



## Chapter 7

### Utilities

This chapter contains two of the most important discussions in this book. One concerns the importance of the report population, knowing what it is and defining it properly. The second concerns how to fix things when they go wrong.

Let's start with the latter. You should expect problems to occur in ad hoc queries. This is their nature; in fact, if problems don't occur, you must be very skilled or must be doing boring repetitive queries where you've already solved the problems.

Error-free first drafts of queries should not be an important goal. But finding and fixing the errors easily are important. You don't want to be stuck; you want an easy way to identify when errors exist, an easy way to find the source of the errors, and an easy way to fix the problem. This chapter discusses several debugging tools that allow you this flexibility.

When debugging queries, you absolutely *must* have a clear and accurate understanding of the report population. This probably seems obvious and maybe even seems trivial. But populations are neither obvious nor trivial, and if you don't get them right, your query will suffer.

This chapter discusses a very simple name and address report to drive home the choices that you as a query writer must make about report populations. Even in this simple case at least four different populations are possible. No one population is inherently right or wrong; it all depends on the query in question. However, you must explicitly deal with the alternatives before you really understand the report population. Only then can you actually debug the query with assurance.

## Debugging

Most query problems originate in the joins. It's here that you define the report population and make available for display items of information about the population. Thus you'll need to start debugging a query with a firm understanding of the report population and tools that let you find and fix join problems easily. This section discusses these tools.

### *How do I identify join problems?*

I've written a fairly complex query with joins to eight tables. The query doesn't return the correct number of population items, however, so one or more of the joins must be the problem. How can I find the error?

### **Approach**

To identify join problems, it is absolutely essential that you know what your population is and how many items it includes. Otherwise, when things go wrong, you may be unable to detect it.

Each query contains a population that is defined by all or some portion of the `WHERE` conditions. This is the *report population*. You must know exactly what portion of the `WHERE` clause defines the population because all testing of joins will use this information.

Let's consider a simple situation that hammers home the nuances in population definition. Assume you want a name and address report on 10 people, 1 of whom has no address and 2 of whom have two addresses. What's the report population? How many items does the population include? Is it 9, 10, 13, or 14? Each of these represents a possible population. Take, for example, the population of 10. This is a population of people without regard to their address. The population of 9 is a population of people who have addresses; the population of 13 is a population of addresses for people with at least one address; and the population of 14 is a population of addresses for all 10 people.

Figure 7-1 defines each of these four populations using tables introduced in earlier chapters. Note how the population definitions differ for each of these four populations.

Identifying join problems must start with a clear definition of the report population. If you decide your population includes 10 people, then a query that returns 9 or 14 items means one of two things: (1) it contains an error, or (2) you don't understand the report population.

Here's a test to ensure that you really do understand your report

```

People who have addresses: count = 9
  name_seriesno = name_now(name_id)
  and address_id = name_id
    and address_seriesno = max_address(name_id, 'PR')

People without regard for addresses: count = 10
  name_seriesno = name_now(name_id)
  and address_id(+) = name_id
    and address_type(+) = 'PR'
    and address_seriesno(+) = max_address(name_id, 'PR')

Addresses for people with at least one address: count = 13
  name_seriesno = name_now(name_id)
  and address_id = name_id
    and address_type = 'PR'

Addresses for all 10 people: count = 14
  name_seriesno = name_now(name_id)
  and address_id(+) = name_id
    and address_type(+) = 'PR'
    
```

Figure 7-1: SQL definitions of four possible populations.

population. Start writing each new query by defining the report population in SQL. Test your definition by doing a select COUNT(\*) and COUNT(DISTINCT ...). The two numbers should be identical. Figure 7-2 shows an example of this test.

```

SELECT
  COUNT(*) "count",
  COUNT(DISTINCT name_id) "dist"
FROM
  name
WHERE
  name_seriesno = name_now(name_id);

      count
count(*) distinct
-----
      6         6
    
```

Figure 7-2: Testing the query population definition.

**Comments**

▶ When you properly understand the report population, the two counts will be identical. In this query the population includes 6 people from the *NAME* table.

Consider a situation in which the two counts differ. Suppose the query in Figure 7-2 contained a join to the *ADDRESS* table and that `COUNT(*)` was 8 and `COUNT(DISTINCT name_id)` was 6. This means that you either don't understand the report population or that you've defined it incorrectly. If the report population contains 6 people, then the two counts should each be 6. However, if the report population is the 8 addresses, then the two counts should each be 8. Either could be correct depending on what you want the query to do, but you *must* know the difference. If the population is 8, then the correct counts will be `COUNT(*)` and `COUNT(DISTINCT name_id || address_seriesno)`. This literally forces you to acknowledge that the population items are addresses and not people.

Once the report population is properly defined, testing the joins is straightforward. Use a procedure called *stepping through the joins*. It works like this:

- Do a `COUNT(*)` and `COUNT(DISTINCT . . .)` for the entire query. If the two counts differ from your population counts, you have a problem in the joins.
- Add one join at a time to the basic population definition and perform the `COUNT(*)` and `COUNT(DISTINCT . . .)` again. Whenever the counts change from the population values, you've located a join that contains a problem. For example, suppose the query contains 5 joins in addition to those included in the population definition. Include the first join with the population definition and run the counts. If they're the same as the population definitions, then move to the next join. Proceed in this fashion through all the joins until you find the one(s) where the counts differ from the population values.
- Two types of errors can occur in joins: Either you can lose population items (called *falling through a join*), or you can gain population items (called *multiplying through a join*). For example, assume the population `COUNT(*)` was 6 and the `COUNT(DISTINCT . . .)` was 6 (designated here as 6/6 for ease of discussion). Also assume that when you add one of the joins, you now get a count of 5/5. In this case you've lost 1 population item. If the counts are 6/7, then you got a multiplication through the join and one population item appears twice. If you got counts of 5/7, it means the join has both types of errors — population loss and population gain.

Sometimes the error in the joins is obvious. But often it is not.

Correcting the join conditions requires that you first identify which rows create the problems. You can then compare the data values in these rows with the join conditions and determine why they either fell through or multiplied through the join. The next sections discuss these topics.

### See Also

1. Correlated subqueries as predicates to `EXISTS` or `NOT EXISTS` are marvelous adjectives qualifying a population. One reason they're so useful is that falling through a join and multiplying through a join are never a problem with these query structures. You simply look for the existence or absence of a characteristic; they become part of your population definition. For more discussion on this point, see the section on subqueries that begins in Chapter 5 on page 148.
2. Figure 7-1 uses two PL/SQL functions — `name_now` and `max_address`. The PL/SQL for `name_now` appears in Chapter 1 (Figure 1-9 on page 13); the PL/SQL for `max_address` is in Chapter 4 (Figure 4-4 on page 112).

### How do I find rows that fell through a join?

I've written a query that contains a problem in one join. Population items fall through this join, but I can't figure out why. How can I identify the rows that fall through the join so that I can examine their data values and determine why they cause problems with the join logic?

### Approach

After you isolate a query problem to a specific join, it's often helpful to examine the rows that fail the join condition. Let's continue with the example from Figure 7-2 where the population included 6 people. Figure 7-3 shows a query where join errors occur.

```

SELECT
    COUNT(*) "count",
    COUNT(DISTINCT name_id) "dist"
FROM
    address,
    person,
    name
WHERE
    name_seriesno = name_now(name_id)
    AND person_id = name_id
    AND address_id = name_id
      AND address_type = 'PR'
      AND address_status = 'A';

```

count(*)	count distinct
5	4

Figure 7-3: Query that contains two types of join failures.

### Comments

► The counts should be 6 and 6. Apparently, two people fell through the joins and 1 person multiplied through the join.

To find the offending join, use the procedure discussed in the preceding section and step through the joins one-by-one. First, add the *PERSON* join and rerun the counts. Then add the *ADDRESS* join and do the counts again. In the example above, both problems occur in the *ADDRESS* join.

To identify rows that fall through a join, use a simple SQL query with a *NOT EXISTS* correlated subquery. Figure 7-4 illustrates how this works to identify the two people who fell through the join in Figure 7-3.

```

SELECT
  name_id
FROM
  name
WHERE
  name_seriesno = name_now(name_id)
  AND NOT EXISTS
  (SELECT 'x'
   FROM address
   WHERE address_id = name_id
         AND address_type = 'PR'
         AND address_status = 'A');

id
----
@226
@925
    
```

Figure 7-4: SQL structure to identify items that fall through a join.

**Comments**

▶ Use NOT EXISTS and a correlated subquery with the exact WHERE conditions that you used in the problematic join (e.g., see Figure 7-3 on page 222). This will identify the rows that fell through the join. In this case, the two people affected by the query have ID numbers @226 and @925.

Then examine the data for the rows that fell through the join. You’ll find the reasons for the join failures by examining the data values. Occasionally it’s a problem with the data, such as invalid codes that were not trapped with referential constraints. Most frequently, however, the problem is query writer error in the join logic. Figure 7-5 shows ADDRESS data for the two people who fell through the ADDRESS join in our example.

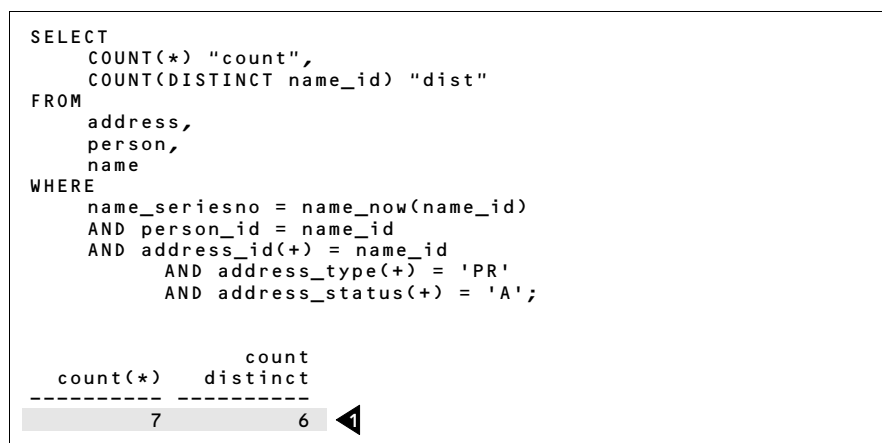
id	type	seq	status	city	loc	postal code	nat
@226	BU	1	A	Charleston	IL	60954	
@925							

Figure 7-5: Data for two people who fell through the ADDRESS join.

**Comments**

▶ This report makes it immediately clear why @226 and @925 fell through the address join. One person (@226) has no PR address; the second (@925) has no address at all.

At this point you're prepared to revise the original query. Both situations in our example can be fixed with outer joins. Figure 7-6 shows COUNT(\*) and COUNT(DISTINCT . . .) after including outer joins.



**Figure 7-6:** Query that prevents population loss.

**Comments**

▶ The query now includes the correct number of people (i.e., six), indicating that we've plugged the join problems that cause population loss. Note, however, that one person still multiplies through a join. The next section discusses this problem.

**See Also**

1. This section used a correlated subquery to find rows that fell through a join. You also could use other methods to identify these rows. For a discussion of these methods, see the section in this chapter on DBA tools that begins on page 228.
2. You can use correlated subqueries as predicates to EXISTS or NOT EXISTS to help define report populations. They behave very much like adjectives qualifying nouns. For a discussion of this, see Chapter 5 on page 151.

3. Outer joins often pose problems for query writers. For a discussion of some of these problems, see the sections that begin Chapter 5 on page 156
4. The PL/SQL for the name\_now function appears in Chapter 1 (Figure 1-9 on page 13).

### How do I find rows that multiplied through a join?

I've written a query that contains a problem in one join. Population items multiply through this join, but I can't figure out why. How can I identify the rows that multiply through the join so that I can examine their data values and determine why they cause problems with the join logic?

### Approach

Let's continue with our example from preceding sections. You can identify rows that multiply through a join using a distinctive SQL structure with a `GROUP BY` and a `HAVING`. Figure 7-7 illustrates how this works.

```

SELECT
    name_id "id",
    COUNT(*) "count"
FROM
    address, name
WHERE
    name_seriesno = name_now(name_id)
    AND address_id(+) = name_id
        AND address_type(+) = 'PR'
        AND address_status(+) = 'A'
GROUP BY
    name_id
HAVING
    COUNT(*) > 1;

```

id	count(*)
@330	2

**Figure 7-7:** SQL structure that identifies items that multiply through a join.

### Comments

► `GROUP BY` and `HAVING` are the keys to this query, returning all population items that appear more than once in the result set.

Based on this information, you can examine the data for population items that multiplied through the joins. In most cases the problem occurs in the join logic, but occasionally you'll uncover data problems that the database design did not prevent.

In our example the situation is simple; the person has two active permanent addresses. By choosing only one of the two, you can avoid the problem of multiplying through the join. Figure 7-8 shows the final query and its accompanying `COUNT(*)` and `COUNT(DISTINCT . . .)`.

```

SELECT
  COUNT(*) "count",
  COUNT(DISTINCT name_id) "dist"
FROM
  address,
  person,
  name
WHERE
  name_seriesno = name_now(name_id)
  AND person_id = name_id
  AND address_id(+) = name_id
  AND address_type(+) = 'PR'
  AND address_status(+) = 'A'
  AND address_seriesno(+) =
    max_address(name_id, 'PR');

```

count(*)	distinct
6	6

Figure 7-8: Final query that solves all join problems.

**Comments**

▶ The PL/SQL function max\_address introduced in Chapter 4 (Figure 4-4 on page 112) returns the last active address of a specified type. The outer join preserves the previous fix in Figure 7-6 on page 224 that prevented population loss.

In complicated queries with multiple joins that need debugging, make sure you run the last COUNT(\*) and COUNT(DISTINCT . . .) on the entire query. It's possible that by fixing one join at a time you'll get interactions between your join solutions so that the entire query still does not return the correct counts.

**See Also**

1. Instead of GROUP BY and HAVING, you can also use other methods to identify rows that multiply through joins. For a discussion of these methods, see the section in this chapter on DBA tools that begins on page 230.
2. See Chapter 1 (Figure 1-9 on page 13) for the name\_now function and Chapter 4 (Figure 4-4 on page 112) for the max\_address function.

## DBA Tools

Debugging tools used by query writers to find and fix join problems originate in tools that database administrators use for other purposes. This section discusses those tools and emphasizes alternative methods of accomplishing the same task. It's important for query writers to have a variety of methods they can employ as needed. Not any one method works best in all situations.

### *How do I find foreign key violations?*

How can I identify rows in one table that should have corresponding rows in a second table but do not? For example, I might have a master table and a detail table and want to delete all rows in the master table without detail information. Or I might want to find foreign key violations by identifying all rows in one table where the foreign key references a primary key that does not exist in a second table. Are there alternative approaches to this problem?

### Approach

This question is of interest to query writers because it's essentially the same one as finding rows that fall through a join. The solutions to this question provide query writers with alternative methods to debug their queries by finding rows that fall through a join.

Frequently it is difficult to classify one approach as the *best*. Generally speaking, however, as long as two queries both return accurate results, the query that completes in the shortest time is preferred (this assumes, of course, that the query writer doesn't spend hours finding the optimal query when he could have written an acceptable query in minutes).

Figure 7-9 shows four alternative methods that all identify rows from one table that do not have corresponding information in a second table. In the figure, the table names are A and B. These tables are joined through the common column named `xid`.

### Comments

- ▶ Method 1 uses an outer join to create `NULL` rows in table B for items in table A that do not exist in table B. The query then uses `ROWID IS NULL` to identify these rows.
- ▶ Method 2 uses a correlated subquery as a predicate to `NOT EXISTS`.

```

Method 1: where B.xid(+) = A.xid and B.rowid is null
Method 2: where not exists
          (select 'x' from B where B.xid = A.xid)
Method 3: where A.xid not in
          (select B.xid from B)
Method 4: select A.xid from A
          MINUS
          select B.xid from B
    
```

Figure 7-9: Identifying rows in one table not represented in a second table.

- ▶ Method 3 uses a subquery but not a correlated subquery.
- ▶ Method 4 uses a compound query and the MINUS operation.

Method 2, which is the approach used to identify rows that fell through the *ADDRESS* join in Figure 7-4 on page 223, works well when the join columns are the leading edge of indexes in both tables A and B. When indexes do not exist, or you cannot take advantage of the leading edge of available indexes, use the MINUS compound query. Figure 7-10 illustrates this when debugging the *ADDRESS* join from our example.

```

SELECT
  name_id "id"
FROM
  name
MINUS
SELECT
  address_id "id"
FROM
  address
WHERE
  address_type = 'PR'
  AND address_status = 'A';

id
----
@226
@925
    
```

Figure 7-10: Identifying rows that fell through the join in Figure 7-3.

### See Also

1. Chapter 9 (Table 9-12 on page 278) compares the performance of the four methods for finding rows in one table without corresponding rows in a second table.

*How do I find duplicates in a table?*

I tried to create a unique index on a combination of several columns yet got error `ORA-01452` that duplicate keys existed. How do I find which rows contain duplicates on these columns?

**Approach**

A DBA might ask this question; query writers never consider duplicate keys. But solutions to the problem do affect query writers, because you can use the same techniques to debug your queries when rows multiply through joins. Figure 7-7 on page 226 found multiple address records when only one was expected. This is very similar to finding duplicates.

Figure 7-11 provides several alternative methods for identifying duplicate keys in a table. The figure presents the simplest situation where the key is only a single column, but the methods can be extended easily to multiple columns. Note that the debugging technique used to find rows that multiplied through the address join in Figure 7-7 relied on Method 1.

```

Method 1: SELECT col1, COUNT(*)
          FROM table_name A
          WHERE ...
          GROUP BY col1
          HAVING COUNT(*) > 1; 1

Method 2: SELECT col1
          FROM table_name A
          WHERE ...
          AND 1 <
            (SELECT COUNT(*) FROM table_name B
             WHERE B.col1 = A.col1); 2

Method 3: SELECT col1
          FROM table_name A
          WHERE ...
          AND EXISTS
            (SELECT 'x'
             FROM table_name B
             WHERE B.col1 = A.col1
             AND B.ROWID != A.ROWID); 3

```

**Figure 7-11:** Methods to identify duplicate keys in a table.

**Comments**

- ▶ Method 1 uses a `GROUP BY` and `HAVING` to identify `col1` values that appear more than once.
- ▶ Method 2 uses a correlated subquery to identify `col1` values

that appear more than once.

➤ Method 3 also uses a correlated subquery, this time as a predicate to an `EXISTS`. The condition is met when two rows have the same `col1` value but their `ROWIDs` differ.

Chapter 9 discusses the performance implications of these alternative methods for finding duplicates. Figure 7-12 illustrates how Method 2 can be used to debug the address join in our earlier example and find rows that multiplied through this join.

```
SELECT
  name_id "id"
FROM
  name
WHERE
  name_seriesno = name_now(name_id)
  AND 1 <
  (SELECT COUNT(*)
   FROM address
   WHERE address_id = name_id
         AND address_type = 'PR'
         AND address_status = 'A');
```

id
-----
@330

Figure 7-12: Identifying rows that multiplied through the join in Figure 7-3.

### See Also

1. Chapter 9 (Table 9-13 on page 280) compares the performance of the three methods for identifying duplicates.
2. The `PL/SQL` for the `name_now` function used in Figure 7-12 appears in Chapter 1 (Figure 1-9 on page 13).

