Lab 1: Installation of MSSQL(structured query language)

Introduction

MSSQL is one of the most widely used open-source relational database management systems (RDBMS). It uses Structured Query Language (SQL) to manage and manipulate data efficiently. The purpose of this lab is to install MSSQL on the system and become familiar with the basic setup required to run and manage a database. Understanding how to install and configure MSSQL is the first step in learning database management and working with SQL commands.

Steps while downloading and installing MSSQL

Step1: Download MSsql installer from community downloads choosing developer option.

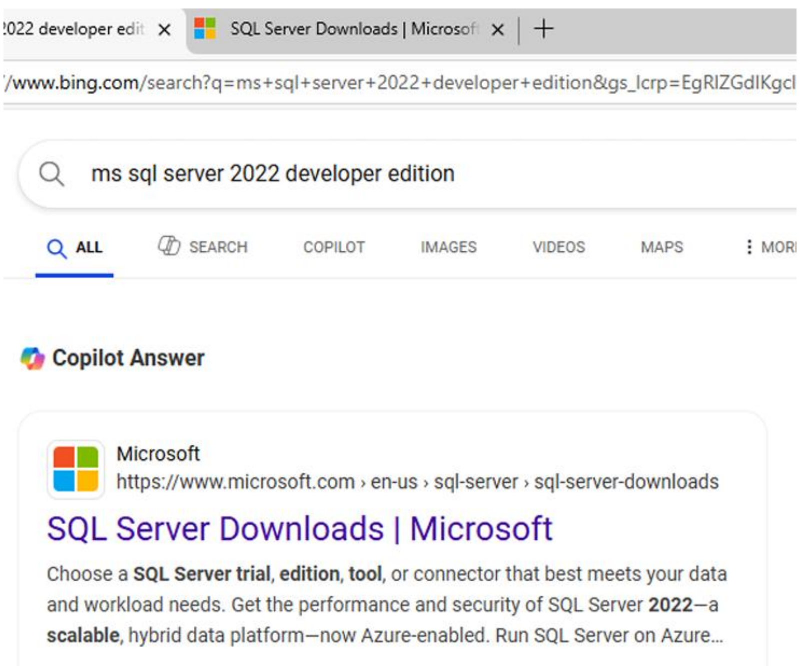


Fig 1.1: Bing search result for "MS SQL Server 2022 Developer Edition" showing the official Microsoft download link.

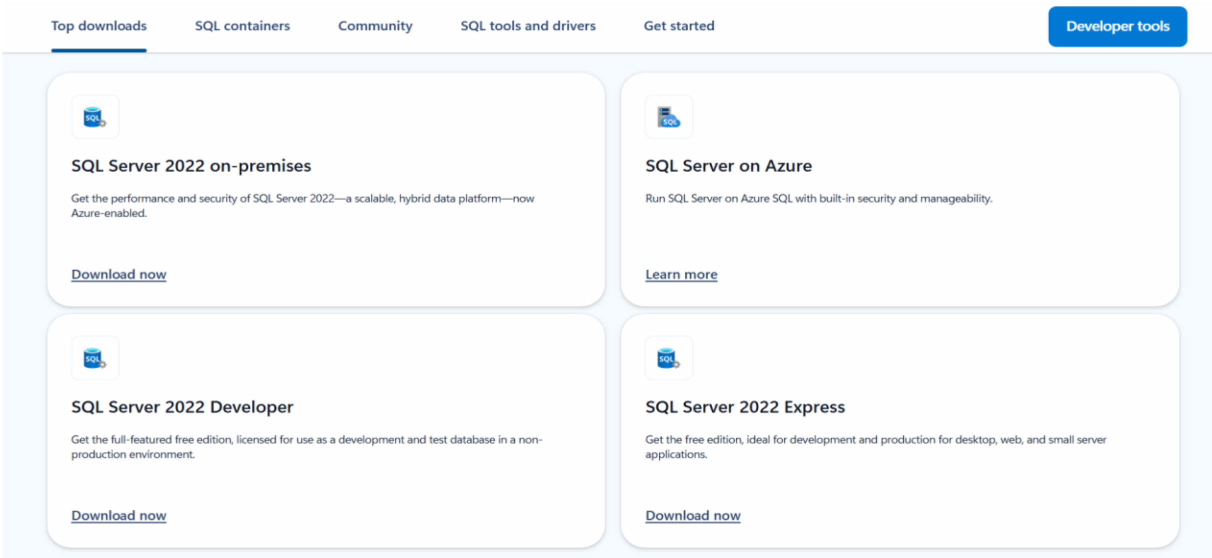


Fig1.2: The figure shows various SQL Server 2022 editions available for download, including On-premises, Developer, Express, and Azure versions from the official Microsoft website.

Step2: Download and install the basic option in from developer edition.

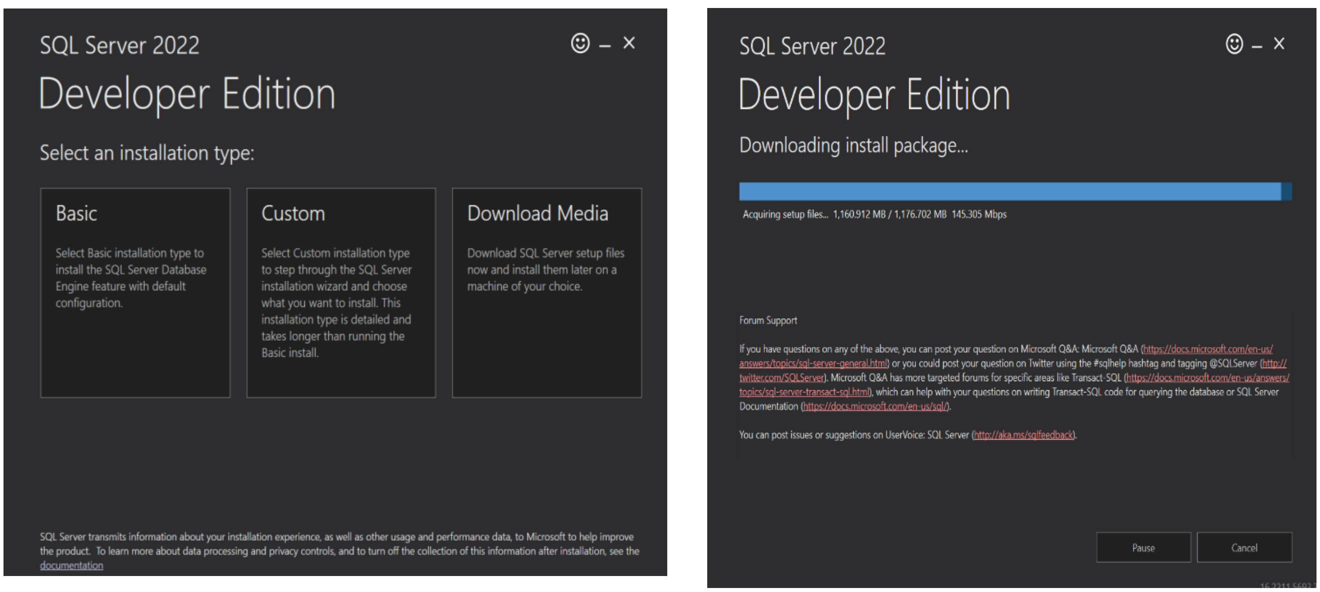
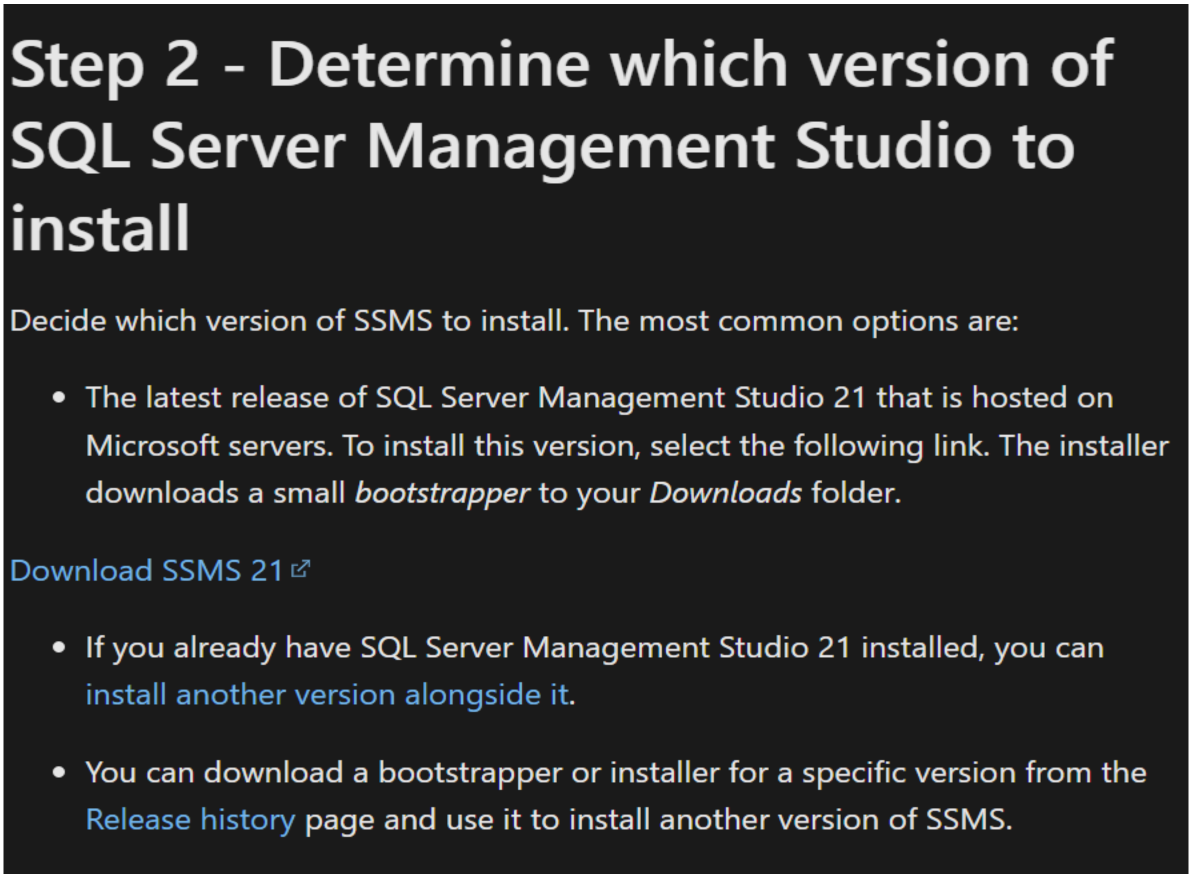


Fig 1.3: The figure displays the installation screen for SQL Server 2022 Developer Edition, offering three options: Basic, Custom, and Download Media.

Step 3: Install ssms(SQL Server Management Studio).



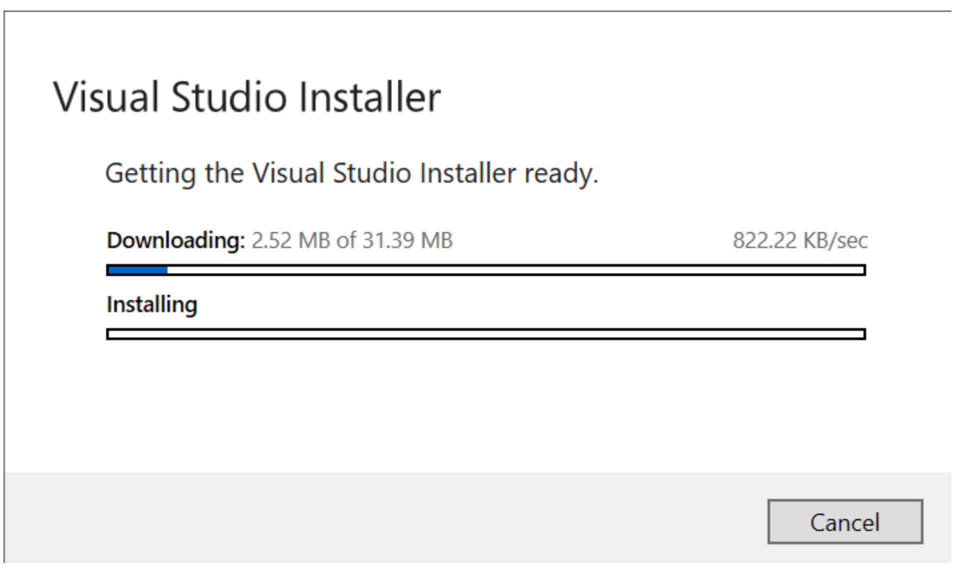


Fig 1.4: This step shows how to choose and download the right version of SQL Server Management Studio, with a link to get the latest SSMS 21 and installing it.

Step4: Download and install SQL server management and launch it.

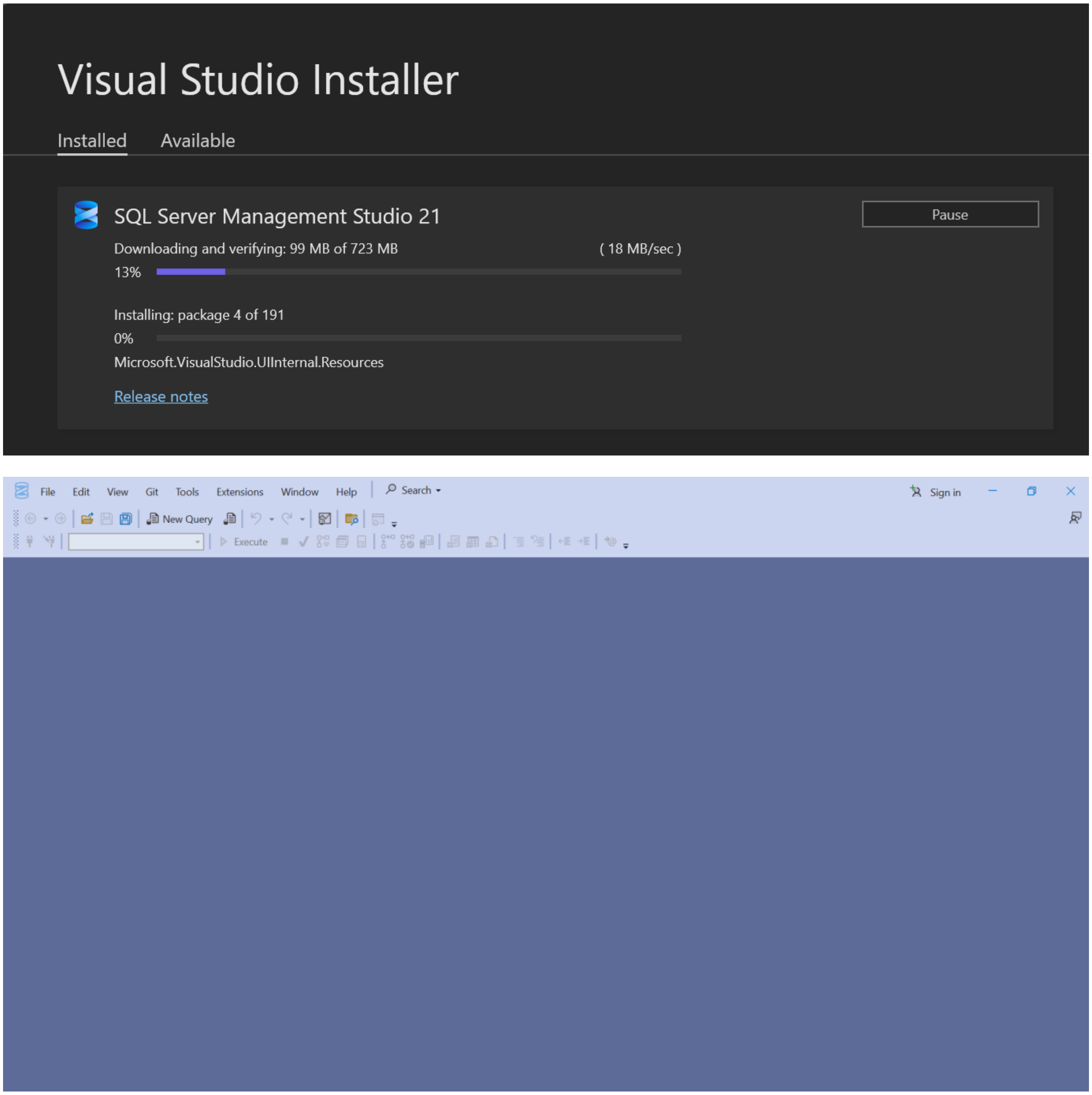


Fig 1.5: The Visual Studio Installer is downloading and installing SQL Server Management Studio 21 and The SSMS 21 interface is open, showing the main workspace with toolbar options like File, Edit, View, and New Query.

Conclusion

In this lab, we successfully installed MSSQL and ensured that it runs properly on our system. This setup provides the foundation for practicing SQL commands and developing database-driven applications. By completing this lab, we are now ready to explore the core functionalities of MSSQL and perform operations such as creating tables, inserting data, and running queries.

Lab 2: Creating a Database and tables using query

Introduction

In this lab, we learn how to create a **database** and define **tables** using SQL queries. Understanding how to structure a database is essential for organizing data efficiently. Using the CREATE DATABASE and CREATE TABLE commands, we can design a relational database that suits the needs of any application. This lab focuses on writing basic SQL commands to set up a database and its tables from scratch.

Creating a database and connecting to local server

Step1: Establish a connection to a server name the server local inside the parenthesis.

Step2: Right click and click create a new database and name the database lab 1.

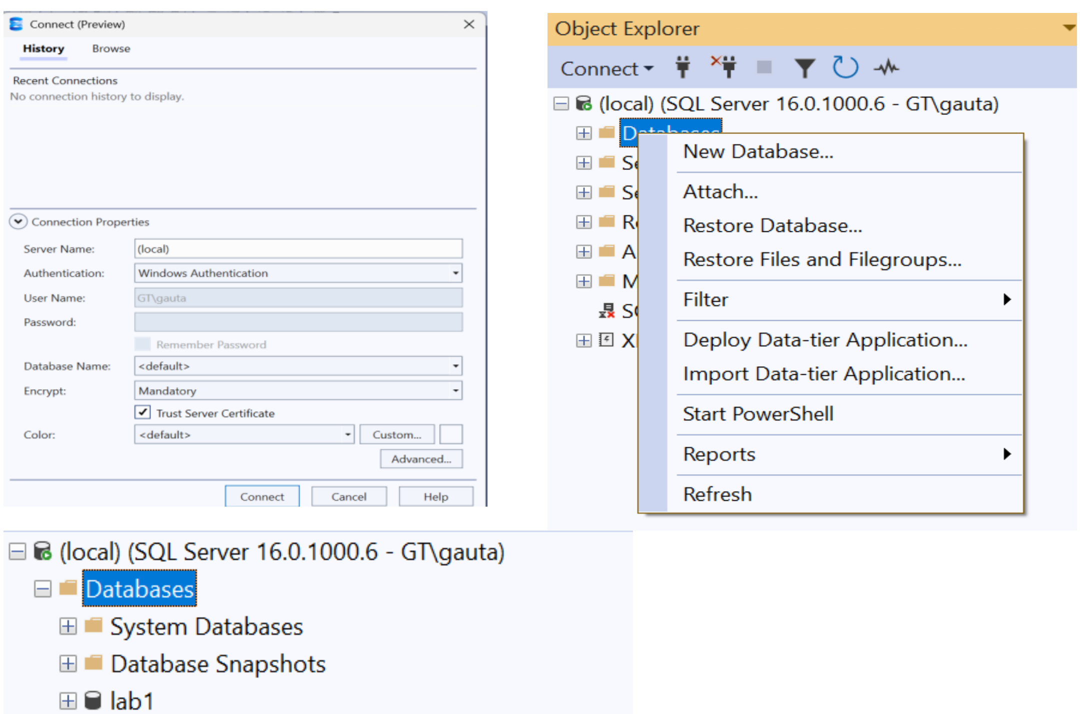


fig2.1: The images show connecting to a local SQL Server in SSMS, accessing the database options, and viewing the existing "lab1" database.

Creating a table for relational database.

Steps for creating tables and showing its diagrammatic view

Step 1: Right click and click create new query.

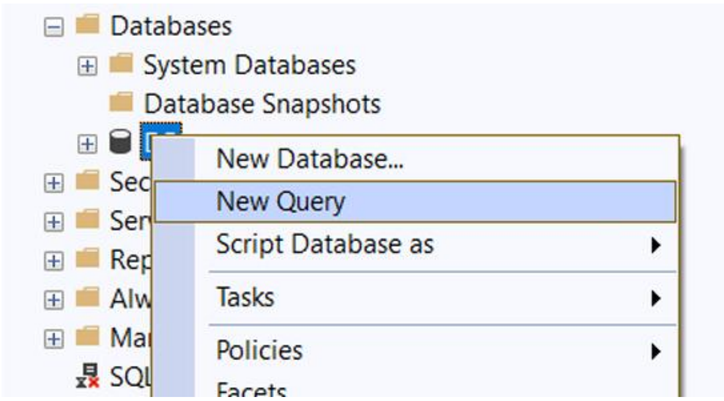


fig2.2: The image shows the context menu in SSMS for starting a **New Query** or creating a **New Database** under the "Databases" section.

Step2: Write the given query given below and execute the code.

```
CREATE TABLE StudentDetails

(

Rollno Int,

StudentName Varchar(100)

)

CREATE TABLE SubjectDetails

(

SubjectID INT,

SubjectName VARCHAR(100)

)

CREATE TABLE MarksheetDetails

(

Rollno INT,

SubjectID INT,

Marks INT)
```

Step3: Right click and click new database diagram and add the necessary tables.

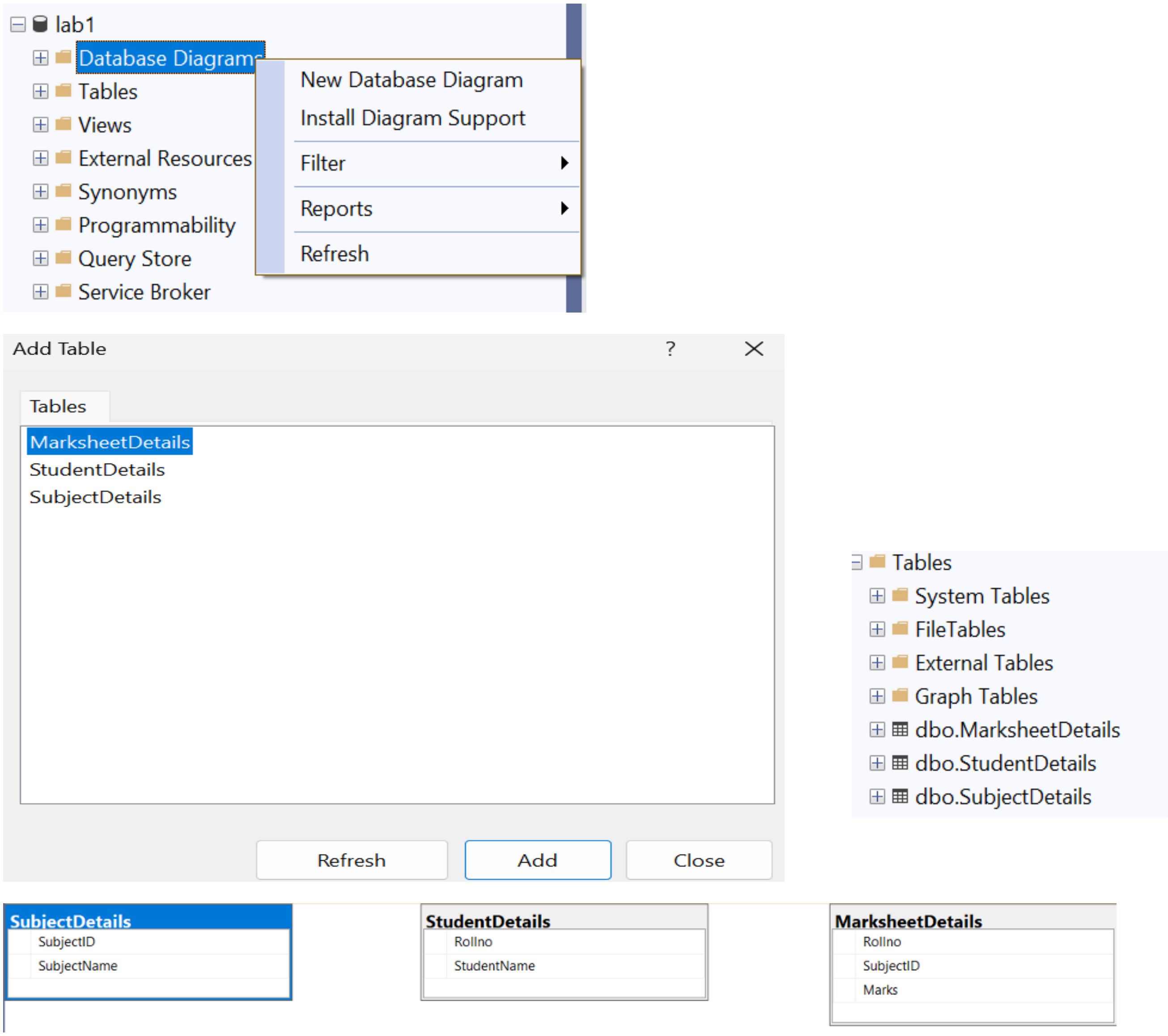


Fig 2.3: These images show the creation of a **Database Diagram** in SSMS using tables: `StudentDetails`, `SubjectDetails`, and `MarksheetDetails`. It visualizes the table structures and their columns, helping understand the relationships between student records, subjects, and marks.

Conclusion

Through this lab, we successfully created a database and multiple tables using SQL queries. This practical experience helped us understand how to define table structures with appropriate data types and constraints. Learning to create databases and tables manually provides a solid foundation for managing real-world data in relational database systems.

Madan Bhandari Memorial College

Lab 3: Showing Keys and various functions of DDL.

Keys in SQL

Keys in SQL are used to uniquely identify rows in a table and to create relationships between tables. They help maintain data accuracy and integrity.

Common types of keys:

- **Primary Key:** Uniquely identifies each record in a table. It cannot be NULL or duplicated.
- **Foreign Key:** A field in one table that refers to the **primary key** in another table. It creates a link between two tables.
- **Candidate Key:** A column (or set of columns) that can uniquely identify rows in a table. One candidate key becomes the **primary key**.
- **Composite Key:** A key made up of two or more columns used together to uniquely identify a record.
- **Unique Key:** Ensures all values in a column are different, like a primary key, but it can accept a NULL value.

Database Language in SQL

SQL (Structured Query Language) includes different types of commands grouped as **database languages**. These are used to create, manage, and manipulate data in a database.

Types of Database Languages:

1. **DDL (Data Definition Language)** – Defines the structure of the database.
   Commands: CREATE, ALTER, DROP, TRUNCATE
2. **DML (Data Manipulation Language)** – Deals with data manipulation.
   Commands: SELECT, INSERT, UPDATE, DELETE
3. **DCL (Data Control Language)** – Controls access to data.
   Commands: GRANT, REVOKE
4. **TCL (Transaction Control Language)** – Manages transactions in the database.
   Commands: COMMIT, ROLLBACK, SAVEPOINT

Some Functions in DDL

CREATE(): Used to create a new table or database.

ALTER(): Used to change the structure of an existing table (e.g., add or remove columns).

DROP(): Used to delete a table or database completely.

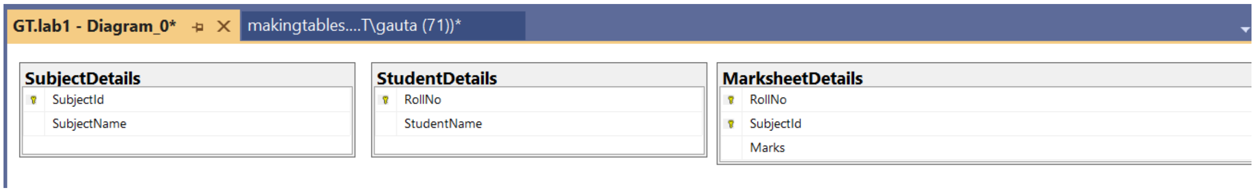
Steps for showing keys and DDL language in a relational database

Step1: Create a new query and write the following code which includes primary key and composite primary key and DDL functions then executing it.

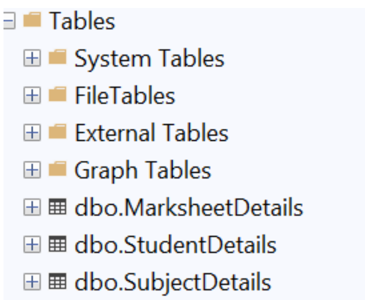
```
-- Dropping table if it exists

DROP TABLE IF EXISTS StudentDetails;
DROP TABLE IF EXISTS SubjectDetails;
DROP TABLE IF EXISTS MarksheetDetails;
GO

--CreatingStudentDetailstable

CREATE TABLE StudentDetails

(

    RollNo INT NOT NULL,

    StudentName VARCHAR(100),

    PRIMARY KEY (RollNo)

);

-- Creating SubjectDetails table

CREATE TABLE SubjectDetails

(

    SubjectId INT NOT NULL,

    SubjectName VARCHAR(100),

    PRIMARY KEY (SubjectId)

);

-- Creating MarksheetDetails table

CREATE TABLE MarksheetDetails

(

    RollNo INT NOT NULL,

    SubjectId INT NOT NULL,

    Marks INT,
```

```sql
FOREIGN KEY (RollNo) REFERENCES StudentDetails(RollNo),
FOREIGN KEY (SubjectId) REFERENCES SubjectDetails(SubjectId)

);
```

Step2: Showing the diagrammatic view of relational tables including the database tables.



`Fig 3.1:` The image shows a database diagram in SSMS with three tables—`StudentDetails`, `SubjectDetails`, and `MarksheetDetails`—visualizing keys and relationships.



`Fig 3.2:` The image shows three user-created tables—`MarksheetDetails`, `StudentDetails`, and `SubjectDetails`—listed under the "Tables" section in SSMS.

Conclusion

In this lab, we applied various DDL commands to create, alter, and drop database objects, and we learned how keys are used to uniquely identify and relate records in tables. This hands-on practice helped us understand the structural foundation of relational databases and how to manage them effectively using SQL.

Lab 4: Insertion, deletion and update of Data into Relational Tables Using Queries with Foreign Keys

Introduction:

In this lab, we focus on **DML (Data Manipulation Language)** commands, which are used to manage the data within relational database tables. These commands do not modify the table structure but allow users to:

- **SELECT** – Retrieve data from tables.
- **INSERT** – Add new records into a table.
- **UPDATE** – Modify existing records.
- **DELETE** – Remove records from a table.

The lab demonstrates the use of the **INSERT** and **SELECT** commands to insert and retrieve student-related data from multiple interrelated tables, while ensuring **referential integrity** using **foreign keys**.

Step 1: Creating Tables with Foreign Keys

Three relational tables are created using SQL Server Management Studio (SSMS):

1. StudentDetails – Stores student information.
2. SubjectDetails – Stores subject information.
3. MarksheetDetails – Stores marks obtained by students, linked to the other two tables using foreign keys.

This design enforces relational integrity and ensures that entries in MarksheetDetails cannot exist unless corresponding records exist in StudentDetails and SubjectDetails.

SQL Source Code:

```
-- Dropping tables if they exist
DROP TABLE IF EXISTS StudentDetails;
DROP TABLE IF EXISTS SubjectDetails;
DROP TABLE IF EXISTS MarksheetDetails;
GO

-- Creating StudentDetails table
CREATE TABLE StudentDetails (
   RollNo INT NOT NULL,
   StudentName VARCHAR(100),
```

```
    PRIMARY KEY (RollNo)
);

-- Creating SubjectDetails table
CREATE TABLE SubjectDetails (
    SubjectId INT NOT NULL,
    SubjectName VARCHAR(100),
    PRIMARY KEY (SubjectId)
);

-- Creating MarksheetDetails table
CREATE TABLE MarksheetDetails (
    RollNo INT NOT NULL,
    SubjectId INT NOT NULL,
    Marks FLOAT,
    FOREIGN KEY (RollNo) REFERENCES StudentDetails(RollNo),
    FOREIGN KEY (SubjectId) REFERENCES SubjectDetails(SubjectId)
);
```

Inserting Data into Tables:

```
-- Inserting students
INSERT INTO StudentDetails (RollNo, StudentName) VALUES
(1, 'SANJOG'),
(2, 'KUSHAL'),
(3, 'KHEWANG');

-- Inserting subjects
INSERT INTO SubjectDetails (SubjectId, SubjectName) VALUES
(101, 'MATHS'),
(102, 'C++'),
(103, 'DSA');

-- Inserting marks
INSERT INTO MarksheetDetails (RollNo, SubjectId, Marks) VALUES
(1, 101, 88),
(1, 102, 78),
(1, 103, 92),
(2, 101, 82),
(2, 102, 88),
(3, 101, 90),
(3, 103, 86);
```

Step 2: Retrieving Data Using SELECT Queries

After inserting the data, we use **SELECT** queries to verify and view the inserted records.

SQL Source Code:

```
SELECT * FROM MarksheetDetails;
SELECT * FROM SubjectDetails;
SELECT * FROM StudentDetails;
```

91 % No issues found Ln: 3 Ch: 28 TABS CRLF

Results Messages

	RollNo	SubjectId	Marks
1	1	101	88
2	1	102	78
3	1	103	92
4	2	101	82
5	2	102	88
6	3	101	90
7	3	103	86

	SubjectId	SubjectName
1	101	MATHS
2	102	C++
3	103	DSA

	RollNo	StudentName
1	1	SANJOG
2	2	KUSHAL
3	3	KHEWANG

Figure Description:

Fig 4.1: The figure displays the output of SQL `SELECT` queries showing the contents of the StudentDetails, SubjectDetails, and MarksheetDetails tables after inserting data. It confirms that referential integrity is maintained across the related tables using foreign key constraints.

Step3: To perform an update operation in a database table using the `UPDATE` SQL command.

SQL Source Code:

```sql
--update
UPDATE StudentDetails
SET StudentName = 'Sanjog GT'
WHERE RollNo = 1;
select * from StudentDetails
```

90 % No issues found

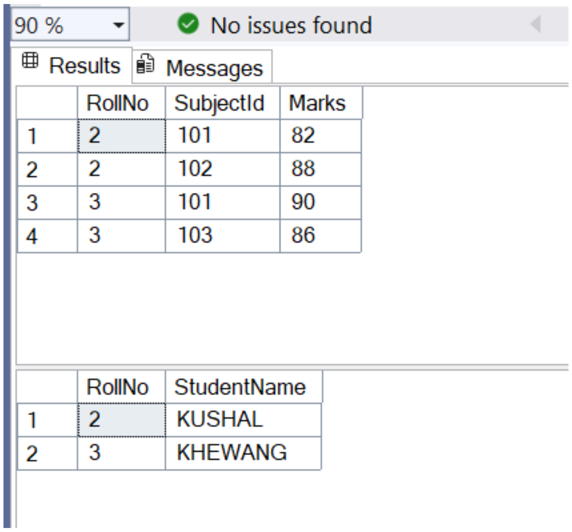
Results Messages

	RollNo	StudentName
1	1	Sanjog GT
2	2	KUSHAL
3	3	KHEWANG

Fig 4.2: The output shows that the `StudentName` for `RollNo = 1` has been successfully updated to **Sanjog GT**, while other records remain unchanged.

Step4: To delete a row of data with primary detail of RollNo=1.

```sql
--delete
delete from MarksheetDetails
where RollNo=1;
select * from MarksheetDetails;
--primary key is refrenced as foreign key in
delete from StudentDetails
where RollNo=1;

select * from StudentDetails;
```



	RollNo	SubjectId	Marks
1	2	101	82
2	2	102	88
3	3	101	90
4	3	103	86

	RollNo	StudentName
1	2	KUSHAL
2	3	KHEWANG

Fig 4.3: The output shows that the record with `RollNo = 1` has been successfully deleted from both `MarksheetDetails` and `StudentDetails`. The remaining data confirms that the delete operation worked as expected without violating any foreign key constraints.

Similarly

Step 1: Creating Tables with Foreign Keys

Three relational tables are created in SQL Server Management Studio (SSMS):

1. **ProductDetails** – Stores product information including name, price, discount, and description.
2. **CustomerDetails** – Stores customer information such as name, email, mobile, DOB, and address.
3. **TransactionDetails** – Stores transaction records, linked to `ProductDetails` and `CustomerDetails` using foreign keys to maintain relational integrity.

This design ensures that every transaction must reference existing customer and product records, thus enforcing referential integrity.

SQL Source Code:

```sql
-- Drop in correct dependency order
DROP TABLE IF EXISTS TransactionDetails;
DROP TABLE IF EXISTS ProductDetails;
DROP TABLE IF EXISTS CustomerDetails;

-- Create tables
CREATE TABLE ProductDetails (
    ProductID INT NOT NULL,
    ProductName VARCHAR(100),
    Price FLOAT,
    Discount FLOAT,
    Descriptionproduct varchar(100),
```

```sql
    PRIMARY KEY (ProductID)
);

CREATE TABLE CustomerDetails (
    CustomerID INT NOT NULL,
    CustomerName VARCHAR(100),
    Email VARCHAR(100),
    Mobile BIGINT,
    DOB DATE,
    Address VARCHAR(100),
    PRIMARY KEY (CustomerID)
);

CREATE TABLE TransactionDetails (
    TransactionID INT NOT NULL,
    Traxndatetime DATETIME,
    CustomerID INT,
    ProductID INT,
    Paymentstatus CHAR,
    PRIMARY KEY (TransactionID),
    FOREIGN KEY (CustomerID) REFERENCES CustomerDetails(CustomerID),
    FOREIGN KEY (ProductID) REFERENCES ProductDetails(ProductID)
);
```

Step 2: Inserting Data into Tables

Insert sample records to validate table structure and relationships.

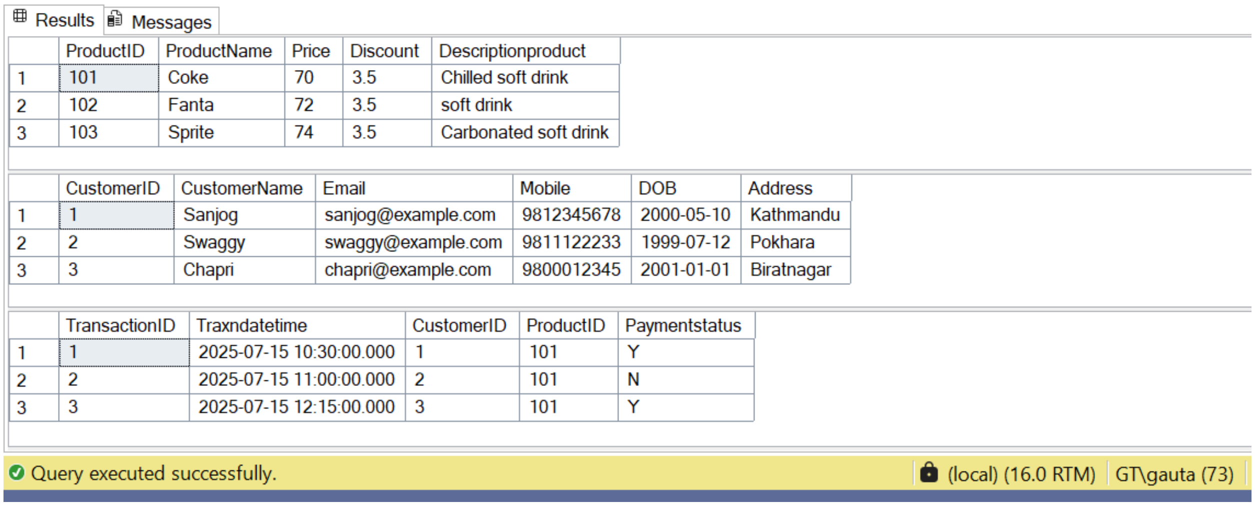
```sql
INSERT INTO ProductDetails(ProductID, ProductName, Price, Discount,
Descriptionproduct) values
(101,'Coke',70,3.5,'Chilled soft drink'),
(102,'Fanta',72,3.5,'soft drink'),
(103,'Sprite',74,3.5,'Carbonated soft drink');

INSERT INTO CustomerDetails (CustomerID, CustomerName, Email, Mobile, DOB, Address)
VALUES
(1, 'Sanjog', 'sanjog@example.com', 9812345678, '2000-05-10', 'Kathmandu'),
(2, 'Swaggy', 'swaggy@example.com', 9811122233, '1999-07-12', 'Pokhara'),
(3, 'Chapri', 'chapri@example.com', 9800012345, '2001-01-01', 'Biratnagar');

INSERT INTO TransactionDetails (TransactionID, Traxndatetime, CustomerID, ProductID,
Paymentstatus) VALUES
(1, '2025-07-15 10:30:00', 1, 101, 'Y'),
(2, '2025-07-15 11:00:00', 2, 101, 'N'),
(3, '2025-07-15 12:15:00', 3, 101, 'Y');
```

Step 3: Retrieving Data Using SELECT Queries

Verify the inserted data by querying all records from each table.

Sql code:

```sql
SELECT * FROM ProductDetails;
SELECT * FROM CustomerDetails;
SELECT * FROM TransactionDetails;
```



Results | Messages

	ProductID	ProductName	Price	Discount	Descriptionproduct
1	101	Coke	70	3.5	Chilled soft drink
2	102	Fanta	72	3.5	soft drink
3	103	Sprite	74	3.5	Carbonated soft drink

	CustomerID	CustomerName	Email	Mobile	DOB	Address
1	1	Sanjog	sanjog@example.com	9812345678	2000-05-10	Kathmandu
2	2	Swaggy	swaggy@example.com	9811122233	1999-07-12	Pokhara
3	3	Chapri	chapri@example.com	9800012345	2001-01-01	Biratnagar

	TransactionID	Traxndatetime	CustomerID	ProductID	Paymentstatus
1	1	2025-07-15 10:30:00.000	1	101	Y
2	2	2025-07-15 11:00:00.000	2	101	N
3	3	2025-07-15 12:15:00.000	3	101	Y

Query executed successfully. | (local) (16.0 RTM) | GT\gauta (73)

`Fig 4.4:` Displays the output of SQL `SELECT` queries showing the contents of `ProductDetails`, `CustomerDetails`, and `TransactionDetails` tables after inserting data. It confirms that referential integrity is maintained by foreign key constraints linking transactions to valid customers and products.

Step 4: Joining three tables for each valid transaction, who bought what and whether they paid**.**

Sql code:

```
SELECT
CustomerDetails.CustomerName,
ProductDetails.ProductName,
TransactionDetails.Paymentstatus
from TransactionDetails
join CustomerDetails on CustomerDetails.CustomerID=TransactionDetails.CustomerID
join ProductDetails on ProductDetails.ProductID=TransactionDetails.ProductID
```

An **alias** is a temporary name given to a table or column in a SQL query.
It exists **only for the duration of that query** and does **not** change the name in the database.

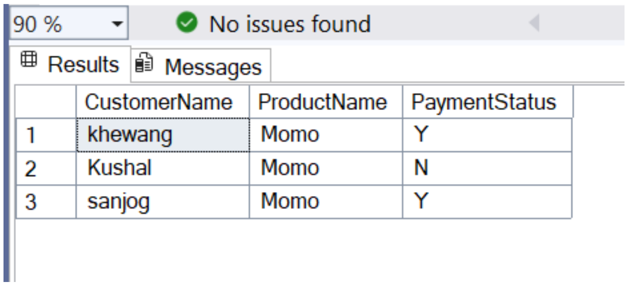
Aliases are mainly used to:

- Make queries shorter and easier to read.
- Avoid ambiguity when multiple tables have the same column names.
- Give more meaningful or user-friendly names to columns in the output.

Sql code:

```
SELECT
CD.CustomerName,
PD.ProductName,
TD.PaymentStatus
FROM TransactionDetails AS TD
JOIN CustomerDetails AS CD ON  CD.CustomerID=TD.CustomerID
```

```sql
JOIN ProductDetails AS PD ON PD.ProductID=TD.ProductID
```



	CustomerName	ProductName	PaymentStatus
1	khewang	Momo	Y
2	Kushal	Momo	N
3	sanjog	Momo	Y

Fig 4.5: The figure shows a query result listing customers, the product “Momo” they purchased, and whether they paid (`Y` = Yes, `N` = No).

Conclusion:

This lab demonstrated the practical use of **DML commands** (`INSERT`, `SELECT`, `UPDATE`, and `DELETE`) in a relational database. By designing tables with **foreign key constraints**, we enforced **referential integrity** and ensured that all operations followed relational rules. The results verified the proper insertion, modification, and deletion of records while maintaining data consistency across the tables.

Lab 5: Querying Relational Tables with Foreign Keys

Introduction

This lab demonstrates the creation of relational tables with **foreign key constraints** and the execution of various **SELECT queries** to retrieve and analyze data. The lab focuses on:

- Creating related tables (`customer`, `loan`, `borrows`) to enforce **referential integrity**.
- Inserting data into these tables.
- Using **SELECT** queries with filtering, ordering, joins, and aggregate functions to retrieve meaningful information.

Step 1: Creating Tables with Foreign Keys

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 code:
drop table if exists customer;
drop table if exists borrows;
drop table if exists loan;

create table customer
(
customerid int not null,
customername varchar(100),
address varchar(100),
phone bigint,
email varchar(100),
primary key (customerid)
)
create table loan
(
	loannumber bigint not null,
	loantype char,
	amount float,
	primary key (loannumber)
)
-- e.g., 'H' for home, 'P' for personal, etc.
create table borrows
(
	customerid int not null,
	loannumber bigint not null,
```

```sql
	foreign key (customerid) references customer(customerid),
	foreign key (loannumber) references loan(loannumber)
)
```

Step 2: Inserting Data

We insert sample records to validate the table relationships.

Sql code:

```sql
insert into customer  (customerid,customername,address,phone,email) values
(100,'Sanjog','Lalitpur',97000000,'gautamsanjok32@gmail.com'),
(101,'Salin','Lalitpur',99000000,'Salin@gmail.com'),
(102,'Jenish','Kirtipur',98000000,'jenish@gmail.com'),
(103,'Kushal','Kalanki',96000000,'kushal@gmail.com'),
(104,'Arun','Bhotahiti',95000000,'arun32@gmail.com'),
(105,'Swagat','Basundhara',94000000,'swagat@gmail.com');

insert into loan (loannumber, loantype, amount) values
  (1001, 'H', 50000.00),
  (1002, 'P', 75000.50);

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

Step 3: Running Queries

3.1 Customers from 'Lalitpur' in ascending name order

Sql code:

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

Fig 5.1:

Results | Messages

	customername
1	Salin
2	Sanjog

Description: The output lists customers living in *Lalitpur*, sorted alphabetically by name.

3.2 Count of customers having at least one loan

Sql code:

```sql
select count(customer.customerid) as num_customers_with_loans
from customer
```

```
join borrows on customer.customerid = borrows.customerid;
```

Fig 5.2:

	num_customers_with_loans
1	3

Description: The output shows that **three customers** currently have loans.

3.3 Customers with loan amount ≥ 50000

```
Sql code:
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;
```

Fig 5.3:

	customername
1	Kushal
2	Salin
3	Sanjog

Description: Displays names of customers who have loans worth at least 50,000.

3.4 Average loan amount for each loan type

```
Sql code:
select loantype, avg(amount) as average_amount
from loan
group by loantype;
```

Fig 5.4:

	loantype	average_amount
1	H	50000
2	P	75000.5

Description: Shows the average loan amount grouped by loan type — `H` (Home Loan) and `P` (Personal Loan).

Conclusion

In this lab, we successfully:

- Created relational tables with **foreign keys** to maintain referential integrity.
- Inserted valid records into related tables.
- Used **SELECT** queries with `WHERE`, `ORDER BY`, `JOIN`, and `GROUP BY` clauses to filter, combine, and aggregate data.
- Verified correct retrieval of data such as customers from a specific location, customers with loans, high-value loan holders, and average loan amounts.

Lab 6: Creating and Executing Stored Procedures with Dynamic Table Operations

Introduction:

In this lab, we focus on advanced SQL concepts including stored procedures, dynamic table creation, and data manipulation within procedural contexts. Stored procedures are precompiled SQL statements that can be executed multiple times with different parameters, providing better performance, security, and code reusability. This lab demonstrates the creation of a stored procedure that dynamically manages table structures and performs data operations based on foreign key relationships.

The lab covers:

- **ALTER PROCEDURE** – Modifying existing stored procedures
- **Dynamic Table Operations** – Creating and dropping tables within procedures
- **Conditional Data Insertion** – Inserting data based on specific criteria
- **Foreign Key Constraints** – Maintaining referential integrity in procedural contexts

Step 1: Creating the Base Table Structure

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

Sql code:

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

Fig 6.1: The students table is successfully created with 7 records containing students from different batch years (2080-2083). This table serves as the parent table for establishing foreign key relationships.

Step 2: Creating the Stored Procedure

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

Sql code:

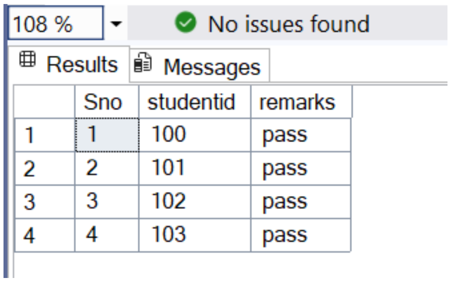
```
alter Procedure xyz
As
begin
drop table if exists studentexamdetails
-- Create the new table
CREATE TABLE studentexamdetails (
    Sno INT PRIMARY KEY IDENTITY(1,1),
    studentid INT,
    remarks VARCHAR(50),
    FOREIGN KEY (studentid) REFERENCES students(studentid)
);

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

Step 3: Executing the Stored Procedure

We execute the stored procedure and verify the results.

Sql code:

```
exec xyz
select * from studentexamdetails
```



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

`Fig 6.2: The students id who are from 2080 batch is shown.`

Conclusion

This lab successfully demonstrated the creation and execution of stored procedures that perform dynamic table operations while maintaining referential integrity. The procedure `xyz` effectively:

- Manages table lifecycle (drop and create)
- Implements conditional data insertion based on batch criteria
- Maintains foreign key relationships for data consistency
- Provides a reusable solution for exam result processing

Lab 7: Automated Loan Management System with Email Notification Using Stored Procedures

Introduction:

In this lab, we focus on advanced SQL concepts including stored procedures, automated data processing, and relational database management for a loan management system. Stored procedures are precompiled SQL statements that can be executed multiple times, providing better performance, security, and code reusability. This lab demonstrates the creation of a stored procedure that automatically generates email notifications for customers with overdue loan payments.

The lab covers:

- **CREATE PROCEDURE** – Creating stored procedures for business logic automation
- **Date Functions** – Using GETDATE () for current date operations
- **Window Functions** – Using ROW_NUMBER () for sequential numbering
- **JOIN Operations** – Combining data from multiple related tables
- **Conditional Data Insertion** – Inserting data based on specific criteria
- **Foreign Key Constraints** – Maintaining referential integrity in procedural contexts

Step 1: Creating the Database Tables

First, we establish the foundation by creating four interconnected tables: customerdetails, loantype, loandetails, and emaildetails.

SQL Code:

```
-- 1 Drop existing tables if they exist
DROP TABLE IF EXISTS emaildetails;
DROP TABLE IF EXISTS loandetails;
DROP TABLE IF EXISTS loantype;
DROP TABLE IF EXISTS customerdetails;

-- 2 Create customerdetails table
CREATE TABLE customerdetails (
    cid INT NOT NULL PRIMARY KEY,
    customername VARCHAR(100),
    email VARCHAR(100),
    phoneno BIGINT,
    address VARCHAR(100)
);

-- 3 Create loantype table
```

```sql
CREATE TABLE loantype (
    sno INT NOT NULL PRIMARY KEY,
    loantype CHAR(50) UNIQUE -- unique so it can be referenced
);

-- 4 Create loandetails table with EMI tracking
CREATE TABLE loandetails (
    sno INT NOT NULL PRIMARY KEY,
    cid INT, -- link to customer
    loanname VARCHAR(100),
    loantype CHAR(50),
    interest INT,
    due_date DATE,
    paid_date DATE NULL, -- NULL if unpaid
    FOREIGN KEY (loantype) REFERENCES loantype(loantype),
    FOREIGN KEY (cid) REFERENCES customerdetails(cid)
);

-- 5 Create emaildetails table
CREATE TABLE emaildetails (
    sno INT NOT NULL,
    [From] VARCHAR(100),
    [To] VARCHAR(100),
    cc VARCHAR(100),
    bcc VARCHAR(100),
    emailbody VARCHAR(100),
    status VARCHAR(100)
);
```

Step 2: Inserting Sample Data

```sql
-- Insert sample customers
INSERT INTO customerdetails (cid, customername, email, phoneno, address)
VALUES
(1, 'Sanjog Gautam', 'Sanjog@gmail.com', 9876543210, 'New York'),
(2, 'Sarfaraz Alam', 'Sarfaraz Alam@gmial.com', 9123456780, 'Los Angeles'),
(3, 'Susan Chaudhary', 'Susan Chaudhary@gmail.com', 9988776655, 'Chicago');

-- Insert loan types
INSERT INTO loantype (sno, loantype)
VALUES
(1, 'Home Loan'),
(2, 'Car Loan'),
(3, 'Personal Loan');

-- Insert loans with EMI info
INSERT INTO loandetails (sno, cid, loanname, loantype, interest, due_date,
paid_date)
VALUES
(1, 1, 'Dream Home Plan', 'Home Loan', 7, '2025-08-01', NULL), -- overdue
(2, 2, 'Swift Car Plan', 'Car Loan', 9, '2025-08-15', '2025-08-10'), -- paid

(3, 3, 'Holiday Special', 'Personal Loan', 12, '2025-08-05', NULL); -- overdue
```

Step 3: Creating the Stored Procedure

We create a stored procedure named 'xyz' that automatically generates email notifications for overdue loans.

```
CREATE PROCEDURE xyz
AS
BEGIN
    SET NOCOUNT ON;

    DECLARE @today DATE = GETDATE();

    INSERT INTO emaildetails (sno, [From], [To], cc, bcc, emailbody, status)
    SELECT
        ISNULL((SELECT MAX(sno) FROM emaildetails), 0)
        + ROW_NUMBER() OVER (ORDER BY c.cid) AS sno,
        'bank@example.com' AS [From],
        c.email AS [To],
        NULL AS cc,
        NULL AS bcc,
        CONCAT('Dear ', c.customername, ', your EMI for loan "', l.loanname, '" is
overdue. Please pay immediately.') AS emailbody,
        'Pending' AS status
    FROM loandetails l
    JOIN customerdetails c ON l.cid = c.cid
    WHERE l.paid_date IS NULL
      AND l.due_date < @today;
END;

GO
```

Step 4: Executing the Stored Procedure

```
EXEC xyz;

SELECT * FROM emaildetails;

Output:
```

Results | Messages

	sno	From	To	cc	bcc	emailbody	status
1	1	bank@example.com	Sanjog@gmail.com	NULL	NULL	Dear Sanjog Gautam, your EMI for loan "Dream Ho...	Pending
2	2	bank@example.com	Susan Chaudhary@gmail.com	NULL	NULL	Dear Susan Chaudhary, your EMI for loan "Holiday S...	Pending

Fig 7.1: The emaildetails table shows two automated email notifications generated for customers with overdue loans. Sanjog Gautam has an overdue Home Loan "Dream Home Plan" and Susan Chaudhary has an overdue Personal Loan "Holiday Special". The customer with paid loan (Sarfaraz Alam) did not receive any notification, demonstrating the procedure's conditional logic.

Conclusion:

This lab successfully demonstrated the creation and execution of stored procedures that automate business processes while maintaining referential integrity. The procedure 'xyz' effectively:

- **Identifies overdue loans** by comparing due dates with current date
- **Generates personalized emails** using customer and loan information
- **Automates notification process** reducing manual effort and human error
- **Maintains data consistency** through proper foreign key relationships
- **Provides scalable solution** for handling multiple customers and loan types.

Lab 8: Parameterized Stored Procedures with CRUD Operations

Introduction:

In this lab, we create a stored procedure that can perform multiple operations (SELECT and INSERT) on a database table using parameters. Instead of writing separate procedures for each operation, we use one procedure with a flag parameter to control what action to perform. This approach is commonly used in real-world applications for efficient database management.

The lab covers:

- **Parameterized Procedures** – Creating procedures with input parameters
- **Flag-Based Operations** – Using a flag to select different operations
- **Input Validation** – Checking if required data is provided
- **Error Handling** – Using TRY-CATCH to handle errors gracefully

Step 1: Creating the SubjectDetails Table

```
-- Drop existing table if it exists
DROP TABLE IF EXISTS SubjectDetails;

-- Create SubjectDetails table
CREATE TABLE SubjectDetails (
    SubjectId INT NOT NULL PRIMARY KEY,
    SubjectName VARCHAR(100)
);

-- Insert sample data
INSERT INTO SubjectDetails (SubjectId, SubjectName) VALUES
(101, 'MATHS'),
(102, 'C++'),
(103, 'DSA');

-- Verify the data
SELECT * FROM SubjectDetails;
```

97 % No issues found

Results Messages

	SubjectId	SubjectName
1	101	MATHS
2	102	C++
3	103	DSA

Fig 8.1: The SubjectDetails table is created with 3 sample records.

Step 2: Creating the Stored Procedure

We create a procedure called 'sp_SubjectDetails' that can do two things:

1. **SELECT**: Retrieve data from the table (when @flag = 's')
2. **INSERT**: Add new data to the table (when @flag = 'i')

```
Alter Procedure sp_SubjectDetails
(
	@flag Char,
	@SubjectId Int = NULL,
	@SubjectName Varchar(100) = NULL
)
AS
BEGIN
	If @flag = 's' -- SELECT Data From Table
	BEGIN
		IF @SubjectId IS NULL
		BEGIN
			Select '0' As STATUS_CODE,
				'Data Retrived From Table' AS STATUS_MSG
			Select
				SubjectId,
				SubjectName
			From SubjectDetails
			RETURN
		END
		Select
			SubjectId,
			SubjectName
		From SubjectDetails WHERE SubjectId = @SubjectId


	END

	If @flag = 'i' -- INSERT DATA INTO TABLE
	BEGIN
		IF @SubjectId IS NULL OR @SubjectId = ''
		BEGIN
			Select '100' As STATUS_CODE,
			'Subject ID is Missing' AS STATUS_MSG
			RETURN
		END

		IF @SubjectName IS NULL OR @SubjectName = ''
		BEGIN
```

```
            Select '101' As STATUS_CODE,
            'Subject Name is Missing' AS STATUS_MSG
            RETURN
        END

        BEGIN TRY
            Insert Into SubjectDetails(SubjectId, SubjectName)
            Values(@SubjectId, @SubjectName)

            Select '0' AS STATUS_CODE,
            'New Subject is Added successfully' AS STATUS_MSG

        END TRY

        BEGIN CATCH
            Select ERROR_NUMBER() AS STATUS_CODE,
            ERROR_MESSAGE() AS STATUS_MSG
        END CATCH


    END
END
```

How It Works:

Parameters:

- `@flag`: Tells the procedure what to do ('s' = SELECT, 'i' = INSERT)
- `@SubjectId`: The subject ID number
- `@SubjectName`: The subject name

For SELECT (@flag = 's'):

- If no SubjectId is given, it shows all subjects
- If SubjectId is given, it shows only that subject

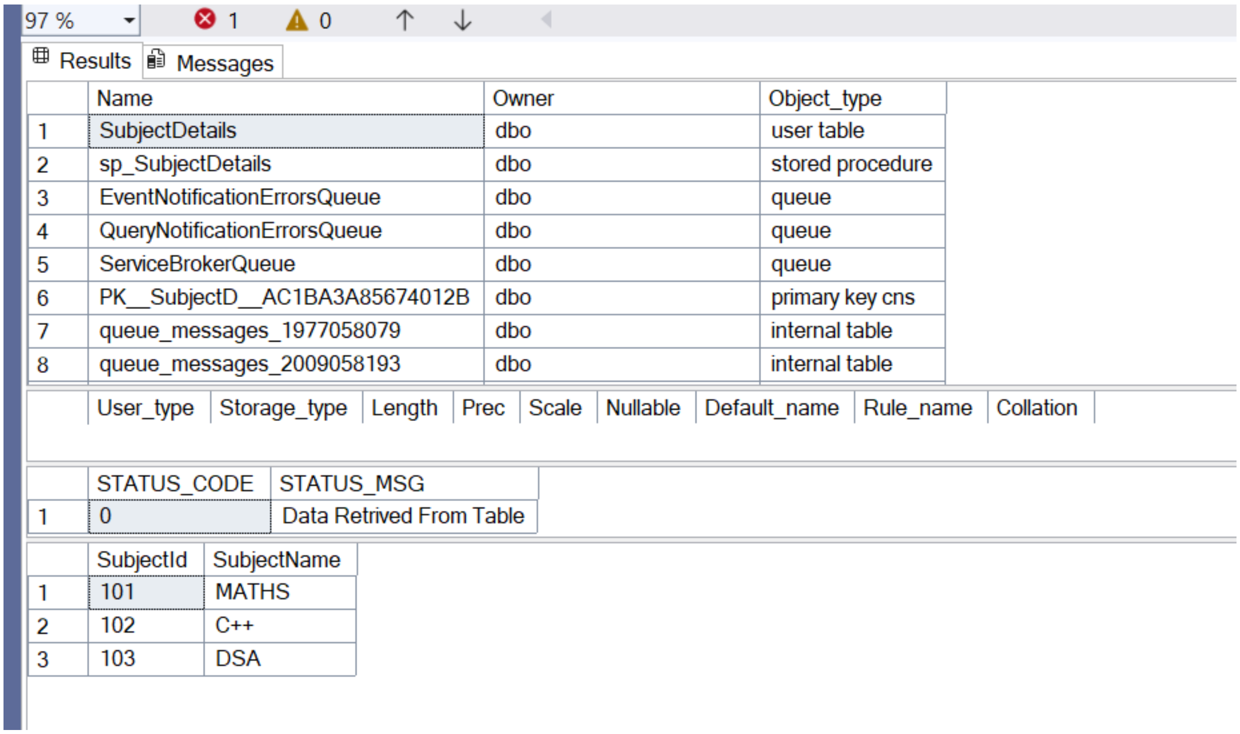
For INSERT (@flag = 'i'):

- First checks if SubjectId is provided (if not, returns error code 100)
- Then checks if SubjectName is provided (if not, returns error code 101)
- If both are provided, it inserts the new subject
- Uses TRY-CATCH to handle any database errors

Step 3: Retrieving All Data (SELECT Operation)

We execute the procedure with flag 's' to get all subjects.

```
sp_help

EXEC sp_SubjectDetails @flag='s', @SubjectId = NULL, @SubjectName = NULL;
```



	Name	Owner	Object_type
1	SubjectDetails	dbo	user table
2	sp_SubjectDetails	dbo	stored procedure
3	EventNotificationErrorsQueue	dbo	queue
4	QueryNotificationErrorsQueue	dbo	queue
5	ServiceBrokerQueue	dbo	queue
6	PK__SubjectD__AC1BA3A85674012B	dbo	primary key cns
7	queue_messages_1977058079	dbo	internal table
8	queue_messages_2009058193	dbo	internal table

	User_type	Storage_type	Length	Prec	Scale	Nullable	Default_name	Rule_name	Collation

	STATUS_CODE	STATUS_MSG
1	0	Data Retrived From Table

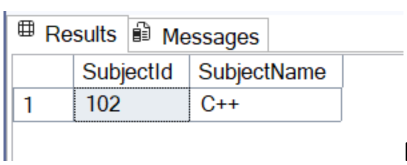
	SubjectId	SubjectName
1	101	MATHS
2	102	C++
3	103	DSA

Fig 8.2: The procedure successfully retrieves all subjects from the table. Status code '0' means the operation was successful.

Step 4: Retrieving Specific Data

We can also get a specific subject by providing its ID.

```
EXEC sp_SubjectDetails @flag='s', @SubjectId = 102, @SubjectName = NULL;
```



	SubjectId	SubjectName
1	102	C++

Fig 8.3: The procedure returns only the subject with ID 102.

Step 5: Inserting New Data (INSERT Operation)

Now we use the same procedure to add a new subject to the table.

```
EXEC sp_SubjectDetails @flag='i',
    @SubjectId = 104,
    @SubjectName = 'Compiler Construction and Design';

-- Verify the insertion

SELECT * FROM SubjectDetails;
```

Results | Messages

	STATUS_CODE	STATUS_MSG
1	0	New Subject is Added successfully

	SubjectId	SubjectName
1	101	MATHS
2	102	C++
3	103	DSA
4	104	Compiler Construction and Design

Fig 8.4: The new subject "Compiler Construction and Design" with ID 104 is successfully added to the table.

Step 6: Testing Error Handling

Let's see what happens if we try to insert without providing the SubjectId.

```
EXEC sp_SubjectDetails @flag='i',
    @SubjectId = NULL,

    @SubjectName = 'Database Management';
```

Results | Messages

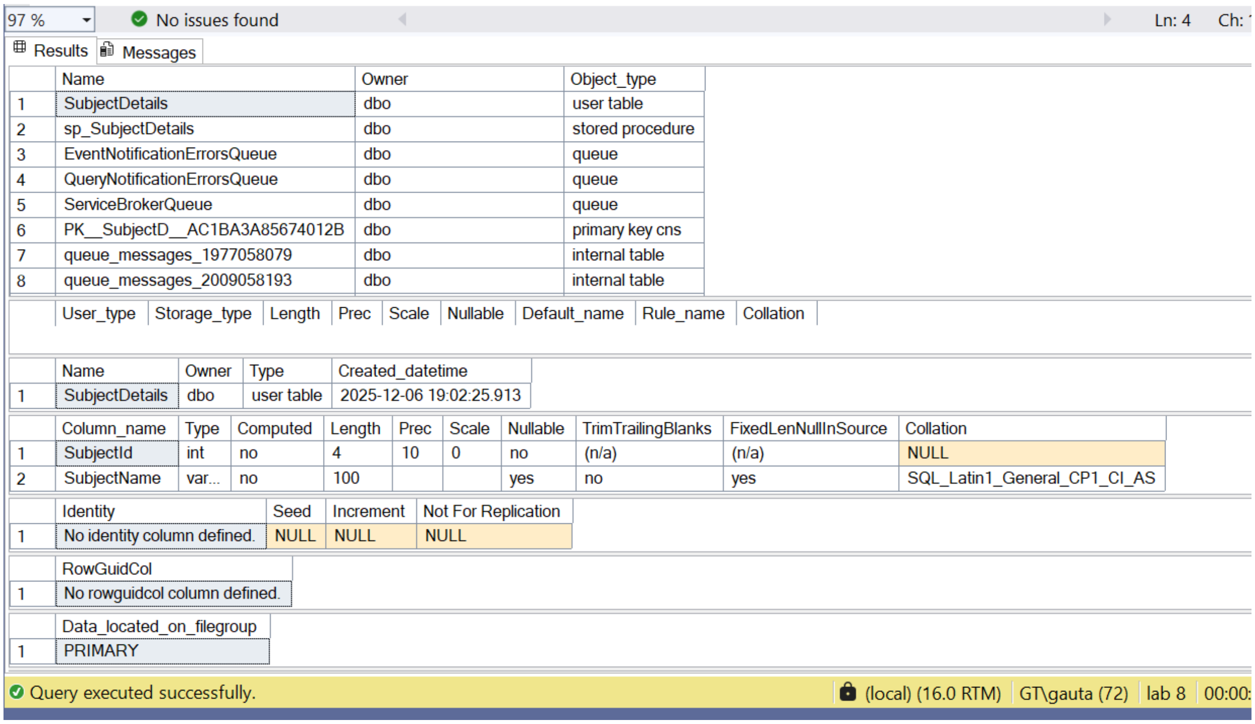
	STATUS_CODE	STATUS_MSG
1	100	Subject ID is Missing

Fig 8.5: The procedure checks for missing data and returns an error message instead of crashing.

Step 7: Using sp_help Command

We can use the built-in sp_help command to see information about our table.

```
sp_help

exec sp_help SubjectDetails
```



97 % No issues found Ln: 4 Ch:

Results Messages

	Name	Owner	Object_type
1	SubjectDetails	dbo	user table
2	sp_SubjectDetails	dbo	stored procedure
3	EventNotificationErrorsQueue	dbo	queue
4	QueryNotificationErrorsQueue	dbo	queue
5	ServiceBrokerQueue	dbo	queue
6	PK__SubjectD__AC1BA3A85674012B	dbo	primary key cns
7	queue_messages_1977058079	dbo	internal table
8	queue_messages_2009058193	dbo	internal table

User_type	Storage_type	Length	Prec	Scale	Nullable	Default_name	Rule_name	Collation

	Name	Owner	Type	Created_datetime
1	SubjectDetails	dbo	user table	2025-12-06 19:02:25.913

	Column_name	Type	Computed	Length	Prec	Scale	Nullable	TrimTrailingBlanks	FixedLenNullInSource	Collation
1	SubjectId	int	no	4	10	0	no	(n/a)	(n/a)	NULL
2	SubjectName	var...	no	100			yes	no	yes	SQL_Latin1_General_CP1_CI_AS

	Identity	Seed	Increment	Not For Replication
1	No identity column defined.	NULL	NULL	NULL

	RowGuidCol
1	No rowguidcol column defined.

	Data_located_on_filegroup
1	PRIMARY

Query executed successfully. (local) (16.0 RTM) GT\gauta (72) lab 8 00:00:

Fig 8.6: The sp_help command shows detailed information about the SubjectDetails table including column names, data types, and constraints.

Conclusion:

This lab successfully demonstrated how to create a parameterized stored procedure that handles multiple operations. The procedure 'sp_SubjectDetails' effectively:

- **Performs SELECT operations** to retrieve all or specific subjects
- **Performs INSERT operations** to add new subjects to the table
- **Validates input data** before processing to ensure data quality
- **Handles errors gracefully** using TRY-CATCH blocks
- **Returns status codes** to indicate success or failure

This approach is practical and efficient for real-world applications where we need to perform multiple database operations through a single, reusable procedure.

Lab 9: Transaction Management in Banking System Using BEGIN TRANSACTION, COMMIT, and ROLLBACK

Introduction:

In this lab, we implement a banking transaction system that demonstrates the critical concept of database transactions. A transaction is a set of SQL operations that must be executed as a single unit - either all operations succeed together, or all fail together. This is essential in banking systems where money transfer between accounts must be atomic (all-or-nothing) to maintain data consistency and prevent financial errors.

The lab covers:

- **Transaction Control** – Using BEGIN TRANSACTION, COMMIT, and ROLLBACK
- **ACID Properties** – Ensuring Atomicity, Consistency, Isolation, and Durability
- **Balance Validation** – Preventing negative account balances
- **Error Handling** – Rolling back transactions when conditions fail
- **Stored Procedures** – Encapsulating transaction logic for reusability

Scenario: Transfer money from Account A (source) to Account B (destination). If Account A has insufficient balance, the entire transaction should be cancelled.

Step 1: Creating the Database and Table

First, we create a new database and the Accounts table to store customer account information.

```
-- Create a new database
CREATE DATABASE txn;
GO

-- Use the database
USE txn;
GO

-- Create the Accounts table
CREATE TABLE Accounts (
    AccountID INT PRIMARY KEY,
    Balance DECIMAL(10, 2) NOT NULL
);

-- Insert sample data
```

```sql
INSERT INTO Accounts (AccountID, Balance)
VALUES (1, 500.00), (2, 300.00);

-- Verify the data

SELECT * FROM Accounts;
```

Results | Messages

	AccountID	Balance
1	1	500.00
2	2	300.00

Fig 9.1: The Accounts table is created with two accounts. Account 1 has a balance of 500.00 and Account 2 has a balance of 300.00.

Step 2: Creating the Transaction Stored Procedure

We create a stored procedure that handles money transfer between two accounts using transaction control.

```sql
CREATE PROCEDURE TransferAmount
    @fromAccount INT,
    @toAccount INT,
    @transferAmount DECIMAL(10, 2)
AS
BEGIN
    DECLARE @balanceA DECIMAL(10, 2);

    -- Start the transaction
    BEGIN TRANSACTION;

    BEGIN TRY
        -- Check the balance of the source account
        SELECT @balanceA = Balance
        FROM Accounts
        WHERE AccountID = @fromAccount;

        -- Perform the transaction if balance is sufficient
        IF @balanceA >= @transferAmount
        BEGIN
            -- Debit from source account
            UPDATE Accounts
            SET Balance = Balance - @transferAmount
            WHERE AccountID = @fromAccount;

            -- Credit to target account
            UPDATE Accounts
            SET Balance = Balance + @transferAmount
            WHERE AccountID = @toAccount;

            -- Commit the transaction
            COMMIT TRANSACTION;
            SELECT 'Transaction completed successfully.' AS Message;
        END
```

```sql
        ELSE
        BEGIN
            -- Rollback the transaction if insufficient balance
            ROLLBACK TRANSACTION;
            SELECT 'Transaction failed: Insufficient balance in the source account.'
AS Message;
        END
    END TRY
    BEGIN CATCH
        -- Rollback in case of any error
        IF @@TRANCOUNT > 0
            ROLLBACK TRANSACTION;
        SELECT ERROR_MESSAGE() AS Message;
    END CATCH
END;

GO
```

How the Procedure Works:

1. Parameters:

- `@fromAccount`: Source account ID (where money is debited)
- `@toAccount`: Destination account ID (where money is credited)
- `@transferAmount`: Amount to transfer

Note: SQL Server uses `@` prefix for parameters, unlike MySQL which uses `IN` keyword.

2. Transaction Steps:

- **BEGIN TRANSACTION**: Begins a new transaction
- **SELECT @balanceA = Balance**: Checks the current balance of source account
- **IF @balanceA >= @transferAmount**: Validates if transfer is possible
- **UPDATE (Debit)**: Subtracts amount from source account
- **UPDATE (Credit)**: Adds amount to destination account
- **COMMIT TRANSACTION**: Makes all changes permanent if successful
- **ROLLBACK TRANSACTION**: Cancels all changes if validation fails
- **TRY-CATCH**: Handles any unexpected errors during the transaction

Step 3: Testing Successful Transaction

We test the procedure by transferring 200.00 from Account 1 to Account 2.

```sql
-- Transfer 200.00 from Account 1 to Account 2
EXEC TransferAmount @fromAccount = 1, @toAccount = 2, @transferAmount = 200.00;

-- Check the updated balances

SELECT * FROM Accounts;
```

Results | Messages

	Message
1	Transaction completed successfully.

	AccountID	Balance
1	1	300.00
2	2	500.00

Fig 9.2: The transaction is successful. Account 1 balance decreased from 500.00 to 300.00 (debited 200.00), and Account 2 balance increased from 300.00 to 500.00 (credited 200.00). The transaction was committed successfully.

Step 4: Testing Failed Transaction (Insufficient Balance)

Now we test what happens when trying to transfer more money than available in the source account.

```
-- Attempt to transfer 400.00 from Account 1 (which only has 300.00)
EXEC TransferAmount @fromAccount = 1, @toAccount = 2, @transferAmount = 400.00;

-- Check the balances (should remain unchanged)

SELECT * FROM Accounts;
```

Results | Messages

	Message
1	Transaction failed: Insufficient balance in the ...

	AccountID	Balance
1	1	300.00
2	2	500.00

Fig 9.3: The transaction fails because Account 1 only has 300.00 but the transfer amount is 400.00. The ROLLBACK command cancels any changes, so both account balances remain unchanged. This demonstrates transaction atomicity.

Step 5: Testing Edge Case (Exact Balance Transfer)

Let's test transferring the exact balance available in an account.

```sql
-- Transfer exactly 300.00 from Account 1 (empties the account)
EXEC TransferAmount @fromAccount = 1, @toAccount = 2, @transferAmount = 300.00;

-- Check the updated balances

SELECT * FROM Accounts;
```

Results | Messages

	Message
1	Transaction completed successfully.

	AccountID	Balance
1	1	0.00
2	2	800.00

Fig 9.4: The transaction successfully transfers all 300.00 from Account 1 to Account 2. Account 1 now has zero balance, which is acceptable (not negative). This shows the procedure handles exact balance transfers correctly.

Conclusion:

This lab successfully demonstrated transaction management in a banking system using stored procedures with transaction control commands. The TransferAmount procedure effectively:

- **Ensures atomicity** by using START TRANSACTION, COMMIT, and ROLLBACK
- **Validates business rules** by checking sufficient balance before transfer
- **Prevents data inconsistency** by rolling back failed transactions
- **Maintains data integrity** by preventing negative account balances
- **Provides clear feedback** through success/failure messages

The implementation showcases essential database concepts that are critical for any financial or business-critical application. Understanding transaction management is fundamental for building reliable systems that handle sensitive data and operations where partial completion is not acceptable. This approach ensures that the database always remains in a consistent state, even when operations fail.

Lab 10: Cartesian Product and JOIN Operations

Introduction:

In this lab, we explore the concept of Cartesian Product in database systems. A Cartesian Product combines every row from one table with every row from another table. We also learn how to filter this product to create meaningful relationships between students and their courses based on their academic stream.

The lab covers:

- **Cartesian Product** – All possible combinations between tables
- **Filtered Cartesian Product** – Using conditions to get relevant data
- **JOIN Operations** – Combining tables efficiently
- **Views** – Creating simplified data representations

Step 1: Creating Tables and Inserting Data

We create four tables to manage student course enrollment and populate them with sample data.

```sql
-- Create tables
CREATE TABLE Students (
    std_id INT PRIMARY KEY,
    student_name VARCHAR(50),
    streamm VARCHAR(50)
);

CREATE TABLE Courses (
    course_id INT PRIMARY KEY,
    course_name VARCHAR(50)
);

CREATE TABLE StudentCourses (
    std_id INT,
    course_id INT,
    semester INT,
    PRIMARY KEY (std_id, course_id, semester),
    FOREIGN KEY (std_id) REFERENCES Students(std_id),
    FOREIGN KEY (course_id) REFERENCES Courses(course_id)
);

CREATE TABLE StudentSemester (
    std_id INT,
    semester INT,
    PRIMARY KEY (std_id, semester),
```

```sql
    FOREIGN KEY (std_id) REFERENCES Students(std_id)
);

-- Insert sample data
INSERT INTO Students (std_id, student_name, streamm)
VALUES
(1, 'Student A', 'Science'),
(2, 'Student B', 'Humanities'),
(3, 'Student C', 'Science'),
(4, 'Student D', 'Humanities');

INSERT INTO Courses (course_id, course_name)
VALUES
(1, 'NM'),
(2, 'CA'),
(3, 'DSA'),
(4, 'History'),
(5, 'Sociology'),
(6, 'Psychology');

INSERT INTO StudentSemester (std_id, semester)
VALUES
(1, 3),
(2, 3),
(3, 2),
(4, 1);

-- Verify data
SELECT * FROM Students;
SELECT * FROM Courses;

SELECT * FROM StudentSemester;
```

96 % No issues found

Results Messages

	std_id	student_name	streamm
1	1	Student A	Science
2	2	Student B	Humanities
3	3	Student C	Science
4	4	Student D	Humanities

	course_id	course_name
1	1	NM
2	2	CA
3	3	DSA
4	4	History
5	5	Sociology
6	6	Psychology

	std_id	semester
1	1	3
2	2	3
3	3	2
4	4	1

Fig 10.1: The tables are created with 4 students (2 Science, 2 Humanities) and 6 courses (3 Science courses, 3 Humanities courses).

Step 2: Understanding Cartesian Product

A Cartesian Product creates all possible combinations between two tables.

```sql
-- Cartesian Product: All possible student-course combinations
SELECT
    s.student_name,
    s.streamm,
    c.course_name
FROM Students s, Courses c

ORDER BY s.std_id, c.course_id;
```

96 % ⊗ 7 ⚠ 0

Results Messages

	student_name	streamm	course_name
1	Student A	Science	NM
2	Student A	Science	CA
3	Student A	Science	DSA
4	Student A	Science	History
5	Student A	Science	Sociology
6	Student A	Science	Psychology
7	Student B	Humanities	NM
8	Student B	Humanities	CA
9	Student B	Humanities	DSA
10	Student B	Humanities	History
11	Student B	Humanities	Sociology
12	Student B	Humanities	Psychology
13	Student C	Science	NM
14	Student C	Science	CA
15	Student C	Science	DSA
16	Student C	Science	History
17	Student C	Science	Sociology
18	Student C	Science	Psychology
19	Student D	Humanities	NM
20	Student D	Humanities	CA
21	Student D	Humanities	DSA
22	Student D	Humanities	History
23	Student D	Humanities	Sociology
24	Student D	Humanities	Psychology

Fig 10.2: The Cartesian Product produces 24 rows (4 students × 6 courses = 24). Notice that Science students are paired with Humanities courses and vice versa, which doesn't make sense academically.

Step 3: Creating Filtered Course Enrollment

Now we filter the Cartesian Product to enroll students only in courses matching their stream.

```sql
-- Insert filtered enrollments based on stream
INSERT INTO StudentCourses (std_id, course_id, semester)
SELECT s.std_id, c.course_id, ss.semester
FROM Students s
JOIN StudentSemester ss ON s.std_id = ss.std_id
JOIN Courses c ON (
    (s.streamm = 'Science' AND c.course_name IN ('NM', 'CA', 'DSA')) OR
    (s.streamm = 'Humanities' AND c.course_name IN ('History', 'Sociology',
'Psychology'))
);

-- Verify enrollments

SELECT * FROM StudentCourses ORDER BY std_id, course_id;
```

	std_id	course_id	semester
1	1	1	3
2	1	2	3
3	1	3	3
4	2	4	3
5	2	5	3
6	2	6	3
7	3	1	2
8	3	2	2
9	3	3	2
10	4	4	1
11	4	5	1
12	4	6	1

Fig 10.3: The filtered result shows only 12 rows (reduced from 24). Each Science student is enrolled in 3 Science courses, and each Humanities student is enrolled in 3 Humanities courses.

How it Works:

- **Start with**: All student-course combinations (24 rows)
- **Filter by**: Student stream matches course type
- **Result**: Only valid enrollments (12 rows)

Step 4: Creating a View for Easy Access

We create a view that combines all information for easy querying.

```sql
-- Create view
CREATE VIEW StudentCourseDetails AS
SELECT
    s.std_id,
    s.student_name,
    sc.semester,
    s.streamm,
```

```
    c.course_name
FROM
    Students s
JOIN
    StudentCourses sc ON s.std_id = sc.std_id
JOIN
    Courses c ON sc.course_id = c.course_id;
GO

-- Query the view

SELECT * FROM StudentCourseDetails ORDER BY std_id, course_name;
```

96 % 14 0

Results Messages

	std_id	student_name	semester	streamm	course_name
1	1	Student A	3	Science	CA
2	1	Student A	3	Science	DSA
3	1	Student A	3	Science	NM
4	2	Student B	3	Humanities	History
5	2	Student B	3	Humanities	Psychology
6	2	Student B	3	Humanities	Sociology
7	3	Student C	2	Science	CA
8	3	Student C	2	Science	DSA
9	3	Student C	2	Science	NM
10	4	Student D	1	Humanities	History
11	4	Student D	1	Humanities	Psychology
12	4	Student D	1	Humanities	Sociology

Query executed successfully

Fig 10.4: The view displays complete student enrollment information in an easy-to-read format, showing each student with their courses, stream, and semester.

Conclusion:

This lab demonstrated how Cartesian Product works and how to filter it for practical use. We learned that:

- **Cartesian Product** generates all possible combinations (24 rows from 4 students × 6 courses)
- **Filtering** reduces results to meaningful data (12 valid enrollments)
- **Views** simplify data access by combining multiple tables
- **Business rules** ensure data quality (students only get courses from their stream)

Understanding Cartesian Products helps us write better queries and understand how database joins work efficiently.