**Roll Up:**

The ROLLUP is an extension of the GROUP BY clause. The ROLLUP option allows you to include extra rows that represent the subtotals, which are commonly referred to as super-aggregate rows, along with the grand total row. By using the ROLLUP option, you can use a single query to generate multiple [grouping sets](https://www.sqltutorial.org/sql-grouping-sets/).

**SQL Query:**

create table emp(id int primary key,

name varchar(50),

gender varchar(20),

salary int,

department varchar(50));



**SQL Query:**

select department, sum(salary) from emp Group By rollup(department);



**SQL Query:**

select coalesce(department, 'Total'), sum(salary) from emp Group By rollup(department);

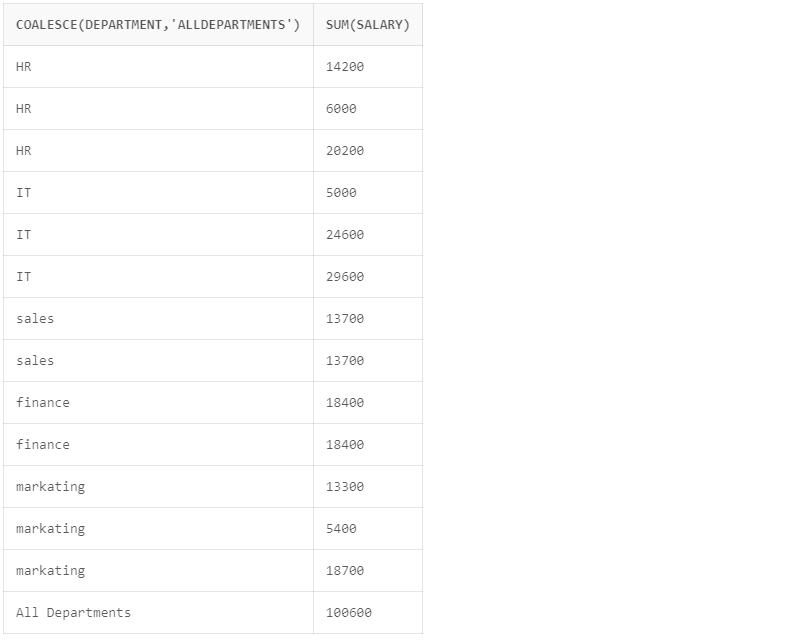
**Output:**



**SQL Query:**

select coalesce(department, 'All Departments'), sum(salary) from emp Group By rollup(department,gender);

**Output:**

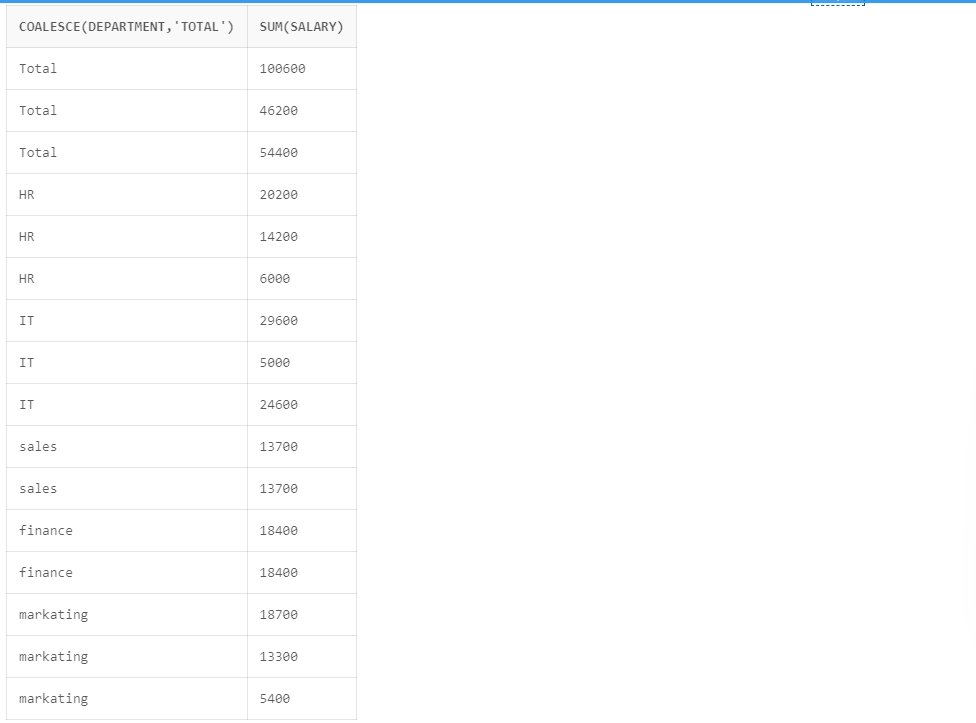


**Cube:**

Similar to the [ROLLUP](https://www.sqltutorial.org/sql-rollup/), CUBE is an extension of the [GROUP BY](https://www.sqltutorial.org/sql-group-by/) clause. CUBE allows you to generate subtotals like the ROLLUP extension. In addition, the CUBE extension will generate subtotals for all combinations of grouping columns specified in the [GROUP BY](https://www.sqltutorial.org/sql-group-by/) clause.

**SQL Query:**

select coalesce(department, 'Total'), sum(salary) from emp Group By cube(department, gender);



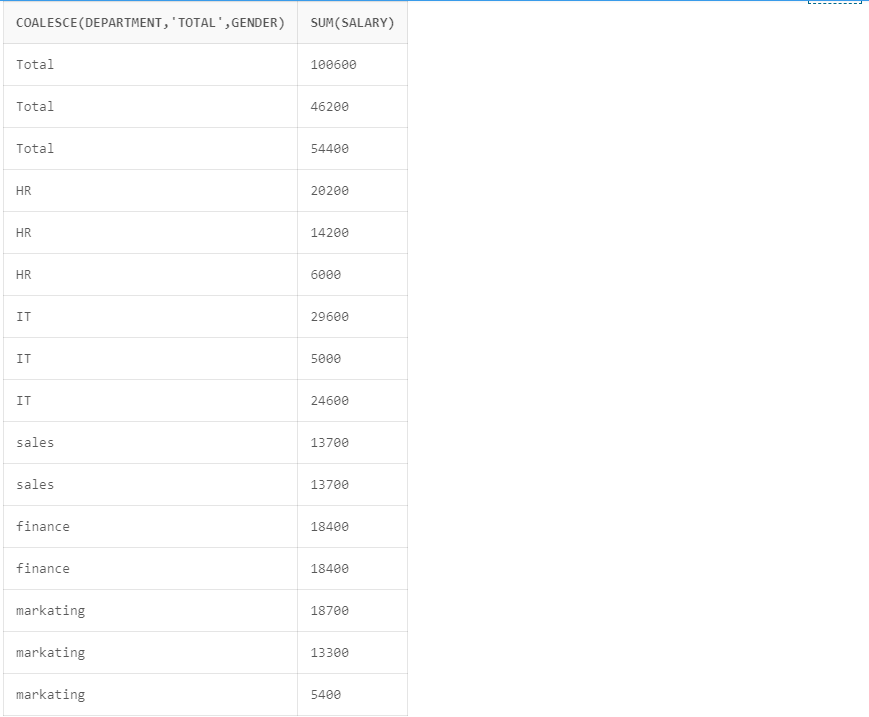
**SQL Query:**

select coalesce(department, 'Sub Total'),coalesce(gender,'Total'), sum(salary) from emp Group By cube(department, gender);



**SQL Query:**

select coalesce(department, 'Total',gender), sum(salary) from emp Group By cube(department, gender);



**Ranking Functions:-**

These functions assign ranks to rows based on some ordering criteria. Functions in this category are RANK, DENSE\_RANK, NTILE, ROW\_NUMBER.

1. **RANK():**-

• RANK is a simple analytical function that does not take any column list arguments. It

returns a number or a rank based on the ordering of the rows; the ordering is based on a

defined condition.

• Similar values are ranked the same

**Syntax**

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

**SQL Query:**

create table Employe(

empid int primary key,

empname varchar(50),

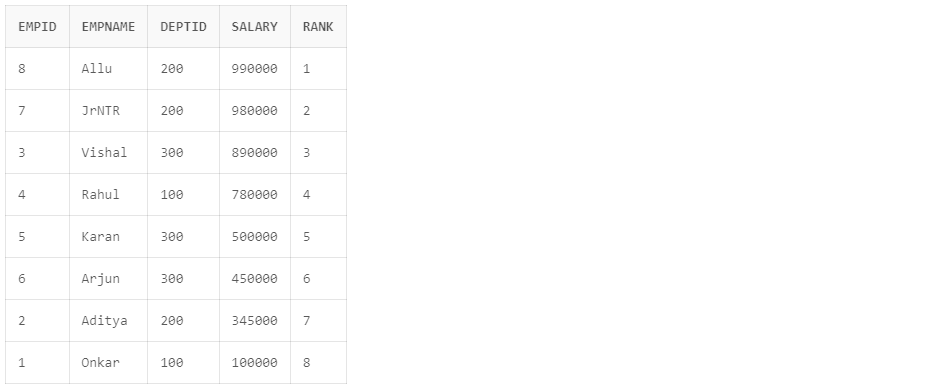
deptid int,

salary int

);



select empid,empname,deptid,salary,rank() over(order by salary desc) as rank from Employe;



**B) DENSE\_RANK():-**

• DENSE RANK works like RANK in that it needs no additional arguments and it ranks items

in descending or ascending order.

• The only difference is that DENSE RANK does not allow “gaps” between groups.

**Syntax**

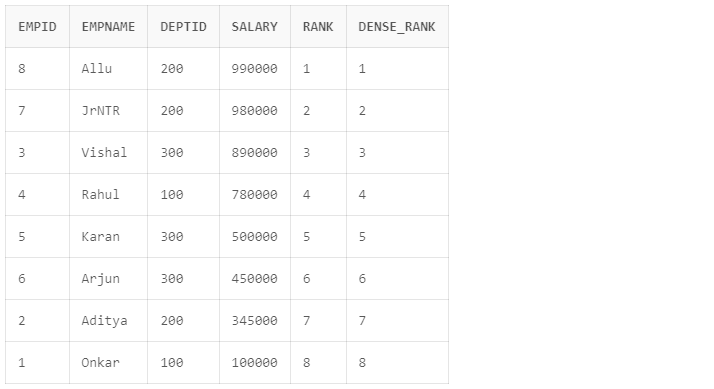
DENSE\_RANK( ) OVER([ query\_partition\_clause ] order\_by\_clause)

**SQL Query:**

select empid,empname,deptid,salary,rank() over(order by salary desc) as rank,

dense\_rank() over(order by salary desc) as dense\_rank

from Employe;



**C) NTILE():-**

• NTILE divides rows into equal groups and returns the number of the group that the row

belongs to.

• This function is not as widely used as either RANK or DENSE RANK.

**Syntax**

NTILE(expr) OVER ([ query\_partition\_clause ] order\_by\_clause)

**SQL Query:**

select empid,empname,deptid,salary,rank() over(order by salary desc) as rank,

ntile(3) over(order by salary desc) as ntile

from Employe;



**D) ROW NUMBER()**

• ROW NUMBER is different than DENSE RANK and RANK in that it does not treat identical

values in any special way.

• It simply lists them as they occur in some order

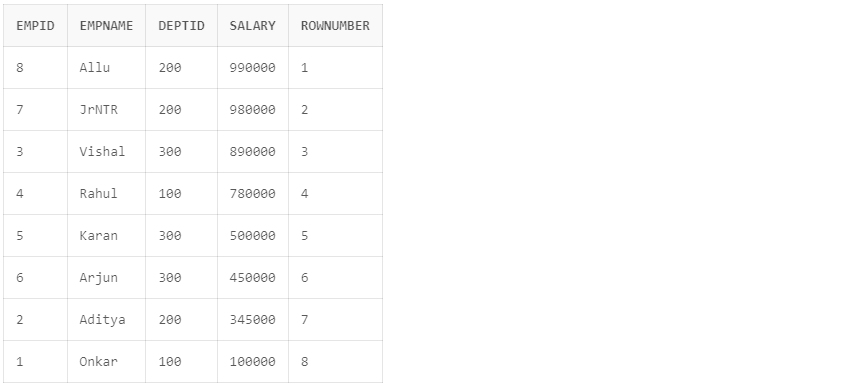
**Syntax**

ROW\_NUMBER( ) OVER ([ query\_partition\_clause ] order\_by\_clause)

**SQL Query:**

select empid,empname,deptid,salary,row\_number() over(order by salary desc) as rownumber

from Employe;



**II. LEAD/LAG Functions**

The LAG and LEAD analytic functions were introduced in 8.1.6 to give access to multiple rows

within a table, without the need for a self-join.

LAG is an analytical function that can be used to get the value of a column in a previous row. If you want to retrieve the value of the next row, use LEAD instead of lag. Because the functions provide access to more than one row of a table at the same time without a self-join, they can enhance processing speed.

**A) LEAD()**

• LEAD returns an offset (incrementally increased) value of an argument column.

• The offset amount can be defined in the code; its default amount is “1”.

• The new value is returned in the same row.

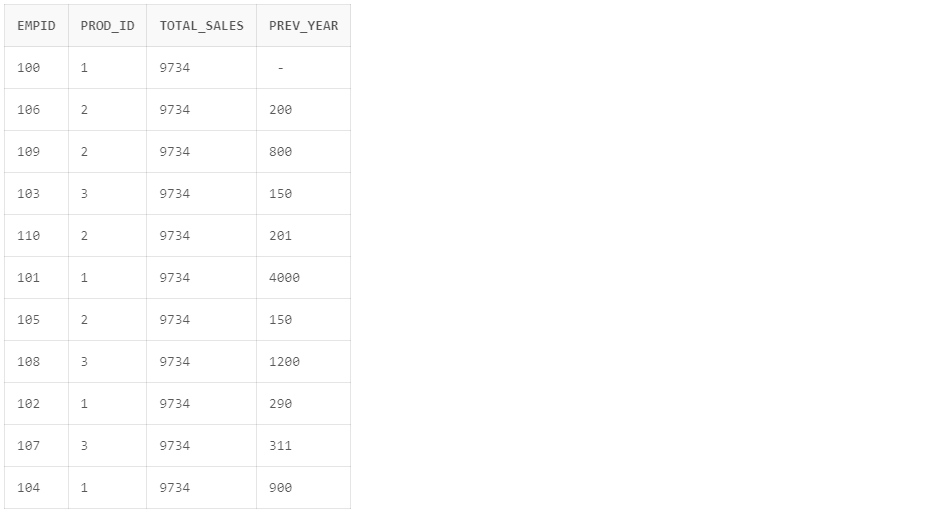
• Syntax

LEAD (value\_expression [,offset] [,default]) OVER ([query\_partition\_clause]

order\_by\_clause)

**SQL Query:-**

select empid, prod\_id, sum(sales) over()as Total\_Sales , Lag(sales) over(order by dept asc) as Prev\_year from employee;



**B) LAG()**

• LAG is the opposite of LEAD. We can even implement LAG using LEAD and vice versa.

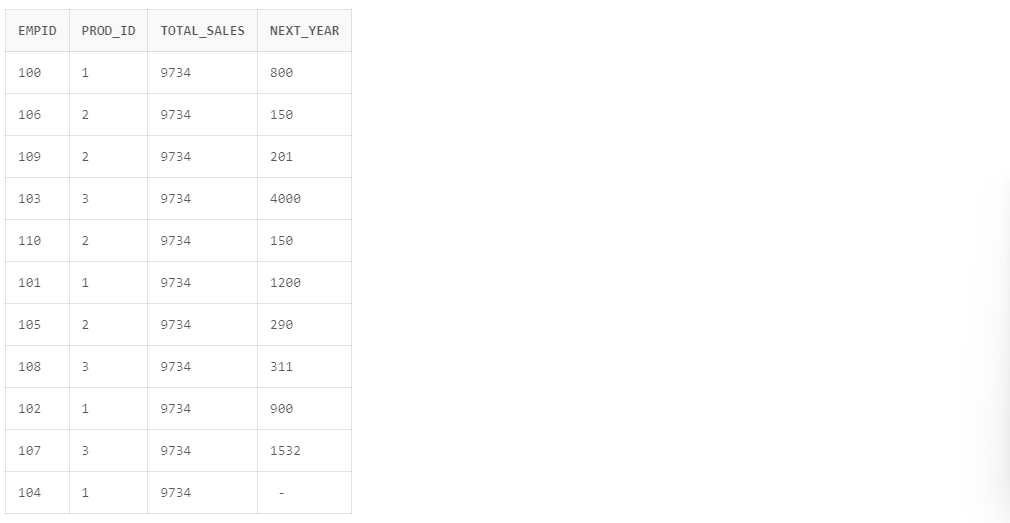
• The difference is in the direction we look for the offset value.

• Syntax

LAG (value\_expression [,offset] [,default]) OVER ([query\_partition\_clause] order\_by\_clause)

**SQL Query:-**

select empid, prod\_id, sum(sales) over()as Total\_Sales , Lead(sales) over(order by dept asc) as Next\_year from employee;



**III. FIRST/LAST Functions**

FIRST and LAST are very similar functions. Both are aggregate and analytic functions that

operate on a set of values from a set of rows that rank as the FIRST or LAST with respect to a

given sorting specification. If only one row ranks as FIRST or LAST, then the aggregate operates

on the set with only one element.

If you omit the OVER clause, then the FIRST and LAST functions are treated as aggregate

functions. You can use these functions as analytic functions by specifying the OVER clause.

1. **FIRST()**

• Return the first value from an ordered sequence.

• Syntax

aggregate\_function KEEP (DENSE\_RANK FIRST ORDER BY expr [ DESC | ASC ] [ NULLS {

FIRST | LAST } ] [, expr [ DESC | ASC ] [ NULLS { FIRST | LAST } ] ]... )

[ OVER ( [query\_partition\_clause] ) ]

**B) LAST()**

• Return the last value from an ordered sequence.

• Syntax

aggregate\_function KEEP (DENSE\_RANK LAST ORDER BY expr [ DESC | ASC ] [ NULLS {

FIRST | LAST } ] [, expr [ DESC | ASC ] [ NULLS { FIRST | LAST } ] ]... )

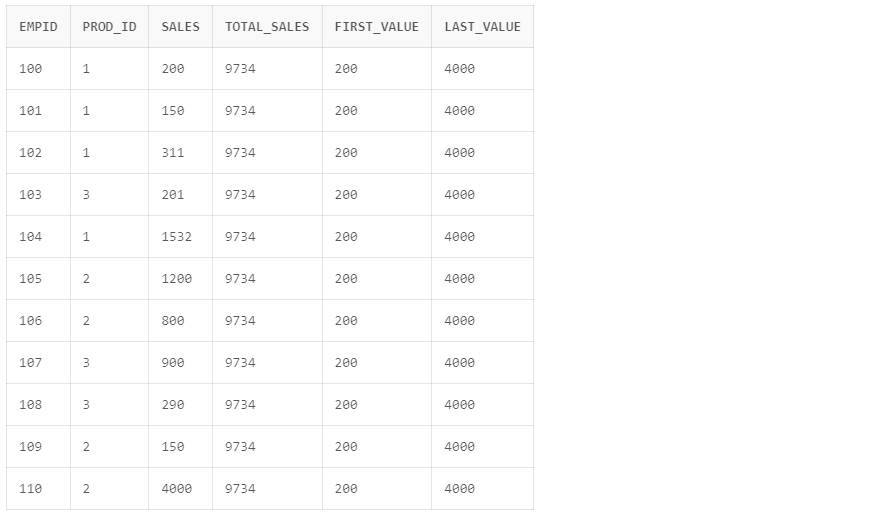
[ OVER ( [query\_partition\_clause] ) ]

**SQL Query:-**

select empid, prod\_id, sales, sum(sales) over()as Total\_Sales ,

first\_value(sales) over() as first\_value ,

last\_value(sales) over() as last\_value from employee;



**IV. Grouping/Aggregate Functions**

**A) AVG()**

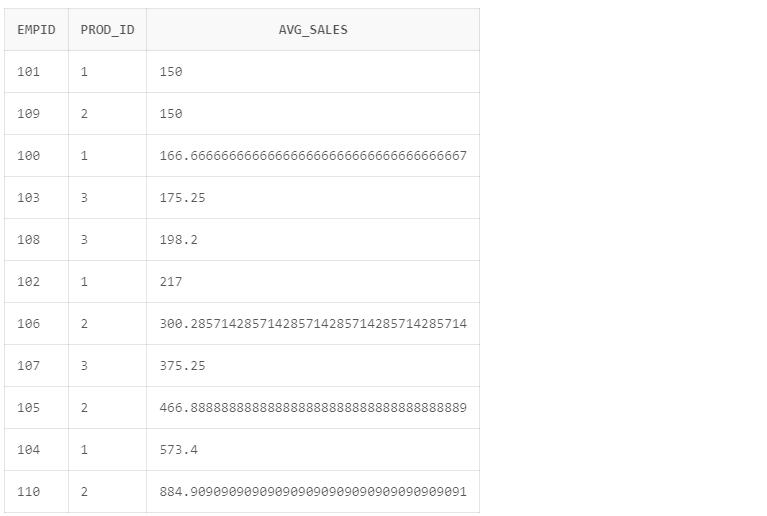
• Returns a running average

• Syntax

AVG([ DISTINCT | ALL ] expr) [ OVER(analytic\_clause) ]

**SQL Query:-**

select empid, prod\_id, avg(sales) over(order by sales asc) as avg\_Sales from employee;



**B) SUM()**

• Computes the cumulative sample running sum

• Syntax

SUM([ DISTINCT | ALL ] expr)

[ OVER (analytic\_clause) ]

**SQL Query:-**

select empid, prod\_id, sum(sales) over(order by sales asc) as Total\_Sales from employee;



**C) COUNT()**

• Returns a running count of all records or by partition

• Syntax

COUNT({ \* | [ DISTINCT | ALL ] expr }) [ OVER (analytic\_clause) ]

**SQL Query:-**

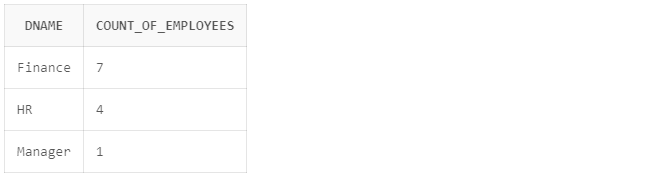
select dname, count(\*) count\_of\_employees

from dept, emp

where dept.deptNo = emp.deptNo

group by DNAME

order by 2 desc;



**D) MIN() & MAX()**

The MIN and MAX aggregate functions are used to calculate the minimum and

maximum values of a set of data respectively.

● The syntax for MIN and MAX aggregate functions are

SELECT MIN(column\_name) FROM table\_name

SELECT MAX(column\_name) FROM table\_name Analytical Functions using query\_partion\_clause

**SQL Query:-**

select department, min(salary) from emp Group By department;



**SQL Query:-**

select department, max(salary) from emp Group By rollup(department);



**V. OLAP Operations**

OLAP provides a user-friendly environment for interactive data analysis. A number of OLAP data cube operations exist to materialize different views of data, allowing interactive querying and analysis of the data.

The most popular end user operations on dimensional data are:

**SQL Query:**

create table loc(loc\_key int NOT NULL primary key,city varchar(50),states varchar(50),country varchar(50));

create table items(items\_key int NOT NULL primary Key,item\_name varchar(50),item\_category varchar(50),color varchar(20));

create table times(time\_Key Date Not NULL primary Key,sdate date, week varchar(50), months Varchar(50),quater Varchar(50), syear int);

create table salesfact(loc\_key int,items\_key int, time\_Key Date,units\_sold int,

FOREIGN KEY(loc\_key) REFERENCES loc(loc\_key),

FOREIGN KEY(items\_key) REFERENCES items(items\_key),

FOREIGN KEY(time\_Key) REFERENCES times(time\_Key)

);

**A) ROLL UP / DRILL UP**

The roll-up operation (also called drill-up or aggregation operation) performs aggregation on a data cube, either by climbing up a concept hierarchy for a dimension or by climbing down a concept hierarchy, i.e. dimension reduction.

**SQL Query:**

select i.item\_name, l.city,t.quater, sum(s.units\_sold) from salesfact s

left join items i on s.items\_key = i.items\_key

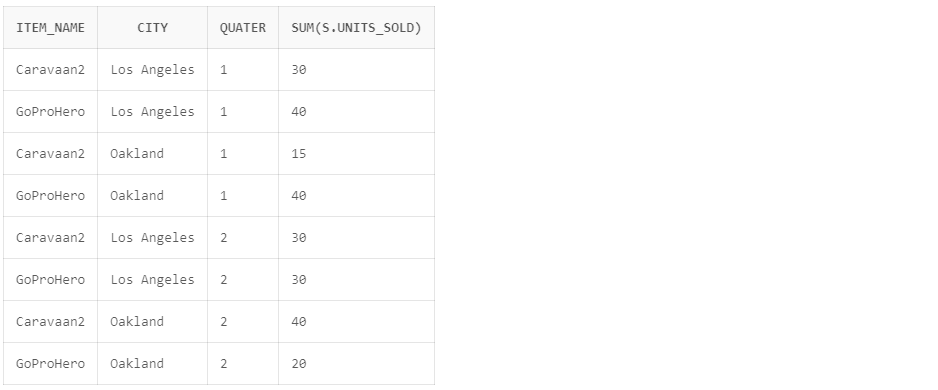
left join loc l on s.loc\_key = l.loc\_key

left join times t on s.time\_Key = t.time\_Key

where(i.item\_name in ('GoProHero','Caravaan2') and l.city in ('Los Angeles','Oakland') and t.quater in (1,2))

group by(i.item\_name, l.city, t.quater)

order by t.quater;



**B) ROLL DOWN**

The roll down operation (also called drill down) is the reverse of roll up. It navigates from less detailed data to more detailed data. It can be realized by either stepping down a concept hierarchy for a dimension or introducing additional dimensions.

**SQL Query:**

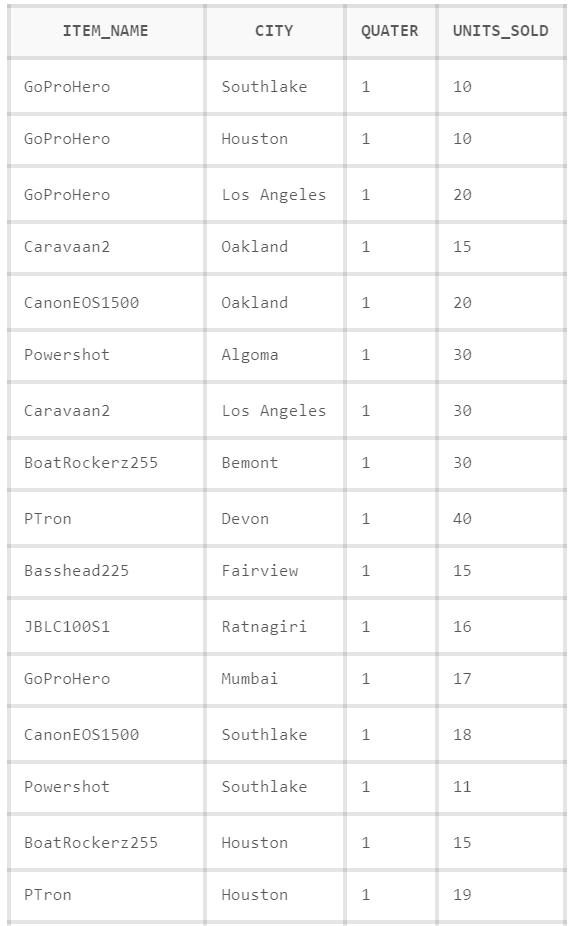
select i.item\_name, l.city,t.quater,s.units\_sold from salesfact s

left join items i on s.items\_key = i.items\_key

left join loc l on s.loc\_key = l.loc\_key

left join times t on s.time\_Key = t.time\_Key

where t.quater = 1;



**C) SLICING**

Slice performs a selection on one dimension of the given cube, thus resulting in a subcube i.e. focus on a particular slice of the cube

WHERE clause in SQL

**D) DICING**

The dice operation defines a subcube by performing a selection on two or more dimensions.

GROUP BY clause in SQL

**SQL Query:**

select i.item\_name, l.country,t.quater,sum(s.units\_sold)

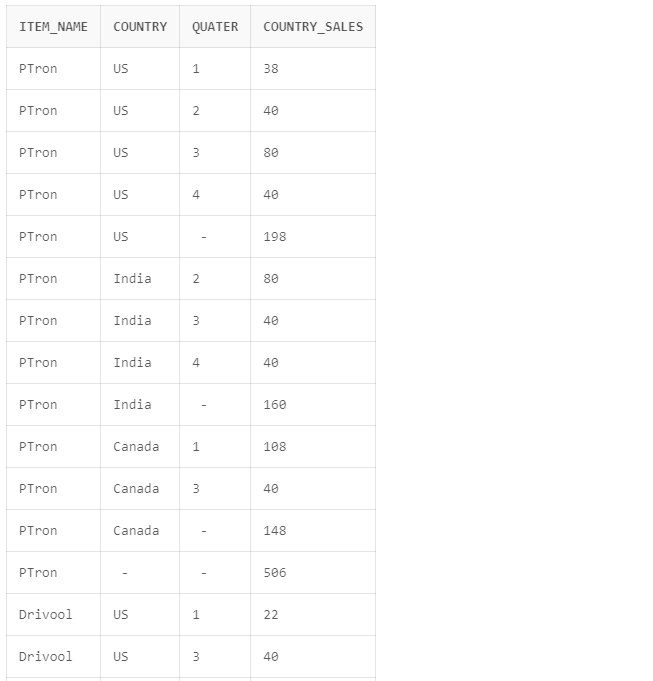
as country\_sales from salesfact s

left join items i on s.items\_key = i.items\_key

left join loc l on s.loc\_key = l.loc\_key

left join times t on s.time\_Key = t.time\_Key

group by rollup( i.item\_name,l.country, t.quater);



**SQL Query:**

select i.item\_name, l.city,sum(s.units\_sold)

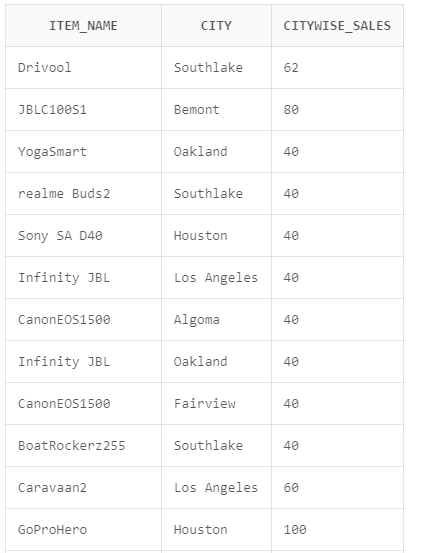
as citywise\_sales from salesfact s

left join items i on s.items\_key = i.items\_key

left join loc l on s.loc\_key = l.loc\_key

left join times t on s.time\_Key = t.time\_Key

group by( i.item\_name,l.city);



**SQL Query:**

select i.item\_name, l.city,t.quater,sum(s.units\_sold)

as country\_sales from salesfact s

left join items i on s.items\_key = i.items\_key

left join loc l on s.loc\_key = l.loc\_key

left join times t on s.time\_Key = t.time\_Key

group by cube( i.item\_name,l.city, t.quater);

