CHAPTER 8

Grouping and Aggregates

Data is generally stored at the lowest level of granularity needed by any of a database's users; if Chuck in accounting needs to look at individual customer transactions, then there needs to be a table in the database that stores individual transactions. That doesn't mean, however, that all users must deal with the data as it is stored in the database. The focus of this chapter is on how data can be grouped and aggregated to allow users to interact with it at some higher level of granularity than what is stored in the database.

Grouping Concepts

Sometimes you will want to find trends in your data that will require the database server to cook the data a bit before you can generate the results you are looking for. For example, let's say that you are in charge of operations at the bank, and you would like to find out how many accounts are being opened by each bank teller. You could issue a simple query to look at the raw data:

```
mysql> SELECT open_emp_id
    -> FROM account;
+-------------+
| open_emp_id |
+-------------+
|           1 |
|           1 |
|           1 |
|           1 |
|           1 |
|           1 |
|           1 |
|           1 |
|          10 |
|          10 |
|          10 |
|          10 |
|          10 |
|          10 |
|          10 |
|          13 |
```

```
|          13 |
|          13 |
|          16 |
|          16 |
|          16 |
|          16 |
|          16 |
|          16 |
+-------------+
24 rows in set (0.01 sec)
```

With only 24 rows in the `account` table, it is relatively easy to see that four different employees opened accounts and that employee ID 16 has opened six accounts; however, if the bank has dozens of employees and thousands of accounts, this approach would prove tedious and error-prone.

Instead, you can ask the database server to group the data for you by using the `group by` clause. Here's the same query but employing a `group by` clause to group the account data by employee ID:

```
mysql> SELECT open_emp_id
    -> FROM account
    -> GROUP BY open_emp_id;
+-------------+
| open_emp_id |
+-------------+
|           1 |
|          10 |
|          13 |
|          16 |
+-------------+
4 rows in set (0.00 sec)
```

The result set contains one row for each distinct value in the `open_emp_id` column, resulting in four rows instead of the full 24 rows. The reason for the smaller result set is that each of the four employees opened more than one account. To see how many accounts each teller opened, you can use an *aggregate function* in the `select` clause to count the number of rows in each group:

```
mysql> SELECT open_emp_id, COUNT(*) how_many
    -> FROM account
    -> GROUP BY open_emp_id;
+-------------+----------+
| open_emp_id | how_many |
+-------------+----------+
|           1 |        8 |
|          10 |        7 |
|          13 |        3 |
|          16 |        6 |
+-------------+----------+
4 rows in set (0.00 sec)
```

The aggregate function `count()` counts the number of rows in each group, and the asterisk tells the server to count everything in the group. Using the combination of a

`group by` clause and the `count()` aggregate function, you are able to generate exactly the data needed to answer the business question without having to look at the raw data.

When grouping data, you may need to filter out undesired data from your result set based on groups of data rather than based on the raw data. Since the `group by` clause runs *after* the `where` clause has been evaluated, you cannot add filter conditions to your `where` clause for this purpose. For example, here's an attempt to filter out any cases where an employee has opened fewer than five accounts:

```
mysql> SELECT open_emp_id, COUNT(*) how_many
    -> FROM account
    -> WHERE COUNT(*) > 4
    -> GROUP BY open_emp_id;
ERROR 1111 (HY000): Invalid use of group function
```

You cannot refer to the aggregate function `count(*)` in your `where` clause, because the groups have not yet been generated at the time the `where` clause is evaluated. Instead, you must put your group filter conditions in the `having` clause. Here's what the query would look like using `having`:

```
mysql> SELECT open_emp_id, COUNT(*) how_many
    -> FROM account
    -> GROUP BY open_emp_id
    -> HAVING COUNT(*) > 4;
+------------+----------+
| open_emp_id | how_many |
+------------+----------+
|           1 |        8 |
|          10 |        7 |
|          16 |        6 |
+------------+----------+
3 rows in set (0.00 sec)
```

Because those groups containing fewer than five members have been filtered out via the `having` clause, the result set now contains only those employees who have opened five or more accounts, thus eliminating employee ID 13 from the results.

Aggregate Functions

Aggregate functions perform a specific operation over all rows in a group. Although every database server has its own set of specialty aggregate functions, the common aggregate functions implemented by all major servers include:

`Max()`
: Returns the maximum value within a set

`Min()`
: Returns the minimum value within a set

`Avg()`
: Returns the average value across a set

`Sum()`
: Returns the sum of the values across a set

`Count()`
: Returns the number of values in a set

Here's a query that uses all of the common aggregate functions to analyze the available balances for all checking accounts:

```
mysql> SELECT MAX(avail_balance) max_balance,
    ->   MIN(avail_balance) min_balance,
    ->   AVG(avail_balance) avg_balance,
    ->   SUM(avail_balance) tot_balance,
    ->   COUNT(*) num_accounts
    -> FROM account
    -> WHERE product_cd = 'CHK';
+-------------+-------------+-------------+-------------+--------------+
| max_balance | min_balance | avg_balance | tot_balance | num_accounts |
+-------------+-------------+-------------+-------------+--------------+
|    38552.05 |      122.37 | 7300.800985 |    73008.01 |           10 |
+-------------+-------------+-------------+-------------+--------------+
1 row in set (0.09 sec)
```

The results from this query tell you that, across the 10 checking accounts in the `account` table, there is a maximum balance of $38,552.05, a minimum balance of $122.37, an average balance of $7,300.80, and a total balance across all 10 accounts of $73,008.01. Hopefully, this gives you an appreciation for the role of these aggregate functions; the next subsections further clarify how you can utilize these functions.

Implicit Versus Explicit Groups

In the previous example, every value returned by the query is generated by an aggregate function, and the aggregate functions are applied across the group of rows specified by the filter condition `product_cd = 'CHK'`. Since there is no `group by` clause, there is a single, *implicit* group (all rows returned by the query).

In most cases, however, you will want to retrieve additional columns along with columns generated by aggregate functions. What if, for example, you wanted to extend the previous query to execute the same five aggregate functions for *each* product type, instead of just for checking accounts? For this query, you would want to retrieve the `product_cd` column along with the five aggregate functions, as in:

```
SELECT product_cd,
  MAX(avail_balance) max_balance,
  MIN(avail_balance) min_balance,
  AVG(avail_balance) avg_balance,
  SUM(avail_balance) tot_balance,
  COUNT(*) num_accounts
FROM account;
```

However, if you try to execute the query, you will receive the following error:

```
ERROR 1140 (42000): Mixing of GROUP columns (MIN(),MAX(),COUNT(),...) with no GROUP
columns is illegal if there is no GROUP BY clause
```

While it may be obvious to you that you want the aggregate functions applied to each set of products found in the `account` table, this query fails because you have not *explicitly* specified how the data should be grouped. Therefore, you will need to add a `group by` clause to specify over which group of rows the aggregate functions should be applied:

```
mysql> SELECT product_cd,
    ->   MAX(avail_balance) max_balance,
    ->   MIN(avail_balance) min_balance,
    ->   AVG(avail_balance) avg_balance,
    ->   SUM(avail_balance) tot_balance,
    ->   COUNT(*) num_accts
    -> FROM account
    -> GROUP BY product_cd;
+------------+-------------+-------------+--------------+-------------+-----------+
| product_cd | max_balance | min_balance | avg_balance  | tot_balance | num_accts |
+------------+-------------+-------------+--------------+-------------+-----------+
| BUS        |     9345.55 |        0.00 |  4672.774902 |     9345.55 |         2 |
| CD         |    10000.00 |     1500.00 |  4875.000000 |    19500.00 |         4 |
| CHK        |    38552.05 |      122.37 |  7300.800985 |    73008.01 |        10 |
| MM         |     9345.55 |     2212.50 |  5681.713216 |    17045.14 |         3 |
| SAV        |      767.77 |      200.00 |   463.940002 |     1855.76 |         4 |
| SBL        |    50000.00 |    50000.00 | 50000.000000 |    50000.00 |         1 |
+------------+-------------+-------------+--------------+-------------+-----------+
6 rows in set (0.00 sec)
```

With the inclusion of the `group by` clause, the server knows to group together rows having the same value in the `product_cd` column first and then to apply the five aggregate functions to each of the six groups.

Counting Distinct Values

When using the `count()` function to determine the number of members in each group, you have your choice of counting *all* members in the group, or counting only the *distinct* values for a column across all members of the group. For example, consider the following data, which shows the employee responsible for opening each account:

```
mysql> SELECT account_id, open_emp_id
    -> FROM account
    -> ORDER BY open_emp_id;
+------------+-------------+
| account_id | open_emp_id |
+------------+-------------+
|          8 |           1 |
|          9 |           1 |
|         10 |           1 |
|         12 |           1 |
|         13 |           1 |
|         17 |           1 |
|         18 |           1 |
```

```
|         19 |           1 |
|          1 |          10 |
|          2 |          10 |
|          3 |          10 |
|          4 |          10 |
|          5 |          10 |
|         14 |          10 |
|         22 |          10 |
|          6 |          13 |
|          7 |          13 |
|         24 |          13 |
|         11 |          16 |
|         15 |          16 |
|         16 |          16 |
|         20 |          16 |
|         21 |          16 |
|         23 |          16 |
+------------+-------------+
24 rows in set (0.00 sec)
```

As you can see, multiple accounts were opened by four different employees (employee IDs 1, 10, 13, and 16). Let's say that, instead of performing a manual count, you want to create a query that counts the number of employees who have opened accounts. If you apply the `count()` function to the `open_emp_id` column, you will see the following results:

```
mysql> SELECT COUNT(open_emp_id)
    -> FROM account;
+--------------------+
| COUNT(open_emp_id) |
+--------------------+
|                 24 |
+--------------------+
1 row in set (0.00 sec)
```

In this case, specifying the `open_emp_id` column as the column to be counted generates the same results as specifying `count(*)`. If you want to count *distinct* values in the group rather than just counting the number of rows in the group, you need to specify the `distinct` keyword, as in:

```
mysql> SELECT COUNT(DISTINCT open_emp_id)
    -> FROM account;
+-----------------------------+
| COUNT(DISTINCT open_emp_id) |
+-----------------------------+
|                           4 |
+-----------------------------+
1 row in set (0.00 sec)
```

By specifying `distinct`, therefore, the `count()` function examines the values of a column for each member of the group in order to find and remove duplicates, rather than simply counting the number of values in the group.

Using Expressions

Along with using columns as arguments to aggregate functions, you can build expressions to use as arguments. For example, you may want to find the maximum value of pending deposits across all accounts, which is calculated by subtracting the available balance from the pending balance. You can achieve this via the following query:

```
mysql> SELECT MAX(pending_balance - avail_balance) max_uncleared
    -> FROM account;
+---------------+
| max_uncleared |
+---------------+
|        660.00 |
+---------------+
1 row in set (0.00 sec)
```

While this example uses a fairly simple expression, expressions used as arguments to aggregate functions can be as complex as needed, as long as they return a number, string, or date. In Chapter 11, I show you how you can use `case` expressions with aggregate functions to determine whether a particular row should or should not be included in an aggregation.

How Nulls Are Handled

When performing aggregations, or, indeed, any type of numeric calculation, you should always consider how `null` values might affect the outcome of your calculation. To illustrate, I will build a simple table to hold numeric data and populate it with the set {1, 3, 5}:

```
mysql> CREATE TABLE number_tbl
    ->  (val SMALLINT);
Query OK, 0 rows affected (0.01 sec)

mysql> INSERT INTO number_tbl VALUES (1);
Query OK, 1 row affected (0.00 sec)

mysql> INSERT INTO number_tbl VALUES (3);
Query OK, 1 row affected (0.00 sec)

mysql> INSERT INTO number_tbl VALUES (5);
Query OK, 1 row affected (0.00 sec)
```

Consider the following query, which performs five aggregate functions on the set of numbers:

```
mysql> SELECT COUNT(*) num_rows,
    ->   COUNT(val) num_vals,
    ->   SUM(val) total,
    ->   MAX(val) max_val,
    ->   AVG(val) avg_val
    -> FROM number_tbl;
+----------+----------+-------+---------+---------+
```

```
| num_rows | num_vals | total | max_val | avg_val |
+----------+----------+-------+---------+---------+
|        3 |        3 |     9 |       5 |  3.0000 |
+----------+----------+-------+---------+---------+
1 row in set (0.08 sec)
```

The results are as you would expect: both `count(*)` and `count(val)` return the value `3`, `sum(val)` returns the value `9`, `max(val)` returns `5`, and `avg(val)` returns `3`. Next, I will add a `null` value to the `number_tbl` table and run the query again:

```
mysql> INSERT INTO number_tbl VALUES (NULL);
Query OK, 1 row affected (0.01 sec)

mysql> SELECT COUNT(*) num_rows,
    ->   COUNT(val) num_vals,
    ->   SUM(val) total,
    ->   MAX(val) max_val,
    ->   AVG(val) avg_val
    -> FROM number_tbl;
+----------+----------+-------+---------+---------+
| num_rows | num_vals | total | max_val | avg_val |
+----------+----------+-------+---------+---------+
|        4 |        3 |     9 |       5 |  3.0000 |
+----------+----------+-------+---------+---------+
1 row in set (0.00 sec)
```

Even with the addition of the `null` value to the table, the `sum()`, `max()`, and `avg()` functions all return the same values, indicating that they ignore any `null` values encountered. The `count(*)` function now returns the value `4`, which is valid since the `number_tbl` table contains four rows, while the `count(val)` function still returns the value `3`. The difference is that `count(*)` counts the number of rows, whereas `count(val)` counts the number of *values* contained in the `val` column and ignores any `null` values encountered.

Generating Groups

People are rarely interested in looking at raw data; instead, people engaging in data analysis will want to manipulate the raw data to better suit their needs. Examples of common data manipulations include:

- Generating totals for a geographic region, such as total European sales
- Finding outliers, such as the top salesperson for 2005
- Determining frequencies, such as the number of new accounts opened for each branch

To answer these types of queries, you will need to ask the database server to group rows together by one or more columns or expressions. As you have seen already in several examples, the `group by` clause is the mechanism for grouping data within a query. In this section, you will see how to group data by one or more columns, how to group data using expressions, and how to generate rollups within groups.

Single-Column Grouping

Single-column groups are the simplest and most-often-used type of grouping. If you want to find the total balances for each product, for example, you need only group on the `account.product_cd` column, as in:

```
mysql> SELECT product_cd, SUM(avail_balance) prod_balance
    -> FROM account
    -> GROUP BY product_cd;
+------------+--------------+
| product_cd | prod_balance |
+------------+--------------+
| BUS        |      9345.55 |
| CD         |     19500.00 |
| CHK        |     73008.01 |
| MM         |     17045.14 |
| SAV        |      1855.76 |
| SBL        |     50000.00 |
+------------+--------------+
6 rows in set (0.00 sec)
```

This query generates six groups, one for each product, and then sums the available balances for each member of the group.

Multicolumn Grouping

In some cases, you may want to generate groups that span *more* than one column. Expanding on the previous example, imagine that you want to find the total balances not just for each product, but for both products and branches (e.g., what's the total balance for all checking accounts opened at the Woburn branch?). The following example shows how you can accomplish this:

```
mysql> SELECT product_cd, open_branch_id,
    ->   SUM(avail_balance) tot_balance
    -> FROM account
    -> GROUP BY product_cd, open_branch_id;
+------------+----------------+-------------+
| product_cd | open_branch_id | tot_balance |
+------------+----------------+-------------+
| BUS        |              2 |     9345.55 |
| BUS        |              4 |        0.00 |
| CD         |              1 |    11500.00 |
| CD         |              2 |     8000.00 |
| CHK        |              1 |      782.16 |
| CHK        |              2 |     3315.77 |
| CHK        |              3 |     1057.75 |
| CHK        |              4 |    67852.33 |
| MM         |              1 |    14832.64 |
| MM         |              3 |     2212.50 |
| SAV        |              1 |      767.77 |
| SAV        |              2 |      700.00 |
| SAV        |              4 |      387.99 |
| SBL        |              3 |    50000.00 |
```

```
+------------+---------------+-------------+
14 rows in set (0.00 sec)
```

This version of the query generates 14 groups, one for each combination of product and branch found in the `account` table. Along with adding the `open_branch_id` column to the `select` clause, I also added it to the `group by` clause, since `open_branch_id` is retrieved from a table and is not generated via an aggregate function.

Grouping via Expressions

Along with using columns to group data, you can build groups based on the values generated by expressions. Consider the following query, which groups employees by the year they began working for the bank:

```
mysql> SELECT EXTRACT(YEAR FROM start_date) year,
    ->   COUNT(*) how_many
    -> FROM employee
    -> GROUP BY EXTRACT(YEAR FROM start_date);
+------+----------+
| year | how_many |
+------+----------+
| 2004 |        2 |
| 2005 |        3 |
| 2006 |        8 |
| 2007 |        3 |
| 2008 |        2 |
+------+----------+
5 rows in set (0.15 sec)
```

This query employs a fairly simple expression, which uses the `extract()` function to return only the year portion of a date, to group the rows in the `employee` table.

Generating Rollups

In "Multicolumn Grouping" on page 151, I showed an example that generates total account balances for each product and branch. Let's say, however, that along with the total balances for each product/branch combination, you also want total balances for each distinct product. You could run an additional query and merge the results, you could load the results of the query into a spreadsheet, or you could build a Perl script, Java program, or some other mechanism to take that data and perform the additional calculations. Better yet, you could use the `with rollup` option to have the database server do the work for you. Here's the revised query using `with rollup` in the `group by` clause:

```
mysql> SELECT product_cd, open_branch_id,
    ->   SUM(avail_balance) tot_balance
    -> FROM account
    -> GROUP BY product_cd, open_branch_id WITH ROLLUP;

+------------+---------------+-------------+
| product_cd | open_branch_id | tot_balance |
```

```
+------------+---------------+-------------+
| BUS        |             2 |     9345.55 |
| BUS        |             4 |        0.00 |
| BUS        |          NULL |     9345.55 |
| CD         |             1 |    11500.00 |
| CD         |             2 |     8000.00 |
| CD         |          NULL |    19500.00 |
| CHK        |             1 |      782.16 |
| CHK        |             2 |     3315.77 |
| CHK        |             3 |     1057.75 |
| CHK        |             4 |    67852.33 |
| CHK        |          NULL |    73008.01 |
| MM         |             1 |    14832.64 |
| MM         |             3 |     2212.50 |
| MM         |          NULL |    17045.14 |
| SAV        |             1 |      767.77 |
| SAV        |             2 |      700.00 |
| SAV        |             4 |      387.99 |
| SAV        |          NULL |     1855.76 |
| SBL        |             3 |    50000.00 |
| SBL        |          NULL |    50000.00 |
| NULL       |          NULL |   170754.46 |
+------------+---------------+-------------+
21 rows in set (0.02 sec)
```

There are now seven additional rows in the result set, one for each of the six distinct products and one for the grand total (all products combined). For the six product rollups, a `null` value is provided for the `open_branch_id` column, since the rollup is being performed across all branches. Looking at the third line of the output, for example, you will see that a total of $9,345.55 was deposited in BUS accounts across all branches. For the grand total row, a `null` value is provided for both the `product_cd` and `open_branch_id` columns; the last line of output shows a total of $170,754.46 across all products and branches.

If you are using Oracle Database, you need to use a slightly different syntax to indicate that you want a rollup performed. The `group by` clause for the previous query would look as follows when using Oracle:

```
GROUP BY ROLLUP(product_cd, open_branch_id)
```

The advantage of this syntax is that it allows you to perform rollups on a subset of the columns in the `group by` clause. If you are grouping by columns a, b, and c, for example, you could indicate that the server should perform rollups on only b and c via the following:

```
GROUP BY a, ROLLUP(b, c)
```

If, along with totals by product, you also want to calculate totals per branch, then you can use the `with cube` option, which generates summary rows for *all* possible combinations of the grouping columns. Unfortunately, `with cube` is not available in version 6.0 of MySQL, but it is available with SQL Server and Oracle Database. Here's an

example using `with cube`, but I have removed the `mysql>` prompt to show that the query cannot yet be performed with MySQL:

```
SELECT product_cd, open_branch_id,
  SUM(avail_balance) tot_balance
FROM account
GROUP BY product_cd, open_branch_id WITH CUBE;
+------------+----------------+-------------+
| product_cd | open_branch_id | tot_balance |
+------------+----------------+-------------+
| NULL       |           NULL |   170754.46 |
| NULL       |              1 |    27882.57 |
| NULL       |              2 |    21361.32 |
| NULL       |              3 |    53270.25 |
| NULL       |              4 |    68240.32 |
| BUS        |              2 |     9345.55 |
| BUS        |              4 |        0.00 |
| BUS        |           NULL |     9345.55 |
| CD         |              1 |    11500.00 |
| CD         |              2 |     8000.00 |
| CD         |           NULL |    19500.00 |
| CHK        |              1 |      782.16 |
| CHK        |              2 |     3315.77 |
| CHK        |              3 |     1057.75 |
| CHK        |              4 |    67852.33 |
| CHK        |           NULL |    73008.01 |
| MM         |              1 |    14832.64 |
| MM         |              3 |     2212.50 |
| MM         |           NULL |    17045.14 |
| SAV        |              1 |      767.77 |
| SAV        |              2 |      700.00 |
| SAV        |              4 |      387.99 |
| SAV        |           NULL |     1855.76 |
| SBL        |              3 |    50000.00 |
| SBL        |           NULL |    50000.00 |
+------------+----------------+-------------+
25 rows in set (0.02 sec)
```

Using `with cube` generates four more rows than the `with rollup` version of the query, one for each of the four branch IDs. Similar to `with rollup`, `null` values are placed in the `product_cd` column to indicate that a branch summary is being performed.

Once again, if you are using Oracle Database, you need to use a slightly different syntax to indicate that you want a cube operation performed. The `group by` clause for the previous query would look as follows when using Oracle:

```
GROUP BY CUBE(product_cd, open_branch_id)
```

Group Filter Conditions

In Chapter 4, I introduced you to various types of filter conditions and showed how you can use them in the `where` clause. When grouping data, you also can apply filter conditions to the data *after* the groups have been generated. The `having` clause is where you should place these types of filter conditions. Consider the following example:

```
mysql> SELECT product_cd, SUM(avail_balance) prod_balance
    -> FROM account
    -> WHERE status = 'ACTIVE'
    -> GROUP BY product_cd
    -> HAVING SUM(avail_balance) >= 10000;
+------------+--------------+
| product_cd | prod_balance |
+------------+--------------+
| CD         |     19500.00 |
| CHK        |     73008.01 |
| MM         |     17045.14 |
| SBL        |     50000.00 |
+------------+--------------+
4 rows in set (0.00 sec)
```

This query has two filter conditions: one in the `where` clause, which filters out inactive accounts, and the other in the `having` clause, which filters out any product whose total available balance is less than $10,000. Thus, one of the filters acts on data *before* it is grouped, and the other filter acts on data *after* the groups have been created. If you mistakenly put both filters in the `where` clause, you will see the following error:

```
mysql> SELECT product_cd, SUM(avail_balance) prod_balance
    -> FROM account
    -> WHERE status = 'ACTIVE'
    ->   AND SUM(avail_balance) > 10000
    -> GROUP BY product_cd;
ERROR 1111 (HY000): Invalid use of group function
```

This query fails because you cannot include an aggregate function in a query's `where` clause. This is because the filters in the `where` clause are evaluated *before* the grouping occurs, so the server can't yet perform any functions on groups.

When adding filters to a query that includes a `group by` clause, think carefully about whether the filter acts on raw data, in which case it belongs in the `where` clause, or on grouped data, in which case it belongs in the `having` clause.

You may, however, include aggregate functions in the `having` clause, that do *not* appear in the `select` clause, as demonstrated by the following:

```
mysql> SELECT product_cd, SUM(avail_balance) prod_balance
    -> FROM account
    -> WHERE status = 'ACTIVE'
    -> GROUP BY product_cd
    -> HAVING MIN(avail_balance) >= 1000
```

```
    ->   AND MAX(avail_balance) <= 10000;
+------------+--------------+
| product_cd | prod_balance |
+------------+--------------+
| CD         |     19500.00 |
| MM         |     17045.14 |
+------------+--------------+
2 rows in set (0.00 sec)
```

This query generates total balances for each active product, but then the filter condition in the `having` clause excludes all products for which the minimum balance is less than $1,000 or the maximum balance is greater than $10,000.

Test Your Knowledge

Work through the following exercises to test your grasp of SQL's grouping and aggregating features. Check your work with the answers in Appendix C.

Exercise 8-1

Construct a query that counts the number of rows in the `account` table.

Exercise 8-2

Modify your query from Exercise 8-1 to count the number of accounts held by each customer. Show the customer ID and the number of accounts for each customer.

Exercise 8-3

Modify your query from Exercise 8-2 to include only those customers having at least two accounts.

Exercise 8-4 (Extra Credit)

Find the total available balance by product and branch where there is more than one account per product and branch. Order the results by total balance (highest to lowest).