MemberID	LastName	FirstName	MemberType	Phone	Handicap	JoinDate	Coach	Team	Gender
118	McKenzie	Melissa	Junior	963270	30	28/05/2005	153		F
138	Stone	Michael	Senior	983223	30	31/05/2009			M
153	Nolan	Brenda	Senior	442649	11	12/08/2006		TeamB	F
176	Branch	Helen	Social	589419		6/12/2011			F
178	Beck	Sarah	Social	226596		24/01/2010			F
228	Burton	Sandra	Junior	244493	26	9/07/2013	153		F
235	Cooper	William	Senior	722954	14	5/03/2008	153	TeamB	M
239	Spence	Thomas	Senior	697720	10	22/06/2006			M
258	Olson	Barbara	Senior	370186	16	29/07/2013			F
286	Pollard	Robert	Junior	617681	19	13/08/2013	235	TeamB	M
290	Sexton	Thomas	Senior	268936	26	28/07/2008	235		M
323	Wilcox	Daniel	Senior	665393	3	18/05/2009		TeamA	M
331	Schmidt	Thomas	Senior	867492	25	7/04/2009	153		M
332	Bridges	Deborah	Senior	279087	12	23/03/2007	235		F
339	Young	Betty	Senior	507813	21	17/04/2009		TeamB	F
414	Gilmore	Jane	Junior	459558	5	30/05/2007	153	TeamA	F
415	Taylor	William	Senior	137353	7	27/11/2007	235	TeamA	M
461	Reed	Robert	Senior	994664	3	5/08/2005	235	TeamA	M
469	Willis	Carolyn	Junior	688378	29	14/01/2011			F
487	Kent	Susan	Social	707217		7/10/2010			F

Figure 2-1. *Retrieving the subset of rows for Senior members.*

Figure 2-1. *Retrieving the subset of rows for Senior members.*

The SQL for the query to retrieve Senior members is as follows:

```
SELECT *
FROM Member
WHERE MemberType = 'Senior'
```

This query has three parts, or *clauses*: The `SELECT` clause says what columns to retrieve. In this case, * means retrieve all the columns. The `FROM` clause says which table(s) the query involves, and the `WHERE` clause describes the condition for deciding whether a particular row should be included in the result. The condition says to check the value in the field `MemberType`. In SQL, when we specify an actual value for a character or text field, we need to enclose the value in single quotes, as in `'Senior'`.

MemberID	LastName	FirstName	MemberType	Phone	Handicap	JoinDate	Coach	Team	Gender
118	McKenzie	Melissa	Junior	963270	30	28/05/2005	153		F
138	Stone	Michael	Senior	983223	30	31/05/2009			M
153	Nolan	Brenda	Senior	442649	11	12/08/2006		TeamB	F
176	Branch	Helen	Social	589419		6/12/2011			F
178	Beck	Sarah	Social	226596		24/01/2010			F
228	Burton	Sandra	Junior	244493	26	9/07/2013	153		F
235	Cooper	William	Senior	722954	14	5/03/2008	153	TeamB	M
239	Spence	Thomas	Senior	697720	10	22/06/2006			M
258	Olson	Barbara	Senior	370186	16	29/07/2013			F
286	Pollard	Robert	Junior	617681	19	13/08/2013	235	TeamB	M
290	Sexton	Thomas	Senior	268936	26	28/07/2008	235		M
323	Wilcox	Daniel	Senior	665393	3	18/05/2009		TeamA	M
331	Schmidt	Thomas	Senior	867492	25	7/04/2009	153		M
332	Bridges	Deborah	Senior	279087	12	23/03/2007	235		F
339	Young	Betty	Senior	507813	21	17/04/2009		TeamB	F
414	Gilmore	Jane	Junior	459558	5	30/05/2007	153	TeamA	F
415	Taylor	William	Senior	137353	7	27/11/2007	235	TeamA	M
461	Reed	Robert	Senior	994664	3	5/08/2005	235	TeamA	M
469	Willis	Carolyn	Junior	688378	29	14/01/2011			F
487	Kent	Susan	Social	707217		7/10/2010			F

Figure 2-2. *Projecting a subset of columns to provide a phone list*

The SQL to retrieve the name and phone columns from the `Member` table is:

```
SELECT LastName, FirstName, Phone
FROM Member
```

MemberID	LastName	FirstName	MemberType	Phone	Handicap	JoinDate	Coach	Team	Gender
118	McKenzie	Melissa	Junior	963270	30	28/05/2005	153		F
138	Stone	Michael	Senior	983223	30	31/05/2009			M
153	Nolan	Brenda	Senior	442649	11	12/08/2006		TeamB	F
176	Branch	Helen	Social	589419		6/12/2011			F
178	Beck	Sarah	Social	226596		24/01/2010			F
228	Burton	Sandra	Junior	244493	26	9/07/2013	153		F
235	Cooper	William	Senior	722954	14	5/03/2008	153	TeamB	M
239	Spence	Thomas	Senior	697720	10	22/06/2006			M
258	Olson	Barbara	Senior	370186	16	29/07/2013			F
286	Pollard	Robert	Junior	617681	19	13/08/2013	235	TeamB	M
290	Sexton	Thomas	Senior	268936	26	28/07/2008	235		M
323	Wilcox	Daniel	Senior	665393	3	18/05/2009		TeamA	M
331	Schmidt	Thomas	Senior	867492	25	7/04/2009	153		M
332	Bridges	Deborah	Senior	279087	12	23/03/2007	235		F
339	Young	Betty	Senior	507813	21	17/04/2009		TeamB	F
414	Gilmore	Jane	Junior	459558	5	30/05/2007	153	TeamA	F
415	Taylor	William	Senior	137353	7	27/11/2007	235	TeamA	M
461	Reed	Robert	Senior	994664	3	5/08/2005	235	TeamA	M
469	Willis	Carolyn	Junior	688378	29	14/01/2011			F
487	Kent	Susan	Social	707217		7/10/2010			F

Figure 2-3. *Retrieving a subset of rows and columns to produce a phone list of Senior members*

The SQL for the query depicted in Figure 2-3 is:

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

Table 2-1. *Comparison Operators*

Operator	Meaning	Examples of True Statement
`=`	Equals	`5=5, 'Junior' = 'Junior'`
`<`	Less than	`4<5, 'Ann' < 'Zebedee'`
`<=`	Less than or equal to	`4<=5, 5<=5`
`>`	Greater than	`5>4, 'Zebedee' > 'Ann'`
`>=`	Greater than or equal to	`5>=4, 5>=5`
`<>`	Not equal	`5<>4, 'Junior' <> 'Senior'`

Just a quick note of caution: in Table 2-1, some of our examples compare numbers, and some compare characters. Recall from Chapter 1 that when we create a table, we specify the type of each field; for example, `MemberID` was declared to be an `INT` (integer or whole number), and `LastName` a `CHAR(20)` (a 20-character field). With fields like integer, comparisons are numerical. With text or character fields, comparisons are alphabetical, and with date and time fields, comparisons are chronological (earlier dates come first).

With comparison operators, we can create many different queries. Table 2-2 shows some examples of Boolean expressions that we can use as conditions in the `WHERE` clause of an SQL statement for selecting rows from the `Member` table.

Table 2-2. *Examples of Boolean Expressions on the Member Table*

Expression	Retrieved Rows
`MemberType = 'Junior'`	All junior members
`Handicap <= 12`	All members with a handicap of 12 or less
`JoinDate >= '01/01/2008'`	Everyone who has joined after the beginning of 2008
`Gender = 'F'`	All the women

```
SELECT *
FROM Member m
WHERE UPPER(m.MemberType) = 'JUNIOR'
```

Logical Operators

We can combine Boolean expressions to create more interesting conditions. For example, we can specify that two expressions must both be true before we retrieve a particular row.

Let's assume we want to find all the junior girls. This requires two conditions to be true: they must be female, and they must be juniors. We can easily express each of these conditions independently. After that, we can use the logical operator `AND` to require that *both* conditions be true:

```
SELECT *
FROM Member m
WHERE m.MemberType = 'Junior' AND m.Gender = 'F'
```

Table 2-3. *Examples of Logical Operators*

Expression	Description of Data
`MemberType = 'Senior' AND Handicap < 12`	Seniors with a handicap under 12
`MemberType = 'Senior' OR Handicap < 12`	All the senior members as well as anyone else with a good handicap (those less than 12)
`NOT(MemberType = 'Social')`	All the members except the social ones (for the current data, that would be just the seniors and juniors)

Figure 2-4 shows a diagrammatic representation of the queries in Table 2-3. Each circle represents a set of rows (that is, those for social members or those for members with handicaps under 12). The shaded area represents the result of the operation.

As our queries get more complicated they will incorporate a number of different tables. Some of the tables may have the same column names, and we might need to distinguish them from each other. In SQL we can preface each of the attributes in our query with the name of the table that it comes from, as shown here:

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

Because typing the whole table name can become tiresome, and also because in some queries we might need to compare data from more than one row of a table, SQL has the notion of an *alias*. Have a look at the following query:

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

```
CREATE VIEW PhoneList AS
SELECT m.LastName, m.FirstName, m.Phone
FROM Member m
```

You can think of `PhoneList` as the instructions to create a "virtual" table that we can use in other queries in the same way that we use real tables. We just need to remember that the virtual table is created on the fly by running the query on the permanent `Member` table and it is then gone. To get our phone list now, we can simply use the `PhoneList` view:

```
SELECT * FROM PhoneList
```

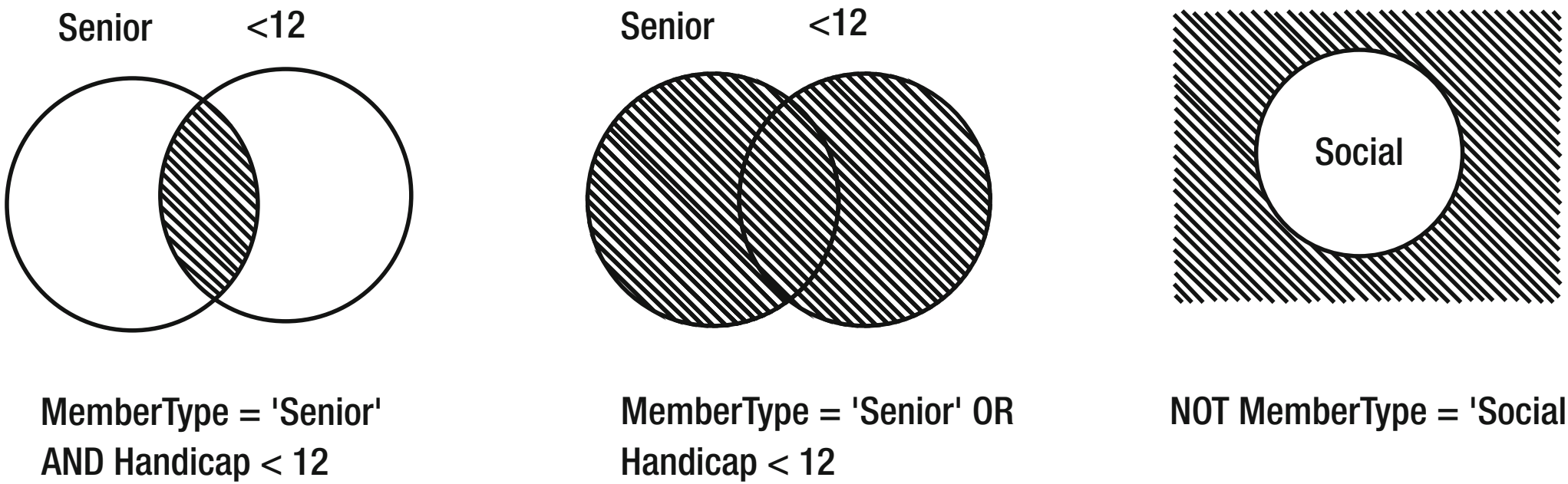

Figure 2-4. *Diagrammatic representation of the logical operators.*

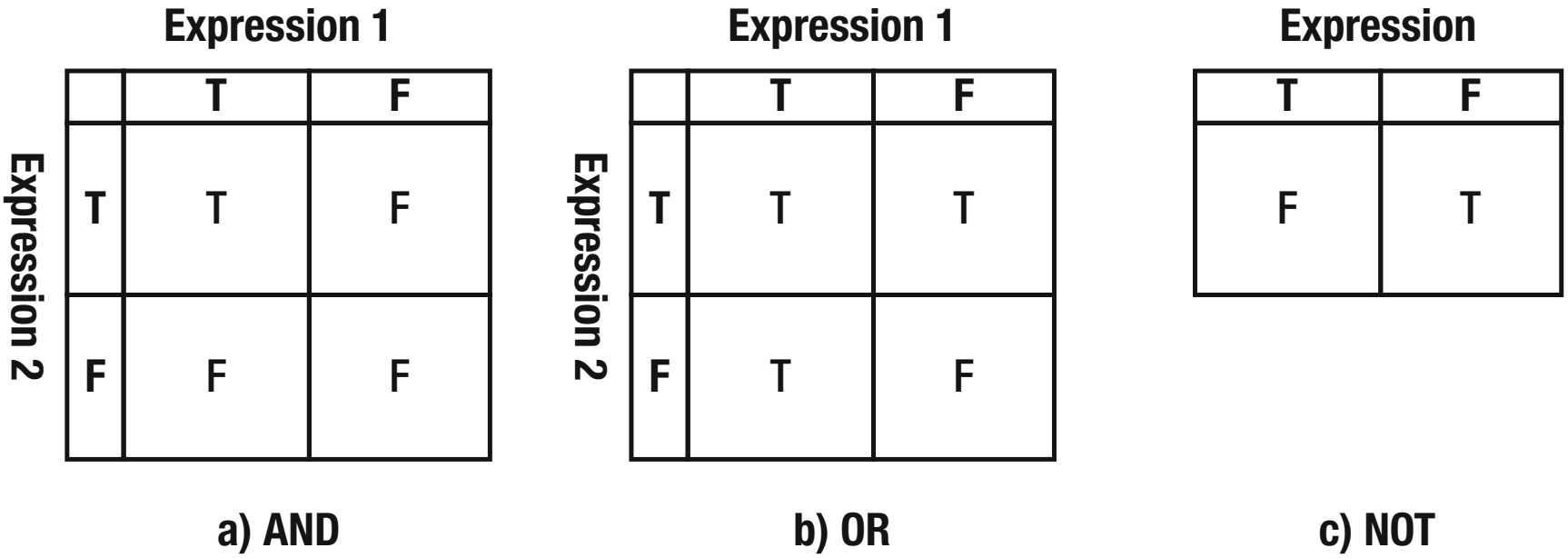

Figure 2-5. *Truth tables for logical operators (T = true, F = false)*

Sometimes it can be a bit tricky turning natural-language descriptions into Boolean expressions. If you were asked for a list that included *all the women and all the juniors* (don't ask why!), you might translate this literally and write the condition `MemberType = 'Junior' AND Gender = 'F'`. However, the `AND` means *both* conditions must be true, so this would give us junior women. What our natural-language statement really means is "I want the row for any member if they are either a woman *or* a junior (or both)." Be careful.

LastName	FirstName	MemberType	Handicap	Gender	JoinDate
McKenzie	Melissa	Junior	30	F	28-May-05
Stone	Michael	Senior	30	M	
Nolan	Brenda	Senior	11	F	12-Aug-06
Branch	Helen	Social		F	06-Dec-11
Beck	Sarah			F	24-Jan-10
Burton	Sandra	Junior	26	F	09-Jul-13
Cooper	William	Senior	14	M	05-Mar-08
Spence	Kim	Senior	10		22-Jun-06
Olson	Barbara	Senior	16	F	29-Jul-13
Pollard	Robert	Junior	19	M	13-Aug-13
Sexton	Thomas	Senior	26	M	28-Jul-08
Wilcox	Daniel	Senior	3	M	18-May-09

Figure 2-6. *Table with missing data*

When there is no value in a cell in a table, it is said to be *null*. Nulls in a database can cause a few headaches. Consider carrying out the following two queries: one to produce a list of male members and the other a list of females. Given that golfers need to identity as either male or female for competition purposes, we might assume that all the members of the club would appear on one list or the other. However, for the data in Figure 2-6, we would leave out Kim Spence. You could argue that the data shouldn't be like that, but we are talking about real people and real clubs with less than accurate and complete data. Maybe Kim forgot (or refused) to fill in the gender part of the application form. We can protect against this by insisting that nulls are not allowed in a particular field when we create a table. The following SQL statement shows how we could make `Gender` a field that always requires a value:

```
CREATE TABLE Member (
MemberID INT PRIMARY KEY,
.....
Gender CHAR(1) NOT NULL,

....)
```

Finding Nulls

Given that in our tables we may have nulls that might cause us problems, it is useful to be able to find them. After we have entered a batch of new members into the database, we can check for problems. We might want to get a list of all the members who don't have a value for `Gender`, say. To do this we can use the SQL phrase `IS NULL`:

```
SELECT *
FROM Member m
WHERE m.Gender IS NULL
```

Alternatively, we might want to retrieve only those members who *do* have a value in a cell. If we want the names and handicaps of only those members who have a value for `Handicap`, we could use the `NOT` operator to create the following query:

```
SELECT *
FROM Member m
WHERE NOT (m.Handicap IS NULL)
```

Comparisons Involving Null Values

Given that we are going to have unexpected nulls in our tables, it is important to know how to deal with them. What rows will match the two conditions shown here?

```
Gender = 'F'
NOT (Gender = 'F')
```

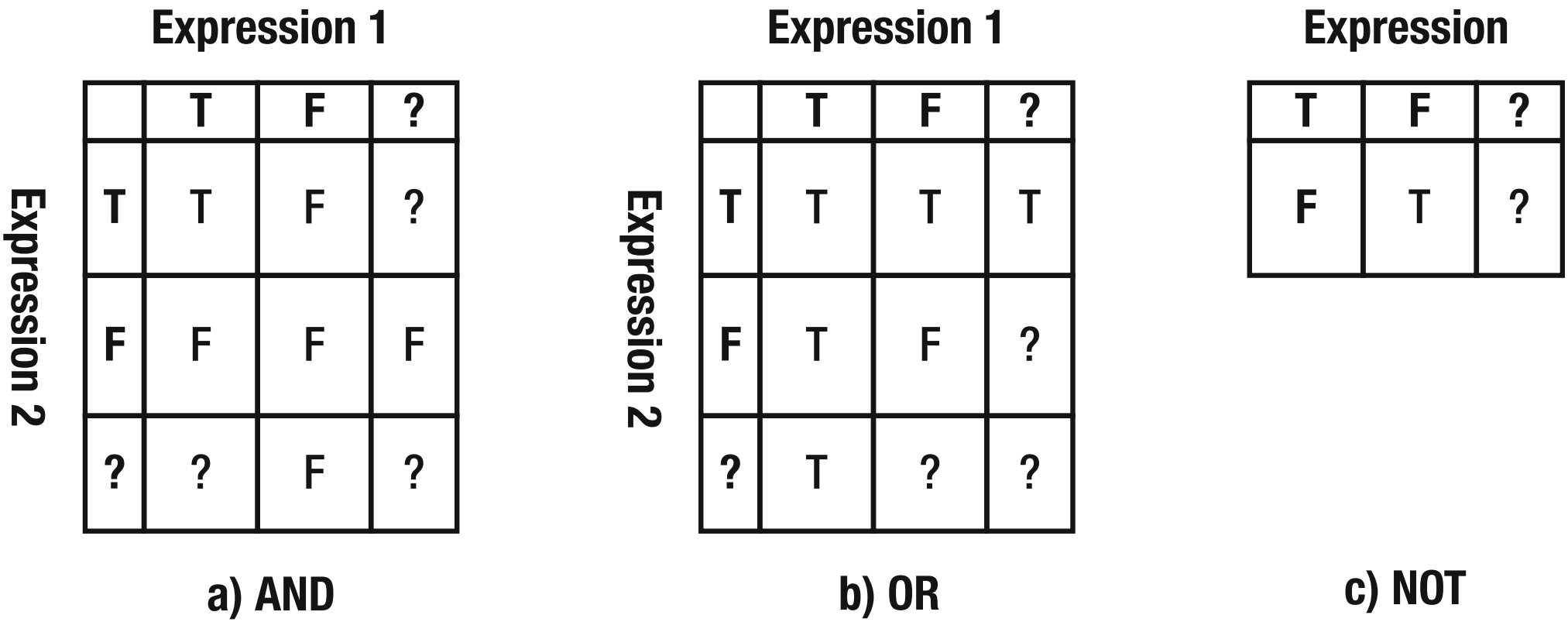

Figure 2-7. *Truth tables with three-valued logic (T = True, F = False, ? = Don't know)*

Figure 2-8. *Projecting the FirstName column from the Member table*

MemberType
Junior
Senior
Senior
Social
Social
Junior
Senior
Senior
Senior
Junior
Senior
Senior
Senior
Senior
Senior
Junior
Senior
Senior
Junior
Social

a) With duplicates

MemberType
Junior
Senior
Social

b) Without duplicates

Figure 2-9. *Projecting the MemberType column from the Member table*

Figure 2-9. *Projecting the MemberType column from the Member table*

It's pretty difficult to think of a situation where you want the duplicated rows in Figure 2-9a. The two operations we have considered sound similar in natural language. "Give me a list of first names" and "Give me a list of membership types" sound like the same sort of question, but they mean quite different things. The first means "Give me a name for each member," and the other means "Give me a list of unique membership types."

What does SQL do? If we say `SELECT MemberType FROM Member`, we will get the output in Figure 2-9a with all the duplicates included. If we do not want the duplicates, then we can use the keyword `DISTINCT`:

```
SELECT DISTINCT m.MemberType
FROM Member m
```

Whether or not you keep the duplicates depends very much on the information you require, so you need to give it careful thought. If you were expecting the set of rows in Figure 2-9b and got Figure 2-9a, you would most likely notice. With the two sets of rows in Figure 2-8, it is much more difficult to spot that you have perhaps made a mistake. Get into the habit of thinking about duplicates for all your queries.

```
SELECT *
FROM Member m
ORDER BY m.LastName
```

We can order by two or more values. For example, if we want to order Senior members with the same LastName by the value of their FirstName, we can include those two attributes (in that order) in the ORDER BY clause:

```
SELECT *
FROM Member m
WHERE m.MemberType = 'Senior'
ORDER BY m.LastName, m.FirstName
```

The type of a field determines how the values will be ordered. By default, text fields will be ordered alphabetically, number fields will be ordered numerically (smallest first), and date and time fields chronologically (earlier dates and times first). We can also specify that the order be reversed with the keyword DESC (for descending). There is an equivalent keyword ASC (for ascending), which is the default if neither is specified. The following will return member names and handicaps ordered in descending order; i.e., with the highest value of handicap first:

```
SELECT m.Lastname, m.FirstName, m.Handicap
FROM Member m
ORDER BY m.Handicap DESC
```

The way nulls are ordered in any output depends on the application; you will need to check. For example, in SQL Server and Microsoft Access, nulls will appear at the top of an ascending list and the bottom of a descending list. Oracle provides keywords such as NULLS FIRST and NULLS LAST so you can choose where the null values go. A little trick to get your nulls at the bottom of an ascending list in SQL Server is to use a case statement:

```
SELECT m.LastName, m.FirstName, m.Handicap
FROM Member m
ORDER BY (CASE
            WHEN m.Handicap IS NULL THEN 1
            ELSE 0
          END), m.Handicap
```

```
SELECT COUNT(*) FROM Member
```

We can also count a subset of rows by adding a `WHERE` clause to specify those rows we want to include. For example, we can use the following query to count the number of senior members:

```
SELECT COUNT(*) FROM Member m
WHERE m.MemberType = 'Senior'
```

Because we have just been talking about nulls and duplicate values, it is worth briefly mentioning here how these will affect our counts. Rather than use * as a parameter to the `COUNT` function so that it counts all the rows, we can put an attribute such as `Handicap` in the parentheses. If we do this only those rows with a value in the `Handicap` field will be included in the count.

```
SELECT COUNT(Handicap) FROM Member
```

We can also specify that we want to count the number of unique values for an attribute. If we want to know how many different values of `MemberType` appear in the `Member` table then we can use the following query:

```
SELECT COUNT(DISTINCT MemberType) FROM Member
```

MemberID	LastName	FirstName
118	McKenzie	Melissa
138	Stone	Michael
153	Nolan	Brenda
176	Branch	Helen
178	Beck	Sarah
228	Burton	Sandra
235	Cooper	William
239	Spence	Thomas
258	Olson	Barbara
286	Pollard	Robert
290	Sexton	Thomas
323	Wilcox	Daniel
331	Schmidt	Thomas
332	Bridges	Deborah
339	Young	Betty
414	Gilmore	Jane
415	Taylor	William
461	Reed	Robert
469	Willis	Carolyn
487	Kent	Susan

a) Member (Some columns)

MemberID	TourID	Year
118	24	2014
228	24	2015
228	25	2015
228	36	2015
235	38	2013
235	38	2015
235	40	2014
235	40	2015
239	25	2015
239	40	2013
258	24	2014
258	38	2014
286	24	2013
286	24	2014
286	24	2015
415	24	2015
415	25	2013
415	36	2014
415	36	2015
415	38	2013

b) Entry

TourID	TourName
24	Leeston
25	Kaiapoi
36	WestCoast
38	Canterbury
40	Otago

c) Tournament

```
SELECT e.MemberID
FROM Entry e
WHERE e.TourID = 36 AND e.Year = 2015
```

Figure 2-10. *Introducing the Tournament and Entry tables*

Say we wanted to find the members who have entered *both* tournaments 36 and 38. There is a temptation to again use the AND operator and write the query as follows:

```
SELECT e.MemberID
FROM Entry e
WHERE e.TourID = 36 AND e.TourID= 38
```

Can you work out what this query will return? This is where it is helpful to think in terms of the row variable e investigating each row in table Entry as in Figure 2-11.

	MemberID	TourID	Year
	286	24	2014
	286	24	2015
	415	24	2015
	415	25	2013
	415	36	2014
e ☞	415	36	2015
	415	38	2013
	415	38	2015
	415	40	2013
	415	40	2014
	415	40	2015

Figure 2-11. *The row variable e investigates each row independently.*

Incorrectly Using a WHERE Clause to Answer Questions with the Word "not"

Now let's consider another common error. It is easy to find the people who have entered tournament 38 with the condition `e.TourID = 38`. It is tempting to try to retrieve the people who have *not* entered tournament 38 by changing the condition slightly. Can you figure out what rows the following SQL query will retrieve?

```
SELECT e.MemberID
FROM Entry e
WHERE e.TourID <> 38
```

What about the row that the finger is pointing to in Figure 2-11? Does this satisfy `e.TourID <> 38`? It certainly does. But this doesn't mean 415 hasn't entered tournament 38 (the following row says he did). The query, in fact, returns all the people who have entered some tournament that isn't tournament 38 (which is unlikely to be a question you'll ever want to ask!).

This is another type of question that can't be answered with a simple `WHERE` clause that looks at independent rows in a table. In fact, we can't even answer this question with a query that involves only the `Entry` table. Member 138, Michael Stone, has not entered tournament 38, but he doesn't even get a mention in the `Entry` table because he has never entered any tournaments at all. We'll see how to deal with questions like this in Chapter 7.