Table of Contents

So far we have been using Select statements to return result sets that display individual rows from our tables. We have included filters so that we return only some of the rows but the result set were always focused on individual rows from the tables. The Where clauses filtered on one row at a time based on the table expression produced by the From clause.

But think about what companies such as Amazon need to know to do business. They do need to know detail information so that they can charge me for the one book that I purchased, but they also need to know about large groups of data- how many people purchased that book? How many books do we carry that no one has purchased in the last 6 months? What is the average amount due for our orders? What was the total sales for last month?

These type of questions ask for **summary information** about our data. SQL provides **aggregate functions** to answer these questions. The aggregate functions are also called multi-row functions or group functions because they return a single value for each group of multiple rows.

The SQL aggregate functions we cover in this class are Max, Min, Sum, Avg, Median, and Stats_Mod ,and Count. Oracle has additional aggregate functions.

In this unit we will discuss using aggregate functions applied to the entire table expression as filtered by the Where clause. We could find the highest price of all of our products; we could filter for pet supplies and find the highest price for pet supplies product.

1. The Max and Min functions

We will start with some examples where we consider the entire table expression collection of rows as a single group. We use the Max function to get the largest value for a column in the table and the Min function to get the smallest value. These queries are using the tables associated with the AltgeldMart database.

Demo 01: max and min order id from the order headers table

```
select max(order_id) as Max_OrderID
, min(order_id) as Min_OrderID
from oe_orderHeaders
;
```

```
MAX_ORDERID MIN_ORDERID
----------- -----------
       4511         105
```

First notice that we get one row returned. We have almost 100 rows in the table, but we are applying the max and min functions to the entire table. The order_id column is numeric so this is just the biggest number and the smallest number for the order id values.

Demo 02: We can use Max, and Min with numeric data, with dates, and with character data. With character data, max and min use the sorting order for the data.

```
select max(order_date) as Max_OrderDate
, min(order_date) as Min_OrderDate
, max(order_mode) as Max_OrderMode
, min(order_mode) as Min_OrderMode
from oe_orderHeaders;
```

```
MAX_ORDERDATE MIN_ORDERDATE MAX_ORDERMODE            MIN_ORDERMODE
------------- ------------- ------------------------ ----------------
12-MAY-16     05-APR-15     ONLINE                   DIRECT
```

2. The Sum function

The Sum function adds the values in the table column. Sum is used with numeric data.

Demo 03: getting the total of the quantities sold

```
select Sum(quantity_ordered) as SumQuantity
from oe_orderDetails;
```

```
SUMQUANTITY
-----------
        950
```

Demo 04: getting the total amount due for all orders in the table.

```
select Sum(quantity_ordered * quoted_price) as totalAmountDue
from oe_orderDetails;
```

```
TOTALAMOUNTDUE
--------------
      84989.98
```

Demo 05: What happens if you try to use Sum with a date type or a character type column? You get an error message that you cannot use Sum with those data types.

```
select sum(ord_mode)
from oe_order_headers;
```

```
ORA-01722: invalid number
```

```
select sum(ord_date)
from oe_order_headers;
```

```
ORA-00932: inconsistent datatypes: expected NUMBER got DATE
```

3. The Avg function

The average function returns the average(mean) of the values. Avg is used with numeric data.

Demo 06: Examples of the avg function

```
select Avg(quantity_ordered) as AvgQuantity
, Avg(quantity_ordered * quoted_price) as AvgAmountDue
from oe_orderDetails;
```

```
AVGQUANTITY  AVGAMOUNTDUE
-----------  ------------
 5.16304348    461.902065
```

4. What about nulls?

We always need to consider nulls.

Demo 07: This is a small table I created for testing.

```
create table z_tst_aggr(
  id      integer primary key
, col_int integer null
);
```

Demo 08: If I do these aggregates on an empty table, I get nulls. But I do get a row of nulls.

```
select max(col_int) as theMax, sum(col_int) as theSum, avg(col_int) as theAvg
from z_tst_aggr;
```

```
     THEMAX      THESUM      THEAVG
---------- ---------- ----------

1 row selected
```

Now I insert 8 rows. most of these have a null in the second column.

```
        ID    COL_INT
---------- ----------
         1         10
         2         15
         3
         4
         5
         6
         7
         8
```

Run the aggregates again.

```
select max(col_int) as theMax, sum(col_int) as theSum, avg(col_int) as theAvg
from z_tst_aggr;
```

```
     THEMAX      THESUM      THEAVG
---------- ---------- ----------
         15          25        12.5
```

What these functions do is ignore nulls and just work with the non-null data values.

You would have a logical point if you argued that these function should return nulls; in this case we have 8 rows, 6 of the rows have unknown values and the average function simply returns the average of two rows. So you have to think of these functions as returning the Max, Sum, Avg of the values they know.

Some people will argue that the average function should take the sum of the values (25) and divide it by the number of rows (8) and the avg would then be about 3. But this means that you are treating the unknown values as zero- which may not be a good business rule.

You can code the avg function call to specifically treat the nulls as zero.

```
select max(col_int) as theMax, sum(col_int) as theSum
, avg(coalesce(col_int,0)) as theAvg
from z_tst_aggr;
```

```
     THEMAX      THESUM      THEAVG
---------- ---------- ----------
         15          25       3.125
```

Don't get mad at the average function and do not assume all unknown numeric values are zero.

Suppose we are using the avg function to find the average salary for all of our employees and some rows have a null for salary. Does the result returned by avg make business sense? This depends on the situation. Suppose we have 40,000 rows in our table and every row except for two have a value for salary. In that case you might decide that it is safe to ignore the fact that we are missing two rows of data and that the value returned by avg makes sense. SQL cannot help you with that decision; you can use sql to find out what percent of rows are missing data; you can use sql to find the average of the known values. But if those two rows were the rows for the two executives in the company who are hiding their salary values because they are paid so very much more than everyone else, then the value returned by avg does not reflect reality.

On the other hand if we have 40,000 rows in our table and 30,000 of those rows are missing a value for salary, then I don't think I would put much trust that the value returned by avg reflects reality.
Business decision always need to be made by the business experts- not the sql coder. The sql person can help supply info to help the business experts make decisions.

5. Using aggregates and a Where clause and a Join

Demo 09: Using aggregates and a criterion. What is the average list price of the houseware items we carry?

```
select Avg (prod_list_price) as "AvgPrice"
from prd_products
where catg_id = 'HW';
```

```
  AvgPrice
----------
    67.641
```

If we change the filter to CEQ we get a null because we do not have any rows with that category; the filtered table expression in the From and Where clause is empty. But we do get a row- it is just a row with a null.

Demo 10: Using aggregates and a join. For the houseware items we have on an existing order, what is the average list price, the average quoted price and the average extended cost?

```
select
  Avg (prod_list_price) as "AvgListPrice"
, Avg (quoted_price) as "AvgQuotedPrice"
, Avg (quoted_price* quantity_ordered) as "AvgExtendedCost"
, Sum (quoted_price* quantity_ordered) as "TotalExtendedCost"
from prd_products
join oe_orderDetails on  prd_products.prod_id = oe_orderDetails.prod_id
where catg_id = 'HW'
;
```

```
AvgListPrice AvgQuotedPrice AvgExtendedCost TotalExtendedCost
------------ -------------- --------------- -----------------
   54.302037     54.0988889      154.728889           8355.36
```

Note that for each of these queries, we have used aggregate functions in the Select clause- and only aggregate functions in the Select clause and we have one row in the result set. An aggregate function is different than a single row function. An aggregate function takes a group (here the whole table) and produces a **single answer** for that group. We can include a Where clause which filters the table produced by the From clause and that filtered set of rows becomes the group. The Where clause is carried out before the aggregates are calculated in the Select.

Demo 11: You can do some additional work in the Select clause such as applying a function such as round to the aggregated value or combining aggregated values.

```
select  round(Avg(prod_list_price),0) as "Avg Price"
, Max(prod_list_price) - Min(prod_list_price) as "Price Range"
from prd_products;
```

6. Aggregate functions and non aggregated columns

You might want to see the name of the most expensive item we sell- but this type of query cannot tell you that. A function, including an aggregate function, returns a single value for its arguments. We could have two or more items selling at that high price. The query is written to return one row for all of the items grouped together and so it cannot show the product id.

Demo 12: What is the highest list price for any item we sell?

```
select Max(prod_list_price) as "Max Price"
from prd_products
;
```

```
  Max Price
-----------
        850
```

It is possible to display a literal and an aggregate function.

```
select sysdate as RunDate
, MAX(prod_list_price) As "Max Price"
from prd_products;
```

```
RUNDATE    Max Price
--------- ----------
19_MAR-16        850
```

6.1. Find the winner queries

Demo 13: What is the highest price for any item we sell and what is the item- this one does not work and we get an error message.

```
select  prod_id, Max(prod_list_price) as "Max Price"
from prd_products
;
```

```
, prod_id
ORA-00937: not a single-group group function
```

This is an important feature of the way most dbms implement the aggregate functions. We are considering the table as a whole and we can ask for the highest value we have for the list price- that is a characteristic of the table as a whole- but we cannot ask for a prod_id since each row (each product) has its own value for that column and there is no single prod_id that represent the entire table.

The following does not work to find the most expensive product. You are not allowed to use an aggregate function in a Where clause this way. It looks like it should work- after all you can determine the Max (product list price) and that is a single value but the Where clause is for single row filters and aggregates work on groups.

Demo 14: THIS DOES NOT WORK

```
select  prod_id
, prod_name
, prod_list_price
from prd_products
where prod_list_price = MAX(prod_list_price)
```

```
where prod_list_price = MAX(prod_list_price)
                        *
ERROR at line 5:
ORA-00934: group function is not allowed here
```

But we have done some simple subqueries that we can use to handle this.

Demo 15: Using a subquery; the subquery returns the amount of the highest price and the outer query displays all items with that price.
With the current data set there is only one match.

```
select  prod_id
, prod_name
, prod_list_price
from prd_products
where prod_list_price = (
   select MAX(prod_list_price) as "Largest Price"
   from prd_products);
```

```
   prod_id prod_name                                prod_list_price
---------- ------------------------- ------------------------------
      1126 WasherDryer                                       850.00
```

Demo 16: Suppose we want to find the highest priced sporting goods items. This runs but the result is incorrect. The mini dryer is not a sporting goods item

```
select  prod_id
, prod_name
from prd_products
where prod_list_price = (
    select  MAX(prod_list_price) as "Largest Price"
    from prd_products
    where  catg_id = 'SPG');
```

```
   prod_id prod_name
---------- -------------------------
      1040 Treadmill
      4569 Mini Dryer
```

This query gives us two items- but if we check, one of them is actually an appliance item. So we need to filter in the outer query for category as well.

Demo 17: Suppose we want to find the highest priced sporting goods items. Corrected query.

```
variable catg_id varchar2(3);
exec :catg_id := 'SPG';

select prod_id
, prod_name
from prd_products
where catg_id = :catg_id
and prod_list_price = (
    select MAX(prod_list_price)
    from prd_products
    where catg_id = :catg_id);
```

```
   prod_id prod_name
---------- -------------------------
      1040 Treadmill
```

Demo 18: If we change the variable to PET and run the same query, we get more than one item tied for the most expensive in that category.

```
exec :catg_id := 'PET';
/
```

```
    prod_id prod_name
---------- -------------------------
      4567 Deluxe Cat Tree
      4568 Deluxe Cat Bed
```

7. Stats_mode and Median

These are two aggregate functions that Oracle implements.

The median is the middle value of a set of numbers or dates or strings and the stats_mode is the most frequent value. We will start with a small test table. Both of these function ignore nulls, so I do not have any nulls in the test data.

```
Select *
from z_tst_median_mode
order by id;
```

```
        ID    COL_INT COL_STR
---------- ---------- ----------
         1         10 red
         2         12 blue
         3         10 blue
         4         25 red
         5         50 red
         6         25 orange
         7          7 blue
```

The queries will specify which rows to include in the set of values.

Stats_mode. The mode is the value that occurs most frequently

Demo 19: Mode **of the first 5 rows.** We have three col_str values for red, so that is the most frequent. In the col_int column in the first 5 rows, 10 is the most frequent value.

```
With dataSource as (
   select col_str, col_int
   from z_tst_median_mode
   where id between 1 and 5
   )
select Stats_Mode(col_str) as "MODE", Stats_Mode(col_int) as "MODE"
from dataSource;
```

```
MODE            MODE
---------- ----------
red                10
```

Demo 20: Change the cte to use where id between 1 and 7. With 7 rows the values 'red' and 'blue' each occur 3 times and the numbers 10 and 25 each occur 2 times. The rule is that if there is more than one mode, then Oracle picks one of the modes and returns that value. This means that stats_mode is a non deterministic function- you cannt guarantee the result even with the same set of table data. It would be equally valid for Oracle to return red or 25.

```
MODE            MODE
---------- ----------
blue               10
```

Median: this function takes a set of numeric or datetime values. You can think of this as putting the values in sorted order and picking the value in the middle

Demo 21: getting the median of the first 5 numbers. Those values are (10, 10, **12**, 25, 50)

```
With dataSource as (
   select col_int
   from z_tst_median_mode
   where id between 1 and 5
   )
select Median(col_int) as "MEDIAN"
from dataSource;
```

```
    MEDIAN
----------
        12
```

Demo 22: getting the median of the first 6 numbers. Those values are (10, 10, **12, 25**, 25, 50) Now we have 2 values in the middle 12, 25; median returns the average of these two middle values.

```
    MEDIAN
----------
      18.5
```

This is the Oracle technique for finding multiple modes. You should be able to read through this SQL and understand it.

```
select col_str
from (
   select col_str, count(col_str) as cnt1
   from z_tst_median_mode
   group by col_str
   )
where cnt1 = (
   select max(cnt2)
   from (
      select count(col_str) as cnt2
      from z_tst_median_mode
      group by col_str
      )
);
```

```
COL_STR
----------
red
blue
```

Demo 23: Which of our products was ordered the most often? **This does not find ties.** It uses the order details table because that contains the product id of the products that were ordered.

```
select stats_mode (prod_id)
from oe_orderDetails
```

```
STATS_MODE(PROD_ID)
-------------------
               1080
```

Demo 24: Which customers ordered that "most ordered product"?

```
select cust_id, cust_name_last
from cust_customers
where cust_id in (
   select cust_id
   from oe_orderHeaders
   join oe_orderDetails using (ord_id)
   where prod_id = (
      select stats_mode (prod_id)
      from oe_orderDetails
      )
   );
```

```
   CUST_ID CUST_NAME_LAST
---------- ------------------
    401250 Morse
    402100 Morise
    403000 Williams
    404100 Button
    404950 Morris
    409150 Martin
    900300 McGold
    915001 Adams
```

Working through that query; from the innermost subquery out to the main.

This finds the product id for the most frequently ordered item.

```
        select stats_mode (prod_id)
        from oe_orderDetails
```

This find the customer ids for customers who ordered that product

```
     select cust_id
     from oe_order_headers
     join oe_orderDetails using (ord_id)
     where prod_id = (. . . )
```

The main query finds the customers with id in that list.

```
select cust_id, cust_name_last
from cust_customers
where cust_id in ( . . . )
```

8. Mistakes you should not make

1. Some people think that if a query runs and produces results, the query is in some sense ok. This is not true. For example, the function Sum will add up numeric values. So you could run this query and get a result. But the result has no meaning in a business sense. There is no point to adding up customer id numbers.

```
select sum(customer_id)
from cust_customers;
```

```
SUM(CUST_ID)
------------
    15262819
```

You are responsible to writing queries that are meaningful.

2.The Where clause cannot contain an aggregate function. The Where clause filters on single rows; the aggregates work with groups of rows. See the Having clause in the next unit.

4.These functions take an argument that is either a column in the table, or an expression based on a column. So you can write `Select Max(prod_list_price) as "Max Price" from prd_products;` But you cannot write `Select Max(45,90) from dual`; to have SQL find the larger value.

```
Select Max(45,90) from dual
       *
ERROR at line 1:
ORA-00909: invalid number of arguments
```

Oracle does have a Greatest function- which accepts a string of values. It is not the same as the Max function which accepts column expressions.

```
Select greatest(45,90) from dual;
```

```
GREATEST(45,90)
---------------
             90
```