**National University of Computer & Emerging Sciences, Karachi**



**FAST School of Computing**

**Fall 2024 Lab Manual - 10**

|  |  |
| --- | --- |
| **Course Code: CL-2005** | **Database Systems Lab** |
| **Instructor(s):** | **Mehak Mazhar** |

**Contents:**

1. **Introduction to Transactions (Commit, Rollback, Savepoint, Auto-commit)**
2. **Activity on Different Transactions**
3. **Additional Tasks and Solutions**

**1. Introduction to Database Transactions:**

**Definition of a Transaction:**

A **transaction** in a database is a sequence of one or more SQL statements that perform a specific unit of work, ensuring data integrity. A transaction allows multiple actions to be grouped so that they succeed or fail together, creating a consistent change in the data.

**Components of a Database Transaction:**

A database transaction can consist of:

* **DML (Data Manipulation Language) statements** that represent a consistent change in the database.
* A single **DDL (Data Definition Language)** statement, which implicitly commits.
* A single **DCL (Data Control Language)** statement, which also implies a commit.

**Example:**

Consider a fund transfer between two bank accounts, where debiting from one account and crediting to another must be successful simultaneously. If either action fails, the transaction should fail entirely, maintaining data integrity.

**2. Transaction Control Commands:**

1. **COMMIT** - Finalizes the transaction, making changes permanent in the database.
2. **ROLLBACK** - Reverts all changes made since the start of the transaction or since the last SAVEPOINT.
3. **SAVEPOINT** - Sets a marker within the transaction, allowing partial rollbacks to specific points.
4. **SET TRANSACTION** - Configures transaction properties such as read-write mode.
5. **AUTOCOMMIT** - Automatically commits each individual DML statement.

**Flow of Transaction Control:**

|  |  |  |
| --- | --- | --- |
| **T#** | **Transaction** | **Explanation** |
| t1 | SET TRANSACTION NAME 'sal\_update'; | Begins a transaction and names it sal\_update. |
| t2 | UPDATE employees SET salary = 7000 WHERE last\_name = 'Banda'; | Updates salary for Banda to 7000. |
| t3 | SAVEPOINT after\_banda\_sal; | Sets a savepoint named after\_banda\_sal. |
| t4 | UPDATE employees SET salary = 12000 WHERE last\_name = 'Greene'; | Updates salary for Greene to 12000. |
| t5 | SAVEPOINT after\_greene\_sal; | Sets a savepoint named after\_greene\_sal. |
| t6 | ROLLBACK TO SAVEPOINT after\_banda\_sal; | Rolls back to after\_banda\_sal, undoing Greene’s update. |
| t7 | UPDATE employees SET salary = 11000 WHERE last\_name = 'Greene'; | Updates Greene’s salary to 11000. |
| t8 | ROLLBACK; | Rolls back all changes, ending sal\_update. |
| t9 | SET TRANSACTION NAME 'sal\_update2'; | Begins a new transaction named sal\_update2. |
| t10 | UPDATE employees SET salary = 7050 WHERE last\_name = 'Banda'; | Updates Banda’s salary to 7050. |
| t11 | UPDATE employees SET salary = 10950 WHERE last\_name = 'Greene'; | Updates Greene’s salary to 10950. |
| t12 | COMMIT; | Commits all changes, finalizing sal\_update2. |

**3. Implicit and Explicit Transaction Control:**

**Implicit Transaction Control:**

Oracle Database performs automatic commits and rollbacks in specific situations:

* **Automatic Commit:** Occurs when a DDL or DCL statement is issued or upon a normal exit from SQL\*Plus.
* **Automatic Rollback:** Happens on an abnormal exit or system failure.

**Explicit Transaction Control:**

Users can manually control transactions using **COMMIT**, **ROLLBACK**, and **SAVEPOINT** commands to ensure precise control over data modifications.

**Advantages of COMMIT and ROLLBACK:**

* Ensures data consistency.
* Allows previewing changes before committing.
* Groups related operations for atomic execution.

**Creating and Modifying a Table in a Transaction**

1. **Create a Table** worker with attributes worker\_id, worker\_name, and salary.

|  |
| --- |
| CREATE TABLE worker (  worker\_id NUMBER PRIMARY KEY,  worker\_name VARCHAR2(50),  salary NUMBER  ); |

1. **Insert a Record** in worker:

|  |
| --- |
| INSERT INTO worker (worker\_id, worker\_name, salary) VALUES (1, mehak, 5000); |

***This action is a DML statement, so a commit is required to finalize it.***

1. **Update the Salary** without committing:

|  |
| --- |
| UPDATE worker SET salary = 6000 WHERE worker\_id = 1; |

1. **Activity on Different Transactions:**

Open a **new unshared worksheet**.

Attempt to **update** the same record from the unshared worksheet:

|  |
| --- |
| UPDATE worker SET salary = 7000 WHERE worker\_id = 1; |

***The transaction will lock the row until the previous worksheet's transaction is committed or rolled back.***

1. **Commit the transaction** in the original worksheet:

|  |
| --- |
| COMMIT; |

***Retry the update from the unshared worksheet. This time, it should succeed.***

**Savepoint and Rollback within a Transaction**

1. **Start Transaction** and Insert Data:

|  |
| --- |
| SET TRANSACTION NAME 'test\_transaction';  INSERT INTO worker (worker\_id, worker\_name, salary) VALUES (2, ‘Erum’, 5500);  SAVEPOINT sp1; |

1. **Update Data** and set a savepoint:

|  |
| --- |
| UPDATE worker SET salary = 6000 WHERE worker\_name = ‘Erum’;  SAVEPOINT sp2; |

1. **Rollback to sp1** (Undo the Update):

|  |
| --- |
| ROLLBACK TO SAVEPOINT sp1; |

***Erum’s salary returns to 5500.***

1. **Commit the Transaction**:

|  |
| --- |
| COMMIT; |

**Using AUTOCOMMIT**

1. **Enable AUTOCOMMIT**:

|  |
| --- |
| SET AUTOCOMMIT ON; |

1. **Insert Data**:

|  |
| --- |
| INSERT INTO worker (worker\_id, worker\_name, salary) VALUES (3, ‘FAST-NU’, 5000); |

***Since AUTOCOMMIT is on, the insert will commit automatically.***

1. **Disable AUTOCOMMIT**:

|  |
| --- |
| SET AUTOCOMMIT OFF; |

**Scenario: Customer and Order Transactions**

**Problem Statement:**

We want to handle a customer’s order placement, ensuring that both customer details and order records are updated only if all actions in the transaction succeed. If any action fails, all changes should be undone to maintain data integrity.

**Tables Involved:**

1. **customer**: Stores customer information.
2. **orders**: Stores order information related to each customer.

**Table Structure**:

A screenshot of a computer

Description automatically generated

**Transaction Steps**

**Step 1: Insert a New Customer**

Start by inserting a new customer into the customer table and creating a savepoint to rollback to this step if necessary.

A screenshot of a computer

Description automatically generated

**Step 2: Insert an Order and Deduct Balance**

Now, insert a new order for the customer and update their balance. If updating the balance fails, we will rollback to the savepoint.

A screenshot of a computer

Description automatically generated

**Step 3: Commit or Rollback**

If all statements execute successfully, we can commit the transaction. However, if any statement fails (such as insufficient balance), rollback to the appropriate savepoint or to the beginning of the transaction.

A screenshot of a computer

Description automatically generated

**Explanation of Flow:**

1. **Start the Transaction** by setting a name (customer\_order\_transaction).
2. **Insert Customer Data**: Insert a new customer with an initial balance and set a savepoint (customer\_added).
3. **Insert Order Data** and **Update Balance**: Deduct the order amount from the customer’s balance and create another savepoint (order\_added).
4. **Commit or Rollback**:
   * If all operations succeed and the balance is sufficient, commit the transaction.
   * If there’s an issue (like insufficient balance), rollback to customer\_added to cancel the order but retain the customer.
   * If there’s another type of error, rollback the entire transaction.

**Class Task:**

Try to create a similar example where:

* You add a new product to a product table.
* Add an inventory entry and set a savepoint.
* Deduct from inventory on order placement.
* Rollback if the inventory quantity goes negative, ensuring inventory consistency.

**Tasks:**

1. Create a new table named product\_inventory with columns for product\_id, product\_name, stock, and price. Insert three different product records with initial stock values. Without committing the transaction, reduce the stock of one of the products and create a savepoint named stock\_update.
2. In the employee table, add a new employee with a salary. Then, increase their salary by 10%, set a savepoint named salary\_increase, and then further increase it by another 5%. Rollback to the salary\_increase savepoint.
3. Use the customer and orders tables. Insert a new customer into the customer table. Then, insert an order for this customer in the orders table. Use a transaction control to ensure that both the customer and order are inserted only if both statements are successful; otherwise, roll back the changes.
4. Enable AUTOCOMMIT mode in your SQL environment. Insert a row in the sales table with sales\_id, customer\_id, and amount. After the insertion, check if the row has been committed automatically. Disable AUTOCOMMIT afterward.
5. Using the transactions table, simulate a transaction where multiple debits and credits are made on an account. Set multiple savepoints after each debit or credit operation, and then rollback to a specific savepoint to undo one of the operations.

**The End 😊**