



Beginning **PHP4** **Databases**

Andrew Hill, Chris Lea, Deepak Thomas, Christopher Scollo
Harish Rawat, Jim Hubbard



Wrox technical support at: support@wrox.com

Updates and source code at: www.wrox.com

Peer discussion at: p2p.wrox.com

Summary of Contents

Introduction		1
Chapter 1:	PHP Fundamentals	11
Chapter 2:	Database Fundamentals	57
Chapter 3:	Understanding Relational Databases	89
Chapter 4:	The Structured Query Language	107
Chapter 5:	SQL: Data Manipulation and Retrieval	141
Chapter 6:	Data Consistency	177
Chapter 7:	Advanced SQL and PHP	199
Chapter 8:	PHP and Relational Databases	227
Chapter 9:	Dissecting PEAR::DB	251
Chapter 10:	Case Study: Using PEAR::DB – An Accounts Receivable Application	275
Chapter 11:	CaseStudy: Using Object Databases – A Library Automation System	315
Chapter 12:	Native XML Databases	337
Chapter 13:	Case Study: Using Xindice – A Recipe Exchange	389
Appendix A:	Administration	453
Appendix B:	PHP Database Functions	471
Appendix C:	Installing PHP, Apache, and iODBC	515
Index		523

3

Understanding Relational Databases

As we have briefly seen in the previous chapter, there are many possible ways to structure data, such as trees, lists, and objects. In relational databases, data is structured as relations. Relations are expressed as sets of **tuples**, which are implemented in the form of tables. Conceptually, tables are easy to grasp, since it is a form that is familiar to most people. Anyone who has read a spreadsheet, a train timetable, or even a television guide is already familiar with the organization of data into columns and rows.

In this chapter, we will lay out the basic concepts of relational databases and describe the process of organizing data in a relational manner. The topics covered in this chapter are:

- ❑ **Schema normalization**
The process by which redundancy and inconsistency are eliminated from the database
- ❑ **Keys**
Fields or combinations of fields that are used to identify records
- ❑ **Referential integrity**
A state of consistency for database schemata
- ❑ **Entity relationship diagrams**
Models used to design databases

Tabular Data

As mentioned earlier, a table is a data structure that uses columns and rows to organize the data. As an example, consider the following ledger sheet sample of a charity containing details of the donations:

Donor	Donation 1	Donation 2	Donation 3
Marco Pinelli	\$200 Solar Scholars		
Victor Gomez	\$100 Pear Creek	\$100 Danube Land Trust	\$50 Forest Asia
Seung Yong Lee	\$150 Forest Asia		

Tables represent **entities**, which are unique items like people, objects, and relationships about which we wish to store data. Each row represents an instance of the entity. In the above example, each row represents an instance of one donor. In relational database terminology, an instance is known as a **record**, but the terms row or tuple are also used.

Each column represents an **attribute** of the entity, or something about the entity. In this case, each column represents a donation made by the donor, listing the amount of the donation and the project to which the money is donated. In relational database terminology, an attribute is known as a **field**, but the term column is also very common. Adding or removing columns would change both the data stored in the table and the actual structure of the table, whereas adding or removing rows would only change data stored in the table. In other words, removing a column removes information about entities whereas removing a row only removes one instance of an entity but no information about them in general.

As we shall see in the next chapter, each field in a table is assigned a data type. The type indicates what sort of data will be stored in that field: text data, integer data, boolean (true or false) data, and so on. The assigned type then applies to that field's value for every record in the table.

Keys

We create databases because we need to store information. For the information in the database to be useful, we need to be able to perform certain operations on it. These operations fall broadly into two categories: reading the data, and changing the data. Whether one wishes to read a record, update it, or delete it, one first needs to identify the record in a way that distinguishes it from the other records in the table.

This is where keys come in. A **key** is a field or a combination of fields whose value identifies a record for a given purpose. One type of key is a unique key, which can be used to identify a single record. For example, every book has a unique ISBN (International Standard Book Number) that marks the book unmistakably. If a table of information about books includes an ISBN field, then that field can serve as a unique key.

A table might have more than one unique key. Suppose that each book in our table also has a unique product ID. While there is no problem with the existence of more than one unique key, it is considered desirable to have one that stands out as the **primary key** – the key that is considered the foremost means of identifying a record. In this case each of the unique keys is known as a **candidate key**, since each has the possibility of serving as the primary key. It is then up to the database designer to designate the primary key from among the candidate keys.

In our example from the previous section, the **Donor** field is a candidate key, if we accept (for now) that each donor is unique within the table. We shall revisit the topic of keys later in this chapter.

A Few Inadequacies

At first glance, a simple table such as the one shown in the chapter seems to meet all of our needs for storing data. When designing and filling the table, we may add as many fields and records as we like to accommodate large amounts of data. But after some examination, we are likely to encounter quite a few failings with our table. What if a donor makes more than three donations? We can add more fields to the table, of course, but to change the structure of a database once it is in use is extremely inconvenient. Also it is difficult to know in advance how many donations would be enough. What if one donor makes dozens of donations?

What if we wish to store more information about a project, such as a description? Or even more information about a donation, such as the date? Again, while it is conceivable that additional columns could address this issue, such a solution would be awkward and wasteful. If columns named `DonationDate1` and `DonationDate2` are added, the same uncertainty over the appropriate number of columns exists. Adding a description after every project name produces a lot of redundant data, since each project appears in the table multiple times. Every time a new donation is made, the description of the project would have to be repeated. Such redundancy is very inefficient as seen in the following table:

Donor	Amount1	Project1	Description1	Amount2	Project2	Description2
Marco Pinelli	\$200	Solar Scholars	Powering schools with solar panels			
Victor Gomez	\$100	Pear Creek cleanup	Cleaning up litter and pollutants from Pear Creek	\$100	Danube Land Trust	Purchasing and preserving land in the Danube watershed
Seung Yong Lee	\$150	Forest Asia	Planting trees in Asia			

The underlying problem is that a table is two-dimensional. It consists of columns and rows. Real-world data is usually multi-dimensional. We wish to store not only the data relevant to the donors and donations, but also data that relates to details in the table, such as additional information about the projects.

There is a solution to our problem. Relational databases allow us to create multiple tables of related data. The database designer uses the relationships between these tables to represent multi-dimensional data. This is also why they are called relational databases.

Let's now look at the process of normalization in relational databases.

Normalization

A **schema** (pl. schemata) is the basic organization of the database – the structure of the tables and the relationships among them. The term **normalization** refers to a series of steps used to eliminate redundancy and reduce the chances of data inconsistency in a database's schema. There are six forms of normalization described in relational database theory:

- ❑ First Normal Form
- ❑ Second Normal Form
- ❑ Third Normal Form
- ❑ Boyce Codd Normal Form
- ❑ Fourth Normal Form
- ❑ Fifth Normal Form

In practice, database designers mainly concern themselves with the first three forms. The last three are somewhat atypical in the practical world, but remain important in the realm of academic database theory.

First Normal Form

For a schema to meet the requirements of the **First Normal Form (1NF)**, every record within a table should have the same "shape", which means that every field should contain a single value only, and each row should contain the same fields. Duplicated fields (or groups of fields) should be removed.

In our charity donation example, the repeated donation fields, like Amount1 and Amount2, are a violation of First Normal Form. Some of the records (such as the one with Victor Gomez) use all of the fields, while others (such as Marco Pinelli), use only a few of them. These records clearly do not have the same shape. We will now change the table so that each record represents a single donation, thereby eliminating the duplicated fields:

Donor	Date	Amount	Project	Description
Marco Pinelli	13Dec2002	\$200	Solar Scholars	Powering schools with solar panels
Victor Gomez	15Dec2002	\$100	Pear Creek cleanup	Cleaning up litter and pollutants from Pear Creek
Victor Gomez	15Dec2002	\$100	Danube Land Trust	Purchasing and preserving land in the Danube watershed
Victor Gomez	15Dec2002	\$50	Forest Asia	Planting trees in Asia
Seung Yong Lee	16Dec2002	\$150	Forest Asia	Planting trees in Asia

This arrangement is a significant improvement over our previous schema, since it offers greater flexibility. No matter how many donations Victor Gomez contributes, our new table can easily accommodate it simply by adding records.

An additional requirement of First Normal Form is that each record be uniquely identifiable. As we learned earlier in the chapter, this is accomplished with a primary key. In our original table, the Donor field could serve as the primary key. There was only one record per donor, and knowing the donor name was all that one needed to locate the specific record for that donor.

Our new table is a little more complicated. The donor name alone cannot serve as a primary key, since there may be multiple records for the same donor. For example, the name Victor Gomez does not uniquely identify any one record. Similarly, none of the values in the Date, Amount, or Project fields is necessarily unique. However, the combination of Donor, Date, and Project is unique. This is known as a **composite key**, a key that is composed of more than one field, as opposed to a **simple key**, which uses only one field. While there are many records for Victor Gomez, there is only one record for the same on 15Dec2002 for the Forest Asia project.

Composite primary keys are often the ideal solution for uniquely identifying records in a table. However, in our particular example, the composite key that we have chosen may be disadvantageous too. Although there are no two records in the table with the same combination of values for Donor, Date, and Project, we need to ask ourselves if it is impossible that the same donor could make more than one donation to the same project in a day. Though unlikely, it does seem possible, and we want our database schema to account for all possibilities. In such a situation, even a composite key is inadequate.

The primary keys that we have seen so far are *natural* or *logical keys* – keys that are derived from data that already existed in the table, such as the Donor field in our original table. When neither a simple key nor a composite key can be derived from among the fields in the table, it is common practice to create a new field dedicated to the purpose of serving as primary key. Such a key is known as an artificial key, or a **surrogate key**. Surrogate keys are fields whose values are changed automatically, to a value previously unused in that field with each new record. Let's add a surrogate key to uniquely represent each donation in the table. We'll place it at the beginning of the table, since that is where primary keys are conventionally kept:

DonationID	Donor	Date	Amount	Project	Description
1	Marco Pinelli	13Dec 2002	\$200	Solar Scholars	Powering schools with solar panels
2	Victor Gomez	15Dec 2002	\$100	Pear Creek cleanup	Cleaning up litter and pollutants from Pear Creek
3	Victor Gomez	15Dec 2002	\$100	Danube Land Trust	Purchasing and preserving land in the Danube watershed

Table continued on following page

DonationID	Donor	Date	Amount	Project	Description
4	Victor Gomez	15Dec 2002	\$50	Forest Asia	Planting trees in Asia
5	Seung Yong Lee	16Dec 2002	\$150	Forest Asia	Planting trees in Asia

It is very common for a surrogate key to have a name that both describes the type of entity the table represents (such as a donation) and indicates that the purpose of the field is to identify the record (such as the ID).

This table now meets the requirements of First Normal Form. To summarize those requirements once more:

- ❑ All records in a table should have the same shape, or number of fields. Repetitive fields or groups of fields should be eliminated.
- ❑ Each record should be uniquely identifiable within the table.

Second and Third Normal Forms

Second Normal Form (2NF) and **Third Normal Form (3NF)** are very similar. Both are primarily concerned with eliminating data redundancy within a table. For a schema to be in Second Normal Form, it must meet the following requirements:

- ❑ The schema must meet all requirements of First Normal Form
- ❑ All non-primary-key fields which are dependent on part but not all of a primary key should be removed and placed in a separate table

For a schema to be in Third Normal Form, it must meet the following requirements:

- ❑ The schema must meet all requirements of Second Normal Form
- ❑ All fields which are dependent on a non-primary-key field should be removed and placed in a separate table

The first point of each list is important to note. The steps of normalization are cumulative. A schema cannot conform to Third Normal Form if it does not also conform to Second Normal Form, which in turn means that it must conform to First Normal Form.

The second points of the two lists are similar and may be combined like this:

All non-primary-key fields which are not fully dependent on the primary key should be removed and placed in a separate table.

For example, the `Date` field is an attribute of the donation itself. It describes something about the donation, and therefore the `Date` field is considered to be dependent on the donation's primary key, the `DonationID`. Similarly, the `Project` field modifies the donation. It describes the project that the donation is for. However, the `Description` field does not directly modify the donation; it describes the project. It is thus dependent on the project, which is not the primary key of the table. Therefore, the `Description` field should be removed from the `Donation` table and placed in a separate table specific to the projects:

Name	Description
Solar Scholars	Powering schools with solar panels
Pear Creek cleanup	Cleaning up litter and pollutants from Pear Creek
Danube Land Trust	Purchasing and preserving land in the Danube watershed
Forest Asia	Planting trees in Asia

In the `Donation` table, we now have:

DonationID	Donor	Date	Amount	Project
1	Marco Pinelli	13Dec2002	\$200	Solar Scholars
2	Victor Gomez	15Dec2002	\$100	Pear Creek cleanup
3	Victor Gomez	15Dec2002	\$100	Danube Land Trust
4	Victor Gomez	15Dec2002	\$50	Forest Asia
5	Seung Yong Lee	16Dec2002	\$150	Forest Asia

Not only does the introduction of a new table simplify our original table's design considerably, it also offers a new level of flexibility. We now have a place where we can store additional information about the project, such as the project's director or date of inception. In other words, our schema is no longer two-dimensional. The new table enables us to handle not only details about the donations, but also details about the projects to which donations are made. Now that we have introduced more than one table, it should be noted that the tables' names must be unique within the database.

We need to ensure that our new table also complies with the First Normal Form. It obviously does not have any duplicated fields. Does it have a primary key to uniquely identify each record? The project's name is unique within the table, but again we must ask, is it possible for two projects to have the same name? It is conceivable that the Pear Creek cleanup might be an annual event, and might be regarded as a new project each year, with the same name as the previous year's project. To be safe, it is a good idea to introduce a surrogate key to the `Project` table to serve as the primary key:

ProjectID	Name	Description
1	Solar Scholars	Powering schools with solar panels
2	Pear Creek cleanup	Cleaning up litter and pollutants from Pear Creek
3	Danube Land Trust	Purchasing and preserving land in the Danube watershed
4	Forest Asia	Planting trees in Asia

Foreign Keys

Now that the Project table has a reliable primary key, it makes sense that the Donation table should refer to the ProjectID rather than the name of the project, since the whole point is to definitively identify the project to which the donation was made:

DonationID	Donor	Date	Amount	ProjectID
1	Marco Pinelli	13Dec2002	\$200	1
2	Victor Gomez	15Dec2002	\$100	2
3	Victor Gomez	15Dec2002	\$100	3
4	Victor Gomez	15Dec2002	\$50	4
5	Seung Yong Lee	16Dec2002	\$150	5

In the Donation table, the ProjectID field is what is known as a **foreign key**. It is a field which refers to the primary key of another table. Like primary keys, a foreign key may be either simple or composite, depending on whether the foreign table's primary key is simple or composite. A table may contain any number of foreign keys, since a table may contain data that relates to any number of foreign tables.

To further normalize our schema, we should create a Donor table, since it is likely that we will want to store more information about the donor. We will use a surrogate key, since it is not uncommon for different people to share the same name. In general, fields that represent the names of entities such as the names of people or companies, make poor primary keys due to the possibility of repeated values.

To demonstrate the flexibility of the added data dimension, we'll add a Country (of residence) field to the new table, but it can be easily imagined that organizations would probably also store many other details, such as contact information and the like:

DonorID	Name	Country
1	Marco Pinelli	Italy
2	Victor Gomez	United States
3	Seung Yong Lee	South Korea

The `Donation` table can now refer to the unique `DonorID` as a foreign key:

DonationID	DonorID	Date	Amount	ProjectID
1	1	13Dec2002	\$200	1
2	2	15Dec2002	\$100	2
3	2	15Dec2002	\$100	3
4	2	15Dec2002	\$50	4
5	3	16Dec2002	\$150	4

A Key To Keys

With all of the choices of keys that we have discussed in this chapter, a quick review will help us at this stage. A key may be:

- ❑ **Primary or Foreign**
A primary key uniquely identifies each record in its table. A foreign key uniquely identifies a record in some other table.
- ❑ **Simple or Composite**
A simple key is composed of a single field. A composite key is a combination of fields whose values together identify the record.
- ❑ **Surrogate or Logical**
A surrogate (or artificial) key consists of unique, arbitrary values that abstractly represent the records in the table. A logical (or natural) key consists of the actual data, which may identify the record.

In the `Donation` table, the `DonationID` field is a surrogate or a primary key.

Boyce-Codd Normal Form

Boyce-Codd Normal Form (BCNF) is hierarchically placed between Third Normal Form and Fourth Normal Form and it is normally considered to be a more restrictive form of Third Normal Form. A table is considered to be in Boyce-Codd Normal Form if every determinant (any attribute whose values determine other values with a row) in the table is a candidate key. If a table contains only one candidate key, the 3NF and BCNF are equivalent.

Fourth and Fifth Normal Forms

Our database schema has come a long way from our original awkward ledger sheet. Third Normal Form is generally considered the acceptable level of normalization for professional database applications. Schemata that conform to Third Normal Form have consistent arrangements of keys and relationships with little to no redundancy within their tables. Hands-on database designers (as opposed to theoreticians) tend to regard Fourth and Fifth Normal Forms as belonging to the realm of academia, although real-world examples do sometimes arise.

Fourth Normal Form (4NF) involves multi-valued dependencies (that may contain multiple values for an entity). For example, a project is an entity within our database schema. Suppose we wish to store information about the project's fund-raising events and milestones (there is no relationship between a fund-raising event and a milestone, but each has a relationship to the project). We will quickly realize that each project may have multiple fund-raising events, as well as multiple milestones. As we are well versed with First Normal Form, we know better than to create a table with fields like `FundEvent1`, `FundEvent2`, and so on. However, we might just end up structuring the table like this:

ProjectID	FundEventID	MilestoneID
1	43	644
1	09	645
1	56	
2	43	679
2	110	780

This is really a violation of First Normal Form, because there is no primary key, but for the purpose of this example, we'll allow it to stand for a moment. If it really bothers you, pretend that there is a `RecordID` surrogate key. The quest for normalization can admittedly be an obsessive pursuit.

The Solar Scholars project (ID 1) has three fund-raising events, and only two milestones. Suppose we wish to remove fund-raising event 89 from the project. Should we replace the value 89 with a `Null` (empty) value? Or should we combine the second and third records? This would involve updating one record and deleting the other – an inefficient operation that increases the potential for error.

As you have probably guessed, we should split the table into two. The requirements for Fourth Normal Form are:

- ☐ The schema must meet all requirements of Third Normal Form
- ☐ No table should contain more than one multi-valued dependency

`FundEventID` and `MilestoneID` are both multi-valued dependencies. Each is dependent on the `ProjectID` and each represents a fact for which there may be multiple values per `ProjectID` (multiple fund-raising events per `ProjectID`, and multiple milestones per `ProjectID`). Separating the table into two solves the problem as shown in the two tables below.

In the `Project_FundEvent` table, we have:

ProjectID	FundEventID
1	43
1	89
1	56
2	43
2	110

In the `Project_Milestone` table, we have:

ProjectID	FundEventID
1	644
1	6445
2	679
2	780

This also solves the primary key problem. Each table may now have a composite primary key. The combination of `ProjectID` and `FundEventID` is unique. The reason why database designers rarely invoke Fourth Normal Form is that it is usually rendered difficult as a result of the pursuit of unique identifiers early in the schema-building process. Ensuring that primary keys exist for each table makes it difficult for two multi-valued dependencies to occur in one table in the first place.

Fifth Normal Form (5NF) is the most esoteric and the least applicable in the real world, therefore we will not discuss it in depth. It concerns multiple multi-valued facts that are related to each other as well as to another key (as opposed to the multi-valued dependencies mentioned above, which have relationships only to the `ProjectID`, not to each other). Untangling such complex relationships usually requires splitting tables into more than three related tables. Fifth Normal Form attempts to eliminate all remaining redundancy from a schema by decomposing each table to the point where it cannot be decomposed any further. It requires each field to be a candidate key, which is the main reason why it is generally considered unattainable in real-world applications.

Denormalization

Now that we've invested a valuable chunk of our day in learning about normalization, it's time to introduce the concept of **denormalization**, which is exactly what it sounds like: decreasing a schema's level of normalization. As we've learned, normalization eliminates the data redundancy within a table, which greatly reduces the risk that data may become inconsistent. Why would one wish to reverse this process?

Normalization usually comes at a cost: speed of retrieval. Before normalization, if we wanted to know a donor's name, the dates of the donations, and the name of the project, it was all right there in one record for us to pick. After normalization, we have to go traversing through three or four tables for the same information. In most cases, the extra work is worth it, considering the benefits of data consistency and reduced storage usage. However, in a few rare cases, the speed of data retrieval is the factor that trumps all others. In large databases with complex schemas, one might sometimes require data from twelve or more tables in a single query, and the application may need to perform this type of query hundreds of times per minute. In such situations, a fully normalized database may be unacceptably slow.

Denormalization should not be done early, however. It is a last desperate resort that one should turn to only after exhausting all other options (like query optimization, improved indexing, and database system tuning, all of which will be discussed later in the book). Normally, follow the simple rule:

When in doubt, normalize.

One alternative to denormalizing the base tables (the tables that make up a database) is to create a separate reporting table so that the base tables are left unaffected. For example, suppose that in our previous example, we very frequently need to retrieve a donor's name, donation ID, and the date of the donation. The query often proves to be too slow in providing results. This may not seem realistic, given that it only involves two tables and any modern RDBMS would handle this with break-neck speed, but just use your imagination. We might be tempted to re-enter the donor's name to our `Donation` table:

DonationID	DonorID	Donor	Date	Amount	ProjectID
1	1	Marco Pinelli	13Dec2002	\$200	1
2	2	Victor Gomez	15Dec2002	\$100	2
3	2	Victor Gomez	15Dec2002	\$100	3
4	2	Victor Gomez	15Dec2002	\$50	4
5	3	Seung Young Lee	16Dec2002	\$150	4

This is a heart-breaking departure from everything we've worked so hard to achieve. See the redundancy? See the wasted space?

With a separate reporting table, our three base tables of `Donation`, `Donor`, and `Project` remain beautifully normalized. The schema as a whole is not fully normalized, because the reporting table itself is redundant, but at least all of the redundancy is concentrated and isolated in one table, whose sole job is to provide quick access to data that comes from multiple sources. Thus in the `Donor` table we have:

DonorID	Name	Country
1	Marco Pinelli	Italy
2	Victor Gomez	United States
3	Seung Yong Lee	South Korea

The `Project` table is as follows:

ProjectID	Name	Description
1	Solar Scholars	Powering schools with solar panels
2	Pear Creek cleanup	Cleaning up litter and pollutants from Pear Creek
3	Danube Land Trust	Purchasing and preserving land in the Danube watershed
4	Forest Asia	Planting trees in Asia

The `Donation` table is as follows:

DonationID	DonorID	Date	Amount	ProjectID
1	1	13Dec2002	\$200	1
2	2	15Dec2002	\$100	2
3	2	15Dec2002	\$100	3
4	2	15Dec2002	\$50	4
5	3	16Dec2002	\$150	4

The purpose of the three tables shown above is to properly store the organization's data in a way that is consistent and reliable. They are the core tables on which the organization's applications will be based. In contrast, the `Report_DonorName_Date` table below is solely designed to facilitate a specific report without involving the base tables. We have not abided strictly to normalization with this table in order to provide a single location where the most frequently requested data might be accessed quickly.

The `Report_DonorName_Date` table is as follows:

DonationID	DonorName	Date
1	Marco Pinelli	13Dec2002
2	Victor Gomez	15Dec2002
3	Victor Gomez	15Dec2002
4	Victor Gomez	15Dec2002
5	Seung Yong Lee	16Dec2002

A reporting table is usually used as a data cache – a place to store amalgamated or semi-amalgamated data for fast access, which reduces demand on the main location where the data are stored in a raw state. Depending on the business requirements of the application, it might even be possible to only fill the `Report_DonorName_Date` table periodically, say overnight when the system is least busy. Others among us are not quite as lucky, and have to ensure that the data in the reporting table is no older than ten minutes, or even ten seconds. Even then, the reporting table offers a performance advantage. It is better to query the base tables once every ten seconds than hundreds or thousands of times per minute. Triggers, which will be discussed in Chapter 7, can be useful in keeping a reporting table up-to-date.

Denormalization is not pretty, but it is often helpful. If you absolutely must do it, then do it; but make sure you feel guilty about it, just like the professionals. A simple rule for beginners to database design is never denormalize.

Referential Integrity

Many relational database management systems include mechanisms that enforce a database's referential integrity. Referential integrity is another measure of the consistency of the data in a database. Referential integrity is violated when the relation to which a foreign key refers no longer exists.

For example, if one deletes a donor from the `Donor` table, without also deleting the corresponding donations from the `Donation` table, then the `DonorID` field in the `Donation` record would refer to a non-existent donor. In later chapters, we will discuss a few mechanisms that can be used to enforce referential integrity, including triggers, constraints, transactions, and stored procedures.

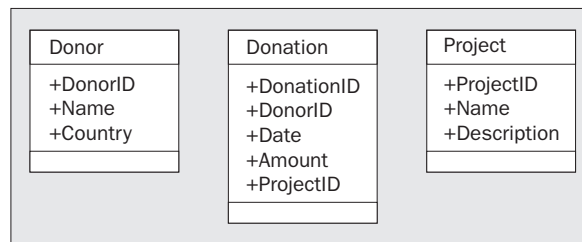
Entity Relationship Diagrams

With databases as with programming, manufacturing, and many other disciplines, before we build we should design. The database's design process begins with the design process of the application as a whole, including consideration of user requirements, design patterns, and system requirements. For more information on these topics, see Chapter 10.

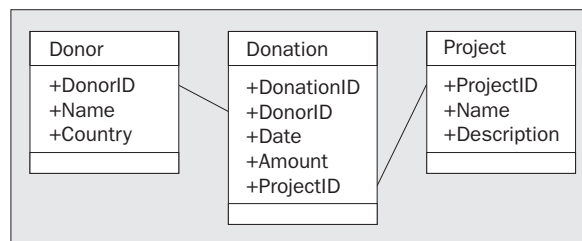
An entity relationship diagram is used to illustrate all of the entities that an application must handle, and the relationships among the entities. This information is the blueprint for building the database tables.

Entities typically correspond to nouns that are involved with the process being automated. In the charity example explained in the chapter, nouns include donors, projects, and donations. Relationships correspond to the verbs. Take a look, and make note of how they interrelate. A donor *makes* donations. A project *has* multiple donations.

In an entity relationship diagram, entities are shown with their attributes listed below them:



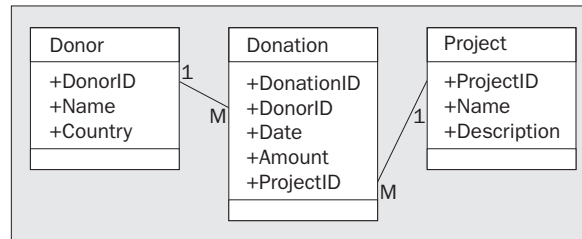
There is no one standard way to represent relationships. Some designers use diamond shapes containing a description of the relationship. Others use lines to connect the entities in a relationship. The line should specifically link each foreign key to its designated primary key as shown in the diagram below:



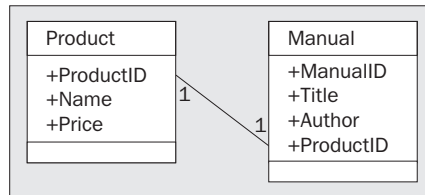
Types of Relationships

Both of the relationships illustrated opposite are one-to-many relationships. Each record in the `Donor` table may correspond to many records in the `Donation` table, however each record in the `Donation` table has only one corresponding record in the `Donor` table. Similarly, each project may have multiple donations, but each donation is made to only one project.

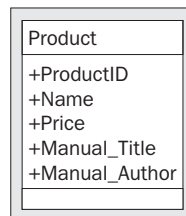
In entity relationship diagrams, the "one" side of the relationship is usually represented with a numeral 1, and the "many" side of the relationship with either an M or the infinity symbol:



Sometimes relationships are one-to-one. Suppose that each of a manufacturer's products comes with one (and only one) user manual, and each manual only describes the product it accompanies:



One-to-one relationships are somewhat rare, since one of the entities can often be represented simply as attributes of the other. Some database designers may choose to collapse the schema above into one entity:



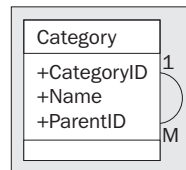
Such a design decision would depend on several factors, such as the performance or significance of the individual entities within the application. For example, if other entities (like an `Author` table) in the schema also have relationships with the manual, then it is probably best to keep the `Manual` entity separate from the `Product` entity.

Many-to-many relationships are usually represented with the use of mediating one-to-many tables. Returning to our charity example, we can see that there is essentially a many-to-many relationship between donors and projects. Each donor may contribute to many projects, and each project may benefit from multiple donors. If we were to try to include the `ProjectIDs` in the `Donor` table, and the `DonorIDs` in the `Project` table, we would end up with a denormalized mess on our hands. The introduction of the `Donation` table deconstructs the many-to-many relationship into two very manageable one-to-many relationships.

A **recursive** relationship occurs when a column in a table refers to the primary key in the same table. This is very useful when describing hierarchical data structures in a table. In the following table of categories for an encyclopedia web site, the `ParentID` field refers to the `CategoryID` field:

CategoryID	Name	ParentID
1	Home	0
2	History	1
3	Mathematics	1
4	Asian History	2
5	Age of Exploration	2
6	Ancient India	4
7	Trigonometry	3
8	Algebra	3

The `History` category parents the `Asian History` category, which in turn parents the `Ancient India` category, and so on. Each record has a relationship with at least one other record in the same table. The following entity relationship diagram demonstrates the relationship:



Summary

In this chapter, we have discussed the principles behind relational database design. Let us skim through them here.

Data is stored in two-dimensional tables consisting of columns (fields) and rows (records).

Multi-dimensional data is represented by a system of relationships among two-dimensional tables. This usually leads to data storage becoming redundant, and also difficult to maintain on account of addition and deletion anomalies. This is only the case if we do not normalize the data.

Normalization is a process by which redundancy and inconsistency are reduced or eliminated from a database's schema. Denormalization is generally undesirable but sometimes necessary for performance reasons. Keys are fields or combinations of fields used to identify records. We also saw how entity relationship diagrams are used to map out the design of a database before it is built.

