

## Chapter 8 Statistics



SQL was not designed as a statistical analysis tool, and it is generally not equipped to handle this type of analysis. In almost every instance when statistical analysis is required, you should look to other tools. Use SQL as a tool to extract and construct a dataset that can then be used in a statistical package like SAS or SPSS.

Having said this, however, there are some statistically related questions that arise frequently among people writing ad hoc queries. For example, Oracle includes a few tantalizing group functions that perform basic statistical operations such as finding the minimum, maximum, and average. But other measures not included as group functions often would prove useful; measures of central tendency like the median or measures of spread like the interquartile range come to mind. Does this mean that every time you want to find the median of a group of values you must extract a data set to a statistical package? Some basic set of statistical functions would prove extremely useful to query writers. This chapter discusses two additions to the group functions provided by Oracle.

This chapter also discusses several ways of sampling data. You might, for example, be asked to draw samples from a population to receive different types of mailings where later analysis will determine the effectiveness of the various mailings. In some situations, a simple systematic sample (i.e., every *N*th row) might work; in other situations, you might need to draw random samples. This chapter discusses how to incorporate both types of samples into your query work. The random number generator included in this chapter improves on most system-supplied generators and should serve adequately for the majority of query applications.

## Samples

Sampling in ad hoc queries usually has an applied focus. You can use samples while debugging to limit the result set or to examine selected rows in more depth. Query writers also sample when they provide subsets of data to government agencies or when they're asked to participate in market research studies. This section discusses several sampling techniques.

### How do I select the Nth row from a query?

I want to select just one row from a query. If I use `WHERE ROWNUM = 1`, I get the same row returned each time. How can I vary the row returned?

#### Approach 1

Let's assume that ordering of the rows is unimportant (the next section relaxes this restriction). Figure 8-1 shows unsorted *NAME* data.

id	Last	rownum
@505	Carrington	1
@133	O'Leary	2
@330	Healey	3
@226	Sacco	4
@409	Alvarez	5
@925	Appiah	6

Figure 8-1: Unsorted *NAME* data.

Suppose you want to examine `ROWNUM 5`. Using `WHERE ROWNUM = 5` returns no rows. By including a `GROUP BY`, however, you can then use `HAVING` to limit the query to the desired `ROWNUM` (see Figure 8-2).

```
SELECT name_id "id", name_Last "Last", ROWNUM "rownum"
FROM name
WHERE name_seriesno = name_now(name_id)
GROUP BY name_id, name_Last, ROWNUM
HAVING ROWNUM = 5;
```

id	Last	rownum
@409	Alvarez	5

Figure 8-2: SQL query that samples one row from unsorted data.

#### Comments

- ▶ Sample one `ROWNUM` using `GROUP BY` and `HAVING` clauses.

## Approach 2

PL/SQL cursors provide a direct solution to this problem. Figure 8-3 illustrates one approach, using a PL/SQL cursor to fetch the *N*th row and a bind variable to save the ROWID. A query then uses this bind variable.

```

SET TRANSACTION READ ONLY
VARIABLE rown VARCHAR2(20)
DECLARE
  i NUMBER;
  CURSOR main_cursor IS
    SELECT ROWIDTOCHAR(ROWID)
    FROM name
    WHERE name_seriesno = name_now(name_id);
BEGIN
  OPEN main_cursor;
  FOR i IN 1 .. &number_of_row
  LOOP
    FETCH main_cursor INTO :rown;
  END LOOP;
  CLOSE main_cursor;
END;
/
SELECT name_id "id", name_last "last"
FROM name
WHERE name_rowid = CHARTOROWID(:rown);

id      last
-----
@409   Alvarez

```

Figure 8-3: PL/SQL approach that selects one row from unsorted data.

### Comments

- ▶ A bind variable called `rown` is created to store the ROWID of the *N*th row fetched from the cursor. The value of *N* is entered by the user at runtime (as `&number_of_row`).
- ▶ The query uses the bind variable to return only one row.

### See Also

1. Chapter 9 (Table 9-14 on page 283) shows results of performance tests for the SQL and PL/SQL approaches to sampling the *N*th row.
2. `ROWNUM` is a beguiling SQL keyword often misused. See, for example, Chapter 6 on page 202.
3. Use the `SET TRANSACTION` command to prevent database changes that occur between the two queries from affecting the results. For a discussion, see Chapter 2 on page 46.
4. Chapter 1 (Figure 1-9 on page 13) lists the PL/SQL for `name_now`.

*How do I select the Nth row from sorted data?*

I'd like to select just one row from a query where the data are first sorted. How can I do this?

**Approach**

Adding an `ORDER BY` to the PL/SQL cursor shown in Figure 8-3 on page 235 lets you easily sort and then select a single row. This is definitely the approach to take.

If you insist on a pure SQL solution, however, things become trickier. You cannot simply add an `ORDER BY` to the SQL shown in Figure 8-2 on page 234 because `ROWNUM` is computed based on the order that data are retrieved from the table *before* the `ORDER BY` takes effect. For example, if you added an `ORDER BY name_last` to the query in Figure 8-2, the query would still return (@409, Alvarez) as `ROWNUM 5`.

You need some way to retrieve sorted data. Using an appropriate index is one method. Figure 8-4 illustrates a second method that makes use of the fact that a `UNION` performs an implicit sort.

```

SELECT
  name_id "id",
  name_last "last",
  ROWNUM "rownum"
FROM
  name
WHERE
  name_seriesno = name_now(name_id)
  AND (name_last, name_id) IN
  (SELECT name_last, name_id
   FROM name
   WHERE name_seriesno = name_now(name_id)
   UNION
   SELECT 'x', 'y' FROM DUAL WHERE 1=2)
GROUP BY
  name_id,
  name_last,
  ROWNUM
HAVING
  ROWNUM = 5;

id      last          rownum
-----
@133   0'Leary          5

```

**Figure 8-4:** SQL query that samples one row from sorted data.

**Comments**

▶ Including `name_last` as the first element in the compound

subquery forces an implicit sort on this column. The portion of the compound subquery that selects from `DUAL` is a dummy query that never equates to `TRUE` but allows you to use `UNION` to force the sort. All very convoluted.

### See Also

1. Subqueries may not include the `ORDER BY` clause. This means that you cannot sort easily under certain circumstances, such as when you create a view or when you do an `INSERT`. Instead, query writers must fall back on side-effects like the implicit sort available in the `UNION` operation. For an example of a query that relies on the implicit sort of a `UNION` operation, see Chapter 6 (Figure 6-24 on page 203) for discussion about the top *N* report.
2. Figure 8-4 uses the `name_now` function. For the `PL/SQL` used to create `name_now`, see Chapter 1 (Figure 1-9 on page 13).

*How can I draw a systematic sample?*

I'd like to select every *N*th row from a query. For example, if I want a 10 percent sample, I'd select every tenth row. How can I construct a query that will do this?

**Approach 1**

By making only a minor modification to the SQL that samples one row from a query (see Figure 8-2 on page 234), you can systematically draw a sample of every *N*th row. Figure 8-5 shows how this works.

```

SELECT
  name_id "id",
  name_last "last",
  ROWNUM "rownum"
FROM
  name
WHERE
  name_seriesno = name_now(name_id)
GROUP BY
  name_id,
  name_last,
  ROWNUM
HAVING
  MOD(ROWNUM,2) = 0;

```

id	last	rownum
a133	O'Leary	2
a226	Sacco	4
a925	Appiah	6

**Figure 8-5:** SQL query that systematically samples every second row.

**Comments**

► The MOD function used in the HAVING clause retains every second row. You can select every *N*th row with MOD(ROWNUM, *N*) = 0; just substitute a value of your choice for *N*.

**Approach 2**

PL/SQL provides a simple, direct approach to systematic sampling. You merely open a cursor and output every *N*th row that's fetched. Figure 8-6 illustrates the general approach. In this case a report is prepared using the PL/SQL package called DBMS\_OUTPUT that sends output to the terminal. You also could modify the PL/SQL to insert data to a temporary table rather than output them to a report.

```

SET SERVEROUTPUT ON
DECLARE
  CURSOR main_cursor IS
    SELECT name_id, name_last, ROWNUM
    FROM name
    WHERE name_seriesno = name_now(name_id);
BEGIN
  DBMS_OUTPUT.PUT_LINE('id      last      rownum');
  DBMS_OUTPUT.PUT_LINE('-----');
  FOR name_rec IN main_cursor
  LOOP
    IF MOD(name_rec.ROWNUM, &every_nth_row) = 0 THEN
      DBMS_OUTPUT.PUT(RPAD(name_rec.name_id,6));
      DBMS_OUTPUT.PUT(RPAD(name_rec.name_last,16));
      DBMS_OUTPUT.PUT_LINE(LPAD(TO_CHAR(name_rec.ROWNUM,'999'),10));
    END IF;
  END LOOP;
END;
/

id      last      rownum
-----
@133    O'Leary      2
@226    Sacco        4
@925    Appiah       6

```

Figure 8-6: PL/SQL that systematically samples every Nth row.

### Comments

- Enable the `SERVEROUTPUT` system variable to display output from the built-in PL/SQL package `DBMS_OUTPUT`.
- The `PUT` and `PUT_LINE` procedures from `DBMS_OUTPUT` are used to prepare the report.
- Every *N*th row is selected with a `MOD` function. You can vary *N* at runtime by specifying a value for the substitution variable `&every_nth_row`.

### See Also

1. Performance comparisons for the SQL and PL/SQL approaches to systematic sampling appear in Chapter 9 (Table 9-15 on page 285).
2. The PL/SQL for the `name_now` function appears in Chapter 1 (Figure 1-9 on page 13).

*How do I randomly sample rows?*

I'd like to randomly sample rows from a query. How can I construct a query that provides this kind of sampling?

**Approach**

Oracle doesn't supply a function that produces uniform random deviates, which is a prerequisite for random sampling. Thus it's first necessary to create a random number generator. Figure 8-7 implements the so-called minimal standard random number generator of Park and Miller. This generator improves on the technique (called the *linear congruential method*) employed in many system-supplied random generators by reducing the sequential correlation possible on successive calls. Since it was first proposed in 1969, the minimal standard generator has passed all new theoretical tests and has become increasingly used. More complex generators exist, but Press et al. (1992) call the minimal standard generator "satisfactory for the majority of applications."

```
-- Implements the Park and Miller minimal standard random number
-- generator. See William H. Press et.al., Numerical Recipes in C:
-- The Art of Scientific Computing (Cambridge University Press, 1992).
--
-- Note: assumes that the first time getrand is called, you seed it
-- by setting previous_random_number to a positive integer
-- eg, to_number(to_char(sysdate,'dddhhsssss'))
--
-- Sample usage: execute getrand (:prev_random, :rnddev)
--
CREATE OR REPLACE PROCEDURE getrand
  (previous_random_number IN OUT NUMBER, random_deviate OUT NUMBER)
AS
  a NUMBER := 16807;      -- POWER(7,5)
  m NUMBER := 2147483647; -- POWER(2,31) - 1
  q NUMBER := 127773;    -- from Schrage algorithm
  r NUMBER := 2836;     -- from Schrage algorithm
  k NUMBER := NULL;
  rnd NUMBER := previous_random_number;

BEGIN
  -- generate uniform random deviate between 0.0 and 1.0
  --
  k := TRUNC(rnd/q);
  rnd := a*(rnd - k*q) - r*k;
  IF rnd < 0 THEN
    rnd := rnd + m;
  END IF;
  random_deviate := rnd/m;
  previous_random_number := rnd;
END;
/
```

**Figure 8-7:** Minimal standard random number generator.

Once you have a random number generator, it can be used to randomly select rows from a query. Figure 8-8 illustrates how to randomly select a single row from a query.

```

SET TRANSACTION READ ONLY;
VARIABLE prev_random NUMBER
VARIABLE rnddev NUMBER
VARIABLE maxrownum NUMBER

DECLARE
  CURSOR main_cursor IS
    SELECT COUNT(*) FROM name
    WHERE name_seriesno = name_now(name_id);
BEGIN
  :prev_random := TO_NUMBER(TO_CHAR(SYSDATE,'ddhhsssss'));
  OPEN main_cursor;
  FETCH main_cursor INTO :maxrownum;
  CLOSE main_cursor;
END;
/

EXECUTE getrand(:prev_random, :rnddev)

SELECT
  name_id "id",
  name_last "last",
  ROWNUM "rownum"
FROM
  name
WHERE
  name_seriesno = name_now(name_id)
GROUP BY
  name_id,
  name_last,
  rownum
HAVING
  ROWNUM = TRUNC(:rnddev * :maxrownum) + 1;

id      last              rownum
-----
@226   Sacco                4

```

Figure 8-8: Randomly selecting one row from an SQL query.

### Comments

▶ The first time in a single session that you use the random number procedure, you'll need to declare two bind variables that will hold a random number and the random deviate (i.e., the random number constrained to the range 0.0 to 1.0). Also declare a bind variable maxrownum to contain the number of rows returned by the query. The value of this variable gets determined in the PL/SQL block.

▶ The first time in a single session that you use the random

number generator you'll also need to seed the generator. Here the date and time are formatted to produce this seed.

▶ The minimal standard random number generator uses `:prev_random` as input. It produces a uniform random deviate output as `:rnddev`. It also produces a new random number that's output as `:prev_random` for use in subsequent calls to `getrand`.

▶ The query uses the random deviate `:rnddev` and the number of rows in the query (`:maxrownum`) to select a single row for display.

You also can use the random number procedure to randomly select a sample for statistical analysis or other research uses. For example, you might want to randomly select 10% of your customers to receive a newly redesigned promotional piece so that you can compare its effectiveness against the previous promotion.

Figure 8-9 illustrates how to use the `getrand` procedure to produce a random sample. The output in this case is a report, but it could just as easily be an ASCII flat file suitable for mail merges or statistical analysis.

#### Comments

▶ This PL/SQL program uses a temporary table called *SAMPLE* that includes only the single data column `sample_rownum`. The table gets populated with randomly generated numbers in the range from 1 to `max_number_rows` (i.e., `COUNT(*)` from the query in question). The program later loops through the `report_cursor` and retains for display only those rows whose `ROWNUM` exists in the *SAMPLE* table.

#### See Also

1. Chapter 6 on page 212 describes how to produce an ASCII flat file from a query.
2. The `name_now` function used in this section appears in Chapter 1 (Figure 1-9 on page 13).

```

SET SERVEROUTPUT ON
DECLARE
  i NUMBER;
  max_number_rows NUMBER;
  sample_size NUMBER := 2;
  get_rownum NUMBER;
  CURSOR maxrow_cursor IS
    SELECT COUNT(*) FROM name
    WHERE name_seriesno = name_now(name_id);
  CURSOR report_cursor IS
    SELECT name_id, name_last, ROWNUM
    FROM name
    WHERE name_seriesno = name_now(name_id);
  CURSOR sample_cursor IS
    SELECT sample_rownum FROM sample ORDER BY 1;
BEGIN
  -- get number of rows in query
  OPEN maxrow_cursor;
  FETCH maxrow_cursor INTO max_number_rows;
  CLOSE maxrow_cursor;

  -- draw sample
  FOR i IN 1 .. sample_size
  LOOP
    getrand(:prev_random, :rnddev);
    INSERT INTO sample VALUES (TRUNC(:rnddev * max_number_rows)+1);
  END LOOP;

  -- prepare report
  DBMS_OUTPUT.PUT_LINE('id      last                rownum');
  DBMS_OUTPUT.PUT_LINE('-----');

  OPEN sample_cursor;
  FETCH sample_cursor INTO get_rownum;

  FOR report_rec IN report_cursor
  LOOP
    IF report_rec.ROWNUM = get_rownum THEN
      DBMS_OUTPUT.PUT(RPAD(report_rec.name_id,6));
      DBMS_OUTPUT.PUT(RPAD(report_rec.name_last,16));
      DBMS_OUTPUT.PUT_LINE(LPAD(TO_CHAR(report_rec.ROWNUM,'999'),10));
      FETCH sample_cursor INTO get_rownum;
      EXIT WHEN sample_cursor%NOTFOUND;
    END IF;
  END LOOP;
  CLOSE sample_cursor;
END;
/

id      last                rownum
-----
@133    O'Leary                2
@226    Sacco                  4
    
```

Figure 8-9: Randomly selecting a sample of report rows.

## Constructed Functions

Oracle group functions provide several simple univariate measures such as the minimum, maximum, and average of a distribution. Often, however, other measures of a distribution would be useful. For example, in many cases the median is preferable to the mean as a measure of central tendency. While most statistical analyses should always be done with a statistical package instead of SQL, this section discusses two types of queries that extend the basic group functions provided by Oracle.

### *How do I find the median?*

Oracle provides a function `AVG` that computes the average of data values for a group, but there is no function that computes the median. How can I find the median of a distribution of data values?

### Approach

For many purposes the median is a better indicator of central tendency than the average. But it is more difficult to compute because the rows must be sorted first. With an odd number of data values, the median is that single value where the number of values less than it equals the number of values greater than it. For example, in a group of five data values, the median is the third value in rank order (i.e., two values are greater than it and two values are less than it). With an even number of data values, the median is determined by averaging the two middle values. For example, in a group of six data values, the median is the average of the third and fourth items in rank order.

Figure 8-10 shows training fees in descending order. There are 10 values; the median is the average of items 5 and 6 (i.e., 500 and 600).

id	fee
a133	100
a226	200
a133	325
a226	400
a505	500
a925	610
a330	740
a925	740
a226	1200
a505	1500

Figure 8-10: Median of 10 student training fees.

**Comments**

▶ This group of training fees consists of 10 values. With an even number of values, the median is computed by rank ordering the data values and averaging the two middle values. In this case the two middle values are 500 and 610, making the median 555.

A pure SQL solution for the median has been proposed by several people. Celko (1995) provides a good discussion of these solutions. Many approaches rely on features available in SQL-92 and not yet available in commercial implementations of SQL. The solution shown in Figure 8-11 uses only standard features in Oracle7 Release 7.2. It does rely on a trick, however, using the implicit sorting that occurs with UNION compound queries to first sort the data values prior to computing the median.

```

BREAK ON REPORT
COMPUTE AVG LABEL 'Median' OF fee ON REPORT
COLUMN fee HEADING 'values|averaged|in median'
SELECT
  ROWNUM "rownum",
  student_total_fee "fee"
FROM
  (SELECT student_total_fee FROM student
   WHERE student_total_fee IS NOT NULL
   UNION
   SELECT 1 FROM DUAL WHERE 1=2)
GROUP BY
  student_total_fee,
  ROWNUM
HAVING
  ROWNUM >=
  (SELECT DECODE(MOD(total_freq,2),
    1,TRUNC(total_freq/2 + 1),
    0,TRUNC(total_freq/2))
  FROM
    (SELECT COUNT(*) AS total_freq FROM student
     WHERE student_total_fee IS NOT NULL))
  AND ROWNUM <=
  (SELECT DECODE(MOD(total_freq,2),
    1,TRUNC(total_freq/2 + 1),
    0,TRUNC(total_freq/2) + 1)
  FROM
    (SELECT COUNT(*) AS total_freq FROM student
     WHERE student_total_fee IS NOT NULL));

```

rownum	values averaged in median
5	500
6	610
Median	555

Figure 8-11: SQL query that computes a median.

**Comments**

- ▶ The main query contains a subquery in the `FROM` clause. This subquery is actually a `UNION`d compound query that implicitly sorts the data values for `student_total_fee`. Note that the second query in the `UNION` is a dummy that never equates to `TRUE`. It is included only to allow `UNION` to do an implicit sort.
- ▶ A `HAVING` clause selects the `ROWNUMS` from the sorted data values that will be used to compute the median. If there are an odd number of data values, then the median is one of the values in the group. However, if there are an even number of data values, the median must be computed by averaging the middle two values. The `DECODE` determines the correct `ROWNUMS` to use for the median based on whether there are an odd or even number of data values in the group.
- ▶ The total number of data values in the group is determined in this subquery. Note that it again illustrates the use of a subquery in the `FROM` clause.

You can simplify the solution in Figure 8-11 by creating a bind variable that counts the number of data values in the group. Figure 8-12 illustrates how this works.

**Comments**

- ▶ A PL/SQL cursor determines the total number of data values. This number then gets stored in a bind variable called `total_freq`.
- ▶ The `HAVING` clause is simplified compared with Figure 8-12 because it uses the bind variable `total_freq` and a simple query against `DUAL` to establish the `ROWNUMS` to include in the median calculation.

**See Also**

1. Sorting is a side-effect of the `UNION` set operation. Oracle does not recommend that you rely upon side-effects when writing ad hoc queries, in case they someday change the algorithm and the side-effect disappears. Until then, however, `UNION` does allow you to sort when you might otherwise be precluded from doing so. For examples, see Figure 8-4 in this chapter and Chapter 6 (Figure 6-24 on page 203).
2. With Oracle7 Release 7.2 and later releases, you can use

```

SET TRANSACTION READ ONLY;
VARIABLE total_freq NUMBER
DECLARE
  CURSOR c1 IS
    (SELECT COUNT(*) FROM student
     WHERE student_total_fee IS NOT NULL);
BEGIN
  OPEN c1;
  FETCH c1 INTO :total_freq;
  CLOSE c1;
END;
/
BREAK ON REPORT
COMPUTE AVG LABEL 'Median' OF fee ON REPORT
COLUMN fee HEADING 'values|averaged|in median'
SELECT
  ROWNUM "rownum",
  student_total_fee "fee"
FROM
  (SELECT student_total_fee FROM student
   WHERE student_total_fee IS NOT NULL
   UNION
   SELECT 1 FROM DUAL WHERE 1=2)
GROUP BY
  student_total_fee,
  ROWNUM
HAVING
  ROWNUM >=
    (SELECT DECODE(MOD(:total_freq,2),
     1,TRUNC(:total_freq/2 + 1),
     0,TRUNC(:total_freq/2))
   FROM DUAL)
  AND ROWNUM <=
    (SELECT DECODE(MOD(:total_freq,2),
     1,TRUNC(:total_freq/2 + 1),
     0,TRUNC(:total_freq/2) + 1)
   FROM DUAL);

```

rownum	values averaged in median
5	500
6	610
Median	555

Figure 8-12: Simplified median calculation using SQL and PL/SQL.

subqueries in the FROM clause. This makes a very nice addition to your query writing toolkit, because it allows you to create data views at runtime without using the CREATE VIEW command. For another example, see Chapter 10 (Figure 10-11 on page 301), or check the index under the heading FROM clause.

3. The DECODE function can be put to good use when you need

IF . . . THEN . . . ELSE logic and don't want to use a procedural language like PL/SQL. See the discussion beginning in Chapter 3 on page 92 for several uses of `DECODE`.

4. Computing a median with the SQL or PL/SQL in this section is pretty messy. Until Oracle gives us a median group function, save the logic in Figure 8-11 or Figure 8-12 as a SQL segment that you can reuse by cutting and pasting the next time you need it.

5. Performance comparisons between the two approaches used in this section to compute a median appear in Chapter 9 (Table 9-16 on page 287).

*How do I find the mode?*

I'd like to find the modal age (i.e., the most common age) for a group of undergraduate students. Is this possible with SQL?

**Approach**

Figure 8-13 shows a frequency distribution of the ages for 100 students.

age	count
16	2
17	7
18	16
19	17
20	19
21	15
22	12
23	9
24	3
sum	100

**Figure 8-13:** Frequency distribution of ages for 100 students.

**Comments**

▶ The age with the largest frequency count is the modal age. In this case the mode occurs at age 20.

Determining the modal age with SQL is straightforward. The traditional method uses a subquery in the HAVING clause to select the age with the maximum frequency count. Figure 8-14 illustrates this approach.

```

COLUMN age HEADING 'modal|age'
SELECT
    undergrad_age "age",
    COUNT(*) "count"
FROM
    undergrad
GROUP BY
    undergrad_age
HAVING COUNT(*) =
    (SELECT MAX(COUNT(*))
     FROM undergrad
     GROUP BY undergrad_age);

```

modal age	count
20	19

**Figure 8-14:** SQL query using a HAVING subquery to determine the mode.

**Comments**

▶ The main query groups rows by age. The HAVING clause in the main query retains for display only the single row with the maximum frequency count (i.e., the mode).

You also can find the mode of a distribution using a FROM clause subquery to create a runtime data view. When SQL-92 compliance is reached, a very simple query will produce the mode. For now, however, this approach still requires a subquery. Figure 8-15 illustrates how this works.

```

COLUMN age HEADING 'modal|age'
SELECT
    undergrad_age "age",
    freq "count"
FROM
    (SELECT undergrad_age, COUNT(*) AS freq
     FROM undergrad
     GROUP BY undergrad_age)
WHERE
    freq =
    (SELECT MAX(freq1)
     FROM
        (SELECT COUNT(*) AS freq1
         FROM undergrad
         GROUP BY undergrad_age));

```

modal	
age	count
20	19

**Figure 8-15:** SQL query using a FROM clause subquery to determine mode.

**Comments**

▶ The FROM clause in the main query contains a subquery. In this case the subquery duplicates the effect of a view showing the frequency distribution by age. Note that one column in this on-the-fly view is given the alias “freq” and that this gets used in the WHERE clause of the main query.

▶ The subquery in the WHERE clause selects the row in the main query with the maximum frequency count (i.e., the mode). In the process, however, a second FROM clause subquery was used. This makes for a convoluted query. In SQL-92 the WHERE clause can be replaced with HAVING freq = MAX(freq).

**See Also**

1. A FROM clause subquery also was used in Figure 8-12 on page 247 to compute a median. For another example, also see Chapter 10 (Figure 10-11 on page 301).

