# **ROLLUP and CUBE with GROUP BY Sql clause**

## Setup

## The examples in this document require the following table.

CREATE TABLE myemp (

myempno SMALLINT,

ename VARCHAR(10),

job VARCHAR(9),

mgr SMALLINT,

hiredate DATETIME,

sal DECIMAL(7,2),

comm DECIMAL(7,2),

deptno TINYINT

);

ALTER TABLE myemp ADD CONSTRAINT myuemp\_pk PRIMARY KEY ( myempno );

INSERT INTO myemp VALUES (7369,'SMITH','CLERK',7902,str\_to\_date('17-12-1980','%d-%m-%Y'),800,NULL,20);

INSERT INTO myemp VALUES (7499,'ALLEN','SALESMAN',7698,str\_to\_date('20-2-1981','%d-%m-%Y'),1600,300,30);

INSERT INTO myemp VALUES (7521,'WARD','SALESMAN',7698,str\_to\_date('22-2-1981','%d-%m-%Y'),1250,500,30);

INSERT INTO myemp VALUES (7566,'JONES','MANAGER',7839,str\_to\_date('2-4-1981','%d-%m-%Y'),2975,NULL,20);

INSERT INTO myemp VALUES (7654,'MARTIN','SALESMAN',7698,str\_to\_date('28-9-1981','%d-%m-%Y'),1250,1400,30);

INSERT INTO myemp VALUES (7698,'BLAKE','MANAGER',7839,str\_to\_date('1-5-1981','%d-%m-%Y'),2850,NULL,30);

INSERT INTO myemp VALUES (7782,'CLARK','MANAGER',7839,str\_to\_date('9-6-1981','%d-%m-%Y'),2450,NULL,10);

INSERT INTO myemp VALUES (7788,'SCOTT','ANALYST',7566,date\_add(str\_to\_date('13-7-87','%d-%m-%y'), interval-85 day),3000,NULL,20);

INSERT INTO myemp VALUES (7839,'KING','PRESIDENT',NULL,str\_to\_date('17-11-1981','%d-%m-%Y'),5000,NULL,10);

INSERT INTO myemp VALUES (7844,'TURNER','SALESMAN',7698,str\_to\_date('8-9-1981','%d-%m-%Y'),1500,0,30);

INSERT INTO myemp VALUES (7876,'ADAMS','CLERK',7788,date\_add(str\_to\_date('13-7-87', '%d-%m-%y'), interval-51 day),1100,NULL,20);

INSERT INTO myemp VALUES (7900,'JAMES','CLERK',7698,str\_to\_date('3-12-1981','%d-%m-%Y'),950,NULL,30);

INSERT INTO myemp VALUES (7902,'FORD','ANALYST',7566,str\_to\_date('3-12-1981','%d-%m-%Y'),3000,NULL,20);

INSERT INTO myemp VALUES (7934,'MILLER','CLERK',7782,str\_to\_date('23-1-1982','%d-%m-%Y'),1300,NULL,10);

COMMIT;

**Simple GROUP BY clause for pairwise aggregate function**.

SELECT

deptno,

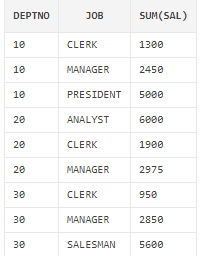
job,

sum(sal)

FROM myemp

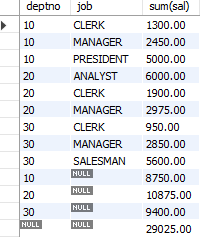
GROUP BY deptno,job

ORDER BY deptno,job;



**ROLLUP with GROUP BY clause:**

The **ROLLUP** clause extends GROUP BY to return a row containing a subtotal for each group of rows, plus a row containing a total for all the groups.



SELECT

deptno,

job,

sum(sal)

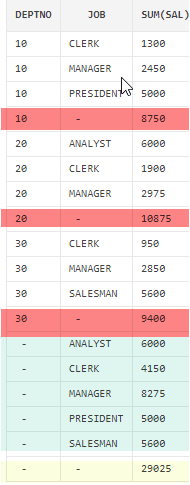
FROM myemp

GROUP BY deptno,job WITH ROLLUP

ORDER BY ISNULL(deptno), ISNULL(job);

**CUBE with GROUP BY clause IN ORACLE:**

The **CUBE** clause extends GROUP BY to return rows containing a subtotal for all combinations of columns, plus a row containing the grand total



SELECT

deptno,

job,

sum(sal)

FROM myemp

GROUP BY CUBE(deptno,job)

ORDER BY deptno,job;

**In MySQL, CUBE can be achieved by doing UNION of combination of ROLLUP queries.**

**SELECT**

**deptno,**

**job,**

**sum(sal)**

**FROM myemp**

**GROUP BY deptno,job WITH ROLLUP**

UNION

**SELECT**

**deptno,**

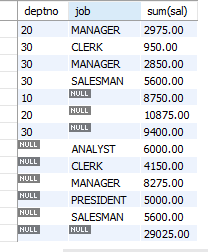
**job,**

**sum(sal)**

**FROM myemp**

**GROUP BY job, deptno WITH ROLLUP**

**ORDER BY ISNULL(deptno),ISNULL(job);**



**RANK**

**The basic description for the RANK analytic function is shown below. The analytic clause is described in more detail;**

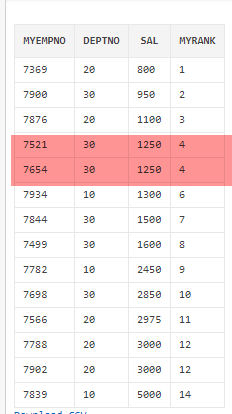
**SELECT myempno,**

**deptno,**

**sal,**

**rank() over (order by sal) as myrank**

**FROM myemp;**



**What we see here is where two employees have the same salary they are assigned the same rank. When multiple rows share the same rank the next rank in the sequence is not consecutive. This is like Olympic medaling in that if two people share the gold, there is no silver medal etc.**

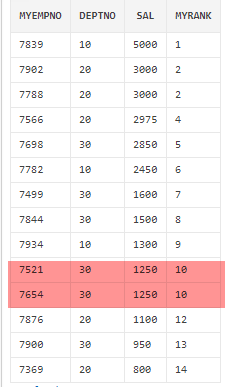
**SELECT myempno,**

**deptno,**

**sal,**

**rank() over (order by sal desc) as myrank**

**FROM myemp;**



* **Bottom two low salaried employees.**

**SELECT \***

**FROM (SELECT myempno,**

**deptno,**

**sal,**

**RANK() OVER ( ORDER BY sal) AS myrank**

**FROM myemp) AS tab**

**WHERE myrank <= 2;**



* **Top two low salaried employees.**

**SELECT \***

**FROM (SELECT myempno,**

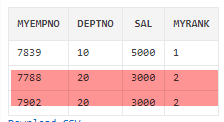
**deptno,**

**sal,**

**RANK() OVER ( ORDER BY sal desc ) AS myrank**

**FROM myemp) tab**

**WHERE myrank <= 2;**



RANK() OVER ([ query\_partition\_clause ] order\_by\_clause)

**Let's assume we want to assign a sequential order, or rank, to people within a department based on salary, we might use the RANK function like this.**

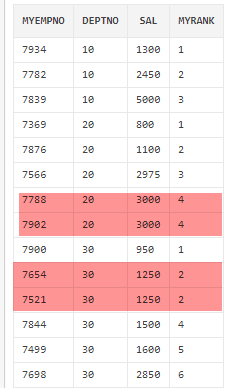
**SELECT myempno,**

**deptno,**

**sal,**

**RANK() OVER (PARTITION BY deptno ORDER BY sal) AS myrank**

**FROM myemp;**



**The fact we can rank the rows in the department means we are able to do a**[**Top-N query**](https://oracle-base.com/articles/misc/top-n-queries)**on a per-department basis. The example below assigns the rank in the inline view, then uses that rank to restrict the rows to the bottom 2 (worst paid) employees in each department.**

**SELECT \***

**FROM (SELECT myempno,**

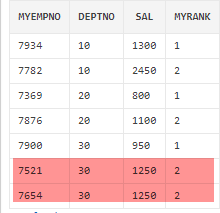
**deptno,**

**sal,**

**RANK() OVER (PARTITION BY deptno ORDER BY sal) AS myrank**

**FROM myemp) AS tab**

**WHERE myrank <= 2;**



**DENSE\_RANK**

DENSE\_RANK() OVER([ query\_partition\_clause ] order\_by\_clause)

**The DENSE\_RANK function acts like the RANK function except that it assigns consecutive ranks, so this is not like Olympic medaling**.

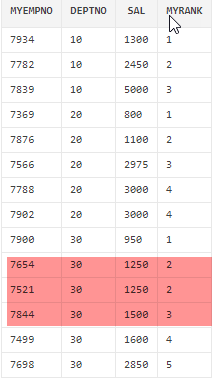
**SELECT myempno,**

**deptno,**

**sal,**

**DENSE\_RANK() OVER (PARTITION BY deptno ORDER BY sal) AS myrank**

**FROM myemp;**



**As with the RANK analytic function, we can do a**[**Top-N query**](https://oracle-base.com/articles/misc/top-n-queries)**on a per-department basis. The example below assigns the dense rank in the inline view, then uses that rank to restrict the rows to the top 2 (best paid) employees in each department.**

**SELECT \***

**FROM (SELECT myempno,**

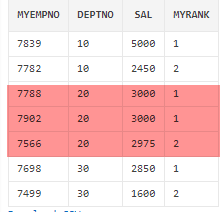
**deptno,**

**sal,**

**DENSE\_RANK() OVER (PARTITION BY deptno ORDER BY sal DESC) AS myrank**

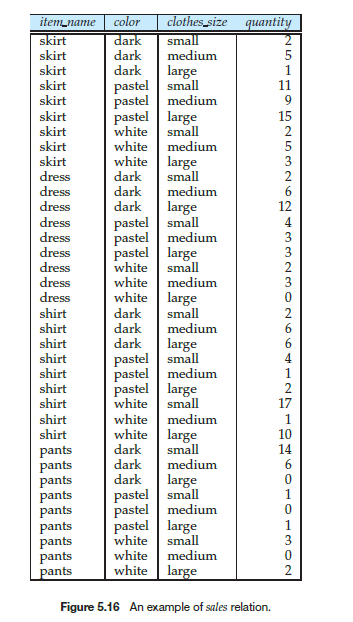
**FROM myemp) AS tab**

**WHERE myrank <= 2;**



**OLAP\*\***

An online analytical processing (OLAP) system is an interactive system that permits an analyst to view different summaries of multidimensional data. CUBE, ROLLUP, Pivot are examples of OLAP.



**Suppose that,**

***Item\_name =>* can take on the values (skirt, dress, shirt, pants),**

***Color =>* can take on the values (dark, pastel, white)**

***Clothes\_size =>* can take on values (small, medium, large), and**

***Quantity =>* is an integer value representing the total number of**

**items of a given *{item name*, *color*, *clothes size }*.**

**Data of Sales relation is shown in Figure 5.16.**

Several RDBMS, such as Microsoft SQL Server, and Oracle, support

a **pivot** clause in SQL, which allows creation of cross-tabs. Given the *sales* relation from Figure 5.16, the query:

**select** \*

**from** *sales*

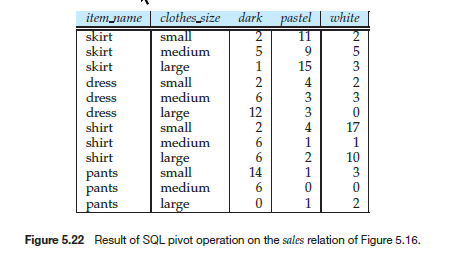
**pivot** ( **sum**(*quantity*) **for** *color* **in** (’dark’,’pastel’,’white’)

)

**order by** *item name*;

Above OLAP query returns the cross-tab shown in Figure 5.22. Note that the **for** clause within the

**Pivot** clause specifies what values from the attribute *color* should appears as attribute names in the pivot result.



# **MySQL Pivot: How to Generate a Pivot Table**

Do you want to learn how to generate pivoted data or pivot tables in MySQL?

You’ll learn:

* what the pivot concept is
* how to generate a pivot table in MySQL
* how to generate the column headings dynamically

## What is PIVOT?

The concept of a “pivot” is where data in rows is transformed to be shown in columns instead. It’s commonly used in Microsoft Excel where you create a Pivot Table to display data differently.

It groups your data based on some values and shows these in columns instead.

Some databases, such as SQL Server, have the built in PIVOT function

However, in MySQL there is no PIVOT feature or keyword. Fortunately, we can still generate this pivot table output.

Let’s see the sample data we’ll use, and then the code to generate the pivot table.

## Sample Data and Expected Output

Let’s see some sample data and our expected output to help demonstrate what PIVOT can do.

We’ve got a product\_sales table which has a product name, store location, and number of sales. Each of these are separate columns:

|  |  |  |
| --- | --- | --- |
| **product\_name** | **store\_location** | **num\_sales** |
| Chair | North | 55 |
| Desk | Central | 120 |
| Couch | Central | 78 |
| Chair | South | 23 |
| Chair | South | 10 |
| Chair | North | 98 |
| Desk | West | 61 |
| Couch | North | 180 |
| Chair | South | 14 |
| Desk | North | 45 |
| Chair | North | 87 |
| Chair | Central | 34 |
| Desk | South | 42 |
| Couch | West | 58 |
| Couch | Central | 27 |
| Chair | South | 91 |
| Chair | West | 82 |
| Chair | North | 37 |
| Desk | North | 68 |
| Couch | Central | 54 |
| Chair | South | 81 |
| Desk | North | 25 |
| Chair | North | 46 |
| Chair | Central | 121 |
| Desk | South | 85 |
| Couch | North | 43 |
| Desk | West | 10 |
| Chair | North | 5 |
| Chair | Central | 16 |
| Desk | South | 9 |
| Couch | West | 22 |
| Couch | Central | 59 |
| Chair | South | 76 |
| Chair | West | 48 |
| Chair | North | 19 |
| Desk | North | 3 |
| Couch | West | 63 |
| Chair | South | 81 |
| Desk | North | 85 |
| Chair | North | 90 |
| Chair | Central | 47 |
| Desk | West | 63 |
| Couch | North | 28 |

**Use the following code to create the table: product\_sales**

CREATE TABLE product\_sales (

product\_name VARCHAR(100),

store\_location VARCHAR(50),

num\_sales INT

);

INSERT INTO product\_sales (product\_name, store\_location, num\_sales) VALUES

('Chair', 'North', 55),

('Desk', 'Central', 120),

('Couch', 'Central', 78),

('Chair', 'South', 23),

('Chair', 'South', 10),

('Chair', 'North', 98),

('Desk', 'West', 61),

('Couch', 'North', 180),

('Chair', 'South', 14),

('Desk', 'North', 45),

('Chair', 'North', 87),

('Chair', 'Central', 34),

('Desk', 'South', 42),

('Couch', 'West', 58),

('Couch', 'Central', 27),

('Chair', 'South', 91),

('Chair', 'West', 82),

('Chair', 'North', 37),

('Desk', 'North', 68),

('Couch', 'Central', 54),

('Chair', 'South', 81),

('Desk', 'North', 25),

('Chair', 'North', 46),

('Chair', 'Central', 121),

('Desk', 'South', 85),

('Couch', 'North', 43),

('Desk', 'West', 10),

('Chair', 'North', 5),

('Chair', 'Central', 16),

('Desk', 'South', 9),

('Couch', 'West', 22),

('Couch', 'Central', 59),

('Chair', 'South', 76),

('Chair', 'West', 48),

('Chair', 'North', 19),

('Desk', 'North', 3),

('Couch', 'West', 63),

('Chair', 'South', 81),

('Desk', 'North', 85),

('Chair', 'North', 90),

('Chair', 'Central', 47),

('Desk', 'West', 63),

('Couch', 'North', 28);

SELECT \* FROM product\_sales;

Let’s say we wanted to see this data with product names on the left and store locations across the top, with the number of sales at each intersection:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **product\_name** | **North** | **Central** | **South** | **West** |
| Chair | 437 | 218 | 376 | 130 |
| Couch | 251 | 218 | 0 | 143 |
| Desk | 226 | 120 | 136 | 134 |

## MySQL Pivot using CASE

You can use a combination of an aggregate function and the CASE statement to show a pivot table.

How can we write a query to do this?

First, we write a SELECT query that gets the product names:

SELECT

product\_name

FROM product\_sales;

Then we add in the SUM function as another column:

SELECT

product\_name,

SUM()

FROM product\_sales;

For this SUM function, we want to show the sum of the num\_sales column, but only where the store\_location is North.

We can do this kind of logic using a CASE statement. And the important thing here is that **a CASE statement can go inside a function**.

So, we put a CASE statement inside the SUM function. We want the CASE statement to say “if the store location is North, then show the number of sales, otherwise show 0”.

SELECT

product\_name,

SUM(CASE

WHEN store\_location = 'North' THEN num\_sales ELSE 0 END

) AS north

FROM product\_sales;

In this example we’ve also put a column alias of “north”, so we get that as a column heading instead of a long CASE statement.

Next, we add a GROUP BY clause because we are showing a column and an aggregate function.

SELECT

product\_name,

SUM(CASE

WHEN store\_location = 'North' THEN num\_sales ELSE 0 END

) AS north

FROM product\_sales

GROUP BY product\_name;

We can run this query, even though there is only one column being calculated, and this is what will be shown:

|  |  |
| --- | --- |
| **product\_name** | **north** |
| Chair | 437 |
| Desk | 226 |
| Couch | 251 |

We can now add on the rest of the columns. Add the same SUM and CASE functions but change the store\_location to the value you want to check.

The query should look like this:

SELECT

product\_name,

SUM(CASE

WHEN store\_location = 'North' THEN num\_sales ELSE 0 END

) AS north,

SUM(CASE

WHEN store\_location = 'Central' THEN num\_sales ELSE 0 END

) AS central,

SUM(CASE

WHEN store\_location = 'South' THEN num\_sales ELSE 0 END

) AS south,

SUM(CASE

WHEN store\_location = 'West' THEN num\_sales ELSE 0 END

) AS west

FROM product\_sales

GROUP BY product\_name;

When you run this query, you should get these results:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **product\_name** | **north** | **central** | **south** | **west** |
| Chair | 437 | 218 | 376 | 130 |
| Desk | 226 | 120 | 136 | 134 |
| Couch | 251 | 218 | 0 | 143 |

We can see the different locations as columns, the different products as rows, and the sum of sales at the intersection of product and location.

That’s how you can do a pivot table in MySQL. You can change the columns and rows to use by changing your query, but we’ve seen the general structure here.

## Dynamic Pivot Columns

In the example above we generated a pivot table using an aggregate function (SUM) and a CASE statement. The downside to that approach is that we need to know the column headings when we write the query, and the more columns that appear the more code we need to write.

This may be OK for smaller results, but what if you don’t know all of the possible values? Or what if the values change?

There is a way to dynamically generate the columns of a PIVOT table output. We use the function called GROUP\_CONCAT.

Here’s what the structure of the code is:

SET @sql = NULL;

SELECT GROUP\_CONCAT(logic)

INTO @sql

FROM your\_table;

SET @sql = CONCAT('select…', @sql, 'from…');

PREPARE stmt FROM @sql;

EXECUTE stmt;

DEALLOCATE PREPARE stmt;

We use the variable of @sql to store the results of GROUP\_CONCAT to avoid characters after the 1024 limit of this function from being cut off.

We then add this value into the rest of the query by concatenating with a SELECT and FROM clause.

We then run it using PREPARE and EXECUTE.

Here’s what our real query would look like on the product\_sales table.

SET @sql = NULL;

SELECT

GROUP\_CONCAT(DISTINCT CONCAT(

'SUM(

CASE WHEN store\_location = "', store\_location, '" THEN num\_sales ELSE 0 END)

AS ', store\_location)

)

INTO @sql

FROM product\_sales;

SET @sql = CONCAT('SELECT product\_name, ', @sql,

' FROM product\_sales GROUP BY product\_name');

SELECT @sql;

PREPARE stmt FROM @sql;

EXECUTE stmt;

DEALLOCATE PREPARE stmt;

We use the DISTINCT keyword inside GROUP\_CONCAT so we get a unique list of store\_locations. Otherwise, we’d get a very long list and our query would show an error.

Now when we run this query, this is what we see:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **product\_name** | **Central** | **North** | **South** | **West** |
| Chair | 218 | 437 | 376 | 130 |
| Desk | 120 | 226 | 136 | 134 |
| Couch | 218 | 251 | 0 | 143 |

With this example, your column headers are generated based on the values in the table.

The @sql has code generated as follow,

**@sql**

**"SELECT product\_name, SUM(**

**CASE WHEN store\_location = "Central" THEN num\_sales ELSE 0 END)**

**AS Central,SUM(**

**CASE WHEN store\_location = "North" THEN num\_sales ELSE 0 END)**

**AS North,SUM(**

**CASE WHEN store\_location = "South" THEN num\_sales ELSE 0 END)**

**AS South,SUM(**

**CASE WHEN store\_location = "West" THEN num\_sales ELSE 0 END)**

**AS West FROM product\_sales GROUP BY product\_name"**

**Conclusion: While there are no MySQL PIVOT keywords available, you can use the SUM function (or other aggregate functions) as well as the CASE statement to generate a pivot table in MySQL. You can also use the GROUP\_CONCAT function and some procedural code to generate the list of columns dynamically.**