ADVANCED SQL

1. Stored Procedures

- The stored procedure is SQL statements wrapped within the ***CREATE PROCEDURE*** statement. The stored procedure may contain a conditional statement like IF or CASE or the Loops. The stored procedure can also execute another stored procedure or a function that modularizes the code.

The syntax to create Stored procedure:

```
CREATE PROCEDURE [Procedure Name]
([Parameter 1], [Parameter 2], [Parameter 3] )
BEGIN
SQL Queries..
END
```

In the syntax:

1. The name of the procedure must be specified after the **Create Procedure** keyword
2. After the name of the procedure, the list of parameters must be specified in the parenthesis. The parameter list must be comma-separated
3. The SQL Queries and code must be written between **BEGIN** and **END** keywords

To execute the store procedure, you can use the CALL keyword. Below is syntax:

CALL [Procedure Name] ([Parameters]..)

In the syntax:

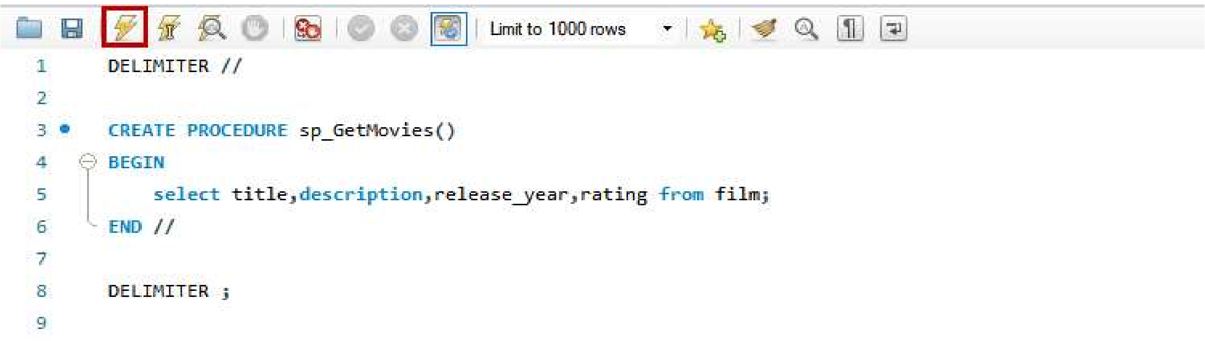
1. The procedure name must be specified after the CALL keyword
2. If the procedure has the parameters, then the parameter values must be specified in the parenthesis

Create a simple stored procedure:

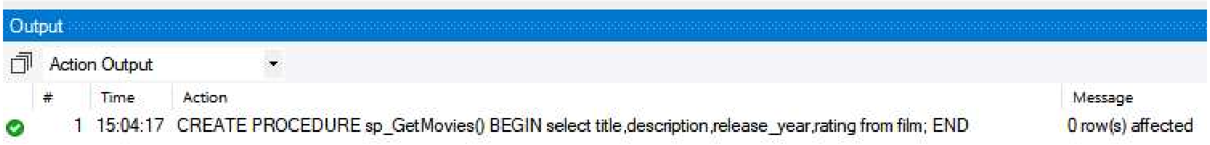
Suppose you want to populate the list of films. The output should contain film_id, title, description, release year, and rating column. The code of the procedure is the following:

```
DELIMITER //
CREATE PROCEDURE sp_GetMovies()
BEGIN
   select title,description,release_year,rating from film;
END //

DELIMITER ;
```

To create the MySQL Stored Procedure, open the **MySQL workbench** Connect to the **MySQL Database** copy-paste the code in the query editor window click on **Execute**.

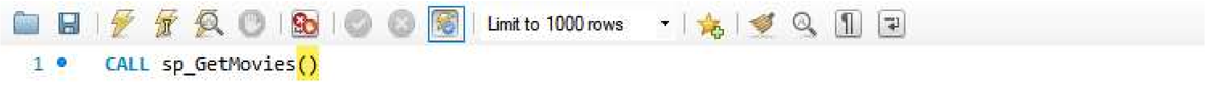


```sql
DELIMITER //

CREATE PROCEDURE sp_GetMovies()
BEGIN
    select title,description,release_year,rating from film;
END //

DELIMITER ;
```

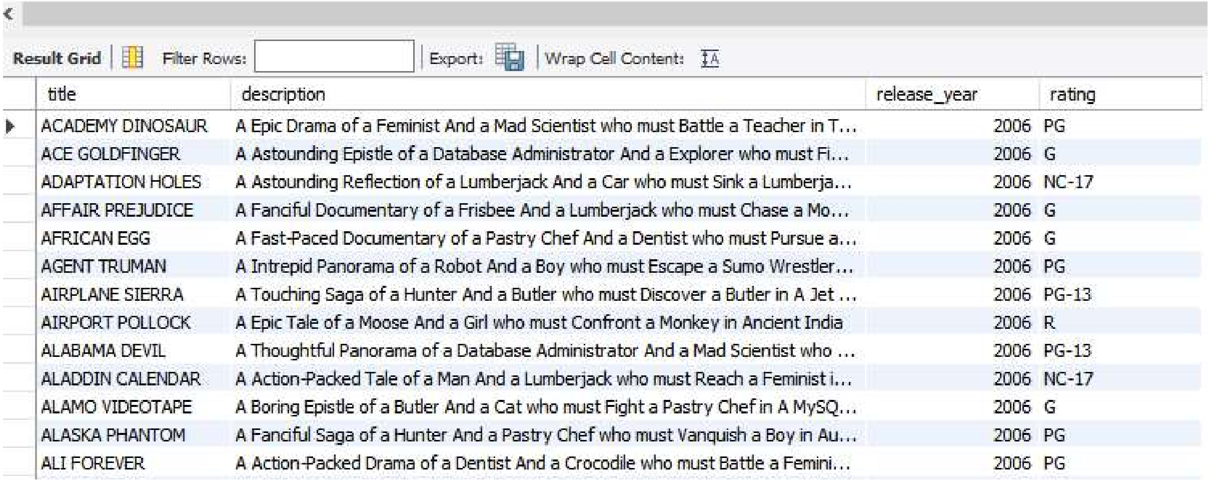


#	Time	Action	Message
1	15:04:17	CREATE PROCEDURE sp_GetMovies() BEGIN select title,description,release_year,rating from film; END	0 row(s) affected

To execute the procedure, run the below command.



```sql
CALL sp_GetMovies()
```



title	description	release_year	rating
ACADEMY DINOSAUR	A Epic Drama of a Feminist And a Mad Scientist who must Battle a Teacher in T…	2006	PG
ACE GOLDFINGER	A Astounding Epistle of a Database Administrator And a Explorer who must Fi…	2006	G
ADAPTATION HOLES	A Astounding Reflection of a Lumberjack And a Car who must Sink a Lumberja…	2006	NC-17
AFFAIR PREJUDICE	A Fanciful Documentary of a Frisbee And a Lumberjack who must Chase a Mo…	2006	G
AFRICAN EGG	A Fast-Paced Documentary of a Pastry Chef And a Dentist who must Pursue a…	2006	G
AGENT TRUMAN	A Intrepid Panorama of a Robot And a Boy who must Escape a Sumo Wrestler…	2006	PG
AIRPLANE SIERRA	A Touching Saga of a Hunter And a Butler who must Discover a Butler in A Jet …	2006	PG-13
AIRPORT POLLOCK	A Epic Tale of a Moose And a Girl who must Confront a Monkey in Ancient India	2006	R
ALABAMA DEVIL	A Thoughtful Panorama of a Database Administrator And a Mad Scientist who …	2006	PG-13
ALADDIN CALENDAR	A Action-Packed Tale of a Man And a Lumberjack who must Reach a Feminist i…	2006	NC-17
ALAMO VIDEOTAPE	A Boring Epistle of a Butler And a Cat who must Fight a Pastry Chef in A MySQ…	2006	G
ALASKA PHANTOM	A Fanciful Saga of a Hunter And a Pastry Chef who must Vanquish a Boy in Au…	2006	PG
ALI FOREVER	A Action-Packed Drama of a Dentist And a Crocodile who must Battle a Femini…	2006	PG

Create a parameterized stored procedure:

- The MySQL Stored procedure parameter has three modes: IN, OUT, and INOUT.
- When we declare an IN type parameter, the application must pass an argument to the stored procedure. It is a default mode.
- The OUT type parameter, the stored procedure returns a final output generated by SQL Statements.
- When we declare the INOUT type parameter, the application has to pass an argument, and based on the input argument; the procedure returns the output to the application.

- When we create a stored procedure, the parameters must be specified within the parenthesis. The syntax is following:

(IN | OUT | INOUT) (Parameter Name [datatype(length)])

In the syntax:

1. Specify the type of the parameter. It can be IN, OUT or INOUT
2. Specify the name and data type of the parameter

Example of IN parameter

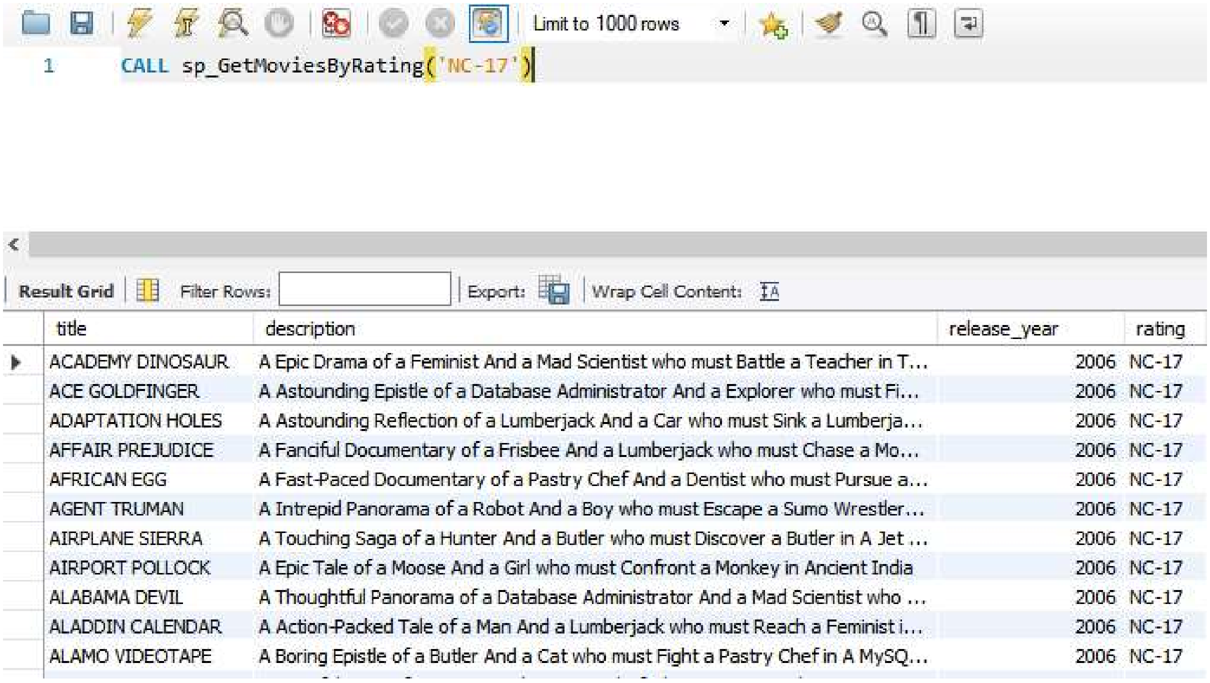
Suppose we want to get the list of films based on the rating. The rating is an input parameter, and the data type is varchar. The code of the procedure is the following:

```
DELIMITER //
CREATE PROCEDURE sp_GetMoviesByRating(IN rating varchar(50))
BEGIN
  select title,description,release_year,rating from film where rating=rating;
END //
DELIMITER ;
```

To populate the list of the films with an **NC-17** rating, we pass the **NC-17** value to the ***sp_getMoviesByRating()*** procedure.

CALL sp_getMoviesByRating('NC-17');

Output:



```
CALL sp_GetMoviesByRating('NC-17')
```

title	description	release_year	rating
ACADEMY DINOSAUR	A Epic Drama of a Feminist And a Mad Scientist who must Battle a Teacher in T...	2006	NC-17
ACE GOLDFINGER	A Astounding Epistle of a Database Administrator And a Explorer who must Fi...	2006	NC-17
ADAPTATION HOLES	A Astounding Reflection of a Lumberjack And a Car who must Sink a Lumberja...	2006	NC-17
AFFAIR PREJUDICE	A Fanciful Documentary of a Frisbee And a Lumberjack who must Chase a Mo...	2006	NC-17
AFRICAN EGG	A Fast-Paced Documentary of a Pastry Chef And a Dentist who must Pursue a...	2006	NC-17
AGENT TRUMAN	A Intrepid Panorama of a Robot And a Boy who must Escape a Sumo Wrestler...	2006	NC-17
AIRPLANE SIERRA	A Touching Saga of a Hunter And a Butler who must Discover a Butler in A Jet ...	2006	NC-17
AIRPORT POLLOCK	A Epic Tale of a Moose And a Girl who must Confront a Monkey in Ancient India	2006	NC-17
ALABAMA DEVIL	A Thoughtful Panorama of a Database Administrator And a Mad Scientist who ...	2006	NC-17
ALADDIN CALENDAR	A Action-Packed Tale of a Man And a Lumberjack who must Reach a Feminist i...	2006	NC-17
ALAMO VIDEOTAPE	A Boring Epistle of a Butler And a Cat who must Fight a Pastry Chef in A MySQ...	2006	NC-17

Example of OUT parameter

Suppose we want to get the count of the films that have a PG-13 rating. The Total_Movies is an output parameter, and the data type is an integer. The count of the movies is assigned to the **OUT** variable (Total_Movies) using the INTO keyword. The code of the procedure is the following:

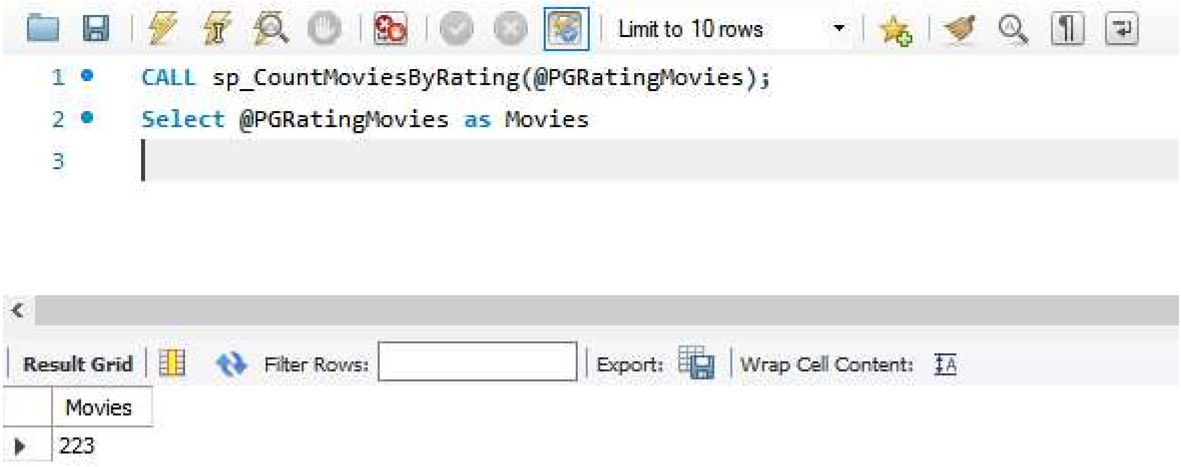
```
DELIMITER //
CREATE PROCEDURE sp_CountMoviesByRating(OUT Total_Movies int)
BEGIN
   select count(title) INTO Total_Movies from film where rating='PG-13';
END //
DELIMITER ;
```

To store the value returned by the procedure, pass a session variable named **@PGRatingMovies**.

CALL sp_CountMoviesByRating(@PGRatingMovies);

Select @PGRatingMovies as Movies;

Output:



```
CALL sp_CountMoviesByRating(@PGRatingMovies);
Select @PGRatingMovies as Movies
```

```
Movies
223
```

Example of an INOUT parameter

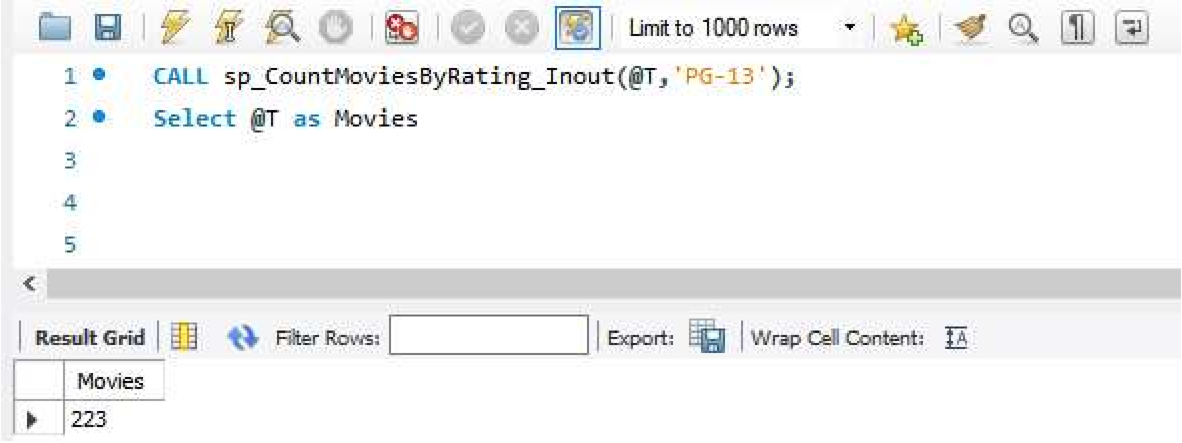
Suppose we want to get the total count of movies based on the rating. The input parameter is param_rating in the procedure, and the data type is **varchar(10)**. The output parameter is **Movies_count,** and the data type is an **integer**.

```
DELIMITER //
CREATE PROCEDURE sp_CountMoviesByRating_Inout(inout Movies_count int, In param_rating varchar(10))
BEGIN
   select count(title) INTO Movies_count from film where rating=param_rating ;
END //
DELIMITER ;
```

Execute the procedure using **CALL** keyword and save the output in session variable named **@MoviesCount**

CALL sp_CountMoviesByRating_Inout(@T,'PG-13');
Select @T as Movies

Output:



```
CALL sp_CountMoviesByRating_Inout(@T,'PG-13');
Select @T as Movies
```

```
Movies
223
```

Drop a Stored Procedure

To drop the stored procedure, you can use the drop procedure command. The syntax is following

Drop procedure [IF EXISTS] <Procedure Name>

In the syntax, the name of the stored procedure must be followed by the **Drop Procedure** keyword. If you want to drop the **sp_getCustomer** procedure from the sakila database, you can run the following query.

Drop Procedure sp_getCustomer;

- When you try to drop the procedure that does not exist on a database, the query shows an error:

ERROR 1305 (42000): PROCEDURE sakila.getCustomer does not exist

- To avoid this, you can include the [IF EXISTS] option in the drop procedure command. When you include the IF EXISTS keyword, instead of an error, the query returns a warning:

Query OK, 0 rows affected, 1 warning (0.01 sec) 1305 PROCEDURE sakila.getCustomer does not exist

2. Trigger in SQL

- A **SQL trigger** is a database object which fires when an event occurs in a database. We can execute a SQL query that will "do something" in a database when a change occurs on a database table such as a record is inserted or updated or deleted.

Types of Triggers

There are two types of triggers:

1. DDL Trigger
2. DML Trigger

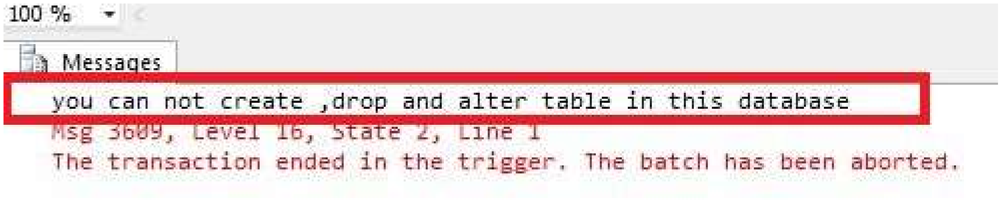
❖ DDL Triggers

The DDL triggers are fired in response to DDL (Data Definition Language) command events that start with **Create, Alter and Drop, such as Create_table, Create_view, drop_table, Drop_view and Alter_table.**

Code of a DDL Trigger

```
create trigger saftey
on database
for
create_table,alter_table,drop_table
as
print'you can not create ,drop and alter table in this database'
rollback;
```

When we create, alter or drop any table in a database then the following message appears:



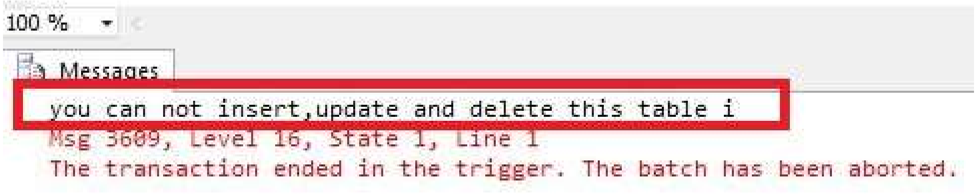
```
you can not create ,drop and alter table in this database
Msg 3609, Level 16, State 2, Line 1
The transaction ended in the trigger. The batch has been aborted.
```

❖ DML Triggers

The DML triggers are fired in response to DML (Data Manipulation Language) command events that start with **Insert, Update, and Delete. Like insert_table, Update_view and Delete_table.**

```
create trigger deep
on emp
for
insert,update,delete
as
print'you can not insert,update and delete this table i'
rollback;
```

When we insert, update or delete in a table in a database then the following message appears,



```
you can not insert,update and delete this table i
Msg 3609, Level 16, State 1, Line 1
The transaction ended in the trigger. The batch has been aborted.
```

Trigger for Insert

Table-1

```
create table students(
name varchar(40),
id int,
std int,
address varchar(50),
fees int,
primary key (id));

select * from students;
```

Table-2

```
create table audit_student(
id int,
descr varchar(50),
primary key (id));

select * from audit_student;
```

Syntax for after insert trigger command:

```
delimiter //
create trigger student_audit_update
after insert
on students
for each row
begin
insert into audit_student values(
new.id,now());
end //
delimiter ;
```

```
delimiter //
CREATE TRIGGER  <trigger name>
[before|after] [ insert| update| delete]
on <table name>
for each row
begin
<trigger body>
end //
delimiter ;
```

```
insert into students(name,id,std,address,fees)
values('bhargav',101,12,'bharuch',1700),
('nirav',102,12,'bharuch',1700),
('abhishek',103,12,'bharuch',1700);

drop trigger student_update;
```

Syntax for before insert trigger command:

```
delimiter //
create trigger student_update
before insert
on students
for each row
begin
set new.fees=new.fees+100;
end //
delimiter ;
```

```
insert into students
values('ruchi',104,12,'bharuch',1700),
('radhu',105,12,'bharuch',1700),
('mansi',106,12,'bharuch',1700);

select * from students;
select * from audit_student;
```

Trigger for update

Table-1

```
create table flight(
name varchar(50),
ticket_id int,
address varchar (50),
price int,
primary key(ticket_id));

select * from flight;
```

Table-2

```
create table flight_passenger_detail(
name varchar(50),
ticket_id int,
boarding varchar (50),
primary key(ticket_id));

select * from flight_passenger_detail
```

Syntax for before update trigger command:

```
delimiter //
create trigger update_price
before update
on flight
for each row
begin
if new.price<4000 then
set new.price=5000;
end if;
end //
delimiter ;
```

```
insert into flight values('abhishek',11214,'chennai',4500);
insert into flight values('ruchi',11215,'pune',8000);
insert into flight values('radhu',11216,'vadodara',2500);
insert into flight values('mansi',11217,'surat',3600);
```

```
update flight set price=2800 where ticket_id=11216;
```

Syntax for after update trigger command:

```
delimiter //

create trigger passenger_detail

after update

on flight

for each row

begin

insert into flight_passenger_detail values(

new.name, new.ticket_id, concat('passenger boarding at ',date_format(now(), '%d %m %y %h:%i:%m  %p')));

end //

delimiter ;
```

```
insert into flight values('roy',21413,'chennai',1500);

insert into flight values('jennil',21414,'pune',8000);

insert into flight values('parul',21415,'vadodara',3900);


update flight set price=3800 where ticket_id=21415;

update flight set price=3800 where ticket_id=21413;

update flight set price=7000 where ticket_id=21414;

select * from flight_passenger_detail;

select * from flight;

drop trigger passenger_detail
```

Trigger for Delete

Syntax for before delete trigger command:

```
delimiter //
create trigger before_delete
before delete
on flight
for each row
begin
signal sqlstate '45000' set message_text="NOT ALLOWED";
end //
delimiter ;
```

```
delete from flight
where ticket_id=11214;
```

```
#backup table
create table backup (name varchar(50), ticket_id int primary key, comment varchar(100));
```

Syntax for after delete trigger command:

```
delimiter //
create trigger after_delete
after delete
on flight
for each row
begin
insert into backup
values(old.name, old.ticket_id, concat("user deleted at ",now()));
end //
delimiter ;
```

```
delete from flight
where ticket_id=11217;
select * from backup;

drop trigger after_delete;
drop table backup;
```

3. View in SQL

- **A view** is a virtual table based on the result-set of an SQL statement.
- A view contains rows and columns, just like a real table. The fields in a view are fields from one or more real tables in the database.
- You can add SQL statements and functions to a view and present the data as if the data were coming from one single table.

Syntax for view command:

Example:1

```
create view flight_passanger_sort
as
select name,ticket_id,price
from flight
where price>4700;
```

```
CREATE VIEW <view_name>

AS
SELECT <column_name>
FROM <table_name>
WHERE condition;
```

Query the view command by: **Select * from [view_name];**

Note: A view always shows up-to-date data! The database engine recreates the view, every time a user queries it.

Example:2

```
create view passanger_ticket_price
as
select name,ticket_id,price
from flight
where price>
(select avg(price) from flight);
```

Example:3

```
create view passanger_detail
as
select fp.ticket_id,f.address,f.price,fp.boarding
from flight f left join
flight_passenger_detail fp
on f.ticket_id=fp.ticket_id;
```

#Rename view command name:

rename table passanger_detail <view_name>

to flight_detail <new_view_name>;

# Display View

```
show full tables

where table_type='view';
```

#updating a view

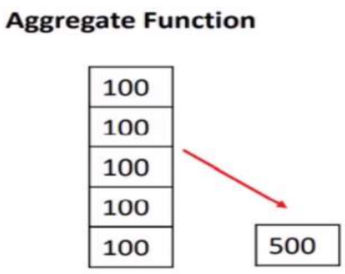
```
create or replace view
passanger_detail

as

select
fp.ticket_id,f.price,fp.boarding

from flight f left join

flight_passenger_detail fp

on f.ticket_id=fp.ticket_id;
```

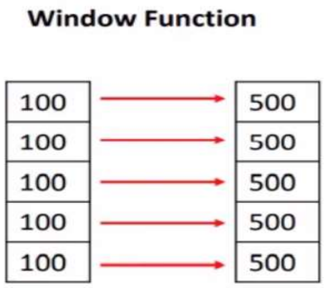
#Delete a view

```
Drop view <view_name>;
```

4. Window function in SQL

- We perform calculations on data using various aggregated functions such as Max, Min, and AVG. We get a single output row using these functions.
- Window functions perform an aggregate operation for each row and returns result in details.





Window function syntax:

```
Window_function_name(expression) OVER(
[PARTITION BY partition_list]
[ORDER BY order_list]
)
```

Note: When you use a window function in a query, define the window using the OVER() clause.

#List of window functions

OVER(): This window function is the replacement of GROUP BY. It creates a window with multiple rows. It is used to determine which rows from the query are applied to the function.

PARTITION(): It is used to divide the result set from the query into data sunsets.

#Ranking window functions

Rank() and Dense_Rank(): Rank() returns a unique rank number for each distinct row within the partition according to a specified column value. Rank() function always work on Over() function with order BY.

Dense_Rank() function is similar to Rank() function except for one difference, it doesn't skip any rank when ranking rows.

Row_Number(): This function is use to get unique sequential number for each row in the specified data.

Ntile(N): This function used to distribute the number of rows in the specified (N) number of groups.

#value window functions

LAG() and LEAD(): The LAG function has the ability to fetch data from a previous row,

While LEAD fetches data from a subsequent or next row.

First_Value() and Last_Value(): Both functions are straight forward. They either return the first or the last value of an ordered set.

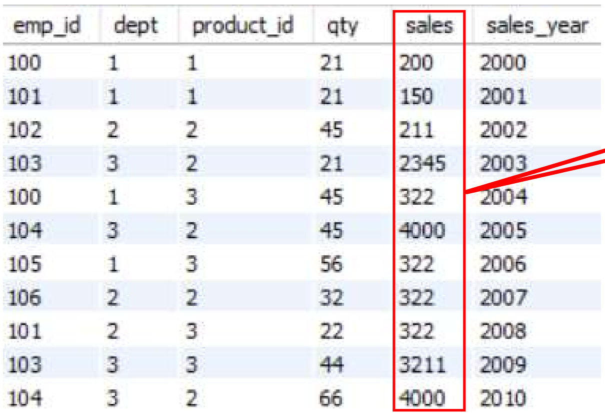
OVER()

Using employee_sales table from abhishek_db database,

```
select emp_id,product_id,

sum(sales) over() as 'total sale',

avg(sales) over () as 'avg sale'

from employee_Sales;
```

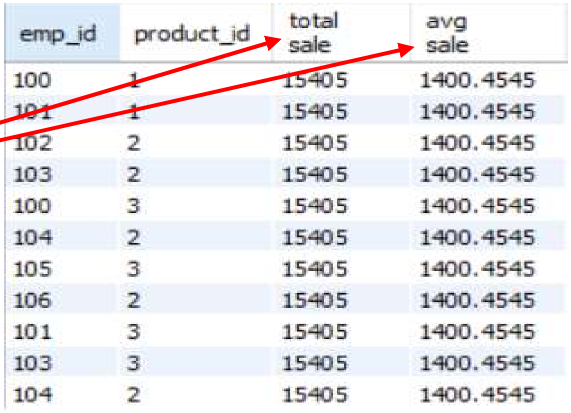
OVER() syntax

Employee_sales database:



emp_id	dept	product_id	qty	sales	sales_year
100	1	1	21	200	2000
101	1	1	21	150	2001
102	2	2	45	211	2002
103	3	2	21	2345	2003
100	1	3	45	322	2004
104	3	2	45	4000	2005
105	1	3	56	322	2006
106	2	2	32	322	2007
101	2	3	22	322	2008
103	3	3	44	3211	2009
104	3	2	66	4000	2010

over() output:

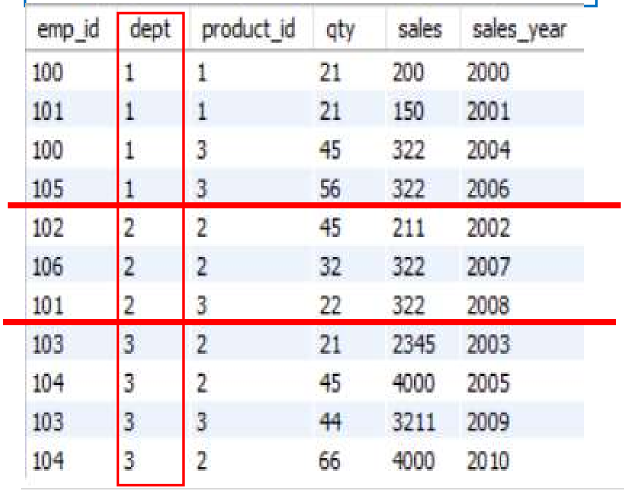


emp_id	product_id	total sale	avg sale
100	1	15405	1400.4545
101	1	15405	1400.4545
102	2	15405	1400.4545
103	2	15405	1400.4545
100	3	15405	1400.4545
104	2	15405	1400.4545
105	3	15405	1400.4545
106	2	15405	1400.4545
101	3	15405	1400.4545
103	3	15405	1400.4545
104	2	15405	1400.4545

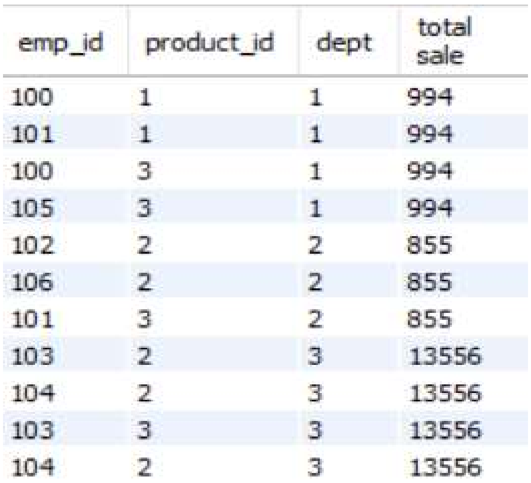
PARTITION BY()

```
select emp_id,product_id,dept,

sum(sales) over(partition by dept)

as 'total sale'

from employee_sales;
```

PARTITION BY() syntax



emp_id	dept	product_id	qty	sales	sales_year
100	1	1	21	200	2000
101	1	1	21	150	2001
100	1	3	45	322	2004
105	1	3	56	322	2006
102	2	2	45	211	2002
106	2	2	32	322	2007
101	2	3	22	322	2008
103	3	2	21	2345	2003
104	3	2	45	4000	2005
103	3	3	44	3211	2009
104	3	2	66	4000	2010



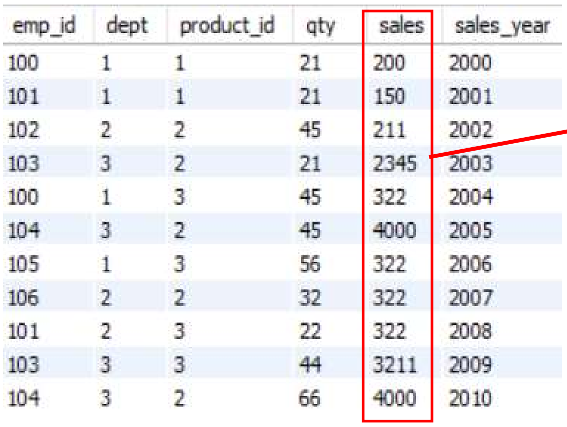
emp_id	product_id	dept	total sale
100	1	1	994
101	1	1	994
100	3	1	994
105	3	1	994
102	2	2	855
106	2	2	855
101	3	2	855
103	2	3	13556
104	2	3	13556
103	3	3	13556
104	2	3	13556

RANK()

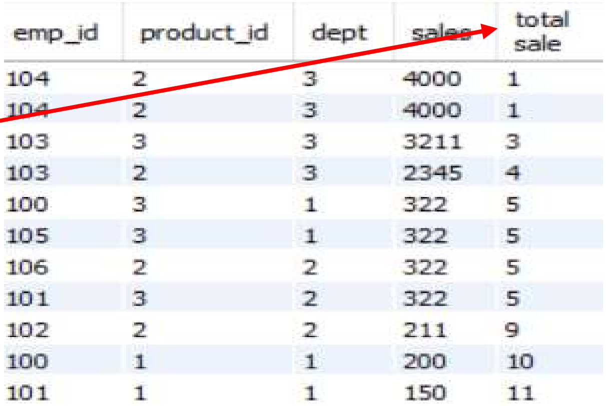
RANK() syntax

Without PARTITION BY

```
select emp_id,product_id,dept,sales,
rank() over(order by sales desc)
as 'total sale'
from employee_sales;
```



emp_id	dept	product_id	qty	sales	sales_year
100	1	1	21	200	2000
101	1	1	21	150	2001
102	2	2	45	211	2002
103	3	2	21	2345	2003
100	1	3	45	322	2004
104	3	2	45	4000	2005
105	1	3	56	322	2006
106	2	2	32	322	2007
101	2	3	22	322	2008
103	3	3	44	3211	2009
104	3	2	66	4000	2010

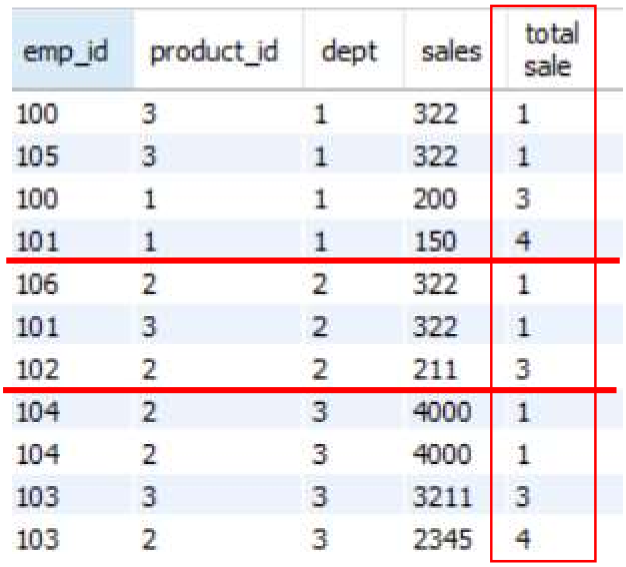


emp_id	product_id	dept	sales	total sale
104	2	3	4000	1
104	2	3	4000	1
103	3	3	3211	3
103	2	3	2345	4
100	3	1	322	5
105	3	1	322	5
106	2	2	322	5
101	3	2	322	5
102	2	2	211	9
100	1	1	200	10
101	1	1	150	11

```
select emp_id,product_id,dept,sales,
rank() over(partition by dept order by
sales desc)
as 'total sale'
from employee_sales;
```

RANK() syntax

With PARTITION BY

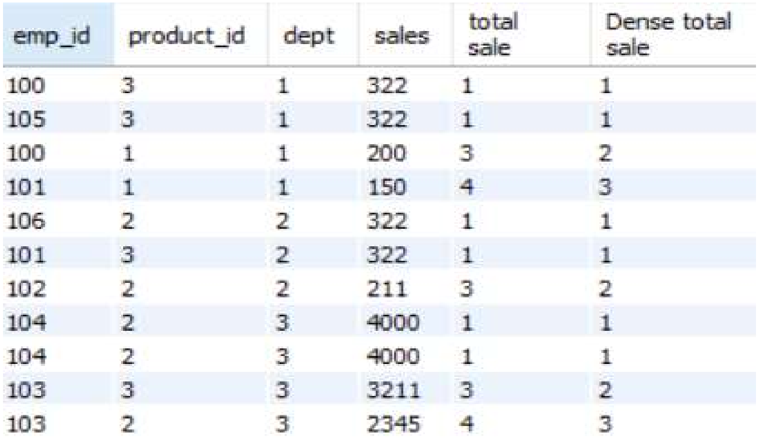


emp_id	product_id	dept	sales	total sale
100	3	1	322	1
105	3	1	322	1
100	1	1	200	3
101	1	1	150	4
106	2	2	322	1
101	3	2	322	1
102	2	2	211	3
104	2	3	4000	1
104	2	3	4000	1
103	3	3	3211	3
103	2	3	2345	4

Dense_Rank()

```
select emp_id,product_id,dept,sales,
rank() over(partition by dept order by sales desc)
as 'total sale',
Dense_rank() over(partition by dept order by sales
desc)
as 'Dense total sale'
from employee_sales;
```

Dense_Rank() syntax



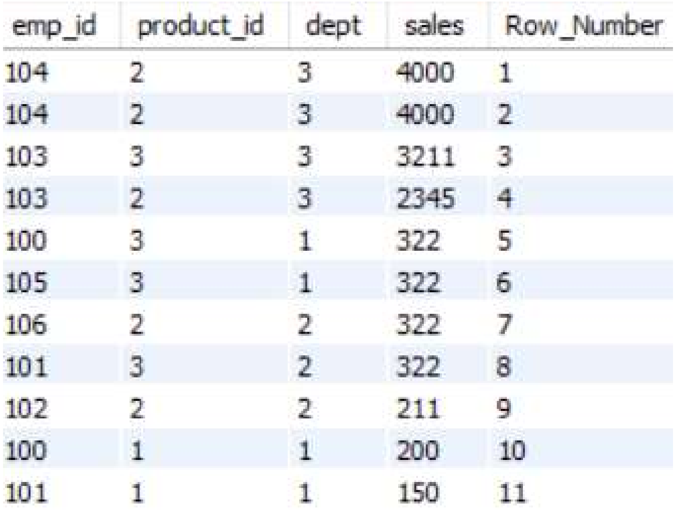
emp_id	product_id	dept	sales	total sale	Dense total sale
100	3	1	322	1	1
105	3	1	322	1	1
100	1	1	200	3	2
101	1	1	150	4	3
106	2	2	322	1	1
101	3	2	322	1	1
102	2	2	211	3	2
104	2	3	4000	1	1
104	2	3	4000	1	1
103	3	3	3211	3	2
103	2	3	2345	4	3

ROW_NUMBER()

```
select emp_id,product_id,dept,sales,

row_number() over( order by sales desc)

as 'Row_Number'

from employee_sales;
```

Row_Number() syntax

Without partition by

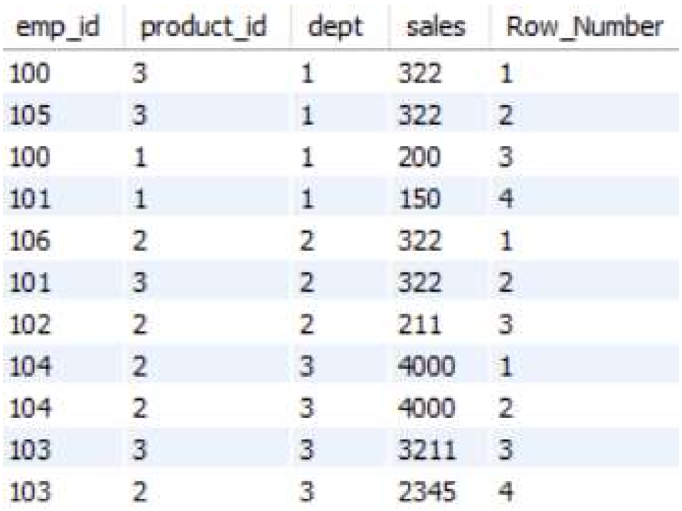


emp_id	product_id	dept	sales	Row_Number
104	2	3	4000	1
104	2	3	4000	2
103	3	3	3211	3
103	2	3	2345	4
100	3	1	322	5
105	3	1	322	6
106	2	2	322	7
101	3	2	322	8
102	2	2	211	9
100	1	1	200	10
101	1	1	150	11

Row_Number() syntax

With partition by

```
select emp_id,product_id,dept,sales,

row_number() over( partition by dept
order by sales desc)

as 'Row_Number'

from employee_sales;
```



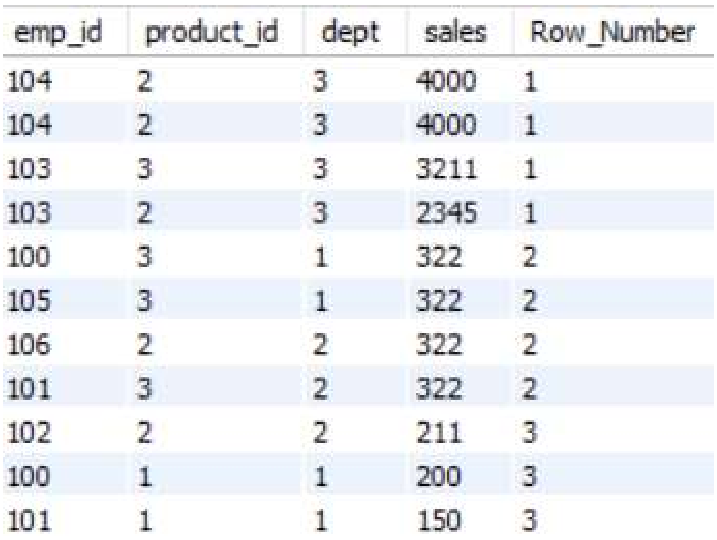
emp_id	product_id	dept	sales	Row_Number
100	3	1	322	1
105	3	1	322	2
100	1	1	200	3
101	1	1	150	4
106	2	2	322	1
101	3	2	322	2
102	2	2	211	3
104	2	3	4000	1
104	2	3	4000	2
103	3	3	3211	3
103	2	3	2345	4

NTILE(N)

```
select emp_id,product_id,dept,sales,

ntile(3) over( order by sales desc)

as 'Row_Number'

from employee_sales;
```

NTILE(N) syntax

Without partition by

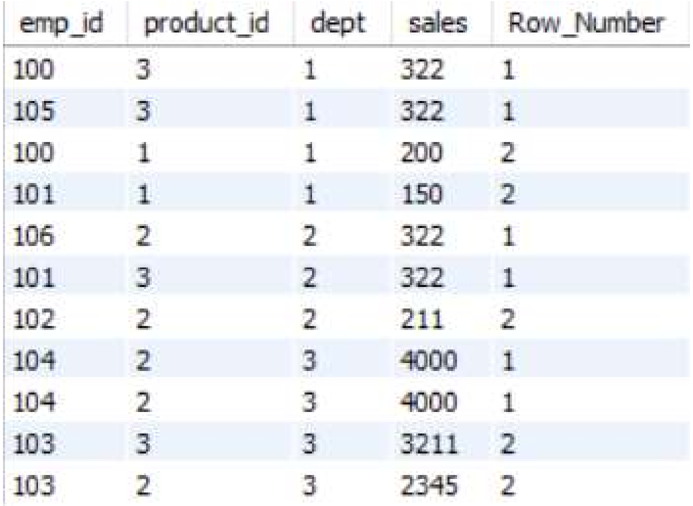


emp_id	product_id	dept	sales	Row_Number
104	2	3	4000	1
104	2	3	4000	1
103	3	3	3211	1
103	2	3	2345	1
100	3	1	322	2
105	3	1	322	2
106	2	2	322	2
101	3	2	322	2
102	2	2	211	3
100	1	1	200	3
101	1	1	150	3

NTILE(N) syntax

With partition by

```
select emp_id,product_id,dept,sales,

ntile(2) over( partition by dept order by sales desc)

as 'Row_Number'

from employee_sales;
```



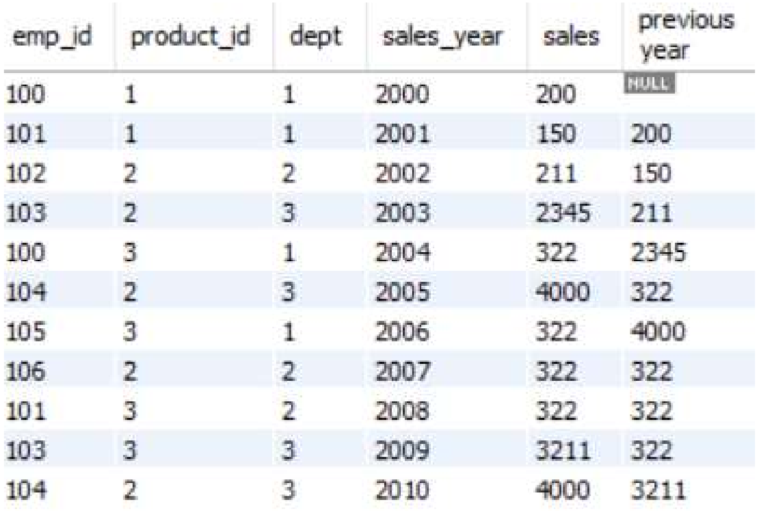
emp_id	product_id	dept	sales	Row_Number
100	3	1	322	1
105	3	1	322	1
100	1	1	200	2
101	1	1	150	2
106	2	2	322	1
101	3	2	322	1
102	2	2	211	2
104	2	3	4000	1
104	2	3	4000	1
103	3	3	3211	2
103	2	3	2345	2

LAG()

```
select emp_id,product_id,dept,sales_year,sales,
LAG(sales) over(order by sales_year  desc)
as 'previous year'
from employee_sales;
```

LAG() syntax

Without partition by



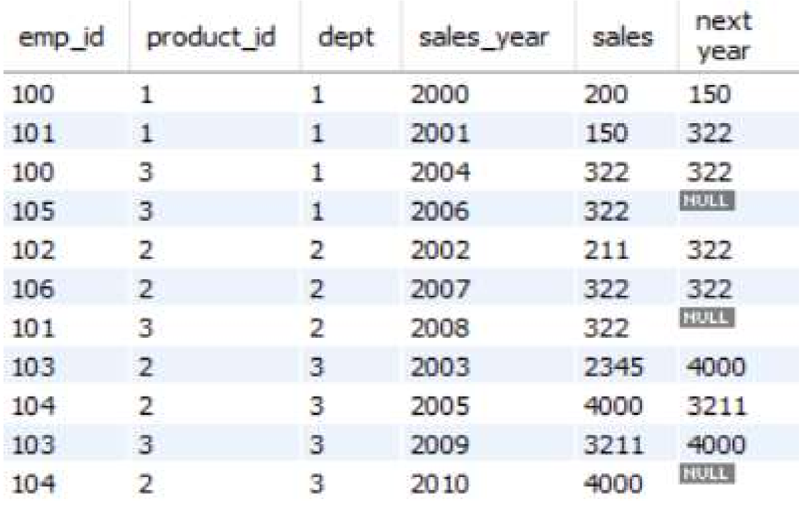
emp_id	product_id	dept	sales_year	sales	previous year
100	1	1	2000	200	NULL
101	1	1	2001	150	200
102	2	2	2002	211	150
103	2	3	2003	2345	211
100	3	1	2004	322	2345
104	2	3	2005	4000	322
105	3	1	2006	322	4000
106	2	2	2007	322	322
101	3	2	2008	322	322
103	3	3	2009	3211	322
104	2	3	2010	4000	3211

LEAD()

LEAD() syntax

With partition by

```
select emp_id,product_id,dept,sales_year,sales,
Lead(sales) over(partition by dept order by
sales_year)
as 'next year'
from employee_sales;
```

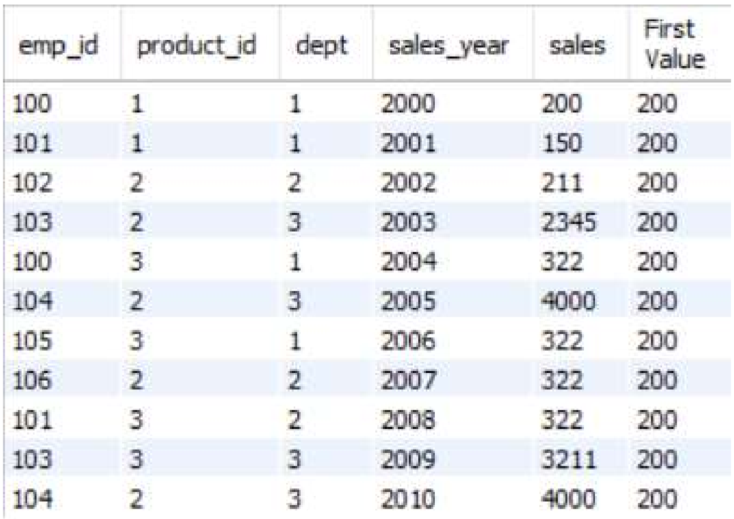


emp_id	product_id	dept	sales_year	sales	next year
100	1	1	2000	200	150
101	1	1	2001	150	322
100	3	1	2004	322	322
105	3	1	2006	322	NULL
102	2	2	2002	211	322
106	2	2	2007	322	322
101	3	2	2008	322	NULL
103	2	3	2003	2345	4000
104	2	3	2005	4000	3211
103	3	3	2009	3211	4000
104	2	3	2010	4000	NULL

FIRST_VALUE()

```
select emp_id,product_id,dept,sales_year,sales,
FIRST_VALUE(sales) over(order by sales_year)
as 'First Value'
from employee_sales;
```

FIRST_VALUE() syntax

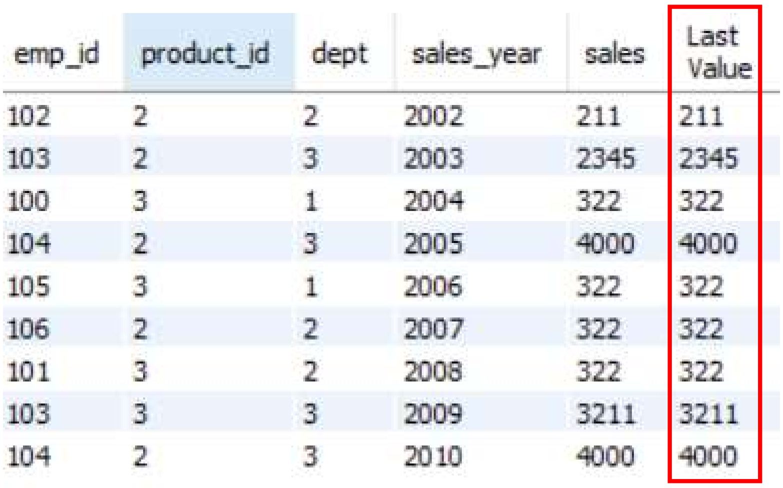


emp_id	product_id	dept	sales_year	sales	First Value
100	1	1	2000	200	200
101	1	1	2001	150	200
102	2	2	2002	211	200
103	2	3	2003	2345	200
100	3	1	2004	322	200
104	2	3	2005	4000	200
105	3	1	2006	322	200
106	2	2	2007	322	200
101	3	2	2008	322	200
103	3	3	2009	3211	200
104	2	3	2010	4000	200

LAST_VALUE()

```
select emp_id,product_id,dept,sales_year,sales,
FIRST_VALUE(sales) over(order by sales_year)
as 'First Value'
from employee_sales;
```

LAST_VALUE() syntax Default



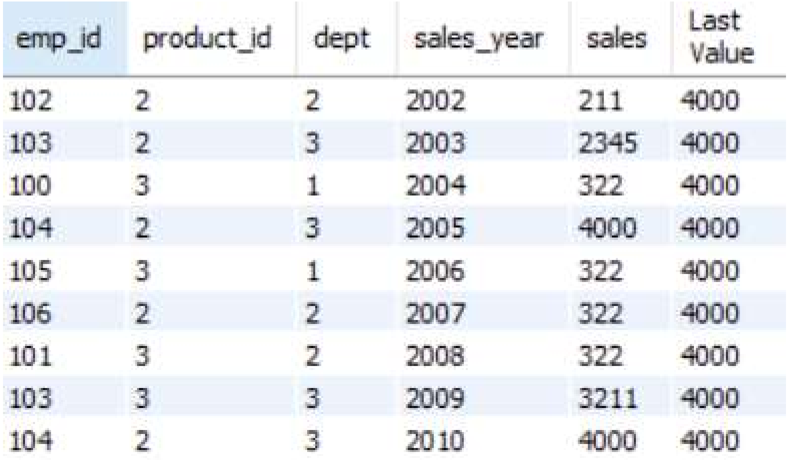
emp_id	product_id	dept	sales_year	sales	Last Value
102	2	2	2002	211	211
103	2	3	2003	2345	2345
100	3	1	2004	322	322
104	2	3	2005	4000	4000
105	3	1	2006	322	322
106	2	2	2007	322	322
101	3	2	2008	322	322
103	3	3	2009	3211	3211
104	2	3	2010	4000	4000

Default value:

RANGE BETWEEN
UNBOUNDED PRECEDING
AND CURRENT ROW

LAST_VALUE() syntax

```
select emp_id,product_id,dept,sales_year,sales,

last_VALUE(sales) over(order by sales_year rows between
unbounded preceding and unbounded following )

as 'Last Value'

from employee_sales;
```



emp_id	product_id	dept	sales_year	sales	Last Value
102	2	2	2002	211	4000
103	2	3	2003	2345	4000
100	3	1	2004	322	4000
104	2	3	2005	4000	4000
105	3	1	2006	322	4000
106	2	2	2007	322	4000
101	3	2	2008	322	4000
103	3	3	2009	3211	4000
104	2	3	2010	4000	4000

5. CASE statement in SQL

- The CASE statement goes through conditions and returns a value when the first condition is met (like an if-then-else statement). So, once a condition is true, it will stop reading and return the result. If no conditions are true, it returns the value in the ELSE clause.

- If there is no ELSE part and no conditions are true, it returns NULL.

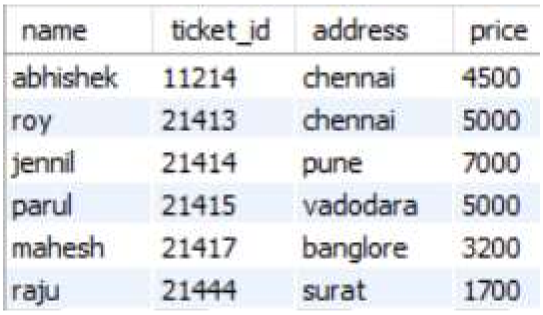
Example:1

CASE statement syntax:

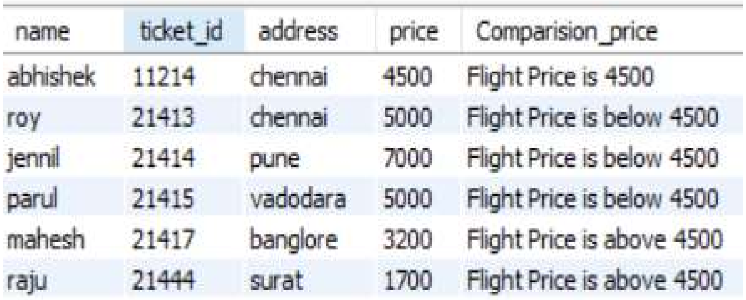
```
select *,
CASE
  WHEN price>4500 THEN 'Flight Price is below 4500'
   WHEN price=4500 THEN 'Flight Price is 4500'
   ELSE 'Flight Price is above 4500'
END AS Comparision_price
from flight;
```

Default Syntax:

```
CASE
    WHEN condition1 THEN result1
    WHEN condition2 THEN result2
    WHEN conditionN THEN resultN
    ELSE result
END;
```



name	ticket_id	address	price
abhishek	11214	chennai	4500
roy	21413	chennai	5000
jennil	21414	pune	7000
parul	21415	vadodara	5000
mahesh	21417	banglore	3200
raju	21444	surat	1700



name	ticket_id	address	price	Comparision_price
abhishek	11214	chennai	4500	Flight Price is 4500
roy	21413	chennai	5000	Flight Price is below 4500
jennil	21414	pune	7000	Flight Price is below 4500
parul	21415	vadodara	5000	Flight Price is below 4500
mahesh	21417	banglore	3200	Flight Price is above 4500
raju	21444	surat	1700	Flight Price is above 4500

Example:2

CASE statement syntax:

```
SELECT * FROM flight
ORDER BY
(CASE
   WHEN price IS NULL THEN name
   ELSE price
END);
```