Table of Contents

1. Partitioning the Table Expression with Group By 2
1.1. Multiple-column grouping 3
1.2. Grouping by expressions 4
2. Using Groups and Aggregate Functions 5
3. Using the Having Clause 8
3.1. Grouping on additional columns 10
4. Group By and Nulls 12
5. Sorting 13
6. SQL layout 13

In the previous documents, we did aggregates on the table as a whole- getting one row of data from the query. We also have the ability to take a table and partition it into a set of sub-tables. With a partition, each row of the table is put into exactly one of the sub-tables- into one of the groups. We can then use an aggregate function with each of these subsets.

Amazon might want to know the total sales by month for each month of the year or average shipping costs it incurs by zip code. This requires partitioning the data in the tables by some characteristic (sales month or zip code) and then calculating summary data for each of these groups.

This uses a new clause in our Select statement- the **Group By** clause. This would let us find the most expensive product in each of our product categories; we can group by the category and use Max for each group.

At times we may want to see aggregate values only if the aggregate itself meets a specified criterion- for this we have another new clause- the **Having** clause. For example, we might want to see customers who have more than 5 orders in our tables.

At the end of this unit, we will have the following model for the Select statement.

```
select . . .
from . . .
where . . .
group by . . .
having . . .
order by . . .
```

In a previous document, we mentioned the Logical processing order. You need to understand that in order to understand how your query is processed. Adding in these two new clauses the order is:

1. The FROM clause is evaluated first
2. The WHERE clause
3. The GROUP BY clause
4. The HAVING clause
5. The SELECT clause
6. The ORDER BY clause is done last

When the query is presented to the dbms, the parser and optimizer determine the actual steps used in the execution and you do not directly control that. But you should consider the statement as being processed in that order.

Please note there is another document in this unit on some MySQL features of the Group by clause.

1. Partitioning the Table Expression with Group By

When you partition a table expression, you create a series of virtual sub-tables/partitions into which each row from the parent table will be placed; each table row appears in only one of these groups. You can partition the table on the basis of one or more attributes or expressions. When you apply an aggregate function to a partitioned table, you will get a separate summary value for each sub-table rather than for the table as a whole.

We start here with just the Group By clause without any aggregates.

Demo 01: Displaying all departments— one per employee.

```
select dept_id
from Employee.employees
order by dept_id;
+---------+
| dept_id |
+---------+
|      10 |
|      20 |
|      30 |
|      30 |
|      30 |
|      30 |
|      30 |
|      30 |
|      30 |
|      30 |
|      35 |
|      35 |
|      35 |
|      80 |
|      80 |
|      80 |
|     210 |
|     210 |
|     215 |
|     215 |
|     215 |
|     215 |
+---------+
```

Demo 02: Displaying one of each different department represented in the employee table.
Here we group by the dept_id and return one row for each grouping
We could also do this by using the keyword Distinct.

```
select dept_id
from Employee.employees
group by dept_id;
+---------+
| dept_id |
+---------+
|      10 |
|      20 |
|      30 |
|      35 |
|      80 |
|     210 |
|     215 |
+---------+
```

Demo 03: But we should not try to show the emp_id since there is no emp_id value that represents the department as a group. But MySQL will allow this and simply show one of the employee id values from that group. This result may not be useful and would need to be explained to the user.

```
select dept_id, emp_id
from Employee.employees
group by dept_id;
+---------+--------+
| dept_id | emp_id |
+---------+--------+
|      10 |    100 |
|      20 |    201 |
|      30 |    101 |
|      35 |    162 |
|      80 |    145 |
|     210 |    103 |
|     215 |    102 |
+---------+--------+
```

1.1. Multiple-column grouping

In the following example we have two columns in the Group By clause and we get more groups. The more grouping columns we have, the finer the distinction we are making between the way that we categorize a row and the more potential groups we will have. If we were to group by the pk columns, we would make a group for each row in the table. This is generally not a good idea.

Note that we get a sub-table only for data combinations that actually exist in our table. We have at least two rows with different job_id values in department 80 so we get two sub-tables for department 80 but for dept 10 we have only one job_id. This is **not** a Cartesian product of all possible combinations of dept_id and job_id.

Demo 04: Using two grouping attributes gives us more sub-tables.

```
select dept_id, job_id
from Employee.employees
group by dept_id, job_id;
+---------+--------+
| dept_id | job_id |
+---------+--------+
|      10 |      1 |
|      20 |      2 |
|      30 |     16 |
|      30 |     32 |
|      35 |      8 |
|      35 |     16 |
|      80 |      4 |
|      80 |      8 |
|     210 |     32 |
|     210 |     64 |
|     215 |     16 |
|     215 |     32 |
|     215 |     64 |
+---------+--------+
```

Demo 05: If we group first by the job_id and then by the dept_id, we get the same result set because we are creating the same set of subtables. We get a group for each combination of dept id and job id,

```
select dept_id, job_id
from Employee.employees
group by job_id, dept_id;
```

Demo 06: You can sort the result set after you group it. (MySQL does do a sort by the grouping columns if that is the order you want.)

```
select dept_id, job_id
from Employee.employees
group by job_id, dept_id
order by job_id, dept_id;
+---------+--------+
| dept_id | job_id |
+---------+--------+
|      10 |      1 |
|      20 |      2 |
|      80 |      4 |
|      35 |      8 |
|      80 |      8 |
|      30 |     16 |
|      35 |     16 |
|     215 |     16 |
|      30 |     32 |
|     210 |     32 |
|     215 |     32 |
|     210 |     64 |
|     215 |     64 |
+---------+--------+
```

In these queries, we have been displaying the grouping attributes. That is not a requirement, but is generally helpful.

Demo 07: This would be a confusing display since it is not obvious why some department id values appear more than once.

```
select dept_id
from Employee.employees
group by job_id, dept_id
order by dept_id;
+---------+
| dept_id |
+---------+
|      10 |
|      20 |
|      30 |
|      30 |
|      35 |
|      35 |
|      80 |
|      80 |
|     210 |
|     210 |
|     215 |
|     215 |
|     215 |
+---------+
```

1.2. Grouping by expressions

You can also use expressions in Group By clauses.

Demo 08: A common business need is to group sales by year or by year and month. This also adds in the count function.

```
select year(order_date) as OrdYear , count(*) as NmbOrders
from OrderEntry.orderHeaders
group by year(order_date)
```

```
order by year(order_date);
+---------+-----------+
| OrdYear | NmbOrders |
+---------+-----------+
|    2015 |        57 |
|    2016 |        40 |
+---------+-----------+
```

Demo 09: Grouping by year and month. There is no row for 2014 month 7 since we have no rows in the table for that month.

```
select year(order_date) as OrdYear,month(order_date) as OrdMonth , count(*) as
NmbOrders
from OrderEntry.orderHeaders
group by year(order_date),month(order_date)
order by year(order_date),month(order_date)
;
+---------+----------+-----------+
| OrdYear | OrdMonth | NmbOrders |
+---------+----------+-----------+
|    2015 |        4 |         1 |
|    2015 |        6 |        12 |
|    2015 |        8 |         8 |
|    2015 |        9 |         7 |
|    2015 |       10 |         9 |
|    2015 |       11 |        12 |
|    2015 |       12 |         8 |
|    2016 |        1 |        14 |
|    2016 |        2 |         8 |
|    2016 |        3 |         5 |
|    2016 |        4 |         7 |
|    2016 |        5 |         6 |
+---------+----------+-----------+
```

Demo 10: Using a subquery in the From clause might make this easier to think about by making the Group By and Order By clauses easier to write.

```
select OrdYear, OrdMonth, count(*)
from (
   select year(order_date) as OrdYear
   , month(order_date) as OrdMonth
   from OrderEntry.orderHeaders
)orderdates
group by OrdYear, OrdMonth
order by OrdYear, OrdMonth ;
```

2. Using Groups and Aggregate Functions

We want to create the sub-tables so that we could get information about the groupings. In this example, we want to know how many employees are in each department and their average salary. Since we want data for **each** department, we group by the department id.

Demo 11: For each department, how many employees are there and what is the average salary for that department?

```
select dept_id
, count(*) as empCount
, avg(salary) as AvgSalary
from Employee.employees
group by dept_id
;
```

```
+---------+----------+---------------+
| dept_id | empCount | AvgSalary     |
+---------+----------+---------------+
|      10 |        1 | 100000.000000 |
|      20 |        1 |  15000.000000 |
|      30 |        8 |  76599.875000 |
|      35 |        3 |  64333.333333 |
|      80 |        3 |  36000.000000 |
|     210 |        2 |  67000.000000 |
|     215 |        4 |  83563.500000 |
+---------+----------+---------------+
```

Demo 12: What is the average list price and the largest list price for **each category** of product we sell?

```
select catg_id
, avg(prod_list_price) as "Avg List Price"
, max(prod_list_price) as "Max List Price"
from Product.products
group by catg_id;
+---------+----------------+----------------+
| catg_id | Avg List Price | Max List Price |
+---------+----------------+----------------+
| APL     |     479.986000 |         850.00 |
| GFD     |       8.750000 |          12.50 |
| HD      |      23.875000 |          45.00 |
| HW      |      67.641000 |         149.99 |
| MUS     |      13.878182 |          15.95 |
| PET     |     123.192727 |         549.99 |
| SPG     |     178.125000 |         349.95 |
+---------+----------------+----------------+
```

Demo 13: Using two groups and aggregate functions; this groups by the different departments and job ids. It then sorts the rows by the avg salary.

```
select dept_id, job_id
, count(*) as NumEmployee
, avg(salary) as AvgSalary
from Employee.employees
group by dept_id, job_id
order by avg(salary);
+---------+--------+-------------+---------------+
| dept_id | job_id | NumEmployee | AvgSalary     |
+---------+--------+-------------+---------------+
|      20 |      2 |           1 |  15000.000000 |
|      80 |      8 |           2 |  24500.000000 |
|      35 |      8 |           1 |  30000.000000 |
|      80 |      4 |           1 |  59000.000000 |
|     210 |     32 |           1 |  65000.000000 |
|     215 |     32 |           1 |  65000.000000 |
|     210 |     64 |           1 |  69000.000000 |
|     215 |     64 |           2 |  74627.000000 |
|      30 |     16 |           4 |  74863.750000 |
|      30 |     32 |           4 |  78336.000000 |
|      35 |     16 |           2 |  81500.000000 |
|      10 |      1 |           1 | 100000.000000 |
|     215 |     16 |           1 | 120000.000000 |
+---------+--------+-------------+---------------+
```

Demo 14: Suppose we wanted to show statistics for average salaries by department- but not identify the department by name (for political reasons!) It is up to the user if this output is useful.

```
select 'not specified' as Department,count(*) as EmployeeeCount
, Round(avg(salary),0) as AvgSalary
from Employee.employees
group by dept_id
order by avg(salary);
+---------------+---------------+-----------+
| Department    | EmployeeeCount | AvgSalary |
+---------------+---------------+-----------+
| not specified |              1 |     15000 |
| not specified |              3 |     36000 |
| not specified |              3 |     64333 |
| not specified |              2 |     67000 |
| not specified |              8 |     76600 |
| not specified |              4 |     83564 |
| not specified |              1 |    100000 |
+---------------+---------------+-----------+
```

In the previous document we found the most expensive SPG item and we found the most expensive PET item. What if we want to find the most expensive item(s) in **each** category?

Demo 15: We can use this query to find the max price in each category

```
select catg_id, max(prod_list_price) as MaxPrice
from Product.products
group by catg_id ;
+---------+----------+
| catg_id | MaxPrice |
+---------+----------+
| APL     |   850.00 |
| GFD     |    12.50 |
| HD      |    45.00 |
| HW      |   149.99 |
| MUS     |    15.95 |
| PET     |   549.99 |
| SPG     |   349.95 |
+---------+----------+
```

Demo 16: We can use this query as an inline view and join to the product table on the category id and then use a test that price equals the max price for that category. Note that we do get ties.

```
select p.catg_id, p.prod_id, left(p.prod_desc, 25) as prod_desc
, p.prod_list_price
from Product.products  P
join (
  select catg_id, max(prod_list_price) as MaxPrice
  from Product.products
  group by catg_id
  ) MP on p.catg_id = MP.catg_id
and p.prod_list_price = MP.maxPrice;
+---------+---------+--------------------------+-----------------+
| catg_id | prod_id | prod_desc                | prod_list_price |
+---------+---------+--------------------------+-----------------+
| APL     |    1126 | Low Energy Washer Dryer c |          850.00 |
| GFD     |    5000 | Cello bag of mixed finger |           12.50 |
| HD      |    5005 | Steel Shingler hammer     |           45.00 |
| HW      |    1090 | Gas grill                 |          149.99 |
| HW      |    1160 | Stand Mixer with attachme |          149.99 |
| MUS     |    2014 | Bix Beiderbecke - Tiger R |           15.95 |
| PET     |    4567 | Our highest end cat tree- |          549.99 |
| PET     |    4568 | Satin four-poster cat bed |          549.99 |
```

```
| SPG     |    1040 | Super Flyer Treadmill     |          349.95 |
+---------+---------+---------------------------+-----------------+
```

3. Using the Having Clause

Sometimes you have grouped your rows but you do not want to see all of the groups displayed. Perhaps you have grouped employees by department and only want to see those departments where the average salary is more than 60,000. In this case you want to filter the returns by the aggregate function return values. **It is not allowed to put an aggregate function in the Where clause.** So we need another clause to do this- the Having clause. The Having clause filters the groups. In most cases you will use aggregate functions in the Having clause.

Demo 17: This groups by department and uses all rows.

```
select dept_id
, count(*) as EmpCount
, avg(salary) as AvgSalary
from Employee.employees
group by dept_id;
+---------+----------+---------------+
| dept_id |   EmpCount| AvgSalary     |
+---------+----------+---------------+
|      10 |         1 | 100000.000000 |
|      20 |         1 |  15000.000000 |
|      30 |         8 |  76599.875000 |
|      35 |         3 |  64333.333333 |
|      80 |         3 |  36000.000000 |
|     210 |         2 |  67000.000000 |
|     215 |         4 |  83563.500000 |
+---------+----------+---------------+
```

Demo 18: This groups by department and returns groups where the average salary exceed 60000. The test on avg(salary) is in the HAVING clause.

```
select dept_id
, count(*) as EmpCount
, avg(salary)
from Employee.employees
group by dept_id
having avg(salary) > 60000;
+---------+----------+---------------+
| dept_id | EmpCount | AvgSalary     |
+---------+----------+---------------+
|      10 |        1 | 100000.000000 |
|      30 |        8 |  76599.875000 |
|      35 |        3 |  64333.333333 |
|     210 |        2 |  67000.000000 |
|     215 |        4 |  83563.500000 |
+---------+----------+---------------+
```

If you have a choice of filtering in the Where clause or in the Having clause- pick the Where clause. Suppose we want to see the average salary for department where the average is more than 60000 but we only want to consider department 30-40. In that case we should filter on the dept_id in the Where clause and filter on the Ave(salary) in the Having clause. There is no sense creating groups for departments that we do not want to consider.

Demo 19: Groups with Where clause and a Having clause

```
select dept_id
, count(*) as EmpCount
, avg(salary) as AvgSalary
from Employee.employees
where dept_id between 30 and 40
group by dept_id
having avg(salary) > 600000;
+---------+----------+-------------+
| dept_id |  EmpCount | AvgSalary    |
+---------+----------+-------------+
|      30 |         8 | 76599.875000 |
|      35 |         3 | 64333.333333 |
+---------+----------+-------------+
```

There are several different meanings to finding averages. The following query allows only employees who earn more than 60000 to be put into the groups- and we get a different answer. The issue here is not "which is the right query?" but rather "what do you want to know?" Do you want to know the average salary of the more highly paid employees or the departments with the higher average salaries?

Demo 20: Using a Where clause— this acts before the grouping. Only employees earning more than 60000 get into the groups.

```
select dept_id
, count(*) as EmpCount
, avg(salary) as AvgSalary
from Employee.employees
where salary > 60000
group by dept_id;
+---------+----------+--------------+
| dept_id |   EmpCount| AvgSalary     |
+---------+----------+--------------+
|      10 |         1 | 100000.000000 |
|      30 |         8 |  76599.875000 |
|      35 |         2 |  81500.000000 |
|     210 |         2 |  67000.000000 |
|     215 |         4 |  83563.500000 |
+---------+----------+--------------+
```

Demo 21: Show statistics only if the department has more than three employees.

```
select dept_id
, count(*) as EmpCount
, avg(salary) as AvgSalary
from Employee.employees
group by dept_id
having count(*) > 3 ;
+---------+----------+-------------+
| dept_id |  EmpCount | AvgSalary    |
+---------+----------+-------------+
|      30 |         8 | 76599.875000 |
|     215 |         4 | 83563.500000 |
+---------+----------+-------------+
```

Demo 22: What if you want to find out if you have more than one employee with the same last name?

```
select name_last as DuplicateName
, count(*) as NumEmp
from Employee.employees
group by name_last
having count(*) > 1;
```

```
+---------------+--------+
| DuplicateName | NumEmp |
+---------------+--------+
| King          |      2 |
| Russ          |      2 |
+---------------+--------+
```

Demo 23: If you want to determine the number of order lines for each order, you can use the order details table and group on the ord_id. How many order lines for each order?

```
select order_id
, count(*) AS "NumberLineItems"
from OrderEntry.orderDetails
group by order_id;
```

selected rows

```
+--------+-----------------+
| ord_id | NumberLineItems |
+--------+-----------------+
|    105 |               3 |
|    106 |               1 |
|    107 |               1 |
|    110 |               2 |
|    111 |               2 |
|    112 |               1 |
|    115 |               4 |
```

3.1. Grouping on additional columns

But suppose I also wanted to show the customer ID and the shipping mode for each of these orders. You know that each order has a single cust_id, and shipping mode, so it would be logical that you could just add those attributes to the Select clause. MySQL allows that syntax..

Demo 24: This is a more traditional syntax for the query. This query adds extra group levels that do not affect the actual groups formed. For any order_id there is exactly one value for the cust_id and one for the shipping mode.

```
select customer_id
, order_id
, shipping_mode_id
, count(*) AS "NumberLineItems"
from OrderEntry.orderHeaders
join OrderEntry.orderDetails using(order_id)
group by order_id, customer_id, shipping_mode_id
order by customer_id, order_id;
+-------------+----------+------------------+-----------------+
| customer_id | order_id | shipping_mode_id | NumberLineItems |
+-------------+----------+------------------+-----------------+
|      400300 |      378 | USPS1            |               2 |
|      401250 |      106 | FEDEX1           |               1 |
|      401250 |      113 | FEDEX2           |               1 |
|      401250 |      119 | NULL             |               1 |
|      401250 |      301 | FEDEX2           |               1 |
|      401250 |      552 | FEDEX1           |               2 |
|      401890 |      112 | USPS1            |               1 |
|      401890 |      519 | USPS1            |               2 |
|      402100 |      114 | USPS1            |               1 |
|      402100 |      115 | USPS2            |               4 |
```

Demo 25: What is the amount due for each order? Note that I am able to use ord_date in the group by clause and use ord_date within a function in the Select ; the ord_date attribute is a datetime value.

```
select
  customer_id
, customer_name_last
, order_id
, CAST(order_date As date) As OrderDate
, SUM(Quantity_ordered * quoted_price) As AmntDue
from Customer.customers
join OrderEntry.orderHeaders using(customer_id)
join OrderEntry.orderDetails  using(order_id)
group by order_id, customer_id, customer_name_last, order_date
order by customer_id, order_id;
+-------------+--------------------+----------+------------+---------+
| customer_id | customer_name_last | order_id | OrderDate  | AmntDue |
+-------------+--------------------+----------+------------+---------+
|      400300 | McGold             |      378 | 2015-06-14 | 4500.00 |
|      401250 | Morse              |      106 | 2015-10-01 |  255.95 |
|      401250 | Morse              |      113 | 2015-11-08 |   22.50 |
|      401250 | Morse              |      119 | 2015-11-28 |  225.00 |
|      401250 | Morse              |      301 | 2015-06-04 |  205.00 |
|      401250 | Morse              |      552 | 2015-08-12 |  157.30 |
|      401890 | Northrep           |      112 | 2015-11-08 |   99.98 |
|      401890 | Northrep           |      519 | 2016-04-04 |  114.74 |
|      402100 | Morise             |      114 | 2015-11-08 |  625.00 |
|      402100 | Morise             |      115 | 2015-11-08 | 2305.00 |
|      402100 | Morise             |      117 | 2015-11-28 |  346.96 |
```

Demo 26: It does allow the following which produces one group per order date and displays the year for that group. It is up to you to decide if this output is meaningful.

```
elect year(order_date) as OrdYear , count(*) as NmbOrders
from OrderEntry.orderHeaders
group by order_date;
+---------+-----------+
| OrdYear | NmbOrders |
+---------+-----------+
|    2015 |         1 |
|    2015 |         1 |
|    2015 |         6 |
|    2015 |         2 |
|    2015 |         1 |
|    2015 |         1 |
|    2015 |         1 |
|    2015 |         2 |
```

It probably makes more sense to group by the year of the order date.

```
select year(order_date) as OrdYear , count(*) as NmbOrders
from OrderEntry.orderHeaders
group by year(order_date);
+---------+-----------+
| OrdYear | NmbOrders |
+---------+-----------+
|    2015 |        57 |
|    2016 |        40 |
+---------+-----------+
```

Demo 27: This one produces a group for each combination of first and last name and then allows you to use those grouping expressions in the Select within a function. Note that we have several situtaions where we have more than one customer with the same name.

```
select
   concat(case when customer_name_first is null then '' else
concat(customer_name_first,' ') end ,  customer_name_last) As Customer
, count(*) As NbrOrders
from Customer.customers
group by customer_name_last, customer_name_first;
```

selected rows

```
+-------------------+-----------+
| Customer          | NbrOrders |
+-------------------+-----------+
| William Max       |         1 |
| Barry Mazur       |         1 |
| Tyner McCoy       |         1 |
| Arnold McGold     |         3 |
| Morris Morise     |         1 |
| William Morise    |         1 |
| William Morris    |         2 |
```

4. Group By and Nulls

Demo 28: The order headers table has several rows where there is no value for the shipping mode. What happens if we group by the column shipping mode?

```
select
  shipping_mode_id
, count(order_id) as OrderCount
from OrderEntry.orderHeaders
group by shipping_mode_id;
+------------------+------------+
| shipping_mode_id | OrderCount |
+------------------+------------+
| NULL             |          8 |
| FEDEX1           |         26 |
| FEDEX2           |          2 |
| UPSEXP           |          5 |
| UPSGR            |         15 |
| USPS1            |         38 |
| USPS2            |          3 |
+------------------+------------+
```

When we group by the shipping mode, we get one group for all of the nulls. We have eight rows in that group.

Demo 29: What if we want to display a message instead of a null for the null group?

```
select
  coalesce(shipping_mode_id, 'No shipping mode')  as ShippingMode
, count(order_id) as OrderCount
from OrderEntry.orderHeaders
group by shipping_mode_id;
+------------------+------------+
| ShippingMode     | OrderCount |
+------------------+------------+
| No shipping mode |          8 |
| FEDEX1           |         26 |
| FEDEX2           |          2 |
| UPSEXP           |          5 |
```

```
| UPSGR            |         15 |
| USPS1            |         38 |
| USPS2            |          3 |
+------------------+------------+
```

5. Sorting

Demo 30: Suppose we take the previous query and try to sort by the ord_id. MySQL allows this- but the output does not seem to be sorted. MySQL is using 1 rows from each group to use as the sort key.

```
select
  coalesce(shipping_mode_id, 'No shipping mode')  as ShippingMode
, count(order_id) as OrderCount
from OrderEntry.orderHeaders
group by shipping_mode_id
order by order_id;
+------------------+------------+
| ShippingMode     | OrderCount |
+------------------+------------+
| UPSGR            |         15 |
| FEDEX1           |         26 |
| USPS1            |         38 |
| FEDEX2           |          2 |
| USPS2            |          3 |
| No shipping mode |          8 |
| UPSEXP           |          5 |
+------------------+------------+
```

6. SQL layout

For a query that uses grouping, the required SQL layout has the Group By clause and the Having clause on new lines

```
select dept_id
, avg(salary) as avgsalary
from Employee.employees
group by dept_id
having count(*) > 1;
```