Subject :		Software :	
		Hardware :	
Branch :	Semester :	Page No.	Prog No. 10

PROBLEM STATEMENT: Implimentation of Transaction Processing and Concurrency Control Concepts.

ALGORITHM & CODE:

Transaction Concepts:

A transaction is a logical unit of work that contains one or more SQL statements.

Transactions have four key properties (ACID):

1. Atomicity - All operations complete successfully or none do
2. Consistency - Database remains in a consistent state.
3. Isolation - Transactions don't interfere with each other
4. Durability - completed transactions persist even after failures

Basic transaction Example:

```
BEGIN:
  -- Start transaction (implicit in Oracle)
INSERT INTO accounts(account_id, balance) VALUES
                                    (1001, 5000);
INSERT INTO accounts(account_id, balance) VALUES
                                    (1002, 3000);
-- Commit makes changes permanent:
  COMMIT;
EXCEPTION
   WHEN OTHERS THEN.
```

INPUT GIVEN	
OUTPUT OBTAINED	
REMARKS	
GRADE :	Signature of Faculty Date :
	Signature of Student Date :

Subject :		Software :	
		Hardware :	
Branch :	Semester :	Page No.	Prog No.

PROBLEM STATEMENT

ALGORITHM & CODE :

```
-- Rollback undoes all changes in case of error
ROLLBACK;
   DBMS_OUTPUT.PUT_LINE ('Error.' || SQLERRM);
END;
```

Concurrency Control Concepts:

Concurrency Control manages simultaneously transaction to maintain data Consistency.

Common concurrency Problem

Lost Update : Two transactions Update same data, second overwrites first

Dirty Read : Transaction reads uncommitted data from another transaction.

Non-Repeatable Read : Same Query Returns different results within a transaction.

Phantom Read : New Rows appear in subsequent reads within a transaction.

To demostrate Concurrency control Problem in an Oracle database, we can use a Scenario where two Sessions try to Update

INPUT GIVEN

OUTPUT OBTAINED

REMARKS

GRADE :	Signature of Faculty Date :	Signature of Student Date :

Subject :		Software :	
		Hardware :	
Branch :	Semester :	Page No.	Prog No.

PROBLEM STATEMENT

ALGORITHM & CODE :

the same row simultaneously. We'll show how to use lock operations to prevent issues like lost updates or dirty reads.

Let's consider the following scenario :
We have a table accounts with columns id and balance.

Two sessions (transactions) will attempt to update the same account's balance.

Step-1 : create the table

First, we need to create a table in the Oracle database.

```
CREATE TABLE accounts (
    id NUMBER PRIMARY KEY,
    balance NUMBER
);

INSERT INTO accounts (id, balance) VALUES (1, 100);
COMMIT;
```

INPUT GIVEN

OUTPUT OBTAINED

REMARKS

GRADE :

Signature of Faculty	Signature of Student
Date :	Date :

Subject :		Software :	
		Hardware :	
Branch :	Semester :	Page No.	Prog No.

PROBLEM STATEMENT

ALGORITHM & CODE :

STEP - 2 Simulate Concurrency Control Problem:

Now, we'll simulate the problem by running two transactions in parallel. We'll use SQL*Plus or any other Oracle client to run these transactions.

Session-1:

```
-- Start transaction:
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
--Read the balance
SELECT balance FROM accounts WHERE id = 1;
--Simulate from processing time
-- (in practice, this could be some application
   logic delay)
DBMS_LOCK.SLEEP(5);
-- Update the balance
UPDATE ~~ACCOUNTS~~ accounts SET balance = balance +
                                 50 WHERE id = 1;
-- Commit the transaction
COMMIT;
```

INPUT GIVEN

OUTPUT OBTAINED

REMARKS

GRADE :	Signature of Faculty Date :	Signature of Student Date :

Subject :		Software :	
		Hardware :	
Branch :	Semester :	Page No.	Prog No.

PROBLEM STATEMENT

ALGORITHM & CODE :

Session-2

```
-- Start transaction
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;

-- Read the balance
SELECT balance FROM accounts WHERE id = 1;

-- Simulate some processing time
-- (in practice, this could be some application
logic delay)
  DBMS_LOCK.SLEEP(5);

-- Update the balance
UPDATE accounts SET balance = balance - 30
WHERE id = 1;

-- Commit the transaction
COMMIT;
```

Step-3 Adding locks to prevent concurrency issues.

To avoid concurrency issues, we can use explicit locking. Here's how you can use SELECT ... FOR UPDATE to lock the row and prevent other transactions from accessing it until

INPUT GIVEN	
OUTPUT OBTAINED	
REMARKS	
GRADE :	Signature of Faculty Date :
	Signature of Student Date :

Subject :		Software :	
		Hardware :	
Branch :	Semester :	Page No.	Prog No.

PROBLEM STATEMENT

ALGORITHM & CODE :

the current transaction completes.

Session 1 (with Locks)

```
-- Start transactions
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
-- lock the row
SELECT ~~BALANCE~~ balance FROM accounts WHERE id = 1 FOR
                                                UPDATE;
-- Simulate som processing time
DBMS_LOCK. SLEEP (5);
--UPDATE the balance
UPDATE accounts SET balance = balance + 50
                                WHERE id = 1;
-- Commit the transaction
COMMIT;
```

Session-2 (with Locks)

```
-- Start transaction
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
-- lock the row
SELECT balance FROM accounts WHERE id = 1 FOR UPDATE;
-- Simulate Some processing time
DBMS_LOCK. SLEEP (5);
-- Update the balance
UPDATE accounts SET balance = balance - 30
                                WHERE id = 1;
```

INPUT GIVEN

OUTPUT OBTAINED

REMARKS

GRADE :	Signature of Faculty Date :	Signature of Student Date :

Subject :		Software :	
		Hardware :	
Branch :	Semester :	Page No.	Prog No.

PROBLEM STATEMENT

ALGORITHM & CODE :

-- Commit the transaction.

```
COMMIT ;
```

Explanation

Without locks: When you run both sessions without locks, they can read the same initial balance and update it independently, leading to a lost update problem.

With locks: When you run both sessions with SELECT ... FOR UPDATE, the second session will wait until the first session completes its transaction. This ensures that only one session updates the balance at a time, preventing lost updates and ensuring data integrity.

Running the Code :

To observe the concurrency control problem and its resolution, you can:

1. Execute the code in the "Without locks" section in two separate sessions simultaneously.
2. Note the balance value before and after the transactions to observe the lost update problem.

INPUT GIVEN

OUTPUT OBTAINED

REMARKS

GRADE :	Signature of Faculty Date :	Signature of Student Date :

Subject :		Software :	
		Hardware :	
Branch :	Semester :	Page No.	Prog No.

PROBLEM STATEMENT

ALGORITHM & CODE :

3. Execute the code in the "With Locks" section in two separate sessions simultaneously.
4. Note that the second session waits for the first session to complete, ensuring proper balance updates.

Using explicit locks in a database transaction ensures that concurrent transactions do not interfere with each other, maintaining data consistency and integrity.

INPUT GIVEN

OUTPUT OBTAINED

REMARKS

GRADE :	Signature of Faculty Date :	Signature of Student Date :