

Lab 6: Join Operation

Introduction:

In SQL, JOIN operations are used to retrieve data from two or more related tables based on a related column between them. Joins help combine rows from multiple tables to form meaningful results that reflect relationships in the data. This is especially useful in relational databases where data is normalized into separate tables.

There are several types of joins in SQL, each serving a different purpose depending on how the tables are related and the desired result.

Types of Joins:

i) Left join:

- Returns all records from the left table, and the matched records from the right table.
- If no match is found, NULL values are returned from the right table.

ii) Right join:

- Returns all records from the right table, and the matched records from the left table.

iii) Inner join:

- Returns only rows that have matching values in both tables.
- Used when you want records that exist on both sides (e.g., students who have marks).

Tables Used:

- StudentDetails: Stores student information (StudentID, StudentName)
- SubjectDetails: Stores subject info (SubjectID, SubjectName)
- MarksheetsDetails: Stores student marks (RollNo, Subject_ID, Marks, Remarks)

-- SQL Source Code:

```
DROP TABLE IF EXISTS StudentDetails;
DROP TABLE IF EXISTS SubjectDetails;
DROP TABLE IF EXISTS MarksheetsDetails;
GO

-- StudentDetails Table
CREATE TABLE StudentDetails (
    StudentID INT PRIMARY KEY,
    StudentName VARCHAR(100)
);

-- SubjectDetails Table
CREATE TABLE SubjectDetails (
    SubjectID INT PRIMARY KEY,
    SubjectName VARCHAR(100)
);

-- MarksheetsDetails Table
CREATE TABLE MarksheetsDetails (
    RollNo INT,
    Subject_ID INT,
    Marks FLOAT,
    Remarks VARCHAR(100),
    PRIMARY KEY (RollNo, Subject_ID), -- Composite PK
    FOREIGN KEY (RollNo) REFERENCES StudentDetails(StudentID),
    FOREIGN KEY (Subject_ID) REFERENCES SubjectDetails(SubjectID)
);
```

```
-- Inserting data
INSERT INTO StudentDetails VALUES (1, 'Joe'), (2, 'Henry'), (3, 'Emma');
INSERT INTO SubjectDetails VALUES (101, 'Math'), (102, 'Biology'), (103, 'Physics');
INSERT INTO MarksheetsDetails VALUES
(1, 101, 85, 'Pass'),
(1, 102, 78, 'Pass'),
(2, 101, 90, 'Pass'),
(3, 103, 65, 'Fail');

-- Inner join
SELECT
    StudentDetails.StudentName,
    SubjectDetails.SubjectName,
    MarksheetsDetails.Marks
FROM MarksheetsDetails
JOIN StudentDetails ON StudentDetails.StudentID = MarksheetsDetails.RollNo
JOIN SubjectDetails ON SubjectDetails.SubjectID = MarksheetsDetails.Subject_ID;
```

	StudentName	SubjectName	Marks
1	Joe	Math	85
2	Joe	Biology	78
3	Henry	Math	90
4	Emma	Physics	65

Figure 1 Inner join

```
-- Left join
SELECT DISTINCT
    StudentDetails.StudentName,
    MarksheetsDetails.Rmarks
FROM MarksheetsDetails
LEFT JOIN StudentDetails ON StudentDetails.StudentID = MarksheetsDetails.Rollno;
```

	StudentName	Rmarks
1	Emma	Fail
2	Henry	Pass
3	Joe	Pass

Figure 2 Left join

```
-- Right join
SELECT DISTINCT
    SubjectDetails.SubjectID,
    MarksheetsDetails.Rmarks
FROM MarksheetsDetails
RIGHT JOIN SubjectDetails ON SubjectDetails.SubjectId = MarksheetsDetails.Subject_Id;
```

	SubjectID	Rmarks
1	101	Pass
2	102	Pass
3	103	Fail

Figure 3 Right join

Conclusion:

In this lab, we explored join operations which are used to combine data from multiple relational tables. We used different types of joins to understand how they fetch related data. These operations are critical for querying complex databases efficiently and meaningfully.

Lab 7: Querying Relational Tables with Foreign Keys

Introduction:

In this lab, we focused on retrieving and manipulating data from relational tables that are linked together using foreign keys. Foreign keys are used to maintain referential integrity between tables, ensuring that relationships remain consistent. This lab demonstrates the creation of relational tables with foreign key constraints and the execution of various SELECT queries to retrieve and analyze data.

We worked with multiple related tables and performed queries to:

- Creating related tables (Customer, Loan, Borrows) to enforce referential integrity.
- Inserting data into these tables.
- Filter results using WHERE clauses.
- Demonstrate how foreign keys ensure that only valid references exist between tables.
- Using SELECT queries with filtering, ordering, joins, and aggregate functions to retrieve meaningful information.

Step 1: Creating Tables with Foreign Keys

Here we create three relational tables:

1. Customer – Stores customer details.
2. Loan – Stores loan details such as type and amount.
3. Borrows – A linking table that records which customer has taken which loan.

--SQL query

```
drop table if exists customer;
drop table if exists borrows;
drop table if exists loan;

create table customer
(
    customerid int primary key,
    customername varchar(100),
    address varchar(100),
    phone bigint,
    email varchar(100)
)

create table loan
(
    loannumber bigint primary key,
    loantype varchar(50),
    amount float
)

create table borrows
(
    customerid int not null,
    loannumber bigint not null,
    PRIMARY KEY (customerid, loannumber),
    foreign key (customerid) references customer(customerid),
    foreign key (loannumber) references loan(loannumber)
)
```

Step 2: Inserting data into the tables

--SQL query

```
insert into customer (customerid, customername, address, phone, email) values
(100, 'Joe', 'Lalitpur', 9875641245, 'joe@gmail.com'),
(101, 'Henry', 'Anamnagar', 9645641249, 'henry@gmail.com'),
(102, 'Emma', 'New baneshor', 9848765423, 'emma@gmail.com'),
(103, 'Rose', 'Thapagaun', 9145697845, 'rose@gmail.com'),
(104, 'Jona', 'Lalitpur', 9800456978, 'jona@gmail.com'),
(105, 'Ben', 'Kalanki', 9645897825, 'ben@gmail.com'),
(106, 'Mona', 'Lalitpur', 9245789615, 'mona@gmail.com');

insert into loan (loannumber, loantype, amount) values
(1001, 'Home Loan', 2500000.00),
(1002, 'Car Loan', 1500000.00),
(1003, 'Personal Loan', 75000.00);

insert into borrows (customerid, loannumber) values
(100, 1001),
(100, 1002),
(101, 1002),
(102, 1003),
(103, 1001),
(106, 1003);
```

Step 3: Running queries

- 3.1. Displaying customers from Lalitpur in ascending order

```
--customer from Lalipur
select c.customername
from customer as c
where c.address='Lalitpur'
order by c.customername asc;
```

	customername
1	Joe
2	Jona
3	Mona

Figure 4: Costumers from Lalitpur

- 3.2. Counting the total number of customers having atleast one loan

```
--Number of customers with loan
select count(distinct customer.customerid) as num_of_customer_with_loan
from customer
join borrows on customer.customerid = borrows.customerid;
```

	num_of_customer_with_loan
1	5

Figure 5: Total customer who took loan

- 3.3. Displaying customers with loan higher than 50000

```
--customer with loan higher than 50000
select distinct c.customername
from customer c
join borrows b on c.customerid = b.customerid
join loan l on l.loannumber = b.loannumber
where l.amount >= 50000;
```

	customername
1	Emma
2	Henry
3	Joe
4	Mona
5	Rose

Figure 6: Customers with loan more than 50,000

- 3.4. Displaying average loan for each loan types

```
--avg loan
select loantype, avg(amount) as average_amount
from loan
group by loantype;
```

	loantype	average_amount
1	Car Loan	1500000
2	Home Loan	2500000
3	Personal ...	75000

Figure 7: Average loan taken

Conclusion:

In this lab, we successfully designed and implemented three relational tables with foreign key constraints to maintain referential integrity. We inserted realistic sample data and executed various SELECT queries using joins to combine related data, applied filters using WHERE, sorted results with ORDER BY, and summarized information with aggregate functions such as COUNT(). This lab reinforced the importance of foreign keys in relational database design and demonstrated how they enable powerful, meaningful queries across multiple tables.

Lab 8: Creating and Executing Stored Procedures with Dynamic Table Operations

Introduction:

In this lab, we explore stored procedures in SQL Server, focusing on the creation and execution of procedures that perform dynamic table operations. Stored procedures are precompiled SQL statements stored in the database, which can accept parameters, perform actions, and return results. Dynamic table operations allow a stored procedure to interact with varying tables or columns determined at runtime. This is useful for tasks such as creating, altering, inserting into, or retrieving data from tables whose names are passed as parameters. This lab demonstrates the creation of a stored procedure that dynamically manages table structures and performs data operations based on foreign key relationships.

Objectives:

- Learn how to create stored procedures in SQL Server.
- Understand dynamic SQL and its integration into stored procedures.
- Perform create, insert, and select operations dynamically.
- Maintaining referential integrity in procedural contexts.

Step 1: Creating a table

First, we establish the foundation by creating a students table that will serve as the parent table for our foreign key relationships.

--SQL Query

```
--Creating a table
Drop table if exists studentexamdetails
DROP TABLE IF EXISTS students;
CREATE TABLE students (
    studentid INT NOT NULL PRIMARY KEY,
    batchid INT
);

--Inserting data
INSERT INTO students(studentid, batchid) VALUES
(100, 2080),
(101, 2080),
(102, 2080),
(103, 2080),
(104, 2081),
(105, 2082),
(106, 2083);

select * from students
```

	studentid	batchid
1	100	2080
2	101	2080
3	102	2080
4	103	2080
5	104	2081
6	105	2082
7	106	2083

Figure 8: Studentid and batchid

Step 2: Creating the Stored Procedure

We create a stored procedure named 'ABC' that performs dynamic table operations and conditional data insertion.

--SQL Query

```
--Creating Stored Procedure
CREATE PROCEDURE ABC
AS
BEGIN
DROP TABLE IF EXISTS studentexamdetails;
-- Create the new table
CREATE TABLE studentexamdetails (
    Sno INT PRIMARY KEY IDENTITY(1,1),
    studentid INT,
    remarks VARCHAR(50),
    FOREIGN KEY (studentid) REFERENCES students(studentid)
);

-- Insert studentid and remarks into studentexamdetails
INSERT INTO studentexamdetails (studentid, remarks)
SELECT studentid, 'pass'
FROM students
WHERE batchid = '2080';
END;
GO
```

Step 3: Executing the Stored Procedure

```
--Excuting the procedure
exec ABC
select * from studentexamdetails
```

	Sno	studentid	remarks
1	1	100	pass
2	2	101	pass
3	3	102	pass
4	4	103	pass

Figure 9: Student from batchid 2080

Conclusion:

In this lab, we created a students table and populated it with sample data, then developed the ABC stored procedure to dynamically create the studentexamdetails table with a foreign key relationship to students. The procedure inserted only those students from batch 2080 with the remark 'pass', demonstrating conditional data insertion. Executing the procedure successfully generated the table and populated it with filtered results, reinforcing concepts of stored procedure creation, dynamic table operations, and referential integrity in SQL Server.