

EXPERIMENT: 10

NAME : PARIJAT PAL

UID : 23BCC70037

Part A: Insert Multiple Fee Payments in a Transaction

Description:

Given a table **FeePayments**, the task is to simulate a transaction where multiple payment entries are inserted at once. The goal is to demonstrate that all inserts happen successfully together as a single transaction unit (Atomicity).

Input Format:

- Table **FeePayments** with columns:
 - **payment_id** (INT, Primary Key)
 - **student_name** (VARCHAR(100))
 - **amount** (DECIMAL(10,2))
 - **payment_date** (DATE)

Output Format:

List of newly inserted payment records when the transaction is committed.

Constraints:

- Each payment has a unique ID.
- All inserts must succeed together as one unit of work.

Sample Input:

FeePayments

payment_id	student_name	amount	payment_date
1	Ashish	5000.00	2024-06-01
2	Smaran	4500.00	2024-06-02
3	Vaibhav	5500.00	2024-06-03

Sample Output:

payment_id	student_name	amount	payment_date
1	Ashish	5000.00	2024-06-01
2	Smaran	4500.00	2024-06-02
3	Vaibhav	5500.00	2024-06-03

Query:

DROP TABLE IF EXISTS FeePayments;

```
CREATE TABLE FeePayments (  
    payment_id INT PRIMARY KEY,  
    student_name VARCHAR(100),  
    amount DECIMAL(10,2),  
    payment_date DATE  
);
```

BEGIN TRANSACTION;

INSERT INTO FeePayments (payment_id, student_name, amount, payment_date)

VALUES

(1, 'Ashish', 5000.00, '2024-06-01'),

(2, 'Smaran', 4500.00, '2024-06-02'),

(3, 'Vaibhav', 5500.00, '2024-06-03');

COMMIT;

SELECT * FROM FeePayments;

OUTPUT:

The screenshot shows the ByteXL SQL editor interface. The top bar includes the ByteXL logo, a menu icon, a back arrow, a gear icon, a grid icon, a user profile icon labeled 'PARIJAT PAL', and buttons for 'AI', 'NEW', 'MYSQL', and 'RUN'. The main editor area displays a SQL script in a file named 'queries.sql'.

```
1 DROP TABLE IF EXISTS FeePayments;
2
3 CREATE TABLE FeePayments (
4     payment_id INT PRIMARY KEY,
5     student_name VARCHAR(100),
6     amount DECIMAL(10,2),
7     payment_date DATE
8 ) ENGINE=InnoDB;
9
10 START TRANSACTION;
11
12 INSERT INTO FeePayments (payment_id, student_name, amount, payment_date)
13 VALUES
14 (1, 'Ashish', 5000.00, '2024-06-01'),
15 (2, 'Smaran', 4500.00, '2024-06-02'),
16 (3, 'Vaibhav', 5500.00, '2024-06-03');
```

On the right side, the 'STDIN' section has an input field labeled 'Input for the program (Optional)'. Below it, the 'Output:' section displays a table with the following data:

payment_id	student_name	amount	payment_date
1	Ashish	5000.00	2024-06-01
2	Smaran	4500.00	2024-06-02
3	Vaibhav	5500.00	2024-06-03

Explanation:

This transaction ensures that **either all inserts succeed or none do**, demonstrating **Atomicity**. The **COMMIT** makes changes durable.

Part B: Demonstrate ROLLBACK for Failed Payment Insertion

Description:

Simulate a transaction failure in a **FeePayments** table by attempting to insert an invalid payment (e.g., duplicate `payment_id`). Use `ROLLBACK` to undo the entire transaction and demonstrate **Atomicity** and **Consistency** — ensuring that no partial data is committed to the table.

Input Format:

- Table **FeePayments** with columns:
 - `payment_id` (INT, Primary Key)
 - `student_name` (VARCHAR(100))
 - `amount` (DECIMAL(10,2))
 - `payment_date` (DATE)

Output Format:

No new records should be present from the failed transaction after `ROLLBACK`.

Constraints:

- `payment_id` must be unique.
- `amount` must be a positive decimal.
- If any operation in the transaction fails, the entire transaction must be rolled back.

Sample Input:

Initial successful inserts:

payment_id	student_name	amount	payment_date
1	Ashish	5000.00	2024-06-01
2	Smaran	4500.00	2024-06-02

payment_id	student_name	amount	payment_date
3	Vaibhav	5500.00	2024-06-03

Transaction with failure (duplicate ID = 1):

Sample Output:

Only the first 3 valid records should exist after rollback:

payment_id	student_name	amount	payment_date
1	Ashish	5000.00	2024-06-01
2	Smaran	4500.00	2024-06-02
3	Vaibhav	5500.00	2024-06-03

Query:

```
DROP TABLE IF EXISTS FeePayments;
```

```
CREATE TABLE FeePayments (
```

```
    payment_id INT PRIMARY KEY,
```

```
    student_name VARCHAR(100),
```

```
    amount DECIMAL(10,2) CHECK (amount > 0),
```

```
    payment_date DATE
```

```
) ENGINE=InnoDB;
```

```
START TRANSACTION;
```

```
INSERT INTO FeePayments (payment_id, student_name, amount, payment_date)
```

```
VALUES
```

```
(1, 'Ashish', 5000.00, '2024-06-01'),
```

(2, 'Smaran', 4500.00, '2024-06-02'),

(3, 'Vaibhav', 5500.00, '2024-06-03');

COMMIT;

SELECT * FROM FeePayments;

START TRANSACTION;

INSERT INTO FeePayments (payment_id, student_name, amount, payment_date)

VALUES

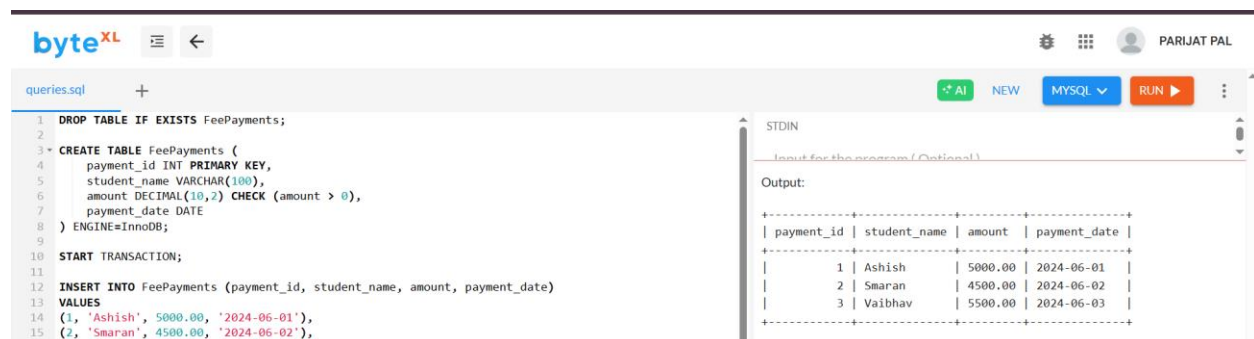
(4, 'Kiran', 4000.00, '2024-06-04'),

(1, 'Ashish', -500.00, '2024-06-05');

ROLLBACK;

SELECT * FROM FeePayments;

OUTPUT:



The screenshot shows a MySQL query editor with the following SQL code:

```
1 DROP TABLE IF EXISTS FeePayments;
2
3 CREATE TABLE FeePayments (
4     payment_id INT PRIMARY KEY,
5     student_name VARCHAR(100),
6     amount DECIMAL(10,2) CHECK (amount > 0),
7     payment_date DATE
8 ) ENGINE=InnoDB;
9
10 START TRANSACTION;
11
12 INSERT INTO FeePayments (payment_id, student_name, amount, payment_date)
13 VALUES
14 (1, 'Ashish', 5000.00, '2024-06-01'),
15 (2, 'Smaran', 4500.00, '2024-06-02'),
16 (3, 'Vaibhav', 5500.00, '2024-06-03');
```

The output window shows the result of the query:

payment_id	student_name	amount	payment_date
1	Ashish	5000.00	2024-06-01
2	Smaran	4500.00	2024-06-02
3	Vaibhav	5500.00	2024-06-03

Explanation:

- The **first transaction** inserts 3 valid records and is committed.
- The **second transaction** attempts 2 inserts:
 - The first insert (Kiran) is valid.
 - The second insert (Ashish) **fails due to duplicate payment_id = 1** and **negative amount** (which violates **CHECK** constraint).

Part C: Simulate Partial Failure and Ensure Consistent State

Description:

Demonstrate how inserting one valid and one invalid record within a transaction causes the entire operation to be rolled back, keeping the table in a consistent state.

Input Format:

- Table **FeePayments** as before.
-

Output Format:

payment_id	student_name	amount	payment_date
1	Ashish	5000.00	2024-06-01
2	Smaran	4500.00	2024-06-02
3	Vaibhav	5500.00	2024-06-03

Constraints:

- Transactions must fail completely if any operation fails.
-

Sample Input:

Invalid record has NULL in `student_name`.

Sample Output:

No new records inserted.

Query:

DROP TABLE IF EXISTS FeePayments;

**CREATE TABLE FeePayments (
 payment_id INT PRIMARY KEY,
 student_name VARCHAR(100) NOT NULL,
 amount DECIMAL(10,2) CHECK (amount > 0),
 payment_date DATE
) ENGINE=InnoDB;**

START TRANSACTION;

**INSERT INTO FeePayments (payment_id, student_name, amount, payment_date)
VALUES**

**(1, 'Ashish', 5000.00, '2024-06-01'),
(2, 'Smaran', 4500.00, '2024-06-02'),
(3, 'Vaibhav', 5500.00, '2024-06-03');**

COMMIT;

SELECT * FROM FeePayments;

START TRANSACTION;

INSERT INTO FeePayments (payment_id, student_name, amount, payment_date)

VALUES

(4, 'Kiran', 4000.00, '2024-06-04'),

(5, NULL, 5000.00, '2024-06-05');

ROLLBACK;

SELECT * FROM FeePayments;

OUTPUT:

The screenshot shows a MySQL query editor interface. The left pane contains a SQL script with the following queries:

```
1 DROP TABLE IF EXISTS FeePayments;
2
3 CREATE TABLE FeePayments (
4     payment_id INT PRIMARY KEY,
5     student_name VARCHAR(100) NOT NULL,
6     amount DECIMAL(10,2) CHECK (amount > 0),
7     payment_date DATE
8 ) ENGINE=InnoDB;
9
10 START TRANSACTION;
11
12 INSERT INTO FeePayments (payment_id, student_name, amount, payment_date)
13 VALUES
14 (1, 'Ashish', 5000.00, '2024-06-01'),
15 (2, 'Smaran', 4500.00, '2024-06-02'),
16 (3, 'Vaibhav', 5500.00, '2024-06-03');
17
18 COMMIT;
19
20 SELECT * FROM FeePayments;
21
22 START TRANSACTION;
23
24 INSERT INTO FeePayments (payment_id, student_name, amount, payment_date)
25 VALUES
26 (4, 'Kiran', 4000.00, '2024-06-04'),
27 (5, NULL, 5000.00, '2024-06-05');
28
```

The right pane shows the output of the queries. The first query (lines 12-16) is successful and displays the following table:

payment_id	student_name	amount	payment_date
1	Ashish	5000.00	2024-06-01
2	Smaran	4500.00	2024-06-02
3	Vaibhav	5500.00	2024-06-03

The second query (lines 24-27) fails with the following error message:

ERROR 1048 (23000) at line 24: Column 'student_name' cannot be null

Explanation:

Even though the first insert was valid, the **second insert fails**, causing the **entire transaction to rollback**, proving **Atomicity** and **Consistency**.

Part D: Verify ACID Compliance with Transaction Flow

Description:

Combine all transaction techniques into one example and verify that all ACID properties — **Atomicity**, **Consistency**, **Isolation**, and **Durability** — are preserved.

Input Format:

- Table **FeePayments**

Output Format:

Final state of the table reflecting successful committed transactions only.

Constraints:

- All four ACID properties should be demonstrated.
- Isolation can be simulated using sessions if DBMS supports.

Sample Input:

Valid inserts and a failed one using the same `payment_id`.

Sample Output:

payment_id	student_name	amount	payment_date
1	Ashish	5000.00	2024-06-01T00:00:00.000Z
2	Smaran	4500.00	2024-06-02T00:00:00.000Z
3	Vaibhav	5500.00	2024-06-03T00:00:00.000Z
7	Sneha	4700.00	2024-06-08T00:00:00.000Z
8	Arjun	4900.00	2024-06-09T00:00:00.000Z

QUERY:

```
DROP TABLE IF EXISTS FeePayments;
```

```
CREATE TABLE FeePayments (  
    payment_id INT PRIMARY KEY,  
    student_name VARCHAR(100) NOT NULL,  
    amount DECIMAL(10,2) CHECK (amount > 0),  
    payment_date DATETIME  
) ENGINE=InnoDB;
```

```
START TRANSACTION;
```

```
INSERT INTO FeePayments (payment_id, student_name, amount, payment_date)
```

```
VALUES
```

```
(1, 'Ashish', 5000.00, '2024-06-01 00:00:00'),
```

```
(2, 'Smaran', 4500.00, '2024-06-02 00:00:00'),
```

```
(3, 'Vaibhav', 5500.00, '2024-06-03 00:00:00');
```

```
COMMIT;
```

```
DELIMITER $$
```

```
CREATE PROCEDURE DuplicateInsert()
```

```
BEGIN
```

```
    DECLARE EXIT HANDLER FOR SQLEXCEPTION
```

```
    BEGIN
```

```
        ROLLBACK;
```

```
    END;
```

```
START TRANSACTION;

INSERT INTO FeePayments (payment_id, student_name, amount, payment_date)

VALUES

(4, 'Kiran', 4000.00, '2024-06-04'),

(1, 'Ashish', 5000.00, '2024-06-05');

COMMIT;

END$$

DELIMITER ;

CALL DuplicateInsert();

DELIMITER $$

CREATE PROCEDURE NullInsert()

BEGIN

    DECLARE EXIT HANDLER FOR SQLEXCEPTION

    BEGIN

        ROLLBACK;

    END;

    START TRANSACTION;

    INSERT INTO FeePayments (payment_id, student_name, amount, payment_date)

    VALUES

    (5, 'Rohan', 6000.00, '2024-06-06'),

    (6, NULL, 4500.00, '2024-06-07');

    COMMIT;

END$$

DELIMITER ;
```

CALL NullInsert();

START TRANSACTION;

INSERT INTO FeePayments (payment_id, student_name, amount, payment_date)

VALUES

(7, 'Sneha', 4700.00, '2024-06-08 00:00:00'),

(8, 'Arjun', 4900.00, '2024-06-09 00:00:00');

COMMIT;

SELECT * FROM FeePayments;

OUTPUT:

The screenshot shows a MySQL query editor interface. The left pane contains a SQL script with the following content:

```
1 DROP TABLE IF EXISTS FeePayments;
2
3 CREATE TABLE FeePayments (
4     payment_id INT PRIMARY KEY,
5     student_name VARCHAR(100) NOT NULL,
6     amount DECIMAL(10,2) CHECK (amount > 0),
7     payment_date DATETIME
8 ) ENGINE=InnoDB;
9
10
11 START TRANSACTION;
12 INSERT INTO FeePayments (payment_id, student_name, amount, payment_date)
13 VALUES
14 (1, 'Ashish', 5000.00, '2024-06-01 00:00:00'),
15 (2, 'Smaran', 4500.00, '2024-06-02 00:00:00'),
16 (3, 'Vaibhav', 5500.00, '2024-06-03 00:00:00');
17 COMMIT;
18
19
20 DELIMITER $$
21 CREATE PROCEDURE DuplicateInsert()
22 BEGIN
23     DECLARE EXIT HANDLER FOR SQLEXCEPTION
24     BEGIN
25         ROLLBACK;
26     END;
27
28     START TRANSACTION;
```

The right pane shows the output of the query, displaying a table with 4 columns: payment_id, student_name, amount, and payment_date. The output is as follows:

payment_id	student_name	amount	payment_date
1	Ashish	5000.00	2024-06-01 00:00:00
2	Smaran	4500.00	2024-06-02 00:00:00
3	Vaibhav	5500.00	2024-06-03 00:00:00
7	Sneha	4700.00	2024-06-08 00:00:00
8	Arjun	4900.00	2024-06-09 00:00:00

LEARNING OUTCOME:

- **Atomicity:** Learned how transactions either fully commit or fully rollback when an error occurs.
- **Consistency:** Observed that database constraints (PRIMARY KEY, NOT NULL, CHECK) maintain valid data.
- **Isolation:** Transactions executed sequentially demonstrate how uncommitted changes do not affect others.
- **Durability:** Committed transactions remain in the database permanently even after failures elsewhere.