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
286	Pollard	Robert	Junior

a) (Abbreviated) Member table

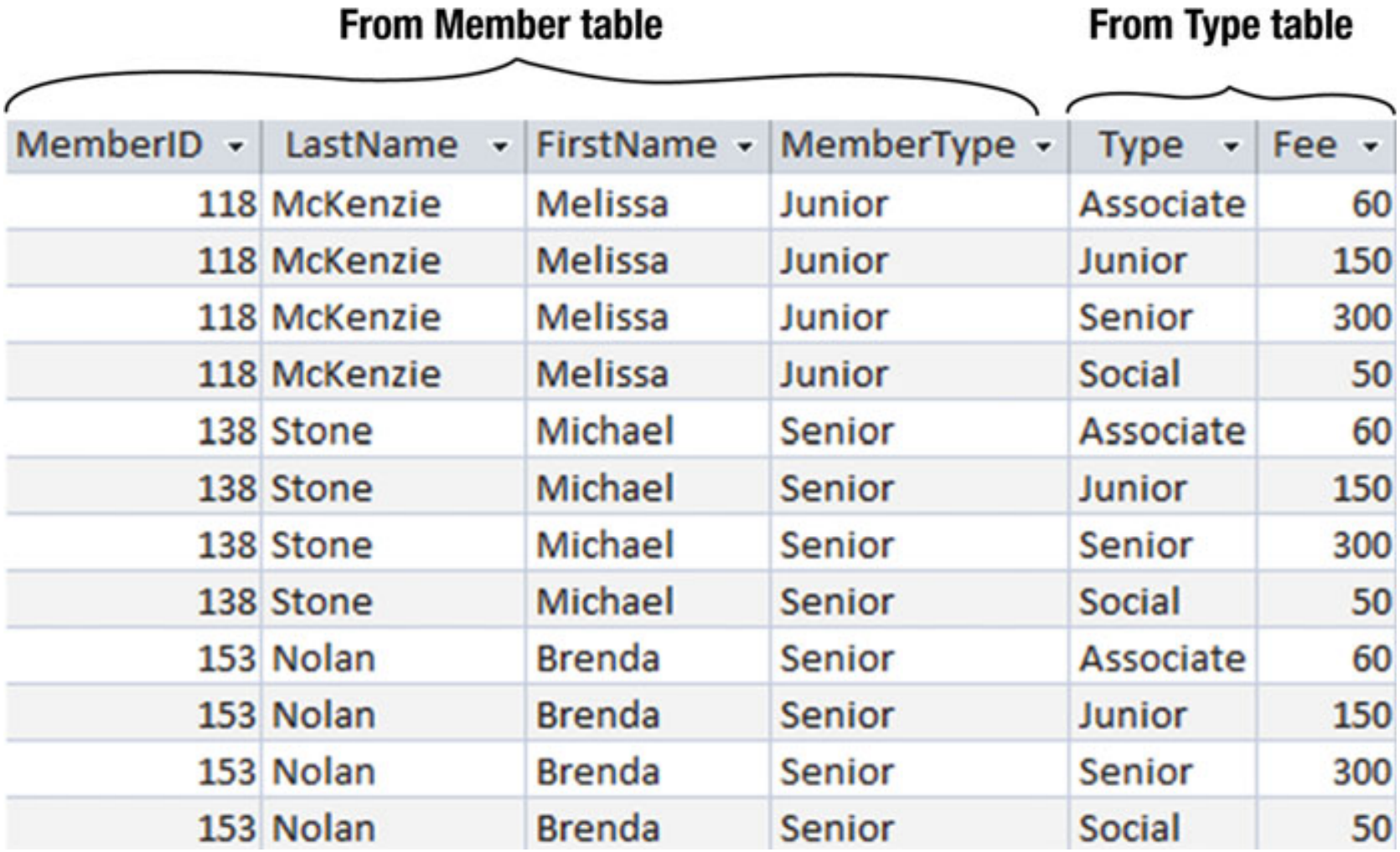
Type	Fee
Associate	60
Junior	150
Senior	300
Social	50

b) Type table

Figure 3-1. *Two permanent tables in the database*

The virtual table resulting from the Cartesian product will have a column for each column in the two contributing tables. The rows in the resulting table consist of every combination of rows from the original tables. Figure 3-2 shows the first few rows of the Cartesian product.

The virtual table resulting from the Cartesian product will have a column for each column in the two contributing tables. The rows in the resulting table consist of every combination of rows from the original tables. Figure 3-2 shows the first few rows of the Cartesian product.



MemberID	LastName	FirstName	MemberType	Type	Fee
118	McKenzie	Melissa	Junior	Associate	60
118	McKenzie	Melissa	Junior	Junior	150
118	McKenzie	Melissa	Junior	Senior	300
118	McKenzie	Melissa	Junior	Social	50
138	Stone	Michael	Senior	Associate	60
138	Stone	Michael	Senior	Junior	150
138	Stone	Michael	Senior	Senior	300
138	Stone	Michael	Senior	Social	50
153	Nolan	Brenda	Senior	Associate	60
153	Nolan	Brenda	Senior	Junior	150
153	Nolan	Brenda	Senior	Senior	300
153	Nolan	Brenda	Senior	Social	50

Figure 3-2. *First few rows of the Cartesian product between Member and Type tables*

We have the four columns from the Member table and the two columns from the Type table, which gives us six columns total. Each row from the Member table appears in the resulting table alongside each row from the Type table. We have Melissa McKenzie appearing on four rows – once with each of the four rows in the Type table (Associate, Junior, Senior, Social). The total number of rows will be the number of rows in each table multiplied together; in other words, for this cut-down Member table, we have 10 rows times 4 rows (from Type), giving a total of 40 rows. Cartesian products can produce very, very large result tables, which is why they don't give us much useful information on their own.

A Cartesian product operation is represented in SQL by CROSS JOIN. The SQL to retrieve the data shown in Figure 3-2 is:

```
SELECT *
FROM Member m CROSS JOIN Type t;
```

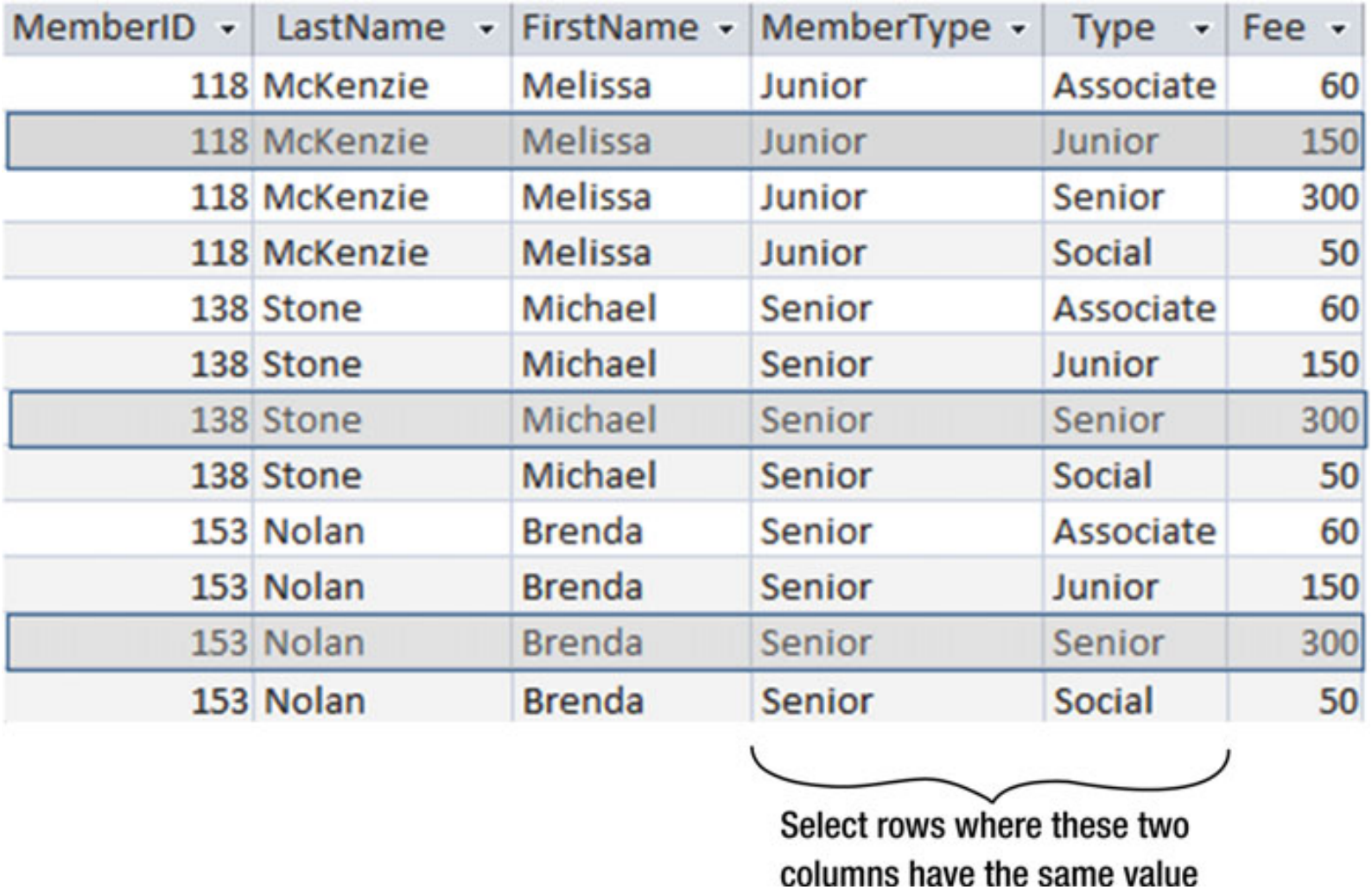


Figure 3-3. *Cartesian product followed by selecting a subset of rows*

The operation shown in Figure 3-3 (a Cartesian product followed by selecting a subset of rows) is known as an *inner join* (often just called a *join*). The condition we use to select the rows is known as the *join condition*. The SQL for the inner join in Figure 3-3 is:

```
SELECT *
FROM Member m INNER JOIN Type t ON m.MemberType = t.Type;
```

Let's start with the Cartesian product: we want a set of rows made up of combinations of rows from each of the contributing tables. Figure 3-4 shows how we can envisage this. We are looking at two tables, so we need two fingers to keep track of the rows. Finger `m` looks at each row of the `Member` table in turn. Currently it is pointing at row 3. For each row in the `Member` table, finger `t` will point to each row in the `Type` table. For the Cartesian product we retain every combination of the rows. In terms of Figure 3-4 the Cartesian product can be expressed in natural language as:

> *I'll write out all the attributes from row `m` and all the attributes from row `t` so long as `m` comes from the `Member` table and `t` comes from the `Type` table.*

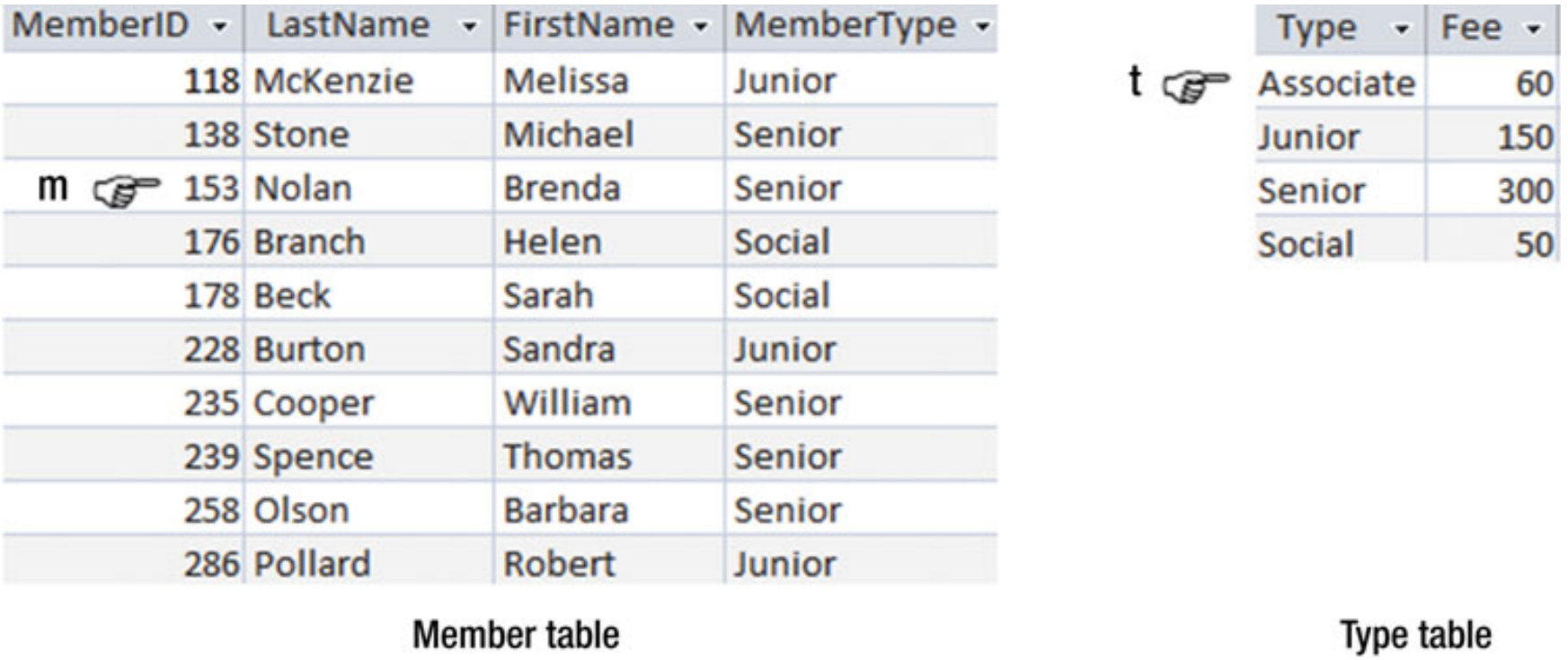


Figure 3-4. *Row variables m and t point to each row in the Member and Types tables, respectively*

The SQL for the query represented in Figure 3-4 and that results in the output shown in Figure 3-2 is:

```
SELECT *
FROM Member m, Type t;
```

The preceding statement will return the same rows as the expression we had previously used that used the `CROSS JOIN` phrase.

For a join we have the extra condition that we want to retrieve only those combinations of rows where the membership type from each table is the same. We can express this in natural language as:

> *I'll write out all the attributes from row* `m` *and all the attributes from row* `t` *so long as* `m` *comes from the* `Member` *table and* `t` *comes from the Type table and* `m.MemberType = t.Type`.

The pair of rows depicted in Figure 3-5 satisfies that condition and so will be retrieved. If `m` stays where it is and `t` moves down a row, then the condition will no longer be satisfied and the new combination will not be included.

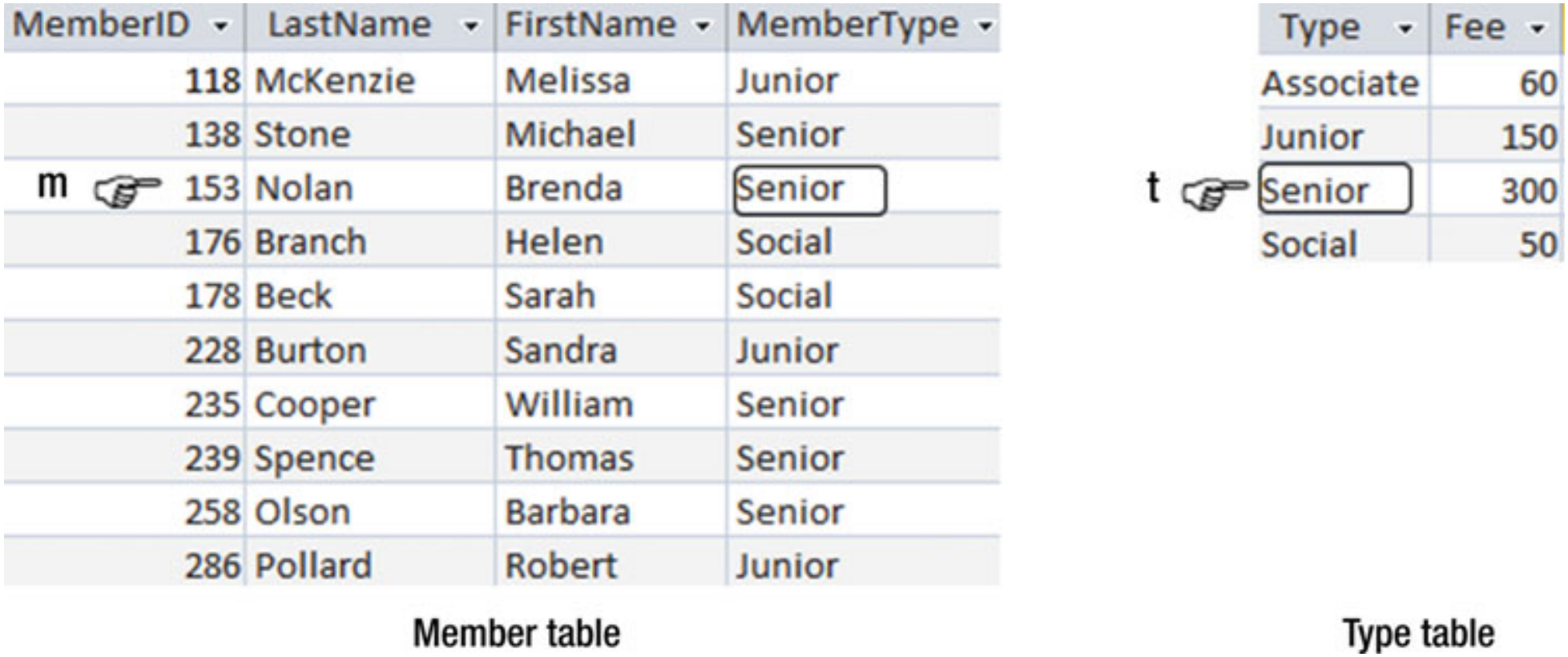


Figure 3-5. *Rows will be retrieved where m.MemberType = t.Type*

We can translate the query depicted in Figure 3-5 into SQL as follows:

```
SELECT *
FROM Member m, Type t
WHERE m.MemberType = t.Type;
```

```
SELECT *
FROM Member m, Type t
WHERE m.MemberType = t.Type;
```

If we look carefully at the preceding statement we can see that the first two lines represent the Cartesian product, and the `WHERE` clause in last line is selecting a subset of the rows where the membership types are the same. This was how we defined an inner join in the previous section. The preceding statement will produce the same rows as our previous statement for an inner join, seen again here:

```
SELECT *
FROM Member m INNER JOIN Type t ON m.MemberType = t.Type;
```

The first statement says what the rows to be retrieved are like (outcome approach) and the second expresses what operation we should use to retrieve those rows (process approach). Which one you use does not matter–it just depends on how you find yourself thinking about the query. Sometimes there is a possibility that the way you express the query may affect the performance, and we will talk about this more in Chapter 9. Actually, most database products are pretty smart at optimizing, or finding the quickest way to perform a query, regardless of how you express it. For example, in SQL Server the two expressions for the join shown are carried out in the same way. In fact, in SQL Server 2013, if you type the code in the first statement into the default interface for creating a view, it will be replaced by the code using the `INNER JOIN` phrase.

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

b) Entry

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

c) Tournament

Figure 3-6. *Permanent tables in the club database*

Member joined with Entry on
m.MemberID = e.MemberID

Join result to Tournament on
e.TourID = t.TourID

m.MemberID	LastName	FirstName	e.MemberID	e.TourID	Year	t.TourID	TourName	TourType
118	McKenzie	Melissa	118	24	2014	24	Leeston	Social
228	Burton	Sandra	228	24	2015	24	Leeston	Social
228	Burton	Sandra	228	25	2015	25	Kaiapoi	Social
228	Burton	Sandra	228	36	2015	36	WestCoast	Open
235	Cooper	William	235	38	2013	38	Canterbury	Open
235	Cooper	William	235	38	2015	38	Canterbury	Open
235	Cooper	William	235	40	2014	40	Otago	Open
235	Cooper	William	235	40	2015	40	Otago	Open
239	Spence	Thomas	239	25	2015	25	Kaiapoi	Social
239	Spence	Thomas	239	40	2013	40	Otago	Open
258	Olson	Barbara	258	24	2014	24	Leeston	Social
258	Olson	Barbara	258	38	2014	38	Canterbury	Open
286	Pollard	Robert	286	24	2013	24	Leeston	Social
286	Pollard	Robert	286	24	2014	24	Leeston	Social
286	Pollard	Robert	286	24	2015	24	Leeston	Social
415	Taylor	William	415	24	2015	24	Leeston	Social
415	Taylor	William	415	25	2013	25	Kaiapoi	Social
415	Taylor	William	415	36	2014	36	WestCoast	Open
415	Taylor	William	415	36	2015	36	WestCoast	Open
415	Taylor	William	415	38	2013	38	Canterbury	Open
415	Taylor	William	415	38	2015	38	Canterbury	Open
415	Taylor	William	415	40	2013	40	Otago	Open
415	Taylor	William	415	40	2014	40	Otago	Open
415	Taylor	William	415	40	2015	40	Otago	Open

From Member table (m)

From Entry table (e)

From Tournament table (t)

415	Taylor	William	415	24	2015	24	Leeston	Social
415	Taylor	William	415	25	2013	25	Kaiapoi	Social
415	Taylor	William	415	36	2014	36	WestCoast	Open
415	Taylor	William	415	36	2015	36	WestCoast	Open
415	Taylor	William	415	38	2013	38	Canterbury	Open
415	Taylor	William	415	38	2015	38	Canterbury	Open
415	Taylor	William	415	40	2013	40	Otago	Open
415	Taylor	William	415	40	2014	40	Otago	Open
415	Taylor	William	415	40	2015	40	Otago	Open

From Member table (m) From Entry table (e) From Tournament table (t)

Figure 3-7. *Joining the Member, Entry, and Tournament tables*

The join condition for the first join between the `Member` and `Entry` tables is that `m.MemberID = e.MemberID` as shown by the rectangular boxes in Figure 3-7. For the second join between the result of the first join and the `Tournament` table, the condition is that `e.TourID = t.TourID` as shown by the circles. It will not make any difference if we choose to do the join between `Entry` and `Tournament` first and then join the result to `Member`.

The SQL to carry out the two joins is:

```
SELECT *
FROM (Member m INNER JOIN Entry e ON m.MemberID = e.MemberID)
      INNER JOIN Tournament t ON e.TourID = t.TourID;
```

The virtual table resulting from the two joins in this query has all the information we require to answer our question. We just need to select the rows satisfying the conditions about the year and tournament name by adding a `WHERE` clause, and then project the name attributes by specifying them in the `SELECT` clause. The complete SQL query to return the names of everyone who entered the Leeston tournament in 2014 is:

```
SELECT LastName, FirstName
FROM (Member m INNER JOIN Entry e ON m.MemberID = e.MemberID)
      INNER JOIN Tournament t ON e.TourID = t.TourID
WHERE TourName = 'Leeston'
AND Year = 2014;
```

Order of Operations

In the description in the previous section, we joined all the tables first and then selected the appropriate rows and columns. The result of the join is an intermediate table (as in Figure 3-7) that is potentially extremely large if there are lots of members and tournaments. We could have done the operations in a different order. We could have first selected just the Leeston tournament from the `Tournament` table and the 2014 tournaments from the `Entry` tables, as shown in Figure 3-8. Joining these two smaller tables with each other and then joining that result with `Member` would result in a much smaller intermediate table.

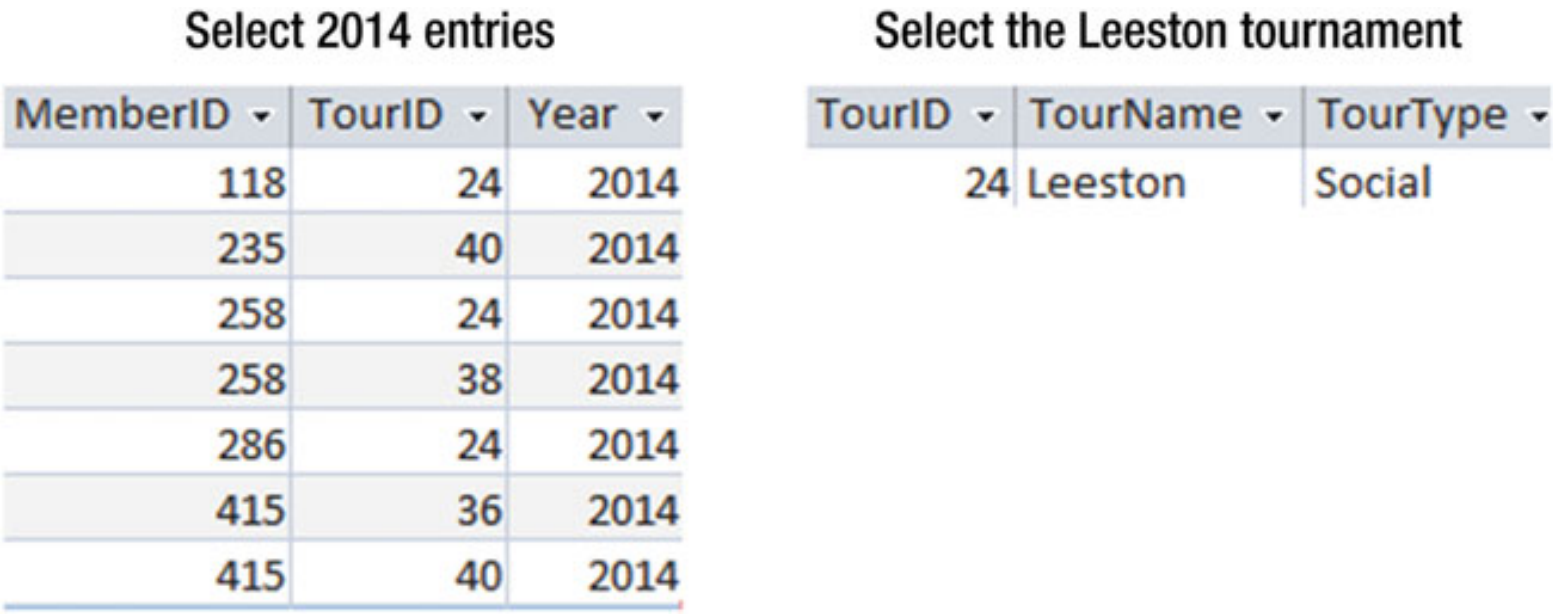


Figure 3-8. *Selecting rows from the Entry and Tournament tables before joining them*

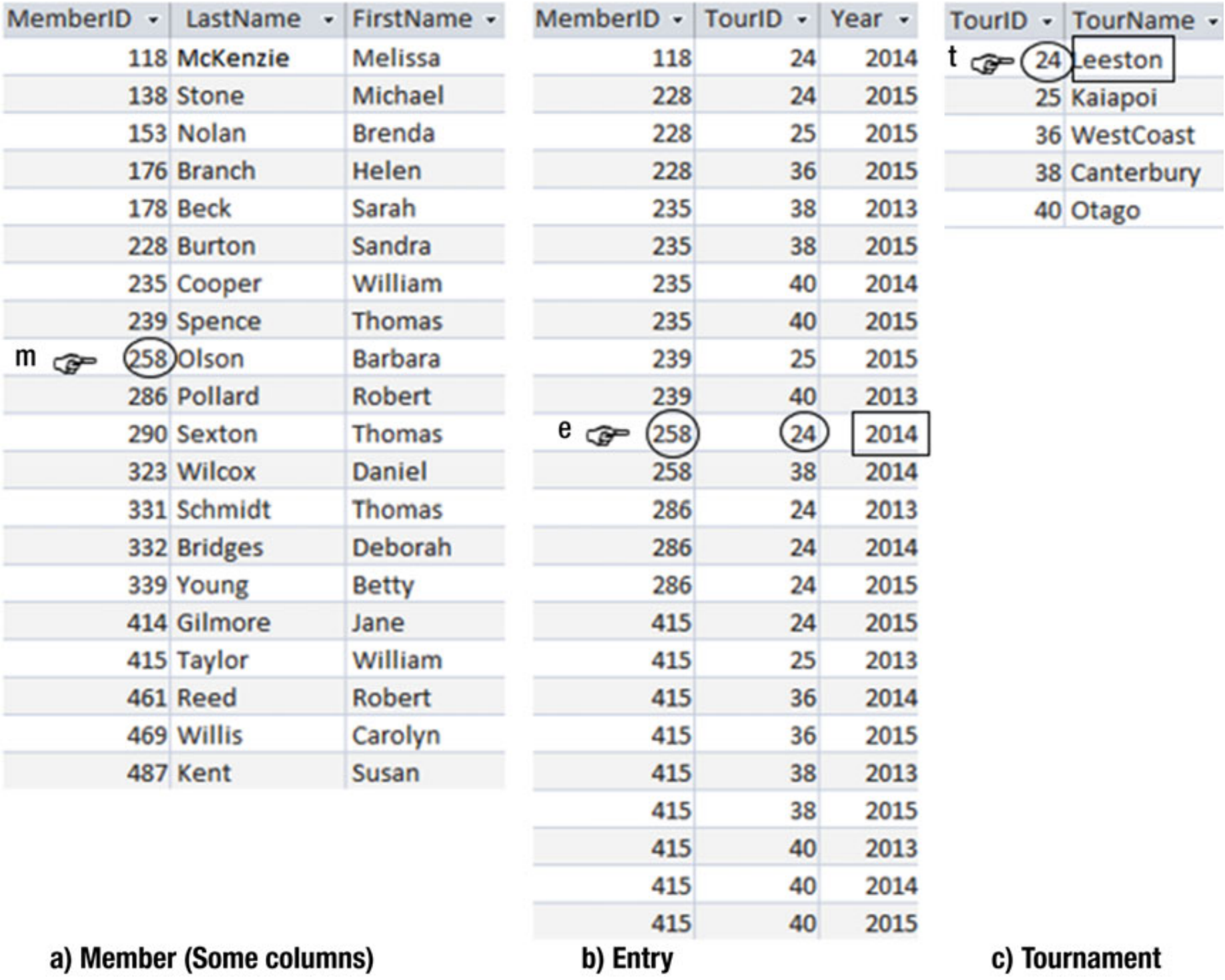


Figure 3-9. *Using row variables to describe the rows that satisfy the query conditions*

I'll write out the names from row `m`*, where* `m` *comes from the* `Member` *table, if there is a row* `e` *in the Entry table where* `m.MemberID` *is the same as* `e.MemberID` *and* `e.Year` *is 2014 and there also exists a row* `t` *in the* `Tournament` *table where* `e.TourID` *is the same as* `t.TourId` *and* `t.TourName` *has the value "Leeston."*

The SQL reflects the preceding paragraph. Look carefully at the following statement with reference to Figure 3-9:

```
SELECT m.LastName, m.FirstName
FROM Member m, Entry e, Tournament t
WHERE m.MemberID = e.MemberID
      AND e.TourID = t.TourID
      AND t.TourName = 'Leeston' AND e.Year = 2014;
```

You can see how the SQL statement describes *what* a retrieved row should be like. If you look carefully at the statement, you can also spot the operations. The second line (the `FROM` clause) is a big Cartesian product, the next two lines are the join conditions (which would result in a table like the one in Figure 3-7), the final line selects the rows with the appropriate year and tournament name, and the `SELECT` clause line tells us to project just the names.

The SQL preceding statement is equivalent to the one using the `INNER JOIN` keywords. They will both return the same set of rows: one reflects the underlying process of *how*, and the other reflects the underlying outcome of *what.*

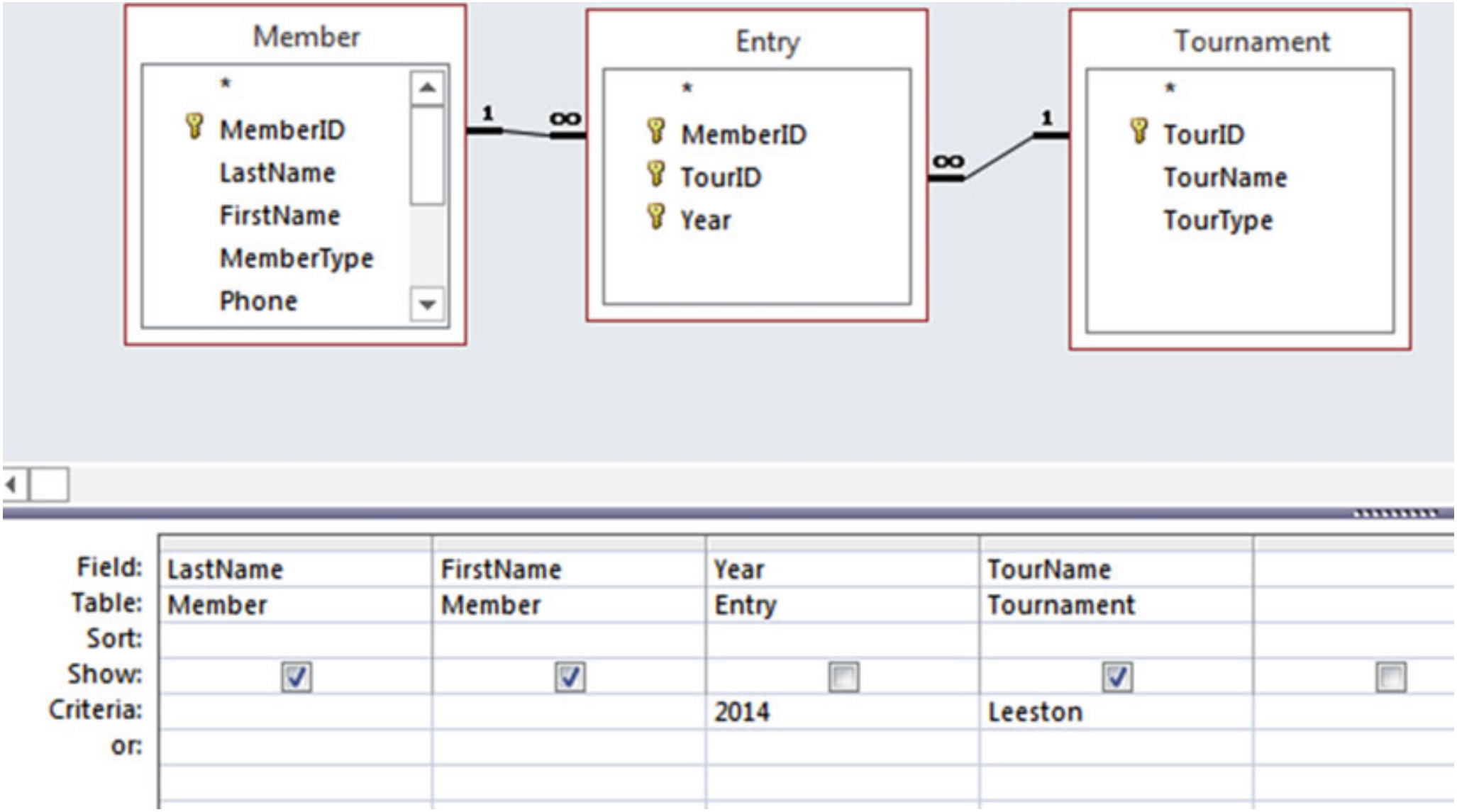


Figure 3-10. *Microsoft Access digrammatic interface for the query to find names of members entering the Leeston tournament in 2014*

Other Types of Joins

The joins we have been looking at in this chapter are *equi-joins*. An equi-join is one where the join condition has an equals operator, as in `m.MemberID = e.MemberID`. This is the most common type of condition, but you can have different operators. A join is just a Cartesian product followed by selecting a subset of rows, and the select condition can consist of different comparison operators (for example, <> or >) and also logical operators (for example, `AND` or `NOT`). These sorts of joins don't turn up all that often.

You might also come across a *natural join*. A natural join assumes that you will be joining on columns that have the same name in both tables. The join condition is that the values in the two columns with the same name are equal, and one of those columns will be removed from the result. For example:

```
SELECT * FROM
Member NATURAL JOIN Entry;
```

This would produce almost the same output as:

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

In the natural join statement, the join condition is implicitly assumed to be equality between the two attributes with the same name, `MemberID`. The only difference between the two queries is that for the natural join only one of the `MemberID` columns will be returned. Oracle supports natural joins but SQL Server and Access do not.

Outer Joins

One type of join that you will use a great deal and that is important to understand is the *outer join*. The best way to understand an outer join is to see where they are useful. Have a look at the (modified) `Member` and Type tables in Figure 3-11.

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

(modified) Member table

Type	Fee
Associate	60
Junior	150
Senior	300
Social	50

Type table

Figure 3-11. *Member and Type tables*

You might want to produce different lists from the `Member` table, such as numbers and names, names and membership types, and so on. In these lists you expect to see all the members (for the table in Figure 3-11, that would be nine rows). Then you might think that as well as seeing the numbers and names in your member list, you will also include the membership fee. You join the two tables (with the condition `MemberType = Type`) and find that you "lose" one of your members – Sarah Beck (see Figure 3-12).

MemberID	LastName	FirstName	MemberType	Type	Fee
118	McKenzie	Melissa	Junior	Junior	150
138	Stone	Michael	Senior	Senior	300
153	Nolan	Brenda	Senior	Senior	300
176	Branch	Helen	Social	Social	50
228	Burton	Sandra	Junior	Junior	150
235	Cooper	William	Senior	Senior	300
239	Spence	Thomas	Senior	Senior	300
258	Olson	Barbara	Senior	Senior	300

Figure 3-12. *Inner join between Member and Type, and we "lose" Sarah Beck*

The reason is that Sarah has no value for `MemberType` in the `Member` table. Let's look at the Cartesian product, which is the first step for doing a join. Figure 3-13 shows those rows of the Cartesian product that include Sarah.

MemberID	LastName	FirstName	MemberType	Type	Fee
176	Branch	Helen	Social	Associate	60
176	Branch	Helen	Social	Junior	150
176	Branch	Helen	Social	Senior	300
176	Branch	Helen	Social	Social	50
178	Beck	Sarah		Associate	60
178	Beck	Sarah		Junior	150
178	Beck	Sarah		Senior	300
178	Beck	Sarah		Social	50
228	Burton	Sandra	Junior	Associate	60
228	Burton	Sandra	Junior	Junior	150
228	Burton	Sandra	Junior	Senior	300
228	Burton	Sandra	Junior	Social	50

Figure 3-13. *Part of the Cartesian product between the Member and Type tables*

Having done the Cartesian product, we now need to do the final part of our join operation, which is to apply the condition (`MemberType = Type`). As you can see in Figure 3-13, there is no row for Sarah Beck that satisfies this condition because she has a null or empty value in `MemberType`.

MemberID	LastName	FirstName	MemberType	Type	Fee
118	McKenzie	Melissa	Junior	Junior	150
138	Stone	Michael	Senior	Senior	300
153	Nolan	Brenda	Senior	Senior	300
176	Branch	Helen	Social	Social	50
178	Beck	Sarah			
228	Burton	Sandra	Junior	Junior	150
235	Cooper	William	Senior	Senior	300
239	Spence	Thomas	Senior	Senior	300
258	Olson	Barbara	Senior	Senior	300

Figure 3-14. *Result of left outer join between Member and Type tables*

The SQL for the outer join depicted in Figure 3-14 is similar to an inner join, but the key phrase `INNER JOIN` is replaced with `LEFT OUTER JOIN` (or in some applications simply `LEFT JOIN`):

```
SELECT *
FROM Member m LEFT OUTER JOIN Type t ON m.MemberType = t.Type;
```

You might quite reasonably say that we wouldn't have needed an outer join if all the members had a value for the MemberType field (as they probably should). That may be true for this case – but remember my cautions in Chapter 2 about assuming that fields that *should* have data *will* have data. In other situations, the data in the join field may be quite legitimately empty. We will see in later chapters queries like "List all members and the names of their coaches – if they have one." "Losing" rows because you have used an inner join when you should have used an outer join is a very common problem and is sometimes quite hard to spot.

What about right and full outer joins? Left and right outer joins are the same and just depend on which order you put the tables in the join statement. The following SQL statement will return the same information as displayed in Figure 3-14, although the columns may be presented in a different order:

```
SELECT *
FROM Type t RIGHT OUTER JOIN Member m ON m.MemberType = t.Type;
```

We have simply swapped the order of the tables in the join statement. Any rows with a null in the join field of the right table (Member) will be included.

A full outer join will retain rows with a null in the join field in either table. The SQL for the full outer join is shown here and will result in the table seen in Figure 3-15:

```
SELECT *
FROM Member m FULL OUTER JOIN Type t ON m.MemberType = t.Type;
```

MemberID	LastName	FirstName	MemberType	Type	Fee
				Associate	60
118	McKenzie	Melissa	Junior	Junior	150
138	Stone	Michael	Senior	Senior	300
153	Nolan	Brenda	Senior	Senior	300
176	Branch	Helen	Social	Social	50
178	Beck	Sarah			
228	Burton	Sandra	Junior	Junior	150
235	Cooper	William	Senior	Senior	300
239	Spence	Thomas	Senior	Senior	300
258	Olson	Barbara	Senior	Senior	300

Figure 3-15. *Result of a full outer join between Member and Type tables*

We have a row for Sarah Beck padded with mull values for the missing columns from the `Type` table. We also have the first row, which shows us the information about the Associate membership type even though there are no rows in the `Member` table with Associate as a member type. In this row, each missing value from the `Member` table is replaced with a null.

A Cartesian product combines two tables. The resulting table has a column for each column in the two tables, and there is a row for every combination of rows from the contributing tables. The SQL for a Cartesian product reflecting the process approach is:

```
SELECT *
FROM <table1> CROSS JOIN <table2>;
```

The SQL for an inner join reflecting the outcome approach is:

```
SELECT *
FROM <table1>,<table2>;
```

An inner join starts with a Cartesian product, and then a join condition determines which combinations of rows from the two contributing tables will be retained.

The SQL for an inner join reflecting the process approach is:

```
SELECT *
FROM <table1> INNER JOIN <table2>
ON <join condition>;
```

The SQL for an inner join reflecting the outcome approach is:

```
SELECT *...
FROM <table1>, <table2>
WHERE <join condition>;
```

If one (or both) of the tables has rows with a null in the field involved in the join condition, then that row will not appear in the result for an inner join. If that row is required, you can use outer joins.

The SQL for an outer join, which will retain all the rows in the left-hand table including those with a null in the join field, is:

```
SELECT *
FROM  <table1> LEFT OUTER JOIN <table2>
ON <join condition>;
```

Similar expressions exist for right outer joins and full outer joins.