

## Table of Contents

1. Partitioning the Table Expression with Group By .....	1
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 .....	12
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.

## 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 emp_employees
order by dept_id;
```

```
dept_id
-----
10
20
30
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 emp_employees
group by dept_id
order by dept_id;
```

```
dept_id
-----
10
20
30
35
80
210
215
```

Demo 03: But we cannot show the emp\_id since there is no emp\_id value that represents the department as a group.

```
select dept_id, emp_id
from emp_employees
group by dept_id;
```

```
select dept_id, emp_id
      *
ERROR at line 1:
ORA-00979: not a GROUP BY expression
```

### 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 emp_employees
group by dept_id, job_id
order 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 emp_employees
group by job_id, dept_id
order by dept_id, job_id ;
```

Demo 06: You can sort the result set after you group it.

```
select dept_id, job_id
from emp_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

Note that Oracle does not promise that the ordering of the result set is determined by any grouping. If you want the result set sorted, then use an Order By clause.

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 emp_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 extract( year from order_date) as OrdYear , count(*) as NmbOrders
from oe_orderHeaders
group by extract( year from order_date)
order by extract( year from order_date)
;
```

ORDYEAR	NMBORDERS
2015	57
2016	40

Demo 09: Grouping by year and month. There is no row for 2015 month 5 or 7 since we have no rows in the table for that month.

```
select extract( year from order_date)as OrdYear
, extract( month from order_date)as OrdMonth
, count(*) as NmbOrders
from oe_orderHeaders
group by extract( year from order_date), extract( month from order_date)
order by extract( year from order_date), extract( month from 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

Using a CTE can help make this easier to think about.

```

With orderdates as (
    select extract (year from order_date) as OrdYear
    ,extract (month from order_date) as OrdMonth
    from oe_orderHeaders
)
select OrdYear, OrdMonth, count(*)
from 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 10: 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 emp_employees
group by dept_id;

```

DEPT_ID	EMPCOUNT	AVGSALARY
30	8	76599.875
20	1	15000
210	2	67000
215	4	83563.5
35	3	64333.3333
80	3	36000
10	1	100000

Demo 11: 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 prd_products
group by catg_id
;

```

CATG_I	Avg List Price	Max List Price
PET	123.167273	549.99
HD	23.875	45
MUS	13.8781818	15.95
HW	67.641	149.99
SPG	178.125	349.95
APL	479.986	850
GFD	8.75	12.5

**Basic Rule for Grouping and Aggregates:**

If you use an aggregate function and a non-aggregated attribute in the SELECT clause, then you must GROUP BY the non-aggregated attribute(s).

If you use a GROUP BY clause, then the SELECT clause can include only aggregate functions and the attributes used for grouping. You are not required to show the grouping column in the output.

Demo 12: 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 emp_employees
group by dept_id, job_id
order by avg(salary);
```

DEPT_ID	JOB_ID	NUMEMPLOYEE	AVGSALARY
20	2	1	15000
80	8	2	24500
35	8	1	30000
80	4	1	59000
210	32	1	65000
215	32	1	65000
210	64	1	69000
215	64	2	74627
30	16	4	74863.75
30	32	4	78336
35	16	2	81500
10	1	1	100000
215	16	1	120000

Demo 13: 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 EmployeeCount
, Round(avg(salary),0) as AvgSalary
from emp_employees
group by dept_id
order by avg(salary);
```

DEPARTMENT	EMPLOYEECOUNT	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

Demo 14: Oracle allows a syntax such as this which does the aggregates over the entire table and uses a group by with empty parentheses.

```
select count(*) as NumEmployee
, avg(salary) as AvgSalary
from emp_employees
group by ();
```

NUMEMPLOYEE	AVGSALARY
22	68047.8636

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 prd_products
group by catg_id
;
```

CATG_I	MAXPRICE
PET	549.99
HD	45
MUS	15.95
HW	149.99
SPG	349.95
APL	850
GFD	12.5

Demo 16: We can use this query as a CTE 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.

```
with maxprices as (
  select catg_id, max(prod_list_price) as MaxPrice
  from prd_products
  group by catg_id
)
select p.catg_id, p.prod_id, substr(p.prod_desc,1, 25) as prod_desc
, p.prod_list_price
from prd_products P
join maxprices MP on p.catg_id = MP.catg_id
and p.prod_list_price = MP.maxPrice
;
```

CATG_I	PROD_ID	PROD_DESC	PROD_LIST_PRICE
HW	1090	Gas grill	149.99
SPG	1040	Super Flyer Treadmill	349.95
HW	1160	Stand Mixer with attachme	149.99
PET	4567	Our highest end cat tree-	549.99
PET	4568	Satin four-poster cat bed	549.99
APL	1126	Low Energy Washer Dryer c	850
GFD	5000	Cello bag of mixed finger	12.5
HD	5005	Steel Shingler hammer	45
MUS	2014	Bix Beiderbecke - Tiger R	15.95

### 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. You cannot use a column alias in a Having clause. 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 emp_employees
group by dept_id
order by dept_id;
```

DEPT_ID	EMPCOUNT	AVGSALARY
10	1	100000
20	1	15000
30	8	76599.875
35	3	64333.3333
80	3	36000
210	2	67000
215	4	83563.5

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 emp_employees
group by dept_id
having avg(salary) > 60000
order by dept_id;
```

DEPT_ID	EMPCOUNT	AVGSALARY
10	1	100000
30	8	76599.875
35	3	64333.3333
210	2	67000
215	4	83563.5

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 emp_employees
where dept_id between 30 and 40
group by dept_id
having avg(salary) > 60000
order by dept_id;

```

DEPT_ID	EMPCOUNT	AVGSALARY
30	8	76599.875
35	3	64333.3333

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 emp_employees
where salary > 60000
group by dept_id
order by dept_id;

```

DEPT_ID	EMPCOUNT	AVGSALARY
10	1	100000
30	8	76599.875
35	2	81500
210	2	67000
215	4	83563.5

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

```

select dept_id
, count(*) as EmpCount
, avg(salary) as AvgSalary
from emp_employees
group by dept_id
having count(*) > 3
order by dept_id ;

```

DEPT_ID	EMPCOUNT	AVGSALARY
30	8	76599.875
215	4	83563.5

**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 emp_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 order\_id. How many order lines for each order?

```
select order_id
, count(*) as "NumberLineItems"
from oe_orderDetails
group by order_id
order by order_id;
```

```
selected rows
ORDER_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 customer\_id, and shipping mode, so it would be logical that you could just add them to the Select clause. In Oracle to display these attributes, you either need to include them in the Group By clause, or wrap them in an aggregate function, such as Max, in order to display them. You need to be certain that the extra grouping attributes are not changing the logic of the query.

Demo 24: You need to include extra group levels for this query.

```
select OH.customer_id
, OH.order_id
, OH.shipping_mode_id
, count(*) AS "NumberLineItems"
from oe_orderHeaders OH
join oe_orderDetails OD on OH.order_id = OD.order_id
group by OH.order_id, customer_id, shipping_mode_id
order by OH.customer_id, OH.order_id;
```

```
CUSTOMER_ID ORDER_ID SHIPPING_MODE_ID NumberLineItems
-----
```

400300	378	USPS1	2
401250	106	FEDEX1	1
401250	113	FEDEX2	1
401250	119		1
401250	301	FEDEX2	1
401250	552	FEDEX1	2
401890	112	USPS1	1
401890	519	USPS1	2

. . . rows omitted

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

```
select CS.customer_id, CS.customer_name_last, OH.order_id
```

```
, to_char(order_date, 'yyyy-mm-dd') as OrderDate
, sum( quantity_ordered * quoted_price) as AmntDue
from cust_customers CS
join oe_orderHeaders OH on CS.customer_id = OH.customer_id
join oe_orderDetails OD on OH.order_id = OD.order_id
group by OH.order_id, CS.customer_id, CS.customer_name_last, OH.order_date
order by CS.customer_id, OH.order_id;
```

CUSTOMER_ID	CUSTOMER_NAME_LAST	ORDER_ID	ORDERDATE	AMNTDUE
400300	McGold	378	2015-06-14	4500
401250	Morse	106	2015-10-01	255.95
401250	Morse	113	2015-11-08	22.5
401250	Morse	119	2015-11-28	225
401250	Morse	301	2015-06-04	205
401250	Morse	552	2015-08-12	157.3
401890	Northrep	112	2015-11-08	99.98
401890	Northrep	519	2016-04-04	114.74
. . . rows omitted				

**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.

```
select extract(year from order_date) as OrdYear , count(*)as NmbOrders
from oe_orderHeaders
group by order_date
order by order_date;
```

ORDYEAR	NMBORDERS
2015	1
2015	1
2015	6
2015	2
2015	1
. . .	

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

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

```
select coalesce(customer_name_first || ' ', '') || customer_name_last as
Customer
, count(*) as NbrOrders
from customer_customers
group by customer_name_last, customer_name_first
order by customer_name_first, customer_name_last;
```

CUSTOMER	NBRORDERS
Abigail Adams	1
Al Williams	1
Alexis Hamilton	1

Arnold McGold	3
Barry Mazur	1
. . .	

## 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 oe_orderHeaders
group by shipping_mode_id
order by shipping_mode_id;
```

SHIPPING_MODE_ID	ORDERCOUNT
-----	-----
FEDEX1	26
FEDEX2	2
UPSEXP	5
UPSGR	15
USPS1	38
USPS2	3
	8

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? We can use coalesce. If the column we are displaying is not a string, we would need to cast it to a string.

```
select
  coalesce(shipping_mode_id, 'No shipping mode') as ShippingMode
, count(order_id) as OrderCount
from oe_orderHeaders
group by shipping_mode_id
order by shipping_mode_id;
```

SHIPPINGMODE	ORDERCOUNT
-----	-----
FEDEX1	26
FEDEX2	2
UPSEXP	5
UPSGR	15
USPS1	38
USPS2	3
No shipping mode	8

## 5. Sorting

Demo 30: Suppose we take the previous query and try to sort by the order\_id. This is not allowed.

```
select
  coalesce(shipping_mode_id, 'No shipping mode') as ShippingMode
, count(order_id) as OrderCount
from oe_orderHeaders
group by shipping_mode_id
order by order_id;
```

```
order by order_id
*
ERROR at line 5:
ORA-00979: not a GROUP BY expression
```

Demo 31: This is allowed. The Where clause is simply to reduce the number of rows in the result set. The Order By clause gets the largest ( most recent) order date for each customer. If this does not make sense, add the max(order\_date) expression to the Select and run that.

---

```
select customer_id
from oe_orderHeaders
where customer_id > 900000
group by customer_id
order by max(order_date) desc;
```

## 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 emp_employees
group by dept_id
having count(*) > 1;
```