SELECT Statement: The GROUP BY Clause in SQL

By Rick F. van der Lans

Date: Nov 17, 2006

Sample Chapter is provided courtesy of Addison-Wesley Professional.

Return to the article

Rick F. van der Lans explains how SQL's GROUP BY clause groups rows on the basis of similarities between them. He goes on to explain the use of adding aggregation functions to display summations, averages, frequencies, and subtotals.

10.1 Introduction

The `GROUP BY` clause groups rows on the basis of similarities between them. You could, for example, group all the rows in the `PLAYERS` table on the basis of the place of residence; the result would be one group of players per town. From there you could query how many players there are in each group. The question that is actually answered is then: How many players live in each town? Other examples follow: How many matches have been played per team, and how much has been incurred in penalties per player? In short, the `GROUP BY` clause is frequently used to formulate questions based on the word per.

By adding aggregation functions, such as `COUNT` and `SUM`, to a select block with the use of a `GROUP BY` clause, data can be *aggregated*. These functions owe their name to this. Aggregation means that you ask not for the individual values, but for summations, averages, frequencies, and subtotals.

```
<group by clause> ::=
   GROUP BY <group by specification list>
            [ WITH { ROLLUP | CUBE } ]

<group by specification list> ::=
   <group by specification> [ { , <group by specification> }... ]

<group by specification> ::=
   <group by expression>         |
   <grouping sets specification> |
   <rollup specification>

<grouping sets specification> ::=
   GROUPING SETS ( <grouping sets specification list> )
<grouping sets specification list> ::=
   <grouping sets specification>
   [ { , <grouping sets specification> }... ]

<grouping sets specification> ::=
   <group by expression>                 |
   <rollup specification>                |
   ( <grouping sets specification list> )

<rollup specification> ::=
   ROLLUP ( <group by expression list> ) |
   CUBE ( <group by expression list> )   |
   ( )

<group by expression> ::= <scalar expression>
```

10.2 Grouping on One Column

The simplest form of the `GROUP BY` clause is the one in which only one column is grouped. In the previous chapters, we gave several examples of statements with such a `GROUP BY` clause. For the sake of clarity, we specify several other examples in this section.

Example 10.1. Get all the different town names from the `PLAYERS` table.

```
SELECT   TOWN
FROM     PLAYERS
GROUP BY TOWN
```

The intermediate result from the `GROUP BY` clause could look like this:

```
TOWN       PLAYERNO                    NAME
---------  --------------------------  ---------------------
Stratford  {6, 83, 2, 7, 57, 39, 100}  {Parmenter, Hope, ...}
Midhurst   {28}                        {Collins}
Inglewood  {44, 8}                     {Baker, Newcastle}
Plymouth   {112}                       {Bailey}
Douglas    {95}                        {Miller}
Eltham     {27, 104}                   {Collins, Moorman}
```

Explanation: All rows with the same `TOWN` form one group. Each row in the intermediate result has one value in the `TOWN` column, whereas all other columns can contain multiple values. To indicate that these columns are special, the values are placed between brackets. We show those columns in this way for illustrative purposes only; you should realize that SQL probably would solve this internally in a different way. Furthermore, these two columns *cannot* be presented like this. In fact, a column that is not grouped is completely omitted from the end result, but we return to this later in the chapter.

The end result of the statement is:

```
TOWN
---------
Stratford
Midhurst
Inglewood
Plymouth
Douglas
Eltham
```

A frequently used term in this particular context is *grouping*. The `GROUP BY` clause in the previous statement has one grouping, which consists of only one column: the `TOWN` column. In this chapter, we sometimes represent this as follows: The result is grouped by `[TOWN]`. Later in this chapter, we give examples of groupings with multiple columns and `GROUP BY` clauses consisting of multiple groupings.

We could have solved the earlier question more easily by leaving out the `GROUP BY` clause and adding `DISTINCT` to the `SELECT` clause (work this out for yourself). Using the `GROUP BY` clause becomes interesting when we extend the `SELECT` clause with aggregation functions.

Example 10.2. For each town, find the number of players.

```
SELECT   TOWN, COUNT(*)
FROM     PLAYERS
GROUP BY TOWN
```

The result is:

```
TOWN       COUNT(*)
---------  --------
Stratford         7
Midhurst          1
Inglewood         2
Plymouth          1
Douglas           1
Eltham            2
```

Explanation: In this statement, the result is grouped by [TOWN]. The COUNT(*) function is now executed against each grouped row instead of against all rows. In other words, the function `COUNT(*)` is calculated for each grouped row (for each town).

In this result, it is obvious that the data is aggregated. The individual data of players cannot be displayed anymore, and the data is aggregated by `TOWN`. Or the aggregation level of this result is `TOWN`.

Example 10.3. For each team, get the team number, the number of matches that has been played for that team, and the total number of sets won.

```
SELECT   TEAMNO, COUNT(*), SUM(WON)
FROM     MATCHES
GROUP BY TEAMNO
```

The result is:

```
TEAMNO  COUNT(*)  SUM(WON)
------  --------  --------
     1         8        15
     2         5         9
```

Explanation: This statement contains one grouping consisting of the `TEAMNO` column.

Example 10.4. For each team that is captained by a player resident in Eltham, get the team number and the number of matches that has been played for that team.

```
SELECT   TEAMNO, COUNT(*)
FROM     MATCHES
WHERE    TEAMNO IN
        (SELECT   TEAMNO
         FROM     TEAMS INNER JOIN PLAYERS
                  ON TEAMS.PLAYERNO = PLAYERS.PLAYERNO
         WHERE    TOWN = 'Eltham')
GROUP BY TEAMNO
```

The result is:

```
TEAMNO  COUNT(*)
------  --------
     2         5
```

The column on which the result has been grouped might also appear in the `SELECT` clause as a parameter within an aggregation function. This does not happen often, but it is allowed.

Example 10.5. Get each different penalty amount, followed by the number of times that the amount occurs, in the `PENALTIES` table, and also show the result of that amount multiplied by the number.

```
SELECT   AMOUNT, COUNT(*), SUM(AMOUNT)
FROM     PENALTIES
GROUP BY AMOUNT
```

The `PENALTIES` table is grouped on the `AMOUNT` column first. The intermediate result could be presented as follows:

```
PAYMENTNO  PLAYERNO  PAYMENT_DATE              AMOUNT
---------  --------  ------------------------  ------
{5, 6}     {44, 8}   {1980-12-08, 1980-12-08}  25.00
{7}        {44}      {1982-12-30}              30.00
{4}        {104}     {1984-12-08}              50.00
{2, 8}     {44, 27}  {1981-05-05, 1984-11-12}  75.00
{1, 3}     {6, 27}   {1980-12-08, 1983-09-10} 100.00
```

Again, the values of the columns that are not grouped are placed between brackets, and the `AMOUNT` column shows only one value. However, that is not entirely correct. Behind the scenes, SQL also creates a group for this column. So, the intermediate result should, in fact, be presented as follows:

```
PAYMENTNO  PLAYERNO  PAYMENT_DATE              AMOUNT
---------  --------  ------------------------  --------------
{5, 6}     {44, 8}   {1980-12-08, 1980-12-08}  {25.00, 25.00}
{7}        {44}      {1982-12-30}              {30.00}
{4}        {104}     {1984-12-08}              {50.00}
{2, 8}     {44, 27}  {1981-05-05, 1984-11-12}  {75.00, 75.00}
{1, 3}     {6, 27}   {1980-12-08, 1983-09-10}  {100.00, 100.00}
```

The values in the `AMOUNT` column are also represented as a group now. Of course, only equal values appear in each group. And because it is a group, aggregation functions can be used.

The result is:

```
AMOUNT  COUNT(*)  SUM(AMOUNT)
------  --------  -----------
```

```
 25.00         2        50.00
 30.00         1        30.00
 50.00         1        50.00
 75.00         2       150.00
100.00         2       200.00
```

However, in this book, we do not present the values of the grouped columns between brackets.

Exercise 10.1: Show the different years in which players joined the club; use the `PLAYERS` table.

Exercise 10.2: For each year, show the number of players who joined the club.

Exercise 10.3: For each player who has incurred at least one penalty, give the player number, the average penalty amount, and the number of penalties.

Exercise 10.4: For each team that has played in the first division, give the team number, the number of matches, and the total number of sets won.

10.3 Grouping on Two or More Columns

A `GROUP BY` clause can contain two or more columns—or, in other words, a grouping can consist of two or more columns. We illustrate this with two examples.

Example 10.6. For the `MATCHES` table, get all the different combinations of team numbers and player numbers.

```
SELECT   TEAMNO, PLAYERNO
FROM     MATCHES
GROUP BY TEAMNO, PLAYERNO
```

The result is grouped not on one column, but on two. All rows with the same team number and the same player number form a group.

The intermediate result from the `GROUP BY` clause is:

```
TEAMNO  PLAYERNO  MATCHNO    WON        LOST
------  --------  ---------  ---------  ---------
     1         2  {6}        {1}        {3}
     1         6  {1, 2, 3}  {3, 2, 3}  {1, 3, 0}
     1         8  {8}        {0}        {3}
     1        44  {4}        {3}        {2}
     1        57  {7}        {3}        {0}
     1        83  {5}        {0}        {3}
     2         8  {13}       {0}        {3}
     2        27  {9}        {3}        {2}
     2       104  {10}       {3}        {2}
     2       112  {11, 12}   {2, 1}     {3, 3}
```

The end result is:

```
TEAMNO  PLAYERNO
------  --------
     1         2
     1         6
     1         8
     1        44
     1        57
     1        83
     2         8
     2        27
     2       104
     2       112
```

The sequence of the columns in the `GROUP BY` clause has no effect on the end result of a statement. The following statement, therefore, is equivalent to the previous one:

```
SELECT   TEAMNO, PLAYERNO
FROM     MATCHES
GROUP BY PLAYERNO, TEAMNO
```

As an example, let us add some aggregation functions to the previous `SELECT` statement:

```
SELECT   TEAMNO, PLAYERNO, SUM(WON),
         COUNT(*), MIN(LOST)
```

```
FROM     MATCHES
GROUP BY TEAMNO, PLAYERNO
```

The result is:

```
TEAMNO  PLAYERNO  SUM(WON)  COUNT(*)  MIN(LOST)
------  --------  --------  --------  ---------
     1         2         1         1          3
     1         6         8         3          0
     1         8         0         1          3
     1        44         3         1          2
     1        57         3         1          0
     1        83         0         1          3
     2         8         0         1          3
     2        27         3         1          2
     2       104         3         1          2
     2       112         3         2          3
```

In this example, the grouping is equal to `[TEAMNO, PLAYERNO]` and the aggregation level of the result is the combination of team number with player number. This aggregation level is lower than that of a statement in which the grouping is equal to `[TEAMNO]` or `[TOWN]`.

Example 10.7. For each player who has ever incurred at least one penalty, get the player number, the name, and the total amount in penalties incurred.

```
SELECT   P.PLAYERNO, NAME, SUM(AMOUNT)
FROM     PLAYERS AS P INNER JOIN PENALTIES AS PEN
         ON P.PLAYERNO = PEN.PLAYERNO
GROUP BY P.PLAYERNO, NAME
```

The result is:

```
P.PLAYERNO  NAME       SUM(AMOUNT)
----------  ---------  -----------
         6  Parmenter       100.00
         8  Newcastle        25.00
        27  Collins         175.00
        44  Baker           130.00
       104  Moorman          50.00
```

Explanation: This example also has a grouping consisting of two columns. The statement would have given the same result if the `PEN.PLAYERNO` column had been included in the grouping. Work this out for yourself.

Exercise 10.5: For each combination of won–lost sets, get the number of matches won.

Exercise 10.6: For each combination of year–month, get the number of committee members who started in that year and that month.

Exercise 10.7: Group the matches on town of player and division of team, and get the sum of the sets won for each combination of town[nd]division.

Exercise 10.8: For each player who lives in Inglewood, get the name, initials, and number of penalties incurred by him or her.

Exercise 10.9: For each team, get the team number, the division, and the total number of sets won.

10.4 Grouping on Expressions

Up to now, we have shown only examples in which the result was grouped on one or more columns, but what happens when we group on expressions? Again, here are two examples.

Example 10.8. For each year present in the `PENALTIES` table, get the number of penalties paid.

```
SELECT   YEAR(PAYMENT_DATE), COUNT(*)
FROM     PENALTIES
GROUP BY YEAR(PAYMENT_DATE)
```

The intermediate result from the `GROUP BY` clause is:

```
YEAR(PAYMENT_DATE)  PAYMENTNO  PLAYERNO    PAYMENT_DATE  AMOUNT
```

```
------------------  ---------  ----------  ------------  --------
1980                {1, 5, 6}  {6, 44, 8}  {1980-12-08,   {100.00,
                                            1980-12-08,     25,00,
                                            1980-12-08}     25,00}
1981                {2}        {44}        {1981-05-05}    {75,00}
1982                {7}        {44}        {1982-12-30}    {30,00}
1983                {3}        {27}        {1983-09-10}   {100,00}
1984                {4, 8}     {104, 27}   {1984-12-08,    {50,00,
                                            1984-11-12}     75,00}
```

The result is:

```
YEAR(PAYMENT_DATE)  COUNT(*)
------------------  --------
1980                       3
1981                       1
1982                       1
1983                       1
1984                       2
```

Explanation: The result is now grouped on the values of the scalar expression `YEAR(PAYMENT_DATE)`. Rows for which the value of the expression `YEAR(PAYMENT_DATE)` is equal form a group.

Example 10.9. Group the players on the basis of their player numbers. Group 1 should contain the players with number 1 up to and including 24. Group 2 should contain the players with numbers 25 up to and including 49, and so on. For each group, get the number of players and the highest player number.

```
SELECT   TRUNCATE(PLAYERNO/25,0), COUNT(*), MAX(PLAYERNO)
FROM     PLAYERS
GROUP BY TRUNCATE(PLAYERNO/25,0)
```

The result is:

```
TRUNCATE(PLAYERNO/25,0)  COUNT(*)  MAX(PLAYERNO)
-----------------------  --------  -------------
                      0         4              8
                      1         4             44
                      2         1             57
                      3         2             95
                      4         3            112
```

The scalar expression on which is grouped can be rather complex. This can consist of system variables, functions and calculations. Even certain scalar subqueries are allowed. Section 10.7 gives a few examples.

Portability

Several SQL products do not allow you to group on compound expressions, but require on column specifications only. However, a comparable result can be obtained by using views; see Chapter 21, "Views."

Exercise 10.10: Group the players on the length of their names and get the number of players for each length.

Exercise 10.11: For each match, determine the difference between the number of sets won and lost, and group the matches on that difference.

10.5 Grouping of NULL Values

If grouping is required on a column that contains `NULL` values, all these `NULL` values form one group. When rows are grouped, `NULL` values are also considered to be equal. The reason is that, with a `GROUP BY`, a vertical comparison is applied. This is in accordance with the rules described in Section 9.5, in Chapter 9, "`SELECT` Statement: `SELECT` Clause and Aggregation Functions."

Example 10.10. Find the different league numbers.

```
SELECT   LEAGUENO
FROM     PLAYERS
GROUP BY LEAGUENO
```

The result is:

```
LEAGUENO
--------
1124
1319
1608
2411
2513
2983
6409
6524
7060
8467
?
```

Explanation: Players 7, 28, 39, and 95 do not have a league number and, therefore, form one group (the last row) in the end result.

10.6 General Rules for the GROUP BY Clause

This section describes a number of important rules that relate to select blocks with a `GROUP BY` clause.

Rule 1: In Section 9.6, in Chapter 9, we gave several rules for the use of aggregation functions in the `SELECT` clause. We now add the following rule: If a select block does have a `GROUP BY` clause, any column specification specified in the `SELECT` clause must exclusively occur as a parameter of an aggregated function or in the list of columns given in the `GROUP BY` clause, or in both.

Therefore, the following statement is incorrect because the `TOWN` column appears in the `SELECT` clause, yet it is *not* the parameter of an aggregation function and does not occur in the list of columns by which the result is grouped.

```
SELECT   TOWN, COUNT(*)
FROM     PLAYERS
GROUP BY PLAYERNO
```

The reason for this restriction is as follows. The result of an aggregation function always consists of one value for each group. The result of a column specification on which grouping is performed also always consists of one value per group. These results are compatible. In contrast, the result of a column specification on which *no* grouping is performed consists of a set of values. This would not be compatible with the results of the other expressions in the `SELECT` clause.

Rule 2: In most examples, the expressions that are used to form groups also occur in the `SELECT` clause. However, that is not necessary. An expression that occurs in the `GROUP BY` clause *can* appear in the `SELECT` clause.

Rule 3: An expression that is used to form groups can also occur in the `SELECT` clause within a compound expression. We give an example next.

Example 10.11. Get the list with the different penalty amounts in cents.

```
SELECT   CAST(AMOUNT * 100 AS SIGNED INTEGER)
         AS AMOUNT_IN_CENTS
FROM     PENALTIES
GROUP BY AMOUNT
```

The result is:

```
AMOUNT_IN_CENTS
---------------
           2500
           3000
           5000
           7500
          10000
```

Explanation: A grouping is performed on a simple expression consisting of a column name: `AMOUNT`. In the `SELECT` clause, that same `AMOUNT` column occurs within a compound expression. This is allowed.

This rule is followed by the fact that no matter how complex a compound expression is, if it occurs in a `GROUP BY` clause, it can be included in its entirety only in the `SELECT` clause. For example, if the compound expression `PLAYERNO * 2` occurs in a `GROUP BY` clause, the expressions `PLAYERNO * 2`, `(PLAYERNO * 2) - 100`, and `MOD(PLAYERNO * 2, 3) - 100` can occur in the `SELECT` clause. On the other hand, the expressions `PLAYERNO`, `2 * PLAYERNO`, `PLAYERNO * 100`, and `8 * PLAYERNO * 2` are not allowed.

Rule 4: If an expression occurs twice or more in a `GROUP BY` clause, double expressions are simply removed. The `GROUP BY` clause `GROUP BY TOWN, TOWN` is converted to `GROUP BY TOWN`. Also, `GROUP BY SUBSTR(TOWN,1,1), SEX, SUBSTR(TOWN,1,1)` is converted to `GROUP BY SUBSTR(TOWN,1,1), SEX`. Therefore, it has no use for double expressions.

Rule 5: In Section 9.4, in Chapter 9, we described the cases in which the use of `DISTINCT` in the `SELECT` clause is superfluous. The rules given in that section apply to `SELECT` statements without a `GROUP BY` clause. We add a rule for `SELECT` statements with a `GROUP BY` clause: `DISTINCT` (if used outside an aggregation function) that is superfluous when the `SELECT` clause includes all the columns specified in the `GROUP BY` clause. The `GROUP BY` clause groups the rows in such a way that the columns on which they are grouped no longer contain duplicate values.

Exercise 10.12: Describe why the following statements are incorrect:

1.
   ```
   SELECT    PLAYERNO, DIVISION
   FROM      TEAMS
   GROUP BY PLAYERNO
   ```

2.
   ```
   SELECT   SUBSTR(TOWN,1,1), NAME
   FROM     PLAYERS
   GROUP BY TOWN, SUBSTR(NAME,1,1)
   ```

3.
   ```
   SELECT   PLAYERNO * (AMOUNT + 100)
   FROM     PENAL TIES
   GROUP BY AMOUNT + 100
   ```

Exercise 10.13: In which of the following statements is `DISTINCT` superfluous?

1.
   ```
   SELECT   DISTINCT PLAYERNO
   FROM     TEAMS
   GROUP BY PLAYERNO
   ```

2.
   ```
   SELECT   DISTINCT COUNT(*)
   FROM     MATCHES
   GROUP BY TEAMNO
   ```

3.
   ```
   SELECT   DISTINCT COUNT(*)
   FROM     MATCHES
   WHERE    TEAMNO = 2
   GROUP BY TEAMNO
   ```

10.7 Complex Examples with GROUP BY

Here are several other examples to illustrate the extensive possibilities of the `GROUP BY` clause.

Example 10.12. What is the average total amount of penalties for players who live in Stratford and Inglewood?

```
SELECT  AVG(TOTAL)
FROM   (SELECT   PLAYERNO, SUM(AMOUNT) AS TOTAL
        FROM     PENALTIES
        GROUP BY PLAYERNO) AS TOTALS
WHERE   PLAYERNO IN
       (SELECT   PLAYERNO
        FROM     PLAYERS
        WHERE    TOWN = 'Stratford' OR TOWN = 'Inglewood')
```

The result is:

```
AVG(TOTAL)
----------
85
```

Explanation: The intermediate result of the subquery in the `FROM` clause is a table consisting of two columns, called `PLAYERNO` and `TOTAL`, and contains five rows (players 6, 8, 27, 44, and 104). This table is passed on to the `WHERE` clause, where a subquery selects players from Stratford and Inglewood (players 6, 8, and 44). Finally, the average is calculated in the `SELECT` clause of the column `TOTAL`.

Example 10.13. For each player who incurred penalties and is captain, get the player number, the name, the number of penalties that he or she incurred, and the number of teams that he or she captains.

```
SELECT   PLAYERS.PLAYERNO, NAME, NUMBER_OF_PENALTIES,
         NUMBER_OF_TEAMS
FROM     PLAYERS,
        (SELECT   PLAYERNO, COUNT(*) AS NUMBER_OF_PENALTIES
         FROM     PENALTIES
         GROUP BY PLAYERNO) AS NUMBER_PENALTIES,
        (SELECT   PLAYERNO, COUNT(*) AS NUMBER_OF_TEAMS
         FROM     TEAMS
         GROUP BY PLAYERNO) AS NUMBER_TEAMS
WHERE    PLAYERS.PLAYERNO = NUMBER_PENALTIES.PLAYERNO
AND      PLAYERS.PLAYERNO = NUMBER_TEAMS.PLAYERNO
```

The result is:

```
PLAYERNO  NAME       NUMBER_OF_PENALTIES  NUMBER_OF_TEAMS
--------  ---------  -------------------  ---------------
       6  Parmenter                    1                1
      27  Collins                      2                1
```

Explanation: The `FROM` clause contains two subqueries that both have a `GROUP BY` clause.

The previous statement could have been formulated more easily by including subqueries in the `SELECT` clause, which makes `GROUP BY` clauses no longer required; see the next example. Now, the only difference is that all players appear in the result.

```
SELECT  PLAYERS.PLAYERNO, NAME,
       (SELECT   COUNT(*)
        FROM     PENALTIES
        WHERE    PLAYERS.PLAYERNO =
                 PENALTIES.PLAYERNO) AS NUMBER_OF_PENALTIES,
       (SELECT   COUNT(*)
        FROM     TEAMS
        WHERE    PLAYERS.PLAYERNO =
                 TEAMS.PLAYERNO) AS NUMBER_OF_TEAMS
FROM    PLAYERS
```

Example 10.14. Get the player number and the total number of penalties for each player who played a match.

```
SELECT  DISTINCT M.PLAYERNO, NUMBERP
FROM    MATCHES AS M LEFT OUTER JOIN
          (SELECT   PLAYERNO, COUNT(*) AS NUMBERP
           FROM     PENALTIES
           GROUP BY PLAYERNO) AS NP
        ON M.PLAYERNO = NP.PLAYERNO
```

Explanation: In this statement, the subquery creates the following intermediate result (this is the `NP` table):

```
PLAYERNO  NUMBERP
--------  -------
       6        1
       8        1
      27        2
      44        3
     104        1
```

Next, this table is joined with the `MATCHES` table. We execute a left outer join, so no players disappear from this table. The final result is:

```
PLAYERNO  NUMBERP
--------  -------
       2        ?
       6        1
       8        1
      27        2
      44        3
      57        ?
      83        ?
     104        1
     112        ?
```

Example 10.15. Group the penalties on the basis of payment date. Group 1 should contain all penalties between January 1, 1980, and June 30, 1982; group 2 should contain all penalties between July 1, 1981, and December 31, 1982; and group 3 should contain all penalties between January 1, 1983, and December 31, 1984. Get for each group the sum of all penalties.

```
SELECT   GROUPS.PGROUP, SUM(P.AMOUNT)
FROM     PENALTIES AS P,
        (SELECT 1 AS PGROUP, '1980-01-01' AS START,
                '1981-06-30' AS END
         UNION
         SELECT 2, '1981-07-01', '1982-12-31'
         UNION
         SELECT 3, '1983-01-01', '1984-12-31') AS GROUPS
WHERE    P.PAYMENT_DATE BETWEEN START AND END
GROUP BY GROUPS.PGROUP
ORDER BY 1
```

The result is:

```
GROUP  SUM(P.AMOUNT)
-----  -------------
    1         225.00
    2          30.00
    3         225.00
```

Explanation: In the `FROM` clause, a new (virtual) table is created in which the three groups have been defined. This `GROUPS` table is joined with the `PENALTIES` table. A `BETWEEN` operator is used to join the two tables. If there are penalties that fall outside these groups with respect to payment date, they will not be included in the result.

Example 10.16. For each penalty, get the penalty amount plus the sum of that amount and the amounts of all penalties with a lower payment number (cumulative value).

```
SELECT   P1.PAYMENTNO, P1.AMOUNT, SUM(P2.AMOUNT)
FROM     PENALTIES AS P1, PENALTIES AS P2
WHERE    P1.PAYMENTNO >= P2. PAYMENTNO
GROUP BY P1. PAYMENTNO, P1.AMOUNT
ORDER BY P1. PAYMENTNO
```

For convenience, we assume that the `PENALTIES` table consists of the following three rows only (you can create this, too, by temporarily removing all penalties with a number greater than 3):

```
PAYMENTNO  PLAYERNO  PAYMENT_DATE  AMOUNT
---------  --------  ------------  ------

        1         6  1980-12-08       100
        2        44  1981-05-05        75
        3        27  1983-09-10       100
```

The desired result is:

```
PAYMENTNO  AMOUNT  SUM
---------  ------  ---
        1     100  100
        2      75  175
        3     100  275
```

The intermediate result of the `FROM` clause (we show only the columns `PAYMENTNO` and `AMOUNT`):

```
P1.PAYNO  P1.AMOUNT  P2.PAYNO  P2.AMOUNT
--------  ---------  --------  ---------
       1        100         1        100
       1        100         2         75
       1        100         3        100
       2         75         1        100
       2         75         2         75
       2         75         3        100
       3        100         1        100
       3        100         2         75
       3        100         3        100
```

The intermediate result of the `WHERE` clause:

```
P1.PAYNO  P1.AMOUNT  P2.PAYNO  P2.AMOUNT
--------  ---------  --------  ---------
       1        100         1        100
       2         75         1        100
       2         75         2         75
       3        100         1        100
       3        100         2         75
       3        100         3        100
```

The intermediate result of the `GROUP BY` clause:

```
P1.PAYNO  P1.AMOUNT  P2.PAYNO  P2.AMOUNT
--------  ---------  --------  --------------
       1        100  {1}       {100}
       2         75  {1, 2}    {100, 75}
       3        100  {1, 2, 3} {100, 75, 100}
```

The intermediate result of the `SELECT` clause:

```
P1.PAYNO  P1.AMOUNT  SUM(P2.AMOUNT)
--------  ---------  --------------
       1        100             100
       2         75             175
       3        100             275
```

This final result is equal to the desired table.

Most joins in this book and in reality are equi joins. Non-equi joins are rare. The previous statement is an example that shows that non-equi joins can be useful and that powerful statements can be formulated with them.

Example 10.17. For each penalty, get the payment number, the penalty amount, and the percentage that the amount forms of the sum of all amounts (again, we use the same `PENALTIES` table as in the previous example).

```
SELECT   P1.PAYMENTNO, P1.AMOUNT,
         (P1.AMOUNT * 100) / SUM(P2.AMOUNT)
FROM     PENALTIES AS P1, PENALTIES AS P2
GROUP BY P1.PAYMENTNO, P1.AMOUNT
ORDER BY P1.PAYMENTNO
```

The intermediate result of the `FROM` clause is equal to that of the previous example. However, the intermediate result of the `GROUP BY` clause differs:

```
P1.PAYNO  P1.AMOUNT  P2.PAYNO  P2.AMOUNT
--------  ---------  --------  --------------
       1        100  {1, 2, 3} {100, 75, 100}
       2         75  {1, 2, 3} {100, 75, 100}
       3        100  {1, 2, 3} {100, 75, 100}
```

The intermediate result of the `SELECT` clause:

```
P1.PAYNO  P1.AMOUNT  (P1.AMOUNT * 100) / SUM(P2.AMOUNT)
--------  ---------  ----------------------------------
       1        100                               36.36
       2         75                               27.27
       3        100                               36.36
```

Find out whether this is the final result as well.

Exercise 10.14: How many players live in a town, on average?

Exercise 10.15: For each team, get the team number, the division, and the number of players that played matches for that team.

Exercise 10.16: For each player, get the player number, the name, the sum of all penalties that he or she incurred, and the number of teams from the first division that he or she captains.

Exercise 10.17: For each team captained by a player who lives in Stratford, get the team number and the number of players who have won at least one match for that team.

Exercise 10.18: For each player, get the player number, the name, and the difference between the year in which he or she joined the club and the average year of joining the club.

Exercise 10.19: For each player, get the player number, the name, and the difference between the year in which he or she joined the club and the average year in which players who live in the same town joined the club.

10.8 Grouping with WITH ROLLUP

The `GROUP BY` clause has many features to group data and to calculate aggregated data, such as the total number of penalties or the sum of all penalties. However, so far all statements return results in which all data is on the same level of aggregation. But what if we want to see data belonging to different aggregation levels within one statement? Imagine that we want to see in one statement the total penalty amount for each player and also the total penalty amount for all players. This is not possible with the forms of the `GROUP BY` clauses that we have discussed so far. For this purpose, more than two groupings within one `G ROUP BY` clause are required. By adding the specification `WITH ROLLUP` to the `GROUP BY` clause, it becomes possible.

Portability

Not all SQL products support `WITH ROLLUP`. However, because many products, including MySQL, offer this feature, we discuss it here.

Example 10.18. For each player, find the sum of all his or her penalties, plus the sum of all penalties.

A way to combine these two groupings in one statement is to use the `UNION` operator.

```
SELECT   PLAYERNO, SUM(AMOUNT)
FROM     PENALTIES
GROUP BY PLAYERNO
UNION
SELECT   CAST(NULL AS SIGNED INTEGER), SUM(AMOUNT)
FROM     PENALTIES
```

The result is:

```
PLAYERNO  SUM(AMOUNT)
--------  -----------
       6       100.00
       8        25.00
      27       175.00
      44       130.00
     104        50.00
       ?       480.00
```

Explanation: The rows in this intermediate result in which the `PLAYERNO` column is filled form the result of the first select block. The rows in which `PLAYERNO` is equal to `NULL` make up the result of the second select block. The first five rows contain data on the aggregation level of the player numbers, and the last row contains data on the aggregation level of all rows.

The specification `WITH ROLLUP` has been introduced to simplify this kind of statement. `WITH ROLLUP` can be used to ask for multiple groupings with one `GROUP BY` clause. The previous statement will then be:

```
SELECT   PLAYERNO, SUM(AMOUNT)
FROM     PENALTIES
GROUP BY PLAYERNO WITH ROLLUP
```

Explanation: The result of this statement is the same as the previous one. The specification `WITH ROLLUP` indicates that after the result has been grouped with `[PLAYERNO]`, another grouping is needed—in this case, on all rows.

We give a formal description here. Imagine that in a `GROUP BY` clause, the expressions E_1, E_2, E_3, and E_4 are specified. The grouping that is performed then is [E_1, E_2, E_3, E_4]. When we add the specification `WITH ROLLUP` to this `GROUP BY`, an entire set of groupings will be performed: [E_1, E_2, E_3, E_4], [E_1, E_2, E_3], [E_1, E_2], [E_1], and finally []. The specification `[]` means that all rows are grouped into one group. The specified grouping is seen as the highest aggregation level that is asked, and also indicates that all higher aggregation levels must be calculated again. To aggregate upward is called *rollup* in literature. So, the result of this statement contains data on five different levels of aggregation.

If in the `SELECT` clause an expression occurs in which the result of a certain grouping is not grouped, the `NULL` value is placed in the result.

Example 10.19. For each combination of sex-town, get the number of players, and get the total number of players per sex and the total number of players in the entire table as well.

```
SELECT   SEX, TOWN, COUNT(*)
FROM     PLAYERS
GROUP BY SEX, TOWN WITH ROLLUP
```

The result is:

```
SEX  TOWN       COUNT(*)
---  ---------  --------
M    Stratford         7
M    Inglewood         1
M    Douglas           1
M    ?                 9
V    Midhurst          1
V    Inglewood         1
V    Plymouth          1
V    Eltham            2
V    ?                 5
?    ?                14
```

Explanation: This result has three levels of aggregation. Rows 1, 2, 3, 5, 6, 7, and 8 form the lowest level and have been added because of the grouping `[SEX, TOWN]`; rows 4 and 9 have been added because of the grouping `[SEX]`; and the last row forms the highest level of aggregation and has been added because of the grouping []. It contains the total number of players.

Exercise 10.20: For each team, get the number of matches played and also the total number of matches.

Exercise 10.21: Group the matches by the name of the player and the division of the team, and execute a `ROLLUP`.

10.9 Grouping with WITH CUBE

Another way to get multiple groupings within one `GROUP BY` clause is to use the `WITH CUBE` specification.

Portability

Not all SQL products support `WITH CUBE`. However, because many products, including MySQL, offer this feature, we discuss it here.

Again, we use a formal way to explain this new specification. Imagine that the specification `WITH CUBE` is added to a `GROUP BY` clause consisting of the expressions E_1, E_2, and E_3. As a result, many groupings are performed: [E_1, E_2, E_3], [E_1, E_2], [E_1, E_3], [E_2, E_3], [E_1], [E_2], [E_3], and finally []. The list begins with a grouping on all three expressions, followed by three groupings with each two expressions (one grouping for each possible combination of two expressions), and followed by a grouping for each expression separately; it closes with a grouping of all rows.

Example 10.20. Group the `PLAYERS` table on the columns `SEX` with `TOWN` and add a `WITH CUBE` specification.

```
SELECT   ROW_NUMBER() OVER () AS SEQNO,
         SEX, TOWN, COUNT(*)
FROM     PLAYERS
GROUP BY SEX, TOWN WITH CUBE
ORDER BY SEX, TOWN
```

The result is:

```
SEQNO  SEX  TOWN       COUNT(*)
```

```
-----  ---  ---------  --------
    1    M  Stratford        7
    2    M  Inglewood        1
    3    M  Douglas          1
    4    M  ?                9
    5    F  Midhurst         1
    6    F  Inglewood        1
    7    F  Plymouth         1
    8    F  Eltham           2
    9    F  ?                5
   10    ?  Stratford        7
   11    ?  Midhurst         1
   12    ?  Inglewood        2
   13    ?  Plymouth         1
   14    ?  Douglas          1
   15    ?  Eltham           2
   16    ?  ?               14
```

Explanation: Rows 1, 2, 3, 5, 6, 7, and 8 have been added because of the grouping `[SEX, TOWN]`. Rows 4 and 9 have been added because of the grouping `[SEX]`. Rows 10 through 15 have been added because of the grouping on `[TOWN]`, and row 16 has been included because of the grouping of all rows.

Exercise 10.22: Describe what the difference is between a `WITH ROLLUP` and a `WITH CUBE` specification.

Exercise 10.23: Group the `MATCHES` table on the columns `TEAMNO`, `PLAYERNO`, and `WON`, and add a `WITH CUBE` specification.

10.10 Grouping Sets

The `GROUP BY` clauses that have been described so far use the short notation for the specification of groupings. SQL also has a more extensive notation. So-called *grouping sets specifications* indicate on which expressions groupings must be performed.

Portability

Not all SQL products support grouping sets. However, because many products, including MySQL, offer this feature, we discuss it here.

Example 10.21. For each town, get the most recent date of birth.

With the shortest notation form, this statement looks as follows:

```
SELECT   TOWN, MIN(BIRTH_DATE)
FROM     PLAYERS
GROUP BY TOWN
```

When the extensive notation form is used, this similar formulation occurs:

```
SELECT   TOWN, MIN(BIRTH_DATE)
FROM     PLAYERS
GROUP BY GROUPING SETS ((TOWN))
```

The result of both statements is:

```
TOWN       MIN(BIRTH_DATE)
---------  ---------------
Stratford  1948-09-01
Midhurst   1963-06-22
Inglewood  1962-07-08
Plymouth   1963-10-01
Douglas    1963-05-14
Eltham     1964-12-28
```

Explanation: Behind the term `GROUPING SETS`, the grouping sets specifications can be found. Within such a specification, several groupings can be specified. Each grouping is placed between brackets and the whole should also be placed between brackets—hence, the double brackets.

The advantage of the extensive notation form is that it offers more ways to group data. Several groupings can be specified, among other things, and combinations of `ROLLUP` and `CUBE` can be used.

Example 10.22. For each town, get the number of players, and for each sex, get the number of players as well.

Because a grouping is needed on two different columns, this question cannot be formulated with one `GROUP BY` clause (in which the short notation form is used). A way to combine these two groupings in one statement is to use an `UNION` operator.

```
SELECT   CAST(NULL AS CHAR), TOWN, COUNT(*)
FROM     PLAYERS
GROUP BY TOWN
UNION
SELECT   SEX, CAST(NULL AS CHAR), COUNT(*)
FROM     PLAYERS
GROUP BY SEX
ORDER BY 2, 1
```

The result is:

```
SEX  TOWN       COUNT(*)
---  ---------  --------
?    Stratford         7
?    Midhurst          1
?    Inglewood         2
?    Plymouth          1
?    Douglas           1
?    Eltham            2
M    ?                 9
F    ?                 5
```

Explanation: The rows in this intermediate result in which the `TOWN` column has been filled came from the first select block. The rows in which `TOWN` is equal to `NULL` form the intermediate result of the second select block. In fact, these two rows form subtotals for each sex.

To simplify this type of statement, the grouping sets specification has been added to SQL. With this, one `GROUP BY` clause can be used to specify several groupings. The previous statement becomes:

```
SELECT   SEX, TOWN, COUNT(*)
FROM     PLAYERS
GROUP BY GROUPING SETS ((TOWN), (SEX))
ORDER BY 2, 1
```

Explanation: Behind the words `GROUPING SETS`, two groupings are now specified: `(TOWN)` and (`SEX)`.

If one grouping consists of one expression, the brackets can be removed. So, `GROUP BY GROUPING SETS ((TOWN), (SEX))` is equivalent to `GROUP BY GROUPING SETS (TOWN, SEX)`.

Again, the `GROUP BY` clause, as we discussed in the previous chapters, is, in fact, a shortened notation for the one with grouping sets. Table 10.1 contains several examples of original formulations without grouping sets and their equivalents with grouping sets.

Table 10.1. Original `GROUP BY` Clauses and Their Equivalent Grouping Sets Specifications

Original Specification	Specification with Grouping Sets
`GROUP BY A`	`GROUP BY GROUPING SETS ((A))`
	or
	`GROUP BY GROUPING SETS (A)`
`GROUP BY A, B`	`GROUP BY GROUPING SETS ((A, B))`

GROUP BY YEAR(A), SUBSTR(B)	GROUP BY GROUPING SETS ((YEAR(A), SUBSTR(B)))

A special grouping is `()`. There is no expression between the brackets. In this case, all rows are placed in one group. We can calculate a grand total with that, for example.

Example 10.23. Find for each combination of sex-town the number of players, get for each sex the number of players, and get the total number of players in the entire table.

```
SELECT   SEX, TOWN, COUNT(*)
FROM     PLAYERS
GROUP BY GROUPING SETS ((SEX, TOWN), (SEX), ())
ORDER BY 1, 2
```

The result is:

```
SEX  TOWN       COUNT(*)
---  ---------  --------
M    Stratford         7
M    Inglewood         1
M    Douglas           1
M    ?                 9
F    Midhurst          1
F    Inglewood         1
F    Plymouth          1
F    Eltham            2
F    ?                 5
?    ?                14
```

Explanation: The last row contains the total number of players and is added because of the grouping ().

Example 10.24. Get for each team and for each player individually the number of matches played.

```
SELECT   TEAMNO, PLAYERNO, COUNT(*)
FROM     MATCHES
GROUP BY GROUPING SETS (TEAMNO, PLAYERNO)
ORDER BY 1, 2
```

The result is:

```
TEAMNO  PLAYERNO  COUNT(*)
------  --------  --------
     1         ?         8
     2         ?         5
     ?         2         1
     ?         6         3
     ?         8         2
     ?        27         1
     ?        44         1
     ?        57         1
     ?        83         1
     ?       104         1
     ?       112         2
```

Explanation: The first two rows in the result have been included because of the grouping on the `TEAMNO` column and the other rows because of the grouping on the `PLAYERNO` column.

This example clearly shows that brackets are important. The specification `GROUPING SETS (TEAMNO, PLAYERNO)` returns a different result than `GROUPING SETS ((TEAMNO, PLAYERNO))`. The second grouping sets specification results in one grouping on the combination of the two columns specified and the second grouping sets specification leads to two groupings.

Finally, here are a few abstract examples of certain `GROUP BY` clauses, including the groupings that are executed. E1, E2, and E3 stand for random expressions, and the symbol `∪` represents the union operator.

Table 10.2. The Relationship Between Grouping Sets Specifications and Groupings

`GROUP BY` Clause	Groupings
`GROUP BY E1, E2, E3`	`[E1, E2, E3]`
`GROUP BY GROUPING SETS (())`	`[]`
`GROUP BY GROUPING SETS ((E1, E2, E3))`	`[E1, E2, E3]`
`GROUP BY GROUPING SETS (E1, E2, E3)`	`[E1] ∪ [E2] ∪ [E3]`
`GROUP BY GROUPING SETS ((E1), (E2), (E3))`	`[E1] ∪ [E2] ∪ [E3]`
`GROUP BY GROUPING SETS ((E1, E2), (E3))`	`[E1, E2] ∪ [E3]`
`GROUP BY GROUPING SETS ((E1, E2), E3)`	`[E1, E2] ∪ [E3]`
`GROUP BY GROUPING SETS ((E1, E2), (E3, E4))`	`[E1, E2] ∪ [E3, E4]`
`GROUP BY GROUPING SETS ((E1, (E2, E3)))`	Not allowed

Exercise 10.24: Get the total number of penalties by using a grouping sets specification.

Exercise 10.25: Get for each combination of team number and player number the number of matches, give the number of matches for each team number, and find the total number of matches as well.

Exercise 10.26: Indicate which groupings must be specified for the following `GROUP BY` clauses:

1. `GROUP BY GROUPING SETS ((), (), (E1), (E2))`
2. `GROUP BY GROUPING SETS (E1, (E2, E3),(E3, E4, E5))`
3. `GROUP BY GROUPING SETS ((E1, E2), (), E3, (E2, E1))`

10.11 Grouping with ROLLUP and CUBE

Section 10.8 describes the `WITH ROLLUP` specification. This specification cannot be used if the `GROUP BY` clause contains grouping sets specifications. In that case, an alternative specification must be used.

Portability

Many SQL products, including MySQL, do not support grouping with `ROLLUP` and `CUBE`. But because several other products offer this feature, it is discussed here.

It often happens that data has to be aggregated on different levels. Example 10.23 is a clear example. For such a situation, a short notation form has been added, the `ROLLUP`. Imagine that E1 and E2 are two expressions. In that case, the specification `GROUP BY ROLLUP (E1, E2)` is equal to the specification `GROUP BY GROUPING SETS ((E1, E2), ((E1), ())`. So, `ROLLUP` does not offer extra functionality; it makes only the formulation of some `GROUP BY` clauses easier. This means that the `SELECT` statement in Example 10.23 can be simplified by using `ROLLUP`.

Example 10.25. Get for each combination of sex-town the number of players, get for each sex the number of players, and get the total number of players in the entire table.

```
SELECT   SEX, TOWN, COUNT(*)
FROM     PLAYERS
GROUP BY ROLLUP (SEX, TOWN)
ORDER BY 1, 2
```

The result is (of course, equal to that of Example 10.23):

```
SEX  TOWN       COUNT(*)
---  ---------  --------
M    Stratford         7
M    Inglewood         1
M    Douglas           1
M    ?                 9
F    Midhurst          1
F    Inglewood         1
F    Plymouth          1
F    Eltham            2
F    ?                 5
?    ?                14
```

Explanation: The term `ROLLUP` comes from the OLAP world. It is an operator that is supported by many OLAP tools. It indicates that data must be aggregated on different levels, beginning at the lowest level. That lowest level is, of course, specified at `ROLLUP`. In this example, it is formed by the combination of the `SEX` and `TOWN` columns. After that, the data is aggregated by sex and then the total.

Example 10.26. For each combination of sex-town-year of birth, get the number of players; for each combination of sex-town, get the number of players; for each sex, get the number of players; and, finally, get the total number of players.

```
SELECT   ROW_NUMBER() OVER () AS SEQNO,
         SEX, TOWN, YEAR(BIRTH_DATE), COUNT(*)
FROM     PLAYERS
GROUP BY ROLLUP (SEX, TOWN, YEAR(BIRTH_DATE))
ORDER BY 2, 3, 4
```

The result is:

```
SEQNO  SEX  TOWN       YEAR(BIRTH_DATE)  COUNT(*)
-----  ---  ---------  ----------------  --------
    1  M    Stratford              1948         1
    2  M    Stratford              1956         2
    3  M    Stratford              1963         2
    4  M    Stratford              1964         1
    5  M    Stratford              1971         1
    6  M    Stratford                           7
    7  M    Inglewood              1963         1
    8  M    Inglewood                           1
    9  M    Douglas                1963         1
   10  M    Douglas                             1
   11  M                                        9
   12  F    Midhurst               1963         1
   13  F    Midhurst                            1
   14  F    Inglewood              1962         1
   15  F    Inglewood                           1
   16  F    Plymouth               1963         1
   17  F    Plymouth                            1
   18  F    Eltham                 1964         1
   19  F    Eltham                 1970         1
   20  F    Eltham                              2
   21  F                                        5
   22                                          14
```

Explanation: The grouping `[SEX, TOWN, YEAR(BIRTH_DATE)]` returns the rows 1, 2, 3, 4, 5, 7, 9, 12, 14, 16, 18, and

19. The grouping `[SEX, TOWN]` results in the rows 6, 8, 10, 13, 15, 17, and 20. The grouping `[SEX]` leads up to the rows 11 and 21, and, finally, the grouping `[]` returns the last row.

By adding more brackets, certain aggregation levels can be skipped.

Example 10.27. For each combination of sex-town-year of birth, get the number of players; for each sex, get the number of players; and, finally, get the total number of players.

```
SELECT   ROW_NUMBER() OVER () AS SEQNO,
         SEX, TOWN, YEAR(BIRTH_DATE), COUNT(*)
FROM     PLAYERS
GROUP BY ROLLUP (SEX, (TOWN, YEAR(BIRTH_DATE)))
ORDER BY 2, 3, 4
```

The result is:

```
SEQNO  SEX  TOWN       YEAR(BIRTH_DATE)  COUNT(*)
-----  ---  ---------  ----------------  --------
    1  M    Stratford              1948         1
    2  M    Stratford              1956         2
    3  M    Stratford              1963         2
    4  M    Stratford              1964         1
    5  M    Stratford              1971         1
    6  M    Inglewood              1963         1
    7  M    Douglas                1963         1
    8  M                                        9
    9  F    Midhurst               1963         1
   10  F    Inglewood              1962         1
   11  F    Plymouth               1963         1
   12  F    Eltham                 1964         1
   13  F    Eltham                 1970         1
   14  F                                        5
   15                                          14
```

Explanation: Because the `TOWN` column is placed between brackets together with the expression `YEAR(BIRTH_DATE)`, it is considered to be a group. The groupings that are performed because of this are, successively, `[SEX, TOWN, YEAR(BIRTH_DATE)]`, `[SEX]`, and `[]`. The grouping `[SEX, TOWN]` is skipped.

By way of illustration, the specification `ROLLUP ((SEX, TOWN), YEAR(BIRTH_DATE))` would lead to the groupings `[SEX, TOWN, YEAR(BIRTH_DATE)]`, `[SEX, TOWN]`, and `[]`. Only the grouping on the `SEX` column is absent here. Another example: The specification `ROLLUP ((SEX, TOWN), (YEAR(BIRTH_DATE), MONTH(BIRTH_DATE)))` results in the following groupings: `[SEX, TOWN, YEAR(BIRTH_DATE), MONTH(BIRTH_DATE)]`, `[SEX, TOWN]`, and `[]`.

Besides `ROLLUP`, SQL has a second specification to simplify long `GROUP BY` clauses: the `CUBE`. If E_1 and E_2 are two expressions, the specification `GROUP BY CUBE` (E_1, E_2, E_3) is equal to the specification `GROUP BY GROUPING SETS` $((E_1, E_2, E_3), (E_1, E_2), (E_1, E_3), (E_2, E_3), (E_1), (E_2), (E_3), ())$.

Example 10.28. Get the number of players for each combination of sex-town, for each sex and for each town, and also get the total number of players in the entire table.

```
SELECT   ROW_NUMBER() OVER () AS SEQNO,
         SEX, TOWN, COUNT(*)
FROM     PLAYERS
GROUP BY CUBE (SEX, TOWN)
ORDER BY 2, 3
```

The result is:

```
SEQNO  SEX  TOWN       COUNT(*)
-----  ---  ---------  --------
    1  M    Stratford         7
    2  M    Inglewood         1
    3  M    Douglas           1
    4  M                      9
    5  F    Midhurst          1
    6  F    Inglewood         1
    7  F    Plymouth          1
    8  F    Eltham            2
    9  F                      5
   10       Stratford         7
   11       Midhurst          1
   12       Inglewood         2
```

```
  13      Plymouth         1
  14      Douglas          1
  15      Eltham           2
  16                      14
```

Explanation: Rows 1, 2, 3, 5, 6, 7, and 8 have been included because of the grouping `[SEX, TOWN]`. Rows 4 and 9 have been included because of the grouping `[SEX]`. Rows 10 up to and including 15 form the result of the grouping `[TOWN]`. Finally, row 16 forms the result of a total grouping.

The `GROUPING` function can also be used in combination with `ROLLUP` and `CUBE`.

As in Section 10.10, we show several other abstract examples of certain `GROUP BY` clauses in which `ROLLUP` and `CUBE` appear, including the groupings that are executed. Again, E $_1$, E $_2$, E $_3$ and E $_4$ represent random expressions, and the symbol `U` represents the union operator.

Table 10.3. The Relationship Between Grouping Sets Specifications and Groupings

`GROUP BY` Clause	Groupings
`GROUP BY ROLLUP (())`	`[]`
`GROUP BY ROLLUP (E1)`	`[E1] U []`
`GROUP BY ROLLUP (E1, E2)`	`[E1, E2] U [E1] U []`
`GROUP BY ROLLUP (E1, (E2, E3))`	`[E1, E2, E3] U [E1] U []`
`GROUP BY ROLLUP ((E1, E2), E3)`	`[E1, E2, E3] U [E1, E2] U []`
`GROUP BY ROLLUP ((E1, E2), (E3, E4))`	`[E1, E2, E3, E4] U [E1, E2] U []`
`GROUP BY CUBE (())`	`[]`
`GROUP BY CUBE (E1)`	`[E1] U []`
`GROUP BY CUBE (E1, E2)`	`[E1, E2] U [E1] U [E2] U []`

GROUP BY CUBE (E1, E2, E3)	[E1, E2, E3] U [E1, E2] U [E1, E3] U [E2, E3] U [E1] U [E2] U [E3] U []
GROUP BY CUBE (E1, E2, E3, E4)	[E1, E2, E3, E4] U [E1, E2, E3] [E1, E2, E4] U [E1, E3, E4] U [E2, E3, E4] U [E1, E2] U [E1, E3] U [E1, E4] U [E2, E3] U [E2, E4] U [E3, E4] U [E1] U [E2] U [E3] U [E4] U []
GROUP BY CUBE (E1, (E2, E3))	[E1, E2, E3] U [E1] U [E2, E3] U []
GROUP BY CUBE ((E1, E2), (E3, E4))	[E1, E2, E3, E4] U [E1, E2] U [E3, E4] U []
GROUP BY CUBE (E1, ())	[E1] U []

Exercise 10.27: For each combination of team number-player number, get the number of matches, and also get the number of matches for each team and the total number of matches. In this result, include only those matches that have been won in this result.

Exercise 10.28: Execute a `CUBE` on the column town, sex, and team number after the two tables `PLAYERS` and `TEAMS` have been joined.

10.12 Combining Grouping Sets

Multiple groupings can be included in one select block. Simple group by expressions may be combined with grouping sets specifications, multiple grouping sets specifications may be specified and even two rollups may be specified. However, the effect of this combining needs some explanation.

If a grouping sets specification is combined with one or more simple group by expressions, the latter simply is added to the grouping sets specification. For example, the specification `GROUPING SETS ((E` $_1$`)), E` $_2$`, E` $_3$ is equal to `GROUPING SETS ((E` $_1$`, E` $_2$`, E` $_3$`))`. If the grouping sets specification contains two groupings, the simple expressions are added to both groupings. The specification `GROUPING SETS ((E` $_1$`), (E` $_2$`)), E` $_3$ is, for example, equal to `GROUPING SETS ((E` $_1$`,E` $_3$`),(E` $_2$`,E` $_3$`))`.

If two grouping sets specifications are included in one GROUP BY clause, some kind of multiplication of the specifications takes place. For example, the specification `GROUPING SETS ((E` $_1$`), (E` $_2$`)), GROUPING SETS ((E` $_3$`))` contains two grouping sets specifications, in which the first consists of two groupings and the second of one grouping. SQL turns it into `GROUPING SETS ((E` $_1$`, E` $_3$`), (E` $_2$`, E` $_3$`))`. Now the expression E $_3$ has been added to both groupings of the first grouping sets specification. The specification `GROUPING SETS ((E` $_1$`), (E` $_2$`)), GROUPING SETS ((E` $_3$`), (E` $_4$`))` is turned into `GROUPING SETS ((E` $_1$`, E` $_3$`), (E` $_1$`, E` $_4$`), (E` $_2$`, E` $_3$`), (E` $_2$`, E` $_4$`))`. It is obvious that E $_1$ is linked to both groupings of the other grouping sets specifications. The same applies to E $_2$.

Finally, the specification `GROUPING SETS ((E` $_1$`), (E` $_2$`)), GROUPING SETS ((E` $_3$`), (E` $_4$`)), E` $_5$ is turned into `GROUPING SETS ((E` $_1$`, E` $_3$`, E` $_5$`)), (E` $_1$`, E` $_4$`, E` $_5$`), (E` $_2$`, E` $_3$`, E` $_5$`), (E` $_2$`, E` $_4$`, E` $_5$`))`.

Table 10.4 gives a few abstract examples of certain `GROUP BY` clauses in which several grouping sets specifications appear, including the groupings that are executed. Again, E $_1$, E $_2$, E $_3$, and E $_4$ stand for random expressions, and the symbol `U` represents the union operator.

Table 10.4. Combining Grouping Sets Specifications

GROUP BY Clause	Groupings
GROUP BY GROUPING SETS (E1, E2), E3	[E1, E3] ∪ [E2, E3]
GROUP BY E1, GROUPING SETS (E2, E3)	[E1, E2] ∪ [E1, E3]
GROUP BY GROUPING SETS ((E1, E2)), E3	[E1, E2, E3]
GROUP BY GROUPING SETS ((E1, E2), (E3, E4)), E5	[E1, E2, E5] ∪ [E3, E4, E5]
GROUP BY ROLLUP (E1, E2)), E3	[E1, E2, E3] ∪ [E1, E2] ∪ [E1] ∪ []
GROUP BY GROUPING SETS (E1, E2),	[E1, E3] ∪ [E1, E4] ∪
GROUPING SETS (E3, E4)	[E2, E3] ∪ [E2, E4] ∪
GROUP BY GROUPING SETS (E1, ROLLUP (E2, E3))	[E1] ∪ [E2, E3] ∪ [E2] ∪ []
GROUP BY GROUPING SETS ((E1, ROLLUP (E2)))	[E1, E2] ∪ [E1] ∪ []
GROUP BY ROLLUP (E1, E2), ROLLUP (E3, E4)	[E1, E2, E3, E4] ∪ [E1, E3, E4] ∪ [E3, E4] ∪ [E1, E2, E3] ∪ [E1, E3] ∪ [E3] ∪ [E1, E2] ∪ [E1] ∪ []

10.13 Answers

10.1	`SELECT   JOINED`
	`FROM     PLAYERS`
	`GROUP BY JOINED`

10.2	`SELECT   JOINED, COUNT(*)`
	`FROM     PLAYERS`
	`GROUP BY JOINED`

10.3	`SELECT   PLAYERNO, AVG(AMOUNT), COUNT(*)`
	`FROM     PENALTIES`
	`GROUP BY PLAYERNO`

	`SELECT   TEAMNO, COUNT(*), SUM(WON)`

10.4

```
FROM     MATCHES
WHERE    TEAMNO IN
         (SELECT  TEAMNO
         FROM     TEAMS
         WHERE    DIVISION = 'first')
GROUP BY TEAMNO
```

10.5

```
SELECT   WON, LOST, COUNT(*)
FROM     MATCHES
WHERE    WON > LOST
GROUP BY WON, LOST
ORDER BY 1, 2
```

10.6

```
SELECT   YEAR(BEGIN_DATE), MONTH(BEGIN_DATE), COUNT(*)
FROM     COMMITTEE_MEMBERS
GROUP BY YEAR(BEGIN_DATE), MONTH(BEGIN_DATE)
ORDER BY 1, 2
```

10.7

```
SELECT   P.NAME, T.DIVISION, SUM(WON)
FROM     (MATCHES AS M INNER JOIN PLAYERS AS P
         ON M.PLAYERNO = P.PLAYERNO)
         INNER JOIN TEAMS AS T
         ON M.TEAMNO = T.TEAMNO
GROUP BY P.NAME, T.DIVISION
ORDER BY 1
```

10.8

```
SELECT   NAME, INITIALS, COUNT(*)
FROM     PLAYERS AS P INNER JOIN PENALTIES AS PEN
         ON P.PLAYERNO = PEN.PLAYERNO
WHERE    P.TOWN = 'Inglewood'
GROUP BY P.PLAYERNO, NAME, INITIALS
```

10.9

```
SELECT   T.TEAMNO, DIVISION, SUM(WON)
FROM     TEAMS AS T, MATCHES AS M
WHERE    T.TEAMNO = M.TEAMNO
GROUP BY T.TEAMNO, DIVISION
```

10.10

```
SELECT   LENGTH(RTRIM(NAME)), COUNT(*)
FROM     PLAYERS
GROUP BY LENGTH(RTRIM(NAME))
```

10.11

```
SELECT   ABS(WON - LOST), COUNT(*)
FROM     MATCHES
GROUP BY ABS(WON - LOST)
```

10.12

1. The result of the `DIVISION` column has not been grouped, while this column appears in the `SELECT` clause.
2. The `NAME` column cannot appear like this in the `SELECT` clause because the result has not been grouped on the full `NAME` column.
3. The `PLAYERNO` column appears in the `SELECT` clause, while the result has not been grouped; furthermore, the column does not appear as parameter of an aggregation function.

10.13

Superfluous.

Not superfluous.

Superfluous.

10.14

```
SELECT  AVG(NUMBERS)
FROM   (SELECT  COUNT(*) AS NUMBERS
        FROM     PLAYERS
        GROUP BY TOWN) AS TOWNS
```

10.15

```
SELECT   TEAMS.TEAMNO, DIVISION, NUMBER_PLAYERS
FROM     TEAMS LEFT OUTER JOIN
        (SELECT   TEAMNO, COUNT(*) AS NUMBER_PLAYERS
         FROM      MATCHES
         GROUP BY  TEAMNO) AS M
         ON (TEAMS.TEAMNO = M.TEAMNO)
```

10.16

```
SELECT   PLAYERS.PLAYERNO, NAME, SUM_AMOUNT,
         NUMBER_TEAMS
FROM    (PLAYERS LEFT OUTER JOIN
        (SELECT   PLAYERNO, SUM(AMOUNT) AS SUM_AMOUNT
         FROM      PENALTIES
         GROUP BY  PLAYERNO) AS TOTALS
        ON (PLAYERS.PLAYERNO = TOTALS.PLAYERNO))
           LEFT OUTER JOIN
          (SELECT  PLAYERNO, COUNT(*) AS NUMBER_TEAMS
           FROM     TEAMS
           WHERE    DIVISION = 'first'
           GROUP BY PLAYERNO) AS NUMBERS
           ON (PLAYERS.PLAYERNO = NUMBERS.PLAYERNO)
```

10.17

```
SELECT   TEAMNO, COUNT(DISTINCT PLAYERNO)
FROM     MATCHES
WHERE    TEAMNO IN
        (SELECT  TEAMNO
         FROM     PLAYERS AS P INNER JOIN TEAMS AS T
                  ON P.PLAYERNO = T.PLAYERNO
         AND      TOWN = 'Stratford')
AND      WON > LOST
GROUP BY TEAMNO
```

10.18

```
SELECT  PLAYERNO, NAME, JOINED - AVERAGE
FROM    PLAYERS,
       (SELECT  AVG(JOINED) AS AVERAGE
        FROM    PLAYERS) AS T
```

10.19

```
SELECT  PLAYERNO, NAME, JOINED - AVERAGE
FROM    PLAYERS,
       (SELECT  TOWN, AVG(JOINED) AS AVERAGE
        FROM     PLAYERS
        GROUP BY TOWN) AS TOWNS
WHERE   PLAYERS.TOWN = TOWNS.TOWN
```

10.20

```
SELECT   TEAMNO, COUNT(*)
FROM     MATCHES
GROUP BY TEAMNO WITH ROLLUP
```

10.21

```
SELECT   P.NAME, T.DIVISION, SUM(WON)
FROM    (MATCHES AS M INNER JOIN PLAYERS AS P
         ON M.PLAYERNO = P.PLAYERNO)
         INNER JOIN TEAMS AS T
         ON M.TEAMNO = T.TEAMNO
GROUP BY P.NAME, T.DIVISION WITH ROLLUP
```

10.22

The `WITH ROLLUP` specification calculates all levels of aggregation; at the bottom is a grouping based upon the expressions specified. The `WITH CUBE` specification returns much more data. For every possible combination of expressions specified, groupings are performed.

10.23

```
SELECT   ROW_NUMBER() OVER () AS SEQNO,
         TEAMNO,  PLAYERNO, WON, COUNT(*)
FROM     MATCHES
GROUP BY TEAMNO, PLAYERNO, WON WITH CUBE
ORDER BY 2, 3
```

10.24

```
SELECT   COUNT(*)
FROM     MATCHES
```

	`GROUP BY GROUPING SETS (())`

10.25	`SELECT   TEAMNO, PLAYERNO, COUNT(*)`
	`FROM     MATCHES`
	`GROUP BY GROUPING SETS ((TEAMNO, PLAYERNO), (TEAMNO), ())`
	`ORDER BY 1, 2`

10.26	1. `[] ∪ [E1] ∪ [E2]` 2. `[E1] ∪ [E2, E3] ∪ [E3, E4, E5]` 3. `[E1, E2] ∪ [] ∪ [E3]`

10.27	`SELECT   TEAMNO, PLAYERNO, COUNT(*)`
	`FROM     MATCHES`
	`WHERE    WON > LOST`
	`GROUP BY ROLLUP (TEAMNO, PLAYERNO)`
	`ORDER BY 1, 2`

10.28	`SELECT   P.TOWN, P.SEX, M.TEAMNO, COUNT(*)`
	`FROM     MATCHES AS M INNER JOIN PLAYERS AS P`
	`         ON M.PLAYERNO = P.PLAYERNO`
	`GROUP BY CUBE (P.TOWN, P.SEX, M.TEAMNO)`
	`ORDER BY 1, 2, 3`

© 2014 Pearson Education, Informit. All rights reserved.
800 East 96th Street, Indianapolis, Indiana 46240