In this chapter, we will look at subqueries and two new SQL keywords, EXISTS and IN. We will see how to use subqueries as an alternative way to approach some of the queries we have already done and also how nesting will open up other possibilities.

IN Keyword

The IN keyword allows us to select rows from a table, where the condition allows an attribute to have one of several values. For example, if we wanted to retrieve the member IDs from the rows in our Entry table for tournaments with ID 36, 38, or 40, we could do this with a Boolean OR operator as in the following query:

```
SELECT e.MemberID
FROM Entry e
WHERE e.TourID = 36 OR e.TourID = 38 OR e.TourID = 40;
```

Clearly, statements of this type will start to become unwieldy as the number of possible options grows. Using the IN keyword, we can construct a more compact statement where the set of possible values are enclosed in parentheses and separated by commas. In the following query, each row of Entry is investigated, and if TourID is one of the values in the parentheses, then the WHERE condition is true, and that row will be returned:

```
SELECT e.MemberID
FROM Entry e
WHERE e.TourID IN (36, 38, 40);
```

It is possible to combine IN with the logical operator NOT. However, you need to be very careful. Consider the following query:

```
SELECT e.MemberID
FROM Entry e
WHERE e.TourID NOT IN (36, 38, 40);
```

The preceding query will return the IDs of members who have entered any tournament that is not in the list. Be aware though that those members may have entered one of the tournaments in the list as well. We will look at how to accurately answer questions such as "who has not entered these tournaments" later in this chapter.

Using IN with Subqueries

The real usefulness of the IN keyword is that we can use another SQL statement to generate the set of values. For example, the reason that someone may have been interested in tournaments 36, 38, and 40 might have been because they are the current Open tournaments. Rather than list the Open tournaments individually, we can use another SQL query to generate the set of values we require. The list will be reconstructed each time the query is run so that the set of Open tournaments will remain current as the data changes.

Let's look at a specific example of using a query to generate the set of values for the IN clause. I've reproduced a few of the columns of the Member table along with the Entry and Tournament tables in Figure 4-1.

MemberID	LastName	FirstName
118	McKenzie	Melissa
138	Stone	Michael

MemberID	TourID	Year
118	24	2014

TourID	TourName	TourType
24	Leeston	Social
25	Kaiapoi	Social

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

(Some columns) Member

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
415	38	2015
415	40	2013
415	40	2014
415	40	2015

Entry

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

Tournament

Figure 4-1. *Member, Entry, and Tournament tables*

The query to generate the set of IDs for the Open tournaments is:

```
SELECT t.TourID
FROM Tournament t
WHERE t.TourType = 'Open';
```

Now we can replace the list of explicit values (36, 38, 40) in the previous queries with the preceding SQL statement:

```
SELECT e.MemberID
FROM Entry e
WHERE e.TourID IN (
     SELECT t.TourID
     FROM Tournament t
     WHERE t.TourType = 'Open');
```

The `SELECT` statement inside the parentheses is sometimes referred to as a *subquery*. To work correctly with the `IN` keyword, the inner part of the query must return a list of single values. I have indented it only to make it easier to read (SQL will ignore the added whitespace). You can understand a nested query by reading it from the "inside out." The inside `SELECT` statement retrieves the set of required tournament IDs from the `Tournament` table, and then the outside `SELECT` finds us all the entries from the `Entry` table for tournaments `IN` that set.

To aid in understanding, it is possible to add comments to SQL statements. In the statement that follows the line beginning with -- is a comment and will be ignored. It is also possible to use /* and */ around a block of more than one line of code.

```
SELECT e.MemberID
FROM Entry e
WHERE e.TourID IN (
     -- Subquery returns IDs of Open tournaments
     SELECT t.TourID
     FROM Tournament t
     WHERE t.TourType = 'Open');
```

Have another look at the tables in Figure 4-1. How else might we have retrieved entries for Open tournaments? We carried out similar queries in the previous chapter using a join. We can join the two tables, `Entry` and `Tournament`, on their common fields `TourID`, select just those rows that are for Open tournaments, and then project the `MemberID` column. See the following:

```
SELECT e.MemberID
FROM Entry e INNER JOIN Tournament t ON e.TourID = t.TourID
WHERE t.TourType = 'Open';
```

The SQL statements with and without the subquery retrieve the same information. As I've said a number of times, there are often several different ways to write a query in SQL. The more methods you are familiar with, the more likely you will be able to find a way to express a complicated query.

In the previous section we constructed two SQL statements for retrieving member IDs for members who have entered an Open tournament. One used a subquery and one a join. To find who has not entered an Open tournament, one might attempt changing `IN` to `NOT IN` in the subquery example, as follows:

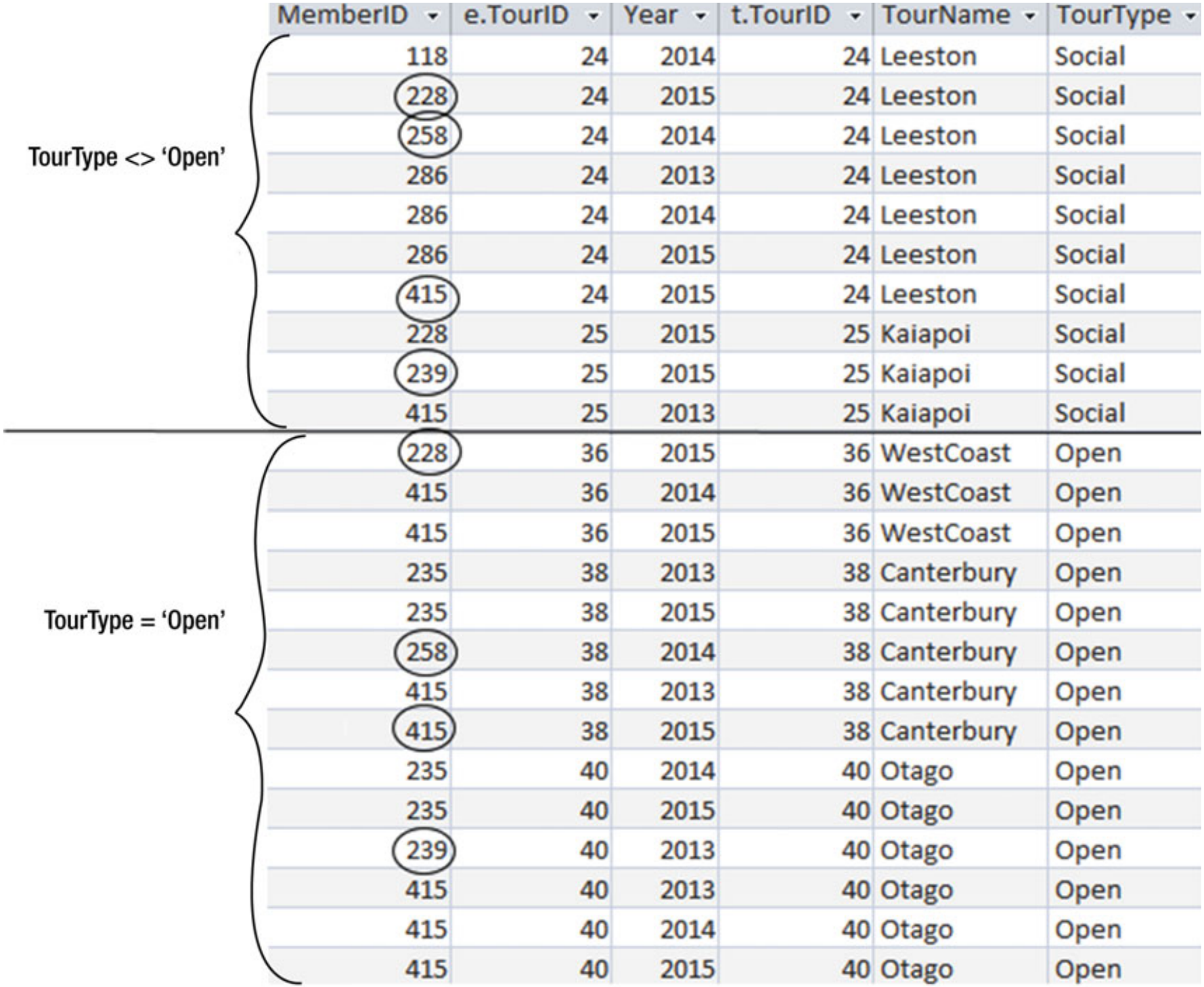
```
SELECT e.MemberID
FROM Entry e
WHERE e.TourID NOT IN
     (SELECT t.TourID
     FROM Tournament t
     WHERE t.TourType = 'Open');
```

In the join example there is a temptation to amend `t.TourType = 'Open'` to `t.TourType <> 'Open'`:

```
SELECT e.MemberID
FROM Entry e INNER JOIN Tournament t ON e.TourID = t.TourID
WHERE t.TourType <>'Open';
```

Carefully think about which rows will be returned by these two queries. They in fact both return the same set of rows, but those rows may include members who have entered an Open tournament as well as those who have not.

The table in Figure 4-2 shows the result of the inner join between `Entry` and `Tournament`. The bottom set of rows are all for Open tournaments, and these will be retrieved by a query that has the condition `WHERE t.TourType = 'Open'`. The top set of entries is for tournaments other than Open and will be retrieved by the query which has the condition `WHERE t.TourType <> 'Open'`.



MemberID	e.TourID	Year	t.TourID	TourName	TourType
118	24	2014	24	Leeston	Social
228	24	2015	24	Leeston	Social
258	24	2014	24	Leeston	Social
286	24	2013	24	Leeston	Social
286	24	2014	24	Leeston	Social
286	24	2015	24	Leeston	Social
415	24	2015	24	Leeston	Social
228	25	2015	25	Kaiapoi	Social
239	25	2015	25	Kaiapoi	Social
415	25	2013	25	Kaiapoi	Social
228	36	2015	36	WestCoast	Open
415	36	2014	36	WestCoast	Open
415	36	2015	36	WestCoast	Open
235	38	2013	38	Canterbury	Open
235	38	2015	38	Canterbury	Open
258	38	2014	38	Canterbury	Open
415	38	2013	38	Canterbury	Open
415	38	2015	38	Canterbury	Open
235	40	2014	40	Otago	Open
235	40	2015	40	Otago	Open
239	40	2013	40	Otago	Open
415	40	2013	40	Otago	Open
415	40	2014	40	Otago	Open
415	40	2015	40	Otago	Open

Figure 4-2. *TourType = 'Open' versus TourType <> 'Open'*

Figure 4-3. *Members who have entered Open tournaments, other tournaments, or both*

Now let's return to the original question. Which members have not entered an Open tournament? We have to be careful to differentiate the two sets depicted in Figure 4-4.

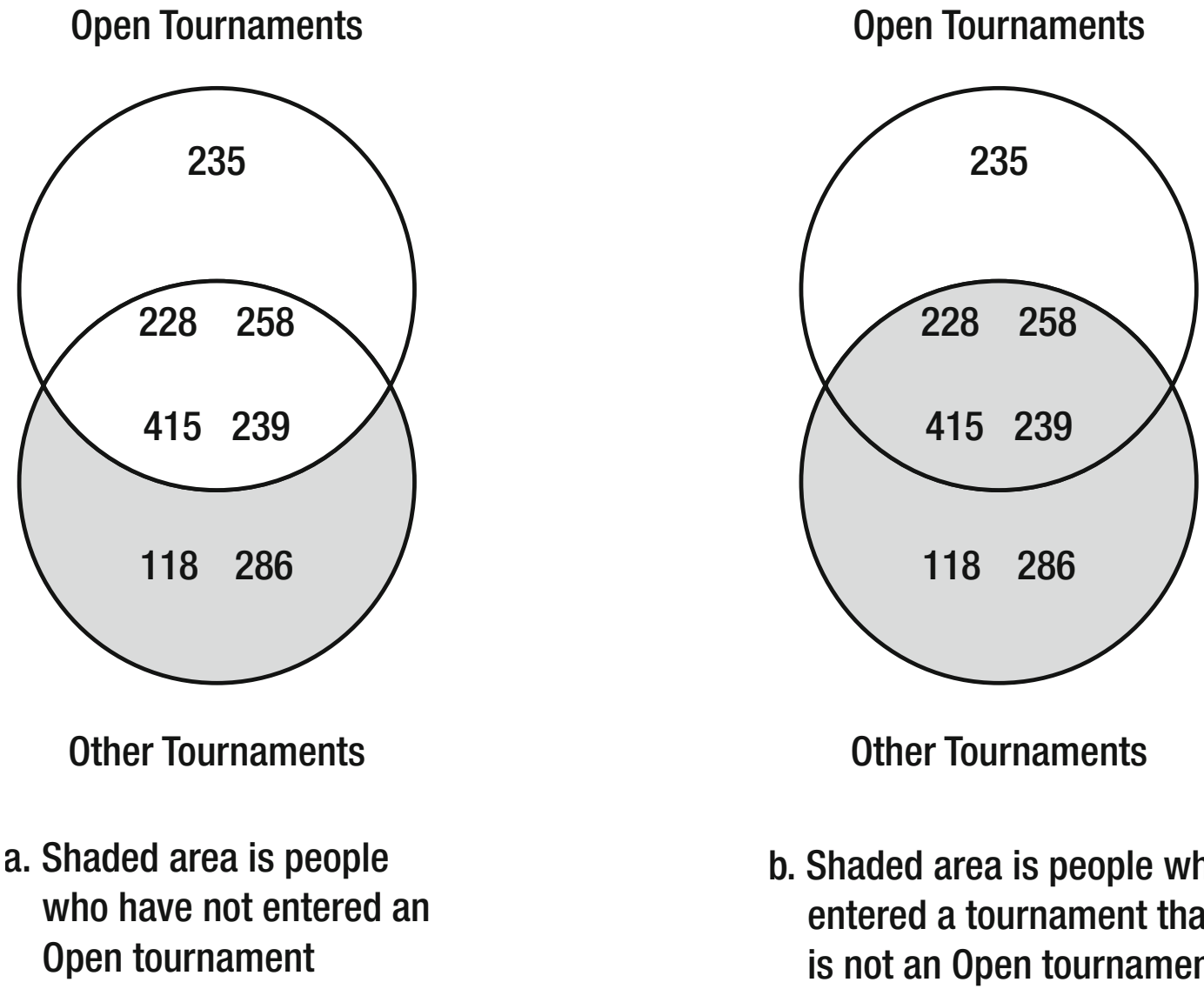


Figure 4-4. *It is important to be careful to distinguish the SQL for these two situations*

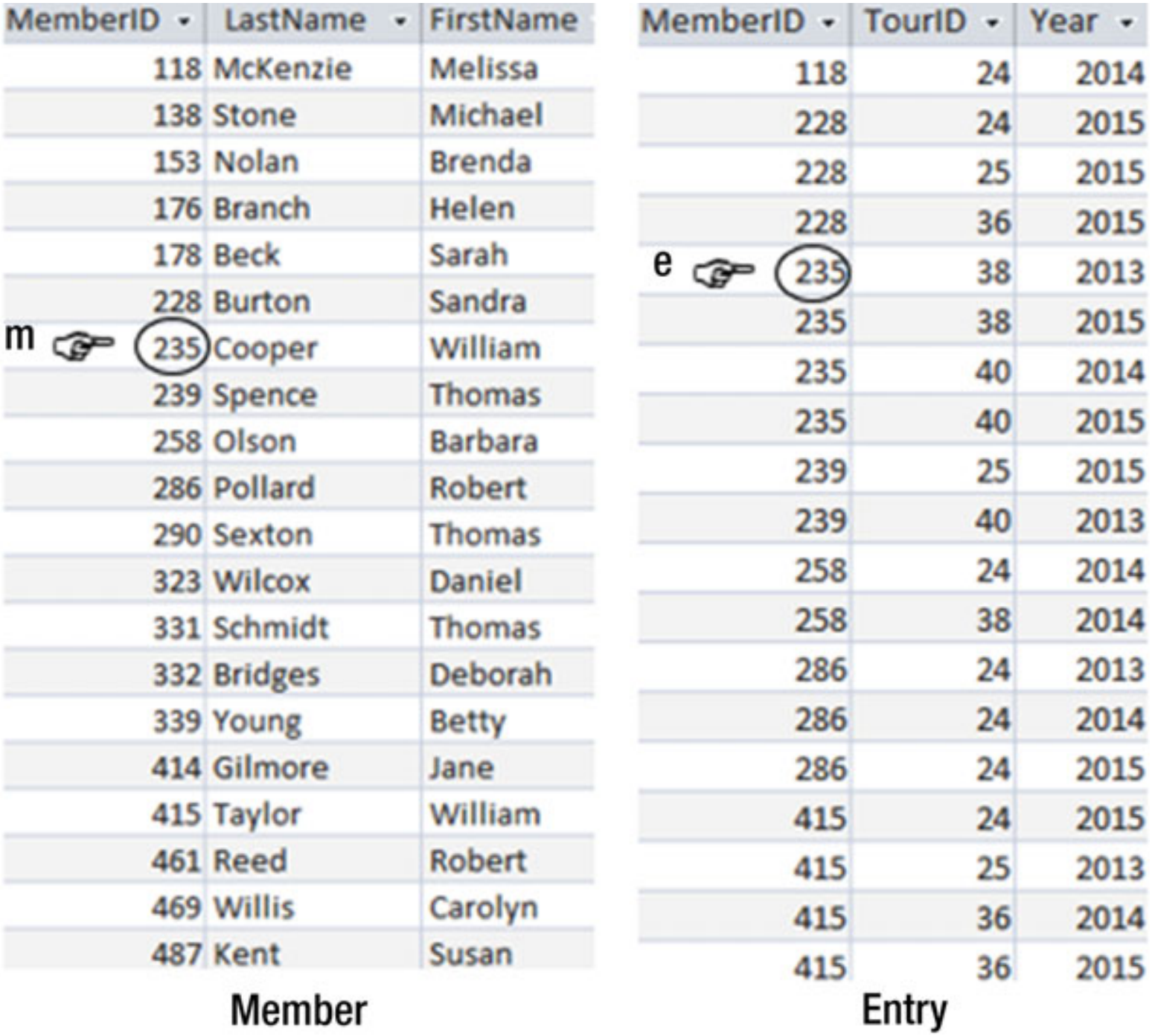


Figure 4-5. *William Cooper has entered a tournament because a matching row exists in the Entry table*

Figure 4-5. *William Cooper has entered a tournament because a matching row exists in the Entry table*

We can translate the statement

> *I'll write out the names from row* `m`*, where* `m` *comes from the* `Member` *table, if there exists a row* `e` *in the* `Entry` *table where* `m.MemberID = e.MemberID`*.*

almost directly into SQL with the use of the keyword `EXISTS`:

```
SELECT m.LastName, m.FirstName
FROM Member m
WHERE EXISTS
     (SELECT * FROM Entry e WHERE e.MemberID = m.MemberID);
```

This is another example of a nested query where we have two SQL `SELECT` statements, one inside the other. This one is a little different from the simpler example we saw earlier in the chapter. The `WHERE` condition in the inner query refers to part of the row being considered in the outer query; that is, `e.MemberID = m.MemberID`. I find the easiest way to interpret these nested queries is with reference to a diagram like Figure 4-5. Variable `m` is checking each row in the `Member` table. The inner query is looking for a row in the `Entry` table with the same value for `MemberID` as the row under consideration in the `Member` table. If such a row (or several such rows) `EXIST`, then we are in business.

```
SELECT m.Lastname, m.FirstName
FROM Member m
WHERE NOT EXISTS
     (SELECT * FROM Entry e WHERE e.MemberID = m.MemberID);
```

The `NOT EXISTS` construction will look through every row `e` in the `Entry` table, checking whether there is a row matching the `MemberID` of the current row in the `Member` table. The name of the member will be retrieved only if *no* matching row is found.

Now we have enough ammunition to tackle the query about members who have not entered an Open tournament. Check out Figure 4-6 to decide if William Cooper should be included in the result.

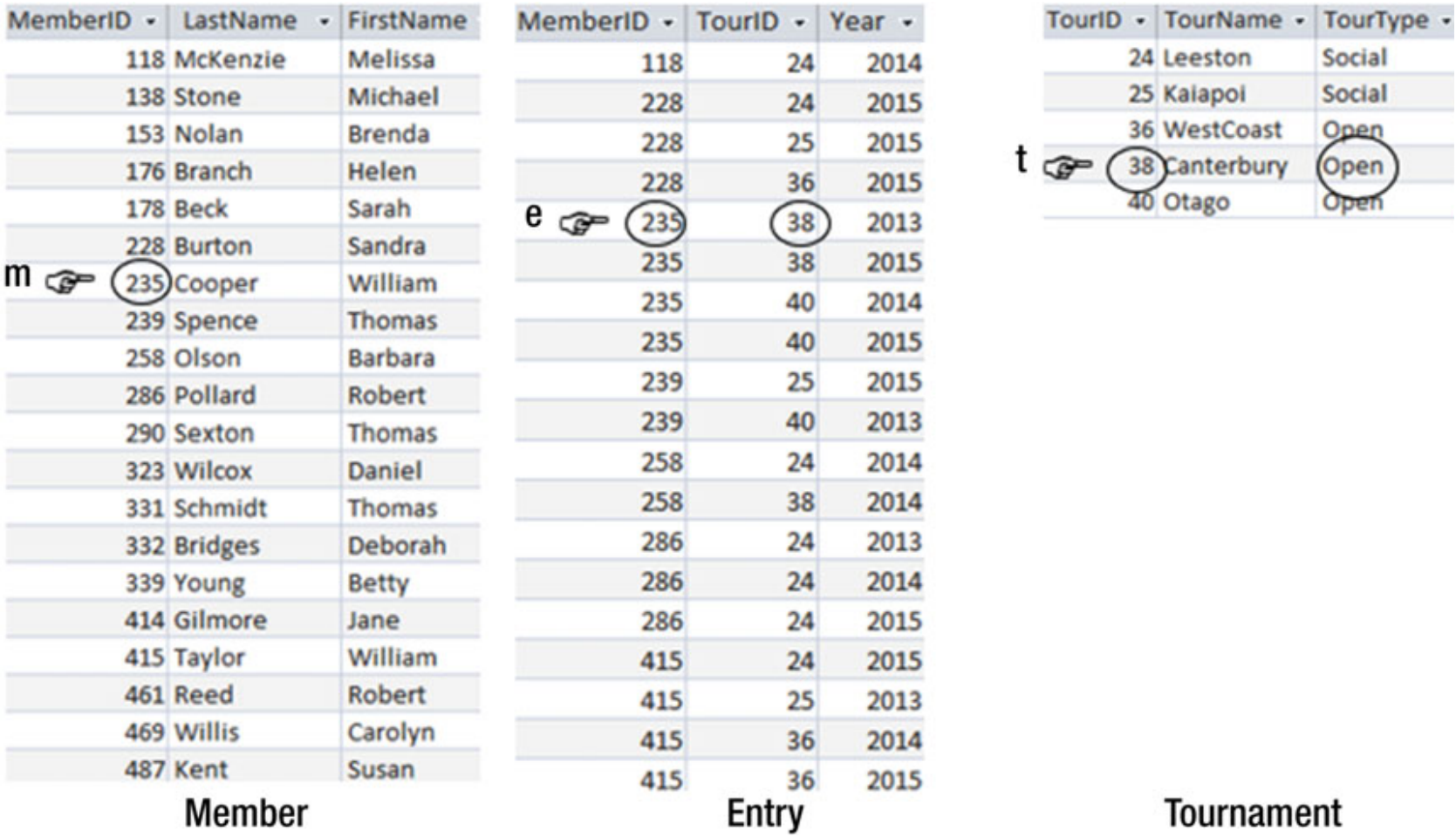


Figure 4-6. *There does exist an entry for an Open tournament for William Cooper*

The rows indicated in Figure 4-6 show that there does exist an entry for William Cooper, so we will not include him our result.

Now, look at this natural language statement that describes Figure 4-6:

> *I'll write out the names from row* `m`*, where m comes from the* `Member` *table, so long as there does not exist (a row* `e` *in the* `Entry` *table where* `m.MemberID = e.MemberID` *along with a row* `t` *in the Tournament table where* `e.TourID = t.TourID and t.TourType = 'Open'`*)*

The SQL reflecting the preceding statement is:

```
SELECT m.Lastname, m.FirstName
FROM Member m
WHERE NOT EXISTS
     (SELECT * FROM Entry e, Tournament t
     WHERE m.MemberID = e.MemberID
     AND e.TourID = t.TourID AND t.TourType = 'Open');
```

We will look at the process approach to queries like this one when we cover set operations in Chapter 7.

Inner Queries Returning a Single Value

Inner queries that return a single value are often useful in the situation where you are simply retrieving subset of rows. Let's consider the handicaps of our members, as shown in Figure 4-7.

MemberID	LastName	FirstName	Handicap
118	McKenzie	Melissa	30
138	Stone	Michael	30
153	Nolan	Brenda	11
176	Branch	Helen	
178	Beck	Sarah	
228	Burton	Sandra	26
235	Cooper	William	14
239	Spence	Thomas	10
258	Olson	Barbara	16
286	Pollard	Robert	19
290	Sexton	Thomas	26

Figure 4-7. *Part of the Member table showing names and handicaps*

If we want to find those members with a handicap of less than 16, then this can be done simply with the following SQL:

```
SELECT *
FROM Member m
WHERE m.Handicap < 16;
```

What should we do if we want to find all the members with a handicap less than Barbara Olson's? The preceding query will do that for us, but only if Barbara's handicap of 16 doesn't change. For the query to work for whatever Barbara's current handicap is, we can replace the single value 16 with the result of an inner query:

```
SELECT *
FROM Member m
WHERE Handicap <
     (SELECT Handicap
     FROM Member
     WHERE LastName = 'Olson' AND FirstName = 'Barbara');
```

We need to compare `Handicap` with a single value. If in a situation like this our inner query returns more than one value (for example, if there were more than one Barbara Olson in the club), then we would get an error when trying to run the query.

An inner query returning a single value is also useful if we want to compare values with an aggregate of some sort. For example, we might want to find all the members who have a handicap less than the average. In this case, we can use the inner query to return the average value:

```
SELECT *
FROM Member m
WHERE m.Handicap <
     (SELECT AVG(Handicap)
     FROM Member);
```

If you take it nice and slow, you can gradually build up quite complicated queries. Say we want to see whether any junior members have a lower handicap than the average for seniors. The inner query has to return the average value handicap for a senior member, and then we want to select all juniors with a handicap less than that. In the SQL statement that follows, both the inner and outer queries have an extra `SELECT` condition (the inner retrieves just seniors, and the outer retrieves just juniors):

```
SELECT *
FROM Member m
WHERE m.MemberType = 'Junior' AND Handicap < (
     SELECT AVG(Handicap)
     FROM Member
     WHERE MemberType = 'Senior');
```

Inner Queries Returning a Set of Values

This is where we started this chapter. When we use the `IN` keyword, SQL will expect to find a set of single values. For example, we might ask for rows from the `Entry` table for members with IDs `IN` a set of values. In the following statement, the inner query selects the IDs of all senior members, and the outer query returns the entries for those members:

```
SELECT *
FROM Entry e
WHERE e.MemberID IN
     (SELECT m.MemberID
     FROM Member m
     WHERE m.MemberType = 'Senior');
```

The inner section in this type of query must return just a single column. `IN` is expecting a list of single values (in this case, a list of `MemberID`). If the inner section returns more than one column (for example, `SELECT * FROM Member`), then we will get an error.

Many nested queries such as this can be written in other ways–often by using an inner join as we discussed earlier in the chapter. Some queries will feel more natural to you one way or the other.

Inner Queries Checking for Existence

Another type of inner query is the one we saw working with the EXISTS keyword. A statement using EXISTS just looks to see whether any rows at all are returned by the inner query. The actual values or numbers of rows returned are not important. The query that follows returns any rows from the Member table where we can find a corresponding row in the Entry table for that member:

```
SELECT m.Lastname, m.FirstName
FROM Member m
WHERE EXISTS
     (SELECT * FROM Entry e
     WHERE e.MemberID = m.MemberID);
```

Because the actual values retrieved by the inner query are not important, the inner query often has the form SELECT * FROM.

Another feature of this type of query is that the inner and outer sections are usually correlated. By this we mean that the WHERE clause in the inner section refers to values in the table in the outer section. In this case the inner query is checking if the current row in the Entry table has the same MemberID as the member currently under consideration in the outer query. I find the easiest way to visualize this is as illustrated in Figure 4-5.

It is difficult to think of a sensible EXISTS query that doesn't correlate values in the inner and outer sections. Consider what the following query will return:

```
SELECT m.Lastname, m.FirstName
FROM Member m
WHERE EXISTS
     (SELECT * FROM Entry e);
```

The query above doesn't really make any sense. It says to write out each member's names if there is a row in the Entry table (any row!). If the Entry table is empty, we will get nothing returned; otherwise, we will get all the names of all the members. I can't think why you'd ever want to do that. EXISTS queries are useful when we are looking for matching values somewhere else, and that is why the SELECT condition needs to compare values from both the inner and outer sections.

It is interesting to compare the following two queries. They both return the names of members who have entered a tournament, but the results are slightly different. The first uses an EXISTS clause:

```
SELECT m.Lastname
FROM Member m
WHERE EXISTS
     (SELECT * FROM Entry e
     WHERE e.MemberID = m.MemberID);
```

The second uses an INNER JOIN:

```
SELECT m.LastName
FROM Member m INNER JOIN Entry e ON  e.MemberID = m.MemberID;
```

The difference between the two queries is the number of rows that are returned.

The first query inspects each row in the Member table just once and returns the last name if there exists at least one entry for that member in the Entry table. The last name for any member will be written out only once.

The second query forms a join between the two tables that will consist of every combination of rows in Member and Entry with the same MemberID. The name for a particular member will be written out as many times as the number of tournaments he or she entered.

It's a subtle difference, but an important one – especially if you are wanting to count the returned rows. Adding DISTINCT in the SELECT clause of the second example will make the results of the two queries the same.

Using Subqueries for Updating

This book is mainly about queries for retrieving data, but many of the same ideas can be used for updating data and adding or deleting records. In Chapter 1 we looked at simple queries such as updating the phone number of a particular member, as shown here:

```
UPDATE Member m
SET m.Phone = '875076'
WHERE m.MemberID = 118;
```

We also looked at inserting and deleting rows from a table. To insert a row we list the columns we are providing values for and then the values, as in the following:

```
INSERT INTO Entry (MemberID, TourID, Year)
VALUES (153, 25, 2016);
```

Now, let's consider a situation where we want to add an entry for tournament 25 in 2016 for each of the juniors in the club. We want to add a set of rows to the `Entry` table, as shown in Figure 4-8, where the left column has the member IDs for each of the juniors and the next two columns are the specific tournament (25) and year (2016) for each entry.

We can write an SQL query to return a set of rows like those in Figure 4-8:

```
SELECT m.MemberID, 25, 2016
FROM Member m
WHERE m.MemberType = 'Junior';
```

This query is a little different from others we have looked at because it has constants in the SELECT clause. It will construct a row for each junior member with the member's ID and the two constants 25 (for the tournament) and 2016 (for the year).

We can now use the preceding query as a subquery in our INSERT query. Rather than provide just one value with the VALUES keyword, we can provide a set of values resulting from the subquery. In the following query, the inner SELECT will produce the set of rows seen in Figure 4-8, and the outer INSERT will put them in the Entry table:

```
INSERT INTO Entry (MemberID, TourID, Year)
     -- create an entry in tournament 25, 2016 for each Junior
     SELECT MemberID, 25, 2016
     FROM Member
     WHERE MemberType = 'Junior';
```

The same potential for using subqueries applies to other updating issues. Say, for the purposes of finding an example, that after entering data in the Entry table for the 2016 social tournament at Kaiapoi (tournament 25) you realize that only players with handicaps of 20 or more were allowed to enter. You could use a subquery to delete entries for members with handicaps less than 20:

```
DELETE FROM Entry
WHERE TourID = 25 AND Year = 2016 AND
MemberID IN
      (SELECT MemberID FROM Member WHERE Handicap < 20);
```

A subquery returning a single value

Find the tournaments that member Cooper has entered:

```
SELECT e.TourID, e.Year FROM Entry e WHERE e.MemberID =
     (SELECT m.MemberID FROM Member m
     WHERE m.LastName = 'Cooper');
```

An alternative way to write the preceding query is to use a join:

```
SELECT e.TourID, e.Year
FROM Entry e INNER JOIN Member m ON e.MemberID = m.MemberID
WHERE m.LastName = 'Cooper';
```

A subquery returning a set of single values

Find all the entries for an Open tournament:

```
SELECT *
FROM Entry e
WHERE e.TourID IN
     (SELECT t.TourID FROM Tournament t
     WHERE t.TourType = 'Open');
```

The preceding query can be replaced with:

The preceding query can be replaced with:

```
SELECT e.MemberID, e.TourID, e.Year
FROM Entry e INNER JOIN Tournament t ON e.TourID = t.TourID
WHERE t.TourType = 'Open';
```

A subquery checking for existence

Find the names of members that have entered any tournament:

```
SELECT m.LastName, m.FirstName
FROM Member m
WHERE EXISTS
     (SELECT * FROM Entry e
     WHERE e.MemberID = m.MemberID);
```

This can be replaced with:

```
SELECT DISTINCT m.LastName, m.FirstName
FROM Member m INNER JOIN Entry e
              ON e.MemberID = m.MemberID;
```

Constructing queries with negatives

Find the names of members who have not entered a tournament:

```
SELECT * FROM Member m
WHERE NOT EXISTS
     (SELECT * FROM Entry e
     WHERE e.MemberID = m.MemberID);
```

Comparing values with the results of aggregates

Find the names of members with handicaps less than the average:

```
SELECT m.LastName, m.FirstName FROM Member m WHERE m.Handicap <
     (SELECT AVG(Handicap) FROM Member);
```

Update data

Add a row in the Entry table for every junior for tournament 25 in 2016:

```
INSERT INTO Entry (MemberID, TourID, Year)
     SELECT MemberID, 25, 2016
     FROM Member WHERE MemberType = 'Junior';
```