Self Relationships

Let's add some more information to our `Member` table. Suppose some members have coaches assigned to them. How do we represent that in the class diagrams we talked about in Chapter 1? We could take the approach shown in Figure 5-1, with two classes: `Member` and `Coach`. Recall what the lines and numbers mean. From left to right, a coach might have several members to train (the 0..n nearest the `Member` class). From right to left, a particular member might have a single coach or no coach (the 0..1 nearest the `Coach` class).

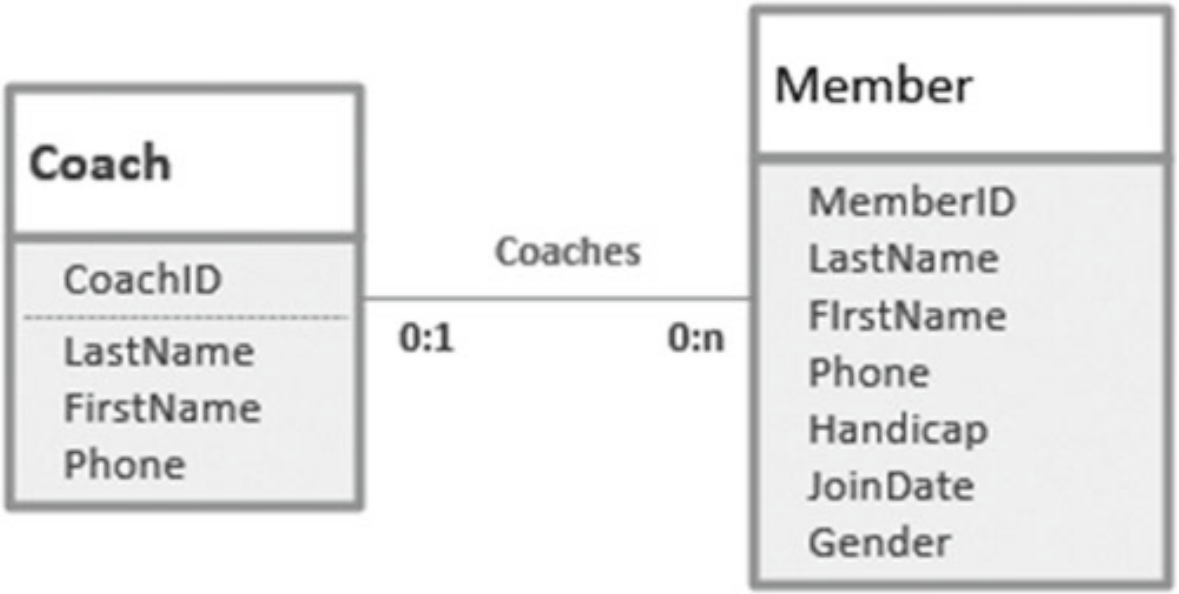


Figure 5-1. *Data model for coaches and members (not recommended!)*

In this example we don't actually have two separate classes of members and coaches. We have just one class of members, some of whom coach other members. This self relationship is shown in Figure 5-2.

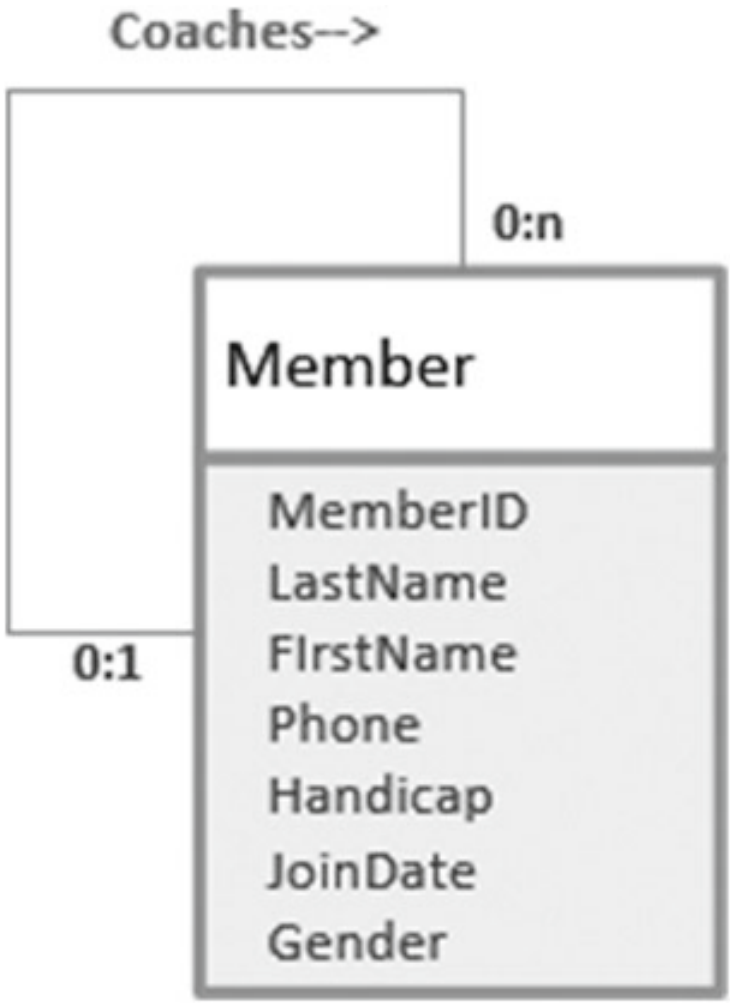


Figure 5-2. *Data model for members coaching other members*

The relationship line in Figure 5-2 can be read in a clockwise direction to say that a particular member might coach several other members or none (0..n). In the other direction, we can read that a particular member might have one coach or none (0..1).

MemberID	LastName	FirstName	Handicap	MemberType	Gender	Coach
118	McKenzie	Melissa	30	Junior	F	153
138	Stone	Michael	30	Senior	M	
153	Nolan	Brenda	11	Senior	F	
176	Branch	Helen		Social	F	
178	Beck	Sarah		Social	F	
228	Burton	Sandra	26	Junior	F	153
235	Cooper	William	14	Senior	M	153
239	Spence	Thomas	10	Senior	M	
258	Olson	Barbara	16	Senior	F	
286	Pollard	Robert	19	Junior	M	235
290	Sexton	Thomas	26	Senior	M	235
323	Wilcox	Daniel	3	Senior	M	
331	Schmidt	Thomas	25	Senior	M	153
332	Bridges	Deborah	12	Senior	F	235
339	Young	Betty	21	Senior	F	
414	Gilmore	Jane	5	Junior	F	153
415	Taylor	William	7	Senior	M	235
461	Reed	Robert	3	Senior	M	235
469	Willis	Carolyn	29	Junior	F	
487	Kent	Susan		Social	F	

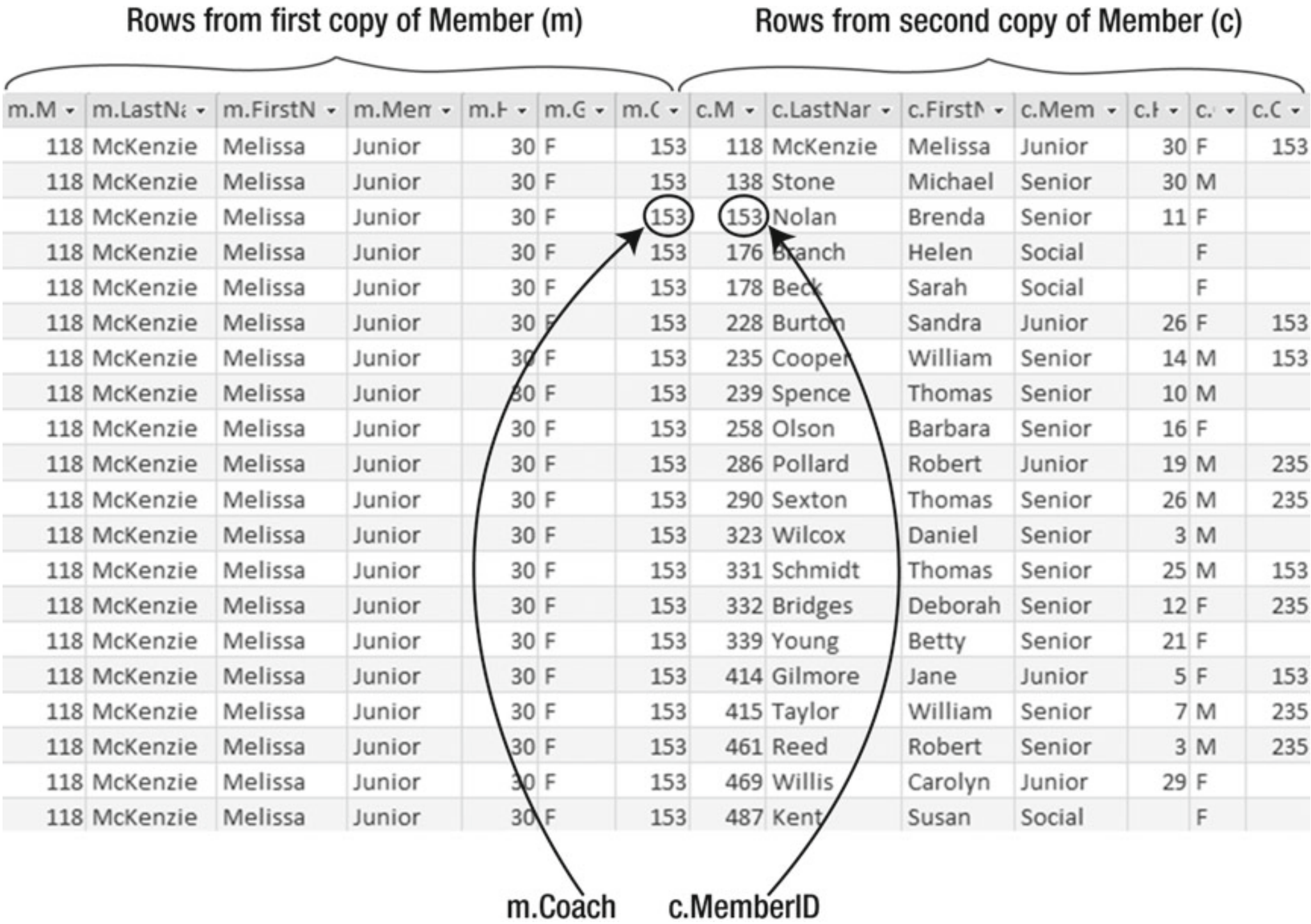
Figure 5-3. *Column Coach added to the Member table*

```
ALTER TABLE Member
ADD Coach INT FOREIGN KEY REFERENCES Member;
```

With the modified `Member` table, we now can answer several different types of questions. For example

- What are the names of the coaches?
- What is the name of Jane Gilmore's coach?
- Is anyone being coached by someone with a higher handicap?
- Are any women being coached by men?

None of these questions can be answered by inspecting a single row in the table. What we require is a *self join* on the `Member` table. The easiest way to understand a self join is to see how we make one.



m.M	m.LastNa	m.FirstN	m.Mem	m.H	m.G	m.C	c.M	c.LastNar	c.FirstN	c.Mem	c.H	c.	c.C
118	McKenzie	Melissa	Junior	30	F	153	118	McKenzie	Melissa	Junior	30	F	153
118	McKenzie	Melissa	Junior	30	F	153	138	Stone	Michael	Senior	30	M	
118	McKenzie	Melissa	Junior	30	F	153	153	Nolan	Brenda	Senior	11	F	
118	McKenzie	Melissa	Junior	30	F	153	176	Branch	Helen	Social		F	
118	McKenzie	Melissa	Junior	30	F	153	178	Beck	Sarah	Social		F	
118	McKenzie	Melissa	Junior	30	F	153	228	Burton	Sandra	Junior	26	F	153
118	McKenzie	Melissa	Junior	30	F	153	235	Cooper	William	Senior	14	M	153
118	McKenzie	Melissa	Junior	30	F	153	239	Spence	Thomas	Senior	10	M	
118	McKenzie	Melissa	Junior	30	F	153	258	Olson	Barbara	Senior	16	F	
118	McKenzie	Melissa	Junior	30	F	153	286	Pollard	Robert	Junior	19	M	235
118	McKenzie	Melissa	Junior	30	F	153	290	Sexton	Thomas	Senior	26	M	235
118	McKenzie	Melissa	Junior	30	F	153	323	Wilcox	Daniel	Senior	3	M	
118	McKenzie	Melissa	Junior	30	F	153	331	Schmidt	Thomas	Senior	25	M	153
118	McKenzie	Melissa	Junior	30	F	153	332	Bridges	Deborah	Senior	12	F	235
118	McKenzie	Melissa	Junior	30	F	153	339	Young	Betty	Senior	21	F	
118	McKenzie	Melissa	Junior	30	F	153	414	Gilmore	Jane	Junior	5	F	153
118	McKenzie	Melissa	Junior	30	F	153	415	Taylor	William	Senior	7	M	235
118	McKenzie	Melissa	Junior	30	F	153	461	Reed	Robert	Senior	3	M	235
118	McKenzie	Melissa	Junior	30	F	153	469	Willis	Carolyn	Junior	29	F	
118	McKenzie	Melissa	Junior	30	F	153	487	Kent	Susan	Social		F	

Figure 5-4. *Cartesian product between two copies of the Member table*

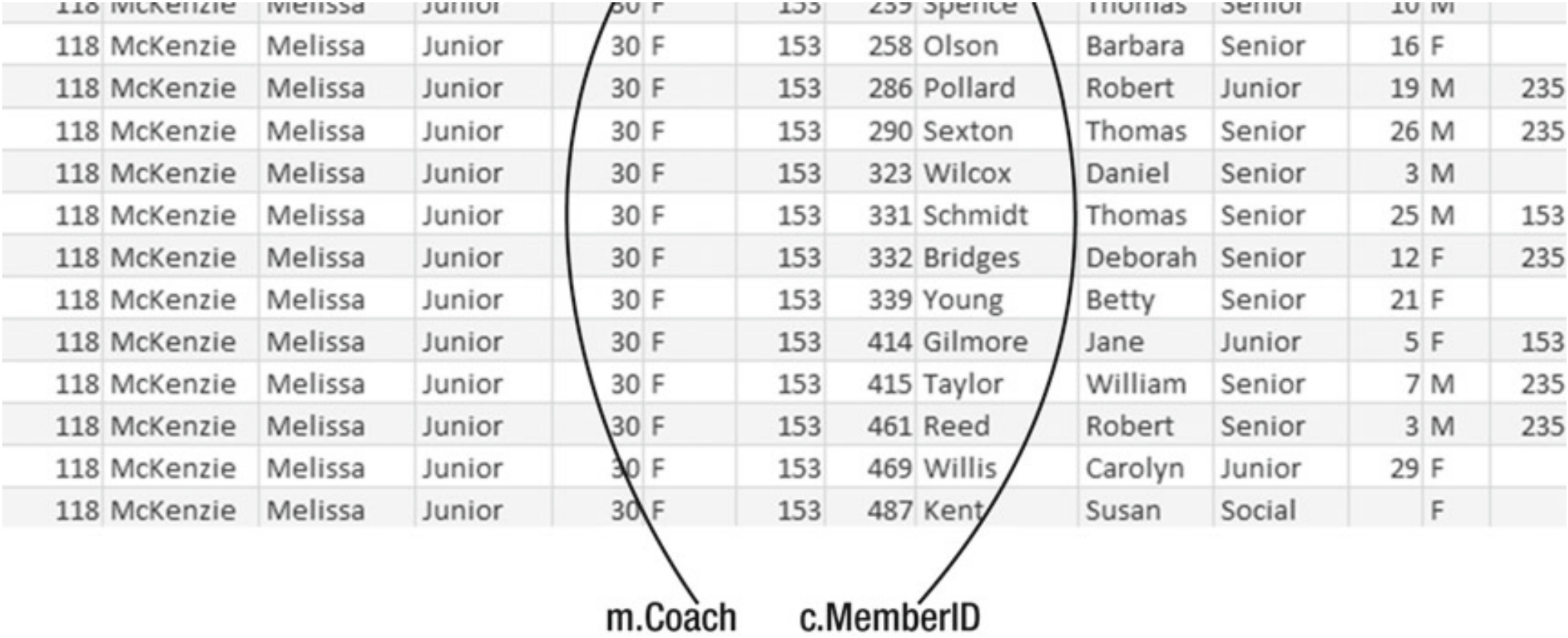


Figure 5-4. *Cartesian product between two copies of the Member table*

For queries about coaching, the interesting rows from the Cartesian product are those where the value of `Coach` from `m` is the same as `MemberID` from `c`. In Figure 5-4, you can see that the third line contains information about Melissa (from the `m` copy of `Member`) and information about her coach (from the `c` copy of `Member`). Now the choice of aliases becomes clear: `m` for columns about a member; `c` for the columns about that member's coach. Choosing helpful aliases can make understanding self joins much easier. The rows we would like to select from the Cartesian product are those satisfying `m.Coach = c.MemberID`. This is the join condition required to find information about members and their coaches. The SQL for the self join is:

```
SELECT *
FROM Member m INNER JOIN Member c ON m.Coach = c.MemberID;
```

The table resulting from the self join is shown in Figure 5-5.

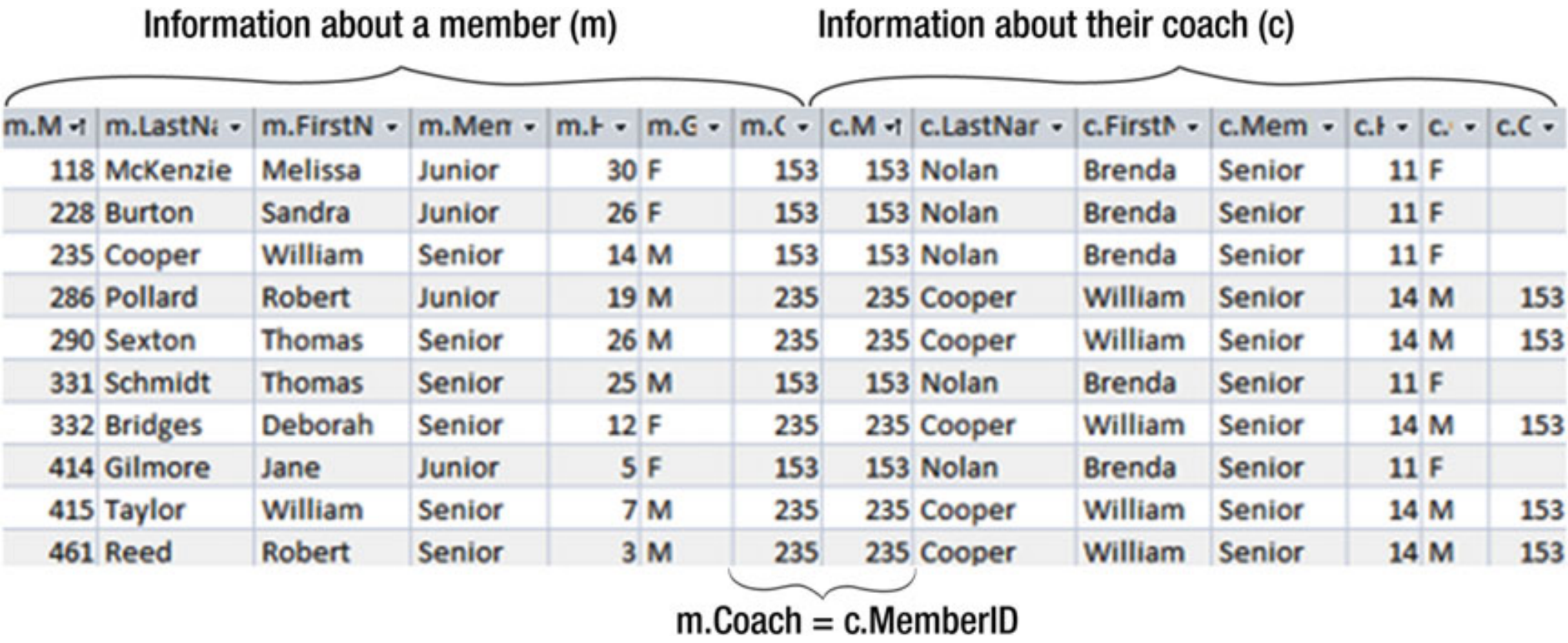


Figure 5-5. *Self join on Member table to retrieve information about members and their coaches*

Now that we have the results of the self join, we can answer the questions posed in the previous section about coaching. The trickiest part of all this was recognizing that maintaining information about members and coaches is a self relationship and designing the `Member` table appropriately in the first place.

Queries Involving a Self Join

With the joined table in Figure 5-5 as our base, we can answer all sorts of questions by simply selecting subsets of rows and projecting the appropriate columns. Whenever I need to do queries involving self joins, I usually perform the join first, retaining all the rows and columns as in Figure 5-5. With the joined table (or a quick sketch of the columns) in front of me, the way forward is usually relatively simple. Let's see how this works with a few questions.

What Are the Names of the Coaches?

Looking at Figure 5-5, we can see that the names of the coaches are in the columns coming from the c part of the join. We just want a list of the names in the columns `c.LastName` and `c.FirstName` so those columns can be included in the `SELECT` clause. We don't want the names repeated, so we use the keyword `DISTINCT`. The following SQL statement will return the names of the two coaches, Brenda Nolan and William Cooper.

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

Who Is Being Coached by Someone with a Higher Handicap?

To find out who is being coached by someone with a higher handicap, we need to compare the handicap of the member (`m.Handicap`) with the handicap of that member's coach (`c.Handicap`). What is required is a `WHERE` clause after the join clause to find where the member's handicap is less than the coach's handicap:

```
SELECT *
FROM Member m INNER JOIN Member c ON m.Coach = c.MemberID
WHERE m.Handicap < c.Handicap;
```

For the data in Figure 5-5, this will retrieve the data in the last four rows. (You don't have to be a great golfer to be a good coach!) Having done the join and selected the appropriate rows, we can then choose which columns we want to appear in the final result and list them in the `SELECT` clause.

List the Names of All Members and the Names of Their Coaches

Listing the names of members and their coaches sounds pretty trivial, but if we are not careful, we can get it wrong. A first thought might be to project just the four columns containing the names of member and coach from the joined table in Figure 5-5. However, there are only 10 rows in the joined table, whereas there are 20 members in the `Member` table. The issue here is that not all the members have coaches. We looked at situations like this in the section on outer joins in Chapter 3.

To recap, let's go back to the Cartesian product of two copies of the `Member` table, but look at some rows involving a member with no coach, as shown in Figure 5-6.

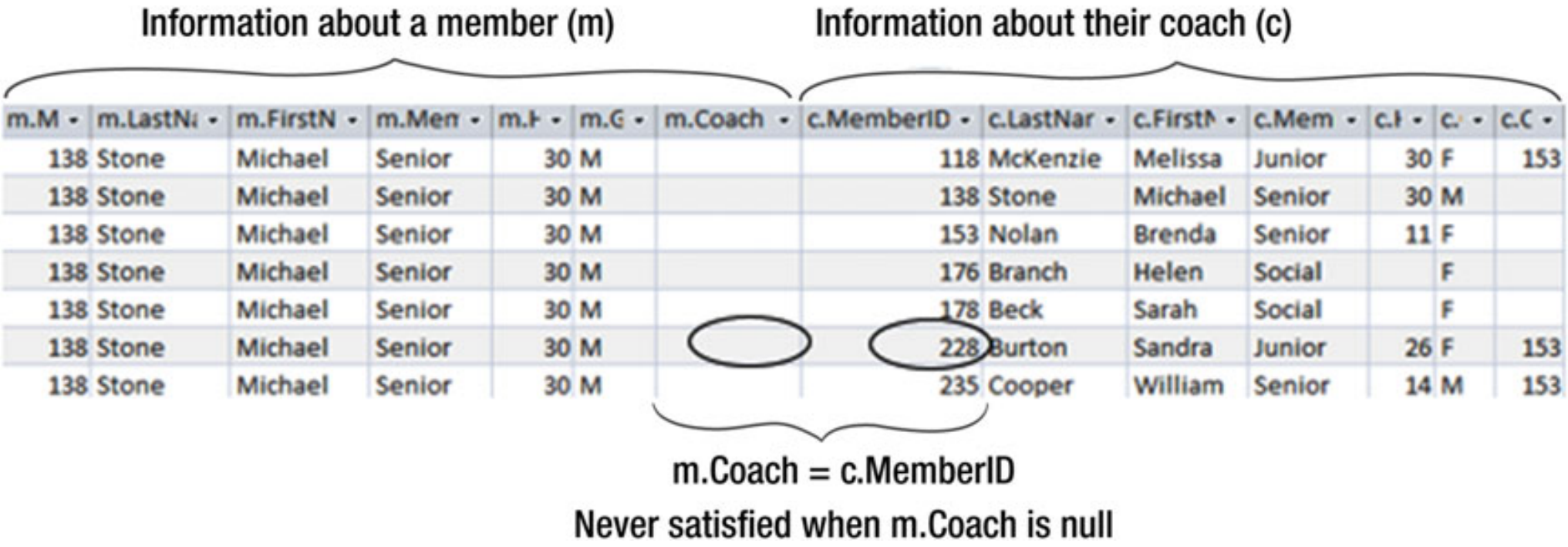


Figure 5-6. *Part of the Cartesian product between two copies of the Member table*

The join condition (`m.Coach = c.MemberID`) is never satisfied for a member with a null in the `Coach` column, so all those members will be missing from our joined table. We just need to be careful to understand what we really want. Do we want a list of all the members with coaches (10 rows), or a list of all the members along with their coach's name if they have one (20 rows)? If it's the latter, we need an outer join. We need to see the name of each member (from the `m` copy of the `Member` table), along with the name of his coach, if any (from the `c` copy). The SQL for this outer join is:

```
SELECT m.LastName AS MemberLast, m.FirstName AS MemberFirst,
       c.LastName AS CoachLast, c.FirstName AS CoachFirst
FROM Member m LEFT OUTER JOIN Member c ON m.Coach = c.MemberID;
```

In the preceding query we have given each output attribute a *column alias*. A column alias temporarily renames a column in order to improve the readability of the output. In this case it helps the reader distinguish which name belongs to whom, as shown in Figure 5-7. Without the aliases, the attributes would be labelled as `m.LastName` and `c.LastName` and so on, which are not quite so easy to understand. Recall from Chapter 3 that for a left outer join, where there is no matching row from the right-hand table, those columns will be filled with nulls. Figure 5-7 shows the output of the left outer join.

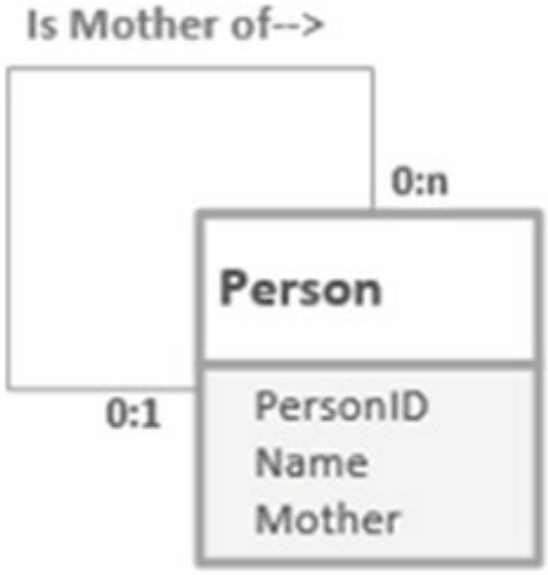
MemberLast	MemberFirst	CoachLast	CoachFirst
McKenzie	Melissa	Nolan	Brenda
Stone	Michael		
Nolan	Brenda		
Branch	Helen		

MemberLast	MemberFirst	CoachLast	CoachFirst
McKenzie	Melissa	Nolan	Brenda
Stone	Michael		
Nolan	Brenda		
Branch	Helen		
Beck	Sarah		
Burton	Sandra	Nolan	Brenda
Cooper	William	Nolan	Brenda
Spence	Thomas		
Olson	Barbara		
Pollard	Robert	Cooper	William
Sexton	Thomas	Cooper	William
Wilcox	Daniel		
Schmidt	Thomas	Nolan	Brenda
Bridges	Deborah	Cooper	William
Young	Betty		
Gilmore	Jane	Nolan	Brenda
Taylor	William	Cooper	William
Reed	Robert	Cooper	William
Willis	Carolyn		
Kent	Susan		

Figure 5-7. *Left outer join to list all members and coaches*

Who Coaches the Coaches, or Who Is My Grandmother?

The self join between two copies of the `Member` table shows us one level of members and coaches. If we look at the rows in Figure 5-7, we can see that Thomas Sexton is coached by William Cooper, who is in turn coached by Brenda Nolan, who doesn't have a coach. The hierarchy isn't all that interesting for this problem, but there are several analogous situations where the hierarchy is of considerable interest. Genealogy is one. Consider the data model and part of the `Person` table in Figure 5-8. For the sake of keeping things really simple, we will consider only a tiny bit of information about just women and birth mothers.



PersonID	Name	Mother
1001	Agnes	1002
1002	Mary	1006
1003	Linda	1002
1004	Grace	1002
1005	Sue	1001
1006	Brenda	
1007	Bo	1003
1008	Lily	1003

Person Table

Figure 5-8. *Data model for women and their birth mothers*

Figure 5-8. *Data model for women and their birth mothers*

The relationship in Figure 5-8 can be read clockwise as "a person can be the mother of several other people" and in the other direction as "a person has at most one mother and might have none." Now. in real life, that last statement doesn't sound right–surely everyone has a mother. However, as with all databases, this database is only an approximation of the complexities of real life, and it can only keep data that is available. Unless we trace everyone back to the primeval slime, there will be some people in our table whose mother we do not know. Brenda is one. The table and model in Figure 5-8 have exactly the same structure as our member and coach example, but a question like "Who is Sue's grandmother?" seems a bit more likely than "Who coaches my coach?"

So, how do we get information about people along with information about their mothers? Just as in the previous section, we need to join the `Person` table to itself. (Don't forget to make the join an outer join so you don't lose Brenda.) The SQL is:

```
SELECT *
FROM Person p LEFT OUTER JOIN Person m on p.Mother = m.ID;
```

The Access diagrammatic interface for the join is shown in Figure 5-9, along with the resulting table. I've given the first copy of the table the alias `p` for *person* and the second copy the alias `m` for *mother*.

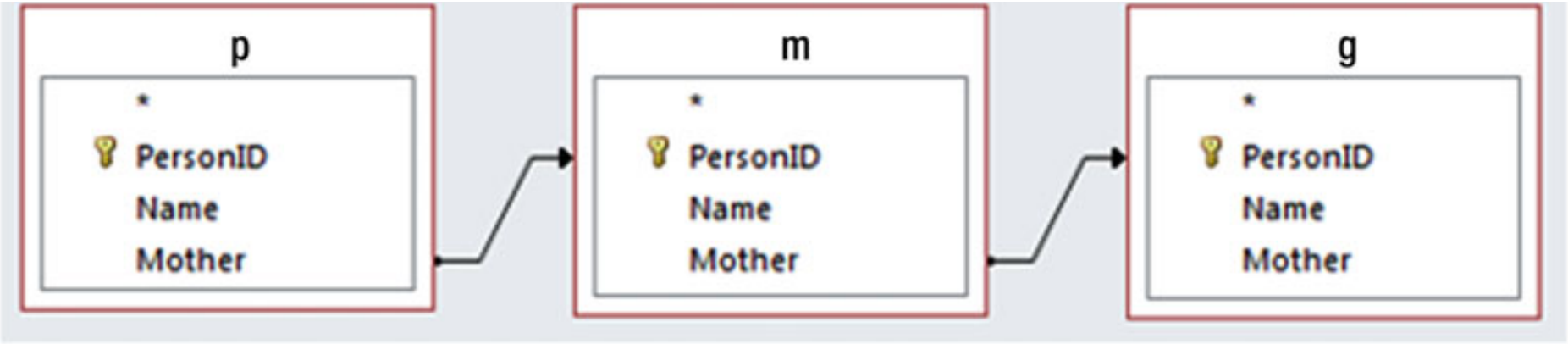
p.PersonID	p.Name	p.Mother	m.PersonID	m.Name	m.Mother
1001	Agnes	1002	1002	Mary	1006
1002	Mary	1006	1006	Brenda	
1003	Linda	1002	1002	Mary	1006
1004	Grace	1002	1002	Mary	1006
1005	Sue	1001	1001	Agnes	1002
1006	Brenda				
1007	Bo	1003	1003	Linda	1002
1008	Lily	1003	1003	Linda	1002

Figure 5-9. *Finding people and their mothers: Access diagram for the left outer join and the resulting table*

Now, what about going back to the previous generation? For that we need to perform another left outer join between the result table in Figure 5-9 and another copy of the `People` (with the alias `g` for *grandmother*). The SQL for the two left outer joins is:

```
SELECT *
FROM (Person p LEFT JOIN Person m ON p.Mother = m.ID)
     LEFT JOIN Person g ON m.Mother = g.ID;
```

The resulting table is shown in Figure 5-10.



p.PersonID	p.Name	p.Mother	m.PersonID	m.Name	m.Mother	g.PersonID	g.Name	g.Mother
1001	Agnes	1002	1002	Mary	1006	1006	Brenda	
1002	Mary	1006	1006	Brenda				
1003	Linda	1002	1002	Mary	1006	1006	Brenda	
1004	Grace	1002	1002	Mary	1006	1006	Brenda	
1005	Sue	1001	1001	Agnes	1002	1002	Mary	1006
1006	Brenda							
1007	Bo	1003	1003	Linda	1002	1002	Mary	1006
1008	Lily	1003	1003	Linda	1002	1002	Mary	1006

Figure 5-10. *Finding three generations: Access diagram for the left outer joins and the resulting table*

Clearly, we can keep making more and more self joins until we run out of generations. These sorts of hierarchical queries are likely to turn up whenever we have self relationships. One small catch is that we must specify the number of joins in each query. Standard SQL doesn't have the notion of a query that automatically keeps doing the self joins until it runs out of generations, such as "Find all my female ancestors"; however, some implementations do support this.[1]

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

Figure 5-11. *Finding Melissa's coach*

To find Melissa's coach, we first find the row for Melissa (`m` in Figure 5-11) and then note that her coach is member 153. Then we find another row (`c` for coach) that has the `MemberID` value of 153; we can see that Melissa's coach is Brenda. You don't need to know anything about self relationships or foreign keys or joins to figure that out. But once you have that logic clearly in your mind, you can write it down in natural language, and then the translation to SQL is pretty straightforward.

Let's write a description of Figure 5-11:

> *I need to look at two rows (`m` and `c`) in the `Member` table, and I want to write out `c.FirstName` where `c.MemberID` has the same value as `m.Coach` and `m.FirstName` is 'Melissa'*

Let's write a description of Figure 5-11:

> *I need to look at two rows (`m` and `c`) in the `Member` table, and I want to write out `c.FirstName` where `c.MemberID` has the same value as `m.Coach` and `m.FirstName` is 'Melissa'*

And here is the corresponding SQL:

```
SELECT c.FirstName
FROM Member m, Member c
WHERE c.MemberID = m.Coach AND m.FirstName = 'Melissa';
```

So, how does this output approach correspond to the process approach we considered earlier? As you might expect, the preceding SQL is just an alternative way of stating the same query as the one where we used the self join. In the preceding SQL statement, the middle line is the Cartesian product between two copies of the `Member` table, and the first part of the `WHERE` clause is the join condition. The statement `FROM Member m, Member c WHERE c.MemberID = m.Coach` is just another way of expressing the self join we used in the previous sections.

Let's try one of the other queries using an outcome approach: Who is being coached by someone with a higher handicap? The picture I would need in my head to answer this question is shown in Figure 5-12.

MemberID	LastName	FirstName	Handicap	Coach
118	McKenzie	Melissa	30	153
138	Stone	Michael	30	
153	Nolan	Brenda	11	
176	Branch	Helen		
178	Beck	Sarah		
228	Burton	Sandra	26	153
c 235	Cooper	William	14	153
239	Spence	Thomas	10	
258	Olson	Barbara	16	
286	Pollard	Robert	19	235
290	Sexton	Thomas	26	235
323	Wilcox	Daniel	3	
331	Schmidt	Thomas	25	153
m 332	Bridges	Deborah	12	235
339	Young	Betty	21	
414	Gilmore	Jane	5	153
415	Taylor	William	7	235
461	Reed	Robert	3	235
469	Willis	Carolyn	29	
487	Kent	Susan		

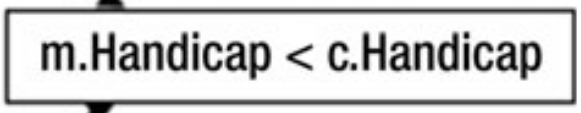


Figure 5-12. *Finding members who are coached by someone with a higher handicap*

Figure 5-12. *Finding members who are coached by someone with a higher handicap*

We can see that Deborah, whose handicap is 12, is being coached by member 235. Member 235, William, has a handicap of 14, so Deborah satisfies our criteria. Here is the more general statement representing the logic depicted in Figure 5-12:

> *I'm going to look at every row (`m`) in the `Member` table and will write out `m.FirstName` and `m.LastName` if there exists some other row (`c`) in the `Member` table where `c.MemberID` is the same as `m.Coach` and `m.Handicap` is less than `c.Handicap`*

The SQL follows in a straightforward manner:

```
SELECT m.FirstName, m.LastName
FROM Member m, Member c
WHERE c.MemberID = m.Coach AND m.Handicap < c.Handicap;
```

Once again, you can see the equivalent of the self join in the preceding query (`FROM Member m, Member c WHERE c.MemberID = m.Coach`). The usefulness of this outcome approach is that you don't need to understand what a self join is, nor must you make the mental leap that you need one. By thinking in terms of virtual fingers and which rows are involved in helping you with your decision, you can sketch a statement of the criteria. The SQL usually follows quite easily from that.

Questions Involving "Both"

In the "Avoiding Common Mistakes" section of Chapter 2, we looked at a questions such as, "Which members have entered *both* tournaments 24 and 36?" To recap, I've reproduced the `Entry` table in Figure 5-13.

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

A common first attempt at an SQL statement to find entries in both tournaments is the following:

```
-- Will not produce the desired result
SELECT e.MemberID
FROM Entry e
WHERE e.TourID = 24 AND e.TourID = 36;
```

Remember that a `WHERE` condition is applied to each row of the table individually. The condition (`e.TourID = 24 AND e.TourID = 36`) is never true for any individual row, as each row has only a single value for `TourID`. The preceding query will never return any rows because the value in `TourID` cannot be two different things (24 and 36) simultaneously. Such a query can be quite dangerous, because the user may interpret the empty result as meaning that no members have entered both tournaments, whereas the query statement is actually incorrect.

To answer the question, we need to look at more than one row in the `Entry` table. I find an outcome approach to be the most natural for dealing with questions involving "both."

An Outcome Approach to Questions Involving "Both"

The picture I need in my head to answer "Which members have entered both tournaments 24 and 36?" is shown in Figure 5-14.

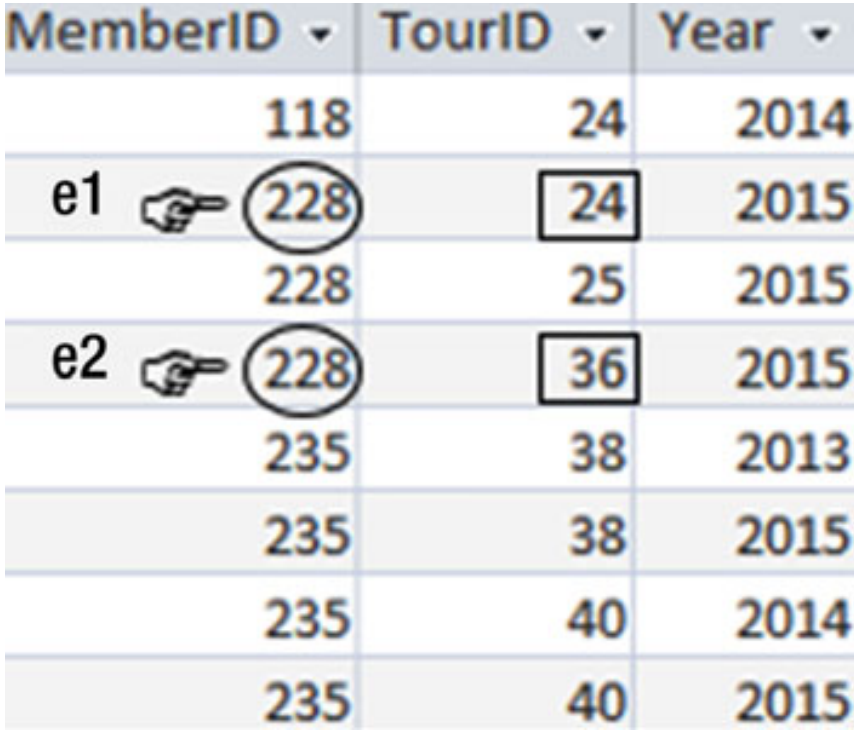


Figure 5-14. *Which members have entered both tournaments 24 and 36?*

Looking at Figure 5-14, it is pretty clear that member 228 has entered both the tournaments. We are to looking for two rows (two fingers, `e1` and `e2`) with matching `MemberID` values and where the rows have the required two TourID values.

A more general expression of the logic displayed in Figure 5-14 is:

> *I'm going to look at every row (`e1`) in the `Entry` table. I'll write out that row's member ID if `TourID` has the value 24 and I can also find another row (`e2`) in the `Entry` table with the same value for `MemberID` and that has 36 as the value for `TourID`.*

The SQL follows from here. If you have trouble with it, refer to Figure 5-14.

```
SELECT e1.MemberID
FROM Entry e1, Entry e2
WHERE e1.MemberID = e2.MemberID
     AND e1.TourID = 24 AND e2.TourID = 36;
```

A Process Approach to Questions Involving "Both"

As always, we have several ways to think about a query. Take a look at the middle two lines of the last query. `FROM Entry e1, Entry e2` is a Cartesian product (which will give us every combination of pairs of rows), followed by selecting a subset of rows satisfying (`WHERE e1.MemberID = e2.MemberID`). This is a join. In fact, it is a self join between two copies of the `Entry` table. Part of the join between two copies of the `Entry` table is shown in Figure 5-15.

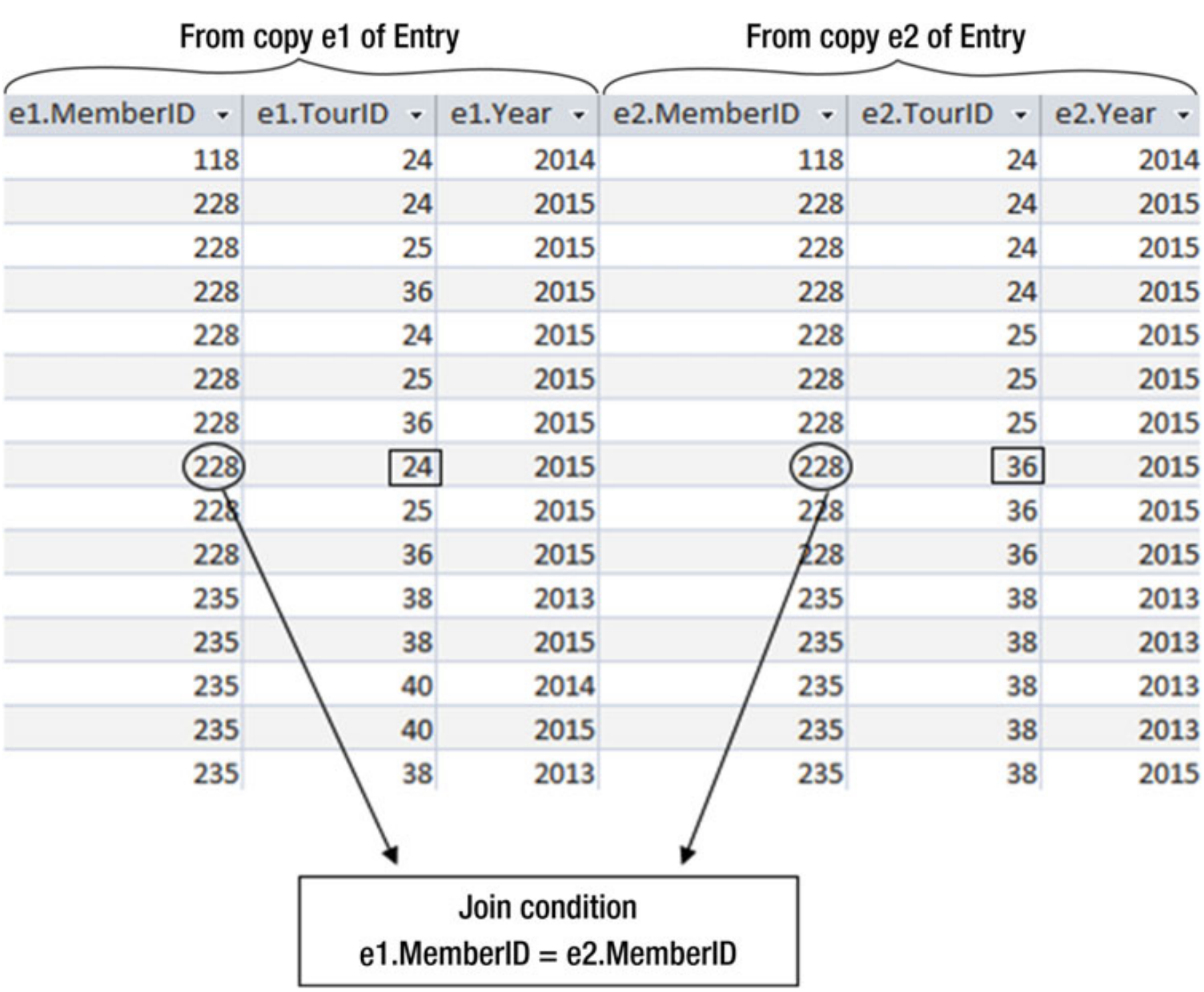


Figure 5-15. *Part of the self join between two copies of the Entry table*

The self join in Figure 5-15 shows those combinations of rows from the `Entry` table for the same member. For example, we can see every combination of rows involving member 228. We can use this self join to answer the question about members who have entered both tournaments 24 and 36. We just need to find a row that has 24 from the first copy and 36 from the second copy (or vice versa) – that is, `e1.TourID = 24 AND e2.TourID = 36`.

The SQL for this self join followed by the `WHERE` clause to select the rows with the appropriate values of `TourID` is shown here:

```
SELECT e1.MemberID
FROM Entry e1 INNER JOIN Entry e2 ON e1.MemberID = e2.MemberID
WHERE e1.TourID = 24 AND e2.TourID = 36;
```

If you compare the two queries for finding the entries in both tournaments 24 and 26, you will see how similar they are. They will both produce exactly the same result. You will probably find one or the other to be more intuitive.

Self Relationships

We have a self relationship when different instances of a class are related to each other. In the example in this chapter, we had that some members are coaches of other members.

From a process perspective, queries about coaches or coaching relationships require self joins, which take two copies of the table and join them. In the following example, the copy of the `Member` table with the information about the member has the alias `m`, and the copy with information about the coach has the alias `c`:

```
SELECT m.LastName, m.FirstName, c.LastName, c.FirstName
FROM Member m INNER JOIN Member c ON m.Coach = c.MemberID
```

Alternatively, from an output approach we might come up with this equivalent query:

```
SELECT m.FirstName, m.LastName, c.LastName, c.FirstName
FROM Member m, Member c
WHERE c.MemberID = m.Coach
```

Both these queries can form the basis of queries to answer a number of questions about coaching.

Questions Involving the Word "Both"

From an outcome approach we needed to find two rows in the Entry table (e1 and e2) for the same member. One of the rows needed to be for tournament 24 and the other for tournament 36. The following shows the outcome-based SQL query:

```
SELECT e1.MemberID
FROM Entry e1, Entry e2
WHERE e1.MemberID = e2.MemberID AND e1.TourID = 24 AND e2.TourID = 36;
```

Alternatively, from a process approach we might recognize the need for a self join between two copies of the Entry table, which is done using the join condition e1.MemberID = e2.MemberID. This would need to be followed by a WHERE clause to return the rows with the appropriate TourID values.

The self join query equivalent to the preceding query is:

```
SELECT e1.MemberID
FROM Entry e1 INNER JOIN Entry e2 ON e1.MemberID = e2.MemberID
WHERE e1.TourID = 24 AND e2.TourID = 36;
```