HIVE FUNCTIONS

HIVE FUNCTIONS

HIVE HAS SEVERAL TYPES OF BUILT IN FUNCTIONS

WHICH ADD A LOT MORE FUNCTIONALITY AS OPPOSED TO TRADITIONAL SQL

HIVE FUNCTIONS

THERE ARE 3 TYPES OF FUNCTIONS

BASED ON THE WAY THEY PROCESS THE DATA

HIVE FUNCTIONS

THERE ARE 3 TYPES OF FUNCTIONS

STANDARD FUNCTIONS

AGGREGATE FUNCTIONS

TABLE GENERATING FUNCTIONS

HIVE FUNCTIONS

THERE ARE 3 TYPES OF FUNCTIONS

STANDARD FUNCTIONS

AGGREGATE FUNCTIONS

TABLE GENERATING FUNCTIONS

HIVE BUILT-IN FUNCTIONS

STANDARD FUNCTIONS

STANDARD FUNCTIONS TAKE A ROW/
COLUMNS IN A ROW AS ARGUMENTS

```
SELECT concat(first_name," ",last_name) from employees;
```

HIVE BUILT-IN FUNCTIONS

STANDARD FUNCTIONS

THEY RETURN A SINGLE RESULT FOR EACH ROW

```
SELECT concat(first_name," ",last_name) from employees;

Vitthal Srinivasan
Janani Ravi
Swetha Kolalapudi
```

HIVE BUILT-IN FUNCTIONS

STANDARD FUNCTIONS

THESE INCLUDE MATHEMATICAL FUNCTIONS
LIKE EXP(), LN(), SQRT(), POW() ETC

HIVE BUILT-IN FUNCTIONS

STANDARD FUNCTIONS

STRING FUNCTIONS LIKE LENGTH(), REVERSE(), REGEXP_REPLACE() AND MANY OTHERS

HIVE FUNCTIONS

THERE ARE 3 TYPES OF FUNCTIONS

STANDARD FUNCTIONS

AGGREGATE FUNCTIONS

TABLE GENERATING FUNCTIONS

HIVE FUNCTIONS

THERE ARE 3 TYPES OF FUNCTIONS

STANDARD FUNCTIONS

AGGREGATE FUNCTIONS

TABLE GENERATING FUNCTIONS

HIVE BUILT-IN FUNCTIONS

AGGREGATE FUNCTIONS

SUM(), COUNT(),AVG() ETC

THESE FUNCTIONS TAKE MULTIPLE ROWS AS INPUT

HIVE BUILT-IN FUNCTIONS

AGGREGATE FUNCTIONS

SUM(), COUNT(),AVG() ETC

THEY RETURN A SINGLE RESULT

HIVE BUILT-IN FUNCTIONS

AGGREGATE FUNCTIONS

SUM(), COUNT(),AVG() ETC

THESE FUNCTIONS ARE NORMALLY USED WITH A GROUP BY CLAUSE

HIVE BUILT-IN FUNCTIONS

AGGREGATE FUNCTIONS

```
SELECT Productname, sum(Revenue)
from sales_data
group by Productname;
```

HIVE FUNCTIONS

THERE ARE 3 TYPES OF FUNCTIONS

STANDARD FUNCTIONS

AGGREGATE FUNCTIONS

TABLE GENERATING FUNCTIONS

HIVE FUNCTIONS

THERE ARE 3 TYPES OF FUNCTIONS

STANDARD FUNCTIONS

AGGREGATE FUNCTIONS

TABLE GENERATING FUNCTIONS

HIVE BUILT-IN FUNCTIONS

TABLE GENERATING FUNCTIONS

THESE FUNCTIONS TAKE 1
ROW AS INPUT AND
OUTPUT MULTIPLE ROWS

HIVE BUILT-IN FUNCTIONS

TABLE GENERATING FUNCTIONS

THESE FUNCTIONS NORMALLY OPERATE ON COLLECTION DATA TYPES LIKE ARRAYS, MAPS, STRUCTS

```
SELECT explode(array(1,2,3));
```

HIVE BUILT-IN FUNCTIONS

TABLE GENERATING FUNCTIONS

THEY OUTPUT 1 ROW FOR EACH ELEMENT OF THE COLLECTION

```
SELECT explode(array(1,2,3));
1
2
3
```

HIVE FUNCTIONS

THERE ARE 3 TYPES OF FUNCTIONS

STANDARD FUNCTIONS

AGGREGATE FUNCTIONS

TABLE GENERATING FUNCTIONS

HIVE FUNCTIONS

THERE ARE 3 TYPES OF FUNCTIONS

STANDARD FUNCTIONS

AGGREGATE FUNCTIONS

TABLE GENERATING FUNCTIONS

LET'S STUDY A FEW SPECIFIC EXAMPLES OF HOW TO USE FUNCTIONS

HIVE FUNCTIONS

CASE..WHEN

COMPUTING A COLUMN BASED ON CONDITIONALS

CASE..WHEN

SAY WE HAD A TABLE WITH EMPLOYEE NAMES AND TENURE IN YEARS

EmpID	EmpName	Tenure
1	Vitthal	1
2	Swetha	4
3	Janani	2
4	Navdeep	3

EMPLOYEES WITH HIGHER TENURES GET EXTRA VACATION DAYS

CASE..WHEN

THE RULE FOR EXTRA VACATION DAYS IS

IF TENURE < 2, THEN 0

IF TENURE IN 2-3 YRS, THEN 2 DAYS

IF TENURE > 3 YRS, THEN 3 DAYS

CASE..WHEN

IF TENURE < 2, THEN 0

IF TENURE IN 2-3 YRS, THEN 2 DAYS

IF TENURE > 3 YRS, THEN 3 DAYS

THIS IS A PERFECT EXAMPLE FOR THE USE OF CASE..WHEN

CASE..WHEN

```
from employeeTenures SELECT empname,
```

IF TENURE < 2, THEN 0

IF TENURE IN 2-3 YRS, THEN 2 DAYS

IF TENURE > 3 YRS, THEN 3 DAYS

CASE..WHEN

IF TENURE < 2, THEN 0

IF TENURE IN 2-3 YRS, THEN 2 DAYS

IF TENURE > 3 YRS, THEN 3 DAYS

```
from employeeTenures SELECT empname,

CASE WHEN TENURE<2 THEN 0
```

CASE..WHEN

IF TENURE < 2, THEN 0

IF TENURE IN 2-3 YRS, THEN 2 DAYS

IF TENURE > 3 YRS, THEN 3 DAYS

```
from employeeTenures SELECT empname,
CASE WHEN TENURE<2 THEN 0
WHEN TENURE>=2 AND TENURE<=3 THEN 2
```

CASE..WHEN

IF TENURE < 2, THEN 0

IF TENURE IN 2-3 YRS, THEN 2 DAYS

IF TENURE > 3 YRS, THEN 3 DAYS

```
from employeeTenures SELECT empname,
CASE WHEN TENURE<2 THEN 0
WHEN TENURE>=2 AND TENURE<=3 THEN 2
WHEN TENURE>3 THEN 3
```

CASE..WHEN

IF TENURE < 2, THEN 0

IF TENURE IN 2-3 YRS, THEN 2 DAYS

IF TENURE > 3 YRS, THEN 3 DAYS

```
from employeeTenures SELECT empname,
CASE WHEN TENURE<2 THEN 0
WHEN TENURE>=2 AND TENURE<=3 THEN 2
WHEN TENURE>3 THEN 3
END AS EXTRA_VACATION_DAYS;
```

CASE..WHEN

```
from employeeTenures SELECT empname,

CASE WHEN TENURE<2 THEN 0

WHEN TENURE>=2 AND TENURE<=3 THEN 2

WHEN TENURE>3 THEN 3

END AS EXTRA_VACATION_DAYS;
```

A SERIES OF CONDITIONS AND THE CORRESPONDING RESULTS

CASE..WHEN

```
from employeeTenures SELECT empname,
CASE WHEN TENURE<2 THEN 0
WHEN TENURE>=2 AND TENURE<=3 THEN 2
WHEN TENURE>3 THEN 3
END AS EXTRA_VACATION_DAYS;
```

USE END TO SIGNAL THE END OF THE LIST OF CONDITIONS

HIVE FUNCTIONS

SIZE()

FINDING THE SIZE OF AN ARRAY/MAP

HIVE FUNCTIONS

SIZE()

HIVE HAS A FEW BUILT IN FUNCTIONS FOR COLLECTION DATA TYPES LIKE ARRAYS, MAPS, STRUCTS ETC

HIVE FUNCTIONS

SIZE()

THE SIZE() FUNCTION WORKS FOR BOTH ARRAYS AND MAPS

HIVE FUNCTIONS

SIZE()

FOR ARRAYS IT RETURNS THE NUMBER OF ELEMENTS

HIVE FUNCTIONS

SIZE()

FOR MAPS IT RETURNS THE NUMBER OF KEY-VALUE PAIRS

HIVE FUNCTIONS

SIZE()

```
SELECT SIZE(ARRAY(1,2,3));

3
```

THE ARRAY HAS 3 ELEMENTS

HIVE FUNCTIONS

SIZE()

```
SELECT SIZE(MAP("NAME","SWETHA","AGE",30));
```

2

THE MAP HASE 2 KEY-VALUE PAIRS

HIVE FUNCTIONS

CAST()

CONVERTING FROM 1 DATATYPE TO ANOTHER

HIVE FUNCTIONS

CAST()

HIVE ENFORCES TYPE SAFETY

HIVE FUNCTIONS

CAST()

THIS MEANS THAT YOU CANNOT INSERT DATA OF 1 TYPE INTO A COLUMN OF A DIFFERENT TYPE

EX: IF YOU TRY TO INSERT A STRING INTO A FLOAT COLUMN , AN ERROR WILL BE THROWN

HIVE FUNCTIONS

CAST() HIVE FUNCTIONS HAVE A SIGNATURE

WITH INPUT DATA TYPE

AND RETURN DATA TYPE

ERRORS ARE THROWN IF THESE ARE VIOLATED

HIVE FUNCTIONS

CAST()

WHAT IF YOU HAVE A STRING

AND YOU NEED TO INSERT IT TO AN INTEGER COLUMN?

HIVE FUNCTIONS

CAST()

THE INTEGER WILL NEED TO BE CAST TO A INTEGER DATA TYPE FIRST

HIVE FUNCTIONS

CAST()

```
select cast("25" as bigint);
```

THIS WILL CONVERT THE STRING "25" TO THE DATATYPE BIGINT

HIVE FUNCTIONS

CAST()

```
select cast("25" as bigint);
```

THE COLUMN/VALUE TO BE CAST

HIVE FUNCTIONS

CAST()

```
select cast("25" as bigint);
```

THE DATA TYPE TO BE CAST INTO

HIVE FUNCTIONS

CAST()

IF A VALUE CANNOT BE CONVERTED INTO THE SPECIFIED DATATYPE , NULL IS RETURNED

```
select cast("abc" as bigint);
```

NULL

HIVE FUNCTIONS

EXPLODE()

GENERATING MULTIPLE ROWS FROM A COLLECTION

EXPLODE()

THIS IS AN EXAMPLE OF A TABLE GENERATING FUNCTION

THIS FUNCTION TAKES 1 ROW AND RETURNS MULTIPLE ROWS

EXPLODE()

THE EXPLODE FUNCTION IS USED WITH COLLECTION DATA TYPES LIKE ARRAYS, MAPS ETC

1 ROW IS RETURNED FOR EACH ELEMENT IN THE COLLECTION

EXPLODE()

IN HIVE THE COLLECTION DATA TYPES ARE REALLY USEFUL TO EMBED LOT'S OF INFORMATION IN 1 ROW OF 1 TABLE

EXPLODE()

IN TRADITIONAL DATABASES, WE PREFER TO KEEP THE DATA IN NORMAL FORM

THIS MEANS THAT REDUNDANCY IS MINIMIZED AND DATA IS KEPT IN SEPARATE TABLES

EXPLODE()

IN HIVE, THE IDEA IS TO REDUCE THE NUMBER OF DISK SEEKS

IF MORE INFORMATION STORED IN A SINGLE TABLE, WE JUST NEED TO READ THAT TABLE

EXPLODE()

LET'S TAKE AN EXAMPLE

WE HAVE AN EMPLOYEES DATABASE

WE WANT TO CAPTURE EMPLOYEE NAME, ADDRESS, SUBORDINATES

EXPLODE()

A TRADITIONAL RDBMS WOULD MODEL IT AS 3 TABLES

EmpID	EmpName	AddressId
1	Vitthal	1

AddressId	Street	City
1	Bellandur	Bangalore

EmpID	SubordinateEmpID
1	3
1	4
1	8

EXPLODE()

A TRADITIONAL RDBMS WOULD MODEL IT AS 3 TABLES

EmpID	EmpName	AddressId
1	Vitthal	1

AddressId	Street	City
1	Bellandur	Bangalore

YOU WOULD NEED TO JOIN 2 TABLES TO FETCH THE ADDRESS

EXPLODE()

A TRADITIONAL RDBMS WOULD MODEL IT AS 3 TABLES

EmpID	EmpName	AddressId
1	Vitthal	1

EmpID	SubordinateEmpID
1	3
1	4
1	8

YOU WOULD NEED TO JOIN THESE 2 TABLES TWICE TO GET THE LIST OF SUBORDINATES FOR AN EMPLOYEE

EXPLODE()

A TRADITIONAL RDBMS WOULD MODEL IT AS 3 TABLES

IN AN RDBMS THESE JOINS CAN BE MADE EFFICIENT WITH THE ADDITION OF INDICES AND CONSTRAINTS

EXPLODE()

A TRADITIONAL RDBMS WOULD MODEL IT AS 3 TABLES

IN HIVE THIS WHOLE MODEL WOULD BE VERY INEFFICIENT!

WE WANT TO MINIMIZE THE DISK SEEKS I.E. SCANS OF TABLES AS MUCH AS POSSIBLE

EXPLODE()

IN HIVE WE CAN EMBED ALL 3 TABLES INTO A SINGLE TABLE

EXPLODE()

IN HIVE WE CAN EMBED ALL 3 TABLES INTO A SINGLE TABLE

EmpID	EmpName	Address	Subordinates
1	Vitthal	<STRUCT>	<ARRAY>

```
"Street":"Bellandur",
"City":"Bangalore"
```

```
("Anuradha",
"Arun",
"Swetha")
```

EXPLODE()

IN HIVE WE CAN EMBED ALL 3 TABLES INTO A SINGLE TABLE

EmpID	EmpName	Address	Subordinates
1	Vitthal	<STRUCT>	<ARRAY>

THIS WAY WE CAN READ ALL THE DATA WITH 1 SCAN OF 1 TABLE

EXPLODE()

IN HIVE WE CAN EMBED ALL 3 TABLES INTO A SINGLE TABLE

EmpID	EmpName	Address	Subordinates
1	Vitthal	<STRUCT>	<ARRAY>

THE DATA IS READ IN A NESTED FORM AND THEN BROKEN APART DURING THE PROCESSING PHASE

EXPLODE()

IN HIVE WE CAN EMBED ALL 3 TABLES INTO A SINGLE TABLE

EmpID	EmpName	Address	Subordinates
1	Vitthal	<STRUCT>	<ARRAY>

THE EXPLODE() FUNCTION HELPS IN BREAKING UP THE CONTENTS OF THIS NESTED DATA INTO ROWS

EXPLODE()

```
select explode(subordinates) from
employeeDetails where empName="Vitthal";
```

```
Anuradha
Arun
Swetha
```

THE EXPLODE() FUNCTION BROKE UP THE ARRAY OF SUBORDINATES INTO MULTIPLE ROWS

EXPLODE()

```
select explode(subordinates) from
employeeDetails where empName="Vitthal";
```

```
Anuradha
Arun
Swetha
```

WHAT IF WE WANTED TO INCLUDE THE MANAGER NAME IN THE RESULTS?

EXPLODE()

```
select empName, explode(subordinates)
from employeeDetails where
empName="Vitthal";
```

THIS WILL THROW AN ERROR!

EXPLODE()

```
select empName, explode(subordinates)
from employeeDetails where
empName="Vitthal";
```

THIS IS BECAUSE EMPNAME HAS 1 ROW,
BUT EXPLODE(SUBORDINATES) HAS 3 ROWS

EXPLODE()

```
select empName, explode(subordinates)
from employeeDetails where
empName="Vitthal";
```

THE RESULT OF EXPLODE(SUBORDINATES) IS LIKE A TABLE

EXPLODE()

```
select empName, explode(subordinates)
from employeeDetails where
empName="Vitthal";
```

THIS TABLE NEEDS TO BE JOINED BACK TO THE ORIGINAL TABLE

EXPLODE()

```
select empName, explode(subordinates)
from employeeDetails where
empName="Vitthal";
```

THIS IS ACHIEVED USING A CONSTRUCT CALLED LATERAL VIEW

EXPLODE()

```
select empName, subordinates from
employeeDetails where empName="Vitthal"
LATERAL VIEW explode(subordinates) exp as
subordinates;
```

```
Vitthal Anuradha
Vitthal Arun
Vitthal Swetha
```

LATERAL VIEW

EXPLODE()

```
select empName, subordinates from
employeeDetails where empName="Vitthal"
LATERAL VIEW explode(subordinates) exp as
subordinates;
```

THIS IS A COLUMN THAT IS PART OF THE EXPLODED TABLE

EXPLODE()

```
select empName, subordinates from employeeDetails
where empName="Vitthal"
LATERAL VIEW explode(subordinates) exp as
subordinates;
```

THIS GENERATES THE EXPLODED TABLE

EXPLODE()

```
select empName, subordinates from employeeDetails
where empName="Vitthal"
LATERAL VIEW explode(subordinates) exp as
subordinates;
```

THE EXPLODED TABLE NEEDS TO BE GIVEN AN ALIAS

EXPLODE()

```
select empName, subordinates from employeeDetails
where empName="Vitthal"
LATERAL VIEW explode(subordinates) exp as
subordinates;
```

LATERAL VIEW JOINS THE EXPLODED TABLE BACK TO THE ORIGINAL TABLE

EXPLODE()

```
select empName, subordinates from employeeDetails
where empName="Vitthal"
LATERAL VIEW explode(subordinates) exp as
subordinates;
```

THIS IS BY DEFAULT AN INNER JOIN

EXPLODE()

```
select empName, subordinates from employeeDetails
where empName="Vitthal"
LATERAL VIEW explode(subordinates) exp as
subordinates;
```

IE, IF AN EMPLOYEE HAS NULL IN THE SUBORDINATES COLUMN, THAT EMPLOYEE WILL BE EXCLUDED

EXPLODE()

```
select empName, subordinates from employeeDetails where
empName="Vitthal"
LATERAL VIEW explode(subordinates) exp as
subordinates;
```

THE COLUMN ALIAS OF THE EXPLODED TABLE AFTER IT IS JOINED BACK TO THE ORIGINAL TABLE

EXPLODE()

```
select empName, subordinates from employeeDetails where
empName="Vitthal"
LATERAL VIEW explode(subordinates) exp as subordinates;
```

THIS COLUMN ALIAS APPEARS IN THE SELECT PORTION