Beginning SQL Queries: From Novice to Professional

Copyright © 2008 by Clare Churcher

ISBN-13 (pbk): 978-1-59059-943-3

ISBN-10 (pbk): 1-59059-943-8

ISBN-13 (electronic): 978-1-4302-0550-0

ISBN-10 (electronic): 1-4302-0550-4

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Lead Editor: Jonathan Gennick
Technical Reviewer: Darl Kuhn
Editorial Board: Clay Andres, Steve Anglin, Ewan Buckingham, Tony Campbell, Gary Cornell, Jonathan Gennick, Matthew Moodie, Joseph Ottinger, Jeffrey Pepper, Frank Pohlmann, Ben Renow-Clarke, Dominic Shakeshaft, Matt Wade, Tom Welsh
Project Manager: Beth Christmas
Copy Editors: Marilyn Smith, Kim Wimpsett
Associate Production Director: Kari Brooks-Copony
Production Editor: Ellie Fountain
Compositor: Susan Glinert
Proofreaders: Linda Seifert, Liz Welch
Indexer: Broccoli Information Management
Artist: April Milne
Cover Designer: Kurt Krames
Manufacturing Director: Tom Debolski

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail orders-ny@springer-sbm.com, or visit http://www.springeronline.com.

For information on translations, please contact Apress directly at 2855 Telegraph Avenue, Suite 600, Berkeley, CA 94705. Phone 510-549-5930, fax 510-549-5939, e-mail info@apress.com, or visit http://www.apress.com.

Apress and friends of ED books may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Special Bulk Sales–eBook Licensing web page at http://www.apress.com/info/bulksales.

CHAPTER 1

Relational Database Overview

A *query* is a way of retrieving some subset of information from a database. That information might be a single number such as a product price, a list of members with overdue subscriptions, or some sort of calculation such as the total amount of products sold in the past 12 months. Once we retrieve this subset of data, we might want to update the database records or include the information in some sort of report.

Before getting into the nuts and bolts of how to build queries, it is necessary to understand some of the ideas and terminology associated with relational databases. In particular, it is useful to have a way of depicting how a particular database is put together, that is, what data is being kept in what tables and how everything is interrelated.

It is imperative that any database has been designed to accurately represent the situation it is dealing with. With all the fanciest SQL in the world, you are unlikely to be able to get accurate responses to queries if the underlying database design is faulty. If you are setting up a new database, you should refer to a design book[1] before embarking on the project.

In this chapter, we will look at some of the basic ideas of data models and relational theory so we can get started on formulating queries in SQL. Later chapters will expand on these ideas, as required. You will also learn about two important ways to think about queries: relational algebra and relational calculus.

What Is a Relational Database?

In simple terms, a relational database is a set of tables.[2] Each table keeps information about aspects of one thing, such as a customer, an order, a product, a team, or a tournament. It is possible to set up constraints on the data in individual tables and also between tables. For example, when designing the database, we might specify that an order entered in the `Order` table must exist for a customer who exists in the `Customer` table. How the tables are interrelated can be usefully depicted with a data model.

1. For instance, you can refer to my other Apress book, *Beginning Database Design: From Novice to Professional* (Apress, 2007).
2. Really it's a set of relations, but I'll explain that in the "Introducing Tables" section.

Introducing Data Models

A *data model* provides us with information about how the data items in the database are interrelated. In this book, I will use an example of a golf club that has members who belong to teams and enter tournaments. One convenient way to give an overview of the different tables in a database is by using the class diagram notation from the Unified Modeling Language (UML).[3] In this section, we will look at how to interpret a class diagram.

A *class* is like a template for a set of things (or events, people, and so on) about which we want to keep similar data. For example, we might want to keep names and other details about the members of our golf club. Figure 1-1 shows the UML notation for a `Member` class. The name of the class is in the top panel, and the middle panel shows the *attributes*, or pieces of data, we want to keep about each member. Each member can have a value for `LastName`, `FirstName`, and so on.

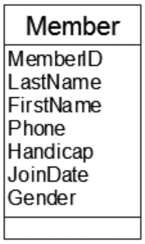


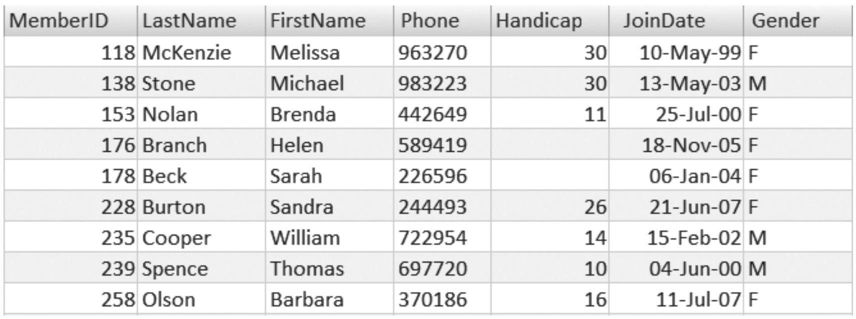
Figure 1-1. *UML representation of a Member class*

Each class in a data model will be represented in a relational database as a *table*. The attributes are the *columns* (often referred to as *fields*) in the table, and the details of each member form the *rows* in the table. Figure 1-2 shows some example data.

The data model can also depict the way the different classes in our database depend on each other. Figure 1-3 shows two classes, `Member` and `Team`, and how they are related.

The pair of numbers at each end of the *plays for* line in Figure 1-3 indicates how many members play for one particular team, and vice versa. The first number of each pair is the minimum number. This is often 0 or 1 and is therefore sometimes known as the *optionality* (that is, it indicates whether a member *must* have an associated team, or vice versa). The second number (known as the *cardinality*) is the greatest number of related objects. It is usually 1 or many (denoted by *n* or ***), although other numbers are possible.

3. If you want more information about UML, then refer to *The Unified Modeling Language User Guide* by Grady Booch, James Rumbaugh, and Ivar Jacobsen (Addison Wesley, 1999).



MemberID	LastName	FirstName	Phone	Handicap	JoinDate	Gender
118	McKenzie	Melissa	963270	30	10-May-99	F
138	Stone	Michael	983223	30	13-May-03	M
153	Nolan	Brenda	442649	11	25-Jul-00	F
176	Branch	Helen	589419		18-Nov-05	F
178	Beck	Sarah	226596		06-Jan-04	F
228	Burton	Sandra	244493	26	21-Jun-07	F
235	Cooper	William	722954	14	15-Feb-02	M
239	Spence	Thomas	697720	10	04-Jun-00	M
258	Olson	Barbara	370186	16	11-Jul-07	F

Figure 1-2. *A table representing the instances of our Member class*

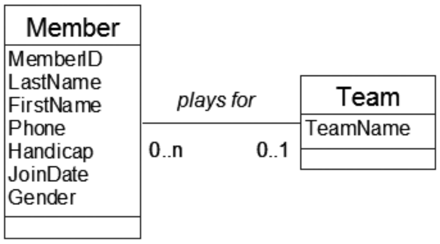


Figure 1-3. *A relationship between two classes*

Relationships are read in both directions. Reading Figure 1-3 from left to right, we have that one particular member doesn't have to *play for* a team and can *play for* at most one team (the numbers 0 and 1 at the end of the line nearest the `Team` class). Reading from right to left, we can say that one particular team doesn't need to have any members and can have many (the numbers 0 and n nearest the `Member` class). A relationship like the one in Figure 1-3 is called a 1-Many relationship (a member can belong to just one team, and a team can have many members). Most relationships in a relational database will be 1-Many relationships.

For members of a team, you might think there should be exactly four members (say for an interclub team). Although this might be true when the team plays a round of golf, our database might record different numbers of members associated with the team as we add and remove players through the year. A data model usually uses 0, 1, and many to model the relationships between tables. Other constraints (such as the maximum number in a team) are more usually expressed with business rules or with UML use cases.[4]

4. For more information, see *Writing Effective Use Cases* by Alistair Cockburn (Addison Wesley, 2001).

Introducing Tables

The earlier description of a relational database as a set of tables was a little oversimplified. A more accurate definition is that a relational database is a set of *relations*. When people refer to tables in a relational database, they generally assume (whether they know it or not) that they are dealing with relations. The reason for the distinction between tables and relations is that there is a well-defined set of operations on relations that allow them to be combined and manipulated in various ways.[5] This is exactly what we need in order to be able to extract accurate information from a database. We won't be covering the actual mathematics in this book, but we will be using the operations. So, in nonmathematical speak, what is so special about relations?

One of the most important features of a relation is that it is a set of *unique* rows.[6] No two rows in a relation can have identical values for every attribute. A table does not generally have this restriction. If we consider our member data, it is clear why this uniqueness constraint is so important. If, in the table in Figure 1-2, we had two identical rows (say for Brenda Nolan), we would have no way to differentiate them. We might associate a team with one row and a subscription payment with the other, thereby generating all sorts of confusion.

The way that a relational database maintains the uniqueness of rows is by specifying a primary key. A *primary key* is an attribute, or set of attributes, that is guaranteed to be different in every row of a given relation. For data such as the member data in this example, we cannot guarantee that all our members will have different names or addresses (a dad and son may share a name and an address and both belong to the club). To help distinguish different members, we have included an ID number as one of the member attributes, or fields. You will find that adding an identifying number (colloquially referred to as a *surrogate key*) is very common in database tables. If `MemberID` is defined as the primary key for the `Member` table, then the database system will ensure that in every row the value of `MemberID` is different. The system will also ensure that the primary key field always has a value. That is, we can never enter a row that has an empty `MemberID` field. These two requirements for a primary key field (uniqueness and not being empty) ensure that given the value of `MemberID`, we can always find a single row that represents that member. We will see that this is very important when we start establishing relationships between tables later in this chapter. Once a table has a primary key nominated, then it satisfies the uniqueness requirement of a relation.

Another feature of a relation is that each attribute (or column) has a domain. A *domain* is a set of allowed values and might be something very general. For example, the domain for the `FirstName` attribute in the `Member` table is just any string of characters, for example, "Michael" or "Helen." The domain for columns storing dates might be any valid date (so that February 29 is allowed only in leap years), whereas for columns keeping quantities,

5. The relational theory was first introduced by the mathematician E. F. Codd in June 1970 in his article "A Relational Model of Data for Large Shared Data Banks" in *Communications of the ACM*: *13*.
6. More accurately, a relation is a set of *tuples*.

the domain might be integer values greater than 0. All database systems have built-in domains or types such as text, integer, or date that can be chosen for each of the fields in a table. Systems vary as to whether users can define their own more highly specified domains that they can use across different tables; however, all good database systems allow the designer to specify constraints on a particular attribute in a table. For example, in a particular table we might specify that a birth date is a date in the past, that the value for a gender field must be "M" or "F", or that a student's exam mark is between 0 and 100. The idea of domains becomes important for queries when we need to compare values of columns in different tables.

When I refer to a database *table* in this book, I mean a set of rows with a nominated primary key to ensure every row is different and where every column has a domain of allowed values. Listing 1-1 shows the SQL code for creating the `Member` table with the attribute names and domains specified. In SQL, the keyword INT means an integer or nonfractional number, and CHAR(n) means a string of characters n long. The code also specifies that `MemberID` will be the primary key. (Listing 1-1 doesn't create the relationship with the `Team` table yet.) The code is fairly self-explanatory.

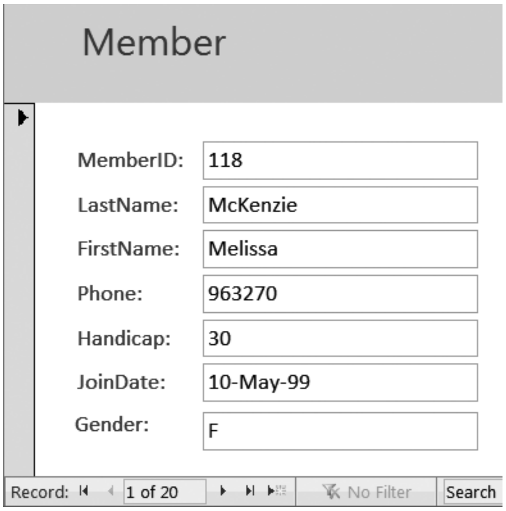
Listing 1-1. *SQL to Create the Member Table*

```
CREATE TABLE Member (
MemberID INT PRIMARY KEY,
LastName CHAR(20),
FirstName CHAR(20),
Phone CHAR(20),
Handicap INT,
JoinDate DATETIME,
Gender CHAR(1))
```

Inserting and Updating Rows in a Table

The emphasis in this book is on getting accurate information out of a database, but the data has to get in somehow. Most database application developers will provide user-friendly interfaces for inserting data into the various tables. Often a form is presented to the user for entering data that may end up in several tables. Figure 1-4 shows a simple Microsoft Access form that allows a user to enter and amend data in the `Member` table.

It is also possible to construct web forms or mechanical readers—(such as the bar-code readers at supermarkets)—that collect data and insert it into a database. Behind all the different interfaces for updating data, SQL update queries are generated. I will show you three types of queries for inserting or changing data just so you get an idea of what they look like. I think you will find them quite easy to understand.


Member

MemberID:	118
LastName:	McKenzie
FirstName:	Melissa
Phone:	963270
Handicap:	30
JoinDate:	10-May-99
Gender:	F

Figure 1-4. *A form allowing entry and updating of data in the Member table*

Listing 1-2 shows the SQL to enter one complete row in our `Member` table. The data items are in the same order as specified when the table was created (Listing 1-1). Note that the date and string values need to be enclosed in single quotes.

Listing 1-2. *Inserting a Complete Row into the Member Table*

```
INSERT INTO Member
VALUES (118, 'McKenzie', 'Melissa', '963270', 30, '05/10/1999', 'F')
```

If many of the data items are empty, we can specify which attributes or fields will have values. If we had only the ID and last name of a member, we could insert just those two values as in Listing 1-3. Remember that we always have to provide a value for the primary key field.

Listing 1-3. *Inserting a Row into the Member Table When Only Some Attributes Have Values*

```
INSERT INTO Member (MemberID, LastName)
VALUES (258, 'Olson')
```

We can also alter records that are already in the database with an update query. Listing 1-4 shows a simple example. The query needs to identify which records are to be changed (the WHERE clause in Listing 1-4) and then specify the field or fields to be updated (the SET clause).

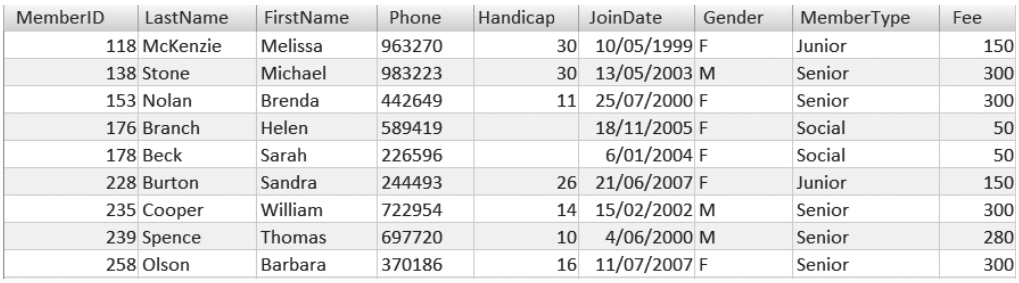
Listing 1-4. *Updating a Row in the Member Table*

```
UPDATE Member
SET Phone = '875077'
WHERE MemberID = 118
```

Designing Appropriate Tables

Even a quite modest database system will have hundreds of attributes: names, dates, addresses, quantities, descriptions, ID numbers, and so on. These all have to find their way into tables, and getting them in the right tables is critical to the overall accuracy and usefulness of the database. Many problems can arise from having attributes in the wrong tables. As a simple illustration of what can go wrong, I'll briefly show the problems associated with having redundant information.

Say we want to add membership types and fees to the information we are keeping about members of our golf club. We could add these two fields to the `Member` table, as in Figure 1-5.



MemberID	LastName	FirstName	Phone	Handicap	JoinDate	Gender	MemberType	Fee
118	McKenzie	Melissa	963270	30	10/05/1999	F	Junior	150
138	Stone	Michael	983223	30	13/05/2003	M	Senior	300
153	Nolan	Brenda	442649	11	25/07/2000	F	Senior	300
176	Branch	Helen	589419		18/11/2005	F	Social	50
178	Beck	Sarah	226596		6/01/2004	F	Social	50
228	Burton	Sandra	244493	26	21/06/2007	F	Junior	150
235	Cooper	William	722954	14	15/02/2002	M	Senior	300
239	Spence	Thomas	697720	10	4/06/2000	M	Senior	280
258	Olson	Barbara	370186	16	11/07/2007	F	Senior	300

Figure 1-5. *Possible Member table*

If the fee for all senior members is the same (that is, there are no discounts or other complications), then immediately we can see there has been a problem with the data entry because Thomas Spence has a different fee from the other senior members. The piece of information about the fee for a senior member is being stored several times, so inevitably inconsistencies will arise. If we formulated a query to find the fee for seniors, what would we expect for an answer? Should it be $300, $280, or both?

The problem here is that (in database parlance) the table is not properly *normalized.* Normalization is a formal way of checking whether attributes are in the correct table. It is outside the scope of this book to delve into normalization, but I'll just briefly show you how to avoid the problem in this particular case.

The problem is that we are trying to keep information about two different things in our `Member` table: information about each member (IDs, names, and so on) and information about membership types (the different fees). This means the `Fee` attribute is in the wrong table. Figure 1-6 shows a better solution with two classes: one for information about members and one for information about membership types. The tables have a 1-Many relationship between them that can be read from left to right as "each member has one membership type" and in the other direction as "a particular membership type can have many associated members."

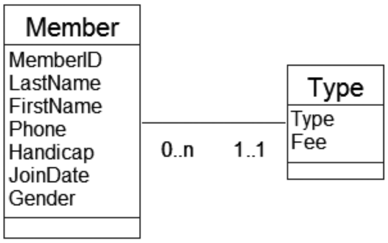
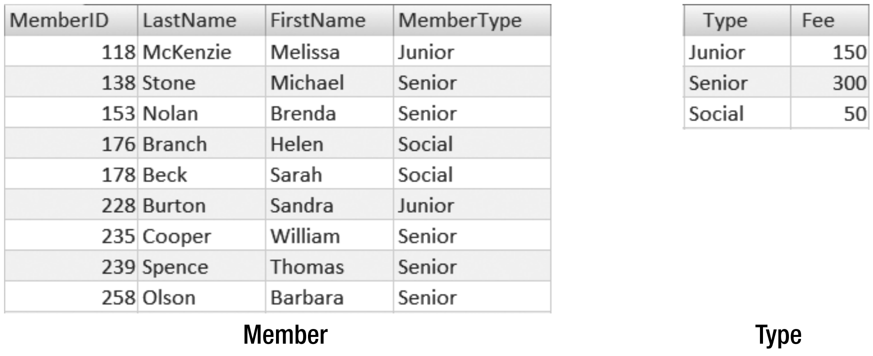


Figure 1-6. *Separating members and their types*

We can represent the model in Figure 1-6 with the two tables in Figure 1-7. (A few of the fields in the `Member` table have been hidden in Figure 1-7 just to keep the size manageable.) You can see that we have now avoided keeping the information about the fee for a senior member more than once, so inconsistencies do not arise. Also, if we need to change the value of the fee, we need to change it in only one place.



MemberID	LastName	FirstName	MemberType
118	McKenzie	Melissa	Junior
138	Stone	Michael	Senior
153	Nolan	Brenda	Senior
176	Branch	Helen	Social
178	Beck	Sarah	Social
228	Burton	Sandra	Junior
235	Cooper	William	Senior
239	Spence	Thomas	Senior
258	Olson	Barbara	Senior

Member

Type	Fee
Junior	150
Senior	300
Social	50

Type

Figure 1-7. *Member and Type tables*

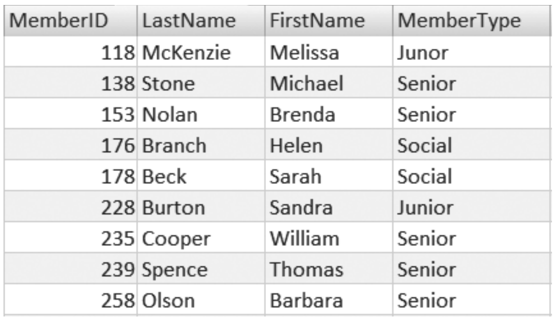
If we need to find out what fee Thomas Spence pays, we now need to consult two tables: the `Member` table to find his type and then the `Type` table to find the fee for that type. The bulk of this book is about how to do just that sort of data retrieval. We can formulate queries to accurately retrieve all sorts of information from several tables in the database.

At the risk of repeating myself, I do want to caution you about the necessity of ensuring that the database is properly designed. The simple model in Figure 1-6 is almost certainly quite unsuitable even for the tiny amount of data it contains. A real club will probably want to keep track of fees and how they change over the years. They may need to keep records of when members graduated from junior to senior. They may offer discounts for prompt payments. Designing a useful database is a tricky job and outside the scope of this book.[7]

Maintaining Consistency Between Tables

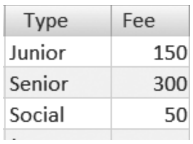
Even the smallest database will have many, many tables. We saw in the previous section that keeping just a tiny amount of data for members required two tables if it was to be accurately maintained. Database systems provide domains or other constraints to ensure that values in particular columns of a given table are sensible, but we can also set up constraints between tables.

Look at the modified data in Figure 1-8. What fee does Melissa McKenzie pay now?



MemberID	LastName	FirstName	MemberType
118	McKenzie	Melissa	Junor
138	Stone	Michael	Senior
153	Nolan	Brenda	Senior
176	Branch	Helen	Social
178	Beck	Sarah	Social
228	Burton	Sandra	Junior
235	Cooper	William	Senior
239	Spence	Thomas	Senior
258	Olson	Barbara	Senior

Member



Type	Fee
Junior	150
Senior	300
Social	50

Type

Figure 1-8. *Inconsistent data between tables*

We can probably make an educated guess that Melissa is probably meant to be a "Junior" rather than a "Junor", but we don't want our database to be second-guessing what it thinks we mean. We can prevent typos like this by placing a constraint called a *foreign key* on the `Member` table. We tell the database that the `MemberType` column in the `Member` table can have only a value that already exists as a primary key value in the `Type` table (for this example, that means it must be either "Junior", "Senior", or "Social"). The terminology for this is to "create the `MemberType` field as a foreign key that references the `Type` table." With this constraint in place, "Junior" is OK because we have a row in the `Type`

7. For more information about database design, refer to my other Apress book, *Beginning Database Design: From Novice to Professional* (Apress, 2007).

table with "Junior" in the primary key, but "Junor" will not be accepted. In general, all 1-Many relationships between classes in a data model are set up this way.

Listing 1-5 shows the SQL for creating the `Member` table with a foreign key constraint.

Listing 1-5. *SQL to Create the Member Table with a Foreign Key*

```
CREATE TABLE Member(
MemberID INT PRIMARY KEY,
LastName CHAR(20),
FirstName CHAR(20),
Phone CHAR(20),
Handicap INT,
JoinDate DATETIME,
Gender CHAR(1),
MemberType CHAR(20) FOREIGN KEY REFERENCES Type)
```

Most database products also have graphical interfaces for setting up and displaying foreign key constraints. Figure 1-9 shows the interfaces for SQL Server and Access. These diagrams, which are essentially implementations of the data model, are invaluable for understanding the structure of the database so we know how to extract the information we require.

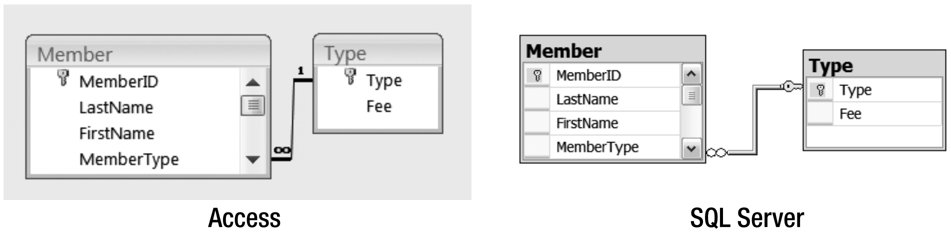
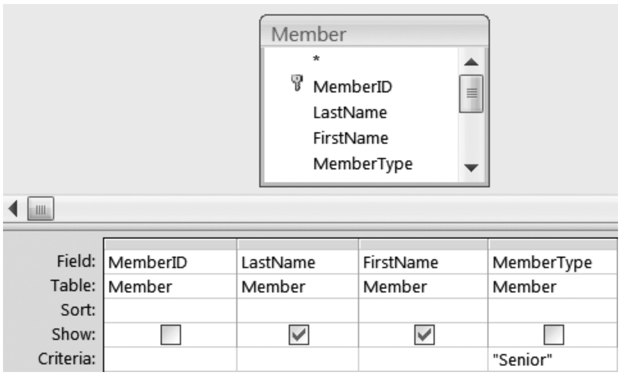


Figure 1-9. *Diagrams for implementing 1-Many relationships using foreign keys*

Retrieving Information from a Database

Now that we have the starting point of a well-designed database consisting of a set of interrelated, normalized tables, we can start to look at how to extract information by way of queries. Many database systems will have a diagrammatic interface that can be very useful for many simple queries. Figure 1-10 shows the Access interface for retrieving the names of senior members from the `Member` table. The check marks denote which columns we want to see, and the Criteria row enables us to specify particular rows.



Field:	MemberID	LastName	FirstName	MemberType
Table:	Member	Member	Member	Member
Sort:				
Show:	☐	✓	✓	☐
Criteria:				"Senior"

Figure 1-10. *Access interface for a simple query on the Member table*

We can express the same query in SQL as shown in Listing 1-6. It contains three clauses: SELECT specifies which columns to return, FROM specifies the table(s) where the information is kept, and WHERE specifies the conditions the returned rows must satisfy. We'll look at the structure of SQL statements in more detail later, but for now the intention of the query is pretty clear.

Listing 1-6. *SQL to Retrieve the Names of Senior Members from the Member Table*

```
SELECT FirstName, LastName
FROM Member
WHERE MemberType = 'Senior'
```

These two methods of expressing a simple query are quite straightforward, but as we need to include more and more tables connected in a variety of ways, the diagrammatic interface rapidly becomes unwieldy and the SQL commands more complex.

Often, it is easier to think about a query in a more abstract way. With a clear abstract understanding of what is required, it then becomes more straightforward to turn the idea into an appropriate SQL statement. There are two different abstract ways to consider queries on a relational database. Because relational theory was developed by a mathematician, it is couched in quite mathematical terms. The two equivalent ways of thinking about queries are called *relational algebra* and *relational calculus*. Do not be alarmed! We will not be getting into quadratic equations or integration—I promise. However, these two methods might take a bit of getting used to, so treat the examples in this chapter as just a taster; we will be going over the details in later chapters.

Relational Algebra: Specifying the Operations

With relational algebra, we describe queries by considering a sequence of operations or manipulations on the tables involved. Some operations act on one table, while others are different ways of combining data from two tables. (Remember that when I talk about tables, I really mean ones with unique rows.) Every time we use one of the operations on a table, the result is another table. This means we can build up quite complicated queries by taking the result of one operation and applying another operation to it.

We will look at all the different operations in detail throughout the book, but just as a simple example we will discuss how to use relational algebra to retrieve the names of the senior members of our golf club. We will need two operations. The *select* operation returns just those rows from a table that satisfy a particular condition. The *project* operation returns just the specified columns.

First we'll get just the rows we need. We can say it like this:

> *Apply the select operation to the `Member` table with the condition that the `MemberType` field must have the value "Senior".*

Clearly, this is all going to get a bit wordy as we apply more and more operations, so it is useful to introduce some shorthand, as shown in Listing 1-7. σ (the Greek letter sigma) stands for the select operation, and the condition is specified in the subscript. For convenience I have called the resulting table `SenMemb`.

Listing 1-7. *The Select Operation to Retrieve the Subset of Rows for Seniors*

$$\text{SenMemb} \leftarrow \sigma_{\text{MemberType='Senior'}}(\text{Member})$$

Figure 1-11 shows the result of this operation. Having retrieved a table with the appropriate rows, we now apply the project operation to get the right columns. Listing 1-8 shows the shorthand for this, where π (pi) denotes the project operation and the columns are specified in the subscript.

Listing 1-8. *The Project Operation to Retrieve a Subset of Columns*

$$\text{Final} \leftarrow \pi_{\text{LastName, FirstName}}(\text{SenMemb})$$

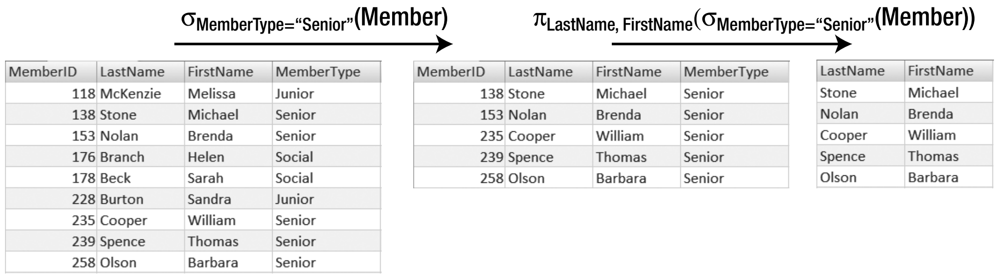
You can express the whole algebra expression in one go, as shown in Listing 1-9.

Listing 1-9. *The Complete Algebra Expression*

$$\text{Final} \leftarrow \pi_{\text{LastName, FirstName}}(\sigma_{\text{MemberType='Senior'}}(\text{Member}))$$

Figure 1-11 shows the original, intermediate, and final tables. Note that the intermediate and final tables are not permanent in the database.

The example in Figure 1-11 shows how we can apply two relational algebra operations in succession to retrieve a final relation with the required data. We do not really need the power of the relational algebra to visualize how to formulate a query this simple; however, most queries are not this simple.



$\sigma_{\text{MemberType="Senior"}}(\text{Member})$

$\pi_{\text{LastName, FirstName}}(\sigma_{\text{MemberType="Senior"}}(\text{Member}))$

MemberID	LastName	FirstName	MemberType
118	McKenzie	Melissa	Junior
138	Stone	Michael	Senior
153	Nolan	Brenda	Senior
176	Branch	Helen	Social
178	Beck	Sarah	Social
228	Burton	Sandra	Junior
235	Cooper	William	Senior
239	Spence	Thomas	Senior
258	Olson	Barbara	Senior

MemberID	LastName	FirstName	MemberType
138	Stone	Michael	Senior
153	Nolan	Brenda	Senior
235	Cooper	William	Senior
239	Spence	Thomas	Senior
258	Olson	Barbara	Senior

LastName	FirstName
Stone	Michael
Nolan	Brenda
Cooper	William
Spence	Thomas
Olson	Barbara

Figure 1-11. *Result of two successive relational algebra operations*

Relational Calculus: Specifying the Result

Relational algebra lets us specify a sequence of operations that eventually result in a set of rows with the information we require. As we will see throughout this book, there may be several different ways of applying a sequence of relational operations that will retrieve the same data. The other method that relational theory provides for describing a query is relational calculus. Rather than specifying *how* to do the query, we describe *what* conditions the resulting data should satisfy. Once again, this may take a bit of getting used to, so we will go over all this more carefully in later chapters.

In nonformal language, a relational calculus description of a query has the following form:

I want the set of rows that obey the following conditions . . .

As with the algebra version, this can become very wordy, so shorthand is convenient, as shown in Listing 1-10.

Listing 1-10. *General Form of a Query Expressed in Relational Calculus*

```
{ m | condition(m) }
```

The part on the left of the bar will contain a description of the attributes or columns we want returned, while the part on the right describes the criteria they must satisfy. The letter m is a way of referring to a particular row (m) in a table, and we will need to introduce other labels when we have several tables to contend with. An example is the best way to clarify what a relational calculus expression means. Listing 1-11 shows the relational calculus for the query to retrieve senior club members.

Listing 1-11. *Relational Calculus to Retrieve Senior Members*

```
{m | Member(m) and m.MemberType = 'Senior'}
```

We can interpret Listing 1-11 like this:

> *Retrieve each row (m) from the* `Member` *table where the* `MemberType` *attribute of that row has the value "Senior".*

We can further refine the query as in Listing 1-12, which retrieves just the names of the senior members.

Listing 1-12. *Relational Calculus to Retrieve the Names of Senior Members*

```
{m.LastName, m.FirstName | Member(m) and m.MemberType = 'Senior'}
```

We can interpret Listing 1-12 like this:

> *Retrieve the values of the* `FirstName` *and* `LastName` *attributes from all the rows m where m comes from the* `Member` *table and the* `MemberType` *attribute of those rows has the value "Senior".*

Why are we doing this? Admittedly, it is over the top to introduce this notation for such a simple query, but as our queries become more complex and involve several tables, it is useful to have a way to express the criteria in an unambiguous way. Also, SQL is based on relational calculus. In Listing 1-12 if you replace the bar (`|`) with the SQL keyword FROM and the "and" with the keyword WHERE, then you essentially have the SQL query of Listing 1-6.

Why Do We Need Both Algebra and Calculus?

It would be reasonable to also ask, why do we need either? As mentioned earlier, we do not need these abstract ideas for simple queries. However, if all queries were simple, you would not be reading this book. In the first instance, queries are expressed in everyday language that is often ambiguous. Try this simple expression: "Find me all students who are younger than 20 or live at home and get an allowance." This can mean different things depending on where you insert commas. Even after we have sorted out what the natural-language expression means, we then have to think about the query in terms of the actual tables in the database. This means having to be quite specific in how we express the query. Both relational algebra and relational calculus give us a powerful way of being accurate and specific.

So, we need a way of expressing our queries. Why not skip all this abstract stuff and go right ahead and learn SQL? Well, the SQL language consists of elements of both calculus and algebra. Ancient versions of SQL were purely based on relational calculus in that you described *what* you wanted to retrieve rather than *how*. In the SELECT clause you specified the attributes, in the FROM clause you listed the tables, and in the WHERE clause you specified the criteria (much as in Listing 1-6). Modern implementations of SQL allow you to explicitly specify algebraic operations such as joins, products, and unions on the tables as well.

There are often several equivalent ways of expressing an SQL statement. Some ways are very much based on calculus, some are based on algebra, and some are a bit of both. I have been teaching queries to university students for several years. For some complicated queries, I often ask the class whether they find the calculus or algebra expressions more intuitive. The class is usually equally divided. Personally I find some queries just feel obvious in terms of relational algebra, whereas others feel much more simple expressed in relational calculus. Once I have the idea pinned down with one or the other, the translation into SQL (or some other query language) is usually straightforward.

The more tools you have at your disposal, the more likely you will be able to express complex queries accurately.

Summary

This chapter has presented an overview of relational databases. We have seen that a relational database consists of a set of tables that represent the different aspects of our data (for example, a table for members and a table for types). Each table has a primary key that is a field(s) that is guaranteed to have a different value for every row, and each field (or column) of the table has a set of allowed values (a domain).

We have also seen that it is possible to set up relationships between tables with foreign keys. A foreign key is a value in one table that has to already exist as the primary key in another table. For example, the value of `MemberType` in the `Member` table must be one of the values in the primary key field of the `Type` table.

It is often helpful to think about queries in an abstract way, and there are two ways to do this. Relational algebra is a set of operations that can be applied to tables in a database. It is a way of describing *how* we need to manipulate the tables to extract the information we require. Relational calculus is a way of describing *what* criteria our required information must satisfy.

SQL is a language for specifying queries on a database. There are usually many equivalent ways to specify a query in SQL. Some are like calculus, and some are like algebra. And some are a bit of both.