Database Management System (DBMS)
#2305CS101

Darshan UNIVERSITY

Unit-1

Advanced SQL Concepts

Prof. Firoz A. Sherasiya
Computer Science & Engineering Department
Darshan University, Rajkot

firoz.sherasiya@darshan.ac.in
9879879861

✓ Outline

- Group by
- Joins
- Subquery
- Keys
- System Functions
- User Defined Functions (UDF)
- Stored Procedures
- Parameters in Stored Procedures
- Procedures v/s Functions
- Cursor
- Trigger
- Exception Handling
- TCL and DCL Commands

Group by

Section - 1

Aggregate Functions

- An aggregate function in SQL **performs a calculation on multiple values and returns a single scalar value**.
- SQL provides many aggregate functions that include avg(), count(), sum(), min(), max(), etc.
- An aggregate function **ignores NULL values when it performs the calculation.**
- We often use aggregate functions with the GROUP BY and HAVING clauses of the SELECT statement.

Syntax : Aggregate Functions

```
aggregate_function ( DISTINCT |  ALL expression)
```

1. Specify the name of function that you want to use such as AVG(), SUM(), MAX() etc.
2. Use DISTINCT if you want only distinct values are considered in the calculation or ALL if all values are considered in the calculation. By default, ALL is used if you don't specify.
3. The expression can be a column of a table or an expression that consists of multiple columns with arithmetic operators.

Aggregate Functions (Cont..)

- The following table shows the SQL Server aggregate functions:

Sr.	Aggregate function	Description
1	AVG()	The AVG() aggregate function calculates the average of non-NULL values in a set.
2	COUNT()	The COUNT() aggregate function returns the number of rows in a group, including rows with NULL values.
3	MAX()	The MAX() aggregate function returns the highest value (maximum) in a set of non-NULL values.
4	MIN()	The MIN() aggregate function returns the lowest value (minimum) in a set of non-NULL values.
5	SUM()	The SUM() aggregate function returns the summation of all non-NULL values a set.

Aggregate Functions Example

System Functions

Student

Rno	Name	Branch	Semester	CPI
101	Ramesh	CE	3	9
102	Mahesh	EC	3	8
103	Suresh	ME	4	7
104	Amit	EE	4	8
105	Anita	CE	4	8
106	Reeta	ME	3	7
107	Rohit	EE	4	9
108	Chetan	CE	3	8
109	Rakesh	CE	4	9

Example Find out sum of CPI of all students.

Answer

```
Select SUM(CPI) AS [Sum] From Student
```

Output

Sum
73.00

Example Find out maximum & minimum CPI.

Answer

```
Select MAX(CPI) AS [Max], MIN(CPI) AS [Min]
From Student
```

Output

Max	Min
9.00	7.00

Example Count the number of students.

Answer

```
Select COUNT(RNo) AS [Total] From Student
```

Output

Total
9

Example Find out average of CPI of all students.

Answer

```
Select AVG(CPI) AS [Avg] From Student
```

Output

Avg
8.111111

Darshan UNIVERSITY

Exercise – Aggregate functions

1. Display highest salary.
2. Display lowest salary.
3. Display total salary.
4. Display average of salary.
5. Display total of all faculties salary.
6. Count total record in the table.
7. Count total ID.
8. Display highest salary from Computer department.
9. Display minimum salary from civil department.
10. Display average salary from Rajkot city.
11. Display maximum, minimum, average and total salary.
12. Display all the faculties whose salary is less then average salary.

Faculty

ID	Name	Salary	City	Branch
258	Ankit Patel	50000	Jetpur	Electrical
742	Ketan Parmar	75000	Baroda	Computer
325	Manan Doshi	65000	Gondal	Civil
125	Mitesh Manek	55000	Rajkot	Computer
312	Ketan Akbari	28000	Rajkot	Civil
Null	Bhavin Patel	35000	Jamnagar	Mechanical
258	Ankit Patel	50000	Jetpur	Electrical
742	Ketan Parmar	75000	Baroda	Computer
325	Manan Doshi	65000	Gondal	Civil

Group by

GROUP OF ENGINEERS

COMPUTER ENGINEER

CIVIL ENGINEER

ELECTRIC ENGINEER

MECHANICAL ENGINEER

What is Group by?

- It creates a group of distinct values from available records.
- It groups records based on the distinct values for specified columns.
- **Syntax:**

```
SELECT  COLUMN1, COLUMN2,  AGGREGATE FUNCTION (COLUMN)   FROM   NAME OF TABLE
GROUP BY   COLUMN1, COLUMN2,   …COLUMNn  ;
```

Aggregate Functions with Group By Example

Student

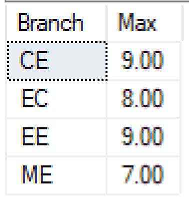
Rno	Name	Branch	Semester	CPI
101	Ramesh	CE	3	9
102	Mahesh	EC	3	8
103	Suresh	ME	4	7
104	Amit	EE	4	8
105	Anita	CE	4	8
106	Reeta	ME	3	7
107	Rohit	EE	4	9
108	Chetan	CE	3	8
109	Rakesh	CE	4	9

Example Find out Branch wise Maximum CPI.

Answer

```
Select Branch, MAX(CPI) AS [Max] From Student Group By Branch
```

Output



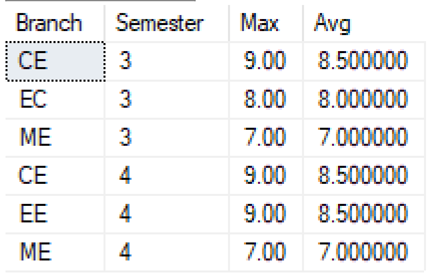
```
Branch  Max
CE      9.00
EC      8.00
EE      9.00
ME      7.00
```

Example Find out Branch wise Semester wise Minimum & Average CPI.

Answer

```
Select Branch, Semester, MAX(CPI) AS [Max], AVG(CPI) AS
[Avg] From Student Group By Branch, Semester
```

Output



```
Branch  Semester  Max   Avg
CE      3         9.00  8.500000
EC      3         8.00  8.000000
ME      3         7.00  7.000000
CE      4         9.00  8.500000
EE      4         9.00  8.500000
ME      4         7.00  7.000000
```

Aggregate Functions Group By with Filter Example (Cont..)

Student

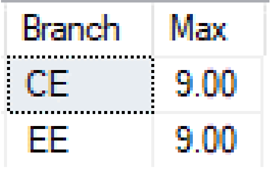
Rno	Name	Branch	Semester	CPI
101	Ramesh	CE	3	9
102	Mahesh	EC	3	8
103	Suresh	ME	4	7
104	Amit	EE	4	8
105	Anita	CE	4	8
106	Reeta	ME	3	7
107	Rohit	EE	4	9
108	Chetan	CE	3	8
109	Rakesh	CE	4	9

Example Find out All the Branches with maximum CPI, whose maximum CPI is more than 8.

Answer

```
Select Branch, MAX(CPI) AS [Max] From Student
Group By Branch
Having MAX(CPI) > 8
```

Output



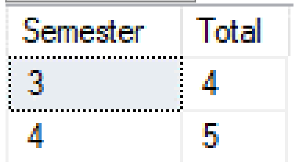
```
Branch  Max
CE      9.00
EE      9.00
```

Example Find out semester wise total students & arrange them in order with their count.

Answer

```
Select Semester, Count(Rno) AS [Total] From Student
Group By Semester
Order By Total
```

Output



```
Semester  Total
3         4
4         5
```

Aggregate Functions Group By with Filter Example (Cont..)

System Functions

Student

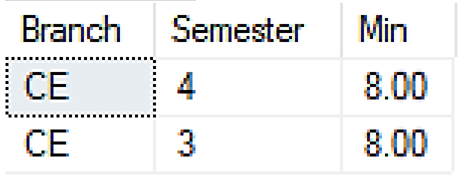
Rno	Name	Branch	Semester	CPI
101	Ramesh	CE	3	9
102	Mahesh	EC	3	8
103	Suresh	ME	4	7
104	Amit	EE	4	8
105	Anita	CE	4	8
106	Reeta	ME	3	7
107	Rohit	EE	4	9
108	Chetan	CE	3	8
109	Rakesh	CE	4	9

Example Find out Branch wise & Semester wise minimum CPI details of CE branch's students in which minimum CPI is greater than 7. Do arrange the result in descending order to semester.

Answer

```
Select Branch, Semester, MIN(CPI) AS [Min] From Student
Where Branch='CE'
Group By Branch, Semester
Having MIN(CPI) > 7
Order By Semester Desc
```

Output



Branch	Semester	Min
CE	4	8.00
CE	3	8.00

Exercise – Group by

1. Find branch wise highest salary.
2. Find city wise lowest salary.
3. Find branch wise highest, lowest and average salary.
4. Find average salary of Computer branch.
5. Find branch wise highest salary, where highest salary is more then 50000.
6. Find city wise, branch wise total salary.
7. Find city wise average salary and display then in ascending order.
8. Display branch wise maximum salary in descending order.
9. Find branch wise total faculties in descending order.
10. Find out branch wise & city wise total salary of computer branch with total salary is greater than 50000.do arrange the result in descending order to total salary.

Faculty

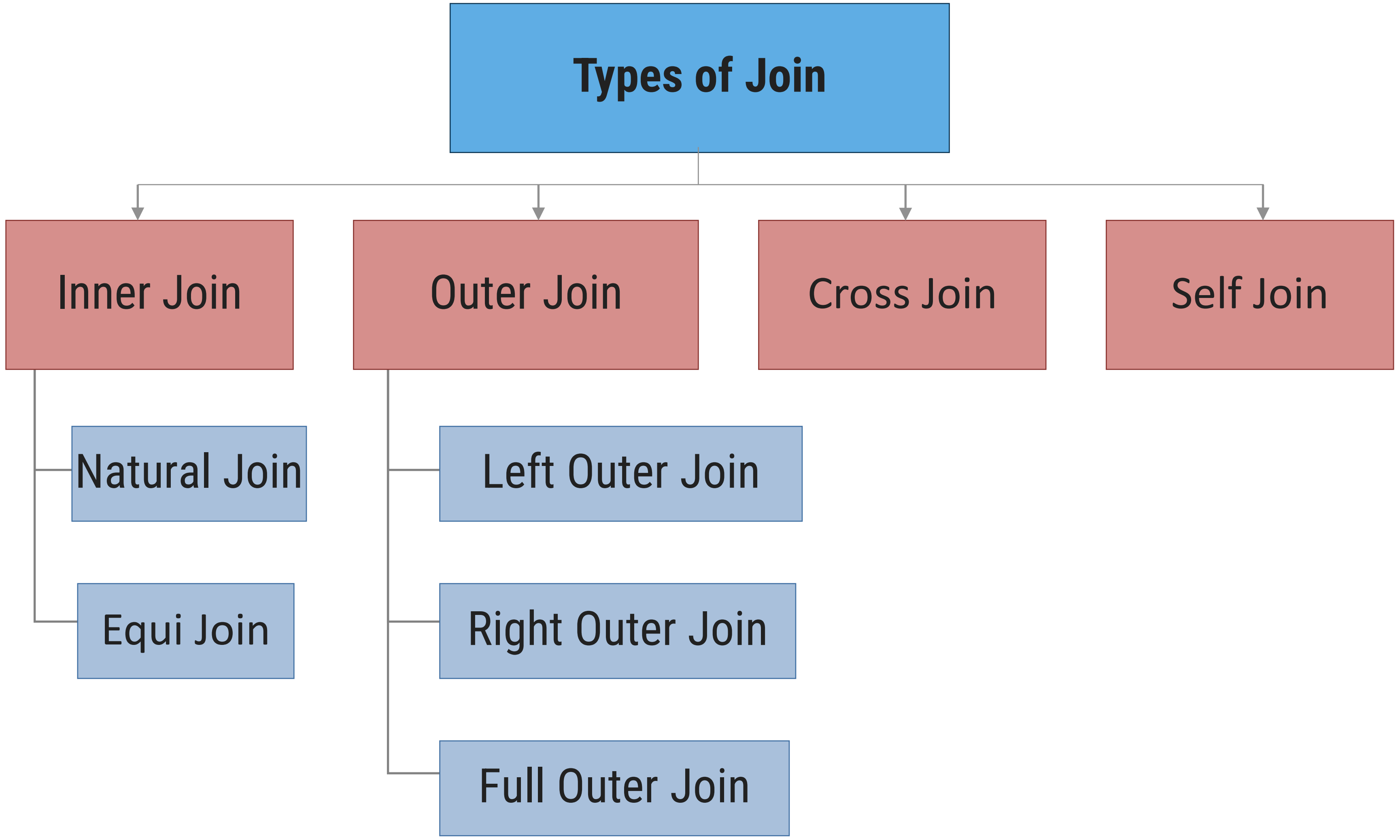
ID	Name	Salary	City	Branch
258	Ankit Patel	50000	Jetpur	Electrical
742	Ketan Parmar	75000	Baroda	Computer
325	Manan Doshi	65000	Gondal	Civil
125	Mitesh Manek	55000	Rajkot	Computer
312	Ketan Akbari	28000	Rajkot	Civil
Null	Bhavin Patel	35000	Jamnagar	Mechanical
258	Ankit Patel	50000	Jetpur	Electrical
742	Ketan Parmar	75000	Baroda	Computer
325	Manan Doshi	65000	Gondal	Civil

Join

Section - 2

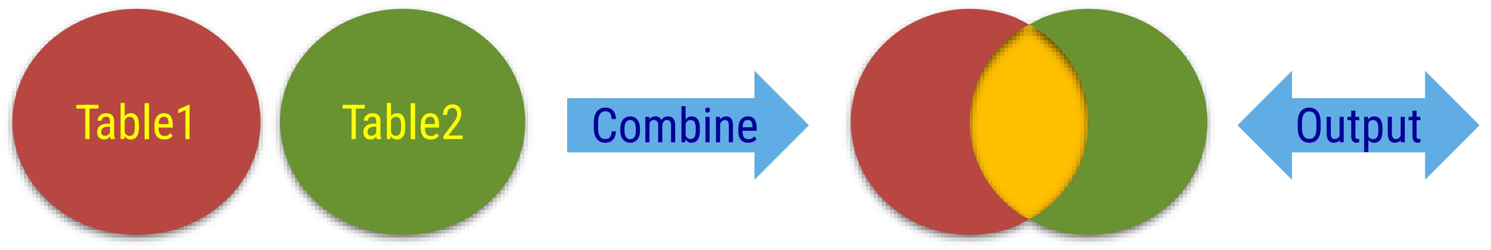
Joins

- An SQL JOIN clause is used to **combine rows** from two or more tables, **based on a common field** between them.



Inner Join

- Inner Join **returns** records that have **matching values** in **both tables**.



Syntax

```
SELECT columns
FROM Table1 INNER JOIN Table2
ON Table1.column=Table2.column ;
```

Inner Join(Cont..)

Example

```
SELECT Student.RNO, Student.Name, Student.Branch, Result.SPI
FROM Student INNER JOIN Result
ON Student.RNO= Result.RNO ;
```

Student

RNO	Name	Branch
101	Raju	CE
102	Amit	CE
103	Sanjay	ME
104	Neha	EC
105	Meera	EE
106	Mahesh	ME

Result

RNO	SPI
101	8.8
102	9.2
104	8.2
105	7
107	8.9

Output

RNO	Name	Branch	SPI
101	Raju	CE	8.8
102	Amit	CE	9.2
104	Neha	EC	8.2
105	Meera	EE	7

Inner Join(Cont..)

Inner Join without using Join Keyword

Syntax

```
SELECT Columns
FROM Table1 , Table2
WHERE Table1.column=Table2.column ;
```

Example

```
SELECT S.RNO, S.Name, S.Branch, R.SPI
FROM Student S , Result R
WHERE Student.RNO= Result.RNO ;
```

Student

RNO	Name	Branch
101	Raju	CE
102	Amit	CE
103	Sanjay	ME
104	Neha	EC
105	Meera	EE
106	Mahesh	ME

Result

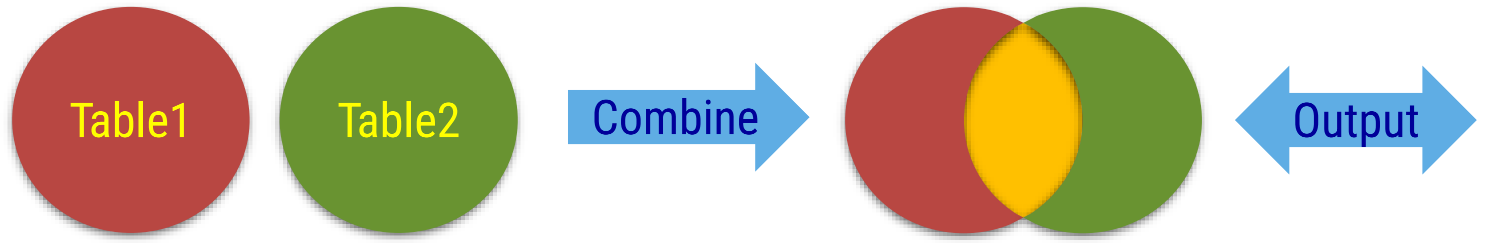
RNO	SPI
101	8.8
102	9.2
104	8.2
105	7
107	8.9

Output

RNO	Name	Branch	SPI
101	Raju	CE	8.8
102	Amit	CE	9.2
104	Neha	EC	8.2
105	Meera	EE	7

Left outer Join

- Left outer join **return all records from the left table**, **and** the **matched records from the right table**.



Syntax

```
SELECT columns
FROM Table1 LEFT OUTER JOIN Table2
ON Table1.column=Table2.column ;
```

Left outer Join(Cont..)

Example

```
SELECT Student.RNO, Student.Name, Student.Branch, Result.SPI
FROM Student LEFT OUTER JOIN Result
ON Student.RNO= Result.RNO ;
```

Student

RNO	Name	Branch
101	Raju	CE
102	Amit	CE
103	Sanjay	ME
104	Neha	EC
105	Meera	EE
106	Mahesh	ME

Result

RNO	SPI
101	8.8
102	9.2
104	8.2
105	7
107	8.9

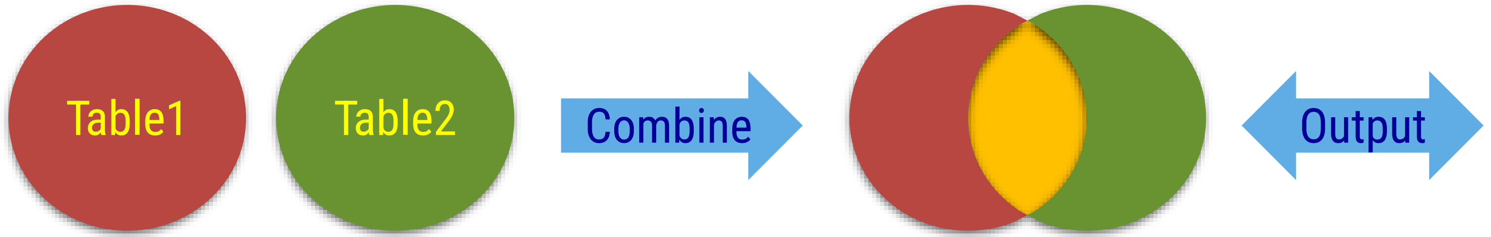
Output

RNO	Name	Branch	SPI
101	Raju	CE	8.8
102	Amit	CE	9.2
103	Sanjay	ME	*NULL*
104	Neha	EC	8.2
105	Meera	EE	7
106	Mahesh	ME	*NULL*

Darshan UNIVERSITY

Right outer Join

- Right outer join **return all records from the right table**, **and** the **matched records from the left table**.



Syntax

```
SELECT columns
FROM Table1 RIGHT OUTER JOIN Table2
ON Table1.column=Table2.column ;
```

Right outer Join(Cont..)

Example

```
SELECT Student.RNO, Student.Name, Student.Branch, Result.SPI
FROM Student RIGHT OUTER JOIN Result
ON Student.RNO= Result.RNO ;
```

Student

RNO	Name	Branch
101	Raju	CE
102	Amit	CE
103	Sanjay	ME
104	Neha	EC
105	Meera	EE
106	Mahesh	ME

Result

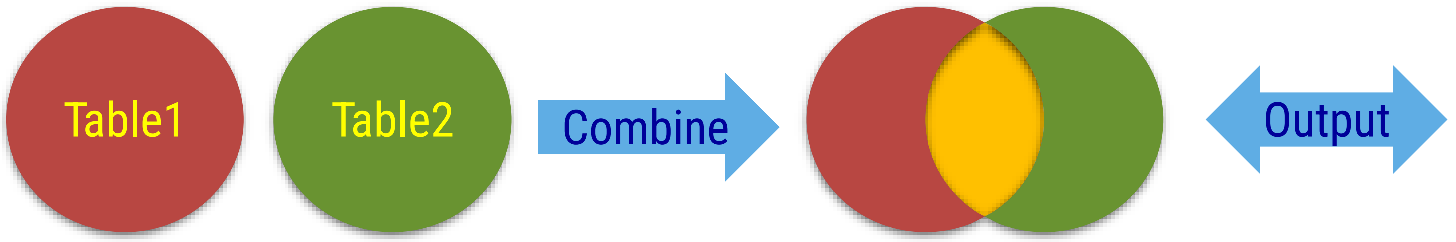
RNO	SPI
101	8.8
102	9.2
104	8.2
105	7
107	8.9

Output

RNO	Name	Branch	SPI
101	Raju	CE	8.8
102	Amit	CE	9.2
104	Neha	EC	8.2
105	Meera	EE	7
NULL	*NULL*	*NULL*	8.9

Full outer Join

- Full outer join return **all records** when there is a match in either left or right table.



Syntax

```
SELECT columns
FROM Table1 FULL OUTER JOIN Table2
ON Table1.column=Table2.column ;
```

Full outer Join(Cont..)

Example

```
SELECT Student.RNO, Student.Name, Student.Branch, Result.SPI
FROM Student FULL OUTER JOIN Result
ON Student.RNO= Result.RNO ;
```

Student

RNO	Name	Branch
101	Raju	CE
102	Amit	CE
103	Sanjay	ME
104	Neha	EC
105	Meera	EE
106	Mahesh	ME

Result

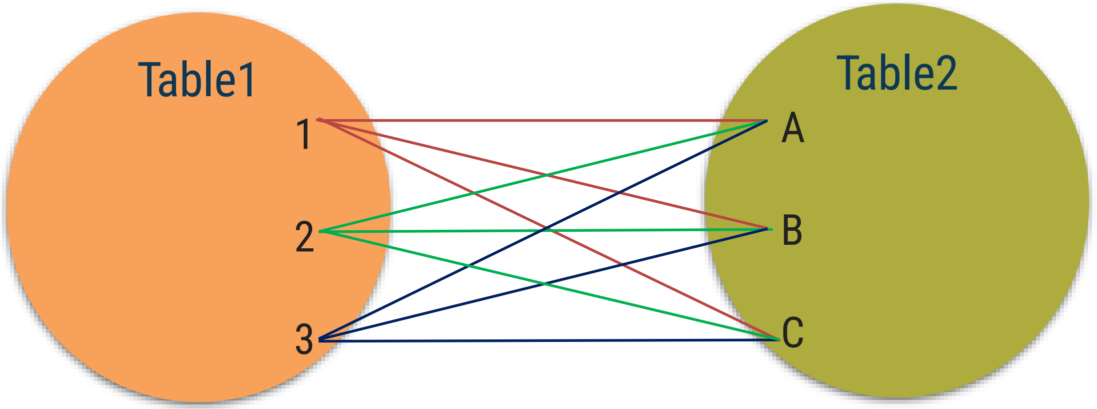
RNO	SPI
101	8.8
102	9.2
104	8.2
105	7
107	8.9

Output

RNO	Name	Branch	SPI
101	Raju	CE	8.8
102	Amit	CE	9.2
103	Sanjay	ME	*NULL*
104	Neha	EC	8.2
105	Meera	EE	7
106	Mahesh	ME	*NULL*
NULL	*NULL*	*NULL*	8.9

Cross Join

- Cross join produces **Cartesian product** of the tables that are involved in the join.
- The size of a Cartesian product is the **number of the rows in the first table** **multiplied by** the **number of rows in the second table** like this.



Syntax

```
SELECT *
FROM table1 , table2 ;
```

Cross Join(Cont..)

Student

RNO	Name	Branch
101	Raju	CE
102	Amit	CE
103	Sanjay	ME
104	Neha	EC
105	Meera	EE
106	Mahesh	ME

Result

RNO	SPI
101	8.8
102	9.2
104	8.2
105	7
107	8.9

Output

RNO	Name	Branch	SPI
101	Raju	CE	8.8
102	Amit	CE	8.8
103	Sanjay	ME	8.8
104	Neha	EC	8.8
105	Meera	EE	8.8
106	Mahesh	ME	8.8
101	Raju	CE	9.2
102	Amit	CE	9.2
103	Sanjay	ME	9.2
104	Neha	EC	9.2
105	Meera	EE	9.2
106	Mahesh	ME	9.2
101	Raju	CE	8.2
102	Amit	CE	8.2
103	Sanjay	ME	8.2
104	Neha	EC	8.2
105	Meera	EE	8.2
106	Mahesh	ME	8.2
101	Raju	CE	7
102	Amit	CE	7
103	Sanjay	ME	7
104	Neha	EC	7
105	Meera	EE	7
106	Mahesh	ME	7
101	Raju	CE	8.9
102	Amit	CE	8.9
103	Sanjay	ME	8.9
104	Neha	EC	8.9
105	Meera	EE	8.9
106	Mahesh	ME	8.9

Example

```
SELECT *
FROM Student , Result ;
```

Self Join

- A Self Join is a regular join, but the **table is joined with itself**.
- Self join is used to retrieve the records having similarity between records in the same table.
- Here, **we need to use aliases for the same table to set a self join between single table**.
- Self join would be of any type like inner self join, left self join, right self join etc.

Self Join(Cont..)

Syntax

```
SELECT  a.Column_Name,  b.Column_Name
FROM    Table1 a  Inner Join  Table1 b
ON      a.Column=b.Column  ;
```

Example

```
SELECT  e.Name as Employee,  m.Name as Manager
FROM    Employee e  Inner Join  Employee m
ON      e.MngrNo=m.EmpNo  ;
```

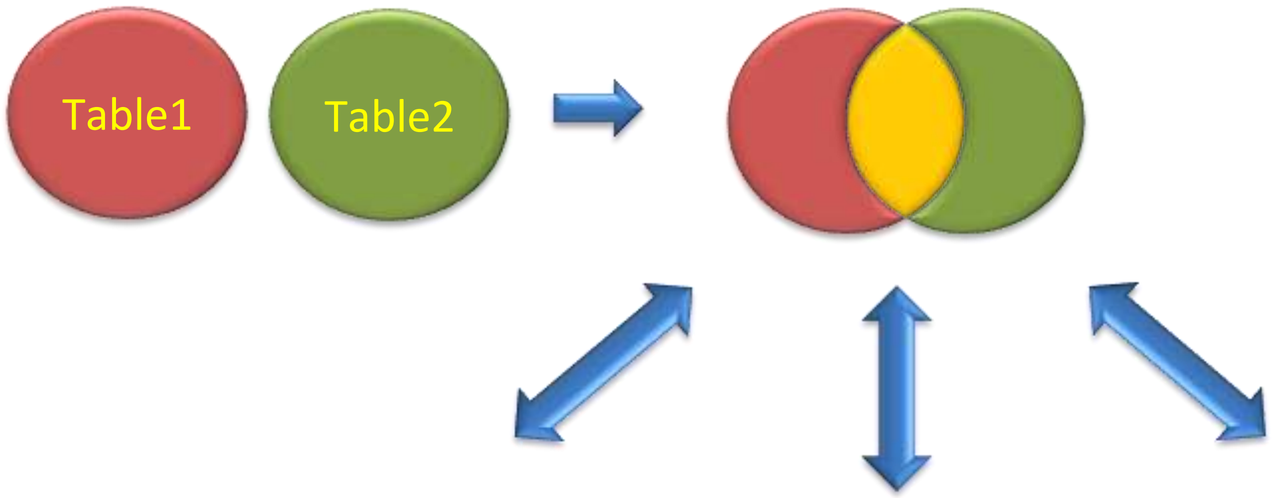
Employee

EmpNo	Name	MngrNo
E00001	Tarun	E00002
E00002	Rohan	E00005
E00003	Priya	E00005
E00004	Milan	NULL
E00005	Jay	NULL
E00006	Anjana	E00003

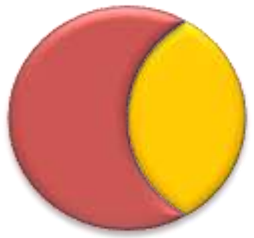
Employee

Employee	Manager
Tarun	Rohan
Rohan	Jay
Priya	Jay
Anjana	Priya

Join Examples

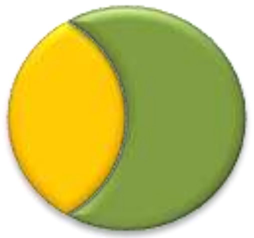


Left Outer Join



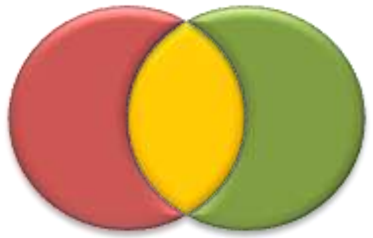
Select * From
T1 LEFT JOIN T2
ON T1.Id1=T2.Id2;

Right Outer Join



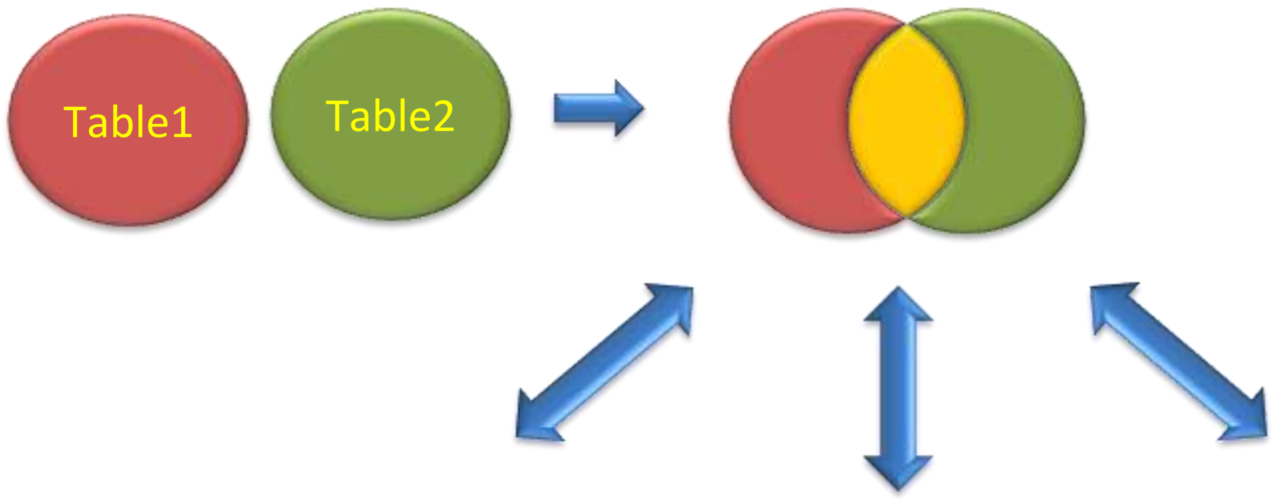
Select * From
T1 RIGHT JOIN T2
ON T1.Id1=T2.Id2;

Full Outer Join

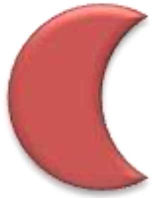


Select * From
T1 FULL JOIN T2
ON T1.Id1=T2.Id2;

Join Examples(Cont..)

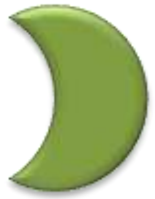


Unmatched Rows
From Left Table



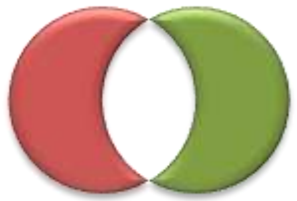
```
Select * From
T1 LEFT JOIN T2
ON T1.Id1=T2.Id2
Where T2.Id2 IS NULL
```

Unmatched Rows
From Right Table



```
Select * From
T1 RIGHT JOIN T2
ON T1.Id1=T2.Id2
Where T1.Id1 IS NULL
```

Unmatched Rows From Left
and Right Table



```
Select * From
T1 FULL JOIN T2
ON T1.Id1=T2.Id2
Where T1.Id1 IS NULL Or
T2.Id2 IS NULL
```

Join Examples(Cont..)

- Unmatched Rows From the Left Table

```
SELECT Employee.EID, Employee.Name, Employee.Dept, EmpSalary.Amount
FROM Employee Full Outer Join EmpSalary
ON Employee.EID= EmpSalary.EID
WHERE EmpSalary.EID IS NULL ;
```

Employee

EID	Name	Dept
101	Raju	Admin
102	Amit	Admin
103	Sanjay	HR
104	Neha	IT
105	Meera	Sales
106	Mahesh	HR

EmpSalary

EID	Amount
101	1000
102	5000
104	3000
105	8000
107	2500

Output

EID	Name	Dept	Amount
106	Mahesh	HR	*NULL*

Join Examples(Cont..)

- Unmatched Rows From the Right Table

```
SELECT Employee.EID, Employee.Name, Employee.Dept, EmpSalary.Amount
FROM Employee Full Outer Join EmpSalary
ON Employee.EID= EmpSalary.EID
WHERE Employee.EID IS NULL ;
```

Employee

EID	Name	Dept
101	Raju	Admin
102	Amit	Admin
103	Sanjay	HR
104	Neha	IT
105	Meera	Sales
106	Mahesh	HR

EmpSalary

EID	Amount
101	1000
102	5000
104	3000
105	8000
107	2500

Output

EID	Name	Dept	Amount
NULL	*NULL*	*NULL*	2500

Join Examples(Cont..)

- Unmatched Rows From the Left and Right Table

```
SELECT Employee.EID, Employee.Name, Employee.Dept, EmpSalary.Amount
FROM Employee Full Outer Join EmpSalary
ON Employee.EID= EmpSalary.EID
WHERE Employee.EID IS NULL OR EmpSalary.EID IS NULL ;
```

Employee

EID	Name	Dept
101	Raju	Admin
102	Amit	Admin
103	Sanjay	HR
104	Neha	IT
105	Meera	Sales
106	Mahesh	HR

EmpSalary

EID	Amount
101	1000
102	5000
104	3000
105	8000
107	2500

Output

EID	Name	Dept	Amount
106	Mahesh	HR	*NULL*
NULL	*NULL*	*NULL*	2500

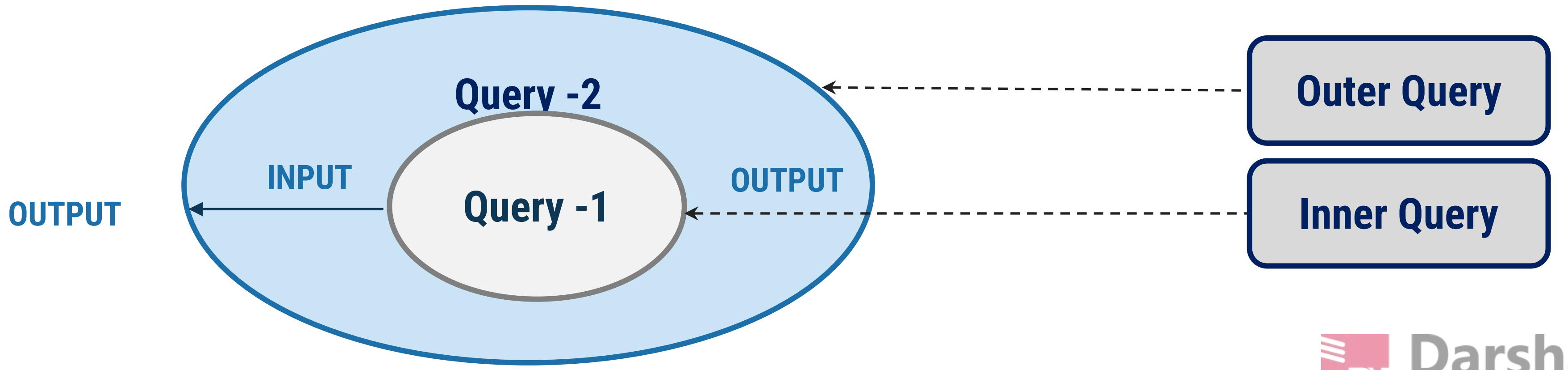
Darshan UNIVERSITY

Subquery

Section - 3

Sub Query

- We will use sub query when we want to **search some data** using select query but we **don't know the exact value from data.**
- For Example, if you want to find out the **name of the employee** whose **salary is maximum**.
  - Step 1: Find out maximum salary
  - Step 2: Then Search for the name of employee
- Query inside a query.
- Sub query is usually added in a where clause.



Types of Sub Query

1. Single Row Sub Query
2. Multiple Row Sub Query
3. Correlated Sub Query

1. Single Row Sub Query

- Returns **0** or **1** row.
- Can be used with **<,>,<=,>=** etc. operators
- **Example:** Display name of staff who has maximum salary.
  - First we have to find maximum salary from Faculty Table.
  - Then who has that maximum salary's value we will find out his/her name.

```
SELECT FNAME FROM Faculty WHERE SAL = ( SELECT MAX(SAL) FROM Faculty );
```

Faculty

FID	HID	FNAME	SUB	DID	SAL
11	12	PATEL	CP	7	10000
12	-	PANDYA	SM	6	30000
13	12	DOSHI	TOM	19	15000
14	12	MAKWANA	BE	9	18000
15	12	MEHTA	ACP	7	12000
16	12	SHAH	-	-	50000

Output

FNAME
SHAH

2. Multiple Row Sub Query

- Returns one or more rows
- Can be used with IN, NOT IN, ANY, ALL etc. operators
- **Example:** Display roll no., department id and spi of those students who are from RAJKOT.
  - First we will find out the roll no. of those students who are from RAJKOT
  - Here we will **get more than one records/rows** who are from RAJKOT. (That's why it is known as **Multiple Row Subquery**)
  - Next we will compare all those roll no. and find their department id and spi from Academic Table.

```
SELECT RNO,DID,SPI FROM ACADEMIC WHERE RNO IN ( SELECT RNO FROM STUDENT WHERE CITY='RAJKOT' );
```

To compare more than one value we can not use =

Academic

RNO	DID	SPI	CPI	BACKLOG
101	19	6.8	6.1	1
102	7	7.2	6.5	0
103	6	8.5	7.2	0
104	7	6.1	5.2	3
105	9	8.1	7.5	3

Student

RNO	SNAME	ADDRESS	CITY	MOBILE
101	MITESH	RING ROAD	RAJKOT	7845128956
102	KAUSHAL	SADHU VASVANI ROAD	RAJKOT	8989547412
103	ANKUR	GONDAL ROAD	RAJKOT	8866552241
104	KISHAN	SANADA ROAD	MORBI	9663322110
105	MUKESH	RAJKOT ROAD	JAMNAGAR	9425814789

Output

RNO	DID	SPI
101	19	6.8
102	7	7.2
103	6	8.5

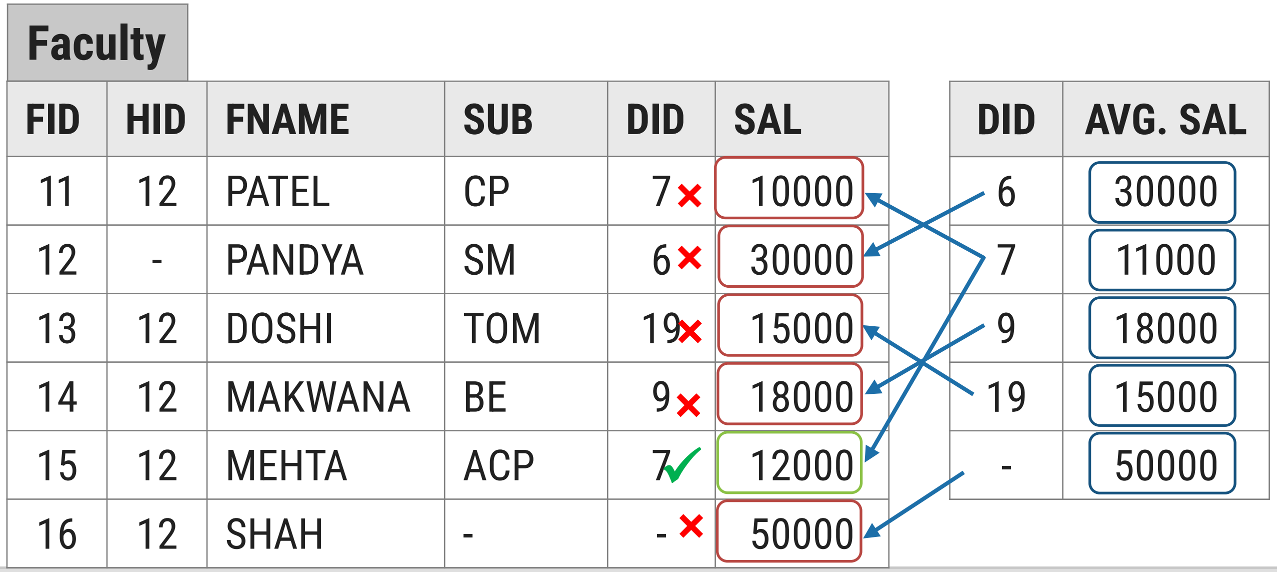
3. Correlated Sub Query

- Each subquery is executed once for every row of the outer query.
- Here inner query is executes more than one time where as in other subqueries inner query executes only one time.
- Here inner query is dependent on outer query.

3. Correlated Sub Query (Conti..)

- **Example:** Find out Name of person and his salary whose salary is greater than then their department's average salary.
  - First of all find average salary of each department.
  - Now compare each department's average salary with faculty's salary of same department only.
  - Display Name of staff and salary whose salary is greater than the average salary of department

```
SELECT FNAME,SAL FROM Faculty PARENT  WHERE SAL > ( SELECT AVG(SAL) FROM Faculty  WHERE PARENT.DID = DID  )
```



Faculty

FID	HID	FNAME	SUB	DID	SAL
11	12	PATEL	CP	7 ✗	10000
12	-	PANDYA	SM	6 ✗	30000
13	12	DOSHI	TOM	19 ✗	15000
14	12	MAKWANA	BE	9 ✗	18000
15	12	MEHTA	ACP	7 ✓	12000
16	12	SHAH	-	- ✗	50000

DID	AVG. SAL
6	30000
7	11000
9	18000
19	15000
-	50000

Faculty

DID	TOTAL SAL	TOTAL FAC.
6	30000	1
7	22000	2
9	18000	1
19	15000	1
-	50000	1

Keys

Section - 4

What is Constraints?

- SQL constraints are used to **specify rules** for data in a table.
- Constraints are used to **limit the type of data** that can go into a table. This **ensures** the **accuracy** and **reliability** of the data in the table.
- If there is any violation between the constraint and the data action, the action is aborted.
- Constraints can be column level or table level. Column level constraints apply to a column, and table level constraints apply to the whole table.
- Constraints can be specified when the table is created with the **CREATE TABLE** statement, or after the table is created with the **ALTER TABLE** statement.

Constraints used in SQL

- The following constraints are commonly used in SQL:
  - NOT NULL - Ensures that a column **cannot have a NULL value**
  - UNIQUE KEY- Ensures that **all values in a column are different/unique**
  - PRIMARY KEY - A **combination** of a **NOT NULL and UNIQUE**. Uniquely identifies each row in a table
  - FOREIGN KEY - **Prevents actions that would destroy links between tables**
  - CHECK - Ensures that the **values in a column satisfies a specific condition**
  - DEFAULT - Sets **a default value** for a column if no value is specified

NOT NULL Constraint

- By default, a column can hold NULL values.
- The NOT NULL constraint **enforces a column to NOT accept NULL values**.
- This enforces a field to always contain a value, which means that you cannot insert a new record, or update a record without adding a value to this field.
- Example: Create a table “Person” in which "ID", "LastName", and "FirstName" columns will NOT accept NULL values.

Example

```
CREATE TABLE Persons (
    ID int NOT NULL,
    LastName varchar(255) NOT NULL,
    FirstName varchar(255) NOT NULL,
    Age int );
```

- Example: Create a NOT NULL constraint on the "Age" column when the "Persons" table is already created.

Example

```
ALTER TABLE Persons
ALTER COLUMN Age int NOT NULL;
```

CHECK Constraint

- The CHECK constraint is used to **limit the value range** that can be placed in a column.
- If you define a CHECK constraint on a column it will allow only certain values for this column.
- Example: Create a table “Person” with CHECK constraint on the "Age" column. The CHECK constraint ensures that the age of a person must be 18, or older:

Example

```
CREATE TABLE Persons (
    ID int NOT NULL,
    LastName varchar(255),
    FirstName varchar(255) NOT NULL,
    Age int CHECK (Age>=18)
 );
```

- Example: Create a CHECK constraint on the "Age" column when the "Persons" table is already created.

Example

```
ALTER TABLE Persons
ADD CHECK (Age>=18);
```

Darshan UNIVERSITY

DEFAULT Constraint

- The DEFAULT constraint is used to **set a default value** for a column.
- The default value will be added to all new records, if no other value is specified.
- Example: Create a table “Person” with DEFAULT constraint on the "Age" column. The DEFAULT constraint will insert default age of a person as 18:

Example

```
CREATE TABLE Persons (
    ID int NOT NULL,
    LastName varchar(255),
    FirstName varchar(255) NOT NULL,
    Age int DEFAULT 18
 );
```

- Example: Create a DEFAULT constraint on the "Age" column when the "Persons" table is already created.

Example

```
ALTER TABLE Persons
ADD CONSTRAINT df_Age
DEFAULT 18 FOR Age;
```

What is Key?

- In SQL, the keys are the set of attributes used to identify a specific row from a table and to find or create the relation between two or more tables.

UNIQUE KEY

- The UNIQUE constraint **ensures that all values in a column are different/Unique**.
- Both the UNIQUE KEY and PRIMARY KEY constraints provide a guarantee for uniqueness for a column or set of columns.
- However, you can have many UNIQUE KEY per table, but only one PRIMARY KEY per table.
- Example: Create a table “Person” with UNIQUE KEY on the "ID" column:

Example

```
CREATE TABLE Persons (
    ID int NOT NULL UNIQUE,
    LastName varchar(255),
    FirstName varchar(255) NOT NULL,
    Age int
 );
```

Example

UNIQUE KEY on multiple columns

```
CREATE TABLE Persons (
    ID int NOT NULL,
    LastName varchar(255),
    FirstName varchar(255) NOT NULL,
    Age int,
    CONSTRAINT UC_Person UNIQUE (ID,FirstName)
 );
```

- Example: Create a UNIQUE KEY on the “ID" column when the "Persons" table is already created.

Example

```
ALTER TABLE Persons
ADD UNIQUE (ID);
```

Example

UNIQUE KEY on multiple columns

```
ALTER TABLE Persons
ADD CONSTRAINT UC_Person UNIQUE (ID,FirstName);
```

PRIMARY KEY

- The PRIMARY KEY constraint **uniquely identifies each record in a table**.
- Primary keys must contain **UNIQUE values**, and **cannot contain NULL values**.
- A table can have only **ONE primary key**; and in the table, this **primary key can consist of single or multiple columns** (fields).
- Example: Create a table “Person” with PRIMARY KEY on the "ID" column:

Example

```sql
CREATE TABLE Persons (
    ID int NOT NULL PRIMARY KEY,
    LastName varchar(255),
    FirstName varchar(255) NOT NULL,
    Age int
 );
```

Example — PRIMARY KEY on multiple columns

```sql
CREATE TABLE Persons (
    ID int NOT NULL,
    LastName varchar(255),
    FirstName varchar(255) NOT NULL,
    Age int,
    CONSTRAINT PK_Person PRIMARY KEY (ID,FirstName)
 );
```

- Example: Create a PRIMARY KEY on the “ID" column when the "Persons" table is already [illegible].

Example

```sql
ALTER TABLE Persons
ADD PRIMARY KEY (ID);
```

Example — PRIMARY KEY on multiple columns

```sql
ALTER TABLE Persons
ADD CONSTRAINT PK_Person PRIMARY KEY (ID,FirstName);
```

FOREIGN KEY

- The FOREIGN KEY constraint is used to **prevent actions that would destroy links between tables**.
- A FOREIGN KEY is a **field (or collection of fields) in one table, that refers to the PRIMARY KEY in another table**.
- The table with the foreign key is called the child table, and the table with the primary key is called the referenced or parent table.
- Example: Create a table “Order” with FOREIGN KEY on the “Person ID" column:

Example

```
CREATE TABLE Orders (
    OrderID int NOT NULL PRIMARY KEY,
    OrderNo int NOT NULL,
    PersonID int FOREIGN KEY REFERENCES Persons(PersonID)
);
```

- Example: Create a FOREIGN KEY on the “ID" column when the “Order" table is already created.

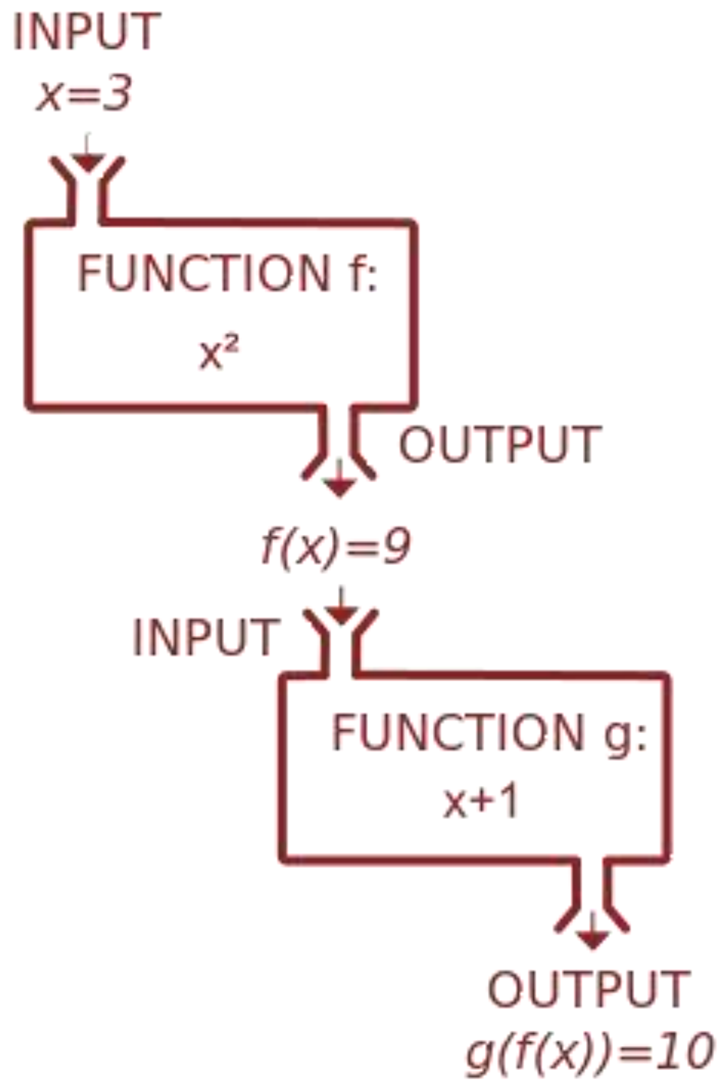
Example

```
ALTER TABLE Orders
ADD FOREIGN KEY (PersonID) REFERENCES Persons(PersonID);
```

System Functions

Section - 5

Introduction : Function



What is Function?

- A function is simply **a "chunk" of code that you can use over and over again, rather than writing it out multiple times**.
- Functions enable programmers to break down or decompose a problem into smaller chunks, each of which performs a particular task.
- The function contains instructions used to create the output from its input.
- A function is a block of organized code that is used to perform a single task.

Introduction : Function

- A function is a database object in SQL Server.
- Basically, it is a set of SQL statements that accept only input parameters, perform actions and returns the result.
- The function can return only a **single value or a table**.
- We can’t use a function to Insert, Update, Delete records in the database table(s).

Types of Functions

- SQL Server Functions are of two types:
  1. **System Functions**
  2. **User Defined Functions (UDFs)**

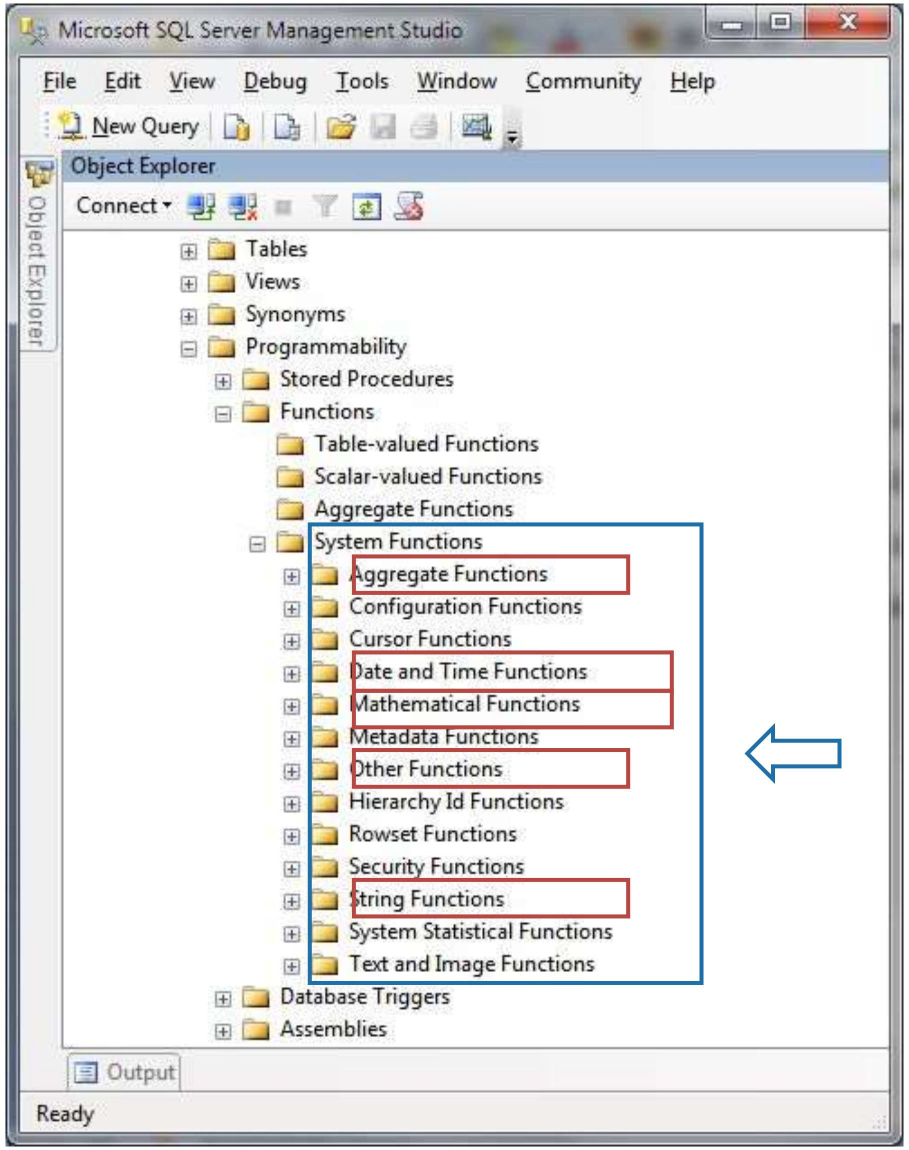
1. **System Functions:**
   - Built-in or System functions are available with every database.
   - Some common types are Aggregate functions, Analytic functions, Ranking functions, Rowset functions, Scalar functions.

2. **User Defined Functions (UDFs):**
   - **Functions created by the database user** are called user-defined functions.
   - UDFs are of two types:
     1. **Scalar functions**: The function that **returns a single data value** is called a scalar function.
     2. **Table-valued functions**: The function that **returns multiple records as a table** data type is called a Table-valued function. It can be a result set of a single select statement.

1. System Functions

- Built-in or System functions are available with every database, we can use it as per our requirement.
- Here, we explore most widely used system functions.

Sr.	System Functions
1	Aggregate Functions
2	Date and Time Functions
3	Mathematical Functions
4	String Functions
5	Other Functions



1. Aggregate Functions

- An aggregate function in SQL **performs a calculation on multiple values and returns a single scalar value**.
- SQL provides many aggregate functions that include avg(), count(), sum(), min(), max(), etc.
- An aggregate function **ignores NULL values when it performs the calculation**, except for the count function.
- We often use aggregate functions with the GROUP BY and HAVING clauses of the SELECT statement.

Syntax : Aggregate Functions

```
aggregate_function ( DISTINCT |  ALL expression)
```

1. Specify the name of function that you want to use such as AVG(), SUM(), MAX() etc.
2. Use DISTINCT if you want only distinct values are considered in the calculation or ALL if all values are considered in the calculation. By default, ALL is used if you don't specify.
3. The expression can be a column of a table or an expression that consists of multiple columns with arithmetic operators.

1. Aggregate Functions (Cont..)

- The following table shows the SQL Server aggregate functions:

Sr.	Aggregate function	Description
1	AVG()	The AVG() aggregate function calculates the average of non-NULL values in a set.
2	COUNT()	The COUNT() aggregate function returns the number of rows in a group, including rows with NULL values.
3	MAX()	The MAX() aggregate function returns the highest value (maximum) in a set of non-NULL values.
4	MIN()	The MIN() aggregate function returns the lowest value (minimum) in a set of non-NULL values.
5	SUM()	The SUM() aggregate function returns the summation of all non-NULL values a set.

1. Aggregate Functions Example

System Functions

Student

Rno	Name	Branch	Semester	CPI
101	Ramesh	CE	3	9
102	Mahesh	EC	3	8
103	Suresh	ME	4	7
104	Amit	EE	4	8
105	Anita	CE	4	8
106	Reeta	ME	3	7
107	Rohit	EE	4	9
108	Chetan	CE	3	8
109	Rakesh	CE	4	9

Example Find out sum of CPI of all students.

Answer

```
Select SUM(CPI) AS [Sum] From Student
```

Output

Sum
73.00

Example Find out maximum & minimum CPI.

Answer

```
Select MAX(CPI) AS [Max], MIN(CPI) AS [Min]
From Student
```

Output

Max	Min
9.00	7.00

Example Count the number of students.

Answer

```
Select COUNT(RNo) AS [Total] From Student
```

Output

Total
9

Example Find out average of CPI of all students.

Answer

```
Select AVG(CPI) AS [Avg] From Student
```

Output

Avg
8.111111

Darshan UNIVERSITY

1. Aggregate Functions with Group By Example

System Functions

Student

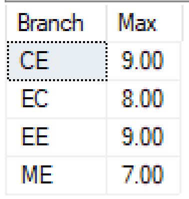
Rno	Name	Branch	Semester	CPI
101	Ramesh	CE	3	9
102	Mahesh	EC	3	8
103	Suresh	ME	4	7
104	Amit	EE	4	8
105	Anita	CE	4	8
106	Reeta	ME	3	7
107	Rohit	EE	4	9
108	Chetan	CE	3	8
109	Rakesh	CE	4	9

Example Find out Branch wise Maximum CPI.

Answer

```
Select Branch, MAX(CPI) AS [Max] From Student Group By Branch
```

Output



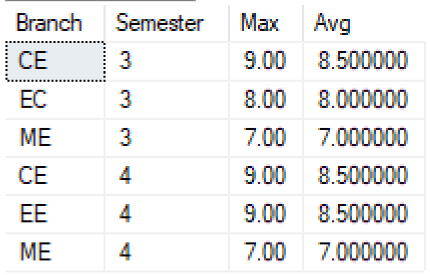
Branch	Max
CE	9.00
EC	8.00
EE	9.00
ME	7.00

Example Find out Branch wise Semester wise Minimum & Average CPI.

Answer

```
Select Branch, Semester, MAX(CPI) AS [Max], MIN(CPI) AS
[Min] From Student
Group By Branch, Semester
```

Output



Branch	Semester	Max	Avg
CE	3	9.00	8.500000
EC	3	8.00	8.000000
ME	3	7.00	7.000000
CE	4	9.00	8.500000
EE	4	9.00	8.500000
ME	4	7.00	7.000000

1. Aggregate Functions Group By with Filter Example (Cont..)

System Functions

Student

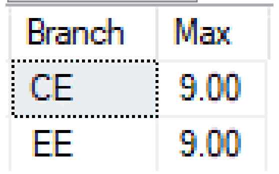
Rno	Name	Branch	Semester	CPI
101	Ramesh	CE	3	9
102	Mahesh	EC	3	8
103	Suresh	ME	4	7
104	Amit	EE	4	8
105	Anita	CE	4	8
106	Reeta	ME	3	7
107	Rohit	EE	4	9
108	Chetan	CE	3	8
109	Rakesh	CE	4	9

Example Find out All the Branches with maximum CPI, whose maximum CPI is more than 8.

Answer

```
Select Branch, MAX(CPI) AS [Max] From Student
Group By Branch
Having MAX(CPI) > 8
```

Output



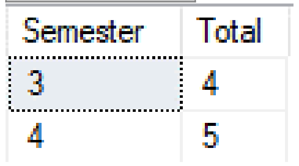
Branch	Max
CE	9.00
EE	9.00

Example Find out semester wise total students & arrange them in order with their count.

Answer

```
Select Semester, Count(Rno) AS [Total] From Student
Group By Semester
Order By Total
```

Output



Semester	Total
3	4
4	5

1. Aggregate Functions Group By with Filter Example (Cont..)

System Functions

Student

Rno	Name	Branch	Semester	CPI
101	Ramesh	CE	3	9
102	Mahesh	EC	3	8
103	Suresh	ME	4	7
104	Amit	EE	4	8
105	Anita	CE	4	8
106	Reeta	ME	3	7
107	Rohit	EE	4	9
108	Chetan	CE	3	8
109	Rakesh	CE	4	9

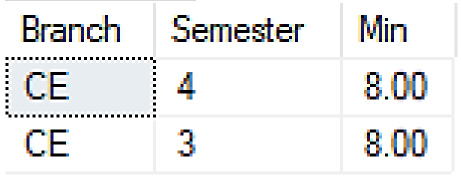
Example

Find out Branch wise & Semester wise minimum CPI details of CE branch's students in which minimum CPI is greater than 7. Do arrange the result in descending order to semester.

Answer

```
Select Branch, Semester, MIN(CPI) AS [Min] From Student
Where Branch='CE'
Group By Branch, Semester
Having MIN(CPI) > 7
Order By Semester Desc
```

Output



Branch	Semester	Min
CE	4	8.00
CE	3	8.00

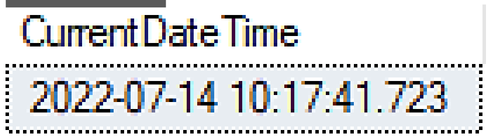
2. Date & Time Functions

- **SQL Server** comes with the following data types for storing a date or a date/time value in the database:
  - DATE– format YYYY-MM-DD
  - DATETIME – format YYYY-MM-DD HH:MI:SS
  - SMALLDATETIME – format: YYYY-MM-DD HH:MI:SS
  - TIMESTAMP – format: a unique identifier
- To retrieve current date time, we can use GETDATE():

Current Date & Time

```sql
Select GETDATE() AS CurrentDateTime
```

Output



```
CurrentDateTime
2022-07-14 10:17:41.723
```

2. Date & Time Functions (Cont..)

List of SQL DATE Functions

Date Functions	Description	Return Value Data Type
DAY (date or datetime)	Returns the day of the week for a given date	Integer like 1 - 31
MONTH (date or datetime)	Returns the month of a given date	Integer like 1 - 12
YEAR (date or datetime)	Returns the year of a given date	Integer for year like 2021
DATEPART (date part, date or datetime)	Returns the date part specified in int format	Integer like 1 – 12 for month, 1 – 31 for day, or year like 2021
DATENAME (date part, date or datetime)	Returns the date part specified in character format	Character like April, May, ‘1’, ‘2’, ‘31’, ‘2020’, ‘2021’
EOMONTH (date [,months to add)	Returns the last do of the month with an optional parameter to add months (+ or -).	Returns end date of specified date
DATEADD (date part, units, date or datetime)	Return date math results	datetime
DATEDIFF (date part, start date, end date)	Give the difference between 2 dates in units specified by date part	Integer of date part units
ISDATE (potential date string)	Use to validate a date string	Returns 1 if the string is a valid date or 0 if not a valid date.

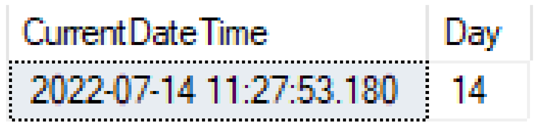
2. Date & Time Functions | DAY() (Cont..)

DAY() : The date function DAY **accepts a date, datetime, or valid date string** and returns the **Day part as an integer value**.

Example - 1

```sql
SELECT GETDATE() AS CurrentDateTime, DAY(GETDATE()) AS [Day]
```

Output

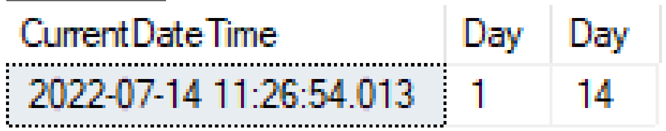


```
CurrentDateTime           Day
2022-07-14 11:27:53.180   14
```

Example - 2

```sql
SELECT GETDATE() AS CurrentDateTime, DAY('20220101') AS [Day],
DAY('2022-07-14 15:46:19.277') AS [Day]
```

Output

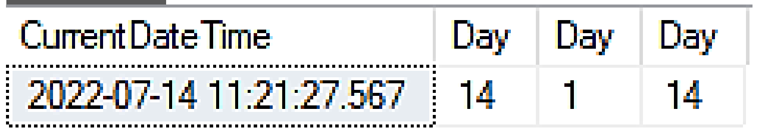


```
CurrentDateTime           Day   Day
2022-07-14 11:26:54.013   1     14
```

Example - 3

```sql
SELECT GETDATE() AS CurrentDateTime, DAY(GETDATE()) AS [Day] ,
DAY('20220101') AS [Day], DAY('2022-07-14 15:46:19.277') AS
[Day]
```

Output



```
CurrentDateTime           Day   Day   Day
2022-07-14 11:21:27.567   14    1     14
```

2. Date & Time Functions | MONTH () (Cont..)

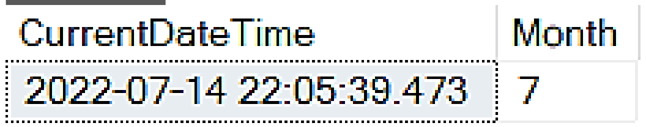
System Functions

MONTH() : The date function MONTH **accepts a date, datetime, or valid date string** and returns the **Month part as an integer value**.

Example - 1

```
SELECT GETDATE() AS CurrentDateTime, MONTH(GETDATE()) AS [Month]
```

Output

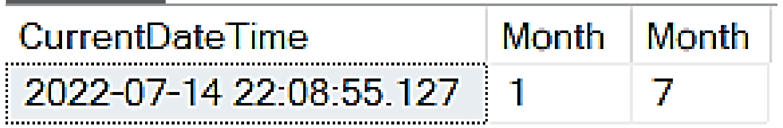


```
CurrentDateTime          Month
2022-07-14 22:05:39.473  7
```

Example - 2

```
SELECT GETDATE() AS CurrentDateTime, MONTH('20220101') AS
[Month], MONTH('2022-07-14 15:46:19.277') AS [Month]
```

Output

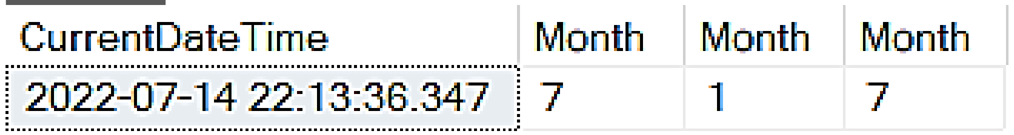


```
CurrentDateTime          Month  Month
2022-07-14 22:08:55.127  1      7
```

Example - 3

```
SELECT GETDATE() AS CurrentDateTime, MONTH(GETDATE()) AS [Month]
, MONTH('20220101') AS [Month], MONTH('2022-07-14 15:46:19.277')
AS [Month]
```

Output



```
CurrentDateTime          Month  Month  Month
2022-07-14 22:13:36.347  7      1      7
```

2. Date & Time Functions | YEAR () (Cont..)

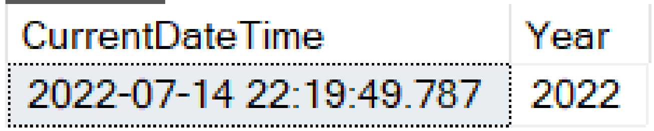
System Functions

YEAR() : The date function YEAR **accepts a date, datetime, or valid date string** and returns the **Year part as an integer value**.

Example - 1

```
SELECT GETDATE() AS CurrentDateTime, YEAR(GETDATE()) AS [Year]
```

Output

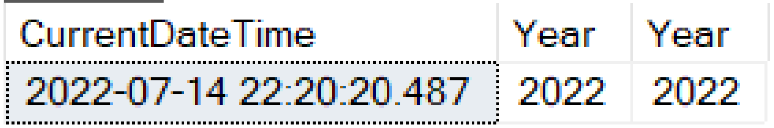


```
CurrentDateTime          Year
2022-07-14 22:19:49.787  2022
```

Example - 2

```
SELECT GETDATE() AS CurrentDateTime, YEAR('20220101') AS [Year],
YEAR('2022-07-14 15:46:19.277') AS [Year]
```

Output

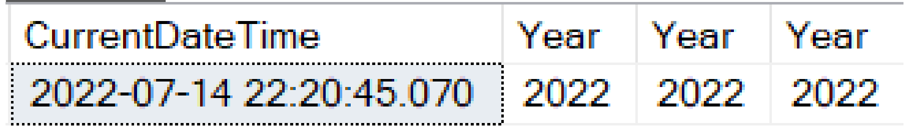


```
CurrentDateTime          Year  Year
2022-07-14 22:20:20.487  2022  2022
```

Example - 3

```
SELECT GETDATE() AS CurrentDateTime, YEAR(GETDATE()) AS [Year] ,
YEAR('20220101') AS [Year], YEAR('2022-07-14 15:46:19.277') AS
[Year]
```

Output



```
CurrentDateTime          Year  Year  Year
2022-07-14 22:20:45.070  2022  2022  2022
```

2. Date & Time Functions | DATEPART () (Cont..)

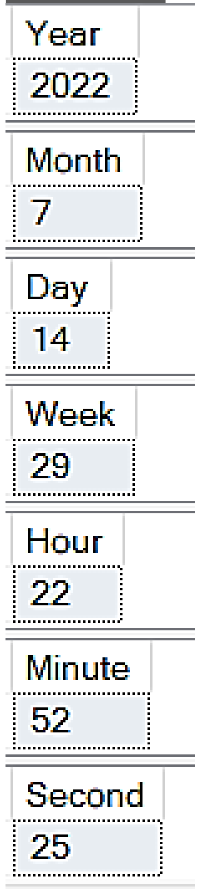
System Functions

DATEPART() : It returns an integer corresponding to the datepart specified in DATEPART function.

Example

```
SELECT DATEPART(YEAR, GETDATE())   AS 'Year';
SELECT DATEPART(MONTH, GETDATE())  AS 'Month';
SELECT DATEPART(DAY, GETDATE())    AS 'Day';
SELECT DATEPART(WEEK, GETDATE())   AS 'Week';
SELECT DATEPART(HOUR, GETDATE())   AS 'Hour';
SELECT DATEPART(MINUTE, GETDATE()) AS 'Minute';
SELECT DATEPART(SECOND, GETDATE()) AS 'Second';
```

Output



```
Year
2022

Month
7

Day
14

Week
29

Hour
22

Minute
52

Second
25
```

2. Date & Time Functions | DATENAME () (Cont..)

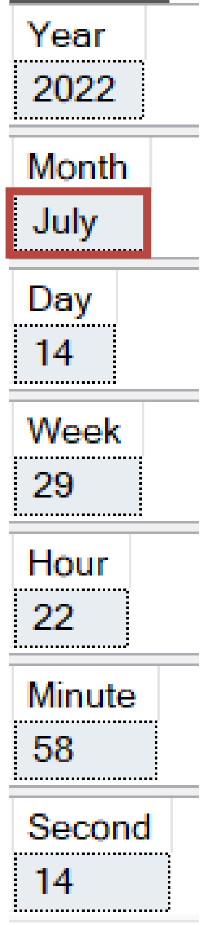
System Functions

DATENAME() : It returns a string corresponding to the datepart specified for the given date

Example

```
SELECT DATENAME(YEAR, GETDATE())   AS 'Year';
SELECT DATENAME(MONTH, GETDATE())  AS 'Month';
SELECT DATENAME(DAY, GETDATE())    AS 'Day';
SELECT DATENAME(WEEK, GETDATE())   AS 'Week';
SELECT DATENAME(HOUR, GETDATE())   AS 'Hour';
SELECT DATENAME(MINUTE, GETDATE()) AS 'Minute';
SELECT DATENAME(SECOND, GETDATE()) AS 'Second';
```

Output



```
Year
2022

Month
July

Day
14

Week
29

Hour
22

Minute
58

Second
14
```

2. Date & Time Functions | EOMONTH () (Cont..)

System Functions

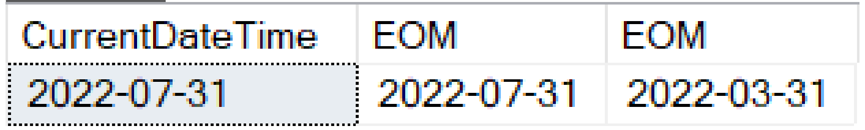
EOMONTH() :

- ✓ The date function EOMONTH **accepts a date, datetime, or valid date string** and returns the end of month date as a datetime.
- ✓ It can also take an optional offset that basically adds or subtracts months from the current passed date.

Example - 1

```sql
SELECT EOMONTH(GETDATE()) AS CurrentDateTime, EOMONTH('20220701') AS EOM,
EOMONTH('March 1, 2022') AS EOM
```

Output

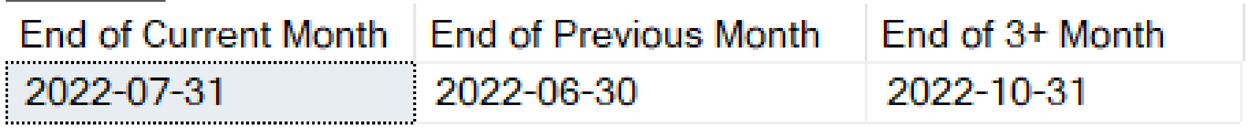


CurrentDateTime	EOM	EOM
2022-07-31	2022-07-31	2022-03-31

Example - 2

```sql
SELECT
EOMONTH(GETDATE()) as 'End Of Current Month',
EOMONTH(GETDATE(),-1) as 'End Of Previous Month',
EOMONTH(GETDATE(),3) as 'End Of 6+ Month';
```

Output



End of Current Month	End of Previous Month	End of 3+ Month
2022-07-31	2022-06-30	2022-10-31

2. Date & Time Functions | DATEADD () (Cont..)

System Functions

DATEADD() : It returns datepart with added interval as a datetime.

Example

Datepart	Query	Output
DateGroup : **Day**		
d	`SELECT DATEADD(d, 1, '2022-07-14 15:15:20') AS ADDEDDATE`	2022-07-15 15:15:20.000
dd	`SELECT DATEADD(dd, 1, '2022-07-14 15:15:20') AS ADDEDDATE`	2022-07-15 15:15:20.000
day	`SELECT DATEADD(day, 1, '2022-07-14 15:15:20') AS ADDEDDATE`	2022-07-15 15:15:20.000
DateGroup : **Month**		
m	`SELECT DATEADD(m, 1, '2022-07-14 15:15:20') AS ADDEDMONTH`	2022-08-14 15:15:20.000
mm	`SELECT DATEADD(mm, 1, '2022-07-14 15:15:20') AS ADDEDMONTH`	2022-08-14 15:15:20.000
month	`SELECT DATEADD(month, 1, '2022-07-14 15:15:20') AS ADDEDMONTH`	2022-08-14 15:15:20.000
DateGroup : **Year**		
yy	`SELECT DATEADD(yy, 1, '2022-07-14 15:15:20') AS ADDEDYEAR`	2023-07-14 15:15:20.000
yyyy	`SELECT DATEADD(yyyy, 1, '2022-07-14 15:15:20') AS ADDEDYEAR`	2023-07-14 15:15:20.000
year	`SELECT DATEADD(year, 1, '2022-07-14 15:15:20') AS ADDEDYEAR`	2023-07-14 15:15:20.000

2. Date & Time Functions | DATEDIFF () (Cont..)

DATEDIFF() :

✓ It gets the difference between two dates with the results returned in date units specified as years, months days, minutes, seconds as an integer value.

Example

DiffPart	Query	Output
Minutes	`SELECT DATEDIFF(MINUTE, '2022-07-13', '2022-07-14')`	1440
Hours	`SELECT DATEDIFF(HOUR, '2022-07-13', '2022-07-14')`	24
Days	`SELECT DATEDIFF(DAY, '2022-07-01', '2022-07-14')`	13
Months	`SELECT DATEDIFF(MONTH, '2022-07-01', '2022-08-14')`	1
Years	`SELECT DATEDIFF(YEAR, '2022-07-01', '2025-08-14')`	3

2. Date & Time Functions | ISDATE () (Cont..)

System Functions

ISDATE() :

- ✓ To check a string to see if it is a valid Date or Datetime field.
- ✓ ISDATE return 1 if true or 0 if false.

Example

Query	Output
`SELECT ISDATE('20220101') as 'Valid';`	1
`SELECT ISDATE('01/01/22') as 'Valid';`	1
`SELECT ISDATE('13/01/2022') as 'Not Valid';`	0
`SELECT ISDATE('2022') as 'Valid';`	1
`SELECT ISDATE('2022-13-01') as 'Not Valid';`	0

3. Mathematical Functions

System Functions

- SQL Server Math/Numeric Functions

Example

Function Name	Description	Output
`ABS()`	Returns the absolute value of a number `SELECT ABS(20), ABS(-50)`	20, 50
`CEILING()`	Returns the smallest integer value that is >= a number `SELECT CEILING(-13.5), CEILING(25)`	-13, 25
`FLOOR()`	Returns the largest integer value that is <= to a number `SELECT FLOOR(-13.5), FLOOR(25.75)`	-14, 25
`PI()`	Returns the value of PI `SELECT PI()`	3.14159265358979
`POWER()`	Returns the value of a number raised to the power of another number `SELECT POWER(4, 2)`	16

3. Mathematical Functions (Cont..)

System Functions

- SQL Server Math/Numeric Functions

Example

Function Name	Description	Output
ROUND()	Rounds a number to a specified number of decimal places `SELECT ROUND(235.415, 0),ROUND(235.415, 1),ROUND(235.415, 2)`	235.000 235.400 235.420
SIGN()	Returns the sign of a number (If number > 0, it returns 1, If number = 0, it returns 0, If number < 0, it returns -1) `SELECT SIGN(-12), SIGN(12), SIGN(0)`	-1 1 0
SQRT()	Returns the square root of a number `SELECT SQRT(64)`	8
SQUARE()	Returns the square of a number `SELECT SQUARE(8)`	64

4. String Functions

String Functions

Function Name	Description	Output
ASCII()	➢ The ASCII() function accepts a character expression and returns the ASCII code value of the leftmost character of the character expression. `SELECT ASCII('A'), ASCII('a’)`	65, 97
CONCAT()	➢ To join two or more strings into one, you use the CONCAT() function, The CONCAT() takes two up to 255 input strings and joins them into one. ➢ It requires at least two input strings. If you pass one input string, the CONCAT() function will raise an error. ➢ If you pass non-character string values, the CONCAT() function will implicitly convert those values into strings before concatenating. ➢ The CONCAT() function also converts NULL into an empty string with the type VARCHAR(1). `SELECT CONCAT('Darshan',' ','University')`	Darshan University
CONCAT_WS()	➢ CONCAT_WS() is very similar to CONCAT() function, but it allows the user to specify a separator between the concatenated input strings. ➢ It can be used to generate comma-separated values. `SELECT CONCAT_WS(',','Darshan','University')`	Darshan,Univ ersity

4. String Functions (Cont..)

String Functions

Function Name	Description	Output
CHARINDEX()	➢ CHARINDEX() is a scalar SQL string function used to return the index of a specific string expression within a given string. ➢ CHARINDEX() has 2 required parameters which are the input string and character and one optional parameter which is the starting index of the search operation (If this argument is not specified or is less or equal than zero (0) value, the search starts at the beginning of input string). `SELECT CHARINDEX('t', 'Customer’), CHARINDEX('World','Hello World')`	4, 7
LEFT(), RIGHT()	➢ LEFT() and RIGHT() functions are one of the most popular SQL string functions. ➢ They are used to extract a specific number of characters from the left-side or right-side of a string. `SELECT LEFT('Darshan University',5) , RIGHT('Darshan University',5)`	Darsh, rsity

4. String Functions (Cont..)

System Functions

String Functions

Function Name	Description	Output
LOWER(), UPPER()	➢ LOWER() is used to change the letter case to a lower case. ➢ UPPER() is used to change the case of the letters into upper case. `Select LOWER('Darshan'), UPPER('Darshan')`	darshan DARSHAN
LTRIM(), RTRIM()	➢ LTRIM() and RTRIM() function are used to remove additional spaces from the left side or right side of an input string. `SELECT RTRIM('Darshan   ') + LTRIM('    University')`	DarshanUniversity
STRING_SPLIT ()	➢ It is a table-valued function that splits a string into a table that consists of rows of substrings based on a specified separator. ➢ **Syntax:** STRING_SPLIT (input_string , separator) ➢ It returns a single-column table, whose column name is **value.** `SELECT value FROM STRING_SPLIT('red,green,,blue', ',')`	

4. String Functions (Cont..)

String Functions

Function Name	Description	Output

REPLACE()	➢ To replace all occurrences of a substring within a string with a new substring.
	Syntax: REPLACE(input_string, substring, new_substring);
	`SELECT REPLACE('Darshan Institute','Institute','University')`
		Darshan University

REPLICATE()	➢ It repeats a string a specified number of times. `SELECT REPLICATE('Darshan',2)`	DarshanDarshan
REVERSE()	➢ It accepts a string argument and returns the reverse order of that string. `SELECT REVERSE('Darshan')`	nahsraD
SPACE()	➢ It returns a string of repeated spaces. `SELECT  SPACE(5)+ 'Darshan'`	Darshan

4. String Functions (Cont..)

System Functions

String Functions

Function Name	Description	Output
SUBSTRING()	➢ It extracts a substring with a specified length starting from a location in an input string. ➢ **Syntax:** SUBSTRING(input_string, **Start**, **Length**); ➢ **Start** is an integer that specifies the location where the returned substring starts. Note that the first character in the input_string is 1, not zero. ➢ **Length** is a positive integer that specifies the number of characters of the substring to be returned. ➢ The SUBSTRING() function raises an error if the length is negative. ➢ If start + length > the length of input_string, the substring will begin at the start and include the remaining characters of the input_string. `SELECT SUBSTRING('SQL Server SUBSTRING', 5, 6)`	Server
LEN()	➢ The LEN function is used to provide the number of characters in a string without including trailing spaces. `Select LEN('Darshan University')`	18

5. Other Functions

System Functions

Other Functions

Function Name	Description	Output
CAST()	➢ It converts a value (of any type) into a specified datatype. ➢ **Syntax:** CAST(*expression* AS *datatype(length)*) `SELECT CAST(25.65 AS varchar)`	25.65

CONVERT()	➢ It converts a value (of any type) into a specified datatype.
	➢ **Syntax:** CONVERT(*data_type(length), expression, style*)
	`SELECT CONVERT(varchar,25.65)`
	➢ Style is optional, The format used to convert between data types, such as a date or string format.
	`SELECT CONVERT(varchar(20), getdate(), 100)`
		25.65
		--------------- 	
		Jul 14 2022 10:59AM

ISNULL()	➢ It replaces NULL with a specified value.
	➢ If any expression or column value is null, then which value you want to put there?
	➢ **Syntax:** ISNULL(expression, replacement)
	`SELECT ISNULL(NULL,20)`
	`SELECT ISNULL('Hello', 'Hi')`
		20
		Hello

5. Other Functions

System Functions

Other Functions

Function Name	Description	Output

COALESCE()	➢ Coalesce function are **used to handle NULL values**.
	➢ It evaluates the arguments in order and **always returns *first non-null* value** from the defined argument list.
	Properties
	• Expressions must be of same data-type
	• It can contain multiple expressions
	• It is a syntactic shortcut for the Case expression
	• Always evaluates for an integer first, an integer followed by character expression yields integer as an output.
	`SELECT COALESCE (NULL,'A','B')`
	`SELECT COALESCE (NULL,100,20,30,40)`
	`SELECT COALESCE (NULL,NULL,20,NULL,NULL)`
	`SELECT COALESCE (NULL,NULL,NULL,NULL,NULL,'Darshan')`
	`SELECT COALESCE (NULL,NULL,NULL,NULL,1,'Darshan')`
	`SELECT COALESCE (NULL,NULL,NULL,NULL,NULL,'Darshan',1)`
		A
		100
		20
		Darshan
		1
		Error

User Defined Functions (UDF)

Section - 6

User Defined Functions

- UDF is a programming construct that accepts parameters, does actions and returns the result of that action.
- The result either is a scalar value or result set.
- UDFs can be used in scripts, Stored Procedures, triggers and other UDFs within a database.

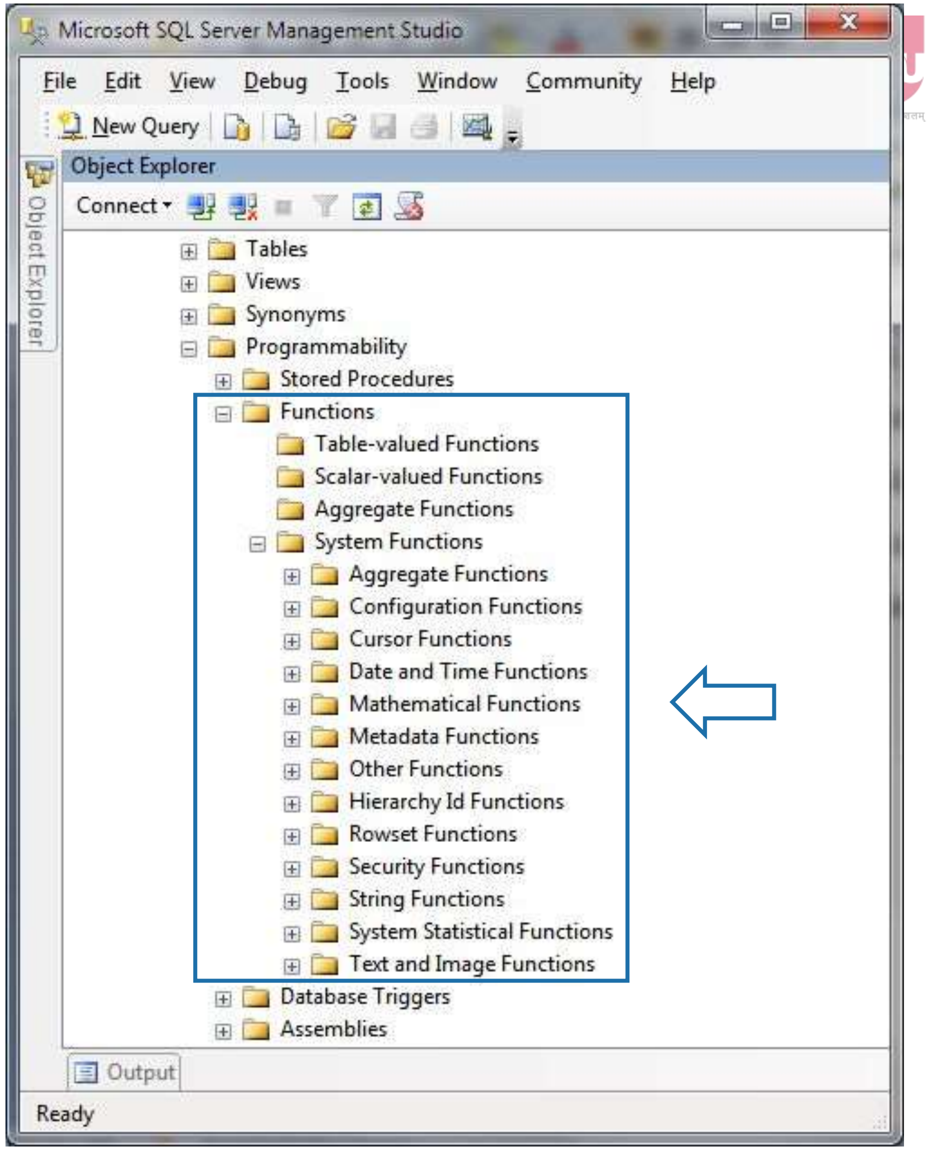
- **Benefits**
  - UDFs support modular programming.
  - Once you create a UDF and store it in a database then you can call it any number of times.
  - You can modify the UDF independent of the source code.

How to create function?

Syntax : User-defined Function

```
CREATE OR ALTER FUNCTION function_name
(
        @parameter1 datatype,
        @parameter2 datatype,
        .,
        .,
        .,
        @parametern datatype
)
RETURNS return_datatype
AS
BEGIN
   [declaration_section]
   [executable_section]
   RETURN return_value
END;
```

You can find all functions, go to your Database → Programmability → Functions in SQL Server



Important note for UDF !

- ✓ Function **must have a name** and a function name can **never start** with a **special character such as @, $, #, and so on**.
- ✓ Functions **compile every time**.
- ✓ Functions **must return a value or result**.
- ✓ Functions **only work with input parameters**.
- ✓ Function is **not used to Insert, Update, Delete data in a database table(s)**.
- ✓ User Defined Function **can't return XML Data Type**.
- ✓ User Defined Function **doesn't support exception handling**, try and catch statements are not used in functions.

Example : UDF

Scalar Valued Function

```
--Scalar Valued function (always returns single value)
Create Function fun_AddNumber(@num1 int,@num2 int)
returns int
as
begin
    return @num1+@num2
end
```

```
--To execute the function
Select dbo.fun_AddNumber (2,3)
```

```
--Answer
5
```

Example : UDF

Concat Two Strings

```sql
Create Function fun_JoinPersonInfo
(
   @FirstName nvarchar(150),
   @LastName nvarchar(500)
)
returns nvarchar(500)
as
begin return(select @FirstName+ ' ' +@LastName)
end
```

```sql
--To execute the function
Select FirstName,LastName,dbo.fun_JoinPersonInfo (FirstName,LastName) AS Merged From Person
```

Output

FirstName	LastName	Merged
Rahul	Anshu	Rahul Anshu
Hardik	Hinsu	Hardik Hinsu
Bhavin	Kamani	Bhavin Kamani
Bhoomi	Patel	Bhoomi Patel
Rohit	Rajgor	Rohit Rajgor

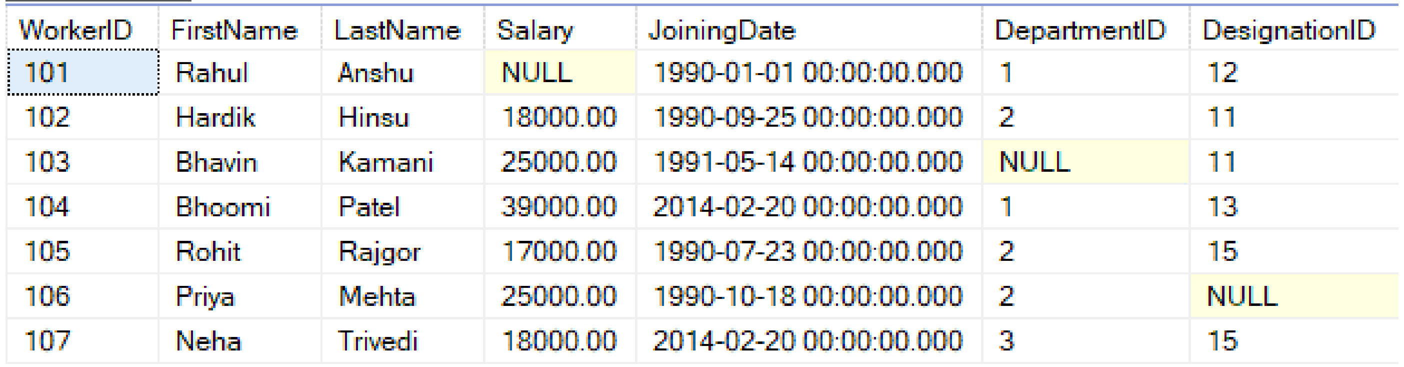
Example : UDF

Table Valued Function

```sql
Create Function Fun_PersonInformation()
returns table
as
return (select * from Person)
```

```sql
--To execute the function
Select * From Fun_PersonInformation()
```

Output



WorkerID	FirstName	LastName	Salary	JoiningDate	DepartmentID	DesignationID
101	Rahul	Anshu	NULL	1990-01-01 00:00:00.000	1	12
102	Hardik	Hinsu	18000.00	1990-09-25 00:00:00.000	2	11
103	Bhavin	Kamani	25000.00	1991-05-14 00:00:00.000	NULL	11
104	Bhoomi	Patel	39000.00	2014-02-20 00:00:00.000	1	13
105	Rohit	Rajgor	17000.00	1990-07-23 00:00:00.000	2	15
106	Priya	Mehta	25000.00	1990-10-18 00:00:00.000	2	NULL
107	Neha	Trivedi	18000.00	2014-02-20 00:00:00.000	3	15

Stored Procedures (SP)

Section - 7

What is Stored Procedure?

- A stored procedure is a **prepared SQL code that you can save**, so the **code can be reused** again whenever needed.
- A procedure has two parts, **header and body**.
- The **header consists** of the **name of the procedure** and the **parameters** passed to the procedure.
- The **body consists** of **declaration section, execution section** and **exception section**.
- A procedure **may or may not return any value**. A procedure **may return more than one value**.

Creating a Stored Procedure

```
CREATE OR ALTER PROCEDURE Procedure_Name
        -- List of Parameters with datatype
AS
BEGIN
     -- SQL statements OR Body
END
```

Deleting the Stored Procedure

```
DROP PROCEDURE Procedure_Name
OR
DROP PROC Procedure_Name
```

What is Stored Procedure? (Cont..)

- **Create** :- It will create a procedure.
- **Alter** :- It will re-create a procedure if it already exists.
- We can pass **parameters** to the procedures in three ways.
    - **IN-parameters** :- These types of parameters are used to send values to stored procedures.
    - **OUT-parameters** :- These types of parameters are used to get values from stored procedures. This is similar to a return type in functions but procedure can return values for more than one parameters.
    - **IN OUT-parameters** :- This type of parameter allows us to pass values into a procedure and get output values from the procedure.
- **AS** indicates the beginning of the body of the procedure.
- **sql_statements** contains the body as a SQL query. (select, insert, update or delete)
- By using `CREATE OR ALTER` together the procedure is created if it does not exist and if it exists then it is replaced with the current code.

Example of Stored Procedure (SP) without parameter [SelectByName]

Create Procedure

```
CREATE PROCEDURE PR_Customer_SelectByName
AS
SELECT Name FROM Customer
```

Execute Procedure

```
EXEC PR_Customer_SelectByName
OR
EXECUTE PR_Customer_SelectByName
```

Customer

CstID	Name	Age	City	Balance
101	Nilesh	32	Rajkot	10000
102	Mayur	35	Jamnagar	25000
103	Hardik	38	Ahmedabad	15000
104	Ajay	42	Surat	20000

Output

Name
Nilesh
Mayur
Hardik
Ajay

Example of Stored Procedure (SP) with one parameter [SelectByPK]

Create Procedure

```
CREATE PROCEDURE PR_Customer_SelectByPK
      @CstID       int IN
AS
SELECT Name, Age, City, Balance FROM Customer
WHERE CstID = @CstID;
```

Execute Procedure

```
EXEC PR_Customer_SelectByPK 103
OR
EXECUTE PR_Customer_SelectByPK 103
```

Customer

CstID	Name	Age	City	Balance
101	Nilesh	32	Rajkot	10000
102	Mayur	35	Jamnagar	25000
103	Hardik	38	Ahmedabad	15000
104	Ajay	42	Surat	20000
105	Nayan	45	Rajkot	25000

Output

Name
Hardik

Example of Stored Procedure (SP) [Insert]

Create Procedure

```
CREATE PROCEDURE PR_Customer_Insert
        @CstID          int,
        @Name           varchar(20),
        @Age            int,
        @City           varchar(20),
        @Balance        decimal(10,2)
AS
INSERT INTO Customer
VALUES
(@CstID, @Name, @Age, @City, @Balance);
```

Customer

CstID	Name	Age	City	Balance
101	Nilesh	32	Rajkot	10000
102	Mayur	35	Jamnagar	25000
103	Hardik	38	Ahmedabad	15000
104	Ajay	42	Surat	20000
105	Nayan	45	Rajkot	25000
106	Umesh	30	Morbi	20000

Execute Procedure

```
EXEC PR_Customer_Insert 106, ‘Umesh’, 30, ‘Morbi’, 20000
OR
EXEC PR_Customer_Insert @CstID=106, @Name=‘Umesh’, @Age=30, @City=‘Morbi’, @Balance=2000
```

Example of Stored Procedure (SP) [Update]

Create Procedure

```
CREATE PROCEDURE PR_Customer_Update
        @CstID          int,
        @Name           varchar(20),
        @Age            int,
        @City           varchar(20),
        @Balance        decimal(10,2)
AS
UPDATE Customer
SET
        Name    = @Name,
        Age     = @Age,
        City    = @City,
        Balance =@Balance
WHERE CstID    = @CstID
```

Execute Procedure

```
EXEC PR_Customer_Update 106, ‘Raj’, 25, ‘Rajkot’, 15000
```

Customer

CstID	Name	Age	City	Balance
101	Nilesh	32	Rajkot	10000
102	Mayur	35	Jamnagar	25000
103	Hardik	38	Ahmedabad	15000
104	Ajay	42	Surat	20000
105	Nayan	45	Rajkot	25000
106	Umesh	30	Morbi	20000

106	Raj	25	Rajkot	15000

Example of Stored Procedure (SP) [Delete]

Create Procedure

```sql
CREATE PROCEDURE PR_Customer_Delete
        @CstID        int
AS
DELETE FROM Customer
WHERE CstID = @CstID;
```

Execute Procedure

```sql
EXEC PR_Customer_Delete 106
```

Customer

CstID	Name	Age	City	Balance
101	Nilesh	32	Rajkot	10000
102	Mayur	35	Jamnagar	25000
103	Hardik	38	Ahmedabad	15000
104	Ajay	42	Surat	20000
105	Nayan	45	Rajkot	25000
106	Umesh	30	Morbi	20000

✗

106	Raj	25	Rajkot	15000

Example of Stored Procedure (SP) [NULL] As Default Parameter

Create Procedure

```
CREATE PROCEDURE PR_Customer_SelectCustName
      @CstName varchar(100) = NULL
AS
SELECT * FROM Customer
WHERE Name = @CstName;
```

Execute Procedure

```
EXEC PR_Customer_SelectCustName
```

When you execute above statement you will get CstID 105 Record in which Name is NULL, If you don't specify any value it will take NULL as default supplied value.

Execute Procedure

```
EXEC PR_Customer_SelectCustName 'Ajay'
```

When you execute above statement you will get records in which name column consist 'Ajay' as value.

Customer

CstID	Name	Age	City	Balance
101	Nilesh	32	Rajkot	10000
102	Mayur	35	Jamnagar	25000
103	Hardik	38	Ahmedabad	15000
104	Ajay	42	Surat	20000
105	*NULL*	45	Rajkot	25000
106	Umesh	30	Morbi	20000

Parameters in Stored Procedures

Section - 8

Stored Procedure OUT/OUTPUT Parameter

- To set output parameters for a stored procedure is basically the same as setting up input parameters, the only difference is that you use the **OUTPUT** clause **after the parameter name** to **specify that it should return a value**.
- The output clause can be specified by either using the keyword "OUTPUT" or just "OUT".

Creating a Stored Procedure with OUT Parameter

```
CREATE PROCEDURE PR_Customer_GetCityCount
      @City          nvarchar(30),
      @Count         int OUTPUT
AS
BEGIN
      SELECT @Count = Count(*)
      FROM Customer
      WHERE City = @City
END
```

Customer

CstID	Name	Age	City	Balance
101	Nilesh	32	Rajkot	10000
102	Mayur	35	Jamnagar	25000
103	Hardik	38	Ahmedabad	15000
104	Ajay	42	Surat	20000
105	Nayan	45	Rajkot	25000
106	Umesh	30	Morbi	20000

Example of Stored Procedure (SP) [OUT] Parameter

Create Procedure

```
CREATE PROCEDURE PR_Customer_GetCityCount
        @City           nvarchar(30),
        @Count          int OUTPUT
AS
BEGIN
        SELECT @Count = Count(*)
        FROM Customer
        WHERE City = @City
END
```

Customer

CstID	Name	Age	City	Balance
101	Nilesh	32	Rajkot	10000
102	Mayur	35	Jamnagar	25000
103	Hardik	38	Ahmedabad	15000
104	Ajay	42	Surat	20000
105	Nayan	45	Rajkot	25000
106	Umesh	30	Morbi	20000

Execute Procedure

```
DECLARE @Count int
EXEC PR_Customer_GetCityCount @City = 'Rajkot', @Count = @Count OUTPUT
SELECT @Count
```

Output

```
1 2
```

Stored Procedure Important Error Messages [Remember]

- If you try to create the stored procedure and it already exists you will get an error message.

```
Msg 2714, Level 16, State 3, Procedure PR_Person_SelectPersonID, Line 1
There is already an object named 'PR_Person_SelectPersonID' in the database.
```

- Error When Parameter Is Not Passed

```
Msg 201, Level 16, State 4, Procedure dbo.PR_Person_SelectPersonID, Line 0
Procedure or function 'PR_Person_SelectPersonID' expects parameter '@WorkerID',
which was not supplied.
```

- If you try to supply other datatype for the parameter

```
Msg 8114, Level 16, State 5, Procedure dbo.PR_Person_SelectPersonID, Line 0
Error converting data type varchar to int.
```

Practice

Student

Rno	Name	Branch	Semester	CPI
101	Ramesh	CE	3	9
102	Mahesh	EC	3	8
103	Suresh	ME	4	7
104	Amit	EE	4	8
105	Anita	CE	4	8
106	Reeta	ME	3	7
107	Rohit	EE	4	9
108	Chetan	CE	3	8
109	Rakesh	CE	4	9

Do it yourself!

- ✓ Create Procedure that returns table with Branch & Semester Wise Maximum SPI Details.
- ✓ Create Procedure that shows student details with CE branch's student's only.
- ✓ Create Procedure that Insert record in student table.
- ✓ Create Procedure that shows first 5 students details.
- ✓ Create Procedure that accepts Semester & Branch and based on that returns record.

Procedures v/s Functions

Section - 9

Function v/s Procedure

Difference

Parameters	**Function**	**Procedure**
Basics	Functions calculate the results of a program on the basis of the given input.	Procedures perform certain tasks in a particular order on the basis of the given inputs.
Try-Catch Blocks	Functions do not provide support for the try-catch Blocks.	Procedures provide support for the try-catch Blocks.
SQL Query	We can call a function in a SQL Query.	We cannot call a procedure in a SQL Query.
SELECT	The SELECT statements can have function calls.	The SELECT statements can never have procedure calls.
Return	A function would return the returning value/control to the code or calling function.	A procedure, on the other hand, would return the control, but would not return any value to the calling function or the code.
DML Statements	We cannot use the DML statements in a function, (functions such as Update, Delete, and Insert).	We can always use the DML statements in the case of a procedure.

Function v/s Procedure (Cont..)

Difference

Parameters	Function	Procedure
Call	A function can be called using a procedure.	A procedure cannot be called using any function.
Compilation	The compilation of a function occurs when we call them in a program.	The compilation of the procedures needs to occur once, and in case it is necessary, these can be called repeatedly, and we don't have to compile them every single time.
Expression	A function must deal with expressions.	A procedure need not deal with expressions.
Explicit Transaction Handling	Functions cannot have explicit transaction handling.	Explicit transaction handling exists in the case of a procedure.

Cursor

Section - 10

Introduction : Cursor

- **Cursor** is a Temporary Memory or Temporary Work Station.
- It is Allocated by Database Server at the time of performing DML(Data Manipulation Language) operations on table by User.
- A SQL cursor is a database object that is used to retrieve data from a result set one row at a time.
- Cursors are used to store Database Tables.
- A SQL cursor is used when the data needs to be updated row by row.
- The purpose for the cursor may be to update one row at a time or perform an administrative process such as SQL Server database backups in a sequential manner.
- We use a cursor to iterate over a set of rows, we can change it to a WHILE loop as **FOR loops are not available in T-SQL**.
- In such cases, the only challenge will be to choose a proper exit condition.

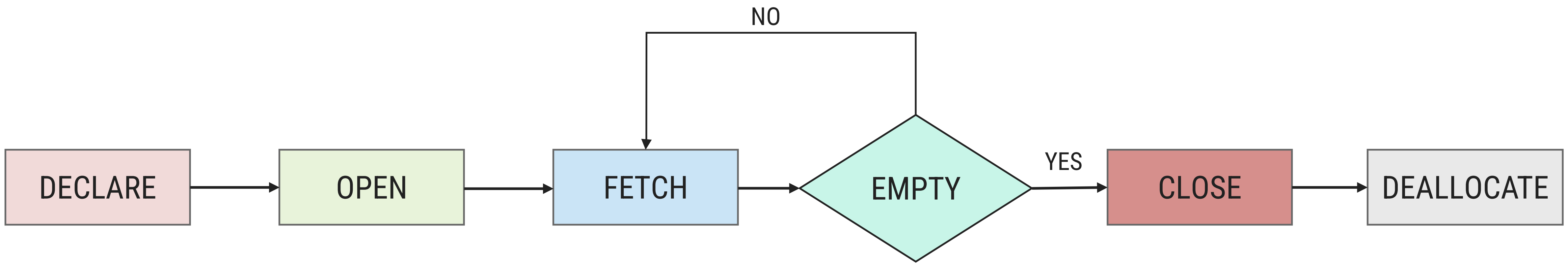
Types of Cursor

1. **Implicit Cursor**
   - Implicit cursors are automatically or default generated by the sql server. It opens a cursor for its internal processing, it is known as Implicit cursor.
   - Implicit cursors are created by default to process the statements when DML statements (INSERT, UPDATE, DELETE) are executed.
2. **Explicit Cursor**
   - If a cursor is opened for processing data through a PL/SQL block as per requirement like user defined cursor, is known as an Explicit cursor.
   - Explicit cursor is created while executing a SELECT statement that returns more than one row.
   - These cursor should be defined in the declaration section of the PL/SQL block and created on a SELECT statement which returns more than one row.

SQL Cursor Life Cycle

- The following steps are involved in a SQL cursor life cycle.
  1. **Declaring Cursor**
     A cursor is declared by defining the SQL statement.
  2. **Opening Cursor**
     A cursor is opened for storing data retrieved from the result set.
  3. **Fetching Cursor**
     When a cursor is opened, rows can be fetched from the cursor one by one or in a block to do data manipulation.
  4. **Closing Cursor**
     The cursor should be closed explicitly after data manipulation.
  5. **Deallocating Cursor**
     Cursors should be deallocated to delete cursor definition and release all the system resources associated with the cursor.

SQL Cursor Life Cycle (Cont..)



SQL Cursor Life Cycle - Steps (Cont..)

1. Declare a cursor.

   ```
   DECLARE cursor_name CURSOR
   FOR select_statement;
   ```

   - To declare a cursor, you specify its name after the DECLARE keyword with the CURSOR data type and provide a SELECT statement that defines the result set for the cursor.

2. Next, open and populate the cursor by executing the SELECT statement:

   ```
   OPEN cursor_name;
   ```

3. Then, fetch a row from the cursor into one or more variables

   ```
   FETCH NEXT FROM cursor INTO variable_list;
   ```

SQL Cursor Life Cycle - Steps (Cont..)

4. SQL Server provides the @@FETCHSTATUS function that returns the status of the last cursor FETCH statement executed against the cursor;
   - If @@FETCHSTATUS returns 0, meaning the FETCH statement was successful.
   - You can use the WHILE statement to fetch all rows from the cursor as shown in the following code

```
WHILE @@FETCH_STATUS = 0
    BEGIN
        FETCH NEXT FROM cursor_name;
    END;
```

5. After that, close the cursor

```
CLOSE cursor_name;
```

6. Finally, deallocate the cursor

```
DEALLOCATE cursor_name;
```

SQL Cursor Life Cycle - Steps (Summary)

Step - 1

```
DECLARE cursor_name CURSOR
FOR select_statement;
```

Step - 2

```
OPEN cursor_name;
```

Step - 3

```
FETCH NEXT FROM cursor INTO variable_list;
```

Step - 4

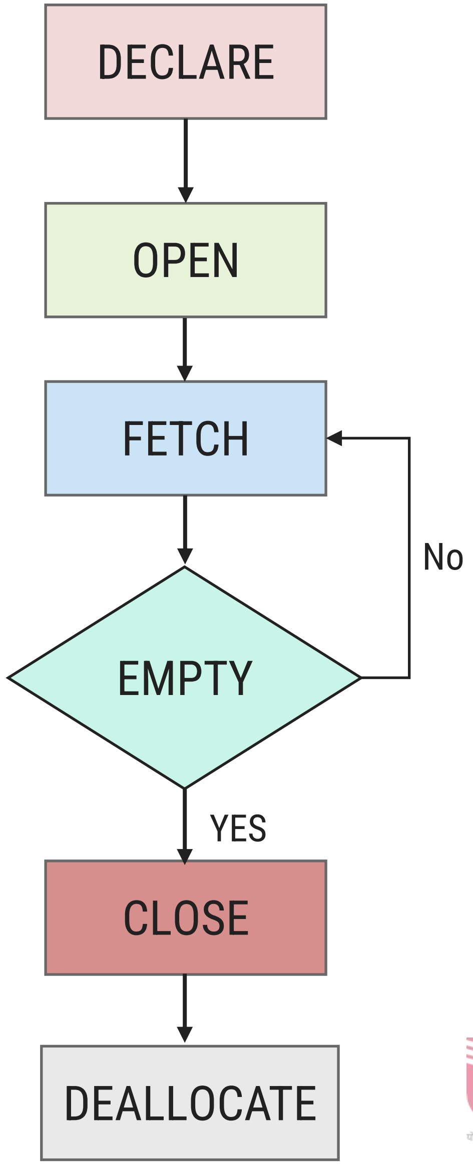
```
WHILE @@FETCH_STATUS = 0
    BEGIN
        FETCH NEXT FROM cursor_name;
    END;
```

Step - 5

```
CLOSE cursor_name;
```

Step - 6

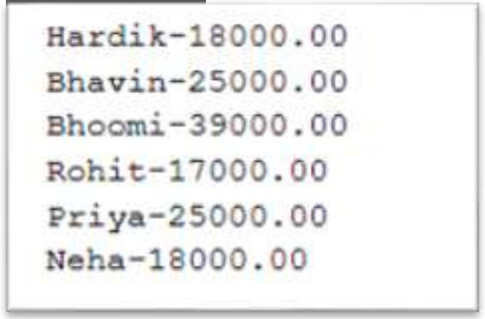
```
DEALLOCATE cursor_name;
```


$$\begin{tikzcd}
\text{DECLARE} \arrow[d] & \\
\text{OPEN} \arrow[d] & \\
\text{FETCH} \arrow[d] & \\
\text{EMPTY} \arrow[u, bend right, "\text{No}"'] \arrow[d, "\text{YES}"] & \\
\text{CLOSE} \arrow[d] & \\
\text{DEALLOCATE} &
\end{tikzcd}$$

Example of Cursor

```sql
DECLARE
    @FirstName VARCHAR(250),
    @Salary    DECIMAL(8,2);
DECLARE cursor_person CURSOR
FOR SELECT
        FirstName,
        Salary
    FROM
        Person;
OPEN cursor_person;
FETCH NEXT FROM cursor_person INTO
    @FirstName,
    @Salary;
WHILE @@FETCH_STATUS = 0
    BEGIN
        PRINT @FirstName + '-' +
CAST(@Salary AS varchar);
        FETCH NEXT FROM cursor_person INTO
            @FirstName,
            @Salary;
    END;
CLOSE cursor_person;
DEALLOCATE cursor_person;
```

Output



```
Hardik-18000.00
Bhavin-25000.00
Bhoomi-39000.00
Rohit-17000.00
Priya-25000.00
Neha-18000.00
```

SQL Cursor Execution

- Cursors use variables to store values returned in each part of the loop.
- Therefore, you'll need to DECLARE all variables you'll need.
- The next thing to do is to **DECLARE … CURSOR FOR SELECT query**, where you'll declare a cursor and also define the query related to populating that cursor.
- You'll OPEN the cursor and FETCH NEXT from the cursor.
- In the WHILE loop you'll test the **@@FETCH_STATUS variable** (WHILE @@FETCH_STATUS = 0). If the condition holds, you'll enter the loop BEGIN … END block and perform statements inside that block.
- After you've looped through the whole result set, you'll exit from the loop.
- You should CLOSE the cursor and DEALLOCATE it.
- Deallocating is important because this delete the cursor definition and free the memory used.

Example of Cursor

We want to get all cities ids and names, together with their related country names.
We will use the PRINT command to print combinations in **each pass of the loop**.

```
-- declare variables used in cursor
DECLARE @city_name VARCHAR(128);
DECLARE @country_name VARCHAR(128);
DECLARE @city_id INT;

-- declare cursor
DECLARE cursor_city_country CURSOR FOR
  SELECT city.id, TRIM(city.city_name),
        TRIM(country.country_name)
  FROM city
  INNER JOIN country
  ON city.country_id = country.id;
 -- open cursor
OPEN cursor_city_country;
```

```
FETCH NEXT FROM cursor_city_country INTO
@city_id, @city_name, @country_name;
WHILE @@FETCH_STATUS = 0
    BEGIN
    PRINT CONCAT('city id: ', @city_id, ' -
city name: ', @city_name, ' - country name:
', @country_name);
    FETCH NEXT FROM cursor_city_country INTO
@city_id, @city_name, @country_name;
    END;

-- close and deallocate cursor
CLOSE cursor_city_country;
DEALLOCATE cursor_city_country;
```

Trigger

Section - 11

Introduction : Trigger

- A SQL Server trigger is a piece of procedural code, like a stored procedure which is only executed when a given event happens.
- There are different types of events that can fire a trigger.
- Like **insertion of rows in a table**, a **change in a table structure** and **even a user logging into a SQL Server instance**.
- There are three main characteristics that make triggers different than stored procedures:
  - Triggers **cannot be manually executed by the user**.
  - There is **no chance for triggers to receive parameters**.
  - You cannot **commit or rollback a transaction inside a trigger**.

Purpose of Triggers

- There are two clear scenarios when triggers are the best choice: auditing and enforcing business rules.
- By using a trigger you can keep track of the changes on a given table by writing a log record with information about the user that made the change and what was changed.
- The main purpose of triggers is to automate execution of code when an event occurs.
- If you need a certain piece of code to always be executed in response to an event, the best option is to use triggers.
- Mostly because they guarantee that the code will be executed or the event that fired the trigger will fail.
- Produce additional checking during insert, update or delete operations on the affected table.
- They allow us to encode complex default values that cannot be handled by default constraints.
- You can calculate aggregated columns in a table using triggers.

Types of Trigger

- In SQL Server we can create the following 3 types of triggers:
  1. Data Definition Language (DDL) triggers
     - DDL triggers are fired when DDL event occurs. i.e. when object is created, altered and dropped by a user.
     - These triggers are created at the database level or server level based on the type of DDL event.
  2. Data Manipulation Language (DML) triggers
     - DML triggers are fired when a DML event occurs. i.e. when data is inserted/ updated/ deleted in the table by a user.
     - These triggers are created at the table level.
     - DML triggers have different types
       - **FOR or AFTER [INSERT, UPDATE, DELETE]**
       - **INSTEAD OF [INSERT, UPDATE, DELETE]**
  3. Logon triggers
     - These triggers are fired when LOGON event occurs.
     - LOGON triggers fired after successful authentication and before establishing the user session.

DML Triggers [Important]

- DML Triggers are of two types:
  1. After trigger (using FOR/AFTER CLAUSE) :
     - After triggers are executed after completing the execution of DML statements.
     - **Example:** If you insert a record/row into a table then the trigger related/associated with the insert event on this table will executed only after inserting the record into that table.
     - If the record/row insertion fails, SQL Server will not execute the after trigger.
  2. Instead of Trigger (using INSTEAD OF CLAUSE) :
     - Instead of trigger are executed before starts the execution of DML statements.
     - An instead of trigger allows us to skip an INSERT, DELETE, or UPDATE statement to a table and execute other statements defined in the trigger instead.
     - The actual INSERT, DELETE or UPDATE operation does not occur at all.
       - **Example**: If you insert a record/row into a table then the trigger related/associated with the insert event on this table will be executed before inserting the record into that table.
     - If the record/row insertion fails, SQL Server will execute the instead of trigger.

Darshan UNIVERSITY

Syntax of Trigger

Trigger Syntax

```
CREATE [OR ALTER] TRIGGER Trigger_name

ON Table_name OR view_name

{ FOR OR AFTER | INSTEAD OF }

{ [ INSERT ] [ , ] [ UPDATE ] [ , ] [ DELETE ] }
AS
BEGIN
        --SQL Statements (Body)
        Executable statements
END;
```

Example of Trigger

- Create a trigger on department table for insert, update and delete statement to display a message “Record is affected”.

Trigger Example

```
CREATE TRIGGER Dept_Msg
ON Department
AFTER INSERT, UPDATE, DELETE
AS
BEGIN
        PRINT 'Record is affected'
END
```

Trigger Executed When…

```
Insert into Department values (101,'Computer Department’)
Update Department Set DepartmentName = 'Acc Dept’ Where DepartmentID = 4
Delete From Department Where DepartmentID = 4
```

Example of Trigger [Insert]

- Create a trigger on department table for insert statement to insert description like (record with deptid=[103] is inserted on [current date]) in MSG table.

Insert Trigger

```
CREATE TRIGGER Department_Insert
ON Department
FOR INSERT
AS
BEGIN
       DECLARE @DepartmentID INT
       SELECT @DepartmentID = DepartmentID FROM INSERTED
       INSERT INTO MSG
       VALUES
       ('RECORD WITH DeptID=' + CAST(@DepartmentID AS VARCHAR(10)) +' IS
       INSERTED ON '+ CAST(GETDATE() AS VARCHAR(50)))
END
```

Example of Trigger [Update]

- Create a trigger on department table for update statement to insert description like (record with deptid=[103] is updated on [current date]) in MSG table.

Update Trigger

```
CREATE TRIGGER Department_Insert
ON Department
FOR UPDATE
AS
BEGIN
        DECLARE @DepartmentID INT
        SELECT @DepartmentID = DepartmentID FROM INSERTED
        INSERT INTO MSG
        VALUES
        ('RECORD WITH DeptID=' + CAST(@DepartmentID AS VARCHAR(10)) +' IS
        UPDATED ON '+ CAST(GETDATE() AS VARCHAR(50)))
END
```

Example of Trigger [Delete]

- Create a trigger on department table for delete statement to insert description like (record with deptid=[103] is deleted on [current date]) in MSG table.

Delete Trigger

```
CREATE TRIGGER Department_Insert
ON Department
FOR Delete
AS
BEGIN
       DECLARE @DepartmentID INT
       SELECT @DepartmentID = DepartmentID FROM DELETED
       INSERT INTO MSG
       VALUES
       ('RECORD WITH DeptID=' + CAST(@DepartmentID AS VARCHAR(10)) +' IS
       DELETED ON '+ CAST(GETDATE() AS VARCHAR(50)))
END
```

Example of Trigger [Custom]

- Create a trigger on result table for insert statement to **update total marks automatically**.
- Here **total marks is sum of sub1, sub2 and sub3**.

Update Trigger

```sql
CREATE TRIGGER TR_TOTALMARKS
ON RESULT
FOR INSERT
AS
BEGIN
        DECLARE @S1 INT, @S2 INT, @S3 INT, @TOTAL INT
        SELECT @S1= SUB1 FROM INSERTED
        SELECT @S2= SUB2 FROM INSERTED
        SELECT @S3= SUB3 FROM INSERTED
        SET @TOTAL= @S1+@S2+@S3
        UPDATE RESULT
        SET TOTAL=@TOTAL
        WHERE SUB1=@S1 AND SUB2=@S2 AND SUB3=@S3
END
```

Example of Trigger [Custom]

- Create a trigger on result table for insert statement to **update total marks automatically**.
- Here **total marks is sum of sub1, sub2 and sub3**.

Update Trigger

```
CREATE TRIGGER TR_TOTALMARKS
ON RESULT
FOR INSERT
AS
BEGIN
        DECLARE @S1 INT, @S2 INT, @S3 INT, @TOTAL INT
        SELECT @S1= INSERTED.SUB1, @S2= INSERTED.SUB2, @S3= INSERTED.SUB3
        FROM INSERTED
        SET @TOTAL= @S1+@S2+@S3
        UPDATE RESULT
        SET TOTAL=@TOTAL
        WHERE SUB1=@S1 AND SUB2=@S2 AND SUB3=@S3
END
```

Trigger [Practice]

Student

Rno	Name	Branch	Semester	CPI
101	Ramesh	CE	3	9
102	Mahesh	EC	3	8
103	Suresh	ME	4	7
104	Amit	EE	4	8
105	Anita	CE	4	8
106	Reeta	ME	3	7
107	Rohit	EE	4	9
108	Chetan	CE	3	8
109	Rakesh	CE	4	9

Do it yourself!

- ✓ Create Trigger on given table that enters Record in another table when any one updates in student table.
- ✓ Create Trigger on given table that enters message in another table with name when any record is Inserted.
- ✓ Create Trigger on given table that enters message with Rno in another log table when any record is deleted.

Pros/Advantages of SQL Server Triggers

- Triggers are easy to code.
- You can call stored procedures and functions from inside a trigger.
- Triggers are useful when you need to validate inserted or updated data in batches instead of row by row.
- Triggers are useful if you need to be sure that certain events always happen when data is inserted, updated or deleted. This is the case when you have to deal with complex default values of columns, or modify the data of other tables.
- Triggers allow recursion, It is recursive when a trigger on a table performs an action on the base table that causes another instance of the trigger to fire.

Cons/Disadvantages of SQL Server Triggers

- Triggers needs to be properly documented.
- Triggers add overhead to DML statements.
- If there are many nested triggers it could get very hard to debug and troubleshoot, which consumes development time and resources.
- Recursive triggers are even harder to debug than nested triggers.
- If you use triggers to enforce referential integrity you have to be aware that triggers can be disabled by users that have the ALTER permission on the table or view on which the trigger was created. To avoid this, you may have to review user permissions.

Exception Handling

Section - 12

Introduction : Error Handling

- An error condition during a program execution is called an exception and the mechanism for resolving such an exception is known as exception handling.
- SQL Server provides TRY, CATCH blocks for exception handling.
- We can put all T-SQL statements into a TRY BLOCK and the code for exception handling can be put into a CATCH block.
- We can also generate user-defined errors using a THROW block.
- Error handling in SQL Server gives us control over the Transact-SQL code.
- For example, when things go wrong, we get a chance to do something about it and possibly make it right again.
- In exception handling all T-SQL statements are put into a try block. If all statements execute without any error then everything is OK else control will go to the catch block.

Types of SQL Server Exceptions

- SQL Server contains the following two types of exceptions:
  1. System Defined
  2. User Defined

- System Defined Exception
  - In a System Defined Exception the exceptions (errors) are generated by the system.

- User Defined Exception
  - This type of exception is user generated, not system generated.

System Defined Exception - Example

System Defined Exception

```
Declare @val1 int;
Declare @val2 int;
BEGIN TRY
        Set @val1=8;
        Set @val2=@val1/0; /* Error Occur Here */
END TRY
BEGIN CATCH
        Print 'Error Occur that is:' + Error_Message()
END CATCH
```

O/P:

Error Occur that is: Divide by zero error encountered.

User Defined Exception – Example [Odd/Even Number]

User Defined Exception

```
Declare @val1 int;
Declare @val2 int;
BEGIN TRY
        Set @val1=8;
        Set @val2=@val1%2;
        IF @val2=1
                PRINT 'Error Occur'
        ELSE
        BEGIN
                PRINT 'Error Not Occur';
                Throw 60000,'Number Is Even',5
        END
END TRY
BEGIN CATCH
        Print 'Error Occur that is : ' + Error_Message()
END CATCH
```

O/P: (For Value 8)

```
Error Not Occur
Error Occur that is : Number Is Even
```

O/P: (For Value 3)

```
Error Occur
```

Stored Procedure – Exception Example

Handling Exception in Stored Procedure

```
CREATE PROCEDURE Sample_Proc
AS
BEGIN
        BEGIN TRY
                SELECT Salary + FirstName From Person Where WorkerID=101
        END TRY
        BEGIN CATCH
                SELECT ERROR_PROCEDURE() AS ProcName;
                SELECT ERROR_MESSAGE() AS Message;
        END CATCH;
END
```

```
Execute Procedure : Exec Sample_Proc
```

O/P:

```
Error converting data type varchar to numeric.
```

@@ERROR

- @@ERROR return the error number for last executed T-SQL statements.
- It returns 0 if the previous Transact-SQL statement encountered no errors else return an error number.

@@Error

```sql
Update Employee set Salary=19000
Where Emp_IID=5
IF @@ERROR = 547
PRINT 'A check constraint violation occurred.';
```

O/P:

```
Msg 547, Level 16, State 0, Line 1
The UPDATE statement conflicted with the CHECK constraint "CK__Employee__Salary__68487DD7". The conflict occurred in database "Home_Management", table "dbo.Employee", column 'Salary'.
The statement has been terminated.

A check constraint violation occurred.
```

ERROR_NUMBER()

- ERROR_NUMBER() returns the error number that caused the error. It returns zero if called outside the catch block.

Error_Number ( )

```
BEGIN TRY
        Update Employee set Salary=19000
        Where Emp_IID=5
END TRY
BEGIN CATCH
        SELECT ERROR_NUMBER() AS ErrorNo;
END CATCH;
```

```
O/P:
Error No
547
```

@@ERROR v/s ERROR_NUMBER ()

- ERROR_NUMBER () can only be used in a catch block, **outside a catch block it returns Null** but @@ERROR can be used inside or outside the catch block.
- ERROR_NUMBER is a contrast to @@ERROR, that only returns the error number in the statement immediately after the one that causes an error, or the first statement of a CATCH block.

@@Error v/s Error_Number ( )

```
BEGIN TRY
        Update Employee set Salary=19000 Where Emp_IID=5
END TRY
BEGIN CATCH
        SELECT ERROR_NUMBER() AS ErrorNumber;
        PRINT @@ERROR
END CATCH;
```

Handling Errors using TRY…CATCH

Here’s how the syntax looks like.

```
BEGIN TRY
     --code to try
END TRY
BEGIN CATCH
     --code to run if an error occurs is generated in try
END CATCH
```

- Anything between the BEGIN TRY and END TRY is the code that we want to monitor for an error.
- So, if an error would have happened inside this TRY statement, the control would have immediately get transferred to the CATCH statement and then it would have started executing code line by line.
- Now, inside the CATCH statement, we can try to fix the error, report the error or even log the error, so we know when it happened, who did it by logging the username, all the useful stuff.

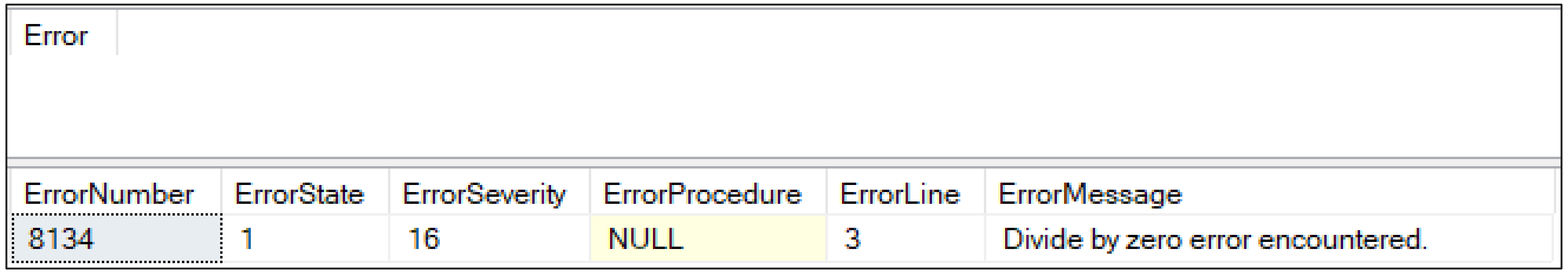
Nested TRY Block

```
BEGIN TRY
    --- Statements that may cause exceptions
END TRY
BEGIN CATCH
    -- Statements to handle exception
    BEGIN TRY
        --- Nested TRY block
    END TRY
    BEGIN CATCH
        --- Nested CATCH block
    END CATCH
END CATCH
```

Error Functions Example

Divide by Zero Exception

```sql
BEGIN TRY
-- Generate a divide-by-zero error
  SELECT
    1 / 0 AS Error;
END TRY
BEGIN CATCH
  SELECT
    ERROR_NUMBER() AS ErrorNumber,
    ERROR_STATE() AS ErrorState,
    ERROR_SEVERITY() AS ErrorSeverity,
    ERROR_PROCEDURE() AS ErrorProcedure,
    ERROR_LINE() AS ErrorLine,
    ERROR_MESSAGE() AS ErrorMessage;
END CATCH;
```

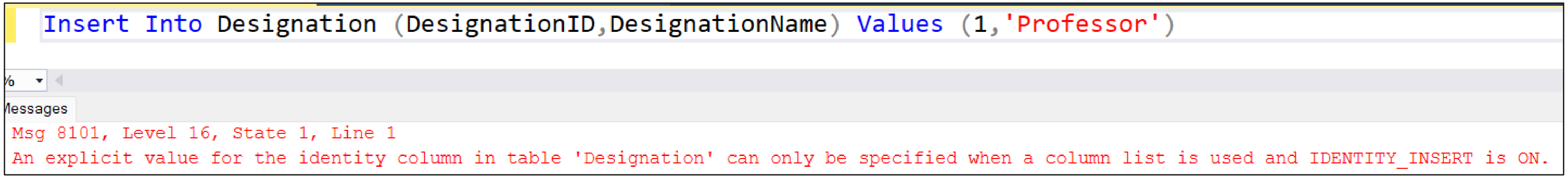


```
Error

ErrorNumber  ErrorState  ErrorSeverity  ErrorProcedure  ErrorLine  ErrorMessage
8134         1           16             NULL            3          Divide by zero error encountered.
```

Error Functions in SQL

We even have access to some special data only available inside the CATCH statement:



```
Insert Into Designation (DesignationID,DesignationName) Values (1,'Professor')
```

```
Messages
Msg 8101, Level 16, State 1, Line 1
An explicit value for the identity column in table 'Designation' can only be specified when a column list is used and IDENTITY_INSERT is ON.
```

Error	Details
ERROR_NUMBER	Returns the internal number of the error
ERROR_STATE	Returns the information about the source
ERROR_SEVERITY	Returns the information about anything from informational errors to errors user of DBA can fix, etc. 13 - Indicates transaction deadlock errors. 14 - Indicates security-related errors, such as permission denied. 15 - Indicates syntax errors in the Transact-SQL command. 16 - Indicates general errors that can be corrected by the user.
ERROR_LINE	Returns the line number at which an error happened on
ERROR_PROCEDURE	Returns the name of the stored procedure or function
ERROR_MESSAGE	Returns the most essential information and that is the message text of the error

Procedure with TRY…CATCH Example

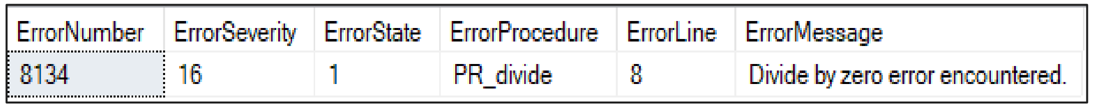
Divide by Zero Exception in Stored Procedure

```
CREATE PROC PR_divide
    @a decimal,
    @b decimal,
    @c decimal output
AS
BEGIN
    BEGIN TRY
        SET @c = @a / @b;
    END TRY
    BEGIN CATCH
        SELECT
            ERROR_NUMBER() AS ErrorNumber
            ,ERROR_SEVERITY() AS ErrorSeverity
            ,ERROR_STATE() AS ErrorState
            ,ERROR_PROCEDURE() AS ErrorProcedure
            ,ERROR_LINE() AS ErrorLine
            ,ERROR_MESSAGE() AS ErrorMessage;
    END CATCH
END;
```

```
--Executing a procedure PR_divide
DECLARE @r decimal;
EXEC PR_divide 10, 2, @r output;
PRINT @r;
```

```
--Output
5
```

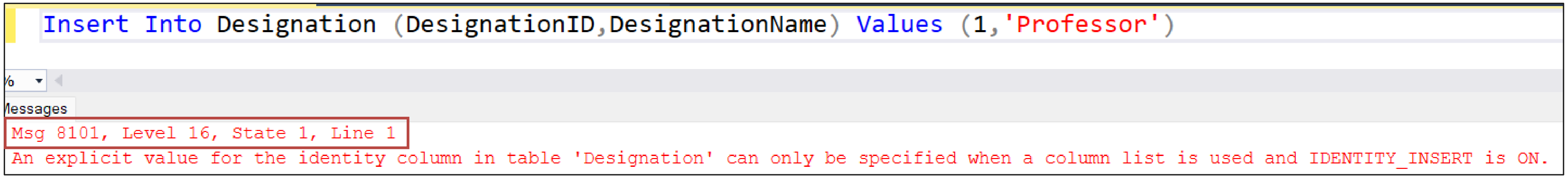
```
--Executing a procedure PR_divide
DECLARE @r decimal;
EXEC PR_divide 10, 0, @r output;
PRINT @r;
```



ErrorNumber	ErrorSeverity	ErrorState	ErrorProcedure	ErrorLine	ErrorMessage
8134	16	1	PR_divide	8	Divide by zero error encountered.

SQL Server RAISERROR

- We use the RAISERROR inside a TRY block to cause execution to jump to the associated CATCH block.
- Inside the CATCH block, we use the RAISERROR to return the error information that invoked the CATCH block.



```
Insert Into Designation (DesignationID,DesignationName) Values (1,'Professor')
```

```
Msg 8101, Level 16, State 1, Line 1
An explicit value for the identity column in table 'Designation' can only be specified when a column list is used and IDENTITY_INSERT is ON.
```

RAISERROR Example

```
BEGIN TRY
 --Syntax: Raiserror (errorid/errormsg, SEVERITY, state)
    RAISERROR('Error occurred in the TRY block.', 17, 1);
END TRY
BEGIN CATCH
    SELECT
       ERROR_MESSAGE(),
       ERROR_SEVERITY(),
       ERROR_STATE();
END CATCH;
```

RAISERROR Example

RAISERROR

```sql
CREATE PROCEDURE spDivideBy1(@No1 INT, @No2 INT)
AS
BEGIN
  DECLARE @Result INT
  SET @Result = 0
  BEGIN TRY
    IF @No2 = 1
        RAISERROR ('DIVISOR CANNOT BE ONE', 16, 1)
        SET @Result = @No1 / @No2
        PRINT 'THE RESULT IS: '+CAST(@Result AS VARCHAR)
  END TRY
  BEGIN CATCH
        PRINT ERROR_NUMBER()
        PRINT ERROR_MESSAGE()
        PRINT ERROR_SEVERITY()
        PRINT ERROR_STATE()
  END CATCH
END
```

```
--Exec spDivideBy1 10,1
50000
DIVISOR CANNOT BE ONE
16
1
```

```
--Exec spDivideBy1 10,2
THE RESULT IS: 5
```

Throw Example

THROW

```sql
CREATE PROCEDURE spDivideBy2(@No1 INT, @No2 INT)
AS
BEGIN
  DECLARE @Result INT
  SET @Result = 0
  BEGIN TRY
    IF @No2 = 1
        THROW 50001,'DIVISOR CANNOT BE ONE', 1
        SET @Result = @No1 / @No2
        PRINT 'THE RESULT IS: '+CAST(@Result AS VARCHAR)
  END TRY
  BEGIN CATCH
    PRINT ERROR_NUMBER()
    PRINT ERROR_MESSAGE()
    PRINT ERROR_SEVERITY()
    PRINT ERROR_STATE()
  END CATCH
END
```

```
--Exec spDivideBy2 10,1
50001
DIVISOR CANNOT BE ONE
16
1

--Exec spDivideBy2 10,2
THE RESULT IS: 5
```

TCL and DCL Commands

Section - 13

What is Transaction ??

- **Transaction** is a set of database operations that performs a particular task.

Person A
Account A
Balance : 2000

Sum of Balance **before** Transfer : 5000

Transfer 1000

Person B
Account B
Balance : 3000

Step 1: Debit 1000 rs. from Account A

Step 2: Credit 1000 rs. into Account B

Sum of Balance after Successful **Transaction** : 5000

Transaction Control Command

- A transaction must be completely successful or completely fail to maintain database consistency.

Example of Successful Transaction

Person A
Account A
Balance : 2000

Sum of Balance **before** Transfer : 5000

Transfer 1000

Person B
Account B
Balance : 3000

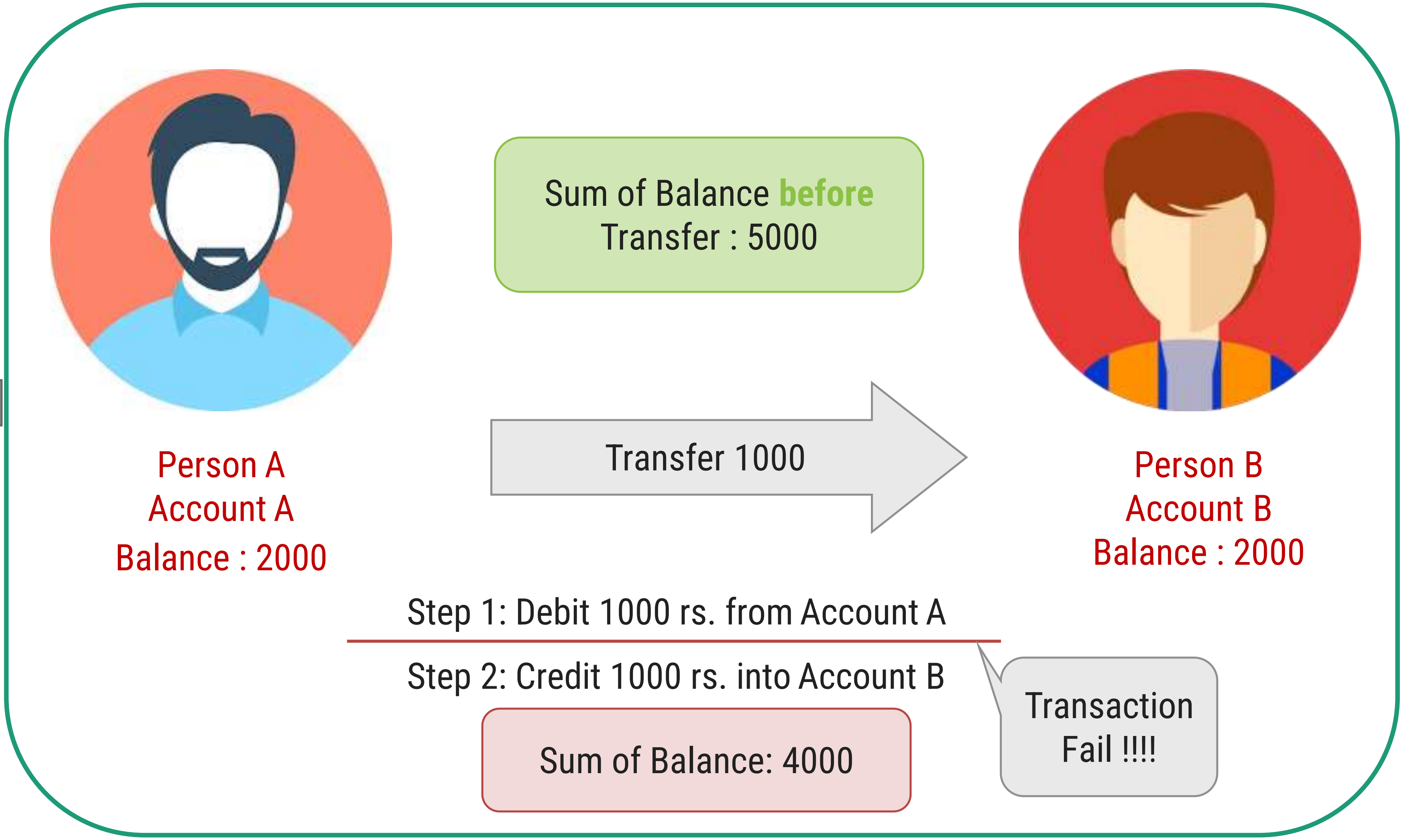
Step 1: Debit 1000 rs. from Account A

Step 2: Credit 1000 rs. into Account B

Sum of Balance after Successful **Transaction** : 5000

Transaction Control Command (Conti…)

- A transaction must be completely successful or completely fail to maintain database consistency.



Example of Fail Transaction

Transaction Control Command (Conti…)

- We can say that a **transaction** is considered as a **sequence of database operations**.
- These operations involve various data manipulation operations such as **insert**, **update** and **delete**.
- These operations are performed in two steps
  - To make changes permanent using **COMMIT** statement
  - To undo a part of or the entire transaction using **ROLLBACK** statement
- A **transaction** begins with the execution of first SQL statement after a **COMMIT** and can be undone using **ROLLBACK** command.
- A **transaction** can be closed by using **COMMIT** or **ROLLBACK** command. When a transaction is closed, all the locks acquired during that transaction are released.

Transaction Control Command (Conti…)

- There are 3 commands which comes under the TCC;

1. **Commit**
2. **Savepoint**
3. **Rollback**

1. Commit

- There are two ways to commit a transaction
  - **Explicit Commit**
  - **Implicit Commit**

1. Commit (Conti…)

- **Explicit Commit**
  - To commit a transaction explicitly, user needs to request COMMIT command explicitly.
  - A COMMIT command terminates the current transaction and makes all the changes permanent.
  - Various data manipulation operations such as INSERT, UPDATE and DELETE are not effect permanently until they are committed.
  - **Syntax;**

```
COMMIT TRAN[SACTION]   [transaction_name | @transaction_variablename];
```

- **Implicit Commit**
  - There are some operations which forces a COMMIT to occur automatically, even user don't specify the COMMIT command.
  - Some of them are as below;
    - Quit Command
    - Exit Command
    - DDL Commands (CREATE, ALTER, DROP, TRUNCATE etc..)

2. Savepoint

- It is required to create a savepoint which help us to cancel transaction partially.
- A savepoint marks and save the current point in the processing of a transaction.
- **Syntax;**

```
SAVE TRAN[SACTION]     [savpoint_name | @savepoint_variablename] ;
```

- When a **ROLLBACK** is used with **SAVEPOINT**, part of the transaction is cancelled.
- All the operations performed after creating a savepoint are undone.
- It is also possible to create more than one savepoint within a single transaction.

3. Rollback

- A transaction can be cancelled using ROLLBACK command either completely or partially.
- A ROLLBACK command terminates the current transaction and undone any changes made during the transaction.
- SQL Server also performs auto rollback.
- In situation like, Computer failure, SQL Server automatically rollbacks any uncommitted work, when the database bought back next time.
- Rollback command can also used to terminate the current transaction partially.
- **Syntax;**

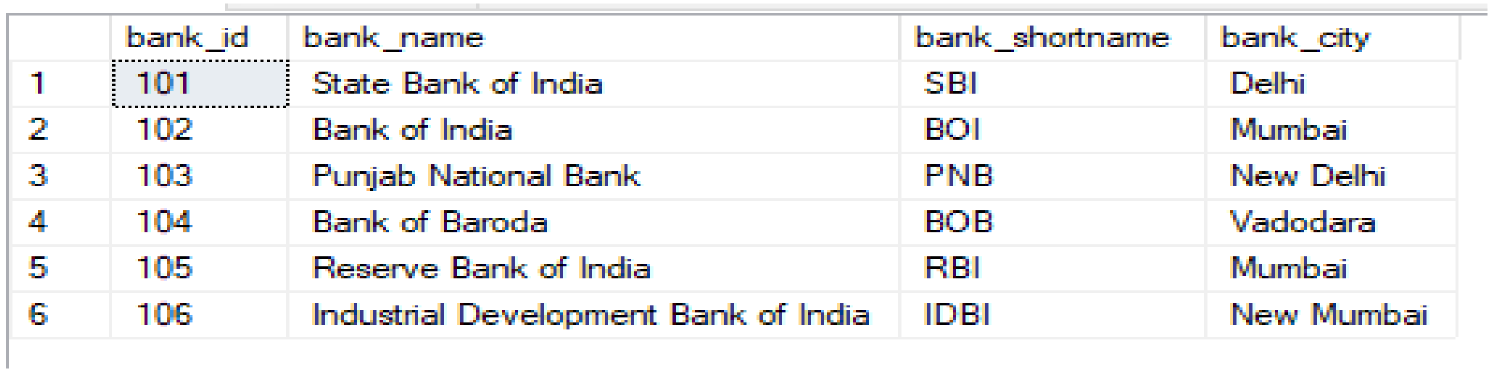
```
ROLLBACK TRAN[SACTION]   [transaction_name | savepoint_name | @transaction_variablename |
                         @savepoint_variablename];
```

Darshan UNIVERSITY

Example of COMMIT, ROLLBACK and SAVEPOINT

- Step 1: First Open the MS SQL server management studio and connect to the database.
- Step 2: Let’s display entity bank_detail with the help of **SELECT** command which we are already created using **CREATE** and add data using **INSERT** command,

```
SELECT * FROM bank_detail;
```



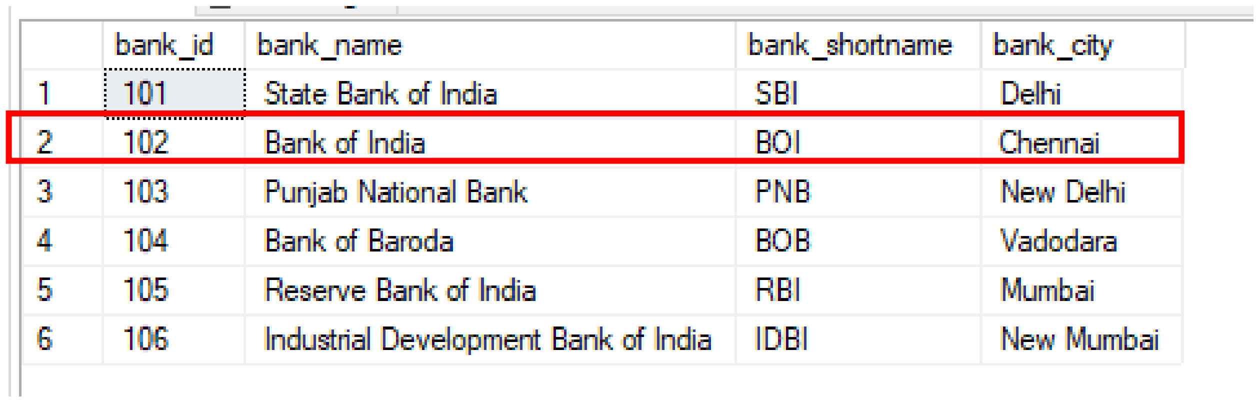
	bank_id	bank_name	bank_shortname	bank_city
1	101	State Bank of India	SBI	Delhi
2	102	Bank of India	BOI	Mumbai
3	103	Punjab National Bank	PNB	New Delhi
4	104	Bank of Baroda	BOB	Vadodara
5	105	Reserve Bank of India	RBI	Mumbai
6	106	Industrial Development Bank of India	IDBI	New Mumbai

Example of COMMIT, ROLLBACK and SAVEPOINT (Conti…)

- Step 3:Now begin new transaction

  BEGIN TRANSACTION tran1 ;

- Step 4: Now update bank_detail record set bank_city = 'Chennai' which bank_id = '102';



	bank_id	bank_name	bank_shortname	bank_city
1	101	State Bank of India	SBI	Delhi
2	102	Bank of India	BOI	Chennai
3	103	Punjab National Bank	PNB	New Delhi
4	104	Bank of Baroda	BOB	Vadodara
5	105	Reserve Bank of India	RBI	Mumbai
6	106	Industrial Development Bank of India	IDBI	New Mumbai

- Step 5: Now create a savepoint as per shown below;

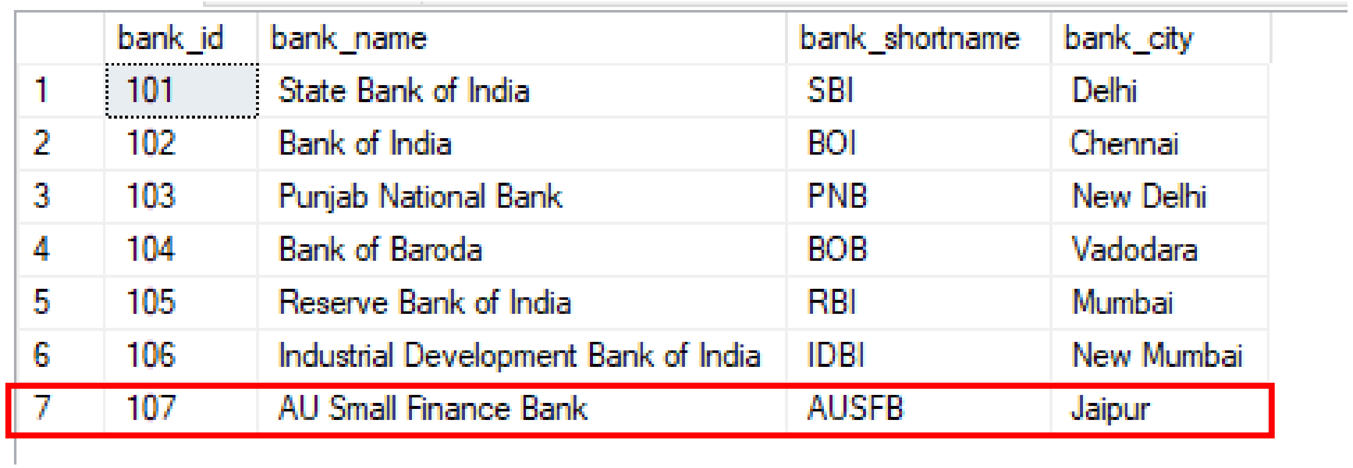
  SAVE TRANSACTION updt1;

- Step 6: Now insert 1 record into bank_detail as per following:

Bank_id	Bank_name	Bank_shortname	Bank_city
107	AU Small Finance Bank	AUSFB	Jaipur

Example of COMMIT, ROLLBACK and SAVEPOINT (Conti…)

- Step 7: Now display the inserted bank_detail record using **SELECT** command;



	bank_id	bank_name	bank_shortname	bank_city
1	101	State Bank of India	SBI	Delhi
2	102	Bank of India	BOI	Chennai
3	103	Punjab National Bank	PNB	New Delhi
4	104	Bank of Baroda	BOB	Vadodara
5	105	Reserve Bank of India	RBI	Mumbai
6	106	Industrial Development Bank of India	IDBI	New Mumbai
7	107	AU Small Finance Bank	AUSFB	Jaipur

- Step 8: Now create a **savepoint** as per shown below;

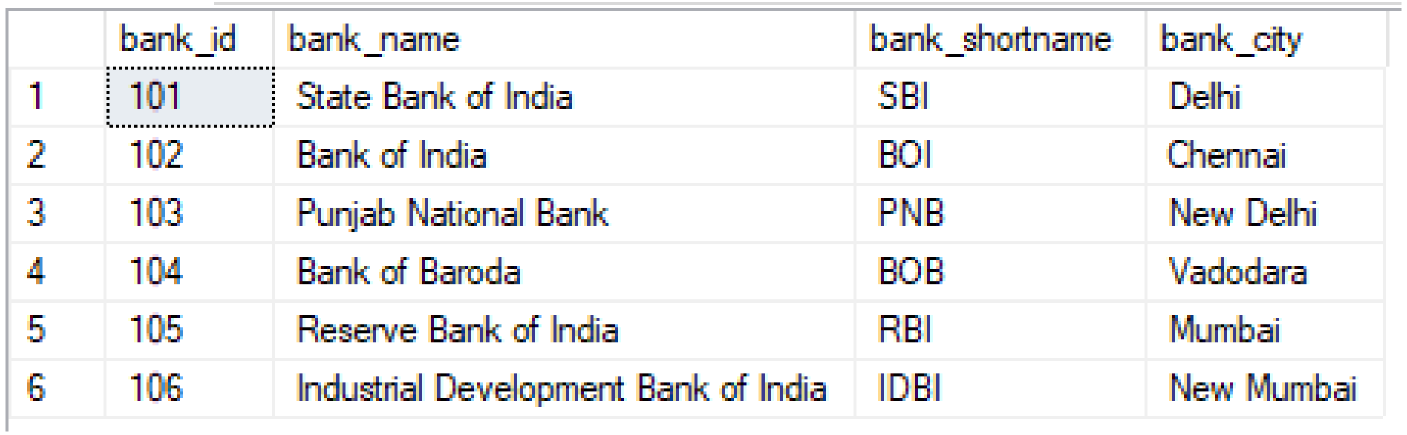
```
SAVE TRANSACTION  insrt1 ;
```

- Step 9: Now use the **Rollback** command and cancel the transaction partially;

```
ROLLBACK TRANSACTION  updt1 ;
```

Example of COMMIT, ROLLBACK and SAVEPOINT (Conti…)

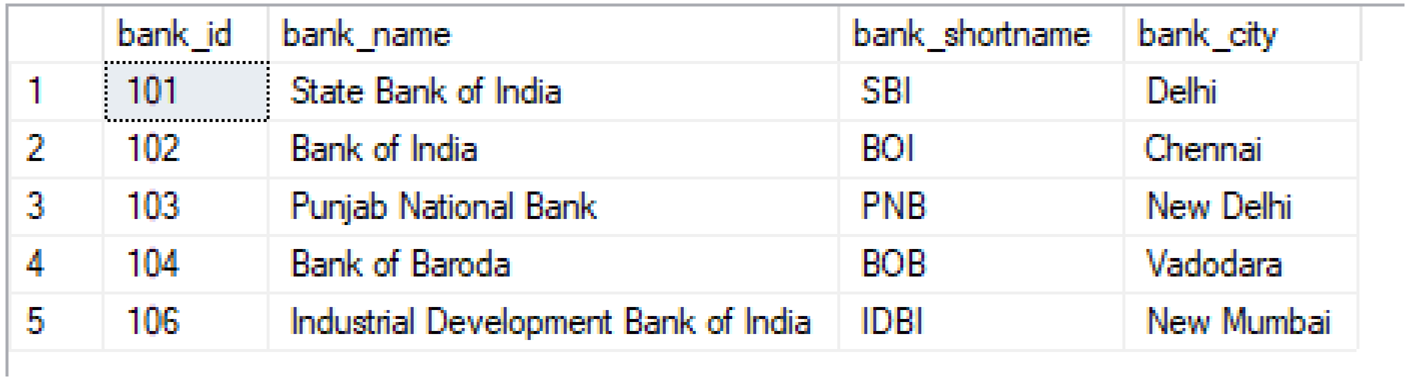
- Step 10: Display entity bank_detail with the help of **SELECT** command;



	bank_id	bank_name	bank_shortname	bank_city
1	101	State Bank of India	SBI	Delhi
2	102	Bank of India	BOI	Chennai
3	103	Punjab National Bank	PNB	New Delhi
4	104	Bank of Baroda	BOB	Vadodara
5	105	Reserve Bank of India	RBI	Mumbai
6	106	Industrial Development Bank of India	IDBI	New Mumbai

- Step 11: Now **DELETE** 1 record from the bank_detail which bank_id = 105 and Fetch data using **SELECT** command;

```
DELETE FROM   bank_detail   WHERE   bank_id = '105';
```

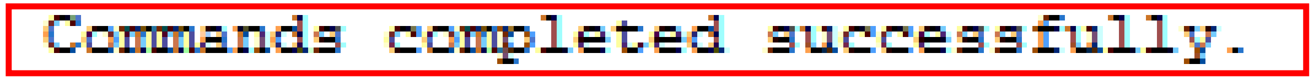


	bank_id	bank_name	bank_shortname	bank_city
1	101	State Bank of India	SBI	Delhi
2	102	Bank of India	BOI	Chennai
3	103	Punjab National Bank	PNB	New Delhi
4	104	Bank of Baroda	BOB	Vadodara
5	106	Industrial Development Bank of India	IDBI	New Mumbai

Example of COMMIT, ROLLBACK and SAVEPOINT (Conti…)

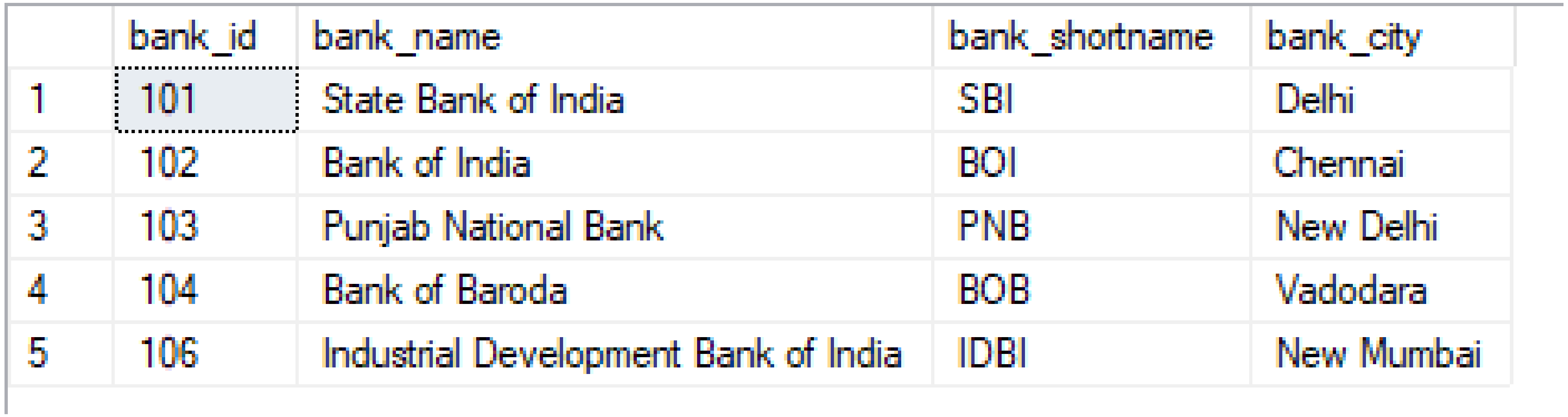
- Step 12: Now **commit** the transaction permanently into the database using;

```
COMMIT TRANSACTION tran1 ;      / COMMIT;
```



```
Commands completed successfully.
```

- Step 13: Now Display the bank_detail using **SELECT** command;



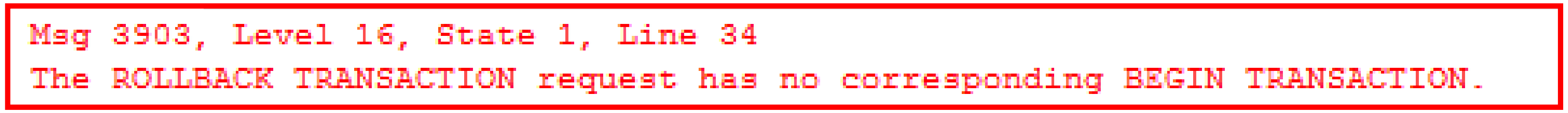
	bank_id	bank_name	bank_shortname	bank_city
1	101	State Bank of India	SBI	Delhi
2	102	Bank of India	BOI	Chennai
3	103	Punjab National Bank	PNB	New Delhi
4	104	Bank of Baroda	BOB	Vadodara
5	106	Industrial Development Bank of India	IDBI	New Mumbai

Example of COMMIT, ROLLBACK and SAVEPOINT (Conti…)

- Step 14: Now after commit try to **ROLLBACK** transaction to 2nd save point;

```
ROLLBACK TRANSACTION  insrt1 ;
```

- Step 15: It will display following **error** because after commit transaction you can't rollback transactions;

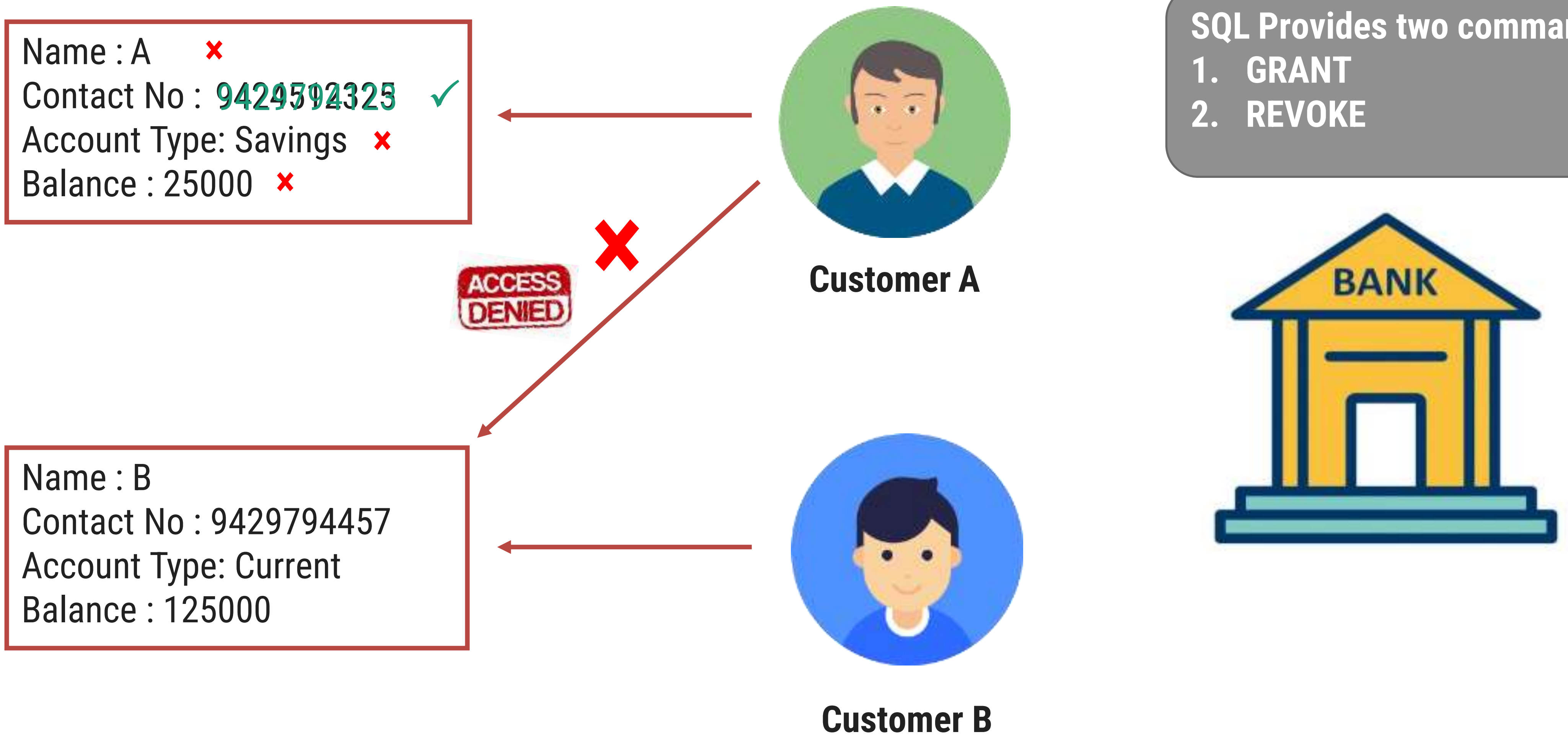


```
Msg 3903, Level 16, State 1, Line 34
The ROLLBACK TRANSACTION request has no corresponding BEGIN TRANSACTION.
```

Data Control Language

- Security of information stored in database is one of the prime concerns for any database management system.
- An unauthorized access to a database must be prevented.
- The rights allow the user to use database contents are called privileges.
- SQL provides security to database contents in two phases
  - User required **valid user id** and **password**
  - User must have **privileges**
- In a multi-user system, different user needs to access different parts of the database.
- The database designer determines which user needs access to which part of the database.
- According to this, various privileges are granted to different users.

Data Control Language Real Life Example



Name : A ✗
Contact No : 9424512325 9429794123 ✓
Account Type: Savings ✗
Balance : 25000 ✗

Name : B
Contact No : 9429794457
Account Type: Current
Balance : 125000

SQL Provides two commands;

1. GRANT
2. REVOKE

GRANT – Grant Privileges

- **GRANT** command is used to granting privileges means to give permission to some user to access database object or a part of a database object.
- This command provides various types of access.
- The owner of a database object can grant all privileges or specific privileges to other users.
- **Syntax;**

```
GRANT        Object_privileges
```

User can grant all or specific privileges owned by him/her. List of various privileges are as below;

1. ALL
2. ALTER
3. DELETE
4. INDEX
5. INSERT
6. REFERENCES
7. SELECT
8. UPDATE

GRANT – Grant Privileges (Conti…)

- **GRANT** command is used to granting privileges means to give permission to some user to access database object or a part of a database object.
- This command provides various types of access.
- The owner of a database object can grant all privileges or specific privileges to other users.
- **Syntax;**

```
GRANT       Object_privileges
ON          Object_Name
```

Name of the object on which object we want to give privileges.

GRANT – Grant Privileges (Conti…)

- **GRANT** command is used to granting privileges means to give permission to some user to access database object or a part of a database object.
- This command provides various types of access.
- The owner of a database object can grant all privileges or specific privileges to other users.
- **Syntax;**

```
GRANT    Object_privileges
ON       Object_Name
TO       UserName
```

Name of the user to which we want to give privileges.

GRANT – Grant Privileges (Conti…)

- **GRANT** command is used to granting privileges means to give permission to some user to access database object or a part of a database object.
- This command provides various types of access.
- The owner of a database object can grant all privileges or specific privileges to other users.
- **Syntax;**

```
GRANT         Object_privileges
ON            Object_Name
TO            UserName
[WITH GRANT OPTION]
```

- Allows the grantee.
- User to which privilege is granted to in turn grant object privilege to other users.

REVOKE – Revoke Privileges

- **Revoking** privileges means to deny (decline) permission to user given previously.
- The owner on an object can **revoke** privileges granted to another user.
- A user of the object, who is not an owner, but has been granted privileges can be revoked.
- **Syntax;**

REVOKE ***Object_privileges***

User can revoke all or specific privileges owned by him/her. List of various privileges are as below;

1. **ALL**
2. **ALTER**
3. **DELETE**
4. **INDEX**
5. **INSERT**
6. **REFERENCES**
7. **SELECT**
8. **UPDATE**

REVOKE – Revoke Privileges (Conti…)

- **Revoking** privileges means to deny (decline) permission to user given previously.
- The owner on an object can **revoke** privileges granted to another user.
- A user of the object, who is not an owner, but has been granted privileges can be revoked.
- **Syntax;**

```
REVOKE    Object_privileges
ON        Object_Name
```

Name of the object on which object we want to revoke privileges.

REVOKE – Revoke Privileges (Conti…)

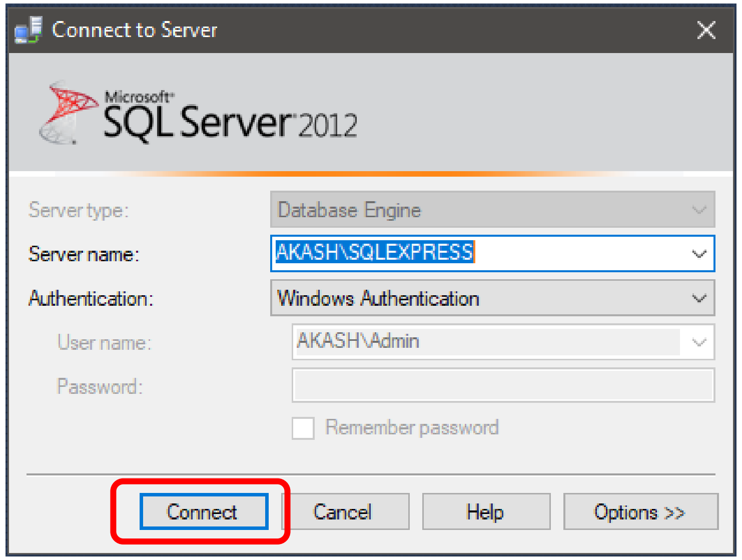
- **Revoking** privileges means to deny (decline) permission to user given previously.
- The owner on an object can **revoke** privileges granted to another user.
- A user of the object, who is not an owner, but has been granted privileges can be revoked.
- **Syntax;**

```
REVOKE      Object_privileges
ON          Object_Name
FROM        UserName
```

Name of the user from which we want to take privileges.

Example of Grant and Revoke

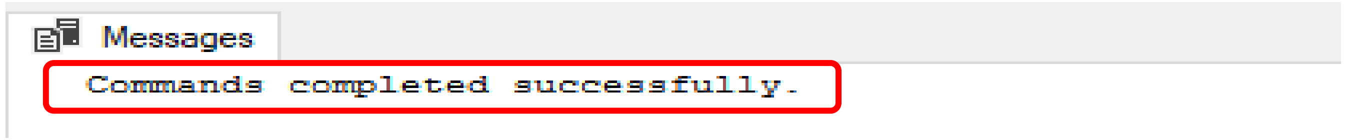
- Step 1: First of all connect to **SQL SERVER** with Default server;



Server type: Database Engine
Server name: AKASH\SQLEXPRESS
Authentication: Windows Authentication
User name: AKASH\Admin
Password:
Remember password

- Step 2: Create a new **login** with SQL server authentication and create **User**(Write following command to create a user).

```
CREATE LOGIN DBMS2 WITH PASSWORD = 'DBMS2' ;
CREATE USER TESTUSER FOR LOGIN DBMS2;
```

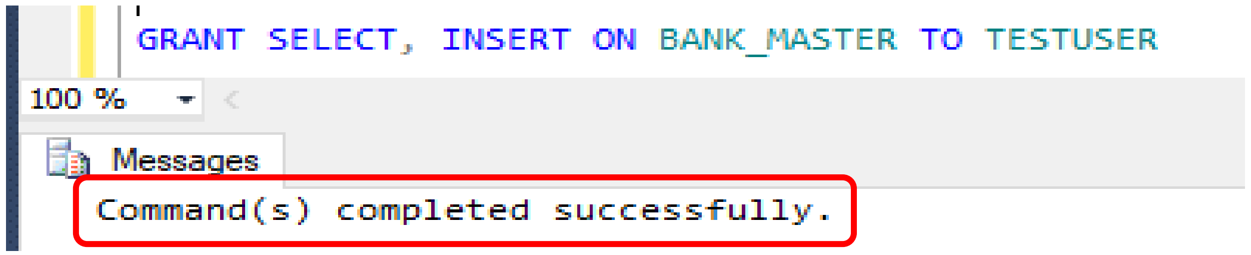


```
Commands completed successfully.
```

Example of Grant and Revoke (Conti…)

- Step 3: Now grant **SELECT,INSERT** to **TESTUSER** from admin connect;

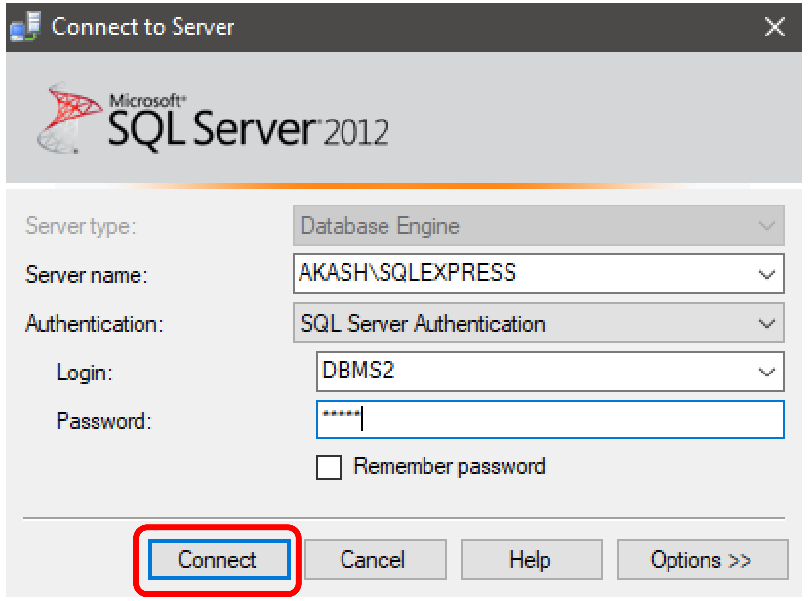
GRANT ***SELECT,INSERT*** **ON** ***bank_master*** **TO** ***TESTUSER;***



```
GRANT SELECT, INSERT ON BANK_MASTER TO TESTUSER
```

```
Command(s) completed successfully.
```

- Step 4: Disconnect from the SQL server admin and connect to TESTUSER SQL server authentication;

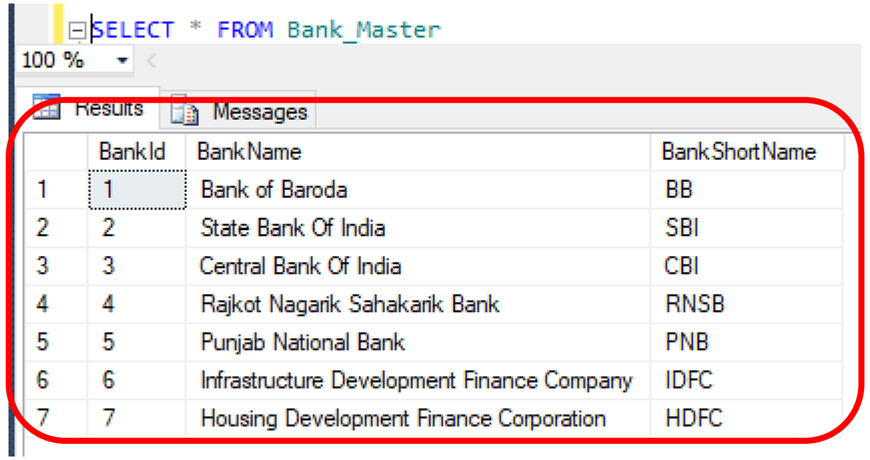


Server type: Database Engine
Server name: AKASH\SQLEXPRESS
Authentication: SQL Server Authentication
Login: DBMS2
Password: *****
☐ Remember password

Example of Grant and Revoke (Conti...)

- Step 5: Fetch the table bank_detail data from **DBMS2's** login;

SELECT * FROM bank_master ;

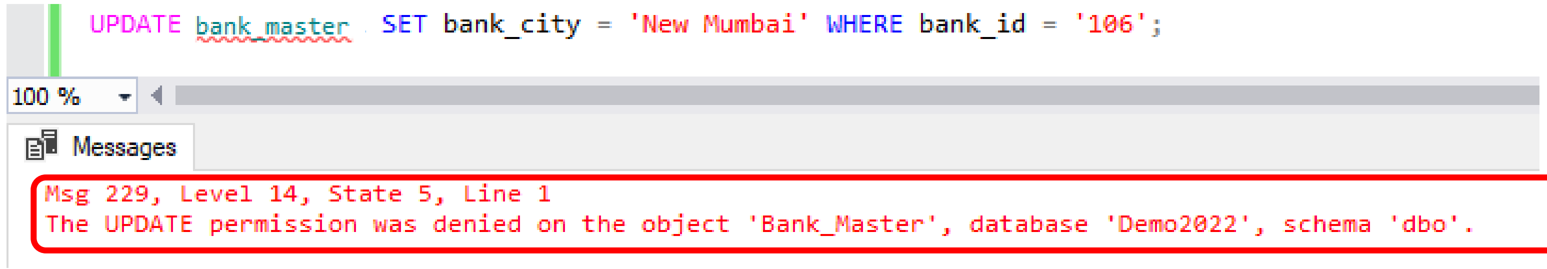


```
SELECT * FROM Bank_Master
```

	BankId	BankName	BankShortName
1	1	Bank of Baroda	BB
2	2	State Bank Of India	SBI
3	3	Central Bank Of India	CBI
4	4	Rajkot Nagarik Sahakarik Bank	RNSB
5	5	Punjab National Bank	PNB
6	6	Infrastructure Development Finance Company	IDFC
7	7	Housing Development Finance Corporation	HDFC

- Step 6: Now if TESTUSER try to **update** record for bank_detail;

UPDATE bank_master SET bank_city = 'RAJKOT' WHERE bank_id = '106';



```
UPDATE bank_master  SET bank_city = 'New Mumbai' WHERE bank_id = '106';
```

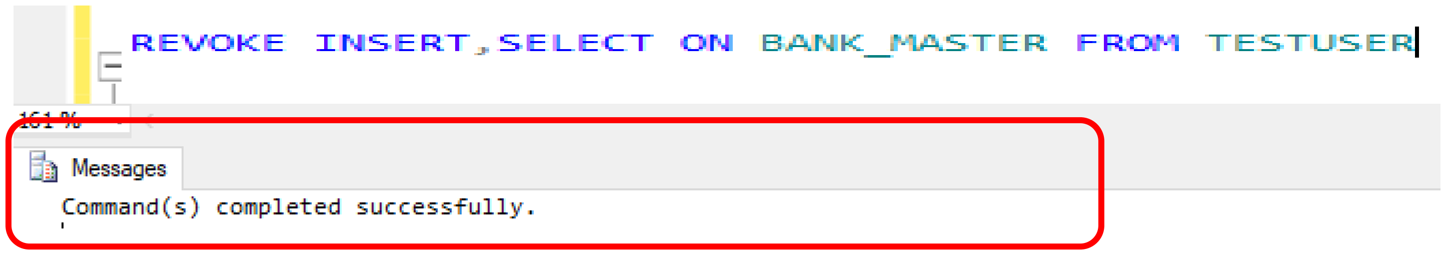
```
Msg 229, Level 14, State 5, Line 1
The UPDATE permission was denied on the object 'Bank_Master', database 'Demo2022', schema 'dbo'.
```

ERROR !!!

Example of Grant and Revoke (Conti…)

- Step 7: After that **disconnect** from the current SQL SERVER user and **Connect** to the main **administrative** server;
- Step 8: Now take **privileges** from user TESTUSER;

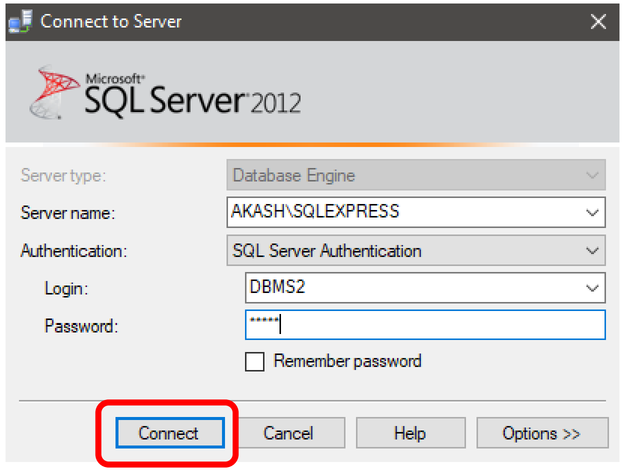
```
REVOKE INSERT,SELECT ON bank_master FROM TESTUSER;
```



```
REVOKE INSERT,SELECT ON BANK_MASTER FROM TESTUSER
```

```
Command(s) completed successfully.
```

- Step 9: **Disconnect** from the SQL Server admin and **connect** to DBMS2;



Connect to Server

Microsoft SQL Server 2012

Server type: Database Engine

Server name: AKASH\SQLEXPRESS

Authentication: SQL Server Authentication

Login: DBMS2

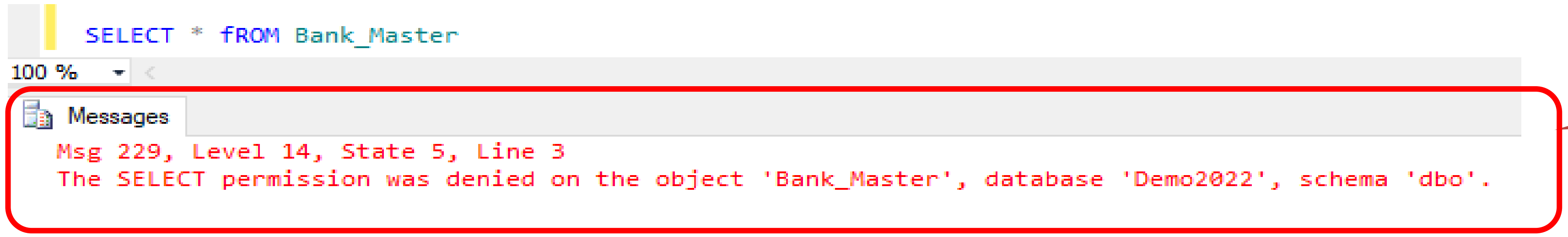
Password: *****

☐ Remember password

Example of Grant and Revoke (Conti…)

- Step 10: Now try to **fetch** the table bank_detail from DBMS2's login

SELECT * FROM bank_master ;



```
SELECT * FROM Bank_Master
```

```
Msg 229, Level 14, State 5, Line 3
The SELECT permission was denied on the object 'Bank_Master', database 'Demo2022', schema 'dbo'.
```

ERROR !!!

Thank You

Prof. Firoz A. Sherasiya
Computer Science & Engineering Department
Darshan University, Rajkot

firoz.sherasiya@darshan.ac.in
9879879861