CHAPTER EIGHT

Where

After the population gets defined in the Who stage of query construction, your job becomes one of maintaining the integrity of that population as you join additional tables required for the report. No population item should fall through any of these joins, nor should a population item multiply through a join. This chapter discusses how to step through Where stage joins one at a time and to test rigorously for join errors.

ch8.fm Page 98 Thursday, January 25, 2001 6:49 AM

Reprise

Let's briefly recall the steps needed to construct an SQL query so as to orient ourselves in this process. Each query consists of several steps:

- **Step 1** (Who): Identify the population, and describe it in words.
- **Step 2** (Who): Identify the tables needed to define the population and select the one table most central to the population as the driver of the query.
- **Step 3** (Who): Translate your verbal description of the population into SQL.
- **Step 4** (Who): Obtain counts of the population to serve as a baseline for testing the remainder of the query.
- **Step 5** (Where): Identify additional tables needed to display data for the report.
- **Step 6** (Where): Add the join for one of these tables to the SQL that defines the population.
- **Step 7** (Where): Test for the two types of join errors that may occur. Fix the SQL if necessary.
- **Step 8** (Where): Repeat the last two steps for each additional table needed to access data for the report.
- **Step 9** (What): Replace the counts in the `SELECT` clause with data columns, expressions, and group summaries that should appear in the report. Also add grouping and sorting criteria to the SQL.
- **Step 10**: Add SQL*Plus commands to format the report.

This chapter discusses the Where steps in query construction, dealing with steps 5 through 8 that join tables needed for the report but not part of the population definition. Since each join represents an opportunity to introduce errors into the query, we'll pay particular attention to identifying and fixing such errors.

Stepping Through Joins

When constructing the Where joins, start by adding one table to the `FROM` clause and its standard equijoin in the `WHERE` clause. Obtain report and population counts, and compare these against the baseline counts. If the counts differ, the join has introduced an error into the SQL. Apply one of the join solutions discussed in Chapter 5 to fix the error. When the two counts agree with the baseline counts, proceed to the next table.

This process is called *stepping through the joins*. It forces you to check for join errors as you develop the query. In addition, it modularizes the query so that you can write SQL in small chunks rather than in one massive step.

Figure 8-1 shows the class roster for a course called Transforming Managerial Leadership that was taught in Fall 1998. The report shows current name and current permanent addresses. It also includes people without any permanent address.

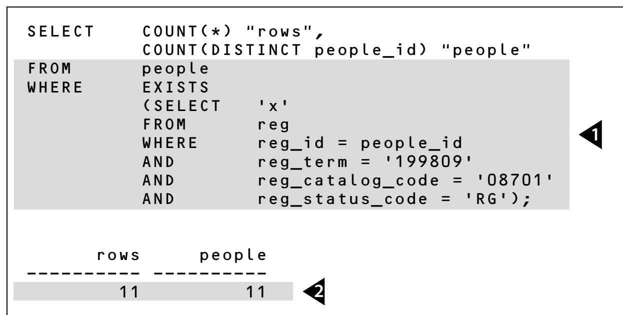
```
                                    name       addr
id      last         first           seq addr  seq city            state
------- ------------ ---------- ---- ---- ---- --------------- -----
@667501 Bailes       Lulane            1 PR      1 Coeur D Alene   ID
@061629 Haines       Merry             1 PR      1 Eufaula         AL
@243432 Hiatt        Wilfred           1 PR      1 Flint           MI
@093560 Letson       Debra             1 PR      2 San Diego       CA
@625070 Mitchell     Mary              2 PR      1 Mitchell        SD
@427616 Neveu        Nicoletta         1 PR      1 Hato Rey        PR
@122954 Pinkerton    David             1 PR      1 St Joseph       MN
@381366 Schultz      Christine         2 PR      1 Grand Island    NE
@153626 Simpson      Martin            1
@200197 Tardiff      Jean              1 PR      1 Kosciusko       MS
@440571 Virtucio     Daniel            1 PR      1 Great Falls     MT
```

FIGURE 8-1 Class roster report.

1 Note that the report includes two people who have multiple names (@625070 and @381366), one person who has multiple permanent addresses (@093560), and one person who has no permanent address (@153626). Yet the report displays only the most recent name and most recent permanent address. It also displays null data for anyone without a permanent address.

Figure 8-2 defines the query population and provides baseline counts. Because the report does not require `REG` data, a correlated subquery used as a modifier works well when defining the population. Note the baseline counts.

ch8.fm Page 100 Thursday, January 25, 2001 6:49 AM



```
SELECT    COUNT(*) "rows",
          COUNT(DISTINCT people_id) "people"
FROM      people
WHERE     EXISTS
          (SELECT   'x'
          FROM      reg
          WHERE     reg_id = people_id
          AND       reg_term = '199809'
          AND       reg_catalog_code = '08701'
          AND       reg_status_code = 'RG');

      rows     people
---------- ----------
        11         11
```

FIGURE 8-2 Population definition and baseline counts for class roster.

1. The population consists of people who are registered in Fall 1998 for the course with catalog code 08701 and with a registration status code of RG. Since the report does not need to include REG data, the population can be defined with the EXISTS operator and a correlated subquery that merely checks for the appropriate registration.
2. The report should include 11 rows, one for each person in the population.

Our job is to step through the additional joins needed for the report. This means completing steps 5 through 8 in the query process.

- **Step 5**: Identify additional tables needed for the report.
  The population definition only provides access to PEOPLE data in the main query. To include names and permanent addresses, we'll need to add joins to the NAME and ADDRESS tables.

- **Steps 6 and 7**: Add the join for one table, and test for join errors.
  Let's start with the NAME table. Add the table to the FROM clause, and then write the standard equijoin between NAME and PEOPLE in the WHERE clause. Figure 8-3 shows the results.

  As you probably anticipated from the report in Figure 8-1, the population multiplies through the standard equijoin. But we know how to solve this problem—use a correlated subquery to force singularity again. We can do this by returning the most recent name for each person. Figure 8-4 shows the result. The counts again match the baseline.

```
SELECT    COUNT(*) "rows",
          COUNT(DISTINCT people_id) "people"
FROM      name,
          people
WHERE     EXISTS        1
          (SELECT   'x'
          FROM      reg
          WHERE     reg_id = people_id
          AND       reg_term = '199809'
          AND       reg_catalog_code = '08701'
          AND       reg_status_code = 'RG')
AND       name_id = people_id;
                                  2
      rows     people
---------- ----------
        13         11  3
```

FIGURE 8-3 Stepping through the NAME *join in the class roster query.*

1. Add the NAME table to the FROM clause.
2. Then join the NAME and PEOPLE tables through data columns common to their primary keys.
3. Note the population gain that occurred with this join.

```
SELECT    COUNT(*) "rows",
          COUNT(DISTINCT people_id) "people"
FROM      name n1,
          people
WHERE     EXISTS
          (SELECT   'x'
          FROM      reg
          WHERE     reg_id = people_id
          AND       reg_term = '199809'
          AND       reg_catalog_code = '08701'
          AND       reg_status_code = 'RG')
AND       name_id = people_id
AND       name_seqno =
          (SELECT   MAX(n2.name_seqno)
          FROM      name n2                      1
          WHERE     n2.name_id = people_id);


      rows     people
---------- ----------
        11         11  2
```

FIGURE 8-4 Preventing population gain in the class roster NAME *join.*

1. Use a correlated subquery to return the most recent name and ensure one row per person.
2. Note that the counts are again the same as the baseline counts.

ch8.fm Page 102 Thursday, January 25, 2001 6:49 AM

- **Step 8**: Add the join for the second table, and test for join errors.

Now let's add the ADDRESS join. From the report in Figure 8-1, we can anticipate both population loss and gain with this join. Figure 8-5 shows that this is the case.

```
SELECT    COUNT(*) "rows",
          COUNT(DISTINCT people_id) "people"
FROM      address,
          name n1,
          people        1
WHERE     EXISTS
          (SELECT   'x'
          FROM      reg
          WHERE     reg_id = people_id
          AND       reg_term = '199809'
          AND       reg_catalog_code = '08701'
          AND       reg_status_code = 'RG')
AND       name_id = people_id
AND       name_seqno =
          (SELECT   MAX(n2.name_seqno)
          FROM      name n2
          WHERE     n2.name_id = people_id)
AND       address_id = people_id
AND       address_code = 'PR';
                             2

      rows     people
---------- ----------
        11         10   3
```

FIGURE 8-5 Stepping through the ADDRESS *join in the class roster query.*

1. Add the ADDRESS table to the FROM clause.
2. Then join the ADDRESS and PEOPLE tables using data columns common to their primary keys. Include the criterion that address_code = 'PR' to restrict addresses to permanent addresses.
3. Note that one person fell through the join. Another person multiplied through the join. Both join errors occur here.

Again, we go to our join solutions. When a join suffers from both population loss and population gain, use an outer join for the population loss, a correlated subquery for the population gain, and OR logic to ensure that the two solutions work together properly. See Figure 5-8 on page 70 for the details. Figure 8-6 shows the result.

This SQL looks like a mess. Yet, taken as a series of steps, first defining the population and then adding Where joins one at a time, the SQL seems quite manageable. It's also thoroughly tested.

ch8.fm Page 103 Thursday, January 25, 2001 6:49 AM


```
SELECT    COUNT(*) "rows",
          COUNT(DISTINCT people_id) "people"
FROM      address a1,
          name n1,
          people
WHERE     EXISTS
          (SELECT   'x'
          FROM      reg
          WHERE     reg_id = people_id
          AND       reg_term = '199809'
          AND       reg_catalog_code = '08701'
          AND       reg_status_code = 'RG')
AND       name_id = people_id
AND       name_seqno =
          (SELECT   MAX(n2.name_seqno)
          FROM      name n2
          WHERE     n2.name_id = people_id)
AND       address_id(+) = people_id              ◀1
AND       address_code(+) = 'PR'
AND       (
          (address_seqno IS NULL)                ◀2
          OR
          (address_seqno =
          (SELECT   MAX(a2.address_seqno)
          FROM      address a2                                     ◀3
          WHERE     a2.address_id = people_id
          AND       a2.address_code = a1.address_code))
          );


      rows     people
---------- ----------
        11         11  ◀4
```

FIGURE 8-6 Preventing errors in the class roster ADDRESS *join.*

1. Use an outer join to prevent population loss.
2. Use this line and `OR` logic to ensure that null rows created by the outer join do not fall through the correlated subquery.
3. Use a correlated subquery to return the most recent permanent address for people having one or more.
4. Note that the counts again agree with the baseline counts.

■ Exercises

1. Figure 8-7 shows the population definition and baseline counts of registrations by three people in a Fall 1998 course called Transforming Managerial Leadership (i.e., the catalog code is 08701). Figure 8-8 shows the grades received by these three people in that course. Review the data

ch8.fm Page 104 Thursday, January 25, 2001 6:49 AM

dictionaries and constraint listings for the REG and GRADE tables in Appendix A (pages 242 and 235, respectively). Then write the GRADE join that will return the current grade if someone has more than one and that also will prevent population loss if someone has no grade.

```
SELECT    COUNT(*) "rows",
          COUNT(DISTINCT reg_id||reg_term
              ||reg_catalog_code||reg_section_number) "registrations"
FROM      reg
WHERE     reg_id IN ('@200197', '@093560', '@625070')
AND       reg_term = '199809'
AND       reg_catalog_code = '08701'
AND       reg_section_number = '01'
AND       reg_status_code = 'RG';


      rows registrations
---------- -------------
         3             3
```

FIGURE 8-7 Population of three registrations in a Fall 1998 course.

```
        grade
id      seqno grade
------- ----- -----
@093560
@200197     1 I
@200197     2 B
@625070     1 C
```

FIGURE 8-8 Course grades received.

An Example

Recall the question posed to us in the last chapter by Komenda's registrar: "Everybody assumes that students who take evening classes are generally older than students who take classes during the day, but I'd like to get a feeling for this. Could you please get me a report that shows the birth dates of enrolled students who take evening courses?"

We defined the report population in the preceding chapter, completing steps 1 through 4 in the construction of a query. Figure 8-9 shows the population definition and baseline counts from Figure 7-21 on page 95.

Figure 8-9 also includes a timing statistic that tells you the elapsed time needed for the query to complete. The Where joins provide a good opportunity to use this statistic effectively. Because you step through the Where

ch8.fm Page 105 Thursday, January 25, 2001 6:49 AM


```
SELECT    COUNT(*) "rows",
          COUNT(DISTINCT people_id) "people"
FROM      people
WHERE     EXISTS
          (SELECT   'x'
          FROM      schedule,
                    reg
          WHERE     reg_id = people_id
          AND       reg_term = '199909'
          AND       reg_status_code = 'RG'
          AND       schedule_term = reg_term
          AND       schedule_catalog_code = reg_catalog_code
          AND       schedule_section_number = reg_section_number
          AND       schedule_start_time >= '18:00');


      rows     people
---------- ----------
       508        508

real: 14890   ◀1
```

FIGURE 8-9 Baseline counts of Fall 1999 evening population.

1 Use the SQL*Plus commands `SET TIMING ON` and `SET PAUSE OFF` to display the elapsed time it took the query to complete. The display of this timing statistic is operating system–specific. It's value in Windows appears in thousandths of a second. This query took 14.890 seconds to execute. As we step through the Where joins, the increase in the timing statistic can identify potential problems in the SQL that impact response time.

joins one at a time, the timing statistic will identify any joins that slow down the query significantly. This allows you to investigate performance questions like the indexes that the query uses when retrieving data.

Here are steps 5 through 8 for our query about evening students. Step 8 merely calls for a repeat of steps 6 and 7 until all the joins are added.

- **Step 5**: Identify additional tables needed for the report.
  We need joins to the NAME and DEMOG tables to include names and birth dates.
- **Steps 6 and 7**: Add the join for one table, and test for join errors.
  Let's start with the DEMOG table. The table gets added to the main query in order to include birth dates in the report. The primary key here is demog_id (see Figure A-14 on page 234), so the join between

DEMOG and the PEOPLE table occurs on this data column. Figure 8-10 shows the results. But yikes! We lost 12 people through a one-line join. This means that there are 12 people who do not have data in the DEMOG table.

```
SELECT    COUNT(*) "rows",
          COUNT(DISTINCT people_id) "people"
FROM      demog,
          people
WHERE     EXISTS            1
          (SELECT   'x'
          FROM      schedule,
                    reg
          WHERE     reg_id = people_id
          AND       reg_term = '199909'
          AND       reg_status_code = 'RG'
          AND       schedule_term = reg_term
          AND       schedule_catalog_code = reg_catalog_code
          AND       schedule_section_number = reg_section_number
          AND       schedule_start_time >= '18:00')
AND       demog_id = people_id;

      rows     people          2
---------- ----------
       496        496   3

 real: 15440
```

FIGURE 8-10 Stepping through the DEMOG *join.*

1. Add the DEMOG table to the FROM clause.
2. Then join the DEMOG and PEOPLE tables through data columns common to their primary keys.
3. The counts fell from 508 in the baseline to 496 here. This means that 12 people fell through the DEMOG join. Note that the elapsed time increased only slightly from the baseline query, meaning that the join did not appreciably affect response time.

What's the solution when population items fall through a join? Right. Use an outer join. Figure 8-11 shows that this indeed does restore our counts to baseline. But we might never have caught the error without stepping through this join and testing.

- **Step 8**: Add the join for the next table, and test for join errors.

Now let's deal with the NAME table. Because we've used this table in previous examples, you already know what will happen when joining NAME to the population. Name changes mean that population items will multiply through the join. Figure 8-12 shows that this indeed happens.

```
SELECT    COUNT(*) "rows",
          COUNT(DISTINCT people_id) "people"
FROM      demog,
          people
WHERE     EXISTS
          (SELECT   'x'
          FROM      schedule,
                    reg
          WHERE     reg_id = people_id
          AND       reg_term = '199909'
          AND       reg_status_code = 'RG'
          AND       schedule_term = reg_term
          AND       schedule_catalog_code = reg_catalog_code
          AND       schedule_section_number = reg_section_number
          AND       schedule_start_time >= '18:00')
AND       demog_id(+) = people_id;

      rows     people
---------- ----------
       508        508

 real: 15490
```

FIGURE 8-11 Preventing population loss in the DEMOG *join.*

1. Use an outer join between the DEMOG and PEOPLE tables to prevent population items from falling through the join.
2. Note that the counts again equal the baseline population counts. Elapsed time did not increase significantly.

Note that when population items multiply through a join, you don't know how many items of the population actually were affected. In Figure 8-12 there are 10 more rows than we want. This might mean that 10 people each had two names, or it might mean one person had 11 names, or some other combination entirely.

What's the solution when population items multiply through a join? Sure. Use a correlated subquery and knowledge about the primary key to restore population singularity. The primary key in NAME is name_id and name_seqno (Figure A-24 on page 239). To return the current name, we'll just select the maximum name_seqno as we've done in previous examples. Figure 8-13 shows the final result.

This is as far as we can go for now, but we'll return to this query in the next chapter when discussing the What steps. The query is almost complete. The difficult work of defining the population and maintaining the integrity of that population through the Where joins is behind us.

```
SELECT    COUNT(*) "rows", COUNT(DISTINCT people_id) "people"
FROM      name,
          demog,
          people
WHERE     EXISTS
          (SELECT   'x'
          FROM      schedule,reg
          WHERE     reg_id = people_id
          AND       reg_term = '199909'
          AND       reg_status_code = 'RG'
          AND       schedule_term = reg_term
          AND       schedule_catalog_code = reg_catalog_code
          AND       schedule_section_number = reg_section_number
          AND       schedule_start_time >= '18:00')
AND       demog_id(+) = people_id
AND       name_id = people_id;

      rows     people
---------- ----------
       518        508

 real: 16540
```

FIGURE 8-12 Stepping through the NAME *join.*

1. Add the NAME table to the FROM clause.
2. Add the standard equijoin between the NAME and PEOPLE tables.
3. Only population gain occurred. Elapsed time is fine.

```
SELECT    COUNT(*) "rows",COUNT(DISTINCT people_id) "people"
FROM      name n1, demog, people
WHERE     EXISTS
          (SELECT   'x'
          FROM      schedule,reg
          WHERE     reg_id = people_id
          AND       reg_term = '199909'
          AND       reg_status_code = 'RG'
          AND       schedule_term = reg_term
          AND       schedule_catalog_code = reg_catalog_code
          AND       schedule_section_number = reg_section_number
          AND       schedule_start_time >= '18:00')
AND       demog_id(+) = people_id
AND       name_id = people_id
AND       name_seqno =
          (SELECT   MAX(n2.name_seqno)
          FROM      name n2
          WHERE     n2.name_id = people_id);

      rows     people
---------- ----------
       508        508

 real: 16370
```

FIGURE 8-13 Preventing population gain in the NAME *join.*

1. Eliminate the population gain by using a correlated subquery.
2. Everything looks good!