Table of Contents

1. MySQL version 7

This class is based on MySQL version 5.6 and the rest of this document applies to that version. Some people have installed version 5.7. MySQL made some changes to the way it handles the Group By clause.

Per manual: If the `ONLY_FULL_GROUP_BY` SQL mode is enabled (which it is by default), MySQL rejects queries for which the select list, `HAVING` condition, or `ORDER BY` list refer to nonaggregated columns that are neither named in the `GROUP BY` clause nor are functionally dependent on them.

What that means is that you will probably not be able to use the non-traditional features discussed here, if you are using version 5.7. You are determine your version by running the task 00 in the script template.

Some individual dot version of version 5.7 may follow somewhat different rules. But you should be able to run your queries by using the traditional rules shown in the previous documents.

2. Non-traditional Group By

MySQL extends the Group By clause in a non-traditional way that you should carefully consider. There is nothing wrong with this extension and it can be very useful, but you need to take the responsibility to avoid logic errors.

Demo 01: Create the following table in a_testbed.

```
Create table a_testbed.z_tst_group (
  an_id integer primary key
, an_name varchar(15) not null
, an_type varchar(15) not null
, an_price integer not null);

Insert into a_testbed.z_tst_group values
(1, 'Max',  'dog', 500),    (2, 'Max',  'cat', 25),
(3, 'Max',  'dog', 350),    (4, 'Spot', 'dog', 400),
(5, 'Spot', 'snake', 125),  (6, 'Spot', 'dog', 400),
(7, 'Max',  'elephant', 75000);

select *
From a_testbed.z_tst_group;
+-------+---------+----------+----------+
| an_id | an_name | an_type  | an_price |
+-------+---------+----------+----------+
|     1 | Max     | dog      |      500 |
|     2 | Max     | cat      |       25 |
|     3 | Max     | dog      |      350 |
|     4 | Spot    | dog      |      400 |
|     5 | Spot    | snake    |      125 |
|     6 | Spot    | dog      |      400 |
|     7 | Max     | elephant |    75000 |
+-------+---------+----------+----------+
```

We could group these rows by the animal name. This output makes sense- we have three animals named Spot and 4 named Max.

Demo 02: Group by an_name, display the grouping column and an aggregate. There are two different name values in the table and we get two groups.

```
select an_name, count(*)
from a_testbed.z_tst_group
group by an_name;
+---------+----------+
| an_name | count(*) |
+---------+----------+
| Max     |        4 |
| Spot    |        3 |
+---------+----------+
```

We could do the following query. This is a non-traditional extension of the Group By that MySQL allows.

Demo 03: Group by an_name, display the grouping column and aggregate and non-aggregate columns

```
select an_name, an_type, an_price, count(*)
from a_testbed.z_tst_group
group by an_name;
+---------+---------+----------+----------+
| an_name | an_type | an_price | count(*) |
+---------+---------+----------+----------+
| Max     | dog     |      500 |        4 |
| Spot    | dog     |      400 |        3 |
+---------+---------+----------+----------+
```

But what does this output mean? We have a group for animals named Max- but why does this query return an_type dog and an_price 500. Why not an_type elephant and an_price 75000?
So how did MySQL decide which type and price to return for the group of animals named Max? The manual says that it can return any one of the values from that group that it feels like. Kinda a bad attitude!

Demo 04: Here you group by all of the non-aggregated attributes. We have one group that has both of the dogs named Spot that cost 400. The other groups contain one row each.

```
select an_name, an_type, an_price, count(*)
from a_testbed.z_tst_group
group by an_name, an_type, an_price;
+---------+----------+----------+----------+
| an_name | an_type  | an_price | count(*) |
+---------+----------+----------+----------+
| Max     | cat      |       25 |        1 |
| Max     | dog      |      350 |        1 |
| Max     | dog      |      500 |        1 |
| Max     | elephant |    75000 |        1 |
| Spot    | dog      |      400 |        2 |
| Spot    | snake    |      125 |        1 |
+---------+----------+----------+----------+
```

Demo 05: Here we display the an_name which is the grouping attribute and use an aggregate for price- this finds the most expensive price for animals of each name.

```
select an_name,  max(an_price), count(*)
from a_testbed.z_tst_group
group by an_name;
+---------+---------------+----------+
| an_name | max(an_price) | count(*) |
+---------+---------------+----------+
| Max     |         75000 |        4 |
| Spot    |           400 |        3 |
+---------+---------------+----------+
```

We know that we can have groups based on the animal name and that for each of those groups there is a single value that is the largest value of an_price in that group and there is a single value that is the number of rows in that group.

If you are grouping by an attribute and you know from the design of the query and tables that all of the rows in that group will have the same values for another attribute, then you can display that other attribute in the Select without an aggregate function. A typical grouping for this would be the primary key for the table.

The following two queries will return the same record set. The Group By in the first query is traditional SQL and the Group By in the second query uses the MySQL extension. The reason the second query works properly is that we know that if we group by the order_id, all of the rows in each group will have the same cust_id and the same shipping mode.

Demo 06: Traditional syntax

```
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;
```

Demo 07: using the MySQL version

```
select customer_id
, order_id
, shipping_mode_id
, count(*) AS "NumberLineItems"
from OrderEntry.orderHeaders
join OrderEntry.orderDetails using (order_id)
group by order_id
order by customer_id, order_id;
```

We are grouping by the order id; we know from the design of the tables that all of the rows in a order id group will have the same values for the customer id and for the shipping mode. These attributes come from the orders table and there is only one customer id per order and only one shipping mode per order and the order id uniquely identifies the order. The query has added in additional rows as it joined to the order details table but those extra rows are reflected in the count. So it makes sense that we could display the customer ID and shipping mode.

Demo 08: This is a query we did earlier to find the amount due for each order

```
select customer_id, customer_name_last, order_id
, order_date
, 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, cast(order_date as date)
order by customer_id, order_id;
```

Demo 09: We could do this by grouping on just the order id and get the same results. (I get 94 rows with my current data set.)

```
select customer_id, customer_name_last, order_id
, order_date
, 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
order by customer_id, order_id;
```

But suppose we decided to group on the order date; that is also a column in the order table and each order has a single order date. When I run this query I get 64 rows.

Demo 10: Incorrect grouping to find the amount due per order

```
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_date
order by customer_id, order_id;
```

It will be easier to see what happened if we filter on only one order date.

Demo 11: This is grouping by the order id and returns six rows- one for each order for '2015-06-04'

```
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)
where order_date = '2015-06-04'
group by order_id
order by customer_id, order_id;
+---------+----------------+--------+------------+---------+
| cust_id | cust_name_last | ord_id | OrderDate  | amntdue |
+---------+----------------+--------+------------+---------+
|  401250 | Morse          |    301 | 2015-06-04 |  205.00 |
|  403000 | Williams       |    390 | 2015-06-04 | 1400.00 |
|  403000 | Williams       |    395 | 2015-06-04 | 2925.00 |
|  404000 | Olmsted        |    302 | 2015-06-04 |  469.95 |
|  900300 | McGold         |    307 | 2015-06-04 | 4500.00 |
|  903000 | McGold         |    306 | 2015-06-04 | 1000.00 |
+---------+----------------+--------+------------+---------+
```

Demo 12: This is grouping by the order date and returns one row for '2015-06-04'. The amount due is the sum of the amount due for all orders for that date. Customer 401250 is not happy being charged for all of those orders. The other customers seem to be getting their orders at no cost.
Note that MySQL lets you group by the column alias.

```
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)
where order_date = '2015-06-04'
group by  cast(order_date as date)
```

```
order by customer_id, order_id;
+---------+----------------+--------+------------+----------+
| cust_id | cust_name_last | ord_id | OrderDate  | amntdue  |
+---------+----------------+--------+------------+----------+
|  401250 | Morse          |    301 | 2015-06-04 | 10499.95 |
+---------+----------------+--------+------------+----------+
```

The order date does not identify orders so it is not a good candidate for the Group By if you want an amount due per order. If you want to see the amount of the orders on each date, then a Group By order date makes sense; but then you should not display the customer id or name or order id.

Demo 13: Order total by Order date

```
select   cast(order_date as date) as OrderDate
, count(*) as NumberOfOrders
, sum( quantity_ordered * quoted_price) as amntdue
from OrderEntry.orderHeaders
join OrderEntry.orderDetails using (order_id)
group by  cast(order_date as date)
order by OrderDate;
+------------+----------------+----------+
| OrderDate  | NumberOfOrders | amntdue  |
+------------+----------------+----------+
| 2015-04-05 |              1 |    45.00 |
| 2015-06-02 |              3 |   150.24 |
| 2015-06-04 |              9 | 10499.95 |
| 2015-06-07 |              5 |  9530.00 |
| 2015-06-10 |              1 |   125.00 |
| 2015-06-11 |              1 |   599.00 |
| 2015-06-14 |              2 |  4500.00 |
| 2015-08-01 |              4 |   760.39 |
| 2015-08-02 |              2 |  2050.25 |
| 2015-08-03 |              1 |  3500.00 |
| 2015-08-07 |              3 |   657.20 |
```

This is the way the MySQL manual explains this:

"In standard SQL, a query that includes a `GROUP BY` clause cannot refer to nonaggregated columns in the select list that are not named in the `GROUP BY` clause. … MySQL extends the use of `GROUP BY` so that the select list can refer to nonaggregated columns not named in the `GROUP BY` clause. However, this is useful primarily when all values in each nonaggregated column not named in the `GROUP BY` are the same for each group. The server is free to choose any value from each group, so unless they are the same, **the values chosen are indeterminate**. Furthermore, the selection of values from each group cannot be influenced by adding an `ORDER BY` clause. Sorting of the result set occurs after values have been chosen, and `ORDER BY` does not affect which values within each group the server chooses."

There is a way to disable this feature- but for now I want you to work with the default settings for MySQL

3. Group_Concat function

MySQL has a function Group_Concat which you can use to get a comma separated list of the values in a group. This function skips any null values in the list. The list is limited to a default length of 1024.

Demo 14: Here we Group by the an_type and get a list of all of the animals names in that group.

```
select an_type, count(*)
, group_concat(an_name)
from a_testbed.z_tst_group
group by an_type;
```

```
+----------+----------+-----------------------+
| an_type  | count(*) | group_concat(an_name) |
+----------+----------+-----------------------+
| cat      |        1 | Max                   |
| dog      |        4 | Max,Max,Spot,Spot     |
| elephant |        1 | Max                   |
| snake    |        1 | Spot                  |
+----------+----------+-----------------------+
```

Demo 15: You can add distinct to get distinct animal names.

```
select an_type, count(*)
, group_concat(distinct an_name)
from a_testbed.z_tst_group
group by an_type;
+----------+----------+--------------------------------+
| an_type  | count(*) | group_concat(distinct an_name) |
+----------+----------+--------------------------------+
| cat      |        1 | Max                            |
| dog      |        4 | Max,Spot                       |
| elephant |        1 | Max                            |
| snake    |        1 | Spot                           |
+----------+----------+--------------------------------+
```

Demo 16: You can group_concat on prices

```
select an_type, count(*)
, group_concat(an_price order by an_price )
from a_testbed.z_tst_group
group by an_type;
+----------+----------+-------------------------------------------+
| an_type  | count(*) | group_concat(an_price order by an_price ) |
+----------+----------+-------------------------------------------+
| cat      |        1 | 25                                        |
| dog      |        4 | 350,400,400,500                           |
| elephant |        1 | 75000                                     |
| snake    |        1 | 125                                       |
+----------+----------+-------------------------------------------+
```

Demo 17: You can group_concat an expression

```
select an_type, count(*)
, group_concat(concat(an_name, ' at $', an_price) )
from a_testbed.z_tst_group
group by an_type;
+----------+----------+---------------------------------------------------+
| an_type  | count(*) | group_concat(concat(an_name, ' at $', an_price) ) |
+----------+----------+---------------------------------------------------+
| cat      |        1 | Max at $25                                        |
| dog      |        4 | Max at $500,Max at $350,Spot at $400,Spot at $400 |
| elephant |        1 | Max at $75000                                     |
| snake    |        1 | Spot at $125                                      |
+----------+----------+---------------------------------------------------+
```

Demo 18: Group_concat with and without Distinct

```
select cast(order_date as date) as OrdDate
, group_concat(prod_id order by prod_id) as ProductsSold
, group_concat(distinct prod_id order by prod_id ) as DistinctProductsSold
from Customer.customers
join OrderEntry.orderHeaders using (customer_id)
join OrderEntry.orderDetails using (order_id)
where year(order_date) = 2015 and quarter(order_date) = 4
```

```
group by cast(order_date as date);
+------------+------------------------------------+-------------------------------+
| OrdDate    | ProductsSold                       | DistinctProductsSold          |
+------------+------------------------------------+-------------------------------+
| 2015-10-01 | 1010,1020,1030,1060                | 1010,1020,1030,1060           |
| 2015-10-02 | 1080,1110                          | 1080,1110                     |
| 2015-10-12 | 1090,1130,1130                     | 1090,1130                     |
| 2015-10-15 | 4577,5002,5002,5004,5005,5008      | 4577,5002,5004,5005,5008      |
| 2015-10-18 | 4577,5002                          | 4577,5002                     |
| 2015-11-01 | 1141,1150                          | 1141,1150                     |
| 2015-11-08 | 1000,1080,1080,1100,1110,1120,1130 | 1000,1080,1100,1110,1120,1130 |
| 2015-11-15 | 5005,5005,5008                     | 5005,5008                     |
| 2015-11-19 | 1010,1020,1030,1040,1050,1060      | 1010,1020,1030,1040,1050,1060 |
| 2015-11-20 | 1071                               | 1071                          |
| 2015-11-28 | 1030,1070,1125,1141,1150           | 1030,1070,1125,1141,1150      |
| 2015-12-07 | 1151                               | 1151                          |
| 2015-12-09 | 1152                               | 1152                          |
| 2015-12-15 | 1060,1080,1100,1100,1110,1141      | 1060,1080,1100,1110,1141      |
| 2015-12-30 | 1090,1120,1125                     | 1090,1120,1125                |
+------------+------------------------------------+-------------------------------+
```