Exercise 4 : Data Validation and Query Optimization

Exercise 4.1: Data Validation Queries

Aim

To implement data validation queries for ensuring correctness, consistency, and accuracy of database records.

Procedure

1. Create a sample table with constraints (for Age, Salary, Email, etc.).
2. Insert sample data including both valid and invalid values.
3. Write queries to detect invalid/duplicate data.
4. Display outputs to verify results.
5. Conclude with validations ensuring reliable data storage.

```
Input (Table Creation & Sample Data)
CREATE TABLE Employee_Validation (
    EmpID INT PRIMARY KEY,
    EmpName VARCHAR(50) NOT NULL,
    Email VARCHAR(100) UNIQUE,
    Age INT CHECK (Age >= 18),
    Salary DECIMAL(10,2) CHECK (Salary >= 3000)
);

INSERT INTO Employee_Validation (EmpID, EmpName, Email, Age,
Salary) VALUES
(1, 'Arun Kumar', 'arun.kumar@email.com', 28, 35000),
(2, 'Priya', 'priya@email.com', 22, 2800), -- Invalid Salary
(3, 'Kirankumar', NULL, 19, 40000),        -- Invalid Email
(4, 'Meena', 'meena.roy@email.com', 17, 25000), -- Invalid Age
(5, 'vadivel', 'arun.kumar@email.com', 30, 45000); -- Duplicate
Email
```

Queries & Expected Output

Query 1: Find Employees with Invalid Age (< 18)

```
SELECT EmpID, EmpName, Age FROM Employee_Validation WHERE Age < 18;
```

Expected Output

EmpID	EmpName	Age
4	Meena	17

Query 2: Find Employees with Invalid Salary (< 3000)

```
SELECT EmpID, EmpName, Salary FROM Employee_Validation WHERE
Salary < 3000;
```

Expected Output

EmpID	EmpName	Salary
2	Priya	2800

Query 3: Find Employees with Missing Email

```
SELECT EmpID, EmpName FROM Employee_Validation WHERE Email IS
NULL;
```

Expected Output

EmpID	EmpName
3	Kirankumar

Query 4: Find Duplicate Emails

```
SELECT Email, COUNT(*) AS Occurrences
FROM Employee_Validation
GROUP BY Email
HAVING COUNT(*) > 1;
```

Expected Output

Email	Occurrences
arun.kumar@email.co	2

m	

Result

1. Data validation queries successfully identified **invalid ages, salaries, missing emails, and duplicate email entries**.
2. Such validation ensures **data accuracy, consistency, and reliability** in the database.

Exercise 4.2: Performance Optimization (with EXPLAIN)

Aim

To optimize SQL queries using indexing and to analyze query performance using EXPLAIN.

Procedure

1. Create the Orders table and insert sample data.
2. Execute a query without any index and check the performance using EXPLAIN.
3. Create an index on the Product column.
4. Execute the same query again and compare performance using EXPLAIN.
5. Observe differences in type, key usage, rows scanned, and Extra column to understand optimization benefits.

Sample Table and Data

```
CREATE TABLE Orders (
    OrderID INT PRIMARY KEY,
    CustomerName VARCHAR(50),
    Product VARCHAR(50),
    Quantity INT,
    Price DECIMAL(10,2),
    OrderDate DATE
);
```

```
INSERT INTO Orders VALUES
(1, 'Arun Kumar', 'Laptop', 1, 60000, '2024-01-12'),
(2, 'Priya', 'Mobile', 2, 30000, '2024-01-18'),
(3, 'Meena', 'Tablet', 1, 20000, '2024-02-01'),
(4, 'Kiran', 'Laptop', 3, 180000, '2024-02-10'),
(5, 'Suresh', 'Headphones', 5, 15000, '2024-02-15'),
```

```
(6, 'Deepa', 'Laptop', 2, 120000, '2024-02-20'),
(7, 'Anand', 'Mobile', 4, 60000, '2024-03-01'),
(8, 'Ravi', 'Tablet', 2, 40000, '2024-03-05');
```

Query 1: Without Index

```
EXPLAIN SELECT * FROM Orders WHERE Product = 'Laptop';
```

Expected EXPLAIN Output (Before Index)

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	Orders	ALL	NULL	NULL	NULL	NULL	8	Using where

```
type = ALL → full table scan
key = NULL → no index used
rows = 8 → all rows scanned
```

Query 2: Create Index

```
CREATE INDEX idx_product ON Orders(Product);
```

Query 3: After Index

```
EXPLAIN SELECT * FROM Orders WHERE Product = 'Laptop';
```

Expected EXPLAIN Output (After Index)

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	Orders	ref	idx_product	idx_product	102	const	3	Using where

type = ref → uses index to filter

key = idx_product → index applied

rows = 3 → only matching rows scanned

Query 4: Optimized Aggregation

```
SELECT Product, SUM(Quantity*Price) AS TotalSales
FROM Orders
GROUP BY Product;
```

Expected Output

Product	TotalSales
Laptop	360000.00
Mobile	90000.00

Tablet	60000.00
Headphones	15000.00

Index helps filtering, aggregation still scans all rows but query performance improves on large datasets.

Query 5: Limit Result Set

```
SELECT OrderID, CustomerName, Price
FROM Orders
ORDER BY Price DESC
LIMIT 3;
```

Expected Output

OrderID	CustomerName	Price
4	Kiran	180000.00
6	Deepa	120000.00
1	Arun Kumar	60000.00

LIMIT reduces output and memory usage, improving performance for large tables.

Result

- Using indexes significantly reduces rows scanned and query execution time.
- EXPLAIN shows full table scan (ALL) before indexing vs ref/index scan after indexing.
- Optimization techniques like indexing, limiting results, and query rewriting improve efficiency in large datasets.
- Practical understanding of EXPLAIN allows students to identify slow queries and optimize them.

Exercise 5: Transaction Management

Aim:

To understand the concept of Transaction Management in SQL using operations such as COMMIT, ROLLBACK, and SAVEPOINT.

Procedure:

1. Create a table Bank_Account.
2. Insert sample records.
3. Perform different transactions (UPDATE, DELETE, INSERT).
4. Use **COMMIT** to permanently save changes.
5. Use **ROLLBACK** to undo changes.
6. Use **SAVEPOINT** to roll back to a specific point in a transaction.

Sample Table

```
CREATE TABLE Bank_Account (
    Acc_No INT PRIMARY KEY,
    Holder_Name VARCHAR(50),
    Balance DECIMAL(10,2)
);


INSERT INTO Bank_Account VALUES
(101, 'Arun', 15000.00),
(102, 'Bala', 20000.00),
(103, 'Charan', 18000.00),
(104, 'Divya', 25000.00),
(105, 'Esha', 30000.00);
```

Queries & Expected Output

Query 1: Start Transaction and Update Balance

```
START TRANSACTION;

UPDATE Bank_Account SET Balance = Balance - 5000 WHERE
Acc_No = 101;

SELECT * FROM Bank_Account;
```

Expected Output:

Acc_No	Holder_Name	Balance
101	Arun	10000.00
102	Bala	20000.00
103	Charan	18000.00
104	Divya	25000.00
105	Esha	30000.00

Query 2: Rollback the Transaction

```
ROLLBACK;
SELECT * FROM Bank_Account;
```

Expected Output (Original Data Restored):

Acc_No	Holder_Name	Balance
101	Arun	15000.00
102	Bala	20000.00
103	Charan	18000.00
104	Divya	25000.00
105	Esha	30000.00

Query 3: Savepoint Example

```
START TRANSACTION;

UPDATE Bank_Account SET Balance = Balance - 2000 WHERE
Acc_No = 102;

SAVEPOINT sp1;

UPDATE Bank_Account SET Balance = Balance - 3000 WHERE
```

```
Acc_No = 103;

SAVEPOINT sp2;

ROLLBACK TO sp1;

SELECT * FROM Bank_Account;
```

Expected Output (Rolled back to sp1, only Query 1 is applied):

Acc_No	Holder_Name	Balance
101	Arun	15000.00
102	Bala	18000.00
103	Charan	18000.00
104	Divya	25000.00
105	Esha	30000.00

Query 4: Commit Transaction

```
COMMIT;
```

Now changes are **permanently saved**.

Result:

Thus, the concept of **Transaction Management** was successfully implemented using **COMMIT, ROLLBACK, and SAVEPOINT** on the Bank_Account table.