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.

SQL Query:

```
create table emp(id int primary key,
name varchar(50),
gender varchar(20),
salary int,
department varchar(50));
```

```
Table created.
```

SQL Query:

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

DEPARTMENT	SUM(SALARY)
HR	20200
IT	29600
finance	18400
markating	18700
sales	13700
-	100600

SQL Query:

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

Output:

COALESCE(DEPARTMENT,'TOTAL')	SUM(SALARY)
HR	20200
IT	29600
finance	18400
markating	18700
sales	13700
Total	100600

SQL Query:

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

Output:

COALESCE(DEPARTMENT,'ALLDEPARTMENTS')	SUM(SALARY)
HR	14200
HR	6000
HR	20200
IT	5000
IT	24600
IT	29600
sales	13700
sales	13700
finance	18400
finance	18400
markating	13300
markating	5400
markating	18700
All Departments	100600

Cube:

Similar to the ROLLUP, CUBE is an extension of the 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 clause.

SQL Query:

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

COALESCE(DEPARTMENT,'TOTAL')	SUM(SALARY)
Total	100600
Total	46200
Total	54400
HR	20200
HR	14200
HR	6000
IT	29600
IT	5000
IT	24600
sales	13700
sales	13700
finance	18400
finance	18400
markating	18700
markating	13300
markating	5400

SQL Query:

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

COALESCE(DEPARTMENT,'SUBTOTAL')	COALESCE(GENDER,'TOTAL')	SUM(SALARY)
Sub Total	Total	100600
Sub Total	male	46200
Sub Total	female	54400
HR	Total	20200
HR	male	14200
HR	female	6000
IT	Total	29600
IT	male	5000
IT	female	24600
sales	Total	13700
sales	male	13700
finance	Total	18400
finance	female	18400
markating	Total	18700
markating	male	13300
markating	female	5400

SQL Query:

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

COALESCE(DEPARTMENT,'TOTAL',GENDER)	SUM(SALARY)
Total	100600
Total	46200
Total	54400
HR	20200
HR	14200
HR	6000
IT	29600
IT	5000
IT	24600
sales	13700
sales	13700
finance	18400
finance	18400
markating	18700
markating	13300
markating	5400

Ranking Functions:-

These functions assign ranks to rows based on some ordering criteria. Functions in this category are RANK, DENSE_RANK, NTILE, ROW_NUMBER.

A) 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
);
```

```
Table created.
```

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

EMPID	EMPNAME	DEPTID	SALARY	RANK
8	Allu	200	990000	1
7	JrNTR	200	980000	2
3	Vishal	300	890000	3
4	Rahul	100	780000	4
5	Karan	300	500000	5
6	Arjun	300	450000	6
2	Aditya	200	345000	7
1	Onkar	100	100000	8

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;
```

EMPID	EMPNAME	DEPTID	SALARY	RANK	DENSE_RANK
8	Allu	200	990000	1	1
7	JrNTR	200	980000	2	2
3	Vishal	300	890000	3	3
4	Rahul	100	780000	4	4
5	Karan	300	500000	5	5
6	Arjun	300	450000	6	6
2	Aditya	200	345000	7	7
1	Onkar	100	100000	8	8

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;
```

EMPID	EMPNAME	DEPTID	SALARY	RANK	NTILE
8	Allu	200	990000	1	1
7	JrNTR	200	980000	2	1
3	Vishal	300	890000	3	1
4	Rahul	100	780000	4	2
5	Karan	300	500000	5	2
6	Arjun	300	450000	6	2
2	Aditya	200	345000	7	3
1	Onkar	100	100000	8	3

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;
```

EMPID	EMPNAME	DEPTID	SALARY	ROWNUMBER
8	Allu	200	990000	1
7	JrNTR	200	980000	2
3	Vishal	300	890000	3
4	Rahul	100	780000	4
5	Karan	300	500000	5
6	Arjun	300	450000	6
2	Aditya	200	345000	7
1	Onkar	100	100000	8

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;

EMPID	PROD_ID	TOTAL_SALES	PREV_YEAR
100	1	9734	-
106	2	9734	200
109	2	9734	800
103	3	9734	150
110	2	9734	201
101	1	9734	4000
105	2	9734	150
108	3	9734	1200
102	1	9734	290
107	3	9734	311
104	1	9734	900

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;

EMPID	PROD_ID	TOTAL_SALES	NEXT_YEAR
100	1	9734	800
106	2	9734	150
109	2	9734	201
103	3	9734	4000
110	2	9734	150
101	1	9734	1200
105	2	9734	290
108	3	9734	311
102	1	9734	900
107	3	9734	1532
104	1	9734	-

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.

A) 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;
```

EMPID	PROD_ID	SALES	TOTAL_SALES	FIRST_VALUE	LAST_VALUE
100	1	200	9734	200	4000
101	1	150	9734	200	4000
102	1	311	9734	200	4000
103	3	201	9734	200	4000
104	1	1532	9734	200	4000
105	2	1200	9734	200	4000
106	2	800	9734	200	4000
107	3	900	9734	200	4000
108	3	290	9734	200	4000
109	2	150	9734	200	4000
110	2	4000	9734	200	4000

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;

EMPID	PROD_ID	AVG_SALES
101	1	150
109	2	150
100	1	166.66666666666666666666666666666666666667
103	3	175.25
108	3	198.2
102	1	217
106	2	300.28571428571428571428571428571428571428571
107	3	375.25
105	2	466.88888888888888888888888888888888888889
104	1	573.4
110	2	884.90909090909090909090909090909090909091

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;

EMPID	PROD_ID	TOTAL_SALES
101	1	300
109	2	300
100	1	500
103	3	701
108	3	991
102	1	1302
106	2	2102
107	3	3002
105	2	4202
104	1	5734
110	2	9734

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;
```

DNAME	COUNT_OF_EMPLOYEES
Finance	7
HR	4
Manager	1

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;
```

DEPARTMENT	MIN(SALARY)
sales	5700
HR	6000
finance	5500
IT	5000
markating	5400

SQL Query:-

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

DEPARTMENT	MAX(SALARY)
HR	7200
IT	10000
finance	6600
markating	6800
sales	8000
-	10000

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;
```

ITEM_NAME	CITY	QUATER	SUM(S.UNITS_SOLD)
Caravaan2	Los Angeles	1	30
GoProHero	Los Angeles	1	40
Caravaan2	Oakland	1	15
GoProHero	Oakland	1	40
Caravaan2	Los Angeles	2	30
GoProHero	Los Angeles	2	30
Caravaan2	Oakland	2	40
GoProHero	Oakland	2	20

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;
```

ITEM_NAME	CITY	QUATER	UNITS_SOLD
GoProHero	Southlake	1	10
GoProHero	Houston	1	10
GoProHero	Los Angeles	1	20
Caravaan2	Oakland	1	15
CanonEOS1500	Oakland	1	20
Powershot	Algoma	1	30
Caravaan2	Los Angeles	1	30
BoatRockerz255	Bemont	1	30
PTron	Devon	1	40
Basshead225	Fairview	1	15
JBLC100S1	Ratnagiri	1	16
GoProHero	Mumbai	1	17
CanonEOS1500	Southlake	1	18
Powershot	Southlake	1	11
BoatRockerz255	Houston	1	15
PTron	Houston	1	19

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);
```

ITEM_NAME	COUNTRY	QUATER	COUNTRY_SALES
PTron	US	1	38
PTron	US	2	40
PTron	US	3	80
PTron	US	4	40
PTron	US	-	198
PTron	India	2	80
PTron	India	3	40
PTron	India	4	40
PTron	India	-	160
PTron	Canada	1	108
PTron	Canada	3	40
PTron	Canada	-	148
PTron	-	-	506
Drivool	US	1	22
Drivool	US	3	40

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);
```

ITEM_NAME	CITY	CITYWISE_SALES
Drivool	Southlake	62
JBLC100S1	Bemont	80
YogaSmart	Oakland	40
realme Buds2	Southlake	40
Sony SA D40	Houston	40
Infinity JBL	Los Angeles	40
CanonEOS1500	Algoma	40
Infinity JBL	Oakland	40
CanonEOS1500	Fairview	40
BoatRockerz255	Southlake	40
Caravaan2	Los Angeles	60
GoProHero	Houston	100

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);
```

Resize Code Editor

Y	QUATER	COUNTRY_SALES
-	-	6779
-	1	1819
-	2	1840
-	3	1680
-	4	1440
Devon	-	676
Devon	1	156
Devon	2	200
Devon	3	160
Devon	4	160
Algoma	-	650
Algoma	1	170
Algoma	2	160
Algoma	3	160
Algoma	4	160
Bemont	-	648