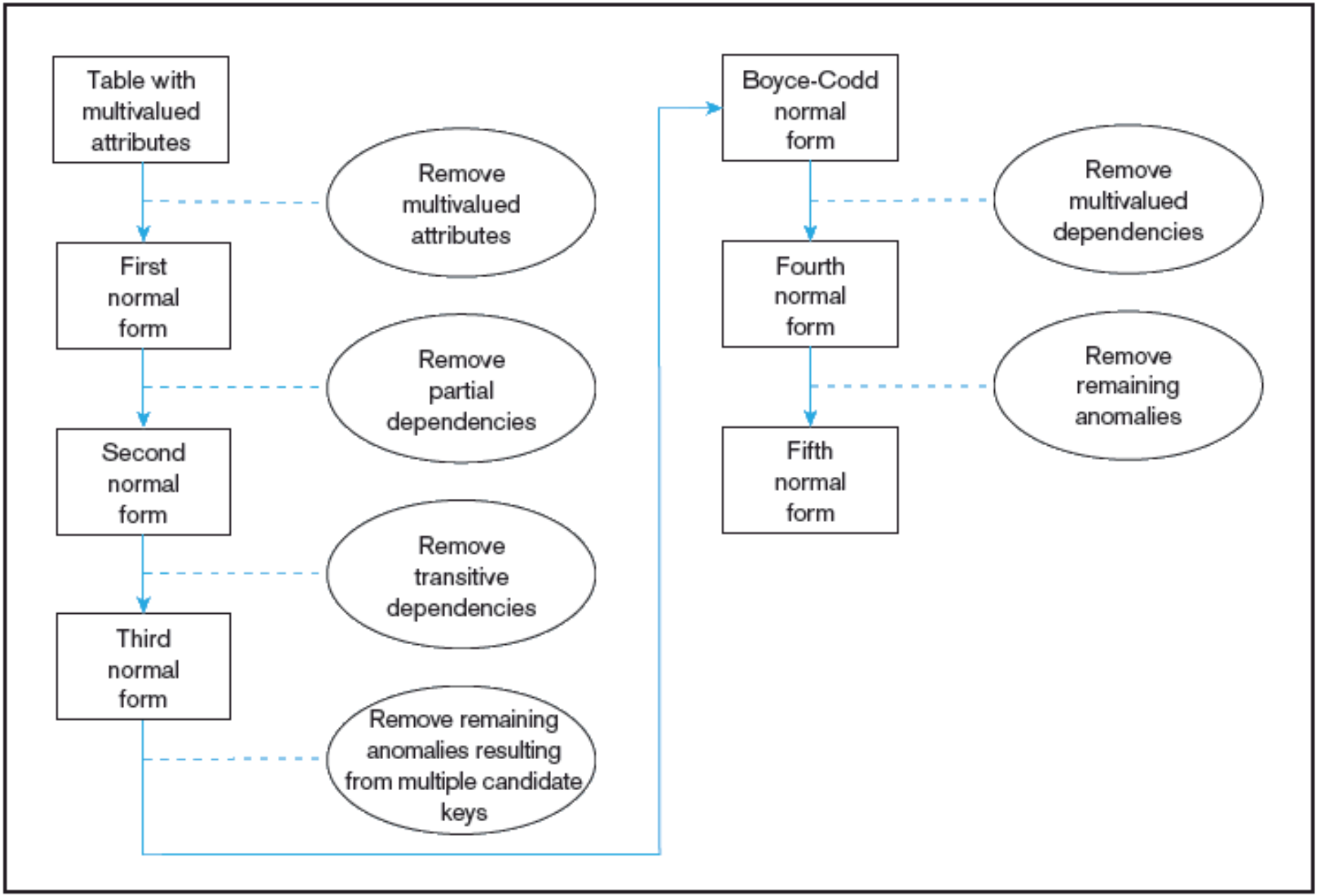
- For example, some database designs aim for only the second normal form; however, some databases strive to achieve conformance to the fourth or fifth normal form.
- There is often a trade-off between strict adherence to the normal forms and system performance.
- Often, the more normalized the data, the more taxing it can be on a system.
- As a result, a DB design must strike a balance between a fully normalized DB and system performance.
- In most situations, the first three normal forms provide that balance.

Steps in Normalization

Normalization can be accomplished and understood in stages, each of which corresponds to a normal form (see Figure 4-22). A **normal form** is a state of a relation that requires that certain rules regarding relationships between attributes (or functional dependencies) are satisfied. We describe these rules briefly in this section and illustrate them in detail in the following sections:

1. *First normal form* Any multivalued attributes (also called *repeating groups*) have been removed, so there is a single value (possibly null) at the intersection of each row and column of the table (as in Figure 4-2b).
2. *Second normal form* Any partial functional dependencies have been removed (i.e., nonkey attributes are identified by the whole primary key).
3. *Third normal form* Any transitive dependencies have been removed (i.e., nonkey attributes are identified by only the primary key).
4. *Boyce-Codd normal form* Any remaining anomalies that result from functional dependencies have been removed (because there was more than one possible primary key for the same nonkeys).
5. *Fourth normal form* Any multivalued dependencies have been removed.
6. *Fifth normal form* Any remaining anomalies have been removed.

Steps in normalization



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	nichollii	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	nichollii	Black peppermint gum	shelter	coppicing	bird food

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

Problems?

- 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 below.

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	nichollii	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	nichollii	Black peppermint gum	shelter	coppicing	bird food

- 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, 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.

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	nichollii	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*

Name	Date of Birth	Address	Email	Date Joined	Meeting Date	Location	Did Member Attend?
Martin	Feb 27, 1972	1 The Avenue, NY	martin@some.com	Jan 10, 2005	Mar 30, 2005	Lower West Side, NY	Y
Jane	Dec 12, 1967	33 Some Road, Washington	Jane@server.net	Jan 12, 2005	Mar 30, 2005	Lower West Side, NY	N
Kim	May 22, 1980	19 The Road, New Townsville	kim@mail.com	Jan 23, 2005	Mar 30, 2005	Lower West Side, NY	Y

You find, though, that each time you want to add a record of details of another meeting you end up duplicating the members' details:

Name	Date of Birth	Address	Email	Date Joined	Meeting Date	Location	Did Member Attend?
Martin	Feb 27, 1972	1 The Avenue, NY	martin@some.com	Jan 10, 2005	Mar 30, 2005	Lower West Side, NY	Y
Martin	Feb 27, 1972	1 The Avenue, NY	martin@some.com	Jan 10, 2005	April 28, 2005	Lower North Side, NY	Y
Jane	Dec 12, 1967	33 Some Road, Washington	Jane@server.net	Jan 12, 2005	Mar 30, 2005	Lower West Side, NY	N
Jane	Dec 12, 1967	33 Some Road, Washington	Jane@server.net	Jan 12, 2005	April 28, 2005	Upper North Side, NY	Y
Kim	May 22, 1980	19 The Road, New Townsville	kim@mail.com	Jan 23, 2005	Mar 30, 2005	Lower West Side, NY	Y
Kim	May 22, 1980	19 The Road, New Townsville	kim@mail.com	Jan 23, 2005	April 28, 2005	Upper North Side, NY	Y

There are a number of problems with this approach. First, it wastes a lot of storage space. You only need to store the members' details once, so repeating the data is wasteful. What you have is repeating groups, something that the second rule of first normal form tells you to remove. You can remove duplicated data by splitting the data into tables: one for member details, another to hold location details, and a final table detailing meetings that took place. Another advantage of splitting data into tables is that it avoids something called the *deletion anomaly*, where deleting a record results in also deleting the data you want to keep. For example, say you want to delete details of meetings held more than a year ago. If your data isn't split into tables and you delete records of meetings, then you also delete members' details, which you want to keep. By separating member details and meeting data, you can delete one and not both.

Artist	CDName	Copyright
Jennifer Warnes	Famous Blue Raincoat	1991
Joni Mitchell	Blue; Court and Spark	1971; 1974
William Ackerman	Past Light	1983
Kitaro	Kojiki	1990
Bing Crosby	That Christmas Feeling	1993
Patsy Cline	Patsy Cline: 12 Greatest Hits	1988
Jose Carreras; Placido Domingo; Luciano Pavarotti	Carreras Domingo Pavarotti in Concert	1990

ArtistID	ArtistName
10001	Jennifer Warnes
10002	Joni Mitchell
10003	William Ackerman
10004	Kitaro
10005	Bing Crosby
10006	Patsy Cline
10007	Jose Carreras
10008	Placido Domingo
10009	Luciano Pavarotti

ArtistID	CDID
10001	99301
10002	99302
10002	99303
10003	99304
10004	99305
10005	99306
10006	99307
10007	99308
10008	99308
10009	99308

CDID	CDName	Copyright
99301	Famous Blue Raincoat	1991
99302	Blue	1971
99303	Court and Spark	1974
99304	Past Light	1983
99305	Kojiki	1990
99306	That Christmas Feeling	1993
99307	Patsy Cline: 12 Greatest Hits	1988
99308	Carreras Domingo Pavarotti in Concert	1990

Relations that conform to the first normal form

ArtistID	ArtistName	Agency	AgencyState
10001	Jennifer Warnes	2305	NY
10002	Joni Mitchell	2306	CA
10003	William Ackerman	2306	CA
10004	Kitaro	2345	NY
10005	Bing Crosby	2367	VT
10006	Patsy Cline	2049	TN
10007	Jose Carreras	2876	CA
10008	Placido Domingo	2305	NY
10009	Luciano Pavarotti	2345	NY

To be in compliance with the 1NF, the following requirements must be met

- Each column in a row must be atomic.
  - In other words, the column can contain only one value for any given row.
- Each row in a table must contain the same number of columns.
  - Given that each column can contain only one value, this means that each row must contain the same number of values.
- All rows in a table must be different.
  - Although rows might include the same values, each row, when taken as a whole, must be unique in the table.

AuthorBook

AuthFN	AuthMN	AuthLN	BookTitle
Hunter	S.	Thompson	Hell's Angels
Rainer	Maria	Rilke	Letters to a Young Poet
Rainer	Maria	Rilke	Letters to a Young Poet
John	Kennedy	Toole	A Confederacy of Dunces
Annie	NULL	Proulx	Postcards, The Shipping News
Nelson	NULL	Algren	Nonconformity

In order to conform to the first normal form, you must eliminate the duplicate values in the BookTitle column, ensure that each row contains the same number of values, and avoid duplicated rows. One way to achieve the necessary normalization is to place the data in separate tables, based on the objects represented by the data. In this case, the obvious place to start is with authors and books. All data related to authors is placed in one table, and all data related to books is placed in another table

Authors

AuthID	AuthFN	AuthMN	AuthLN
1006	Hunter	S.	Thompson
1007	Rainer	Maria	Rilke
1008	John	Kennedy	Toole
1009	Annie	NULL	Proulx
1010	Nelson	NULL	Algren

AuthorBook

AuthID	BookID
1006	14356
1007	12786
1007	14555
1008	17695
1009	19264
1009	19354
1010	16284

Books

BookID	BookTitle	Trans
14356	Hell's Angels	English
12786	Letters to a Young Poet	English
14555	Letters to a Young Poet	French
17695	A Confederacy of Dunces	English
19264	Postcards	English
19354	The Shipping News	English
16284	Nonconformity	English

Second Normal Form

- It is possible for a table in first normal form to still have updating problems. The Assignment table below 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 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*

Third Normal Form

- 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.

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)*

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.

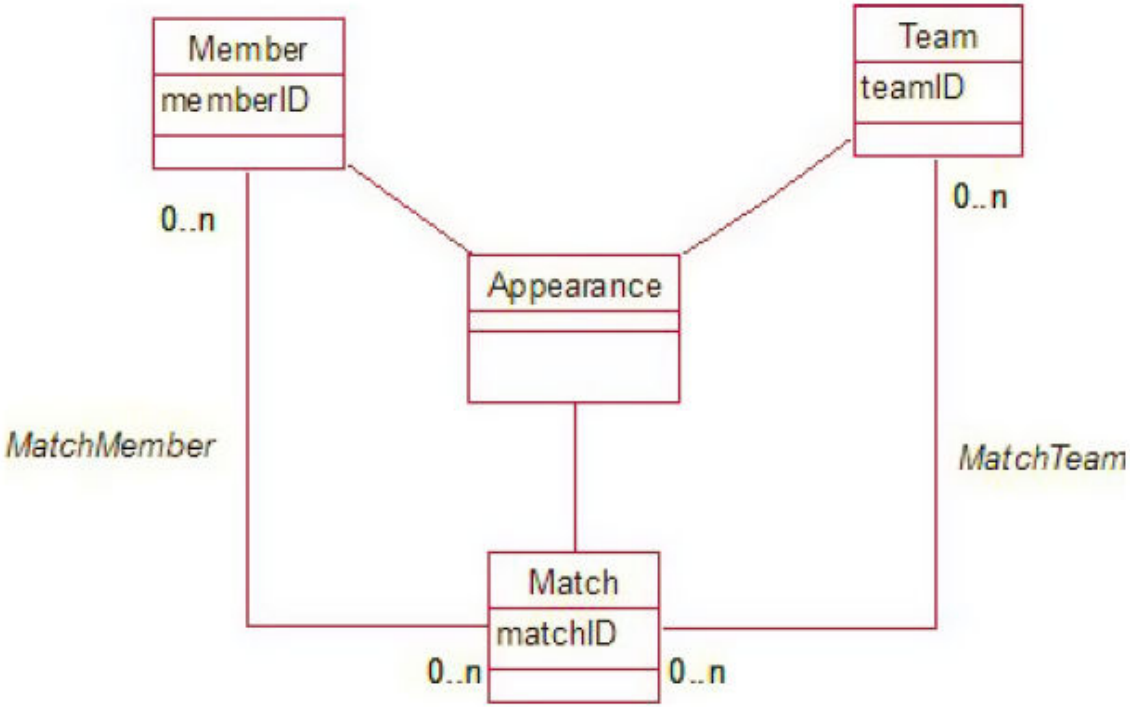


Figure 8-12. *What relationships are needed between Member, Team, and Match?*

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.

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*

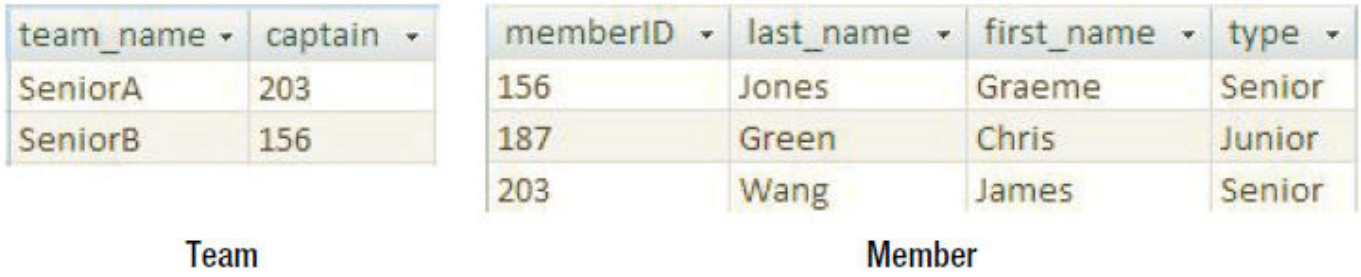
For each of the tables in Figure 8-13, the primary key is made up of all the fields. There are no non–key fields, and there are no functional dependencies. They are all therefore in Boyce–Codd normal form because there are no determinants that are not possible keys (there are no determinants!). The question 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, 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, 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.

Deleting Referenced Records

You have seen how we can use foreign keys to represent relationships between two tables. Have another look at our model of teams and members in Figure 9-5. We can represent the relationship *is captain of* with a foreign key (`captain`) in the `Team` table, as shown in Figure 9-12.



team_name	captain
SeniorA	203
SeniorB	156

Team

memberID	last_name	first_name	type
156	Jones	Graeme	Senior
187	Green	Chris	Junior
203	Wang	James	Senior

Member

Figure 9-12. *Teams and members*

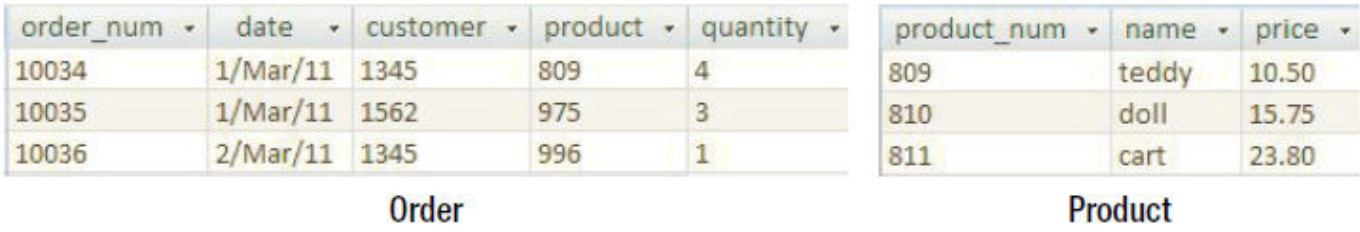
A foreign key ensures that we have referential integrity. Recall from Chapter 7 that referential integrity prevents us from having a value in the foreign key field `captain` if the value does not exist in the primary key field `memberID` in the `Member` table. This ensures that all of our captains are members for which we have already recorded names and other details. Unlike a primary key, a foreign key field is not necessarily mandatory, and the captain field may be empty (i.e., referential integrity does not make it necessary for every team to have a captain). We can of course impose that extra constraint if we want to, by specifying that the `captain` field must be `NOT NULL`.

So far we have only looked at referential integrity from the point of view of adding new teams and captains. However, we also have the situation of deleting members from the `Member` table. If we attempt to delete member 156, for example, we shall have a problem with the referential integrity in the `Team` table. The captain of SeniorB will no longer exist in the `Member` table.

There are three ways to deal with deleting referenced records

- Database software products vary in their ability to provide each of these options, but all will provide the first, as follows:
- **Disallow delete:**
  - You cannot delete a row that is being referenced. For example, the deletion of member 156 will not be allowed while it is being referenced by the Team table. If we want to delete member 156, we will first have to remove the reference to him in the Team table and then delete him from the Member table.
- **Nullify delete:**
  - If member 156 is deleted, the field that is referencing it, captain, will be nullified (made empty). This essentially is saying that if a captain of a team leaves the club, that team has no captain—which is probably quite sensible in this situation.
- **Cascade delete:**
  - If a row is deleted, all the rows referencing it will be deleted also (and the rows referencing them, and on and on). In this case, deleting member 156 would mean that the team SeniorB would be deleted. This is clearly not desirable.

Let's consider a different situation, of orders and products, as in Figure 9-13.



order_num	date	customer	product	quantity
10034	1/Mar/11	1345	809	4
10035	1/Mar/11	1562	975	3
10036	2/Mar/11	1345	996	1

Order

product_num	name	price
809	teddy	10.50
810	doll	15.75
811	cart	23.80

Product

Figure 9-13. *Orders and products: What happens if we delete a product?*

What happens if we no longer stock product 809? If we delete this row in the `Product` table, our referential integrity will be compromised as order number 10034 refers to it. What are our choices? A "nullify delete" means having nothing in the foreign key `product` field in the `Order` table. This makes no sense. We would have that there was once an order for four of some product—but we don't know what that product was and we have no way of finding the price. Clearly, this is not going to be useful. A "cascade delete" would mean that all the orders for product 809 would be deleted. This doesn't seem sensible either, as a business is going to need to keep track of all its orders to determine profits, tax, and so on. Our only choice in this case, then, is the "disallow delete" option. If there is an order for the product, we can't delete that product from the `Product` table.

It is important that we keep discontinued products in the table, but we will want to be able to distinguish them from current products. For a case like this, we might decide to add an additional field to our `Product` table (say `current`) to distinguish current products from discontinued ones. We have a new problem now. How do we prevent new orders from being entered for discontinued products?

Queries with Two or More Tables

- Most of our queries will require information from several tables in our database.
- There are a number of different relational operations that we can use to combine tables.
- One really elegant feature of relational DB operations is that when we carry out operations on one or more tables, we can think of the result as a new table.
- This new table does not exist permanently in the database, but conceptually it is convenient to think of it as a virtual table that exists for the time of the query.
- All the operations that we used before can be applied to the new virtual table.
- We can also take a virtual table that results from combining two tables and then combine that with another real table, and then another.
- So with a few quite simple operations, we can easily build up queries that involve a number of tables that will satisfy quite complex questions.

Inner Join

```
SELECT column_name(s)
FROM table_name1
INNER JOIN table_name2
ON table_name1.column_name=table_name2.column_name
```

PS: INNER JOIN is the same as JOIN.

SQL INNER JOIN Example

The "Persons" table:

P_Id	LastName	FirstName	Address	City
1	Hansen	Ola	Timoteivn 10	Sandnes
2	Svendson	Tove	Borgvn 23	Sandnes
3	Pettersen	Kari	Storgt 20	Stavanger

The "Orders" table:

O_Id	OrderNo	P_Id
1	77895	3
2	44678	3
3	22456	1
4	24562	1
5	34764	15

Now we want to list all the persons with any orders.

We use the following SELECT statement:

```
SELECT Persons.LastName, Persons.FirstName, Orders.OrderNo
FROM Persons
INNER JOIN Orders
ON Persons.P_Id=Orders.P_Id
ORDER BY Persons.LastName
```

The result-set will look like this:

LastName	FirstName	OrderNo
Hansen	Ola	22456
Hansen	Ola	24562
Pettersen	Kari	77895
Pettersen	Kari	44678

The INNER JOIN keyword return rows when there is at least one match in both tables. If there are rows in "Persons" that do not have matches in "Orders", those rows will NOT be listed.

Left Join

- The LEFT JOIN keyword returns all rows from the left table (table_name1), even if there are no matches in the right table (table_name2).

- **SQL LEFT JOIN Syntax**

```
SELECT column_name(s) FROM table_name1
LEFT JOIN table_name2
ON table_name1.column_name=table_name2.column_name
```

- **PS:** In some databases LEFT JOIN is called LEFT OUTER JOIN.

The "Persons" table:

P_Id	LastName	FirstName	Address	City
1	Hansen	Ola	Timoteivn 10	Sandnes
2	Svendson	Tove	Borgvn 23	Sandnes
3	Pettersen	Kari	Storgt 20	Stavanger

The "Orders" table:

O_Id	OrderNo	P_Id
1	77895	3
2	44678	3
3	22456	1
4	24562	1
5	34764	15

Now we want to list all the persons and their orders - if any, from the tables above.

We use the following SELECT statement:

```
SELECT Persons.LastName, Persons.FirstName, Orders.OrderNo
FROM Persons
LEFT JOIN Orders
ON Persons.P_Id=Orders.P_Id
ORDER BY Persons.LastName
```

The result-set will look like this:

LastName	FirstName	OrderNo
Hansen	Ola	22456
Hansen	Ola	24562
Pettersen	Kari	77895
Pettersen	Kari	44678
Svendson	Tove	

The LEFT JOIN keyword returns all the rows from the left table (Persons), even if there are no matches in the right table (Orders).

Right Join

- The RIGHT JOIN keyword returns all the rows from the right table (table_name2), even if there are no matches in the left table (table_name1).

- **SQL RIGHT JOIN Syntax**

```
SELECT column_name(s)
FROM table_name1
RIGHT JOIN table_name2
ON table_name1.column_name=table_name2.column_name
```

- **PS:** In some databases RIGHT JOIN is called RIGHT OUTER JOIN.

The "Persons" table:

P_Id	LastName	FirstName	Address	City
1	Hansen	Ola	Timoteivn 10	Sandnes
2	Svendson	Tove	Borgvn 23	Sandnes
3	Pettersen	Kari	Storgt 20	Stavanger

The "Orders" table:

O_Id	OrderNo	P_Id
1	77895	3
2	44678	3
3	22456	1
4	24562	1
5	34764	15

Now we want to list all the orders with containing persons - if any, from the tables above.

We use the following SELECT statement:

```
SELECT Persons.LastName, Persons.FirstName, Orders.OrderNo
FROM Persons
RIGHT JOIN Orders
ON Persons.P_Id=Orders.P_Id
ORDER BY Persons.LastName
```

The result-set will look like this:

LastName	FirstName	OrderNo
Hansen	Ola	22456
Hansen	Ola	24562
Pettersen	Kari	77895
Pettersen	Kari	44678
		34764

The RIGHT JOIN keyword returns all the rows from the right table (Orders), even if there are no matches in the left table (Persons).

Full Join

- The FULL JOIN keyword return rows when there is a match in one of the tables.

- **SQL FULL JOIN Syntax**

```
SELECT column_name(s)
FROM table_name1
FULL JOIN table_name2
ON table_name1.column_name=table_name2.column_name
```

The "Persons" table:

P_Id	LastName	FirstName	Address	City
1	Hansen	Ola	Timoteivn 10	Sandnes
2	Svendson	Tove	Borgvn 23	Sandnes
3	Pettersen	Kari	Storgt 20	Stavanger

The "Orders" table:

O_Id	OrderNo	P_Id
1	77895	3
2	44678	3
3	22456	1
4	24562	1
5	34764	15

Now we want to list all the persons and their orders, and all the orders with their persons.

We use the following SELECT statement:

```
SELECT Persons.LastName, Persons.FirstName, Orders.OrderNo
FROM Persons
FULL JOIN Orders
ON Persons.P_Id=Orders.P_Id
ORDER BY Persons.LastName
```

The result-set will look like this:

LastName	FirstName	OrderNo
Hansen	Ola	22456
Hansen	Ola	24562
Pettersen	Kari	77895
Pettersen	Kari	44678
Svendson	Tove	
		34764

The FULL JOIN keyword returns all the rows from the left table (Persons), and all the rows from the right table (Orders). If there are rows in "Persons" that do not have matches in "Orders", or if there are rows in "Orders" that do not have matches in "Persons", those rows will be listed as well.

The Join Operation

The most common operation to combine two tables is the *inner join*. Consider the `Student` and `Enrollment` tables in Figure 10-3.

studentID	last_name	first_name	first_enrolled
12654	Green	Linda	2010
13887	Smith	John	2010
17625	King	Steven	2011
18574	Smith	James	2011
19876	Smith	Alison	2010

Student

studentID	course	year	grade
17625	COMP101	2010	A
13887	COMP101	2010	B
19765	COMP101	2011	B
17625	COMP102	2010	E
13887	COMP102	2011	A
17625	COMP102	2011	C
18574	COMP102	2011	B

Enrollment

Figure 10-3. *Parts of the Student and Enrollment tables*

If we want to answer a question such as "Who was enrolled in COMP102 in 2011?" we need data from both tables. If we were answering this question by just looking at the tables, we would first find the rows from the `Enrollment` table that satisfy the condition `course = 'COMP102' AND year = 2011`. We would then need to look at the `Student` table to find the corresponding names. An inner join allows us to combine the two tables so that all the required information appears together. For this query, we are interested in rows from the `Student` table and rows from the `Enrollment` table where the value of the `studentID` is the same in each. This will be the join condition. Let's look at the SQL statement in Listing 10-11 and then consider what it means.

Listing 10-11. *SQL Statement to Join Two Tables*

```
SELECT *
FROM Student INNER JOIN Enrollment ON Student.studentID = Enrollment.studentID
```

It is useful to think of an inner join operation as making a new virtual table that will have all the columns from both original tables. We fill this table up with every combination of rows from each table and then retain those that satisfy the condition `Student.studentID = Enrollment.studentID` (i.e., where the values of `studentID` are the same in each table). Figure 10-4 shows part of the resulting set of rows.

S.studentID	last_name	first_name	first_enrolled	E.studentID	course	year	grade
13887	Smith	John	2010	13887	COMP101	2010	B
13887	Smith	John	2010	13887	COMP102	2011	A
17625	King	Steven	2011	17625	COMP101	2010	A
17625	King	Steven	2011	17625	COMP102	2010	E
17625	King	Steven	2011	17625	COMP102	2011	C
18574	Smith	James	2011	18574	COMP102	2011	B

Figure 10-4. *Rows resulting from joining Student and Enrollment on studentID*

In Figure 10-4, the first four columns are from the `Student` table, and the second four columns are from the `Enrollment` table. We only see the combinations of rows from each table where the `studentID` is the same. Now that we have this virtual table, we can apply all the single table operations to it. We can select just those rows for enrollments in COMP102 for 2011 with a `WHERE` clause and then project or retrieve just the IDs and names of the students. The SQL statement to do this is shown in Listing 10-12 and the resulting rows in Figure 10-5.

Listing 10-12. *SQL Statement to Retrieve IDs and Names of Students in COMP102 in 2011*

```
SELECT Student.studentID, last_name, first_name
FROM Student INNER JOIN Enrollment ON Student.studentID = Enrollment.studentID
WHERE course = 'COMP102' AND year = 2011
```

studentID	last_name	first_name
13887	Smith	John
17625	King	Steven
18574	Smith	James

Figure 10-5. *Rows resulting from combining join with select and project operations*

This is just a very cursory explanation of an inner join, but I'm sure you can see how you can keep joining the resulting virtual table to another table and then another to build up ever more complex queries.

One last point that is worth mentioning in this basic introduction to joins is what happens when we join two tables such as those in Figure 10-6.

courseID	examiner
COMP101	1001
COMP102	1018
COMP205	
COMP303	1018

Course

personID	last_name	first_name
1001	Jones	Jim
1018	Li	Henry
1100	Harrow	Jenny

Lecturer

Figure 10-6. *Course and Lecturer tables*

If we want a list of courses with the names of the examiner, we might first try an inner join where `examiner` in the `Course` table is equal to `personID` in the `Lecturer` table. If we were to do this, then the resulting rows would be those shown in Figure 10-7.

courseID	examiner	personID	last_name	first_name
COMP101	1001	1001	Jones	Jim
COMP102	1018	1018	Li	Henry
COMP303	1018	1018	Li	Henry

Figure 10-7. *Result of inner join between Course and Lecturer tables*

The rows in Figure 10-7 may not be what we were expecting if we thought we were going to see a row for each course. The inner join returns combinations of rows from the two tables where `examiner = personID` and this will never be true where the `examiner` field is Null. The course COMP205 is missing from the resulting table because it does not have an examiner. If the question is more accurately worded as "Retrieve *all* the courses and, *for those courses that have one*, the examiner as well," we can use what is called an *outer join* as shown in Listing 10-13.

Listing 10-13. *Outer Join to Retrieve All Courses Along with Examiners*

```
SELECT *
FROM Course LEFT OUTER JOIN Lecturer ON examiner = personID
```

The result of this query, shown in Figure 10-8, is the same as for the inner join, but in addition, any rows in the left-hand table (`Course`) with nothing in the join field (`examiner`) will appear as well.

courseID	examiner	personID	last_name	first_name
COMP101	1001	1001	Jones	Jim
COMP102	1018	1018	Li	Henry
COMP205				
COMP303	1018	1018	Li	Henry

Figure 10-8. *Result of outer join to retrieve all the courses*

Set Operations

While joins are probably the most often used operation for combining information from several tables, there are a number of other operations. A join can be used between any two tables. Set operations are used on two tables (or virtual tables) that have the same number and type of columns. They are used for queries such as "Retrieve the rows that appear in both these tables" or "Retrieve the rows that are in this table but not that one." We can use the `Enrollment` table in Figure 10-9 to illustrate these ideas.

studentID	course	year	grade
13887	COMP101	2010	B
13887	COMP102	2011	A
17625	COMP101	2010	A
17625	COMP102	2010	E
17625	COMP102	2011	C
18574	COMP102	2011	B
19765	COMP101	2011	B

Figure 10-9. *Enrollment table*

Here are some queries we might like to carry out:

- Retrieve the ID numbers of all students who have done *both* COMP101 *and* COMP102.
- Retrieve the ID numbers of all students who have done *either* COMP101 *or* COMP102.
- Retrieve the ID numbers of all students who have done COMP101 *but not* COMP102.

First we need two queries that will return the IDs of students who have done COMP101 and COMP102, respectively. These queries and the virtual tables they produce are shown in Figure 10-10.

studentID
13887
17625
19765

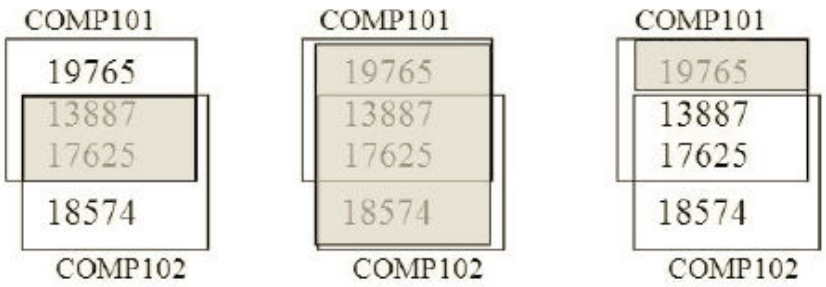
```
SELECT distinct studentID
FROM Enrolment
WHERE course=('COMP101'
```

studentID
13887
17625
18574

```
SELECT distinct studentID
FROM Enrolment
WHERE course=('COMP102'
```

Figure 10-10. *Results of queries to select students who have done particular papers*

A little reordering and overlaying of the two virtual tables as shown in Figure 10-11 can help us see what rows will satisfy each of our questions.




```
COMP101
INTERSECT
COMP102
```

```
COMP101
UNION
COMP102
```

```
COMP101
EXCEPT
COMP102
```

Figure 10-11. *Using set operations to find answers to questions about enrollments*

The three set operations shown in Figure 10-11 show us those students who have done *both* subjects (intersect), *either* subject (union), and COMP101 *but not* COMP102 (except). Oracle uses the key word MINUS instead of EXCEPT. Listing 10-14 shows the SQL to retrieve the union of the two sets of `studentID`s starting from the original real tables.

Listing 10-14. *The studentIDs for Those Students Who Have Done Either COMP101 or COMP102*

```
SELECT distinct studentID FROM Enrollment WHERE course = 'COMP101'
UNION
SELECT distinct studentID FROM Enrollment WHERE course = 'COMP102'
```

In principle, in SQL we can replace the keyword `UNION` in Listing 10-11 with the keywords `INTERSECT` and `EXCEPT` to obtain the other set operations. In practice, not all database systems provide these latter two keywords. This is because it is possible to obtain the same results using some other SQL statements. How this is done is getting beyond this introduction, but can be found in my SQL Queries book. The important thing is to know that your relational database system will allow you to write an SQL statement to retrieve rows equivalent to each of the set operations in Figure 10-11.