**PL/SQL Teaching Resource: Comprehensive Guide**

This resource is designed to provide an in-depth understanding of PL/SQL concepts essential for working with Oracle databases, particularly focusing on Views, Transactions, Indexing, and User Management.

**Module Overview**

**Key Concepts**

* **Views**
  + Creating, updating, and querying views
* **Indexes**
  + Understanding and simulating clustered vs. non-clustered indexes
* **Transactions**
  + Managing transaction boundaries using COMMIT, ROLLBACK, and SAVEPOINT
* **User Management**
  + Implementing role-based access control using GRANT, REVOKE, and ROLES

**Day-Wise Learning Plan**

* **Day 1**: Views
* **Day 2**: Indexes
* **Day 3**: Transactions
* **Day 4**: Role-Based Access Control

**Day 1: Views**

**Objective**

* Understand the use of Views for abstraction, reuse, and security.

**Example Tables**

**1. Products**

CREATE TABLE Products (

product\_id NUMBER,

name VARCHAR2(50),

price NUMBER(10, 2),

stock NUMBER

;

**2. Orders**

CREATE TABLE Orders (

order\_id NUMBER,

product\_id NUMBER,

quantity NUMBER,

order\_date DATE

;

**Trainer Script**

1. **Create View**

CREATE VIEW vw\_product\_summary AS

SELECT name, price FROM Products;

1. **Update via View**

UPDATE vw\_product\_summary

SET price = price + 10

WHERE name = 'Apple';

1. **Inline View**

SELECT \* FROM (SELECT \* FROM Orders WHERE ROWNUM <= 5);

1. **Query View**

1SELECT \* FROM vw\_product\_summary;

**Use Case Scenario**

* **Purpose**: An inventory dashboard shows only the name and price of products.
* **Action**: Updating prices for promotional events is done through the view while maintaining a clear separation from the underlying data.

**Day 2: Indexes**

**Objective**

* Understand indexing to improve query performance.

**Trainer Script**

1. **Create Index**

CREATE INDEX idx\_product\_name ON Products(name);

1. **Drop Index**

DROP INDEX idx\_product\_name;

**Simulation of Clustered Index**

* Oracle does not explicitly support clustered indexes, but you can simulate them by creating indexes on frequently queried columns to enhance performance.

**Use Case**

* In a retail application, indexing the **name** column of the **Products** table to speed up searches performed on product names.

**Day 3: Transactions**

**Objective**

* Understand how COMMIT, ROLLBACK, and SAVEPOINT manage transaction boundaries.

**Trainer Script**

BEGIN

UPDATE Products SET stock = stock - 5 WHERE product\_id = 1;

SAVEPOINT before\_payment;

UPDATE Payments SET payment\_status = 'Failed' WHERE payment\_id = 9001;

ROLLBACK TO before\_payment; -- Reverts the payment status change

COMMIT; -- Confirms the stock update

END;

**Real-Life Scenario**

* **Context**: Suppose an online shop processes an order, if a payment fails, the stock count should remain updated while reverts any changes to the payment status.

drop table products;

use world;

CREATE TABLE Products (

product\_id int,

name VARCHAR(50),

price decimal(10, 2),

stock int

);

drop table orders;

CREATE TABLE Orders (

order\_id int,

product\_id int,

quantity int,

order\_date DATE

);

INSERT INTO Products (product\_id, name, price, stock) VALUES (1, 'Laptop', 1200.00, 10);

INSERT INTO Products (product\_id, name, price, stock) VALUES (2, 'Smartphone', 800.00, 15);

INSERT INTO Products (product\_id, name, price, stock) VALUES (3, 'Tablet', 400.00, 20);

INSERT INTO Orders (order\_id, product\_id, quantity, order\_date) VALUES (1, 1, 1, '2023-10-01');

INSERT INTO Orders (order\_id, product\_id, quantity, order\_date) VALUES (2, 2, 2, '2023-10-02');

INSERT INTO Orders (order\_id, product\_id, quantity, order\_date) VALUES (3, 3, 1, '2023-10-03');

select \* from products;

select \* from orders;

SET SQL\_SAFE\_UPDATES=0;

start transaction;

-- Step 1: Update stock of product 1

UPDATE Products SET stock = stock - 2 WHERE product\_id = 1;

-- Create first savepoint after first update

Savepoint after\_stock\_update;

-- Step 2: Update quantity for product 2 in orders

UPDATE Orders SET quantity = 20 WHERE product\_id = 2;

-- Create second savepoint after order update

SAVEPOINT after\_order\_update;

select \* from products;

select \* from orders;

-- Step 3: Perform a faulty update as an example (e.g., setting quantity to a negative number)

UPDATE Orders SET quantity = -5 WHERE product\_id = 2;

select \* from orders;

-- Rollback to the second savepoint to revert faulty update

ROLLBACK TO after\_order\_update; -- Reverts to the state after the valid order update

select \* from orders;

-- Final commit for stock change

COMMIT; -- Confirms the stock update and keeps the valid order update

**Day 4: User & Role Simulation**

**Objective**

* Implementing role-based access control.

**Trainer Script**

1. **Create Tables for User and Role Simulation**

CREATE TABLE Roles (

role\_id NUMBER PRIMARY KEY,

role\_name VARCHAR2(20)

;

CREATE TABLE Users (

user\_id NUMBER,

username VARCHAR2(50),

role\_id NUMBER REFERENCES Roles(role\_id)

;

1. **GRANT Simulation**

1-- Example to create a grant access scenario.

SELECT role\_name INTO v\_role

FROM Users u

JOIN Roles r ON u.role\_id = r.role\_id

WHERE u.user\_id = :id;

IF v\_role != 'Admin' THEN

RAISE\_APPLICATION\_ERROR(-20001, 'Access Denied');

END IF;

**Use Case**

* **Example**: A web application where different roles (Admin, Customer, Seller) have varying permissions, and actions are checked against user roles.

**Deliverables for Learners**

* **Create Tables**: Set up the example tables provided.
* **Insert Sample Data**: Populate with data for testing.
* **Practice Scripts**: Modify each script slightly to reinforce learning.
* **Output Interpretation**: Write down interpretations of results after executing scripts.
* **Execution**: Run scenarios on Oracle Live SQL.

**Query Performance Analysis Example**

**Execution Plan Overview**

1undefined

| **Id** | **Operation** | **Name** | **Rows** | **Bytes** | **Cost (%CPU)** | **Time** |
| --- | --- | --- | --- | --- | --- | --- |
| 0 | SELECT STATEMENT |  | 1 | 78 | 2 (0) | 00:00:01 |
| 1 | TABLE ACCESS BY INDEX ROWID BATCHED | ACCOUNTS | 1 | 78 | 2 (0) | 00:00:01 |
| \*2 | INDEX RANGE SCAN | IDX\_ACCOUNT\_NAME | 1 | 1 | 1 (0) | 00:00:01 |

1### \*\*Explanation\*\*

2- The execution plan shows that the query accessed the `ACCOUNTS` table using an index (IDX\_ACCOUNT\_NAME) which indicates efficient retrieval of data.

3- Understanding such execution plans is critical for optimizing queries in real-world applications.

4

5---

6

7This structure provides a clear, guided approach for both trainers and learners to understand and practice important PL/SQL concepts.

Bookmark messageCopy messageExport