Exercise 3.1 – Advanced Queries (LIKE, Ordering, Filtering)

Aim

To execute advanced SQL queries using LIKE, ORDER BY, and filtering operations.

Procedure

1. Create a sample table `Students`.
2. Insert sample records.
3. Run queries with LIKE, ordering, and filtering.
4. View expected results.

Sample Input Table – `Students`

ROLLNO	NAME	DEPARTMENT	MARKS	CITY
101	Arjun	CSE	85	Chennai
102	Bhavya	IT	74	Madurai
103	Karthik	CSE	92	Coimbatore
104	Priya	ECE	68	Chennai
105	Divya	CSE	80	Trichy

Queries & Expected Output

Query 1 – LIKE

```
SELECT * FROM Students WHERE Name LIKE 'D%';
```

Expected Output

ROLLNO	NAME	DEPARTMENT	MARKS	CITY
105	Divya	CSE	80	Trichy

Query 2 – ORDER BY

```
SELECT Name, Marks FROM Students ORDER BY Marks DESC;
```

Expected Output

NAME	MARKS
KARTHIK	92
ARJUN	85
DIVYA	80
BHAVYA	74
PRIYA	68

Query 3 – Filtering with condition

```
SELECT * FROM Students WHERE Marks > 75 AND Department='CSE';
```

Expected Output

ROLLNO	NAME	DEPARTMENT	MARKS	CITY
101	Arjun	CSE	85	Chennai
103	Karthik	CSE	92	Coimbatore
105	Divya	CSE	80	Trichy

Result

Thus, advanced SQL queries using LIKE, ordering, and filtering were executed successfully.

Exercise 3.2 – Complex Data Manipulation

Aim

To perform advanced data manipulation operations such as adding new columns, updating data, and creating views.

Procedure

1. Create a `Employees` table.
2. Insert sample records.
3. Perform ALTER, UPDATE, and VIEW operations.

Sample Input Table – `Employees`

EMPID	NAME	DEPT	SALARY
201	Meena	HR	30000
202	Ravi	IT	45000
203	Suresh	IT	40000
204	Kavitha	Finance	35000
205	Arul	HR	28000

Queries & Expected Output

Query 1 – Add new column

```
ALTER TABLE Employees ADD City VARCHAR(20);
```

(No immediate output – column added successfully)

Query 2 – Update values

```
UPDATE Employees SET City='Chennai' WHERE Dept='HR';
```

Expected Output (After Update)

EMPID	NAME	DEPT	SALARY	CITY
201	Meena	HR	30000	Chennai
205	Arul	HR	28000	Chennai

Query 3 – Create View

```
CREATE VIEW IT_Employees AS SELECT Name, Salary FROM Employees WHERE
Dept='IT';
SELECT * FROM IT_Employees;
```

Expected Output

NAME	SALARY
RAVI	45000
SURESH	40000

Result

Thus, complex data manipulation operations were successfully performed.

Exercise 3.3 – Transaction Management

Aim

To implement transaction control in SQL using COMMIT, ROLLBACK, and SAVEPOINT.

Procedure

1. Create an `Accounts` table.
2. Insert records.
3. Perform transaction operations.

Sample Input Table – `Accounts`

AccNo	Name	Balance
301	Ramesh	10000
302	Sangeetha	15000
303	Mohan	20000

Queries & Expected Output

Query 1 – Withdraw amount with SAVEPOINT

```
SAVEPOINT StartPoint;
UPDATE Accounts SET Balance=Balance-2000 WHERE AccNo=301;
SELECT * FROM Accounts;
```

Expected Output

AccNo	Name	Balance
301	Ramesh	8000
302	Sangeetha	15000
303	Mohan	20000

Query 2 – ROLLBACK

```
ROLLBACK TO StartPoint;
SELECT * FROM Accounts;
```

Expected Output (Restored)

AccNo	Name	Balance
301	Ramesh	10000
302	Sangeetha	15000
303	Mohan	20000

Query 3 – COMMIT

```
UPDATE Accounts SET Balance=Balance+1000 WHERE AccNo=303;
COMMIT;
SELECT * FROM Accounts;
```

Expected Output

AccNo	Name	Balance
301	Ramesh	10000
302	Sangeetha	15000
303	Mohan	21000

Result
Thus, transaction management was implemented using COMMIT, ROLLBACK, and SAVEPOINT.

Exercise 3.4 – Data Validation Queries

Aim
To validate data using SQL constraints and validation queries.

Procedure
1. Create `Registration` table.
2. Apply constraints.
3. Insert records and check validation.

Sample Input Table – `Registration`

REGNO	NAME	AGE	EMAIL
401	Anitha	20	anitha@gmail.com
402	Bala	17	bala123@gmail.com
403	Sneha	21	sneha@yahoo.com

Queries & Expected Output

Query 1 – Check age validation (Age >=18)
```
SELECT * FROM Registration WHERE Age<18;
```
Expected Output

REGNO	NAME	AGE	EMAIL
402	Bala	17	bala123@gmail.com

Query 2 – Check for Gmail users
```
SELECT * FROM Registration WHERE Email LIKE '%gmail.com';
```
Expected Output

RegNo	Name	Age	Email
401	Anitha	20	anitha@gmail.com
402	Bala	17	bala123@gmail.com

Query 3 – Unique Email Validation

```
SELECT Email, COUNT(*) FROM Registration GROUP BY Email HAVING COUNT(*)>1;
```

Expected Output

(If duplicates existed, they would be listed. In this case – No rows returned)

Result

Thus, data validation queries were successfully executed.

Exercise 3.5 – Performance Optimization

Aim

To optimize SQL queries using indexes and query optimization techniques.

Procedure

1. Create a `Sales` table.
2. Insert records.
3. Apply indexing and optimized queries.

Sample Input Table – `Sales`

SALEID	PRODUCT	QUANTITY	PRICE
501	Laptop	2	45000
502	Mouse	10	500
503	Keyboard	5	1000
504	Laptop	1	45000
505	Monitor	3	8000

Queries & Expected Output

Query 1 – Create Index

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

(No direct output – index created successfully)

Query 2 – Optimized Search

```
SELECT * FROM Sales WHERE Product='Laptop';
```

Expected Output

SALEID	PRODUCT	QUANTITY	PRICE
501	Laptop	2	45000
504	Laptop	1	45000

Query 3 – Aggregation Optimization

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

Expected Output

PRODUCT	TOTALSALES
LAPTOP	135000
MOUSE	5000
KEYBOARD	5000
MONITOR	24000

Result

Thus, performance optimization was successfully achieved using indexing and optimized queries.

Exercise 3.1 – Advanced Queries (LIKE, Ordering, Filtering)

Aim

To execute advanced SQL queries using LIKE, ORDER BY, and filtering operations.

Procedure

1. Create a sample table `Students`.
2. Insert sample records.
3. Run queries with LIKE, ordering, and filtering.
4. View expected results.

Sample Input Table – `Students`

RollNo	Name	Department	Marks	City
101	Arjun	CSE	85	Chennai
102	Bhavya	IT	74	Madurai
103	Karthik	CSE	92	Coimbatore
104	Priya	ECE	68	Chennai
105	Divya	CSE	80	Trichy

Queries & Expected Output

Query 1 – LIKE

```
SELECT * FROM Students WHERE Name LIKE 'D%';
```

Expected Output

RollNo	Name	Department	Marks	City
105	Divya	CSE	80	Trichy

Query 2 – ORDER BY

```
SELECT Name, Marks FROM Students ORDER BY Marks DESC;
```

Expected Output

Name	Marks
Karthik	92
Arjun	85
Divya	80
Bhavya	74
Priya	68

Query 3 – Filtering with condition

```
SELECT * FROM Students WHERE Marks > 75 AND Department='CSE';
```

Expected Output

RollNo	Name	Department	Marks	City
101	Arjun	CSE	85	Chennai
103	Karthik	CSE	92	Coimbatore
105	Divya	CSE	80	Trichy

Result

Thus, advanced SQL queries using LIKE, ordering, and filtering were executed successfully.

Exercise 3.2 – Complex Data Manipulation

Aim

To perform advanced data manipulation operations such as adding new columns, updating data, and creating views.

Procedure

1. Create a `Employees` table.
2. Insert sample records.
3. Perform ALTER, UPDATE, and VIEW operations.

Sample Input Table – `Employees`

EmpID	Name	Dept	Salary
201	Meena	HR	30000
202	Ravi	IT	45000
203	Suresh	IT	40000
204	Kavitha	Finance	35000
205	Arul	HR	28000

Queries & Expected Output

Query 1 – Add new column

```
ALTER TABLE Employees ADD City VARCHAR(20);
```

(No immediate output – column added successfully)

Query 2 – Update values

```
UPDATE Employees SET City='Chennai' WHERE Dept='HR';
```

Expected Output (After Update)

EmpID	Name	Dept	Salary	City
201	Meena	HR	30000	Chennai
205	Arul	HR	28000	Chennai

Query 3 – Create View

```
CREATE VIEW IT_Employees AS SELECT Name, Salary FROM Employees WHERE
Dept='IT';
SELECT * FROM IT_Employees;
```

Expected Output

Name	Salary
Ravi	45000
Suresh	40000

Result
Thus, complex data manipulation operations were successfully performed.

Exercise 3.3 – Transaction Management

Aim
To implement transaction control in SQL using COMMIT, ROLLBACK, and SAVEPOINT.

Procedure

1. Create an `Accounts` table.
2. Insert records.
3. Perform transaction operations.

Sample Input Table – `Accounts`

AccNo	Name	Balance
301	Ramesh	10000
302	Sangeetha	15000
303	Mohan	20000

Queries & Expected Output

Query 1 – Withdraw amount with SAVEPOINT

```
SAVEPOINT StartPoint;
UPDATE Accounts SET Balance=Balance-2000 WHERE AccNo=301;
SELECT * FROM Accounts;
```

Expected Output

AccNo	Name	Balance
301	Ramesh	8000
302	Sangeetha	15000
303	Mohan	20000

Query 2 – ROLLBACK

```
ROLLBACK TO StartPoint;
SELECT * FROM Accounts;
```

Expected Output (Restored)

AccNo	Name	Balance
301	Ramesh	10000
302	Sangeetha	15000
303	Mohan	20000

Query 3 – COMMIT

```
UPDATE Accounts SET Balance=Balance+1000 WHERE AccNo=303;
```

```
COMMIT;
SELECT * FROM Accounts;
```

Expected Output

AccNo	Name	Balance
301	Ramesh	10000
302	Sangeetha	15000
303	Mohan	21000

Result

Thus, transaction management was implemented using COMMIT, ROLLBACK, and SAVEPOINT.

Exercise 3.4 – Data Validation Queries

Aim

To validate data using SQL constraints and validation queries.

Procedure

1. Create `Registration` table.
2. Apply constraints.
3. Insert records and check validation.

Sample Input Table – `Registration`

RegNo	Name	Age	Email
401	Anitha	20	anitha@gmail.com
402	Bala	17	bala123@gmail.com
403	Sneha	21	sneha@yahoo.com

Queries & Expected Output

Query 1 – Check age validation (Age >=18)

```
SELECT * FROM Registration WHERE Age<18;
```

Expected Output

RegNo	Name	Age	Email
402	Bala	17	bala123@gmail.com

Query 2 – Check for Gmail users

```
SELECT * FROM Registration WHERE Email LIKE '%gmail.com';
```

Expected Output

RegN	Name	Ag	Email

o		e	
401	Anitha	20	anitha@gmail.com
402	Bala	17	bala123@gmail.com

Query 3 – Unique Email Validation

```
SELECT Email, COUNT(*) FROM Registration GROUP BY Email HAVING COUNT(*)>1;
```

Expected Output

(If duplicates existed, they would be listed. In this case – No rows returned)

Result

Thus, data validation queries were successfully executed.

Exercise 3.5 – Performance Optimization

Aim

To optimize SQL queries using indexes and query optimization techniques.

Procedure

1. Create a `Sales` table.
2. Insert records.
3. Apply indexing and optimized queries.

Sample Input Table – `Sales`

SaleID	Product	Quantity	Price
501	Laptop	2	45000
502	Mouse	10	500
503	Keyboard	5	1000
504	Laptop	1	45000
505	Monitor	3	8000

Queries & Expected Output

Query 1 – Create Index

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

(No direct output – index created successfully)

Query 2 – Optimized Search

```
SELECT * FROM Sales WHERE Product='Laptop';
```

Expected Output

SaleID	Product	Quantity	Price
501	Laptop	2	45000

504 Laptop 1 45000

Query 3 – Aggregation Optimization

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

Expected Output

Product	TotalSales
Laptop	135000
Mouse	5000
Keyboard	5000
Monitor	24000

Result

Thus, performance optimization was successfully achieved using indexing and optimized queries.

Exercise 3: Advanced Queries (LIKE, Ordering, Filtering)

Aim

To execute advanced queries using pattern matching, ordering, and filtering conditions in SQL.

Procedure

1. Open MySQL and select the database `UniversityDB`.
2. Use `LIKE` for pattern-based filtering.
3. Use `YEAR()` to filter students by year of birth.
4. Use `ORDER BY` to sort data by single or multiple columns.
5. Verify the output after each query.

Input / Output

```
-- Students whose first name starts with 'K'
SELECT * FROM Students WHERE FirstName LIKE 'K%';

-- Students born in 2000
SELECT * FROM Students WHERE YEAR(DOB) = 2000;

-- Order students by last name
SELECT * FROM Students ORDER BY LastName;

-- Order by multiple criteria
SELECT * FROM Students ORDER BY DepartmentID, LastName, FirstName;
```

Sample Output:

StudentID	FirstName	LastName	DOB	Gender	DepartmentID
5	Kavitha	R	1978-12-05	Female	2
1	Kohul	T	2000-05-14	Male	2

Result

Successfully executed advanced queries using `LIKE`, filtering by `YEAR()`, and `ORDER BY` for sorting data.

Exercise 4: Complex Data Manipulation (Add Columns, Update, Create Views)

Aim

To perform complex data manipulations by altering table structures, updating values, and creating views.

Procedure

1. Add a new column `EnrollmentDate` in the Students table using `ALTER TABLE`.
2. Update the new column with enrollment dates.
3. Create a view combining Students and Departments.
4. Query the view and filter data based on enrollment year.

Input / Output

```
-- Add new column
ALTER TABLE Students ADD COLUMN EnrollmentDate DATE;

-- Update column values
UPDATE Students SET EnrollmentDate = '2023-09-01' WHERE StudentID = 1;
UPDATE Students SET EnrollmentDate = '2022-08-15' WHERE StudentID = 2;
UPDATE Students SET EnrollmentDate = '2023-09-01' WHERE StudentID = 3;

-- Create a view
CREATE VIEW StudentDetails AS
SELECT s.StudentID, s.FirstName, s.LastName, d.DepartmentName,
s.EnrollmentDate
FROM Students s
INNER JOIN Departments d ON s.DepartmentID = d.DepartmentID;

-- Query the view
SELECT * FROM StudentDetails;
```

Sample Output (View):

StudentID	FirstName	LastName	DepartmentName	EnrollmentDate
1	Kohul	T	Mathematics	2023-09-01
3	Sree	T	Physics	2023-09-01

Result

Successfully altered the table, updated data, created a view, and retrieved data from the view.

Exercise 5: Transaction Management

Aim

To perform multiple operations using transaction management commands in SQL.

Procedure

1. Start a transaction with `START TRANSACTION`.
2. Perform update and insert queries.
3. Use `COMMIT` to save changes.
4. If an error occurs, use `ROLLBACK` to undo.

Input / Output

```
START TRANSACTION;

-- Update student record
```

```
UPDATE Students SET ContactNumber = '1112223333' WHERE StudentID = 1;

-- Insert new student
INSERT INTO Students (FirstName, LastName, DOB, Gender, Email, ContactNumber,
DepartmentID, EnrollmentDate)
VALUES ('New', 'Student', '2001-03-15', 'Male', 'new.student@example.com',
'4445556666', 2, '2023-09-01');

COMMIT;
```

Output:

```
Query OK, 2 rows affected
```

Result

Successfully implemented transaction management with update and insert operations followed by `COMMIT`.

Exercise 3.4: Data Validation Queries

Aim

To validate data by finding duplicate records and invalid references using SQL queries.

Procedure

1. Group by `Email` and use `HAVING COUNT()>1` to find duplicates.
2. Use `LEFT JOIN` to detect invalid department references.

Input / Output

```
-- Find duplicate emails
SELECT Email, COUNT(*)
FROM Students
GROUP BY Email
HAVING COUNT(*) > 1;

-- Find students with invalid department references
SELECT s.*
FROM Students s
LEFT JOIN Departments d ON s.DepartmentID = d.DepartmentID
WHERE d.DepartmentID IS NULL;
```

Sample Output (Duplicates):

Email	Count
new.student@example.com	2

Result

Successfully identified duplicate entries and students with invalid department references.

Exercise 3.5: Performance Optimization

Aim

To improve query performance by creating indexes on frequently used columns.

Procedure

1. Use `CREATE INDEX` to index columns used frequently in queries.
2. Check that indexes are created successfully.
3. Execute queries again to observe improved performance.

Input / Output

```
CREATE INDEX idx_students_department ON Students(DepartmentID);
CREATE INDEX idx_students_email ON Students(Email);
CREATE INDEX idx_students_dob ON Students(DOB);
```

Output:

```
Query OK, 0 rows affected
```

Result

Successfully created indexes on Students table columns for optimized query performance.