CS1555 / CS2550 Recitation 11 Solution

Objective: To practice Evaluation Modes, Transactions, Procedures and Functions

PART 0: Review of HW6 Questions

1.d) For each year, list the most read message ID(s).

```
select r.year, r.msgID
from  (
        select msgID, to_char(time_read, 'YEAR') as year, count(*) as num_read
        from Message natural join Recipient
        where time_read is not null
        group by msgID, to_char(time_read, 'YEAR')
        ) r
        join
        (
        select year, max(num_read) as max_num_read
        from (
                select msgID, to_char(time_read, 'YEAR') as year, count(*) as num_read
                from Message natural join Recipient
                where time_read is not null
                group by msgID, to_char(time_read, 'YEAR')
                )
        group by year
        ) t
        on r.year = t.year and r.num_read = t.max_num_read;
```

2.d) Write a trigger called **CreateConversation** that will create an new conversation entry whenever a message is added with a conversation ID that does not yet exist.

```
CREATE OR REPLACE TRIGGER CreateConversation
BEFORE INSERT ON Message
FOR EACH ROW
DECLARE
  CountCov NUMBER;
BEGIN
    SELECT COUNT(*) INTO CountCov FROM Conversation WHERE convID=:new.convID;
    IF CountCov = 0 THEN
            INSERT INTO Conversation (convID, duration, user_ID)
            VALUES (:new.convID, 0, :new.sender_ID);
    END IF;
END;
/
```

-- An alternative 2. d)

```
CREATE OR REPLACE TRIGGER CreateConversation
BEFORE INSERT ON Message
FOR EACH ROW
DECLARE
    CovExists NUMBER;
BEGIN
    SELECT CASE WHEN EXISTS ( SELECT *
                        FROM Conversation
                        WHERE convID=:new.convID ) THEN 1
            ELSE 0
            END INTO CovExists
    FROM Dual;

    IF CovExists = 0 THEN
             INSERT INTO Conversation (convID, duration, user_ID)
             VALUES (:new.convID, 0, :new.sender_ID);
    END IF;
END;
/
```

PART 1: Constraint Evaluation Modes and Transactions

DEFERRED : withheld for or until a stated time (COMMIT)

a) **Not Deferrable** (*default*): *every time a database modification statement is executed, the constraints are checked.*
b) **Deferrable Initially Immediate**: *every time a database modification statement is executed, the constraints are checked IMMEDIATE. BUT, the constraints can be deferred on demand, when needed*
c) **Deferrable Initially Deferred**: the constraints are check just BEFORE each transaction commits.

1. Use the create statement with the deferred statement mentioned below

```
create table NOTDEF (
        ssn number,
        constraint pk_ssn_1 PRIMARY KEY(ssn)
);

create table DEFIMM (
        ssn number,
        constraint pk_ssn_2 PRIMARY KEY(ssn) Deferrable Initially Immediate
);
```

```
create table DEFDEF (
        ssn number,
        constraint pk_ssn_3 PRIMARY KEY(ssn) Deferrable Initially Deferred
);
```

2. For each table created above, run the SQL statements and mention if and when you encounter an error.

NotDef:
insert into NOTDEF values (1234);
insert into NOTDEF values (1234); => primary key constraint violation. The values should be unique.

DefImm:
insert into DEFIMM values (1234);
insert into DEFIMM values (1234); => primary key constraint violation. The values should be unique.

DefDef:
insert into DEFDEF values (1234);
insert into DEFDEF values (1234);
commit; => primary key constraint violation. The values should be unique.

3. Run: < set constraint *constraint_name* deferred > for the constraint set in table DefImm; Run the previous insert again. Do you see any difference?

SQL> set constraint pk_ssn_2 deferred;
SQL> insert into DEFIMM values(1234);
SQL> commit; => primary key constraint violation. The values should be unique.

NOTE: remember that we already have value 1234 in the table because of the previous insert statements.

4. For each table created above, run the SQL statements and show the table content after the inserts.

a) set constraints all deferred
b) insert value 1235
c) insert value 1235
d) commit;

NotDef:
set constraints all deferred;
insert into NOTDEF values (1235);
insert into NOTDEF values (1235);
commit; => First row was inserted successfully.

DefImm:

```
set constraints all deferred;
insert into DEFIMM values (1235);
insert into DEFIMM values (1235);
```

commit; => No row was inserted. With "set constraints all deferred" and declaration deferrable in the schema table, the statements after "set constraints all deferred" and before "commit" are treated as an transaction. Any error violating the state of the database (e.g., IC violation, deadlock, etc. Not including errors that don't violate the state e.g., table not found) will automatically roll back the whole transaction.

DefDef:

```
set constraints all deferred;
insert into DEFDEF values (1235);
insert into DEFDEF values (1235);
```

commit; => No row was inserted. Same reason as for the DefImm table.

PART 2: Procedures and Functions

Before we start:
- Copy and run the file creating the Bank Accounts database using:
  host cp ~panos/1555/recitation/bankdb.sql bankdb.sql
  @ bankdb

---

1. Create a stored procedure **transfer_fund** that, given a from_account, a to_account, and an amount, transfer the specified amount from from_account to to_account if the balance of the from_account is sufficient.

```
create or replace procedure transfer_fund(from_account in varchar2, to_account in
varchar2, amount in number)
as
from_account_balance number;
begin
select balance into from_account_balance from account where acc_no = from_account;
if from_account_balance > amount then
     update account set balance = balance - amount where acc_no = from_account;
     update account set balance = balance + amount where acc_no = to_account;
else
dbms_output.put_line ('balance is too low');
end if;
end;
/
```

2. Call the stored procedure to transfer $100 from account 124 to 123.

```
--There are 2 ways to call a procedure:
--directly outside a PL/SQL block
set transaction read write;
set constraints all deferred;
call transfer_fund(124, 123, 100);
commit;

--inside a PL/SQL block (begin ... end;), including in the body of a trigger
/stored procedure/ function
set transaction read write;
set constraints all deferred;
begin
transfer_fund (124, 123, 100);
end;
/
commit;
```

3. Create a function **compute_balance** that, given a specific ssn, calculate the total balance of the customer (the sum of total account balances less the loan amounts)

```
create or replace function compute_balance (customer_ssn in varchar2) return number
is
balance number;
total_account_balance number;
total_loan number;
begin
select nvl(sum(balance), 0) into total_account_balance from account where ssn =
customer_ssn;
select nvl(sum(amount),0) into total_loan from loan where ssn = customer_ssn;
balance := total_account_balance - total_loan;
return (balance);
end;
/
--NOTE: because a customer might have no account or no loan, resulting in
sum(balance) is null or sum(amount) is null and consequently, the final balance is
also null. To avoid this, we use the nvl function. What nvl does is checking and
replacing a value with another (0 in this case) if the value is null.
```

4. Use the function created, write a query to print the list of customers together with their total balance.

```
select ssn, compute_balance(ssn)
from customer;
```