

CHAPTER NINE



What

The Who and the Where form the foundation of each SQL query. The What is mere window dressing. But it's the What that people actually see when they use a report, so it too is very important. Compared to Who and Where, however, What is easy. This chapter should provide a respite from the rigors of the previous two chapters. There's just an incredible variety of ways that you can display data in SQL. This chapter illustrates that variety.

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 step 9 in query construction. Anything that should appear in the final report gets added during this step, as does the necessary grouping and sorting. When you've completed the What step, you've also completed the SQL `SELECT` command. Only report formatting (step 10) remains.

Select Clause

In the `SELECT` clause you specify what information will appear in the report. For example, a simple name report might include `name_id`, `name_first`, and `name_last`. The `SELECT` clause for this report would be `SELECT name_id, name_first, name_last`. Pretty easy.

Like all SQL, the `SELECT` clause must adhere to a specific syntax. The name example just discussed uses the most common form of `SELECT`, where the syntax is `SELECT expr`. The `expr` is shorthand for “expression” and includes constructed expressions and data columns. Use commas to separate items in the `SELECT` clause. The most common items that appear in `SELECT` clauses are

- Data columns (e.g., `name_id`).
- Constants (e.g., the number 10 or the character string “Not applicable”). When a constant appears in the `SELECT` clause, the report will include a column where every row contains the same value.
- SQL functions. There are many SQL functions, each of which performs a very specific and useful task. *Number functions* work with numeric data values (e.g., to round a number to two decimal places). *Character functions* work with character data values (e.g., to convert a character string to lowercase). *Date functions* work with date values (e.g., to perform date arithmetic). *Conversion functions* convert one type of data value to another type (e.g., converting a date to a character value for special formatting). *Group functions* provide numeric summaries of grouped data (e.g., the average or minimum or maximum for selected data values). And then there are *miscellaneous functions* that perform special operations (e.g., to temporarily substitute a value like “Missing” for a null).

With some functions you have options about how the resulting data appear. The appearance is controlled by something called *format models*. For example, you can format dates and numbers in a multitude of ways. This chapter introduces several common SQL functions. Chapter 13 and Appendix D cover functions in considerably more depth.

- User functions created with PL/SQL to perform user-defined tasks. For those tasks which may be idiosyncratic to your organization or way

of doing business, such as calculating grade point averages in a college, you can create your own functions and use them for reporting.

- Combinations of other items (e.g., using the concatenation function to combine last, first, and middle names into a sort name). This use occurs frequently when using numeric data columns in calculations.

The following items also can appear in the `SELECT` clause, although they tend to appear less frequently in queries than data columns or constructed expressions:

- The keyword `NULL`. This gets used most frequently in a special type of query called a *compound query* involving *set operations* like `UNION` and `INTERSECT`. Chapter 12 discusses set operations.
- Pseudocolumns provided by Oracle. *Pseudocolumns* are special data columns that Oracle makes available for each table. You don't ever see them in a data dictionary, but they exist for each table and can be queried. Three pseudocolumns can appear in a `SELECT` clause. One is `LEVEL`; it gets used in a special type of query called a *hierarchical query* that we won't discuss in this book.

A second pseudocolumn is `ROWNUM`; it is just a sequential number assigned to each row in the order the row gets retrieved in the query. `ROWNUM` also can be used in the `WHERE` clause to limit the number of rows returned, typically so that you can check the format of the report before running the entire query. For example, `WHERE ROWNUM <= 5` would return just five rows. `ROWNUM` is determined before rows get sorted with an `ORDER BY`.

The third pseudocolumn is `ROWID`; it is an address for the location of each row in a table. By location, I mean a physical location—which data file, which data block within the file, and which row within the data block. For a given table, `ROWID` is always unique—a handy feature when trying to enforce singularity in a correlated subquery where a primary key is unavailable or inappropriate.

While `SELECT expr` is the most common syntax used in `SELECT` clauses, there are a few other wrinkles that you should know. Each of the following is also acceptable:

- **SELECT DISTINCT** expr. This is a variation on **SELECT** expr that only returns distinct rows in a query. Here's what I mean by that. Suppose that the expr in the **SELECT** clause is a list of five data columns. If two rows in the query have exactly the same values for all five data columns, then **SELECT DISTINCT** expr will only include one of the two rows in the report, whereas **SELECT** expr will include both rows. I have to admit a dislike for **SELECT DISTINCT** expr. People tend to use it as a way to write sloppy SQL but still see the results they want. They ignore the concept of populations and create queries where population items multiply through the joins and then use the **SELECT DISTINCT** expr to pretend like the population gain never occurred. That's ugly SQL. There is almost always a way to write a query so that you can avoid **SELECT DISTINCT**. When you construct these alternatives, you're crafting a query.
- **SELECT ***. This syntax returns all data columns in all tables or views that appear in the **FROM** clause. For example, the **DEMOG** table contains the four data columns **demog_id**, **demog_sex**, **demog_birthdate**, and **demog_update**. The query **SELECT * FROM demog** produces a report with all four data columns. This is useful for small queries, e.g., when displaying the values used in a code table. However, if your query contains several joins, then **SELECT *** can generate a ferocious number of columns very quickly.
- **SELECT table.*** or **SELECT view.***. This is a variation of **SELECT ***. If the query contains several tables or views in the **FROM** clause, you can use this syntax to return all data columns from a specific table or view rather than from all tables or views. For example, if both the **NAME** and **DEMOG** tables appear in the **FROM** clause, then **SELECT name_last, demog.*** will produce a report with all four data columns from the **DEMOG** table plus the column **name_last**.
- **SELECT expr AS alias**. This is another variation on **SELECT** expr. It allows you to assign an alias to an item in the **SELECT** clause. For example, suppose we concatenate **catalog_subject_code** and **catalog_course_number** to identify courses (e.g., HIS100). We can assign an alias called *course* in the following **SELECT** clause: **SELECT catalog_subject_code || catalog_course_number AS course**. This is

handy because the header in the report can use the alias. You also can sort a report by referring to the alias. Unfortunately, you cannot use an alias in a **GROUP BY**.

When you write a **SELECT** clause, remember to structure it for readability. I generally put each item in the **SELECT** clause on a separate line. Also note that if an item is too long to appear easily on a single line, you can continue it on the next line. No special punctuation is needed to continue SQL on successive lines.

■ Exercises

1. List the different types of items that can appear in a **SELECT** clause.
2. What's the purpose of a format model in some functions?
3. What is the principal difference between SQL functions and user functions?
4. Write a query that would list all the data columns in **SUBJECT_CODES**.

Order By Clause

You sort the rows in a report with the **ORDER BY** clause. For example, if the report shows students' names and you want the list in alphabetical order, you would **ORDER BY** `name_last, name_first, name_middle`. Again, pretty simple.

The basic syntax for the **ORDER BY** clause is **ORDER BY** `expr`, where `expr` is a list of items separated by commas. Like the **SELECT** clause, the items in the **ORDER BY** clause can include data columns, constants, SQL functions, user functions, combinations of other items, **NULL**, and pseudocolumns.

There are also two variations on the basic syntax that you should know:

- **ORDER BY** position. Each item in the **SELECT** clause has a position. The first item has position 1, the second item position 2, and so on. To sort by one of the items in the **SELECT** clause, you can use this *positional notation* in the **ORDER BY**. This is particularly handy when the **SELECT** clause includes a complex expression and you want to sort by it. For

example, suppose that our query includes `catalog_subject_code || catalog_course_number` in position 3 in the `SELECT` clause. To sort the report by this expression, you would use `ORDER BY 3`. It's just a nice shorthand for sorting.

- **ORDER BY** alias. If you assign an alias to an item in the `SELECT` clause with `AS` or without using double quotes, then that alias can be used in the `ORDER BY`. For example, the following query works fine: `SELECT catalog_code, catalog_subject_code || catalog_course_number AS course FROM catalog ORDER BY course`. It's important to note that the alias cannot be assigned with double quotes (e.g. `SELECT name_first "first" FROM name`) if you intend to use it in the `ORDER BY`. See Figure 3-6 on page 39 for an example that assigns aliases in three ways.

There are several points worth noting about the `ORDER BY` clause. These include

- You can sort in ascending or descending order. The default is ascending. To sort in descending order, use the keyword `DESC`. The corresponding keyword for ascending is `ASC`, although specifying this is unnecessary because it is the default.
- When more than one item appears in the `ORDER BY` clause, the report is ordered by the item in position 1. Rows with the same values in position 1 are then ordered by the item in position 2, and so on. For example, `ORDER BY name_last, name_first` will order rows by last name and then first name. If there are five people whose last name is Jackson, the report lists them in order of their first name.
- The actual order in which something sorts depends on parameters established by your database administrator. Say what? Well, it's like this. Sorting is relative; it can change depending on several factors. Typically, you'll never experience this relativity because the factors that influence sorts get determined once by your database administrator and then seldom change. But tuck this into the back of your mind—what sorts one way using a specific character set may very well sort another way using a different character set. This topic gets into National Language Support, which Chapter 13 discusses.

- **ORDER BY** is not restricted to items that appear in the **SELECT** clause. For example, suppose that you want a report to sort by sex and then by last name, but you don't need to include sex as one of the data columns in the report. Then **ORDER BY** `demog_sex, name_last` accomplishes the sorting task even though `demog_sex` does not appear in the **SELECT** clause.
- **ORDER BY** cannot be used in subqueries.

■ Exercises

1. Write a query that would list all the data columns in the **SUBJECT_CODES** table, and sort the results by `subject_code`.
2. Why would someone use positional notation in an **ORDER BY** clause?
3. How will rows in the following query appear in the report: **SELECT** `schedule_catalog_code, schedule_section_number` **FROM** `schedule` **WHERE** `schedule_term = '199909'` **ORDER BY** `schedule_meeting_days, schedule_building_code, schedule_room_code, schedule_start_time` **DESC**?
4. Suppose that you're asked to provide a report to identify the five courses with the largest possible credits. You write the following query: **SELECT** `catalog_max_credits` **FROM** `catalog` **WHERE** `ROWNUM <= 5` **ORDER BY** `catalog_max_credits` **DESC**. Will this work?

Group By Clause

Up to this point, each item in a query population has produced one row in the report, e.g., one student per line in the report. But there are many occasions when you want information about the population as an entire group. In these situations, you'll need to use group functions. For example, we might want to know the average age for a population of students. To do this, we'll need to include the **AVG** group function in the **SELECT** clause.

There are also many occasions when you want information about subgroups of the population, e.g., counts of students by gender or the sum of

student credit hours broken down by academic subject. In these situations, and for reasons we'll discuss shortly, you'll need to use the `GROUP BY` clause in addition to the group functions. If you want only some of the subgroups to appear in the report, the `HAVING` clause is also required.

We've already used one group function quite extensively without actually acknowledging it as such. The `COUNT` function counts the number of rows in a group. We've always used it to count the items in our entire population, but it also can be used to count population items by subgroup. The general form for `COUNT` is `COUNT({* | [DISTINCT | ALL] expr})`. Here's the interpretation. The `|` character reads as an "or". So the general form for `COUNT` includes these specific variations: `COUNT(*)`, `COUNT(DISTINCT expr)`, and `COUNT(ALL expr)`. It also includes `COUNT(expr)` because `ALL` is the default and may be omitted. `COUNT(*)` counts all rows including those with null values, `COUNT(DISTINCT expr)` counts only those expressions which are distinct and not null, whereas `COUNT(ALL expr)` counts only those expressions which are not null but counts all of them, regardless of whether they are distinct from each other.

Thus, when our baseline counts include the clause `SELECT COUNT(*)`, `COUNT(DISTINCT people_id)`, the `COUNT(*)` counts all rows in the query, and the `COUNT(DISTINCT people_id)` counts each `people_id` once. With a `COUNT(*)` of 1010 and a `COUNT(DISTINCT people_id)` of 1000, we know that our population includes 1000 people but that at least one of them appears more than once.

Let's pause for just a moment and ensure that the distinctions between the various counts are clear. Suppose that we have a data column called `col_x` and four rows with the following data values in `col_x`: a, null, b, a. `COUNT(*)` returns 4 because it includes the null value; `COUNT(DISTINCT col_x)` returns 2 because it ignores the null and only counts distinct values a and b, and `COUNT(ALL col_x)` returns 3 because it ignores null values but counts all values regardless of whether they're distinct. Okay, enough for the digression into the inner workings of `COUNT`. But since `COUNT` gets used constantly while debugging the Who and the Where in queries, this digression served a purpose.

One very common type of report is the frequency distribution. It shows counts of a population by subgroups. Figure 9-1 provides an example, showing counts by sex for the population of students enrolled in Fall 1999.

```

SELECT  demog_sex "sex", 1
        count(*) "count"
FROM    demog,
        people
WHERE   EXISTS
        (SELECT 'x'
         FROM   reg
         WHERE  reg_id = people_id
         AND    reg_term = '199909'
         AND    reg_status_code = 'RG')
AND     demog_id(+) = people_id
GROUP BY demog_sex
ORDER BY 1; 2

```

sex	count
F	497
M	586
	101

FIGURE 9-1 Distribution of Fall 1999 students by their sex.

- ▶ The **SELECT** clause specifies what summary statistics will be computed for the groups identified in the **GROUP BY** clause. In this case, we just want counts. The data column `demog_sex` is included so that the counts in the report will be labeled by sex.
- ▶ The **GROUP BY** specifies how population items will be grouped so that summary statistics can be computed. In this case, the query groups students by sex. Note that the **GROUP BY** and **SELECT** clauses are coordinated so that all data columns and expressions appearing in the **SELECT** clause also appear in the **GROUP BY**.

Here are the key points to remember about **GROUP BY**:

- The **GROUP BY** clause specifies which groups will be used to compute summary statistics.
- The **SELECT** clause identifies what summary statistics to compute for the groups.
- Summary statistics for the entire population can be obtained by omitting the **GROUP BY** clause and including only group functions in the **SELECT** clause (e.g., as was done in the baseline population counts used in many previous examples).
- If the **SELECT** clause includes data columns and expressions, the

`SELECT` and `GROUP BY` clauses must be coordinated in a very precise way. Otherwise, you get the dreaded error `ORA-00979: not a GROUP BY expression`. Here's a rule that always works: Make all items in the `GROUP BY` clause identical to items in the `SELECT` clause that are not group functions. However, I have to hedge a bit here. You actually have a little more latitude than maintaining this strict identity between the `SELECT` and `GROUP BY` clauses. Here's the hedge: When the `GROUP BY` uses a data column (as distinct from an expression), you can use that data column in the `SELECT` clause as a data column (i.e., maintaining strict identity) or as an expression (i.e., maintaining only partial identity). I know this probably isn't clear, but an example discussed below should help some.

- Including data columns or expressions in the `SELECT` clause allows you to identify which groups the summary statistics refer to. In Figure 9-1, for example, students were grouped by sex. By including `demog_sex` in the `SELECT` clause, the report links counts to specific genders. If we'd omitted `demog_sex` in the `SELECT` clause, the report would have shown three counts with no indication what they meant.
- Population items may be grouped on more than one criteria. For example, `GROUP BY demog_sex, demog_birthdate` groups first by sex and then, within sex, by birth date.

Here are several queries that meet these general rules and produce valid reports:

- `SELECT COUNT(*) FROM demog`. Counts all rows in the `DEMOG` table.
- `SELECT COUNT(*) FROM demog GROUP BY demog_sex`. Counts rows by sex but does not identify which sex the numbers refer to. This report would not be very useful.
- `SELECT demog_sex, COUNT(*) FROM demog GROUP BY demog_sex`. Counts rows by sex. Also identifies which sex the numbers refer to.
- `SELECT DECODE(demog_sex, NULL, 'Unknown', demog_sex), COUNT(*) FROM demog GROUP BY demog_sex`. Counts rows by sex. Displays an "Unknown" in the report if the sex is null. This is an example where the `GROUP BY` uses a data column (i.e., `demog_sex`), but the

data column appears in the `SELECT` as an expression (i.e., maintaining only partial identity between `SELECT` and `GROUP BY`).

Here are some queries that generate errors:

- `SELECT demog_birthdate, COUNT(*) FROM demog GROUP BY demog_sex`. The `SELECT` clause includes the data column `demog_birthdate` that does not appear in the `GROUP BY`. This generates error `ORA-00979: not a GROUP BY expression`. Make sure the `SELECT` and `GROUP BY` clauses use the same data column.
- `SELECT demog_sex, COUNT(*) FROM demog GROUP BY DECODE(demog_sex, NULL, 'Unknown', demog_sex)`. The `SELECT` clause and the `GROUP BY` both include the data column `demog_sex`, but in the `GROUP BY` this column occurs as an expression. You'll need to include the identical expression in the `SELECT` clause to avoid the error `ORA-00979`.

Having Clause

Think of `HAVING` as a `WHERE` clause for groups. Use it to include or exclude groups from the final report. Figure 9-1 showed the distribution of students by sex. Suppose for some strange reason we only wanted to include groups with 500 or more students. You can do this easily with the following clause: `HAVING COUNT(*) >= 500`. In the report, only the line for men would appear because each of the other groups has fewer than 500 students.

The key points to know about `HAVING` are

- If `WHERE`, `GROUP BY`, and `HAVING` clauses all appear in a query, the `WHERE` clause gets processed first. The remaining rows are then grouped according to the `GROUP BY` criteria. And finally, the groups are restricted to those specified in the `HAVING` clause.
- `HAVING` clauses use group functions; `WHERE` clauses do not. Use `WHERE` clauses rather than `HAVING` clauses to limit individual rows. For example, use `WHERE demog_sex = 'F'` rather than `HAVING demog_sex = 'F'` to limit the query to women.

HAVING frequently gets used when debugging queries where population items multiply through a join. Often it's not clear why the population gain occurred, and you'll need to examine the data for the population items before you can correct the join error. **HAVING** allows you to identify the population items affected by the join.

In Figure 8-12 on page 108 we added the **NAME** join to a population of evening students for Fall 1999 and found that rows multiplied through the join. Figure 9-2 uses a **HAVING** clause to identify which students multiplied through the join.

```

SELECT  people_id "id",
        COUNT(*) "count"
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
GROUP BY people_id
HAVING  COUNT(*) > 1
ORDER BY 1;

```

id	count
@216829	2
@370221	2
@381366	2
@497466	2
@612585	2
@625070	2
@706435	2
@753128	2
@840196	2
@912198	2

FIGURE 9-2 **HAVING** used to debug queries with population gain.

- This query identifies those people who multiplied through the **NAME** join. Including the `people_id` in the **SELECT** clause ensures that this data column displays in the report.
- **GROUP BY** `people_id` to get counts for each student. Use the **HAVING** clause to find only those students with more than one row, i.e., those who multiplied through the **NAME** join.

In this case, there are 10 students each of whom has two **NAME** rows. If you didn't already know what caused the multiplication through the **NAME** join, you could examine the **NAME** data for the 10 students to identify what created the problem.

When the problem causing population gain is not obvious, **HAVING** provides a handy debugging tool. Sometimes you might even find that the SQL you wrote is fine, but there's something wrong with the data. If so, report the error to someone who can correct it.

■ Exercises

- Suppose that we were asked to find the average credits hours taken by students enrolled in Fall 1999. This would require the group function **AVG** in the **SELECT** clause. Would it also require a **GROUP BY** clause? Why or why not?
- To determine the most popular class times at Komenda, suppose that we decided to get a frequency distribution of `schedule_start_time` for Fall 1999. Write the **SELECT** clause for this query. Would the query require a **GROUP BY** clause? If so, write that as well.
- Will the following queries produce valid SQL or generate an error message?
 - `SELECT student_degree_code, COUNT(*) FROM student`
 - `SELECT COUNT(*) FROM student`
 - `SELECT student_degree_code, COUNT(*) FROM student
GROUP BY student_degree_code`
 - `SELECT '***' || student_degree_code || '***', COUNT(*)
FROM student GROUP BY student_degree_code`
 - `SELECT student_degree_code, COUNT(*) FROM student
GROUP BY '***' || student_degree_code || '***'`
- How are rows processed in the following query:


```
SELECT student_degree_code, COUNT(*) FROM student
WHERE student_status_code = 'A'
GROUP BY student_degree_code HAVING COUNT(*) > 100?
```

An Example

Recall the question posed to us in Chapter 7 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 birthdates of enrolled students who take evening courses?"

We defined the population for this report in Chapter 7 (see Figure 7-21 on page 95). In Chapter 8 we stepped through the Where joins. This required an outer join to DEMOG to prevent population items from falling through the join and a correlated subquery to prevent population items from multiplying through the NAME table (see Figure 8-13 on page 108). Figure 9-3 shows the point in Chapter 8 where we left this query.

```

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

```

FIGURE 9-3 Fall 1999 evening population query.

The registrar wanted to see an identification number, name, and birthdate. Let's also provide the age of students and sort the report in descending order by age. Figure 9-4 shows one way to display this information.

This figure demonstrates several new SQL elements. These are

- The **SELECT** clause includes items that actually will appear in the report. In this case, the clause includes one data column (`people_id`)

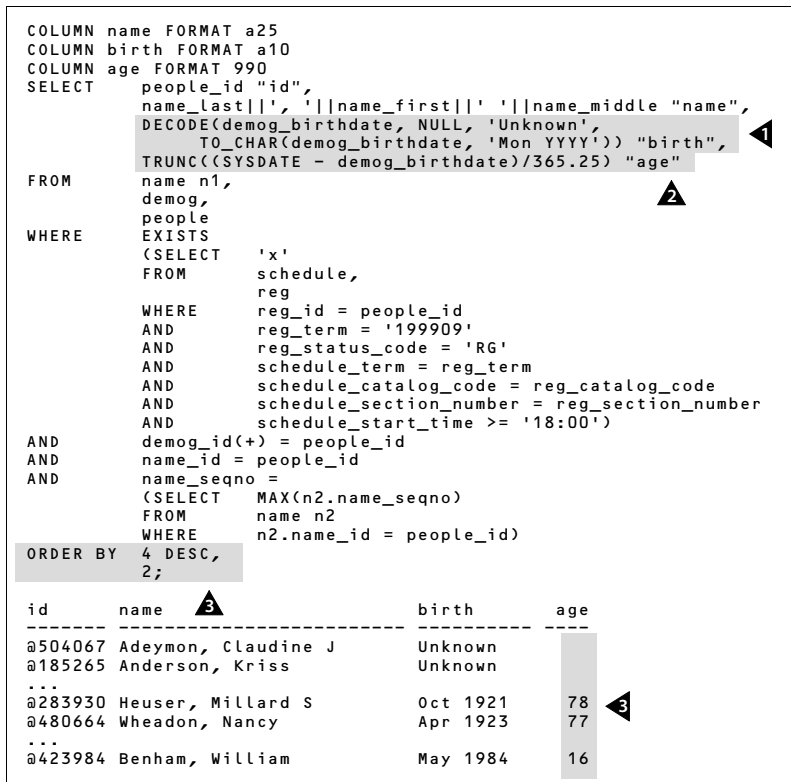


FIGURE 9-4 What steps completed in Fall 1999 evening query.

- ▶ The `DECODE` function displays Unknown for null birth dates. See the text for a discussion. For people who do have birth dates, the `TO_CHAR` function formats birth dates using a “Mon YYYY” format model.
- ▶ The `SYSDATE` function returns the date when the report was run. The expression uses Oracle’s ability to do date arithmetic by subtracting the birth date from `SYSDATE` (an answer expressed in days) and then dividing by 365.25 to determine the years of age. The `TRUNC` function truncates any decimals.
- ▶ The report is sorted by age in descending order and, within any age, by name. Note that null values sort before other values in descending order.

and expressions for name, birth date, and age. Each expression is given a descriptive alias that appears as a column header in the report.

- The name expression uses the concatenation operator `||` to combine last, first, and middle names with a comma and spaces included for punctuation and readability.
- The birth date expression uses a handy function called `DECODE`. This allows you to implement if ... then ... else logic in an expression. The general layout is `DECODE(expr, search1, result1, [search2, result2,] ..., [default])`. Items surrounded by brackets are optional. Thus, the translation is: If `expr` matches `search1`, return `result1`; otherwise, if `expr` matches `search2`, return `result2`; otherwise, keep checking additional criteria in the list; if no matches can be found, then return the default value. In words, this is what the `DECODE` in Figure 9-4 says: If birthdate data are unavailable for a student, display the constant “Unknown.” Otherwise, return the expression that uses the `TO_CHAR` function. For more information on `DECODE`, see miscellaneous functions in Appendix D (page 298).
- The birth date expression also uses the `TO_CHAR` function that converts date values to character values so that they can be formatted in a variety of ways. The general form for `TO_CHAR` is `TO_CHAR(date [, fmt [, 'nlsparams']])`. I know this looks a bit formidable, but actually it's not so bad. Again, the items in brackets are optional. So any of the following is an acceptable use of `TO_CHAR`: `TO_CHAR(date)`, `TO_CHAR(date, fmt)`, and `TO_CHAR(date, fmt, 'nlsparams')`. If you want the formatted date to use a different language than the default language, you can specify `nlsparams` as `NLS_DATE_LANGUAGE = language` (e.g., `NLS_DATE_LANGUAGE = FRENCH`). This option only finds specialized use.

However, the `fmt` option is used frequently. It refers to the format used when converting the date to a character value. In Figure 9-4, `TO_CHAR` displays birth dates using the format model “Mon YYYY.” This format model abbreviates the month as Mon and displays the year as a four digit value. If you omit the format model in `TO_CHAR`, the conversion uses the default date format set up for your database. For more information on `TO_CHAR`, see conversion functions in Appendix

D (page 289). Documentation on the date format models appears in Appendix D in Figure D-8 on page 294.

- The age expression uses the date function called `SYSDATE`. This returns the system date, which is the date when the query actually was run. Sounds simple, but there is something very important about Oracle dates—they all contain a time component. When dates display with the default format, you don't actually see the time component, but it's there. For example, if we formatted `SYSDATE` as `TO_CHAR(SYSDATE, 'dd-MON-yyyy::hh24:mi:ss')`, we'd see something like 15-DEC-1999::10:24:15. In this case the query was run on December 15, 1999 at 10 hours, 24 minutes, and 15 seconds after midnight. The `hh24` formats hours from 0 to 23, the `mi` formats minutes from 0 to 59, and the `ss` formats seconds from 0 to 59.

It is real easy to stub your toe with date comparisons. I can almost guarantee you that sooner or later you will write a query like this: `WHERE some_transaction_date = '01-APR-1999'`. And you'll wonder what happened to all the transactions you know occurred on that date. It's the time. None of the transactions occurred at 0 hours, 0 minutes, and 0 seconds after midnight, so they fall through the criteria!

- In the age calculation, the expression `(SYSDATE - demog_birthdate)` uses Oracle date arithmetic to return the number of days between `SYSDATE` and a birthdate. Dividing this by 365.25 produces an age in years. For more information on `SYSDATE`, see date functions in Appendix D (page 284).
- The `TRUNC` function then truncates the age to a specified number of decimal places. Its general format is `TRUNC(n [, m])`, where `n` is a number and `m` is the number of decimal places. If `m` is included, you must separate `n` and `m` by a comma. If you omit `m`, the default value is 0. In Figure 9-4, age is truncated to an integer. For more information on the `TRUNC` function, see number functions in Appendix D (page 272).
- The `ORDER BY` clause uses positional notation to sort the report in descending order by age and, within a specific age, by the name.

■ Exercises

1. The Astronomy department offers a course called Celestial Navigation, where the subject code is AST and the course number is 102. What value would be returned for this course if the `SELECT` clause is `SELECT catalog_subject_code || '*' || catalog_course_number || ':' || catalog_description`?
2. What value would be returned from each of the following?
 - a) `TRUNC(15.89,1)`
 - b) `TRUNC(15.89,0)`
 - c) `TRUNC(15.89,-1)`
 - d) `TRUNC(15.89)`
3. If `SYSDATE` is January 11, 1999 at 7 hours, 29 minutes, and 15 seconds past midnight, what value is returned from `TO_CHAR(SYSDATE, 'dd-MON-yyyy::hh24:mi:ss')`?
4. Suppose that someone objects to the null values that appear in a report if the sex of a person is unknown. Create an expression using the `DECODE` function that would substitute "Unknown" for null values in `demog_sex`. Then write a second expression using the `NVL` function that accomplishes the same thing. See miscellaneous functions in Appendix D (page 298) for a description of `NVL`.

