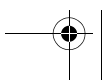
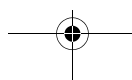
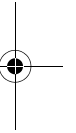
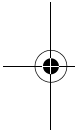


CHAPTER FIVE



Join Solutions

Sometimes joins just don't cooperate with what you as a query writer want to accomplish. When this happens, you need a couple of remedies that will solve the problem. This chapter discusses two common join problems and the solutions for each.



Join Errors

Sometimes the multiplicative effect of joins creates problems. This can occur when the population for a query is already defined, and you're joining to additional tables to display data from those tables in the report. That is, the Who is already defined, and you're adding the Where joins.

Let's consider two tables called A and B. For a single row in A, only three situations can occur:

- One row in A will match with one row in B. This is a one-to-one relationship. The join of A and B produces one result row.
- One row in A will match with many rows in B. This is a one-to-many relationship. The join of A and B produces many result rows.
- One row in A will match with no rows in B. This is a one-to-none relationship. The join of A and B produces no result rows.

Figure 5-1 illustrates these situations with NAME and ADDRESS data for three people. Each person has one name. The first person has one address, the second person has no addresses, and the third person has three addresses. Respectively, these represent one-to-one, one-to-none, and one-to-many relationships between the NAME and ADDRESS tables.

id	name	addr	seq	city	state
@635466	Keller, Linda	PR	1	Decatur	GA
@387261	Laube, Horace				
@803320	Waggoner, James	PR	1	Norfolk	VA
		PR	2	Cookeville	TN
		PR	3	Laredo	TX

FIGURE 5-1 One-to-one, one-to-none, and one-to-many relationships.

A one-to-one relationship never presents a problem in a query. But each of the other two relationships will introduce an error into the query if the population has already been set. A one-to-many relationship causes the population item to multiply through a join. Instead of a single row, the report will include many rows for that population item. A one-to-none relationship causes a population item to fall through the join. That population item will not appear in the report.

Suppose our query population is the three people in Figure 5-1. This means that the report should show three lines, one for each person. But the join of NAME and ADDRESS produces four rows. There will be one row for Linda Keller, no rows for Horace Laube, and three rows for James Waggoner. Both types of join errors occur in this example. Horace Laube falls through the join, and James Waggoner multiplies through the join. Figure 5-2 demonstrates this situation.

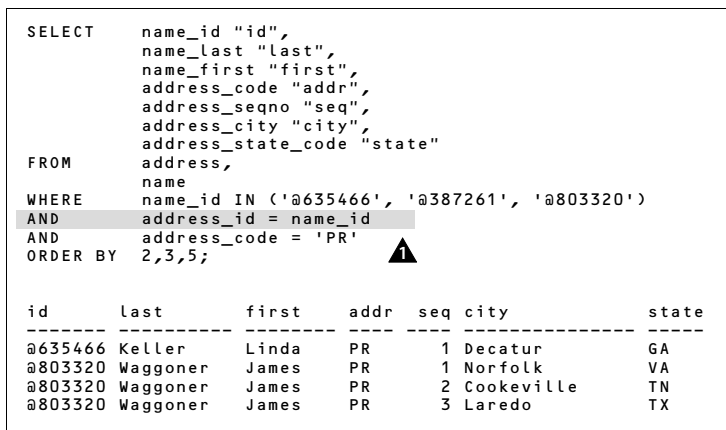


FIGURE 5-2 Join errors with a population of three people.

- ▶ The equijoin between NAME and ADDRESS causes Horace Laube to disappear from the report. It also causes James Waggoner to appear three times. Both represent join errors if the population of the query is people and each person should appear once. Note the IN operator used to list identification numbers. See comparison operators on 260 in Appendix C for more details.

■ Exercises

1. What type of data relationship causes a population item to fall through a join? To multiply through a join?
2. If a population item falls through a join, how does this affect the report?

3. If a population item multiplies through a join, what consequence does this have for the report?
4. Figure 5-3 shows a population of courses from Komenda’s course catalog. Figure 5-4 shows data from the SCHEDULE table for Fall 1999. If the SCHEDULE table is joined to the course population, which courses will fall through the join? Which courses will multiply through the join?

catalog_ code	catalog_ effective_ term	catalog_ subject_ code	catalog_ course_ number	catalog_ description
09388	199509	SPA	156	Cervantes: Other Works
26461	199509	GOV	138	Comparative Study of Civil-Military Relations
37893	199509	AAS	222	The Literature of Possession: Seminar

FIGURE 5-3 A population of courses from the CATALOG table.

schedule_ term	schedule_ catalog_ code	schedule_ section_ number	schedule_ meeting_ days	schedule_ building_ code	schedule_ room_ code	schedule_ start_ time
199909	09388	01	Mon,Wed,Fri	EDG	187	14:00
199909	26461	01	Mon,Wed,Fri	RAU	154	12:00
199909	26461	02	Mon,Wed,Fri	RAU	152	16:00

FIGURE 5-4 SCHEDULE data for the population of courses.

Falling Through a Join

When a row in the population does not have a corresponding row in a table being joined, that row falls through the join, and a member of the population disappears from the report. Not good. The solution is to use a different kind of join than the equijoin used up to this point. The join is called an *outer join*. Outer joins prevent population items from falling through joins.

Here’s how they work. Let’s consider the situation of Horace Laube in Figure 5-1. This person has no known permanent address. In an equijoin between ADDRESS and the population definition, he falls through the join and disappears from the report. When an outer join is made to the ADDRESS table, an empty run-time-only ADDRESS row gets created for everyone like Horace who does not have a permanent address.

What does this mean—*empty run-time-only*? Well, *empty* means that the

data values in the created row are all null; *run-time-only* means that the row only gets created temporarily for the duration of the query that you're running. So Horace stays in the report, but his address information is all null (which is exactly what we want). And after the query stops executing, the empty row created for Horace disappears.

Figure 5-5 shows an outer join to the ADDRESS table. Note that you can distinguish outer joins from equijoins by the (+) sign on the join criteria.

```

SELECT      name_id "id",
            name_last "last",
            name_first "first",
            address_code "addr",
            address_seqno "seq",
            address_city "city",
            address_state_code "state"
FROM        address a1,
            name
WHERE       name_id IN ('@635466', '@387261', '@803320')
AND        address_id(+) = name_id
AND        address_code(+) = 'PR'
ORDER BY   2,3,5;

```

id	last	first	addr	seq	city	state
@635466	Keller	Linda	PR	1	Decatur	GA
@387261	Laube	Horace				
@803320	Waggoner	James	PR	1	Norfolk	VA
@803320	Waggoner	James	PR	2	Cookeville	TN
@803320	Waggoner	James	PR	3	Laredo	TX

FIGURE 5-5 An outer join used to prevent population loss.

- ▶ Outer joins prevent items in a population from falling through a join. Here the outer join prevents people from falling through the ADDRESS join if they have no permanent address (see text).
- ▶ Note that Horace Laube now appears in the report even though he has no permanent address. The data, of course, are all null.

There are also two things to notice about outer joins:

- You can put the outer join sign on either side of the join condition, but they mean distinctly different things. For example, `address_id(+) = name_id` is different from `address_id = name_id(+)`. The first join says this: For every `name_id` without a corresponding `address_id`, create an empty run-time-only ADDRESS row. The second join is just the reverse. It says: For every `address_id` without a corresponding `name_id`, create an empty run-time-only NAME row. Confusing? Yes,

especially when you must decide where to put the (+) sign. Here's how I decide. It goes on the data column that "needs help," where the data are missing. If population items fall through the `ADDRESS` table, the outer join belongs on `ADDRESS` columns and not on `NAME` columns.

- When you use an outer join symbol, you must use it on all criteria for the table being joined. In Figure 5-5 the outer join symbol appears on two lines. One line does an outer join between `address_id` and `name_id`; a second line uses (+) with `address_code` and `PR`. Here's why this second line is necessary. The line `address_id(+) = name_id` creates an empty row for Horace Laube if he has no `ADDRESS` data. But if the next line of SQL said `address_code = 'PR'`, Horace would again fall through the report because his `address_code` is null. That's why the outer join symbol must appear on both lines in Figure 5-5.

■ Exercises

1. Examine the data dictionaries for the `SCHEDULE` and `CATALOG` tables in Appendix A (see pages 246 and 231, respectively). Also examine the constraint listings for these tables to identify their primary keys. Write the standard equijoin between these two tables.
2. Write a join that prevents `CATALOG` courses in Figure 5-3 from falling through a join with the `SCHEDULE` table (see Figure 5-4).

Multiplying Through a Join

James Waggoner multiplies through the `NAME` and `ADDRESS` join because he has three permanent addresses. We want only one row. Here's the critical leap in thought needed to solve this problem—one row ... unique row ... uniqueness ... primary key. If we could somehow specify each of the data columns in the primary key, we'd be guaranteed of returning only one row.

The primary key in the `ADDRESS` table is `address_id`, `address_code`, and `address_seqno` (see Figure 4-4 on page 50). If we can provide the SQL query with specific values for each of these three data columns for

Waggoner, then we'll get one row. For one person, we could hardcode the values, but we need a general solution that will work for anyone.

Suppose that we want the report to show most recent permanent address. Recall from Chapter 4 that this is the one row with the maximum sequence number (i.e., `address_seqno`) for the PR address type. This would solve the multiplicative problem created by the join. Here's the solution in words: "Join the **NAME** and **ADDRESS** tables through the identification number. For each identification number, only return **ADDRESS** data where the address type is PR and the address sequence number is maximum."

We can do this with a wonderful SQL structure called a *correlated subquery*. A subquery is just what the name implies. It's a query within the main query. What it means to be correlated will become evident in a moment.

Figure 5-6 shows the SQL needed to return James' most recent permanent address. Clearly, it works, but what's happening in this query?

The query starts with one of the three people. It gets a value for `address_id` from the **NAME** join (i.e., `WHERE address_id = name_id`). It gets a value for `address_code` from the direct specification (i.e., `AND address_code = 'PR'`). And it gets a value for `address_seqno` from the correlated subquery. The main query now knows specific values for each of the three columns in the **ADDRESS** primary key, and it uses these to return one row, the row with the most recent permanent address. Then the main query goes to person two, and the process begins again.

The subquery works by looping through the **ADDRESS** table and returning to the main query the maximum `address_seqno` for the same person and address type being considered in the main query. The subquery is a *correlated* subquery because it only examines data for the same person and address type being considered in the main query.

■ Exercises

1. Figure 4-1 on page 47 shows two rows of **NAME** data for one person (Margaret). The primary key in the **NAME** table is `name_id` and `name_seqno` (see Figure 4-2 on page 49). Write SQL that uses a correlated subquery to return the current name for Margaret.

```

SELECT  name_id "id",
        name_last "last",
        name_first "first",
        address_code "addr",
        address_seqno "seq",
        address_city "city",
        address_state_code "state"
FROM    address a1,
        name
WHERE   name_id = '@803320'
AND     address_id = name_id
AND     address_code = 'PR'
AND     address_seqno =
        (SELECT  MAX(a2.address_seqno)
         FROM    address a2
         WHERE   a2.address_id = name_id
         AND     a2.address_code = a1.address_code);

```

id	last	first	addr	seq	city	state
@803320	Waggoner	James	PR	3	Laredo	TX

FIGURE 5-6 A correlated subquery used to return most recent address.

- ▶ These three lines specify values for each of the three data columns in the primary key of the ADDRESS table. By doing so, we're guaranteed that only a single row will be returned (if the data exist at all).
- ▶ This is a correlated subquery used to return the maximum address_seqno. It is linked, or correlated, back to the main query through the two lines WHERE a2.address_id = name_id AND a2.address_code = a1.address_code. This ensures that the subquery only checks the data for the same person and same address type being considered in the main query. Note the use of the MAX group function to return the maximum. See Appendix D on 303 for more details. Also note the aliases given to the ADDRESS tables. Because both the main query and the subquery use the ADDRESS table, we need a way to distinguish between the two uses. The alias in the main query is a1, and the alias in the subquery is a2. You're free to use any aliases as long as the two differ.
- ▶ Note the result. Only the most recent permanent address is returned.

Simultaneous Join Problems

You might think that we've now got the tools to solve all join errors that occur in Figure 5-2. One of the three people fell through the **NAME** and **ADDRESS** join. An outer join solved that problem. A second person multiplied through the join. A correlated subquery provided that solution. But guess again.

What happens when we combine an outer join with a correlated subquery? Figure 5-7 shows the result. The subquery seems to work fine. It returns only one address for James Waggoner. But poor Horace Laube again fell through the join, and it's the subquery that did him in. He doesn't have a permanent address, so the subquery returns nothing to the main query when the main query expects an actual value for **address_seqno**.

Maybe we could just put an outer join on **address_seqno** (i.e., **AND address_seqno(+) = subquery**). It's a good thought, but unfortunately, you get Oracle error **ORA-01799: A column may not be outer-joined to a subquery**.

```

SELECT  name_id "id",
        name_last "last",
        name_first "first",
        address_code "addr",
        address_seqno "seq",
        address_city "city",
        address_state_code "state"
FROM    address a1,
        name
WHERE   name_id IN ('@635466', '@387261', '@803320')
AND    address_id(+) = name_id
AND    address_code(+) = 'PR'
AND    address_seqno =
        (SELECT  MAX(a2.address_seqno)
         FROM    address a2
         WHERE   a2.address_id = a1.address_id
         AND    a2.address_code = a1.address_code);

```

id	last	first	addr	seq	city	state
@635466	Keller	Linda	PR	1	Decatur	GA
@803320	Waggoner	James	PR	3	Laredo	TX

FIGURE 5-7 Outer join and correlated subquery incorrectly combined.

- 1 The SQL uses an outer join.
- 2 It also uses a correlated subquery.
- 3 But one person still falls through the SQL.

Figure 5-8 shows one solution to simultaneous join errors. It's only a slight variation of the SQL in Figure 5-7. Here is what happens in this query. The `OR` condition sets up two ways that the main query can select someone for the report. If a person has a permanent address, the subquery

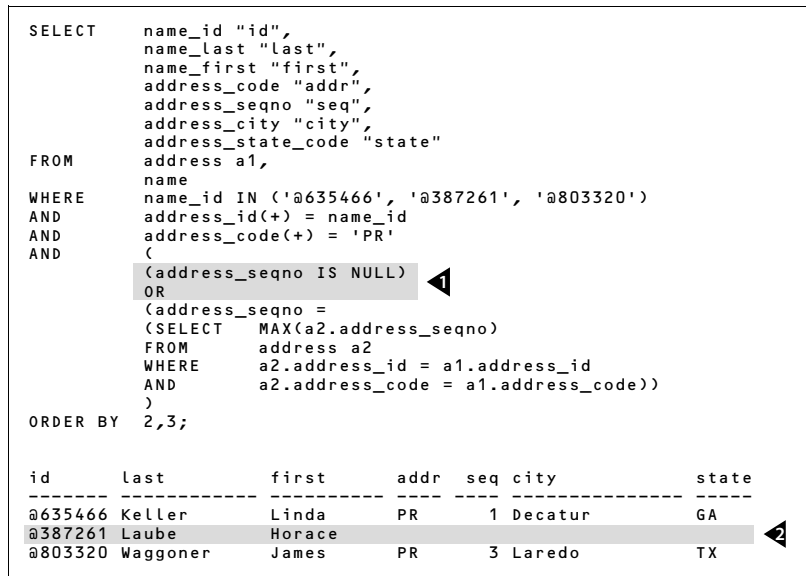
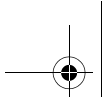


FIGURE 5-8 Outer join and correlated subquery correctly combined.

- ▶ Aside from the parentheses, the only differences between the SQL here and the SQL in Figure 5-7 occurs in these two lines. The `OR` operator sets up two ways for rows to be retained by the main query. The first way is through the subquery. As shown in Figure 5-7, this works well for Keller and Waggoner, who have permanent addresses. The second way a row can appear in the report is if `address_seqno` is null. Since this data column is part of the primary key of the `ADDRESS` table, it can never be null unless it was created by the outer join. It's this line that ensures that Laube appears in the report. For more information on the `IS NULL` operator, see comparison operators in Appendix C on 260.
- ▶ Note the results. Horace Laube is included.



will return the maximum `address_seqno`, and the main query will use it to include the most recent permanent address in the report. If a person does not have a permanent address, the outer joins create an empty row, and that row gets retained for the report by meeting the condition that `address_seqno IS NULL`.

Now you're set. If a query suffers only from population items that fall through a join, use an outer join as the remedy. If a query suffers only from population items that multiply through a join, use a correlated subquery as the remedy. If you're unfortunate enough to get a query that suffers from both join errors simultaneously, then use an outer join and a correlated subquery in combination—but be aware that `OR` logic will be required to provide the desired results.

