One of the great strengths of relational database theory is that the tables (or, more formally, the relations) are made up of *distinct* rows and so can be considered a *set*. We can then use set operations to help with combining and extracting specific information. The types of questions that set operations help with are those such as "which people are in both these sets?" or "which people are in this set but not that one?"

In Appendix 2 you can find some formal notation that is helpful with managing set operations. In this chapter we will keep formalities to a minimum, but the symbols for the set operations are a useful shorthand. Table 7-1 shows the four set operations we will look at along with their common symbols and the associated SQL keyword (for those that have them).

Table 7-1. *Four Set Operations and Their Symbols*

Operation	Symbol	SQL Keyword
Union	∪	`UNION`
Intersection	∩	`INTERSECT`
Difference	–	`EXCEPT`
Division	÷	

Not all implementations of SQL support all the keywords in Table 7-1, so we will look at alternative ways to achieve the same result when the keywords are not available.

Overview of Basic Set Operations

We will look at each of the set operations in turn, but so that you know where we are heading, I'll just give a very quick overview of the three most common operations: union, intersection, and difference. Imagine we have membership tables from two golf clubs. We might want to do the following:

- Determine who is in both clubs.
- Form a large list that combines all the members.
- Find out who is in one club but not the other.

The basic set operations allow us to carry out all these tasks.

Let's assume that the each club keeps the names of its members in a table. The two tables have exactly the same columns (more about this in the next section) and are shown in Figure 7-1. (OK, they are very small clubs!)

LastName	FirstName
Cooper	William
Gilmore	Jane
Kent	Susan
McKenzie	Melissa
Nolan	Brenda
Olson	Barbara
Pollard	Robert

ClubA

LastName	FirstName
Olson	Barbara
Pollard	Robert
Reed	Robert
Schmidt	Thomas
Sexton	Thomas

ClubB

Figure 7-1. *Two tables of member names*

The basic set operations on these two tables are summarized in Figure 7-2. The images of two club tables have been overlaid so that the members in common are superimposed. `ClubA` is the top table in each picture. For each section of Figure 7-2, the box shows the result of the set operation.

Cooper	William
Gilmore	Jane
Kent	Susan
McKenzie	Melissa
Nolan	Brenda
Olso	
Polla	

Olson	Barbara
Pollard	Robert
Reed	Robert
Schmidt	Thomas
Sexton	Thomas

a) ClubA $\cup$ ClubB
Union
(All unique rows from both tables)

Cooper	William
Gilmore	Jane
Kent	Susan
McKenzie	Melissa
Nolan	Brenda
Olso	
Polla	

Olson	Barbara
Pollard	Robert
Reed	Robert
Schmidt	Thomas
Sexton	Thomas

b) ClubA $\cap$ ClubB
Intersection
(Rows that are in both tables)

Cooper	William
Gilmore	Jane
Kent	Susan
McKenzie	Melissa
Nolan	Brenda
Olso	
Polla	

Olson	Barbara
Pollard	Robert
Reed	Robert
Schmidt	Thomas
Sexton	Thomas

c) ClubA – ClubB
Difference
(Rows in ClubA but not in ClubB)

Cooper	William
Gilmore	Jane
Kent	Susan
McKenzie	Melissa
Nolan	Brenda
Olso	
Polla	

Olson	Barbara
Pollard	Robert
Reed	Robert
Schmidt	Thomas
Sexton	Thomas

d) ClubB – ClubA
Difference
(Rows in ClubB but not in ClubA)

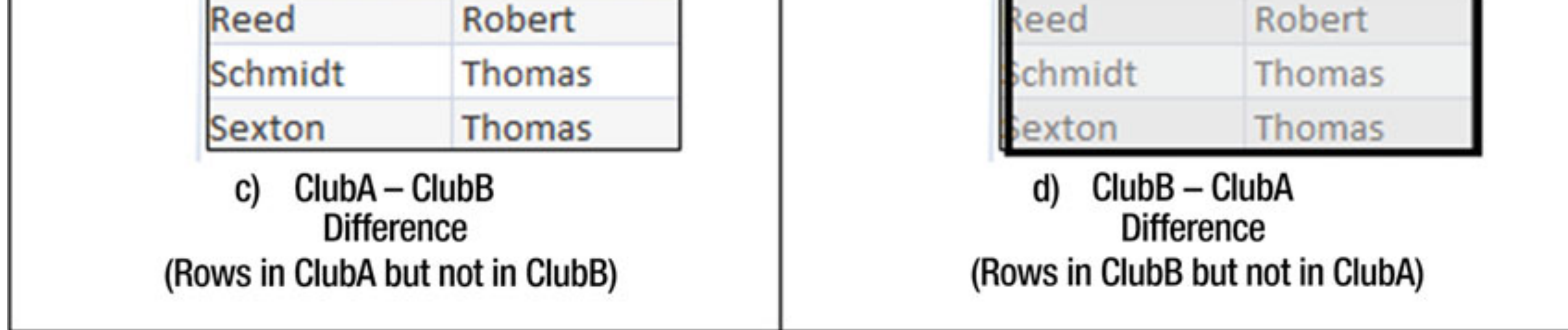


Figure 7-2. *The basic set operations on the two tables ClubA (top) and ClubB (bottom)*

The union operator (top left in Figure 7-2) shows all the names from each table (with duplicates removed). The intersect operator (top right) returns the two rows that appear in both tables. The difference operators (bottom) return those rows that are found in one club but not the other.

Union-Compatible Tables

The set operations union, intersection, and difference operate between two sets of rows. It does not make any sense to try to compare rows in tables that have very different structures, such as those in Figure 7-3.

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

Member Table

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

Entry Table

Figure 7-3. *It makes no sense to try to compare rows from tables with different structures*

So what determines whether two sets of rows can be compared using the set operations union, intersection, and difference? Formally, the two sets must have the same number of columns, and each column must have the same domain. Strictly speaking, a *domain* is a set of possible values. However, in practice, the requirement for set operations is that the corresponding columns (i.e., in order from left to right) in each set of rows have the same types – both character, both integer, and so on.[1] The names of the columns do not need to be the same. Tables that meet these requirements are referred to as being *union compatible*, although the requirement is necessary for the intersection and difference operations as well.

Figure 7-4 shows a pair of tables that are union compatible. Even though the names of the columns are different, they have the same number of columns, and the corresponding columns have the same types.

MemberID	LastName	FirstName	Handicap	MemberType
176	Branch	Helen		Social
178	Beck	Sarah		Social
228	Burton	Sandra	26	Junior
235	Cooper	William	14	Senior
239	Spence	Thomas	10	Senior
258	Olson	Barbara	16	Senior
286	Pollard	Robert	19	Junior
290	Sexton	Thomas	26	Senior

ClubA

RegNum	FamilyName	Name	Handicap	Grade
239	Spence	Thomas	10	Senior
258	Olson	Barbara	16	Senior
286	Pollard	Robert	19	Junior
290	Sexton	Thomas	26	Senior
323	Wilcox	Daniel	3	Senior
331	Schmidt	Thomas	25	Senior
332	Bridges	Deborah	12	Senior
339	Young	Betty	21	Senior

ClubB

Figure 7-4. *Union-compatible tables, even though column names are different*

Figure 7-5 has two tables with the same column names, but they are not union compatible because the order of the columns is such that the fourth column has a number type in the top table and a character type in the bottom, and vice versa for the last column.

MemberID	LastName	FirstName	Handicap	MemberType
176	Branch	Helen		Social
178	Beck	Sarah		Social
228	Burton	Sandra	26	Junior
235	Cooper	William	14	Senior
239	Spence	Thomas	10	Senior
258	Olson	Barbara	16	Senior
286	Pollard	Robert	19	Junior
290	Sexton	Thomas	26	Senior

ClubC

MemberID	LastName	FirstName	MemberType	Handicap
239	Spence	Thomas	Senior	10
258	Olson	Barbara	Senior	16
286	Pollard	Robert	Junior	19
290	Sexton	Thomas	Senior	26
323	Wilcox	Daniel	Senior	3
331	Schmidt	Thomas	Senior	25
332	Bridges	Deborah	Senior	12
339	Young	Betty	Senior	21

ClubD

Figure 7-5. *Tables that are not union compatible*

332	Bridges	Deborah	Senior	12
339	Young	Betty	Senior	21

ClubD

Figure 7-5. Tables that are not union compatible

Different implementations of SQL may interpret the strictness of this requirement for the "sameness" of domains or types differently. Strictly speaking, two fields defined as CHAR(10) and CHAR(12) have different domains, but many implementations of SQL will allow these to be regarded as the same for the purposes of set operations. Some implementations will also convert numbers into characters to enable set operations to be carried out. I find this particularly scary and don't recommend you let your application make these sorts of decisions for you. The following sections demonstrate how you can use SQL to make your tables union compatible.

Ensuring Union Compatibility

When tables are not union compatible, you can often remedy the incompatibility in the SELECT clauses.

For the pair of tables in Figure 7-5, if we just select the columns as follows the order of the columns will prevent the returned rows from being union compatible:

```
SELECT * FROM ClubC;
SELECT * FROM ClubD;
```

However, we can specify the order of the columns in the SELECT clause:

```
SELECT MemberID, LastName, FirstName, Handicap, MemberType FROM ClubC;
SELECT MemberID, LastName, FirstName, Handicap, MemberType FROM ClubD;
```

The two sets of rows returned from these queries are now union compatible.

Another incompatibility problem occurs when the types of the columns have been declared as different types in the original design of the tables. For example, the ClubC table may have the Handicap field declared as an INT, whereas the ClubD table may have (unwisely) stored the Handicap values in a CHAR field. (Recall from Chapter 2 that if we store values in a character or text field then they will order alphabetically, and we will not be able to perform functions such as average on them.) As mentioned earlier, different implementations of SQL will treat these different types in a variety of ways. Many will try to convert the numbers to text or vice versa. You can take control of these conversions yourself (which is probably a good idea) by using type-conversion functions.

For example, in SQL Server, the expression Convert(INT, Handicap) would take a text value in the Handicap field ("14") and convert it to an integer value (14). (If the value in the Handicap field wasn't able to be converted to an integer then an error would occur.) If the Handicap field in the ClubD table were a CHAR type then we could use the conversion function in the SELECT clause. The two sets of rows returned by the following queries will now be union compatible:

```
SELECT MemberID, LastName, FirstName, Handicap FROM ClubC;
SELECT MemberID, LastName, FirstName, Convert(INT, Handicap) FROM ClubD;
```

Union

Union allows us to produce output consisting of all the unique rows from two union-compatible sets of rows To carry out a union in SQL, we need to first retrieve two sets of rows using two SELECT clauses and then combine the two sets with the UNION keyword. The following SQL shows the union of all the rows from the two union-compatible tables (ClubA and ClubB) shown in Figure 7-4.

```
SELECT * FROM ClubA
UNION
SELECT * FROM ClubB;
```

The resulting table will include all the rows from both tables with no duplicates, so you will see only one row each for Barbara Olson, Robert Pollard, and Thomas Sexton, as shown in Figure 7-6. If you wish to retain the duplicates for some reason, you can use the key phrase UNION ALL.

MemberID	LastName	FirstName	Handicap	MemberType
176	Branch	Helen		Social
178	Beck	Sarah		Social
228	Burton	Sandra	26	Junior
235	Cooper	William	14	Senior
239	Spence	Thomas	10	Senior
258	Olson	Barbara	16	Senior
286	Pollard	Robert	19	Junior
290	Sexton	Thomas	26	Senior
323	Wilcox	Daniel	3	Senior
331	Schmidt	Thomas	25	Senior
332	Bridges	Deborah	12	Senior
339	Young	Betty	21	Senior

Figure 7-6. *Union of ClubA and ClubB with no duplicate rows*

As union-compatible tables do not need to have the same column names, the names of the columns in the resulting virtual table will usually be from one of the tables. In the example in Figure 7-6, the column names are the same as the first table mentioned in the union query.

It does not matter for the union operator in which order the two tables are specified. The query that follows will return the same rows as the previous query did. The rows may appear in a different order, and the displayed names of the columns may change, but the data will be the same.

```
SELECT * FROM ClubB
UNION
SELECT * FROM ClubA;
```

Selecting the Appropriate Columns

When using the union operator you need to think carefully about what it is you actually want. The examples with the clubs are rather contrived (as you have no doubt noticed). It is very unlikely that two clubs would have members with the same ID numbers and identical membership types. A more likely scenario is that if Barbara Olson did belong to two clubs, she would have different data in each club table. In the `ClubA` table, she might be a Senior with a value of 258 for `MemberID`. In the `ClubB` table, she might be an Associate with a value of 4573 for `RegNum`. If we do the union shown in Figure 7-6, where we select all the columns from each table, the two rows for Barbara will be different, and so both will appear in the result, as in Figure 7-7.

MemberID	LastName	FirstName	MemberType	Handicap
176	Branch	Helen	Social	
178	Beck	Sarah		
228	Burton	Sandra	Junior	26
235	Cooper	William	Senior	14
239	Spence	Thomas	Senior	10
258	Olson	Barbara	Senior	16
4573	Olson	Barbara	Associate	16
286	Pollard	Robert	Junior	19

Figure 7-7. *Two records appear for Barbara Olson in the union because the rows are different*

We need to consider what we really want from such a union. If we need a list of names for a joint Christmas party for the two clubs, then we don't want everyone listed twice. The way to avoid duplicates is to project just the names from each table before carrying out the union:

```
SELECT FamilyName, Name FROM ClubA
UNION
SELECT LastName, FirstName FROM ClubB;
```

With this query the two rows for Barbara will be the same and will only appear in the union once, as in Figure 7-8.

With this query the two rows for Barbara will be the same and will only appear in the union once, as in Figure 7-8.

LastName	FirstName
Branch	Helen
Beck	Sarah
Burton	Sandra
Cooper	William
Spence	Thomas
Olson	Barbara
Pollard	Robert

Figure 7-8. *Only one row appears for Barbara Olson if only the name columns are in the union*

There is, of course, a serious issue with this last query. There may be two Barbara Olsons, one in each club, and now only one nametag will be printed for the pair of them. Sadly, real data is fraught with these sorts of problems. With any luck there will be some universal national golf association number that might sort this out, but if not you just need to be alert. The intersection operation, discussed in the next section, would produce the names that appear in both club lists, and a manual sanity check could be carried out.

The main use for union is combining data from two or more tables, as we have been doing in the previous sections. For example, if the tournament entry data for different months had been stored in separate tables (not a great design decision!), we could use several union operations to combine the data for the whole year.

It is also possible to combine two sets of rows from the one table. Say we wanted to know how many people have entered either tournament 24 or tournament 36 from the `Entry` table in Figure 7-9.

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

415	36	2015
235	38	2013
235	38	2015

Figure 7-9. *Entry table*

We could try selecting the rows for members entering tournament 24 and the rows for members entering tournament 36, and take the union. How many rows will we get if we perform the following query?

```
SELECT * FROM Entry WHERE TourID = 24
UNION
SELECT * FROM Entry WHERE TourID = 36;
```

We will get ten rows from this query, one for every row with a 24 or a 36. Because we have retained the `TourID` and `Year` columns, the rows we have selected are all different and so will all appear in the result of the union. The query is actually returning all the distinct *entries* into tournaments 24 and 36 rather than all the distinct *members* who have entered the two tournaments. The flowing query takes the union of just the IDs for the two tournaments:

```
SELECT MemberID FROM Entry WHERE TourID = 24
UNION
SELECT MemberID FROM Entry WHERE TourID = 36;
```

Now we will get the five IDs (118, 228, 258, 286, 415) that are the unique IDs for those entering one or the other of the tournaments.

There is a much simpler way of retrieving those who have entered either tournament 24 or 36. We can simply include an OR in the WHERE clause:

```
SELECT MemberID FROM Tournament
WHERE TourID = 24 OR TourID = 36;
```

How many rows will the preceding query return? Again, it will return ten rows–each of the rows with a 24 or a 36 in the TourID column. To get the five unique IDs, we need to add the DISTINCT keyword in the SELECT clause.

Union and Full Outer Joins

In Chapter 3 we looked at different join operations: inner joins, left and right outer joins, and full outer joins. Some products (e.g., Microsoft Access 2013) do not support the FULL OUTER JOIN keyword; however, we can perform an equivalent query using the UNION keyword.

To recap, let's review the different types of join we can carry out between the Member table (just a very little one!) and the Type table shown in Figure 7-10.

MemberID	LastName	FirstName	MemberType
118	McKenzie	Melissa	Junior
138	Stone	Michael	Senior
153	Nolan	Brenda	Senior
176	Branch	Helen	
178	Beck	Sarah	Social

Member

Type	Fee
Associate	60
Junior	150
Senior	300
Social	50

Type

176	Branch	Helen	
178	Beck	Sarah	Social

Member

Social	50

Type

Figure 7-10. *The (small) Member and Type tables*

Figure 7-11 shows the inner join between the two tables, with join condition `MemberType = Type`. We do not get a row for Helen Branch because she has no value in `MemberType` and so the join condition will never be true for her. This may be a problem if someone looking at the table in Figure 7-11 assumes it is showing all members.

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

Figure 7-11. *The inner join between Member and Type on MemberType = Type*

Now we will look at the outer joins. The left outer join ensures that we see all the rows from the left-hand table (`Member`); the right outer join gives us all rows from the right-hand table (`Type`); and the full outer join gives us all rows from both tables. These outer joins, all with join condition `MemberType = Type`, are shown in Figure 7-12.

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			
178	Beck	Sarah	Social	Social	50

Member Left Join 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
178	Beck	Sarah	Social	Social	50

Member Right Join 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			
178	Beck	Sarah	Social	Social	50

Member Full Join Type

Figure 7-12. *Three outer joins between Member and Type on MemberType = Type*

Figure 7-12 shows that, in this case, the full outer join consists of the unique rows from each of the other two outer joins; that is, a union. If your SQL implementation does not explicitly support a full outer join, you can always achieve the same result with the following query:

```
SELECT * FROM Member LEFT JOIN Type ON MemberType = Type
UNION
SELECT * FROM Member RIGHT JOIN Type ON MemberType = Type;
```

If you take the intersection of two union-compatible tables, you will retrieve those rows that are found in both tables. Figure 7-13 reproduces the two tables, ClubA and ClubB, from Figure 7-4. We can see that there are four rows that are identical in both tables.

MemberID	LastName	FirstName	Handicap	MemberType
176	Branch	Helen		Social
178	Beck	Sarah		Social
228	Burton	Sandra	26	Junior
235	Cooper	William	14	Senior
239	Spence	Thomas	10	Senior
258	Olson	Barbara	16	Senior
286	Pollard	Robert	19	Junior
290	Sexton	Thomas	26	Senior

ClubA

RegNum	FamilyName	Name	Handicap	Grade
239	Spence	Thomas	10	Senior
258	Olson	Barbara	16	Senior
286	Pollard	Robert	19	Junior
290	Sexton	Thomas	26	Senior
323	Wilcox	Daniel	3	Senior
331	Schmidt	Thomas	25	Senior
332	Bridges	Deborah	12	Senior
339	Young	Betty	21	Senior

ClubB

As with the union operator, the two sets of rows must be union compatible; that is, they must have the same number of columns, and the corresponding columns must have the same domains. This may mean projecting the appropriate columns from the base tables in the same way as described in the "Selecting the Appropriate Columns" section earlier in this chapter. It makes no difference which of the tables we mention first in the query, as the rows returned by the intersection will be the same regardless of the order of the tables.

Uses of Intersection

A common use of the intersection operation is the one shown in Figure 7-13: finding common rows in two tables with similar information. Another very common use of intersection is answering questions that include the word *both*. A typical example is "Which members have entered *both* tournaments 36 and 38?" The `Entry` table is reproduced in Figure 7-14.

MemberID	TourID	Year
118	24	2014
228	24	2015
258	24	2014
286	24	2013
286	24	2014
286	24	2015
415	24	2015
228	25	2015
239	25	2015

Figure 7-13. The rows in the intersection between the ClubA and ClubB tables

The keyword for the intersection operator in SQL is INTERSECT . The expression to retrieve the four rows common to both tables (i.e., for members Spence, Olson, Pollard, and Sexton) is as follows:

```
SELECT * FROM ClubA
INTERSECT
SELECT * FROM ClubB;
```

What will be returned if we retrieve the rows for each tournament and take the intersection as in the following query?

```
SELECT * FROM Entry WHERE TourID = 36
INTERSECT
SELECT * FROM Entry WHERE TourID = 38;
```

There will be no rows returned. Figure 7-15 will help you understand why.

MemberID	TourID	Year
228	36	2015
415	36	2014
415	36	2015

SELECT * FROM Entry
WHERE TourID = 36

MemberID	TourID	Year
235	38	2013
235	38	2015
258	38	2014
415	38	2013
415	38	2015

SELECT * FROM Entry
WHERE TourID = 38

Figure 7-15. *Two queries have no rows in common, so no rows result from intersection*

The two queries will never have any rows in common because one will always have 36 in the `TourID` column while the other will always have 38. Essentially, the query we were trying to carry out was to find all the entries for tournament 36 that are also entries for tournament 38. The result, given the way we are managing entries, is none.

To retrieve the members who are in common in the two sets of rows in Figure 7-15, we must retrieve just the `MemberID` column before carrying out the intersection, as in the query here:

```
SELECT MemberID FROM Entry WHERE Tourid = 36
INTERSECT
SELECT MemberID FROM Entry WHERE Tourid = 38;
```

This query is illustrated in Figure 7-16. As with a union, the result of the intersection operation returns unique rows.

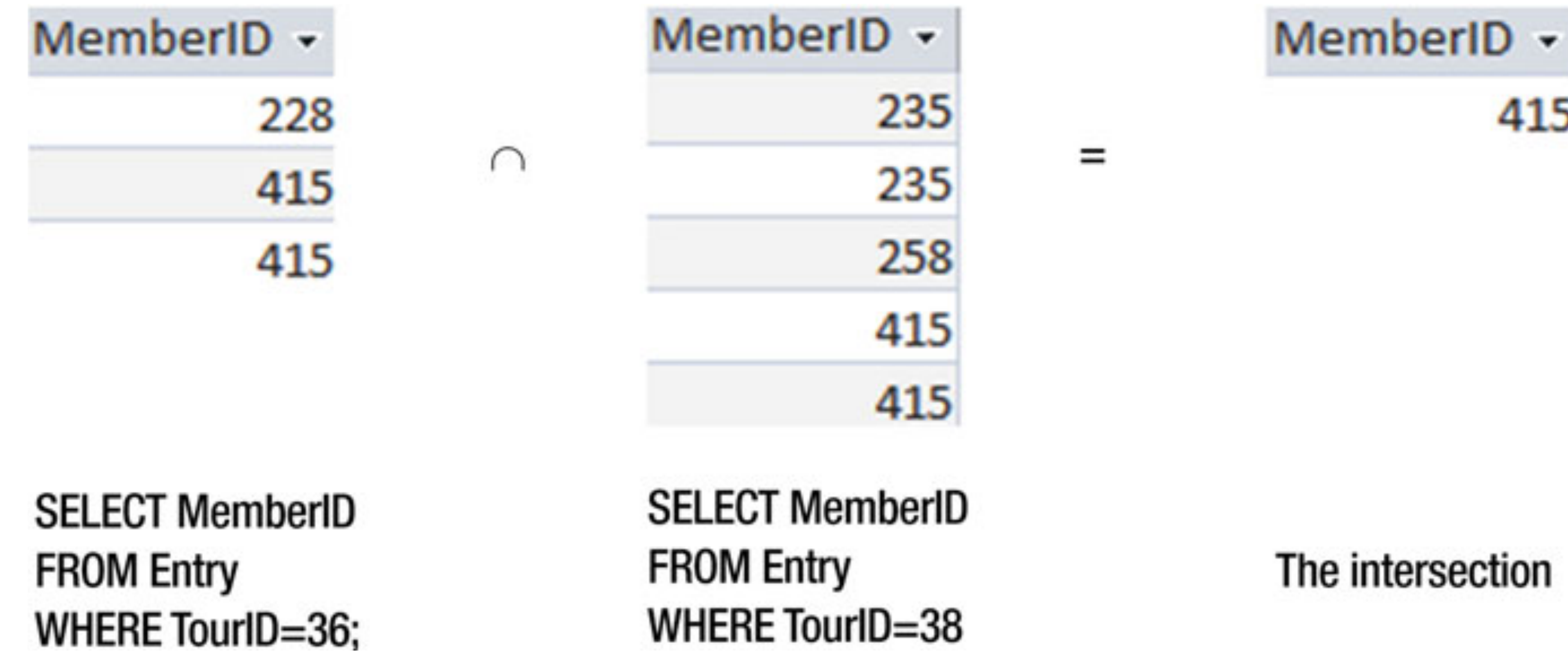


Figure 7-16. *Using intersection to find members entered in both tournaments 36 and 38*

Suppose we now want to find the names of the members. From a process point of view, we can take the result of the intersection and join it with the Member table to get the names, as shown in Figure 7-17.

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

Member Table

MemberID
415

Intersection

MemberID	LastName	FirstName
415	Taylor	William

Member Inner Join Intersection

Figure 7-17. *Joining the intersection with the Member table to find the names*

Member Table Intersection Member Inner Join Intersection

Figure 7-17. *Joining the intersection with the Member table to find the names*

So what does the SQL look like to first do the intersection and then join with the `Member` table? The following is a good first attempt, but unfortunately will not work:

```
--Will not work
SELECT LastName, FirstName
FROM Member m INNER JOIN
     (SELECT e1.MemberID FROM Entry e1 WHERE e1.TourID = 36
      INTERSECT
      SELECT e2.MemberID FROM Entry e2 WHERE e2.TourID = 38)
ON m.MemberID = e1.MemberID;
```

The tables that only appear inside the inner query (the part in parentheses) are not able to be referenced by the outer query (the join). This is easily resolved by giving the nested part of the query an alias. In the same way we have given the `Member` table an alias by putting an `m` after `Member` in the `FROM` clause, we can give the whole inner query an alias of `NewTable` (as an example) by putting `NewTable` after the final parenthesis of the inner query. We can now refer to that alias in the join condition as shown in the query here:

```
SELECT LastName, FirstName
FROM Member m INNER JOIN
     (SELECT e1.MemberID FROM Entry e1 WHERE e1.TourID = 36
```

```
      INTERSECT
      SELECT e2.MemberID FROM Entry e2 WHERE e2.TourID = 36) NewTable
ON m.MemberID = NewTable.MemberID;
```

Another way to retrieve the names is to use a nested query. Here, the inner query retrieves the IDs that are in the intersection, and the outer query finds the corresponding names from the Member table.

```
SELECT LastName, FirstName
FROM Member
WHERE MemberID IN
    (SELECT MemberID FROM Entry WHERE TourID = 36
     INTERSECT
     SELECT MemberID FROM Entry WHERE TourID = 38);
```

The Importance of Projecting Appropriate Columns

It is important to think very carefully about which columns are included in the tables involved in an intersection operation. We saw in the previous section how the following query will return no rows:

```
SELECT * FROM Entry WHERE TourID = 36
INTERSECT
SELECT * FROM Entry WHERE TourID = 38;
```

The rows from the first query will always have 36 as the value of `TourID` and the rows from the second query will have 38. There will never be any rows in common. Retrieving just the `MemberID` in each of the queries solves this problem.

More interesting is that correctly projecting different columns can provide answers to quite different questions. How would you describe the rows returned by the following query?

```
SELECT MemberID, Year FROM Entry WHERE TourID = 25
INTERSECT
SELECT MemberID, Year FROM Entry WHERE TourID = 36;
```

The query is illustrated in Figure 7-18.

MemberID	Year
228	2015
239	2015
415	2013

SELECT MemberID, Year
FROM Entry
WHERE TourID=25

∩

MemberID	Year
228	2015
415	2014
415	2015

SELECT MemberID, Year
FROM Entry
WHERE TourID=36

=

MemberID	Year
228	2015

What does the intersection mean?

Figure 7-18. *What does the intersection mean?*

In Figure 7-18, we are finding all the members who entered tournaments 25 and 36 in the *same year*. This is why there is no entry for member 415 in the intersection: he entered tournament 25 in 2013 and tournament 36 in 2014 and 2015. Although his member ID appears in the two contributing tables, the corresponding rows are for different years. There is no row for member 415 that is the same in both tables.

As you can see, the choice of columns that are projected for the contributing tables is fundamental to what will appear in the intersection. It means there are many different questions that can be answered very elegantly, but it also means that you can easily get incorrect answers if you don't think the query through carefully.

Managing Without the INTERSECT Keyword

Not all implementations of SQL support intersection explicitly. However, we have other ways to perform the queries involving "both." Intersection is a process approach – we are saying what operations we need to carry out on the tables involved in the query. If we don't succeed with this approach then we can try the outcome approach. This involves figuring out some possible answers by inspecting the tables and not worrying about operations such as intersections and joins. In Figure 7-19 we imagine two fingers traversing the rows of the Entry table. We need to find two rows in the `Entry` table with the same `MemberID`: one with `TourID` = 36 and one with `TourID` = 38.

MemberID	Year	TourID
415	2013	25
228	2015	36

	MemberID	Year	TourID
	415	2013	25
	228	2015	36
e1	415	2014	36
	415	2015	36
	235	2013	38
	235	2015	38
	258	2014	38
e2	415	2013	38
	415	2015	38
	235	2014	40

Figure 7-19. *Finding members who have entered both tournaments 36 and 38*

The situation that Figure 7-19 is depicting can be described as:

> *Return me the `MemberID` from a row `e1` in the `Entry` table where `TourID=36` if there is another row `e2` in the `Entry` table that has the same `MemberID` and `TourID=38`.*

The SQL expression equivalent to this description and Figure 7-19 is:

```
SELECT DISTINCT e1.MemberID
FROM Entry e1, Entry e2
WHERE e1.MemberID = e2.MemberID
AND e1.TourID = 36 AND e2.TourID = 38;
```

What about the query to find the rows that appear in both the `ClubA` and `ClubB` tables? The club tables are redisplayed in Figure 7-20. To find the rows that are the same in both tables we need to check each of the values in the corresponding columns to ensure they are the same.

	RegNum	FamilyName	Name
	176	Branch	Helen
	178	Beck	Sarah
	228	Burton	Sandra
	235	Cooper	William
	239	Spence	Thomas
a ☞	258	Olson	Barbara
	286	Pollard	Robert

ClubA

	MemberID	LastName	FirstName
b ☞	258	Olson	Barbara
	286	Pollard	Robert
	290	Sexton	Thomas
	323	Wilcox	Daniel
	331	Schmidt	Thomas
	332	Bridges	Deborah
	339	Young	Betty

Club B

332	Bridges	Deborah
339	Young	Betty

ClubB

Figure 7-20. *Finding the intersection between ClubA and ClubB*

The situation depicted in Figure 7-20 can be described as:

> *I will return row a from table ClubA if there is a row b in ClubB that has identical values in all the fields (i.e., a.RegNum = b.MemberID, a.FamilyName = b.LastName, and a.Name = b.FirstName).*

The SQL for the intersection shown in Figure 7-20 is:

```
SELECT a.RegNum, a.FamilyName, a.Name
FROM ClubA a, ClubB b
WHERE a.RegNum = b.MemberID
AND a.FamilyName = b.LastName
AND a.Name = b.FirstName;
```

Difference

Taking the difference between two tables finds those rows that are in the first table but not the second and vice versa. For our two tiny clubs, I have reproduced the results of the difference operator in Figure 7-21.

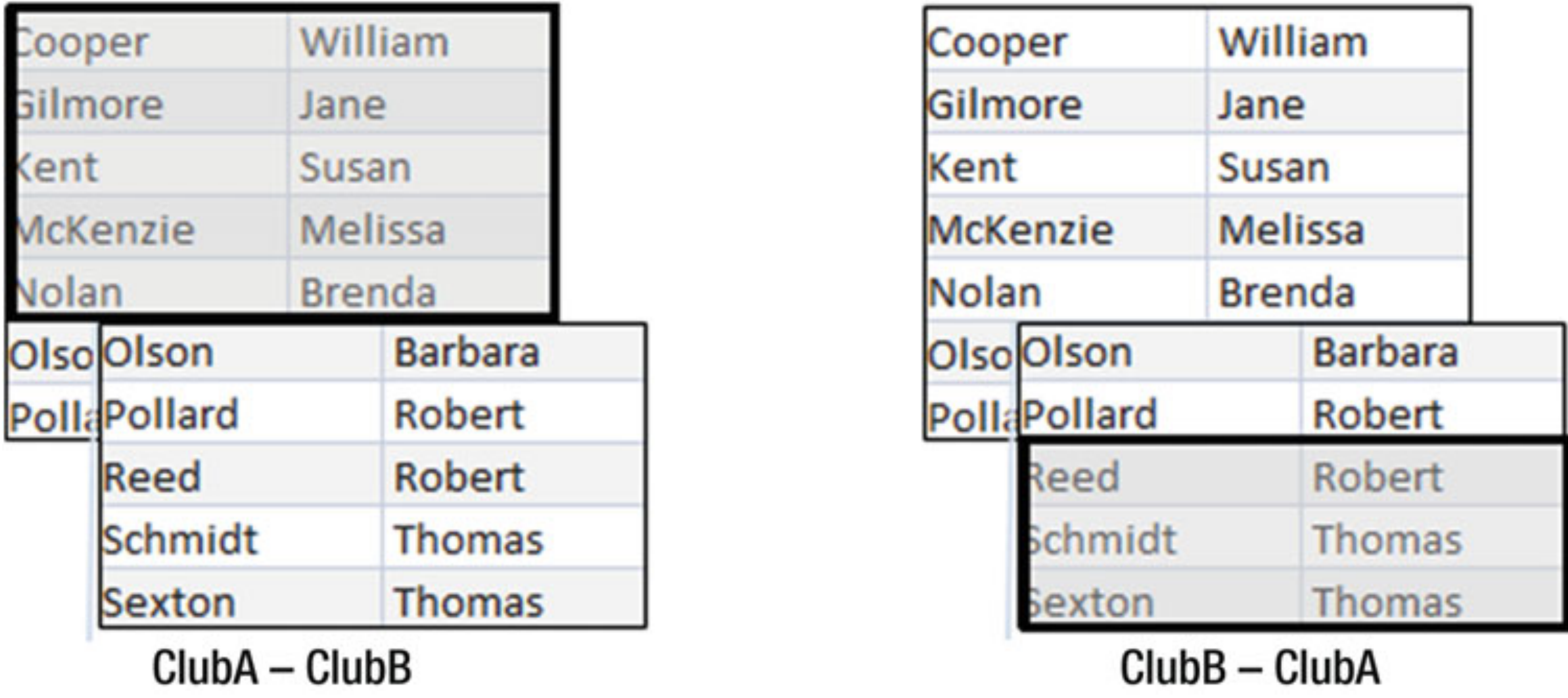


Figure 7-21. *The difference operator finds rows in one table that do not appear in the other.*

The keyword in standard SQL for the difference operator is `EXCEPT`. Oracle differs from the ISO SQL standard, and from most other database systems, in its use of the keyword `MINUS` rather than `EXCEPT`.

As with the union and intersection operators, the tables involved in a difference operation must be union compatible. Unlike with the union and intersection operators, the order of the tables is important for the difference operator; the results for `ClubA - ClubB` are different from those for `ClubB - ClubA` (as shown in Figure 7-21).

The SQL for finding the names of people in the `ClubA` table that do not appear in the `ClubB` table is:

```
SELECT LastName, FirstName FROM ClubA
EXCEPT
SELECT LastName, FirstName FROM ClubB;
```

Uses of Difference

Whenever you have a query that has the word “not,” you should consider the possibility that the difference operator will be useful. For example, how do we find members who have not entered tournament 25? Recall from Chapter 5 why the following query does not return those members who have not entered tournament 25:

```
SELECT MemberID FROM Entry
WHERE TourID <> 25;
```

The query above selects all the rows in the Entry table that are not for tournament 25. Essentially it finds a member who has entered any tournament other than 25 (although they could have entered 25 as well). Looking at Figure 7-15, we see that the query would return the row marked e1 for member 415 entering tournament 36 (TourID <> 25). However, two rows above, we see that member 415 has also entered tournament 25. It is difficult to think of a reason that you might ever want to use this query.

A process approach to this type of query is to use difference. We need to retrieve a set of the IDs of all members and another set of IDs for all the members who have entered tournament 25. We then want the difference; i.e., those IDs that are in the former set but not the latter.

Finding the set of all members who have entered tournament 25 is simple:

```
SELECT MemberID FROM Entry WHERE TourID = 25;
```

We might think a similar query will find us all the member IDs as well:

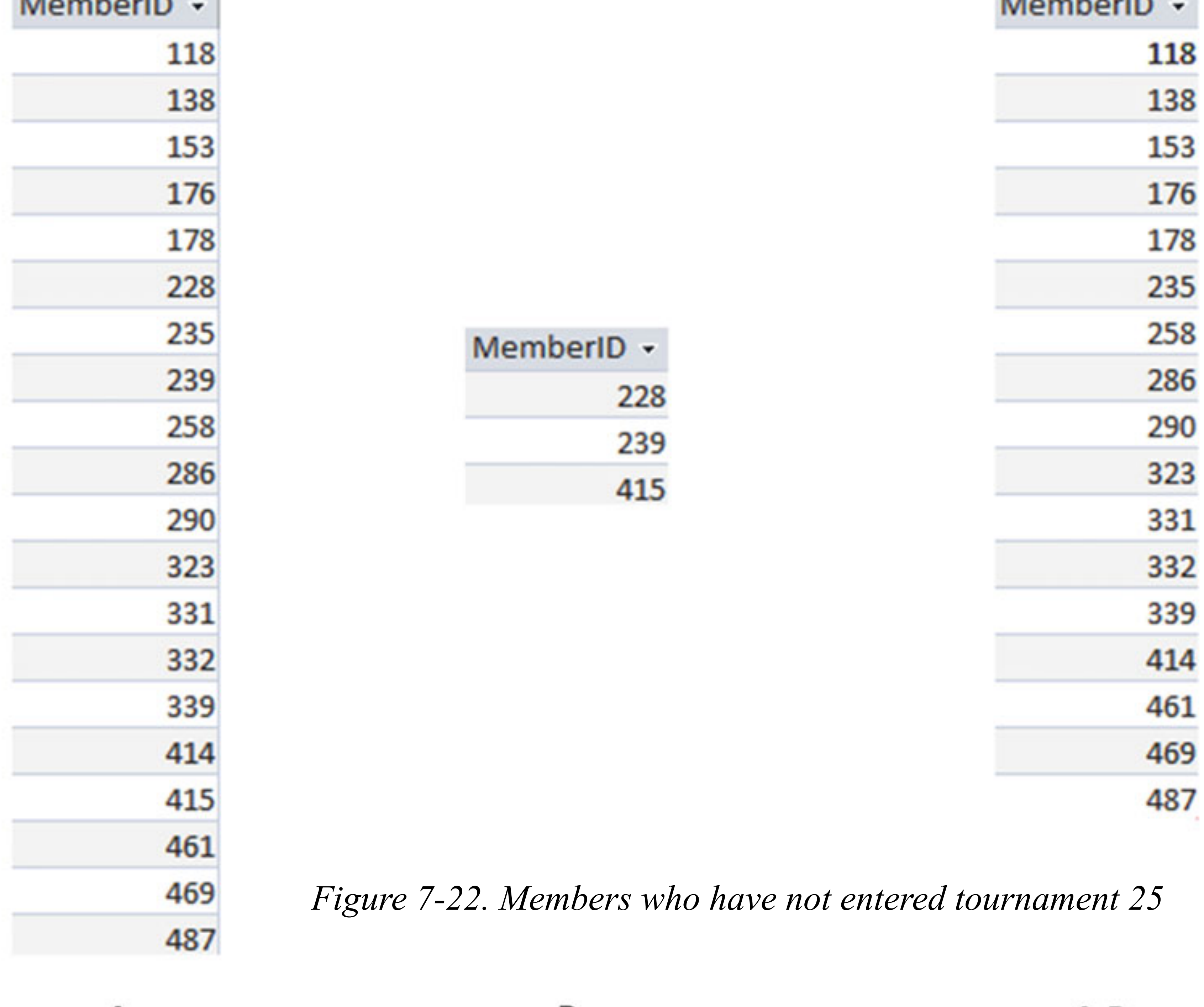
```
SELECT MemberID FROM Entry;
```

However, the preceding query only finds us a set of the members who have entered tournaments. To get a set of all member IDs, we need to query the Member table.

Figure 7-22 is an illustration of how the difference operator can be used to find the member IDs we require.

MemberID
118
138
153

MemberID
118
138
153



MemberID
118
138
153
176
178
228
235
239
258
286
290
323
331
332
339
414
415
461
469
487

MemberID
228
239
415

MemberID
118
138
153
176
178
235
258
286
290
323
331
332
339
414
461
469
487

Figure 7-22. Members who have not entered tournament 25

The SQL expression to retrieve the IDs of members who have not entered tournament 25 is as follows:

```
SELECT MemberID FROM Member
EXCEPT
SELECT MemberID FROM Entry WHERE TourID = 25;
```

As with intersection and union operations, it is important that we project the appropriate columns before we use the difference operator. In Figure 7-22, we have retrieved the IDs from the `Member` and `Entry` tables. If we want to include the names of the members, we can use one of the methods explained in the "Uses of Intersection" section earlier in this chapter.

However, in this difference example, we already had the names of the members in the `Member` table before we removed them to get the set of rows on the left side of Figure 7-22. It seems a bit perverse to remove the names and then put them back later. What is important is that the two sets of rows involved in the difference are union compatible; that is, the corresponding columns must have the same domains. Either both sets have just IDs or both sets have IDs and names. In the operation on the left side of Figure 7-22, we took the first option and removed the names from `Member`. We could have left the names in the `Member` table and added the names to the rows in the middle of Figure 7-22 by joining the `Entry` and `Member` tables, as shown in Figure 7-23. We could then take the difference between these two sets of rows.

MemberID	LastName	FirstName
118	McKenzie	Melissa
138	Stone	Michael
153	Nolan	Brenda

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

IDs and Names from Member

m.Member	LastName	FirstName	e.MemberI	TourID	Year
239	Spence	Thomas	239	25	2015
228	Burton	Sandra	228	25	2015
415	Taylor	William	415	25	2013

B

Entry table joined with Member table
Then rows for tournament 25 selected
Then IDs and Names projected

Figure 7-23. Including names of members in both sets of rows before taking the difference

The SQL equivalent of the operations shown in Figure 7-23 is as follows:

```
SELECT MemberID, LastName, FirstName FROM Member
EXCEPT
SELECT m.MemberID, m.LastName, m.FirstName
FROM Entry e inner join Member m on e.MemberID = m.MemberID
WHERE TourID = 25;
```

Managing Without the EXCEPT Keyword

Not all versions of SQL support the EXCEPT (or MINUS) keyword. As always, there is usually another way to formulate a query. In Chapter 4, we looked at an outcome approach to answering questions involving the word *not*. Figure 7-24 reviews the thought processes used to find the names of members who have not entered tournament 25.

Figure 7-24. Deciding that member 258 has not entered tournament 25

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
m ☞	258	Olson	Barbara
	286	Pollard	Robert
	290	Sexton	Thomas
	323	Wilcox	Daniel
	331	Schmidt	Thomas
	332	Bridges	Deborah
	339	Young	Betty
	414	Gilmore	Jane

Entry

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

The thought process behind Figure 7-24 is:

> *Write out the names from row* `m` *of the* `Member` *table if there does not exist a row* `e` *in the* `Entry` *table for that member (i.e.,* `m.MemberID = e.MemberID`*) where* `TourID=25`*.*

The SQL reflecting Figure 7-24 is:

```
SELECT m.LastName, m.FirstName
FROM Member m
WHERE NOT EXISTS
    (SELECT * FROM Entry e
     WHERE e.MemberID = m.MemberID
     AND e.TourID = 25);
```

Which type of query should you use for questions involving the word *not*? The one using the process approach and the keyword `EXCEPT` or the one using the outcome approach with the keywords `NOT EXISTS` or `NOT IN`? Usually, I'd say it doesn't really matter, as your database engine will probably be smart enough to recognize them as being the same. However, the version of SQL Server I am using at the moment (2013) performs the query using `NOT EXISTS` more efficiently than the corresponding query using `EXCEPT`. You have to ask yourself whether you care! Queries on small databases are usually so quick that it really doesn't matter if they run a bit more slowly. However, if you have a lot of data, then everything changes. The efficiency of queries can become extremely important, and in that case, you will need to also consider other aspects of your database design, such as indexes. I'll talk a little more about this in Chapter 9.

Division

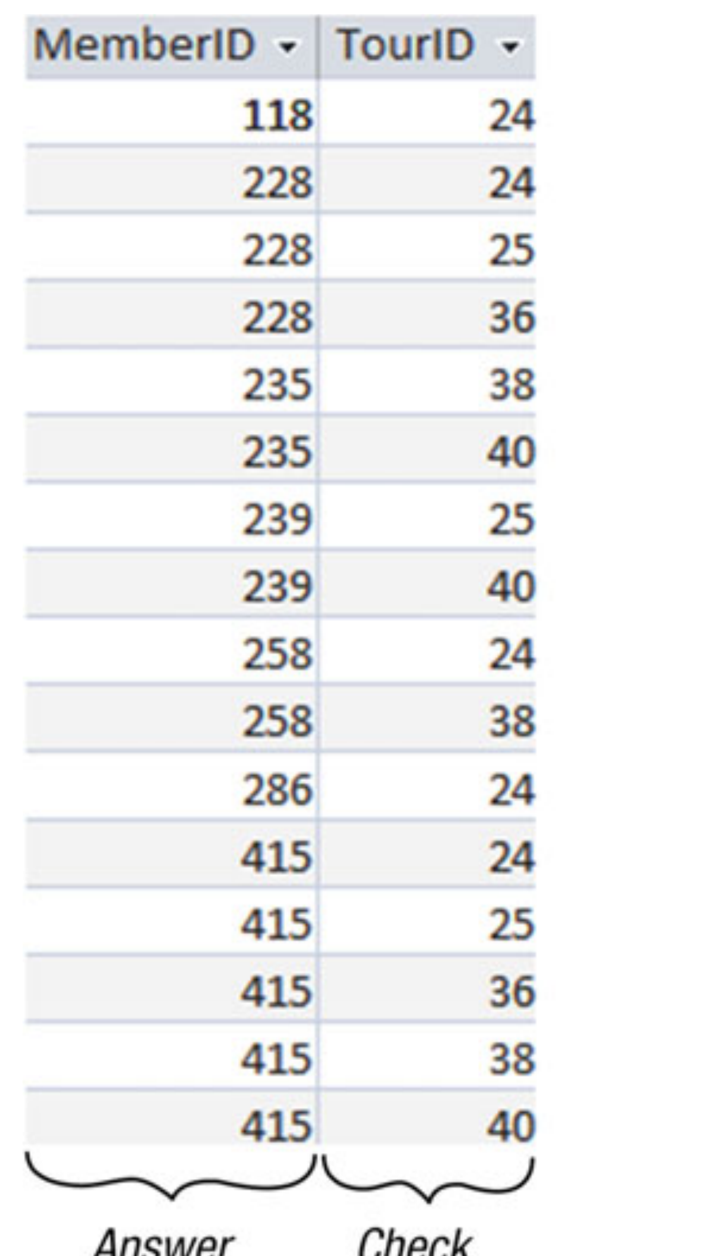
The last set operator we will look at in this chapter is division. Division is useful for queries that involve the word *all* or *every*. An example is "Which members have entered *every* tournament?" Standard SQL doesn't have a keyword for the divide operation, and it can be a little awkward to figure out the SQL for queries involving division.

In Appendix 2 you will find the formal algebraic notation for carrying out division and how to represent it using other operators if you need to. In the section "Universal Quantifier and SQL" in Appendix 2, you will also find an alternative way to carry out division-type queries using calculus (or outcome) expressions. Both these methods help you to construct SQL statements that are analogous to the division operator. In Chapter 8, we'll look at aggregates and see what I think is the simplest way of writing an SQL equivalent of the division operator.

For now we will look at what the division operator does and how to use it to answer different types of questions involving *every* and *all*.

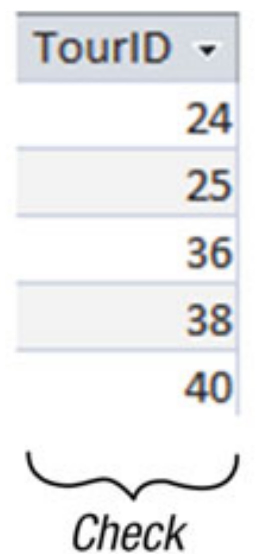
The easiest way to understand the division operation is with an example. If we want to know which members have entered every tournament, we need two bits of information. First, we need information about the members and the tournaments they have entered, which we can get from the `Entry` table. We also need a list of all the tournaments, which needs to come from the `Tournament` table, as not all tournaments may be represented in the `Entry` table.

Figure 7-25 illustrates how division works. I've projected just the `MemberID` and `TourID` columns from the `Entry` table, and the `TourID` column from the `Tournament` table. It is important which columns you project, and I'll come back to that in a moment.



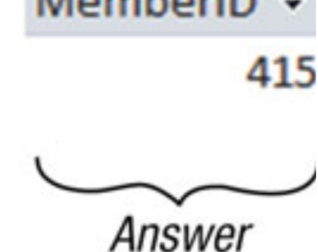
MemberID	TourID
118	24
228	24
228	25
228	36
235	38
235	40
239	25
239	40
258	24
258	38
286	24
415	24
415	25
415	36
415	38
415	40

÷



TourID
24
25
36
38
40

=



MemberID
415

SELECT MemberID, TourID
FROM Entry

SELECT TourID
FROM Tour

Result of
Devision

Figure 7-25. Using division to find members who have entered all tournaments

SELECT MemberID, TourID
FROM Entry

SELECT TourID
FROM Tour

Result of
Devision

Looking at Figure 7-25, we have in the middle a table with all the TourID values (I've labelled that *Check*). The division operation checks the left-hand table to find the values of MemberID that have a row for every TourID. The *Answer* (on the right of the figure) contains the MemberID values for members who have entered every tournament. Member 415 can be found paired with each of the five tournaments in the Entry table, and so appears in the result of the division. Member 228 does not appear in the result because there are no rows in the Entry table with 228 paired with 38 or 40.

It is important to get the correct columns in the two tables involved in the division. I like to think of setting up the division operation like this:

- Decide which attribute I want to find out about. Let's call this *Answer*. In this case, I want to find values of MemberID, so our *Answer* attribute is MemberID.
- On the right-hand side of the division operator, the attribute(s) in the table should be the thing I want to check against. Let's call this attribute(s) *Check*. In this case, the *Check* attribute is TourID. We can get all the values for TourID from the Tournament table.

- On the left-hand side of the division, I want a table containing the just the two sets of attributes *Answer* and *Check*, as shown in Figure 7-25. We need `MemberID` and `TourID` (in this case, which members have entered which tournament, and these come from the `Entry` table). It is important that these are the only two columns in the left-hand table. If extra columns are added, then we will be asking different questions, as explained in the next section.

As a small aside, many people wonder why this operation is called *division*, as it doesn't seem to relate particularly well to something like 4 divided by 2. Division is the inverse (or undoing) of multiplication in normal arithmetic. For set operations, division is like the inverse of the Cartesian product. If you think of taking the Cartesian product of the two tables in the middle and far right of Figure 7-25, you will get a table with the same columns (but not rows) as on the far left of Figure 7-25.

We can answer a number of questions by changing what is on the right-hand side of the division operator. For example, if we wanted to know who had entered all the Open tournaments, we would replace the table in the middle of Figure 7-25 with just the rows for Open tournaments:

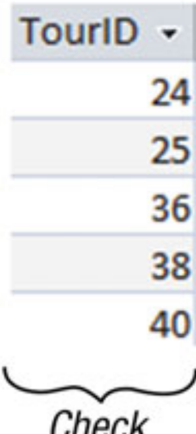
```
SELECT TourID
FROM Tour
WHERE TourType = 'Open';
```

Projecting Appropriate Columns

As with intersection and difference operations, projecting different columns in division operations will give you answers to different questions. Once again, an example is the easiest way to understand this. In Figure 7-26, an extra column has been retrieved from the `Entry` table. Can you understand what this query is finding?

MemberID	Year	TourID
118	2014	24
228	2015	24
228	2015	25
228	2015	36
235	2013	38
235	2015	38
235	2014	40
235	2015	40
239	2015	25
239	2013	40
258	2014	24
258	2014	38
286	2013	24
286	2014	24
286	2015	24
415	2015	24
415	2013	25
415	2014	36
415	2015	36
415	2013	38
415	2015	38
415	2013	40
415	2014	40
415	2015	40
415	2020	40

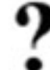
Answer *Check*



TourID
24
25
36
38
40

Check

=



Answer

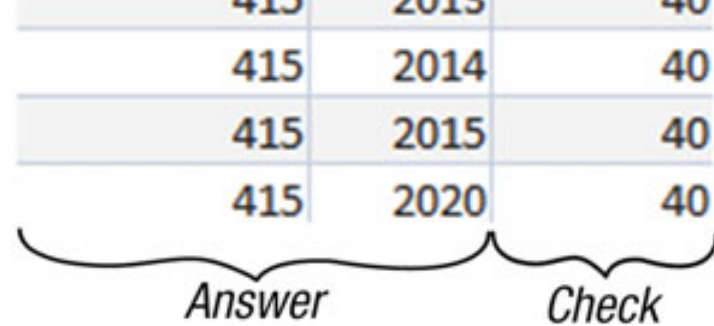


Figure 7-26. *What is the division operation finding?*

The division is looking for a set of *Answer* attributes in the left-hand table that are paired with every attribute from the `Check` table. In this case, the operation looks for a pair `MemberID` and `Year` in the left-hand table that appears with each of the tournaments. This division example is finding those members who have entered all tournaments in the same year.

SQL for Division

Using an output approach, the query we want can be expressed something like this:

> *Write out the value of `m.LastName, m.FirstName` from rows `m` in the `Member` table where for every row `t` in the `Tournament` table there exists a row `e` in the `Entry` table with `e.MemberID = m.MemberID` and `e.TourID = t.TourID`.*

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

The double negatives can be a bit daunting, but as I said at the beginning of the chapter, I promise a conceptually easier method to find members who have entered every tournament in the next chapter.

Summary

Because tables in a relational database have unique rows (if they are properly keyed!), they can be treated like mathematical sets. This allows us to use the set operations union, intersection, difference, and division.

Union, intersection, and difference are operations that act between union-compatible tables. This means the table on each side of the operator must have the same number of columns, and the columns must have the same domains (commonly interpreted as the same types). You can get union-compatible tables by sensibly projecting columns.

SQL has keywords to represent union, intersection, and difference, although not every implementation supports the keywords for all of these operations. If your SQL product does not support keywords for intersection or difference, you can find other ways to express the query. You should formulate your queries in the way you find most natural. Where you have very large amounts of data and speed is important, you may need to investigate the efficiencies of the different ways of formulating some queries.

Here is a summary of the set operations and alternative ways to represent them with SQL. A and B are two union-compatible tables with (for simplicity) just one column called attribute.

Union

A union operation returns all the unique rows that are in either table A or table B:

```
SELECT attribute FROM A
UNION
SELECT attribute FROM B;
```

Intersection

An intersection operation returns all rows that are in both table A and table B:

```
SELECT attribute FROM A
INTERSECT
SELECT attribute FROM B;
```

An alternative way to represent intersection is:

```
SELECT A.attribute
FROM A
WHERE EXISTS
    (SELECT B.attribute FROM B
     WHERE A.attribute = B.attribute);
```

Difference

Difference returns all rows that are in the first table (A) that are not in the second table (B). Some implementations use the keyword MINUS instead of EXCEPT:

```
SELECT attribute FROM A
EXCEPT
SELECT attribute FROM B;
```

An alternative way to represent difference is:

```
SELECT A.attribute
FROM A
WHERE NOT EXISTS
   (SELECT B.attribute FROM B
   WHERE A.attribute = B.attribute);
```

Division

The division operation helps with queries with the words *every* or *all*. Current versions of SQL do not support division directly. Refer to the sections "Division" and "Universal Quantifier and SQL" in Appendix 2 for details of how to express queries involving division.

For completeness, we repeat the following query, which returns the `MemberID` values for those members who have entered every tournament:

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