IE2042: Database Management Systems for Security

Year 2, Semester 1

Database Design, Implementation and Security

University Library Management System

Group Assignment

BSc (Hons) in Information Technology

Sri Lanka Institute of Information Technology

Malabe

Sri Lanka

Group No: 11

Submission Date: 15/10/2024

Table of Contents

1. Declaration and Group Details

We declare that this is our own work, and this Assignment does not incorporate without acknowledgment any material previously submitted by anyone else in SLIIT or any other university/Institute. And we declare that each one of us equally contributed to the completion of this Assignment.

Branch: Malabe

Group No: 11

Batch: Weekday_ CS

Members:

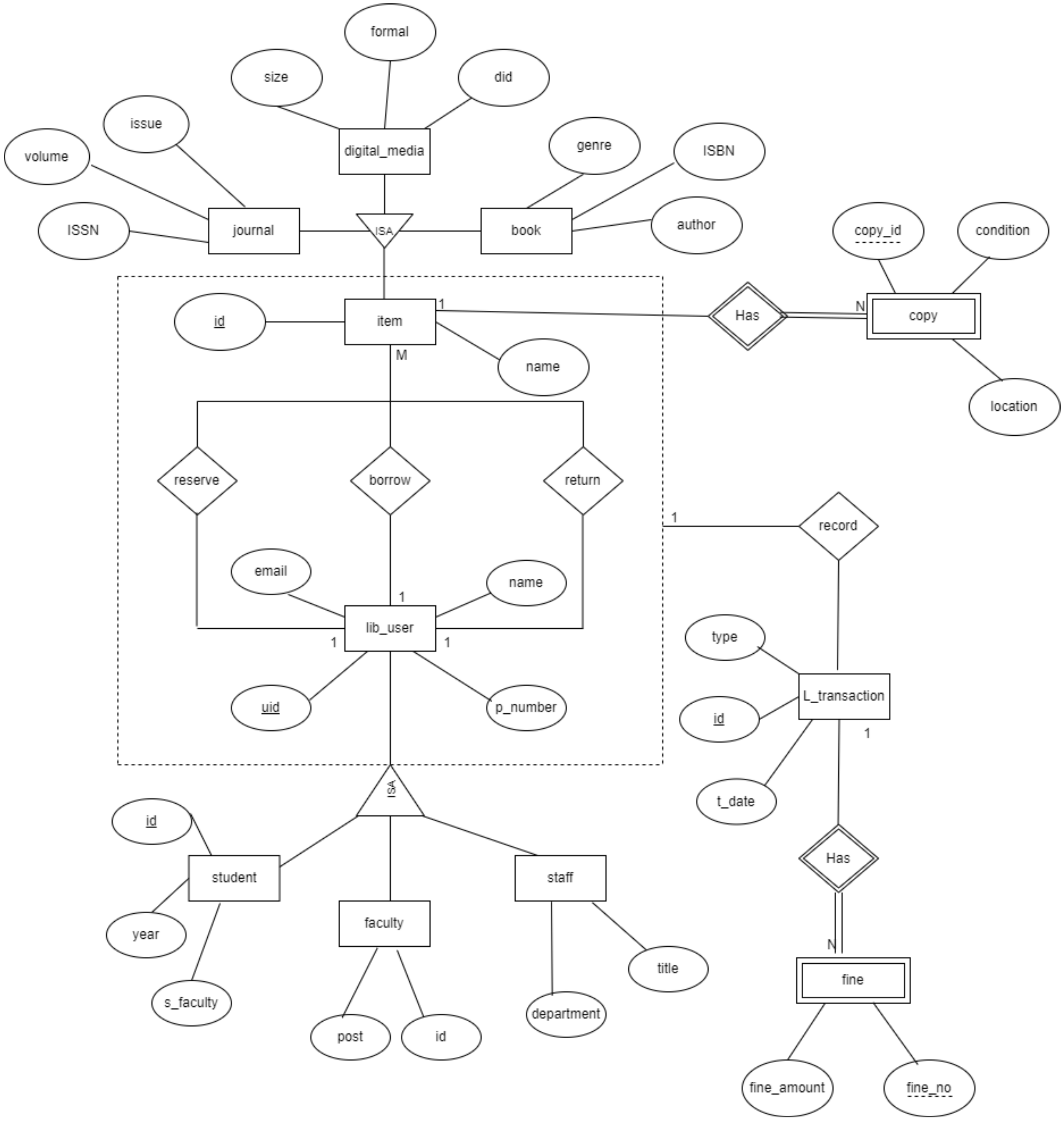
Registration Number	**Name**	**Contact Number**
IT23289802	KUMARARATHNA B.A.G.S.N	070 219 0807
IT23292918	KUMARAGE S.D.	078 593 6464
IT23293830	JAYASEKARA H.D.P.P	077 002 9940
IT23298408	MINIPURA P T D	071 489 9201

2. Part I

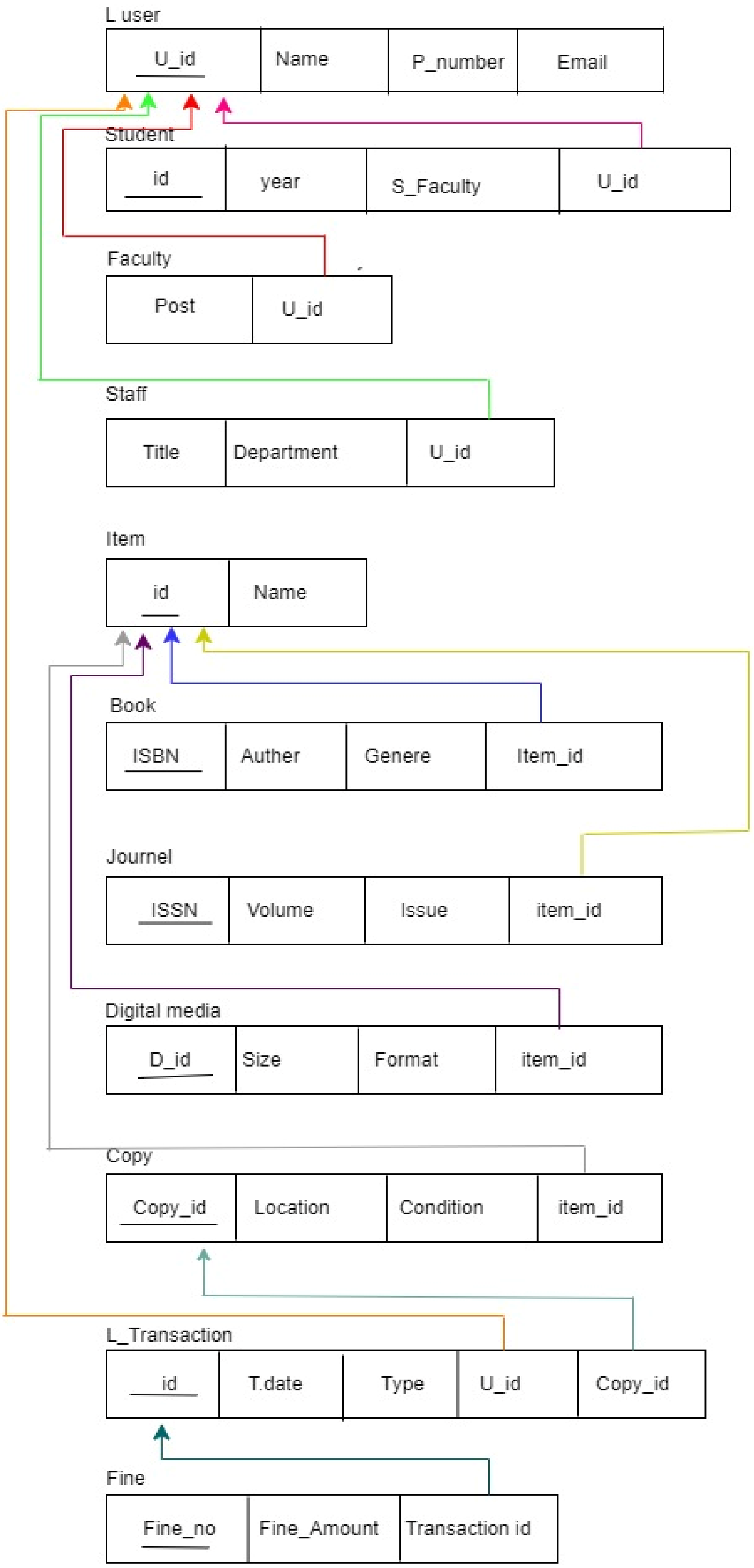
2.1 Assumptions

1. Before using the library, a user (student, faculty, or staff) must be officially registered within the university system.
2. Every entity in the system has a unique identifier to ensure data integrity. For example, users are identified by **U_id**, books by **Item_id**, and transactions by **Transaction_id**.
3. Each item in the library can have multiple copies, with each copy having its own **Copy_id**, Condition, and Location within the library.
4. Fines are imposed on users who do not return items by the due date. The fines may vary based on the type of item (e.g., books, journals, digital media) and the duration of the delay.
5. Users can only reserve items if they are not currently checked out by another user. Reservations are tracked with specific dates.
6. The system tracks the condition of each item copy, which is updated upon return of borrowed items. If an item is damaged, additional fines may be imposed.
7. All borrowing, returning, and reservation transactions are time-stamped to track the exact date and time of each transaction.

2.2 Entity Relationship Diagram



2.3 Logical Model



L user

U_id	Name	P_number	Email

Student

id	year	S_Faculty	U_id

Faculty

Post	U_id

Staff

Title	Department	U_id

Item

id	Name

Book

ISBN	Auther	Genere	Item_id

Journel

ISSN	Volume	Issue	item_id

Digital media

D_id	Size	Format	item_id

Copy

Copy_id	Location	Condition	item_id

L_Transaction

id	T.date	Type	U_id	Copy_id

Fine

Fine_no	Fine_Amount	Transaction id

2.4 Normalization to 3NF.

- First Normal Form (1NF):

  The original model was already in 1NF as all attributes contained atomic values.

- Second Normal Form (2NF):

  The original model was already in 2NF as all non-key attributes were fully functionally dependent on their primary keys.

- Third Normal Form (3NF):

  To achieve 3NF, we need to eliminate transitive dependencies. The main change here is the introduction of the ITEM_TYPE table.

Changes made to achieve 3NF:

- Created ITEM_TYPE table: This new table removes the transitive dependency of LoanPeriod on ItemType in the ITEM table.

- **ITEM_TYPE** <u>ItemTypeID</u> (PK), TypeName, and LoanPeriod.

  The ITEM table now has a foreign key ItemTypeID referencing ITEM_TYPE.

- Removed ItemType attribute from ITEM:

  This is now stored on the ITEM_TYPE table.

- Moved LoanPeriod to ITEM_TYPE:

  LoanPeriod is now associated with the item type rather than individual items.

- USER table remains unchanged:

  It was already in 3NF, with subtype tables (STUDENT, FACULTY, STAFF) properly normalized.

- COPY, BORROW, RESERVATION, and FINE tables remain unchanged:

  These tables were already in 3NF.

2.5 SQL Queries

2.5.1 Table Implementation

2.5.2 Constraints Implementation

```sql
-- SQL Server Script
USE Library_Management_System;

--Create ITEM_TYPE table
CREATE TABLE ITEM_TYPE (
    ItemTypeID INT PRIMARY KEY,
    TypeName VARCHAR(100) NOT NULL,
    LoanPeriod INT NOT NULL
);

--Create ITEM table
CREATE TABLE ITEM (
    ItemID INT PRIMARY KEY,
    Title VARCHAR(255) NOT NULL,
    Author VARCHAR(255),
    ISBN VARCHAR(20) UNIQUE,
    ISSN VARCHAR(20),
    Volume INT,
    Issue INT,
    Format VARCHAR(50),
    Size VARCHAR(50),
    ItemTypeID INT NOT NULL,
    FOREIGN KEY (ItemTypeID) REFERENCES ITEM_TYPE(ItemTypeID)
);

--Create COPY table
CREATE TABLE COPY (
    CopyID INT PRIMARY KEY,
    Condition VARCHAR(50) NOT NULL,
    Location VARCHAR(100) NOT NULL,
    ItemID INT NOT NULL,
    FOREIGN KEY (ItemID) REFERENCES ITEM(ItemID)
);

--Create L_USER table (Library User)
CREATE TABLE Lib_USER (
    UserID INT PRIMARY KEY,
    Name VARCHAR(255) NOT NULL,
    Email VARCHAR(100) UNIQUE NOT NULL,
    PhoneNumber VARCHAR(15),
    Role VARCHAR(50) NOT NULL     --defines whether the user is a student, faculty
member, or staff member
);

--Create BORROW table
CREATE TABLE BORROW (
    BorrowID INT PRIMARY KEY,
    UserID INT NOT NULL,
    CopyID INT NOT NULL,
    BorrowDate DATE NOT NULL,
    ReturnDate DATE,
    FOREIGN KEY (UserID) REFERENCES Lib_USER(UserID),
    FOREIGN KEY (CopyID) REFERENCES COPY(CopyID)
);

--Create RESERVE table
CREATE TABLE RESERVE (
    ReserveID INT PRIMARY KEY,
    UserID INT NOT NULL,
    CopyID INT NOT NULL,
    ReserveDate DATE NOT NULL,
    FOREIGN KEY (UserID) REFERENCES Lib_USER(UserID),
    FOREIGN KEY (CopyID) REFERENCES COPY(CopyID)
);

--Create FINE table
CREATE TABLE FINE (
    FineID INT PRIMARY KEY,
    BorrowID INT NOT NULL,
    FineAmount DECIMAL(10, 2) NOT NULL,
    FineDate DATE NOT NULL,
    FOREIGN KEY (BorrowID) REFERENCES BORROW(BorrowID)
);

--Create L_TRANSACTION table (Library transaction)
CREATE TABLE L_TRANSACTION (
    TransactionID INT PRIMARY KEY,
    TransactionType VARCHAR(100) NOT NULL,
    TransactionDate DATE NOT NULL,
    FineID INT NULL,
    FOREIGN KEY (FineID) REFERENCES FINE(FineID)
);

GO

--Creating VIEWS
CREATE VIEW MemberBorrowedItems AS
SELECT
    u.UserID,
    u.Name AS MemberName,
    i.Title AS ItemTitle,
    b.BorrowDate,
    b.ReturnDate
FROM
    Lib_USER u
JOIN BORROW b ON u.UserID = b.UserID
JOIN COPY c ON b.CopyID = c.CopyID
JOIN ITEM i ON c.ItemID = i.ItemID
WHERE
    u.Role IN ('Student', 'Faculty', 'Staff');

GO
```

2.5.3 Sample Data Entry

```sql
--Entry of sample data
--Sample Data for ITEM_TYPE
INSERT INTO ITEM_TYPE (ItemTypeID, TypeName, LoanPeriod) VALUES
(1, 'Book', 14),
(2, 'Journal', 7),
(3, 'Digital Media', 30);

--Sample Data for ITEM
INSERT INTO ITEM (ItemID, Title, Author, ISBN, ItemTypeID) VALUES
(1, 'The Great Gatsby', 'F. Scott Fitzgerald', '9780743273565', 1),
(2, 'Nature Journal', NULL, '00280836', 2),
(3, 'Introduction to Algorithms', 'Cormen et al.', '9780262033848', 1);

--Sample Data for COPY
INSERT INTO COPY (CopyID, Condition, Location, ItemID) VALUES
(1, 'Good', 'Shelf A1', 1),
(2, 'Good', 'Shelf B2', 2),
(3, 'Fair', 'Shelf C3', 3);

--Sample Data for L_USER
INSERT INTO Lib_USER (UserID, Name, Email, PhoneNumber, Role) VALUES
(1, 'Alice Smith', 'alice.smith@university.edu', '123-456-7890', 'Student'),
(2, 'Bob Johnson', 'bob.johnson@university.edu', '098-765-4321', 'Faculty'),
(3, 'Carol Davis', 'carol.davis@university.edu', '555-555-5555', 'Staff');

--Sample Data for BORROW
INSERT INTO BORROW (BorrowID, UserID, CopyID, BorrowDate, ReturnDate) VALUES
(1, 1, 1, '2024-09-01', '2024-09-15'),
(2, 2, 3, '2024-09-05', NULL);

--Sample Data for RESERVE
INSERT INTO RESERVE (ReserveID, UserID, CopyID, ReserveDate) VALUES
(1, 3, 3, '2024-09-10');

--Sample Data for FINE
INSERT INTO FINE (FineID, BorrowID, FineAmount, FineDate) VALUES
(1, 2, 10.00, '202');
```

2.5.4 Triggers

```
--Creating Triggers
CREATE TRIGGER CheckBorrowLimit
ON BORROW
AFTER INSERT
AS
BEGIN
    DECLARE @UserID INT;
    DECLARE @BorrowCount INT;

    -- Get the UserID from the inserted row
    SELECT @UserID = i.UserID FROM inserted i;

    -- Count how many items the user has currently borrowed
    SELECT @BorrowCount = COUNT(*)
    FROM BORROW
    WHERE UserID = @UserID AND ReturnDate IS NULL;

    -- Check if the limit exceeds 5
    IF @BorrowCount > 5
    BEGIN
        -- Rollback the transaction if the limit is exceeded
        ROLLBACK TRANSACTION;
        RAISERROR('Cannot borrow more than 5 items at a time.', 16, 1);
    END
END;

GO
```

This trigger ensures that members don’t borrow more than a predefined number of items (let's say 5) at a time. If a member tries to borrow more than 5 items, the system will block the action.

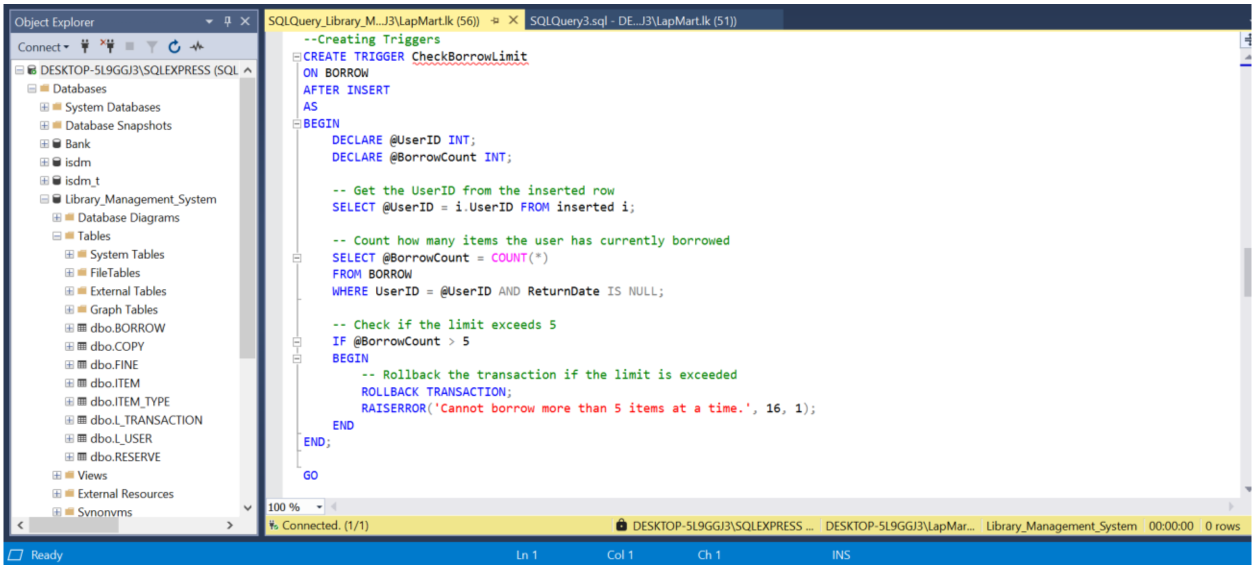
Explanation:

- **Purpose**: To prevent users from borrowing more than 5 items at the same time.
- **Trigger Type**: **AFTER INSERT** ensures the check occurs after an insertion into the **BORROW** table.
- **Logic**: After inserting a borrow record, the trigger checks how many items the user currently has borrowed (i.e., items not yet returned) and blocks the action if the count exceeds 5.

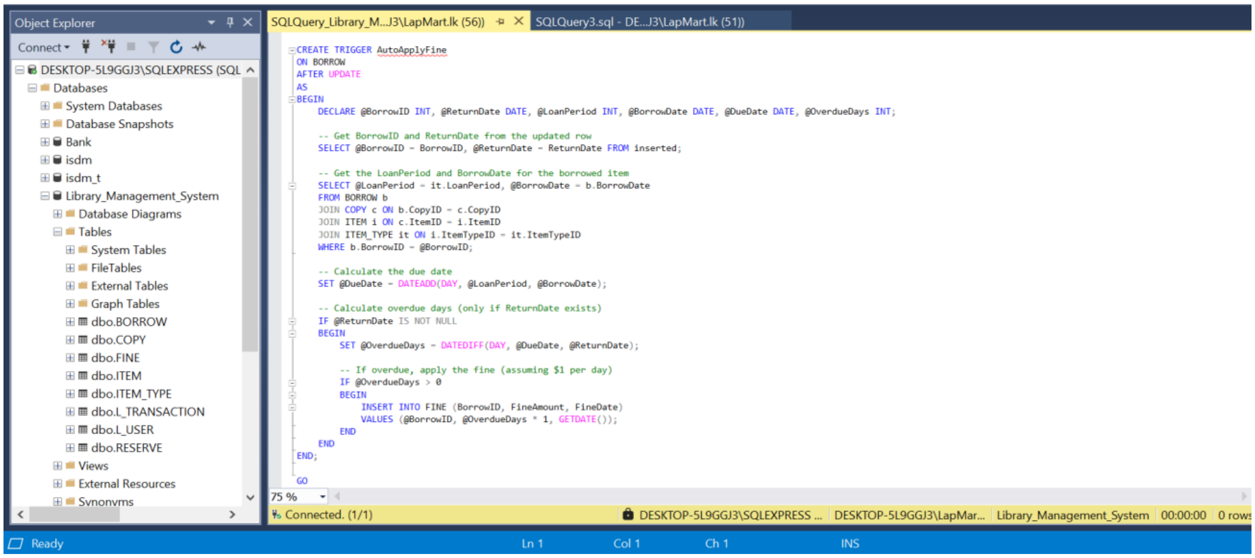
```
CREATE TRIGGER AutoApplyFine
ON BORROW
AFTER UPDATE
AS
BEGIN
    DECLARE @BorrowID INT, @ReturnDate DATE, @LoanPeriod INT, @BorrowDate DATE, @DueDate
DATE, @OverdueDays INT;

    -- Get BorrowID and ReturnDate from the updated row
    SELECT @BorrowID = BorrowID, @ReturnDate = ReturnDate FROM inserted;

    -- Get the LoanPeriod and BorrowDate for the borrowed item
    SELECT @LoanPeriod = it.LoanPeriod, @BorrowDate = b.BorrowDate
    FROM BORROW b
    JOIN COPY c ON b.CopyID = c.CopyID
    JOIN ITEM i ON c.ItemID = i.ItemID
    JOIN ITEM_TYPE it ON i.ItemTypeID = it.ItemTypeID
    WHERE b.BorrowID = @BorrowID;

    -- Calculate the due date
    SET @DueDate = DATEADD(DAY, @LoanPeriod, @BorrowDate);

    -- Calculate overdue days (only if ReturnDate exists)
    IF @ReturnDate IS NOT NULL
    BEGIN
        SET @OverdueDays = DATEDIFF(DAY, @DueDate, @ReturnDate);

        -- If overdue, apply the fine (assuming $1 per day)
        IF @OverdueDays > 0
        BEGIN
            INSERT INTO FINE (BorrowID, FineAmount, FineDate)
            VALUES (@BorrowID, @OverdueDays * 1, GETDATE());
        END
    END
END;
GO
```

The purpose of this trigger is to automatically calculate and apply a fine when a book is returned after its due date.

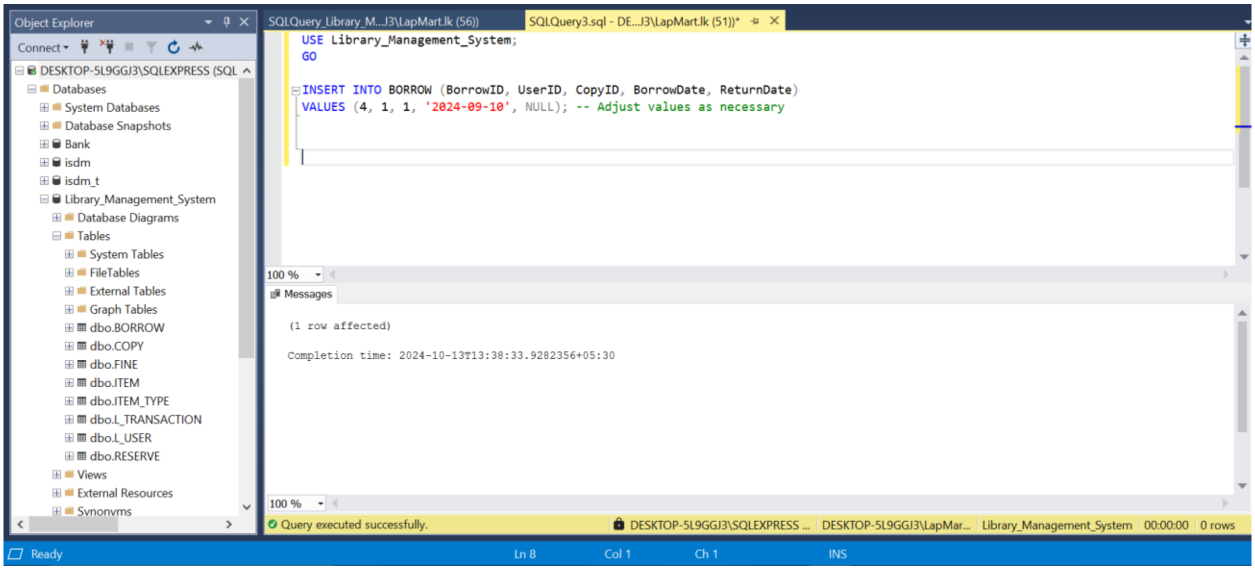
- It grabs the BorrowID and ReturnDate from the updated row.
- It looks up how long the item could be borrowed (loan period) and when it was borrowed.
- It calculates the due date by adding the loan period to the borrow date.
- If the item was returned after the due date, it calculates how many days it was overdue.
- Finally, if there are overdue days, it adds a fine to the FINE table, charging $1 for each overdue day.



```sql
--Creating Triggers
CREATE TRIGGER CheckBorrowLimit
ON BORROW
AFTER INSERT
AS
BEGIN
    DECLARE @UserID INT;
    DECLARE @BorrowCount INT;

    -- Get the UserID from the inserted row
    SELECT @UserID = i.UserID FROM inserted i;

    -- Count how many items the user has currently borrowed
    SELECT @BorrowCount = COUNT(*)
    FROM BORROW
    WHERE UserID = @UserID AND ReturnDate IS NULL;

    -- Check if the limit exceeds 5
    IF @BorrowCount > 5
    BEGIN
        -- Rollback the transaction if the limit is exceeded
        ROLLBACK TRANSACTION;
        RAISERROR('Cannot borrow more than 5 items at a time.', 16, 1);
    END
END;

GO
```

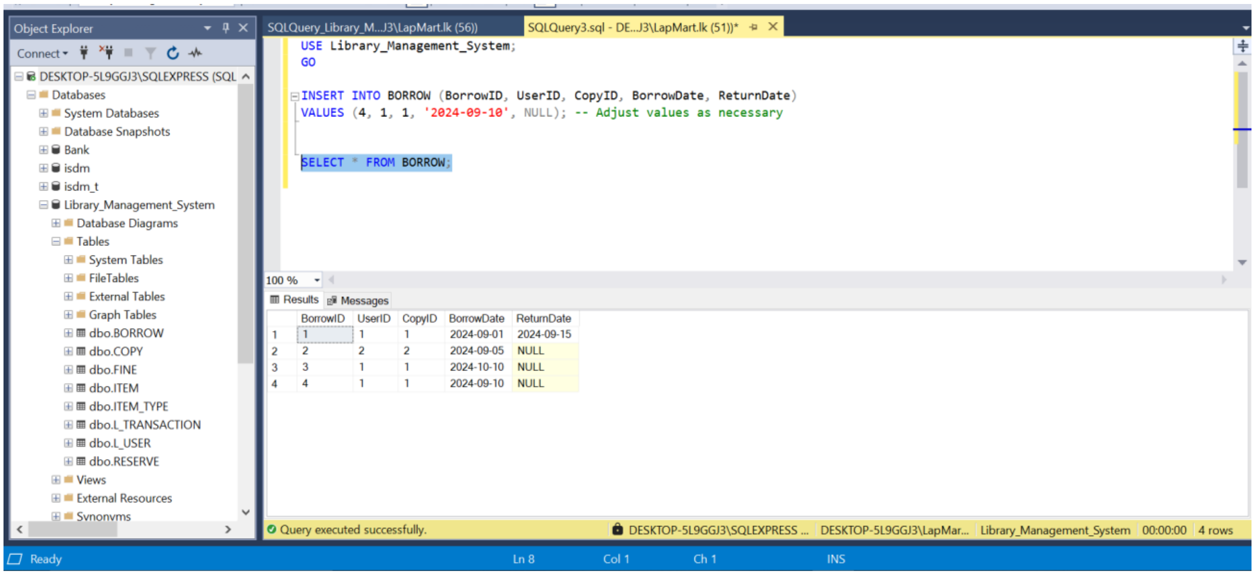


```sql
CREATE TRIGGER AutoApplyFine
ON BORROW
AFTER UPDATE
AS
BEGIN
    DECLARE @BorrowID INT, @ReturnDate DATE, @LoanPeriod INT, @BorrowDate DATE, @DueDate DATE, @OverdueDays INT;

    -- Get BorrowID and ReturnDate from the updated row
    SELECT @BorrowID = BorrowID, @ReturnDate = ReturnDate FROM inserted;

    -- Get the LoanPeriod and BorrowDate for the borrowed item
    SELECT @LoanPeriod = it.LoanPeriod, @BorrowDate = b.BorrowDate
    FROM BORROW b
    JOIN COPY c ON b.CopyID = c.CopyID
    JOIN ITEM i ON c.ItemID = i.ItemID
    JOIN ITEM_TYPE it ON i.ItemTypeID = it.ItemTypeID
    WHERE b.BorrowID = @BorrowID;

    -- Calculate the due date
    SET @DueDate = DATEADD(DAY, @LoanPeriod, @BorrowDate);

    -- Calculate overdue days (only if ReturnDate exists)
    IF @ReturnDate IS NOT NULL
    BEGIN
        SET @OverdueDays = DATEDIFF(DAY, @DueDate, @ReturnDate);

        -- If overdue, apply the fine (assuming $1 per day)
        IF @OverdueDays > 0
        BEGIN
            INSERT INTO FINE (BorrowID, FineAmount, FineDate)
            VALUES (@BorrowID, @OverdueDays * 1, GETDATE());
        END
    END
END;

GO
```



```sql
USE Library_Management_System;
GO

INSERT INTO BORROW (BorrowID, UserID, CopyID, BorrowDate, ReturnDate)
VALUES (4, 1, 1, '2024-09-10', NULL); -- Adjust values as necessary
```

```
(1 row affected)

Completion time: 2024-10-13T13:38:33.9282356+05:30
```



```sql
USE Library_Management_System;
GO

INSERT INTO BORROW (BorrowID, UserID, CopyID, BorrowDate, ReturnDate)
VALUES (4, 1, 1, '2024-09-10', NULL); -- Adjust values as necessary


SELECT * FROM BORROW;
```

	BorrowID	UserID	CopyID	BorrowDate	ReturnDate
1	1	1	1	2024-09-01	2024-09-15
2	2	2	2	2024-09-05	NULL
3	3	1	1	2024-10-10	NULL
4	4	1	1	2024-09-10	NULL

Query executed successfully.

2.5.5 Views

```sql
--Creating VIEWS
CREATE VIEW MemberBorrowedItems AS
SELECT
    u.UserID,
    u.Name AS MemberName,
    i.Title AS ItemTitle,
    b.BorrowDate,
    b.ReturnDate
FROM
    Lib_USER u
JOIN BORROW b ON u.UserID = b.UserID
JOIN COPY c ON b.CopyID = c.CopyID
JOIN ITEM i ON c.ItemID = i.ItemID
WHERE
    u.Role IN ('Student', 'Faculty', 'Staff');
```

- This view is for Members to easily see the list of items they have borrowed, including the borrow and return dates.

Explanation:

- **Purpose**: Provides members with a summary of all the items they have borrowed.
- **Joins**:
  - **USER** and **BORROW** tables are joined based on UserID.
  - **COPY** and **ITEM** tables are joined to get the Title of the borrowed item.
- **Filtering**: Limits the view to users whose role is either a student, faculty, or staff.
- The JOIN statements combine data from multiple tables based on matching values (e.g., user IDs and copy IDs).
- The WHERE clause filters out non-members, so only students, faculty, and staff are included.

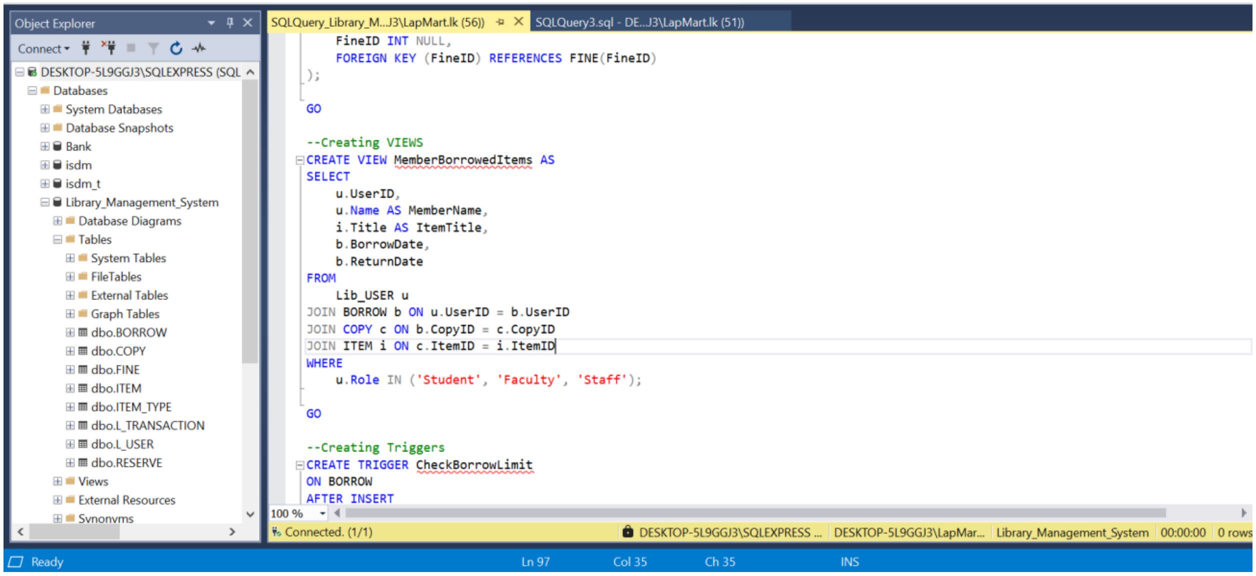
```
CREATE VIEW BorrowedItemsSummary AS
SELECT
    u.UserID,
    u.Name AS MemberName,
    u.Email AS MemberEmail,
    i.Title AS ItemTitle,
    b.BorrowDate,
    b.ReturnDate,
    COALESCE(f.FineAmount, 0) AS FineAmount
FROM
    Lib_USER u
JOIN
    BORROW b ON u.UserID = b.UserID
JOIN
    COPY c ON b.CopyID = c.CopyID
JOIN
    ITEM i ON c.ItemID = i.ItemID
LEFT JOIN
    FINE f ON b.BorrowID = f.BorrowID
WHERE
    b.ReturnDate IS NULL; -- This filters to only show currently borrowed items
GO
```

This view will show each borrowed item's details, the user who borrowed it, the borrow date, return date, and any applicable fine amount.

Explanation

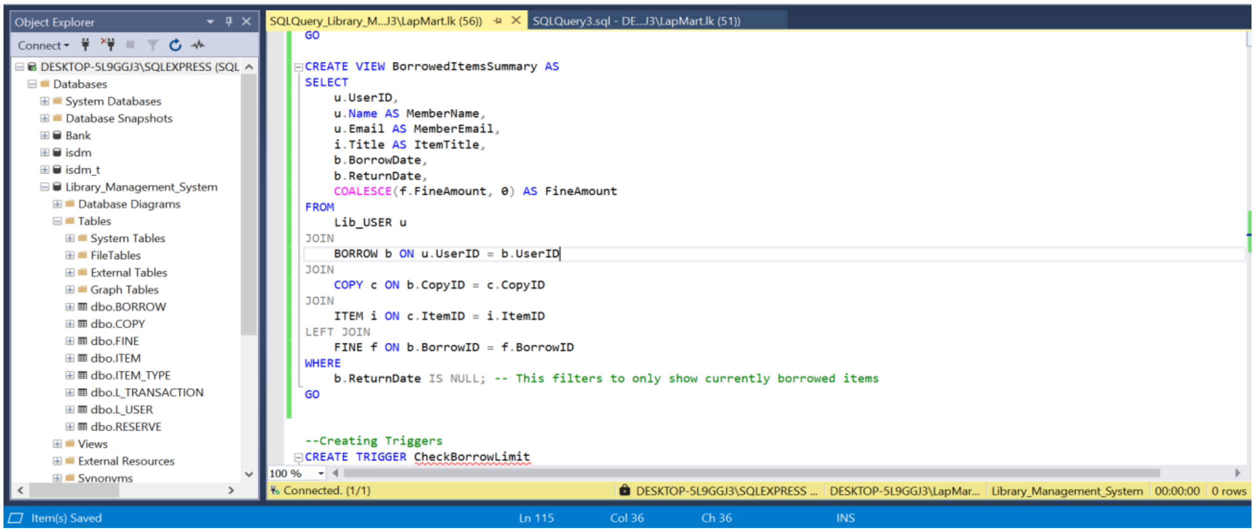
- **User Information**: The view retrieves user details, including UserID, Name, and Email.
- **Item Information**: It fetches the Title of the item borrowed.
- **Borrow Dates**: It includes the BorrowDate and ReturnDate, showing when the item was borrowed and if it has been returned yet.
- **Fine Amount**: It uses COALESCE to ensure that if there is no fine associated with the borrowing, it defaults to 0.
- **Current Borrowings**: The WHERE clause filters the results to show only those items that are currently borrowed (where ReturnDate is NULL).

This view will help library staff manage and monitor the currently borrowed items effectively. You can run the above SQL to create the view in your database.



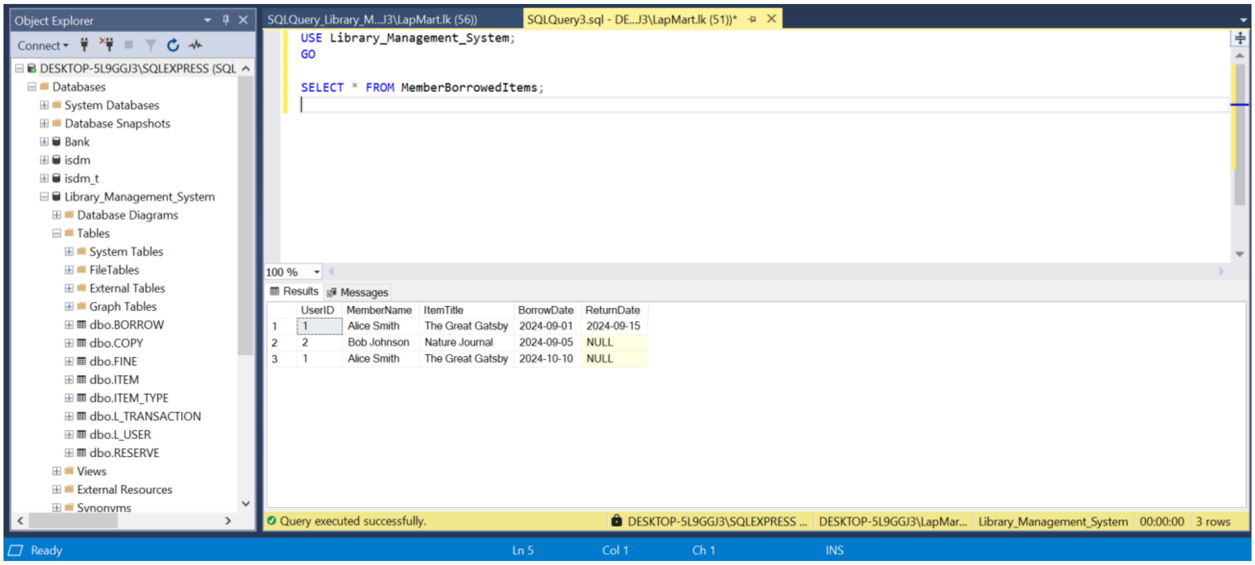
```
    FineID INT NULL,
    FOREIGN KEY (FineID) REFERENCES FINE(FineID)
);

GO

--Creating VIEWS
CREATE VIEW MemberBorrowedItems AS
SELECT
    u.UserID,
    u.Name AS MemberName,
    i.Title AS ItemTitle,
    b.BorrowDate,
    b.ReturnDate
FROM
    Lib_USER u
JOIN BORROW b ON u.UserID = b.UserID
JOIN COPY c ON b.CopyID = c.CopyID
JOIN ITEM i ON c.ItemID = i.ItemID
WHERE
    u.Role IN ('Student', 'Faculty', 'Staff');

GO

--Creating Triggers
CREATE TRIGGER CheckBorrowLimit
ON BORROW
AFTER INSERT
```

Connected. (1/1)



```
GO

CREATE VIEW BorrowedItemsSummary AS
SELECT
    u.UserID,
    u.Name AS MemberName,
    u.Email AS MemberEmail,
    i.Title AS ItemTitle,
    b.BorrowDate,
    b.ReturnDate,
    COALESCE(f.FineAmount, 0) AS FineAmount
FROM
    Lib_USER u
JOIN
    BORROW b ON u.UserID = b.UserID
JOIN
    COPY c ON b.CopyID = c.CopyID
JOIN
    ITEM i ON c.ItemID = i.ItemID
LEFT JOIN
    FINE f ON b.BorrowID = f.BorrowID
WHERE
    b.ReturnDate IS NULL; -- This filters to only show currently borrowed items
GO


--Creating Triggers
CREATE TRIGGER CheckBorrowLimit
```

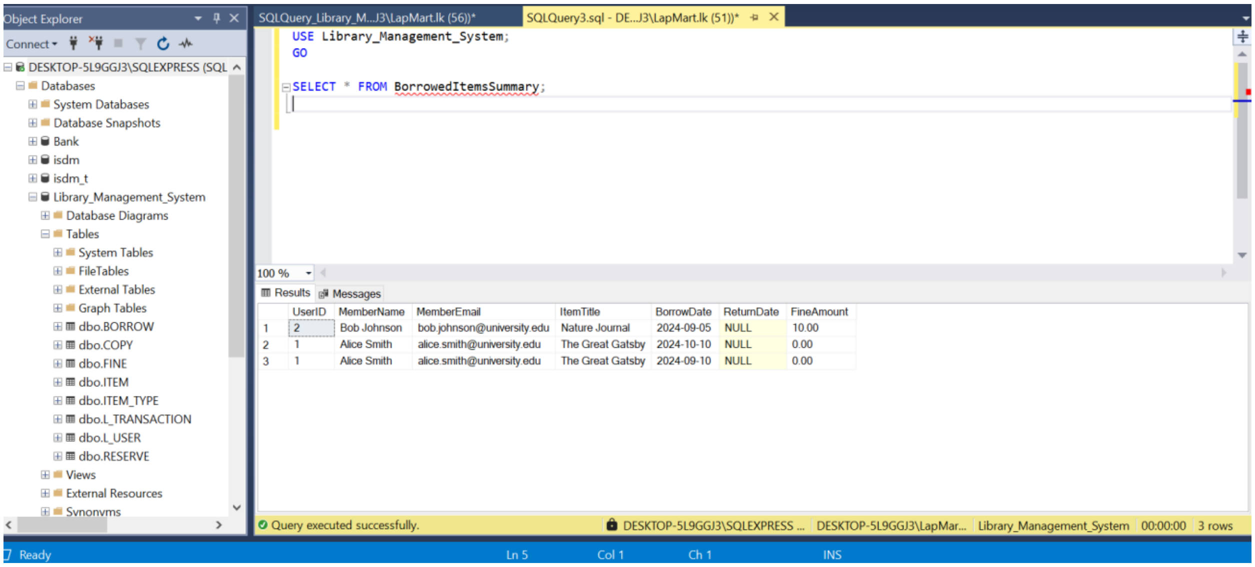
Connected. (1/1)



```
USE Library_Management_System;
GO

SELECT * FROM MemberBorrowedItems;
```

	UserID	MemberName	ItemTitle	BorrowDate	ReturnDate
1	1	Alice Smith	The Great Gatsby	2024-09-01	2024-09-15
2	2	Bob Johnson	Nature Journal	2024-09-05	NULL
3	1	Alice Smith	The Great Gatsby	2024-10-10	NULL

Query executed successfully.



```sql
USE Library_Management_System;
GO

SELECT * FROM BorrowedItemsSummary;
```

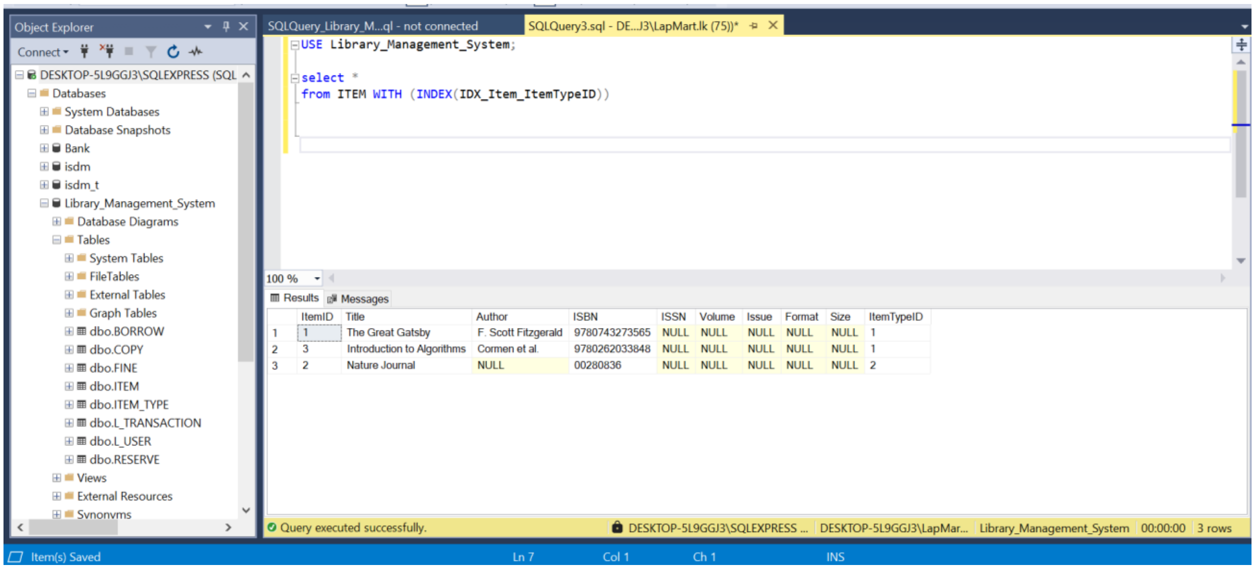
	UserID	MemberName	MemberEmail	ItemTitle	BorrowDate	ReturnDate	FineAmount
1	2	Bob Johnson	bob.johnson@university.edu	Nature Journal	2024-09-05	NULL	10.00
2	1	Alice Smith	alice.smith@university.edu	The Great Gatsby	2024-10-10	NULL	0.00
3	1	Alice Smith	alice.smith@university.edu	The Great Gatsby	2024-09-10	NULL	0.00

Query executed successfully.

2.5.6 Indexes

```sql
--Sample Data for ITEM
INSERT INTO ITEM (ItemID, Title, Author, ISBN, ItemTypeID) VALUES
(1, 'The Great Gatsby', 'F. Scott Fitzgerald', '9780743273565', 1),
(2, 'Nature Journal', NULL, '00280836', 2),
(3, 'Introduction to Algorithms', 'Cormen et al.', '9780262033848', 1);

-- Index on ItemTypeID for faster joins with ITEM_TYPE
CREATE INDEX IDX_Item_ItemTypeID ON ITEM (ItemTypeID);
-- Index on ISBN for faster searches on ISBN
CREATE INDEX IDX_Item_ISBN ON ITEM (ISBN);
-- Index on ISSN for faster searches on ISSN
CREATE INDEX IDX_Item_ISSN ON ITEM (ISSN);

--Sample Data for COPY
INSERT INTO COPY (CopyID, Condition, Location, ItemID) VALUES
(1, 'Good', 'Shelf A1', 1),
(2, 'Good', 'Shelf B2', 2),
(3, 'Fair', 'Shelf C3', 3);

-- Index on ItemID for faster joins with ITEM
CREATE INDEX IDX_Copy_ItemID ON COPY (ItemID);

--Sample Data for Lib_USER
INSERT INTO Lib_USER (UserID, Name, Email, PhoneNumber, Role) VALUES
(1, 'Alice Smith', 'alice.smith@university.edu', '123-456-7890', 'Student'),
(2, 'Bob Johnson', 'bob.johnson@university.edu', '098-765-4321', 'Faculty'),
(3, 'Carol Davis', 'carol.davis@university.edu', '555-555-5555', 'Staff');

-- Index on Email for faster searches on Email
CREATE INDEX IDX_LibUser_Email ON Lib_USER (Email);
-- Index on Role for filtering roles
CREATE INDEX IDX_LibUser_Role ON Lib_USER (Role);

--Sample Data for BORROW
INSERT INTO BORROW (BorrowID, UserID, CopyID, BorrowDate, ReturnDate) VALUES
(1, 1, 1, '2024-09-01', '2024-09-15'),
(2, 2, 3, '2024-09-05', NULL);

-- Index on UserID for faster joins and searches by UserID
CREATE INDEX IDX_Borrow_UserID ON BORROW (UserID);
-- Index on CopyID for faster joins and searches by CopyID
CREATE INDEX IDX_Borrow_CopyID ON BORROW (CopyID);
-- Index on ReturnDate for filtering current borrowed items
CREATE INDEX IDX_Borrow_ReturnDate ON BORROW (ReturnDate);

--Sample Data for RESERVE
INSERT INTO RESERVE (ReserveID, UserID, CopyID, ReserveDate) VALUES
(1, 3, 3, '2024-09-10');

-- Index on UserID for faster joins and searches by UserID
CREATE INDEX IDX_Reserve_UserID ON RESERVE (UserID);
-- Index on CopyID for faster joins and searches by CopyID
CREATE INDEX IDX_Reserve_CopyID ON RESERVE (CopyID);
```

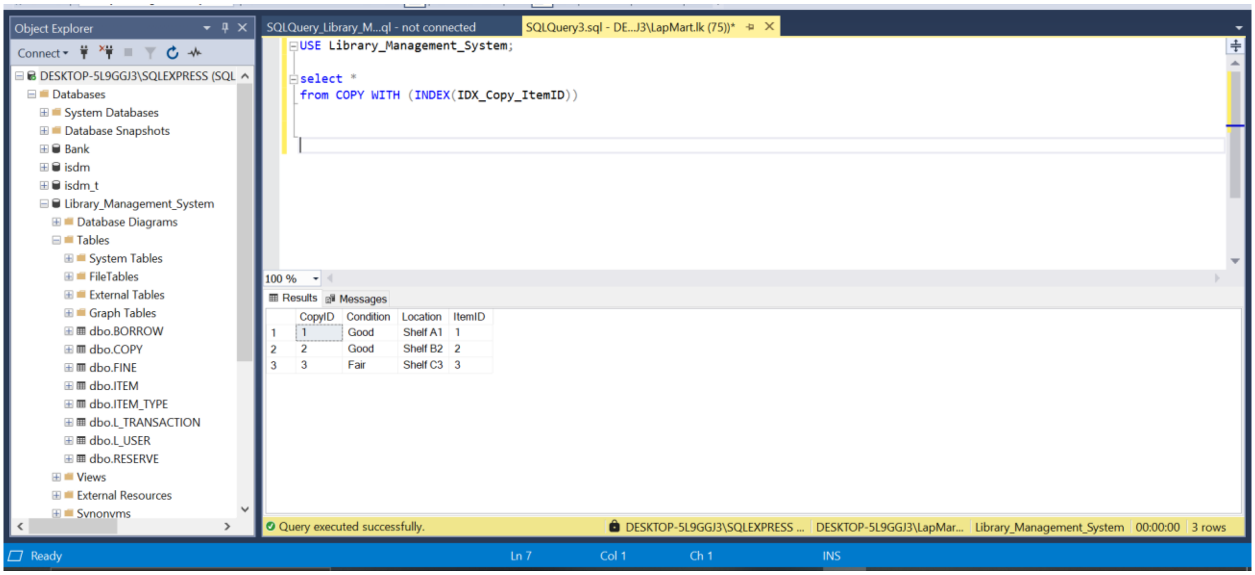
```sql
-- Sample Data for FINE with unique records
INSERT INTO FINE (FineID, BorrowID, FineAmount, FineDate) VALUES
(1, 2, 10.00, '2024-09-01'),  -- Existing fine
(2, 1, 5.00, '2024-09-10'),   -- New fine for BorrowID 1
(3, 2, 15.00, '2024-09-20');   -- New fine for BorrowID 2

-- Index on BorrowID for faster joins with BORROW
CREATE INDEX IDX_Fine_BorrowID ON FINE (BorrowID);

-- Sample Data for L_TRANSACTION
INSERT INTO L_TRANSACTION (TransactionID, TransactionType, TransactionDate, FineID)
VALUES
(1, 'Payment', '2024-09-15', 1),  -- Payment for the fine with FineID 1
(2, 'Payment', '2024-09-20', 2),  -- Payment for the fine with FineID 2
(3, 'Waiver', '2024-09-22', 1),   -- Waiver for the fine with FineID 1
(4, 'Payment', '2024-10-01', 3);  -- Payment for the fine with FineID 3

CREATE INDEX IDX_LTransaction_FineID ON L_TRANSACTION (FineID);
```



```sql
USE Library_Management_System;

select *
from ITEM WITH (INDEX(IDX_Item_ItemTypeID))
```

	ItemID	Title	Author	ISBN	ISSN	Volume	Issue	Format	Size	ItemTypeID
1	1	The Great Gatsby	F. Scott Fitzgerald	9780743273565	NULL	NULL	NULL	NULL	NULL	1
2	3	Introduction to Algorithms	Cormen et al.	9780262033848	NULL	NULL	NULL	NULL	NULL	1
3	2	Nature Journal	NULL	00280836	NULL	NULL	NULL	NULL	NULL	2

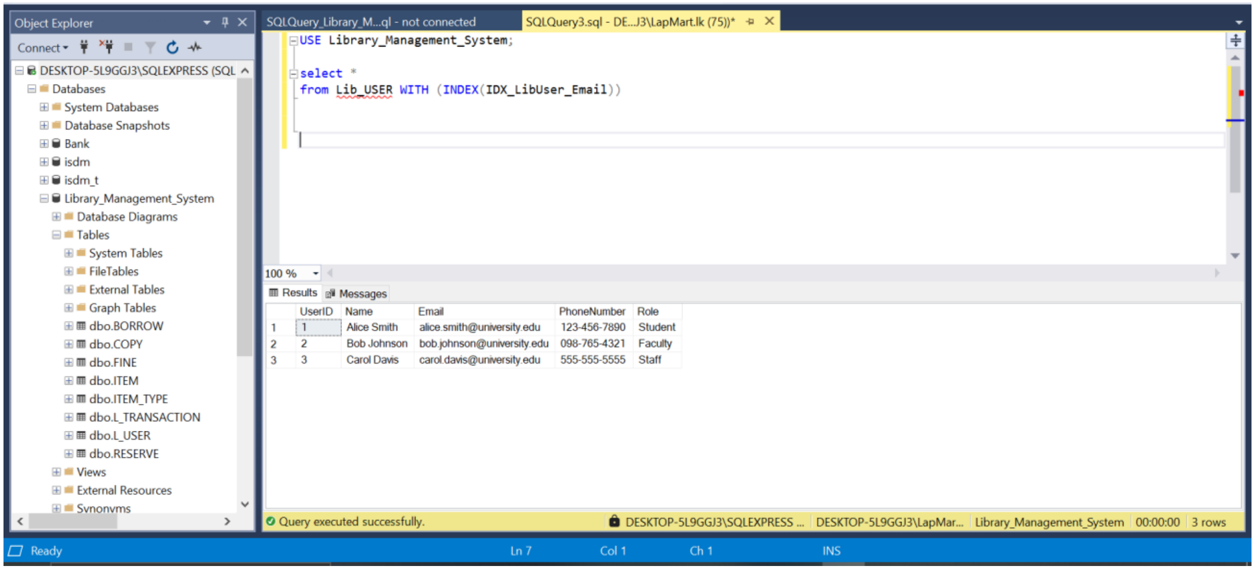
Query executed successfully. DESKTOP-5L9GGJ3\SQLEXPRESS ... DESKTOP-5L9GGJ3\LapMar... Library_Management_System 00:00:00 3 rows



```sql
USE Library_Management_System;

select *
from COPY WITH (INDEX(IDX_Copy_ItemID))
```

	CopyID	Condition	Location	ItemID
1	1	Good	Shelf A1	1
2	2	Good	Shelf B2	2
3	3	Fair	Shelf C3	3

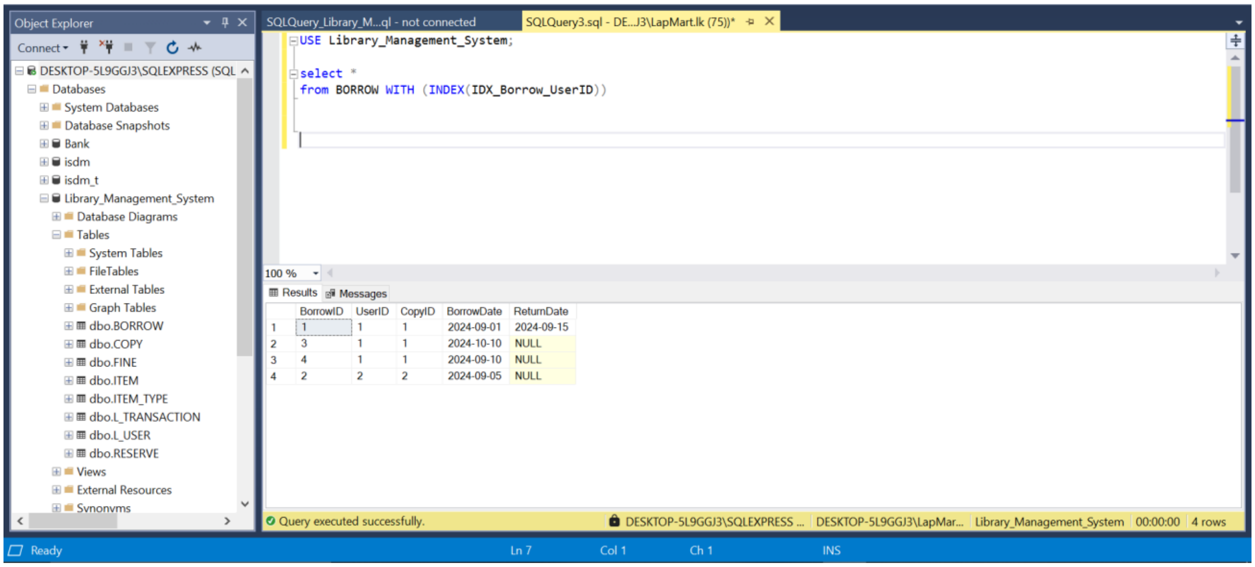
Query executed successfully. DESKTOP-5L9GGJ3\SQLEXPRESS ... DESKTOP-5L9GGJ3\LapMar... Library_Management_System 00:00:00 3 rows



```sql
USE Library_Management_System;

select *
from Lib_USER WITH (INDEX(IDX_LibUser_Email))
```

	UserID	Name	Email	PhoneNumber	Role
1	1	Alice Smith	alice.smith@university.edu	123-456-7890	Student
2	2	Bob Johnson	bob.johnson@university.edu	098-765-4321	Faculty
3	3	Carol Davis	carol.davis@university.edu	555-555-5555	Staff

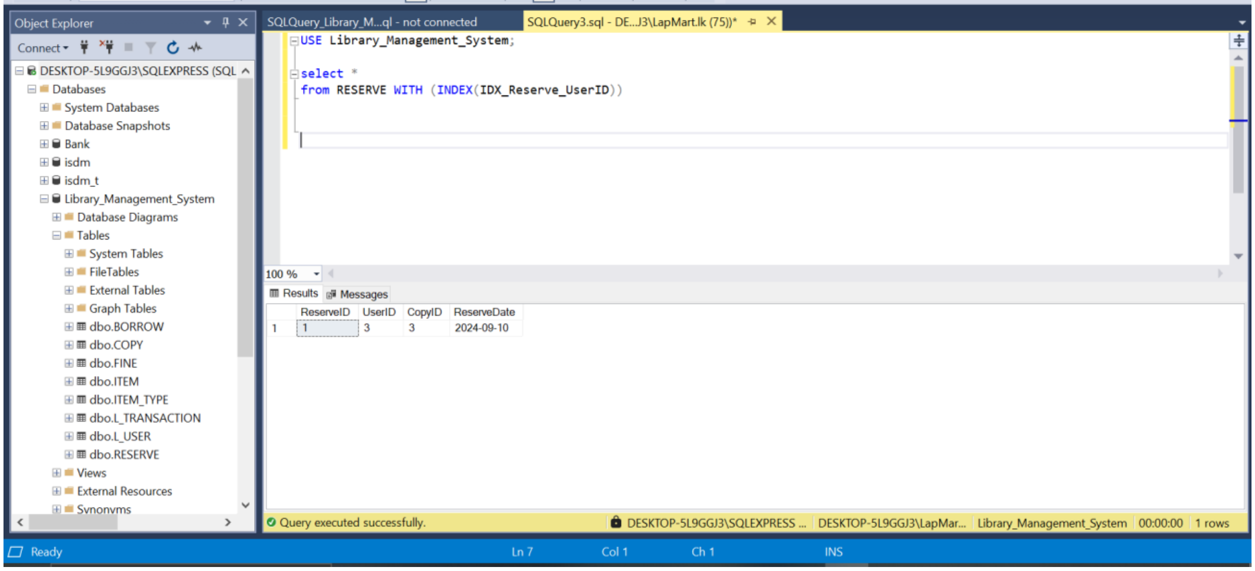
```
Query executed successfully.    DESKTOP-5L9GGJ3\SQLEXPRESS ...  DESKTOP-5L9GGJ3\LapMar...  Library_Management_System  00:00:00  3 rows
```



```sql
USE Library_Management_System;

select *
from BORROW WITH (INDEX(IDX_Borrow_UserID))
```

	BorrowID	UserID	CopyID	BorrowDate	ReturnDate
1	1	1	1	2024-09-01	2024-09-15
2	3	1	1	2024-10-10	NULL
3	4	1	1	2024-09-10	NULL
4	2	2	2	2024-09-05	NULL

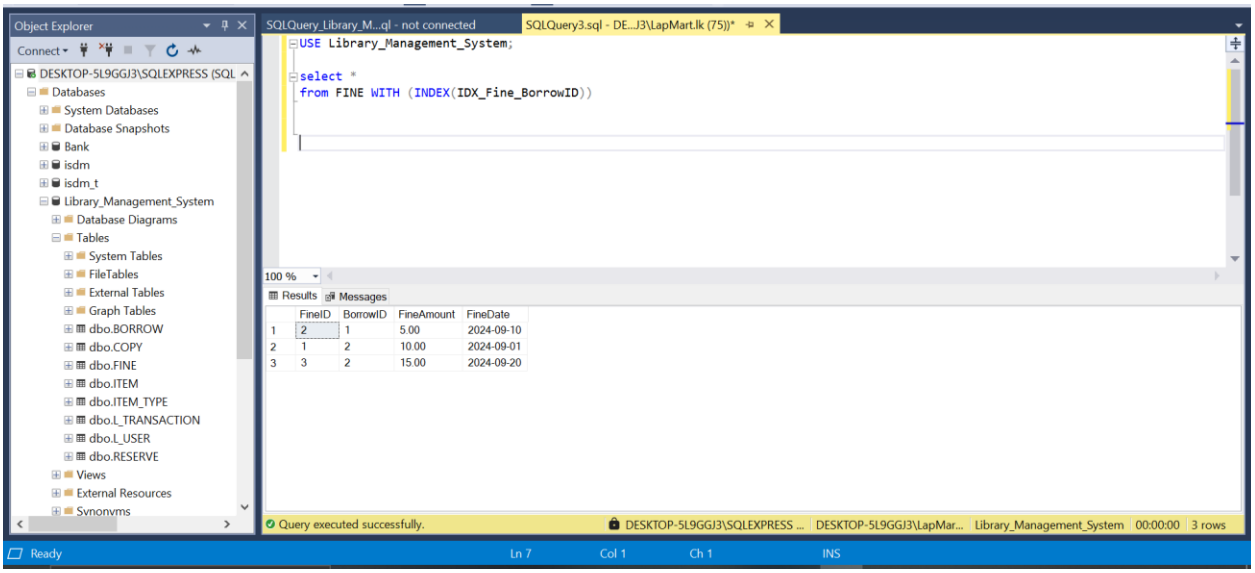
```
Query executed successfully.    DESKTOP-5L9GGJ3\SQLEXPRESS ...  DESKTOP-5L9GGJ3\LapMar...  Library_Management_System  00:00:00  4 rows
```



```sql
USE Library_Management_System;

select *
from RESERVE WITH (INDEX(IDX_Reserve_UserID))
```

	ReserveID	UserID	CopyID	ReserveDate
1	1	3	3	2024-09-10

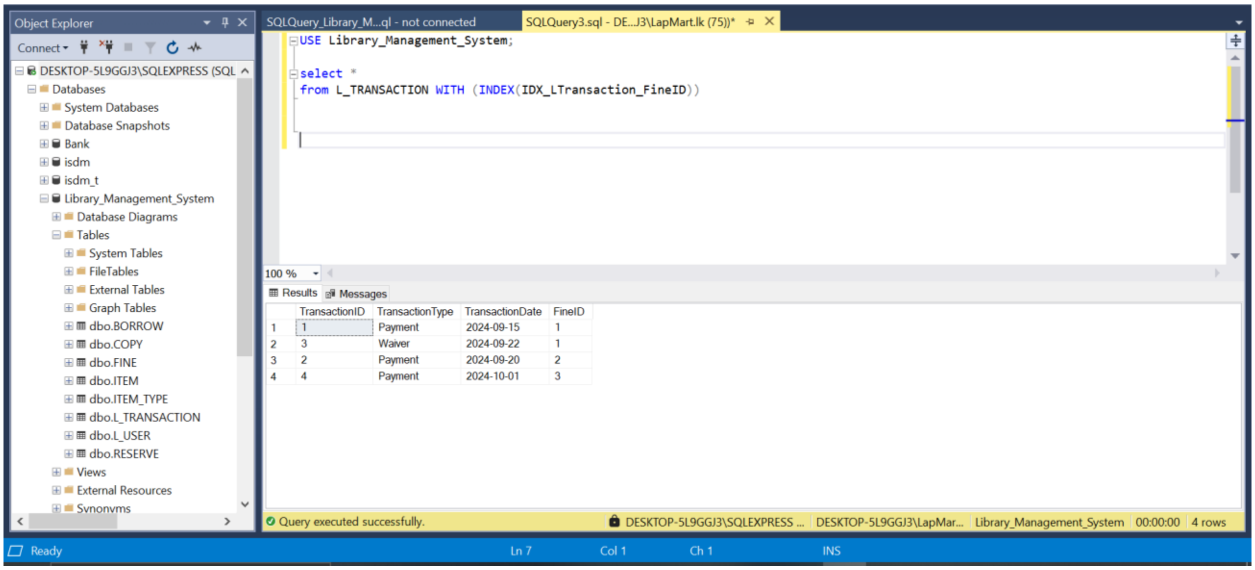
```
Query executed successfully.    DESKTOP-5L9GGJ3\SQLEXPRESS ...  DESKTOP-5L9GGJ3\LapMar...  Library_Management_System  00:00:00  1 rows
```



```
USE Library_Management_System;

select *
from FINE WITH (INDEX(IDX_Fine_BorrowID))
```

	FineID	BorrowID	FineAmount	FineDate
1	2	1	5.00	2024-09-10
2	1	2	10.00	2024-09-01
3	3	2	15.00	2024-09-20

Query executed successfully. DESKTOP-5L9GGJ3\SQLEXPRESS ... | DESKTOP-5L9GGJ3\LapMar... | Library_Management_System | 00:00:00 | 3 rows



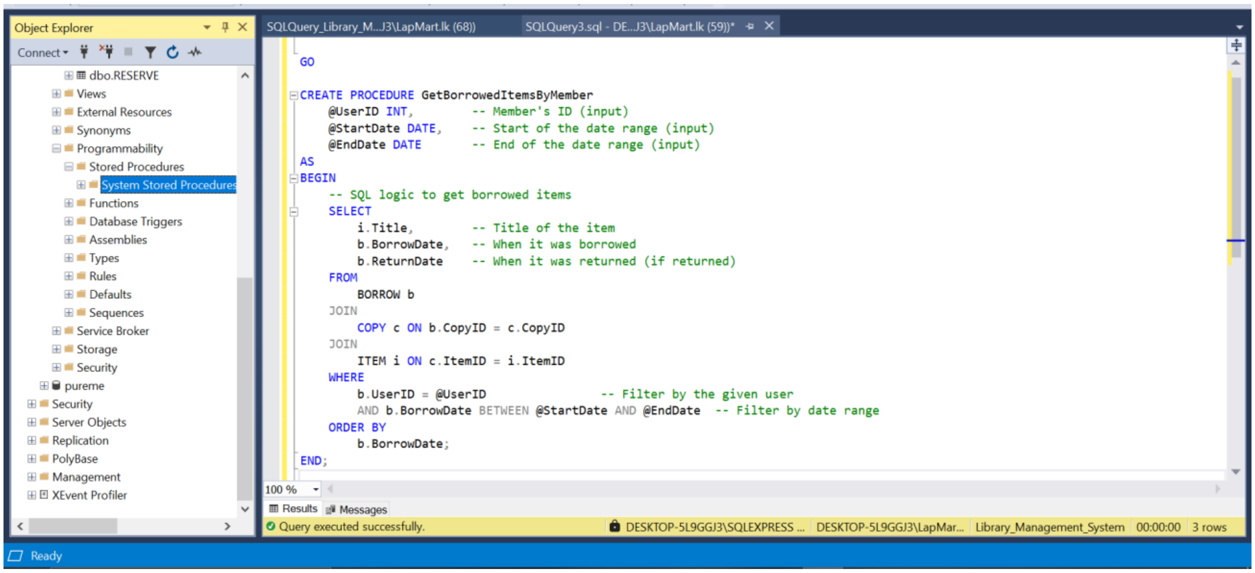
```
USE Library_Management_System;

select *
from L_TRANSACTION WITH (INDEX(IDX_LTransaction_FineID))
```

	TransactionID	TransactionType	TransactionDate	FineID
1	1	Payment	2024-09-15	1
2	3	Waiver	2024-09-22	1
3	2	Payment	2024-09-20	2
4	4	Payment	2024-10-01	3

Query executed successfully. DESKTOP-5L9GGJ3\SQLEXPRESS ... | DESKTOP-5L9GGJ3\LapMar... | Library_Management_System | 00:00:00 | 4 rows

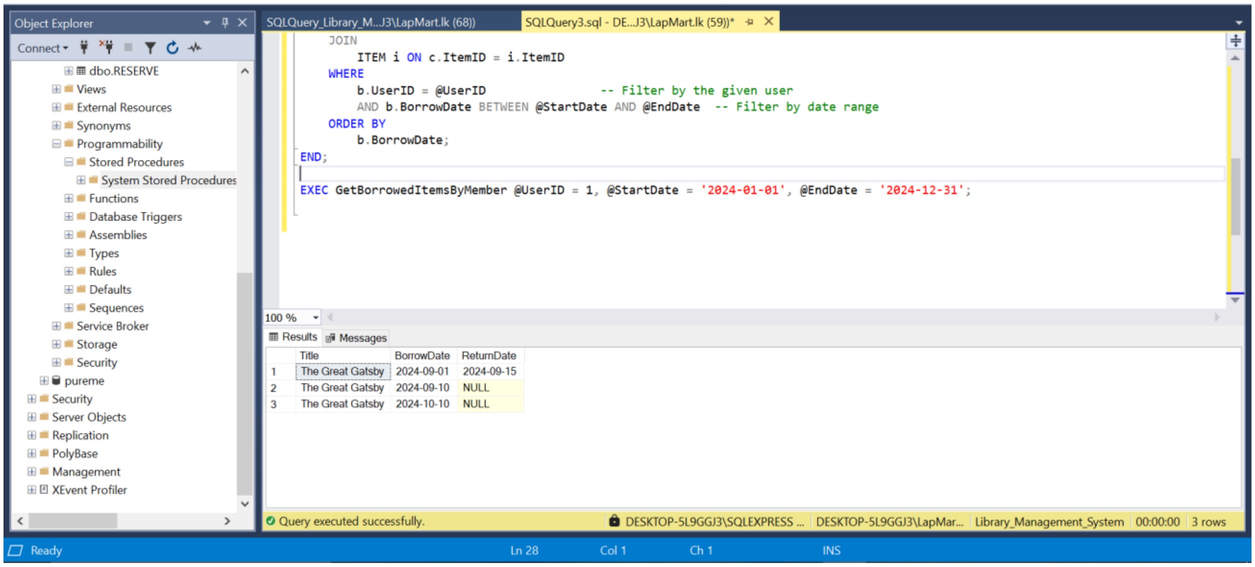
2.5.7 Procedures

```sql
--Create stored procedure for retrieving borrowed items by a member within a date range
CREATE PROCEDURE GetBorrowedItemsByMember
    @UserID INT,         -- Member's ID (input)
    @StartDate DATE,     -- Start of the date range (input)
    @EndDate DATE        -- End of the date range (input)
AS
BEGIN
    -- SQL logic to get borrowed items
    SELECT
        i.Title,         -- Title of the item
        b.BorrowDate,    -- When it was borrowed
        b.ReturnDate     -- When it was returned (if returned)
    FROM
        BORROW b
    JOIN
        COPY c ON b.CopyID = c.CopyID
    JOIN
        ITEM i ON c.ItemID = i.ItemID
    WHERE
        b.UserID = @UserID                  -- Filter by the given user
        AND b.BorrowDate BETWEEN @StartDate AND @EndDate  -- Filter by date range
    ORDER BY
        b.BorrowDate;
END;
```



```sql
GO

CREATE PROCEDURE GetBorrowedItemsByMember
    @UserID INT,         -- Member's ID (input)
    @StartDate DATE,     -- Start of the date range (input)
    @EndDate DATE        -- End of the date range (input)
AS
BEGIN
    -- SQL logic to get borrowed items
    SELECT
        i.Title,         -- Title of the item
        b.BorrowDate,    -- When it was borrowed
        b.ReturnDate     -- When it was returned (if returned)
    FROM
        BORROW b
    JOIN
        COPY c ON b.CopyID = c.CopyID
    JOIN
        ITEM i ON c.ItemID = i.ItemID
    WHERE
        b.UserID = @UserID                  -- Filter by the given user
        AND b.BorrowDate BETWEEN @StartDate AND @EndDate  -- Filter by date range
    ORDER BY
        b.BorrowDate;
END;
```

Query executed successfully. DESKTOP-5L9GGJ3\SQLEXPRESS ... DESKTOP-5L9GGJ3\LapMar... Library_Management_System 00:00:00 3 rows



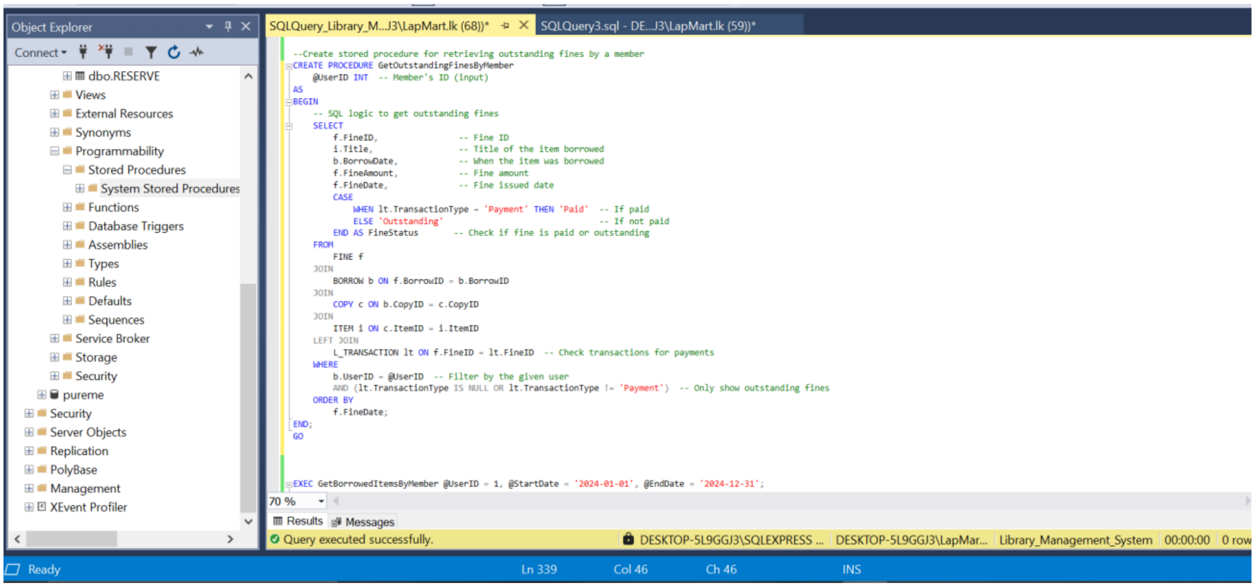
```sql
    JOIN
        ITEM i ON c.ItemID = i.ItemID
    WHERE
        b.UserID = @UserID                -- Filter by the given user
        AND b.BorrowDate BETWEEN @StartDate AND @EndDate  -- Filter by date range
    ORDER BY
        b.BorrowDate;
END;

EXEC GetBorrowedItemsByMember @UserID = 1, @StartDate = '2024-01-01', @EndDate = '2024-12-31';
```

	Title	BorrowDate	ReturnDate
1	The Great Gatsby	2024-09-01	2024-09-15
2	The Great Gatsby	2024-09-10	NULL
3	The Great Gatsby	2024-10-10	NULL

Query executed successfully. DESKTOP-5L9GGJ3\SQLEXPRESS ... DESKTOP-5L9GGJ3\LapMar... Library_Management_System 00:00:00 3 rows

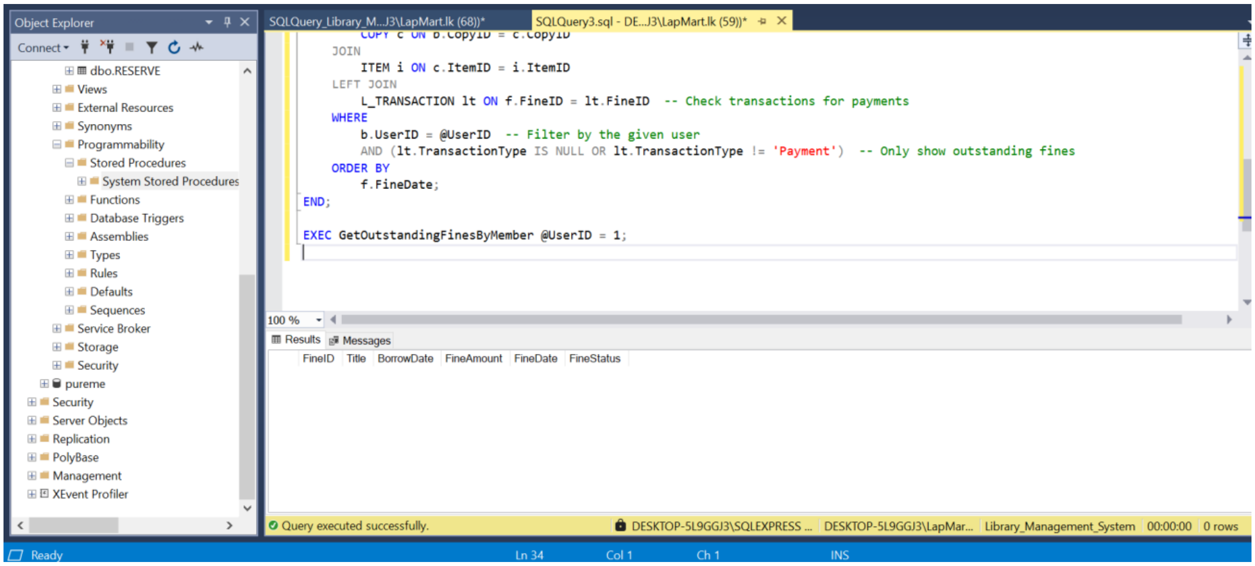
```sql
GO

-- Create stored procedure for retrieving outstanding fines by a member
CREATE PROCEDURE GetOutstandingFinesByMember
    @UserID INT  -- Member's ID (input)
AS
BEGIN
    -- Write the SQL logic to get outstanding fines
    SELECT
        f.FineID,                   -- Fine ID
        i.Title,                    -- Title of the item borrowed
        b.BorrowDate,               -- When the item was borrowed
        f.FineAmount,               -- Fine amount
        f.FineDate,                 -- Fine issued date
        CASE
            WHEN lt.TransactionType = 'Payment' THEN 'Paid'  -- If paid
            ELSE 'Outstanding'                              -- If not paid
        END AS FineStatus         -- Check if fine is paid or outstanding
    FROM
        FINE f
    JOIN
        BORROW b ON f.BorrowID = b.BorrowID
    JOIN
        COPY c ON b.CopyID = c.CopyID
    JOIN
        ITEM i ON c.ItemID = i.ItemID
    LEFT JOIN
        L_TRANSACTION lt ON f.FineID = lt.FineID  -- Check transactions for payments
    WHERE
        b.UserID = @UserID  -- Filter by the given user
        AND (lt.TransactionType IS NULL OR lt.TransactionType != 'Payment')  -- Only show
outstanding fines
    ORDER BY
```

```
        f.FineDate;
END;
```



```
--Create stored procedure for retrieving outstanding fines by a member
CREATE PROCEDURE GetOutstandingFinesByMember
    @UserID INT  -- Member's ID (input)
AS
BEGIN
    -- SQL logic to get outstanding fines
    SELECT
        f.FineID,               -- Fine ID
        i.Title,                -- Title of the item borrowed
        b.BorrowDate,           -- When the item was borrowed
        f.FineAmount,           -- Fine amount
        f.FineDate,             -- Fine issued date
        CASE
            WHEN lt.TransactionType = 'Payment' THEN 'Paid'  -- If paid
            ELSE 'Outstanding'                               -- If not paid
        END AS FineStatus      -- Check if fine is paid or outstanding
    FROM
        FINE f
    JOIN
        BORROW b ON f.BorrowID = b.BorrowID
    JOIN
        COPY c ON b.CopyID = c.CopyID
    JOIN
        ITEM i ON c.ItemID = i.ItemID
    LEFT JOIN
        L_TRANSACTION lt ON f.FineID = lt.FineID  -- Check transactions for payments
    WHERE
        b.UserID = @UserID  -- Filter by the given user
        AND (lt.TransactionType IS NULL OR lt.TransactionType != 'Payment')  -- Only show outstanding fines
    ORDER BY
        f.FineDate;
END;
GO


EXEC GetBorrowedItemsByMember @UserID = 1, @StartDate = '2024-01-01', @EndDate = '2024-12-31';
```

Query executed successfully. | DESKTOP-5L9GGJ3\SQLEXPRESS ... | DESKTOP-5L9GGJ3\LapMar... | Library_Management_System | 00:00:00 | 0 row



```
        COPY c ON b.CopyID = c.CopyID
    JOIN
        ITEM i ON c.ItemID = i.ItemID
    LEFT JOIN
        L_TRANSACTION lt ON f.FineID = lt.FineID  -- Check transactions for payments
    WHERE
        b.UserID = @UserID  -- Filter by the given user
        AND (lt.TransactionType IS NULL OR lt.TransactionType != 'Payment')  -- Only show outstanding fines
    ORDER BY
        f.FineDate;
END;

EXEC GetOutstandingFinesByMember @UserID = 1;
```

FineID	Title	BorrowDate	FineAmount	FineDate	FineStatus

Query executed successfully. | DESKTOP-5L9GGJ3\SQLEXPRESS ... | DESKTOP-5L9GGJ3\LapMar... | Library_Management_System | 00:00:00 | 0 rows

3. Part II

3.1 SQL Injection

Description of Vulnerability:

SQL injection, more commonly known as SQLi, is a well-known type of attack where an attacker injects malicious SQL code into input fields to manipulate the SQL queries of a web application. The aim is to have the database execute unintended commands. These are some fairly dangerous things because they have been known to allow hackers to:

- Bypass authentication mechanisms.
- Access, modify, or delete sensitive data from the database.
- Execute administrative operations on the database.
- Retrieve contents of entire databases, exposing confidential information.

Techniques of SQL Injection:

1. **Union-Based SQL Injection**: Such an attack would exploit the weak input fields by incorporating an evil UNION statement from an attacker in order to retrieve data from other tables. For example, in a login form, an attacker may provide:

OR 1=1 UNION SELECT username, password FROM users

2. **Error-Based SQL Injection:** This technique relies on database error messages to glean information about the structure of the database. Through techniques like tainting input fields and observing the resultant error responses, attackers can typically infer table names, column names, and data types, thus enabling them to craft more focused attacks..

Impact of SQL Injection:

- **Data Breach :**Attackers may gain access to private information such as PII, Finances details and login information**.**
- **Data Integrity Compromise:** Data may be altered, erased, or lost due to an attack which results in distorted data sets.

- **Loss of Trust and Reputation:** One of these is Attacks involving SQL Injection, which can harm an enterprise's brand image especially when there is compromise on sensitive information.
- **Service Downtime:** In severe cases, attackers can bring down a database, rendering services unavailable.
- **Mitigation and Countermeasures:**

1. **Input Validation**: Make sure to validate and sanitize all inputs before carrying out any processing on them. Make use of input validation rules that prohibit certain characters and disallow any potentially dangerous inputs
2. **Parameterized Queries/Prepared Statements:**

Applying parameterized queries and prepared statements, where user inputs are added as parameters and not as part of the SQL statement. This eliminates the risk of SQL code from being inserted and run

3. **Stored Procedures:** The interaction with the database must be carried out with the use of stored procedures. These are precompiled and if used properly prevent the SQL injection.
4. **Web Application Firewalls (WAF):**

Utilize web application firewalls (WAFs) to enhance security by recognizing and preventing SQL injection attempts through comprehensive query filtering against the database.

5. **Error Handling:**

Introduce suitable error management strategies to prevent information leakage about database error messages from being accessed by the user. Make sure to turn off verbose error message output in live systems.

3.2 Privilege Escalation

Description of Vulnerability: Many database applications include built-in permissions to prevent access to certain data for every user or class of user. Privilege escalation occurs in cases when an attacker obtains access to the database at some limited level, and, through various means, raises that access to a level where they might have, for example, a database administrator's (DBA) authority. This can happen due to wrong database or application design or due to access control weaknesses.

Techniques:

1. **Horizontal Privilege Escalation**: An invader having a valid user access may take advantage of application vulnerabilities to hijack other users accounts without permission. Typically, this involves either altering the session identifiers or adjusting the rate control settings.
2. **Vertical Privilege Escalation**: In this strategy, the assailant endeavors to escalate his privileges (for example get access to the database as a DBA). This may be achieved via security flaws that have not been patched, through the presence of weak password policies, or by taking advantage of misconfigured services that run with elevated privileges.

Impact of Privilege Escalation:

- **Unauthorized Data Access:** The assailant could access confidential data which may lead to its disclosure or removal.
- **Data Modification:** Elevated privileges enable the attacker to alter or erase vital information thereby causing possible interruptions in business activities.
- **Complete Control :** Once administrative permissions are obtained, assailants are capable of usurping authority rights over the DB, including creating or removing directories (in the case of user accounts), stopping services, or even erasing complete contents of databases

3.3 Mitigation and Countermeasures:

1. **Least Privilege Principle:**

Regarding database users, make sure that each of them gets no more than the privileges that they require to perform their work. For example, normal users should not be allowed to perform any administrative work, nor should any sensitive tables be readily accessible.

2. **Role-Based Access Control (RBAC):**

Establish Role Based Access Control mechanisms wherein user access to the database is restricted based on the classification of user's role. This makes certain that only the right individuals can carry out the perform high privy actions**.**

3. **Strong Authentication:**

Employ robust multi-factor authentication (MFA) to all accounts including but not limited to high privileged ones. Implement measures where users are required to create complex passwords and change them within determined periods.

4. **Audit and Monitoring:**

Ensure that all database activities including but not limited to Users Account activity, privileges assigned to users, and any administrative activities conducted are logged and audited. Conduct periodic log reviews to check for any suspicious or unauthorized activity.

5. **Patch Management:**

Continually maintain and enhance the database management system (DBMS) and its foundational program by performing the necessary updates and patches so as to fix any known weaknesses that may be available to assail for the purpose of escalating privileges.

4. Contribution

	Student ID 1	Student ID 2	Student ID 3	Student ID 4
Task 1				
Properly Documented Assumptions			✓	
ERD and Logical Model	✓	✓		
Normalization			✓	
Table Implementations	✓	✓		✓
Constraints Implementation			✓	✓
2 Triggers			✓	✓
2 Views	✓	✓		✓
2 Indexes	✓	✓		✓
2 Stored Procedures				✓
Task 2				
Description and analysis of 2 database vulnerabilities			✓	
Mitigation and countermeasure suggestions			✓	
Total				

IT23289802 KUMARARATHNA B.A.G.S.N	IT23292918 KUMARAGE S.D.	IT23293830 JAYASEKARA H.D.P.P	IT23298408 MINIPURA P T D
Member 1	**Member 2**	**Member 3**	**Member 4**