

PHP 06

GROUP BY

HAVING

Join Queries

UNION

Nested Queries

String Functions

Based on Rasmus Lerdorf & Kevin Tatroe: Programming PHP. Sebastopol: O'Reilly, 2002; David Lane, Hugh E. Williams: Web Database Application with PHP and MySQL, 2nd Edition. Sebastopol: O'Reilly, 2004; <http://php.net>

W05/161B/Sauter

GROUP BY

The `GROUP BY` clause is different from `ORDER BY` because it doesn't sort the data for output. Instead, it sorts the data early in the query process, for the purpose of grouping or aggregation. Grouping data using a sort is the easiest way to discover properties such as maximums, minimums, averages, and counts of values.

```
SELECT city, COUNT(*) FROM customer GROUP BY city;
```

This query first sorts the rows in the customer table by `city` and groups the rows with matching values together. The output of the query consists of two columns. The first is a sorted list of unique cities. The second shows, for each city, the `COUNT` of the number of customers who live in that city. The number of rows that are output is equal to the number of different `city` values in the customer table, and the effect of `COUNT (*)` is to count the number of rows per group.

```
+-----+-----+
| city          | COUNT (*) |
+-----+-----+
| Alexandra     |          14 |
| Armidale      |           7 |
| Athlone       |           9 |
| Bauple        |           6 |
| Belmont       |          11 |
| Bentley       |          10 |
| Berala        |           9 |
```

So, for example, there are 14 customers who live in Alexandra, that is, 14 rows in the customer table are grouped together because they have a `city` value of Alexandra. The `GROUP BY` clause should be used only when the query is designed to find a characteristic of a group of rows, not the details of individual rows.

HAVING

The `HAVING` clause permits conditional aggregation of data into groups. For example, consider the following query:

```
SELECT city, count(*), min(birth_date) FROM customer GROUP BY city
HAVING count(*) > 10;
```

The query groups rows by `city`, but only for cities that have more than 10 resident customers. For those groups, the city, count of customers, and earliest birth date of a customer in that city is output. Cities with less than 10 customers are omitted from the result set. The first few rows of the output are as follows:

city	count(*)	min(birth_date)
Alexandra	14	1938-04-01
Belmont	11	1938-04-01
Broadmeadows	11	1955-10-13
Doveton	13	1943-04-04
Eleker	11	1938-04-01
Gray	12	1943-04-04

The `HAVING` clause must contain an attribute or expression (such as a function or an alias) from the `SELECT` clause; in this example, `count(*)` is listed after the `SELECT` and is used in the `HAVING` condition.

The `HAVING` clause should be used exclusively with the `GROUP BY` clause. It is slow and should never be used instead of a `WHERE` clause. For example, don't do this:

Join Queries

You'll often want to output data that's based on relationships between two or more tables. For example, in the winestore database, you might want to know which customers have placed orders, which customers live in Australia, or how many bottles of wine Lucy Williams has bought. These are examples of join queries, queries that match rows between tables based (usually) on primary key values. In SQL, a join query matches rows from two or more tables based on a condition in a `WHERE` clause and outputs only those rows that meet the condition.

Join Queries

In the database tables, the relationship between the winery and region tables is maintained using the primary key of the region table, the attribute `region_id` that's also an attribute in the winery table. To understand this, consider the first three rows from the winery table:

```
mysql> SELECT * FROM winery LIMIT 3;
+-----+-----+-----+
| winery_id | winery_name           | region_id |
+-----+-----+-----+
|          1 | Hanshaw Estates Winery |          2 |
|          2 | De Morton and Sons Wines |          5 |
|          3 | Jones's Premium Wines  |          3 |
+-----+-----+-----+
```

The first winery has a `region_id` of 2, the second a `region_id` of 5, and the third a `region_id` of 3. Consider now the first five rows of the region table:

```
mysql> SELECT * FROM region LIMIT 5;
+-----+-----+
| region_id | region_name           |
+-----+-----+
|          1 | All                   |
|          2 | Goulburn Valley      |
|          3 | Rutherglen           |
|          4 | Coonawarra           |
|          5 | Upper Hunter Valley  |
+-----+-----+
```

Join Queries

```
SELECT winery_name, region_name FROM winery, region
WHERE winery.region_id = region.region_id
ORDER BY winery_name;
```

Several features are shown in this example:

The `FROM` clause contains the two table names `winery` and `region`, and so retrieves rows from both tables.

Attributes in the `WHERE` clause are specified using both the table name and attribute name, separated by a period. This is useful because the same attribute name is often used in different tables, and the query can't figure out which table is meant unless you include it. When an attribute name occurs in only one table, you can omit the table name.

In this example, `region_id` in the `region` table and `region_id` in the `winery` table have to be specified unambiguously as `region.region_id` and `winery.region_id`. In contrast, `winery_name` and `region_name` don't need the table name because they occur only in the `winery` and `region` tables respectively.

The use of both the table and attribute name can also be used for clarity in queries, even if it isn't required. So, for example, you could write `winery.winery_name` in the example query. It can also be used in all parts of the query, not just the `WHERE` clause.

The `WHERE` clause includes a join clause that matches rows between the multiple tables. In this example, the output is reduced to those rows where wineries and regions have matching `region_id` attributes, resulting in a list of all wineries and which region they are located in. This is the key to joining two or more tables to produce sensible results.

UNION

The `UNION` clause allows you to combine the results of two or more queries. If there are occasions where it's not possible to write one query that'll do a task (using `WHERE`, `GROUP BY` AND `HAVING`), the `UNION` clause sometimes saves merging results manually after two queries have been executed.

To use `UNION`, you need to have attributes of the same type listed in the same order in the `SELECT` statement. Consider a simple example where we want to list the three oldest and three newest customers from the customer table:

```
(SELECT cust_id, surname, firstname FROM customer ORDER BY cust_id LIMIT 3)
UNION
(SELECT cust_id, surname, firstname FROM customer ORDER BY cust_id DESC
LIMIT 3);
```

```
+-----+-----+-----+
| cust_id | surname   | firstname |
+-----+-----+-----+
|      1  | Rosenthal | Joshua    |
|      2  | Serrong   | Martin    |
|      3  | Leramonth| Jacob     |
|     650  | Woodburne| Lynette   |
|     649  | Krennan   | Jim       |
|     648  | Cassisi   | Betty     |
+-----+-----+-----+
```

You can also combine queries from different tables, with different attributes of the same type.

```
(SELECT winery_name FROM winery) UNION (SELECT region_name FROM region);
```

Nested Queries

Nested Queries are powerful, but a little harder to learn.

Consider an example nested query that finds the names of the wineries that are in the Margaret River region:

```
SELECT winery_name FROM winery WHERE region_id = (SELECT region_id FROM region WHERE region_name = "Margaret River");
```

The inner query (the one in brackets) returns the `region_id` value of the Margaret River region. The outer query (the one listed first) finds the `winery_name` values from the winery table where the `region_id` matches the result of the inner query.

You can nest to any level, as long as you get the brackets right. Here's another example that finds the name of the region that makes wine #17:

```
SELECT region_name FROM region WHERE region_id = (SELECT region_id FROM winery WHERE winery_id = (SELECT winery_id FROM wine WHERE wine_id = 17));
```

Both of our previous examples can be easily rewritten as a single query with a `WHERE` clause and an `AND` operator. Indeed, you should always try to write join queries where possible and avoid nesting unless you need it; MySQL isn't good at optimizing nested queries and they are therefore usually slower to run. However, sometimes, you need a nested query.

String Functions

```
SELECT 'Apple' LIKE 'A%';  
1
```

```
SELECT 'Apple' LIKE 'App%';  
1
```

```
SELECT 'Apple' LIKE 'A%l%';  
1
```

```
SELECT concat('con','cat');  
concat
```

```
SELECT concat('con','c','at');  
concat
```

```
SELECT concat_ws(", ", "Williams", "Lucy");  
Williams, Lucy
```

```
SELECT length('Apple');  
5
```

```
SELECT locate('pp','Apple');  
2
```

```
SELECT lower('Apple');  
apple
```

String Functions

```
SELECT ltrim('  Apple');  
Apple
```

```
SELECT rtrim('Apple  ');  
Apple
```

```
SELECT trim('  Apple  ');  
Apple
```

```
SELECT replace('The Web', 'Web', 'WWW');  
The WWW
```

```
SELECT upper('Apple');  
APPLE
```

...

More information at: <http://dev.mysql.com/doc/mysql/en/string-functions.html>