TESTIRANJE KONKURENTNIH TRANSAKCIJA U BAZI PODATAKA

SAŽETAK

Često se u bazi podataka izvode transakcije kod kojih integritet podataka ovisi i o ostalim transakcijama koje se istovremeno izvode. Rad tih (konkurentnih) transakcija treba testirati u višekorisničkom radu, ili simulirati višekorisnički rad. Za testiranje rada npr. sto konkurentnih transakcija, vjerojatno nećemo moći koristiti sto testera (ljudi), već odgovarajuće alate za testiranje. Možemo koristiti specijalne alate za tu namjenu, ili možemo sami napraviti testove na relativno jednostavan način, kako je prikazano u ovom radu.

ABSTRACT

The database often performs transactions where data integrity depends on other transactions that run concurrently. The operation of these (concurrent) transactions should be tested in multiuser work, or simulated multiuser work. For testing a hundred concurrent transactions, instead of using a hundred testers (people), we will use appropriate test tools. We can use special tools for that purpose, or we can make tests on our own in a relatively simple way, as shown in this paper.

1. UVOD

Često se u bazi podataka izvode transakcije kod kojih integritet podataka ovisi i o ostalim transakcijama koje se istovremeno izvode. Rad tih (konkurentnih) transakcija treba testirati u višekorisničkom radu, ili simulirati višekorisnički rad. Za testiranje rada npr. sto konkurentnih transakcija, vjerojatno nećemo moći koristiti 100 testera (ljudi), već odgovarajuće alate za testiranje. Možemo koristiti specijalne alate za tu namjenu, ili možemo sami napraviti testove na relativno jednostavan način.

U radu će se prikazati testiranje (složenog) poslovnog pravila u Oracle bazi, koji smo prikazali na CASE 16 (2004. godine): "Kako spriječiti 'začarani krug' (rješavanje određenog tipa poslovnih pravila u Oracle bazi podataka)".

U 2.točki prikazuju se Oracle specifične i ANSI standardne varijante SQL hijerarhijskih upita u Oracle bazu.

U 3.točki i 4. točki ponovit ćemo (radi lakšeg razumijevanja sljedećih točaka) dio teksta iz navedenog rada sa CASE 16.

U 3. točki prikazat ćemo rješenje koje sprečava pojavu petlje (u hijerarhijskoj strukturi podataka) u jednokorisničkom radu, ali to rješenje nije dobro za višekorisnički rad (vrlo lako se nađe primjer koji to pokazuje).

U 4. točki prikazuju se dva rješenja koja bi trebala biti dobra i za višekorisnički rad. Jedno koristi tzv. autonomne transakcije, a drugo (bolje, jer javlja manje lažnih grešaka) koristi (naš) trik – simulaciju "ROLLBACK TO SAVEPOINT" u okidaču baze podataka. Iako navedena rješenja intuitivno izgledaju pouzdano (u smislu da ne dozvoljavaju grešku, tj. pojavu petlje), nije dat formalni dokaz. Prema tome, jako je važno da ta rješenja dobro testiramo.

Prikazat će se testiranje na dva načina: u 5. točki pomoću Java Executora, a u 6.točki (za one koji žele raditi samo s bazom podataka) pomoću jobova u bazi.

2. VARIJANTE SQL HIJERARHIJSKIH UPITA U ORACLE BAZI

Oracle baza je dugo godina imala specifičnu varijantu sintakse za hijerarhijske upite, pomoću CONNECT BY klauzule. Slijede dva primjera – jedan kod kojeg naredba javlja grešku ako već postoji petlja u podacima, i drugi primjer, koji ne javlja grešku:

```
-- varijanta koja puca kod petlje
SELECT empno, ename, mgr, LEVEL
  FROM emp
 START WITH mgr IS NULL
CONNECT BY PRIOR empno = mgr;


-- varijanta koja NE puca kod petlje
SELECT empno, ename, mgr, LEVEL, CONNECT_BY_ISCYCLE iscycle
  FROM emp
 START WITH mgr IS NULL
CONNECT BY NOCYCLE PRIOR empno = mgr;
```

Oracle ANSI standard za hijerarhijski upit koristi tzv. **recursive common table expressions** (recursive CTE). Oracle baza podržava recursive CTE od Oracle verzije 11.2 (2009. godine) – od tada WITH klauzula (u SELECT naredbi) može biti rekurzivna.

```
-- varijanta koja puca kod petlje
WITH each_level (empno, ename, mgr, rlevel) AS
  (SELECT empno, ename, mgr, 1 rlevel -- kao START WITH
     FROM emp
    WHERE mgr IS NULL
   UNION ALL -- kao CONNECT BY
   SELECT emp.empno, emp.ename, emp.mgr, rlevel + 1
     FROM emp, each_level
    WHERE emp.mgr = each_level.empno
  )
SELECT * FROM each_level;


-- varijanta koja NE puca kod petlje
WITH each_level (empno, ename, mgr, rlevel) AS
  (SELECT empno, ename, mgr, 1 rlevel -- kao START WITH
     FROM emp
    WHERE mgr IS NULL
   UNION ALL -- kao CONNECT BY
   SELECT emp.empno, emp.ename, emp.mgr, rlevel + 1
     FROM emp, each_level
    WHERE emp.mgr = each_level.empno
  )
 CYCLE mgr SET iscycle TO 'y' DEFAULT 'n'
SELECT * FROM each_level; -- prikazuje i stupac iscycle
```

3. SPREČAVANJE POJAVE PETLJE U HIJERARHIJSKOJ STRUKTURI PODATAKA U JEDNOKORISNIČKOM RADU

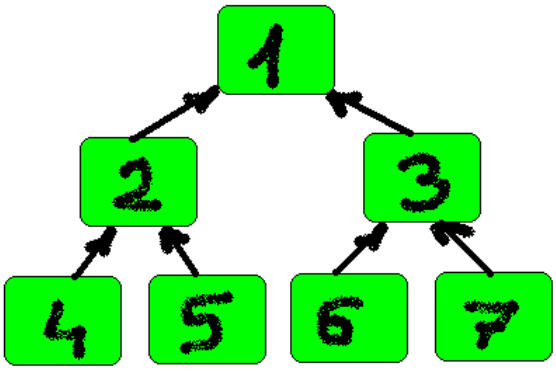
Često se želi zabraniti "začarani krug", tj. pojavu (zatvorene) petlje u jednostablastim ili višestablastim rekurzivnim strukturama podataka. Jednostablasta rekurzivna struktura je ona u kojoj točno jedan objekt nema "roditelja" (on je "vrh" stabla), a svi ostali imaju jednog "roditelja". Višestablasta rekurzivna struktura je sastavljena od više stabala, tj. barem dva objekta nemaju "roditelja", a svi ostali objekti imaju točno jednog "roditelja". Jedan primjer (jedno)stablaste strukture je dat poznatom Oracle tablicom djelatnika – "emp" (employes). Najjednostavniji opis tablice "emp" je:

```
CREATE TABLE emp (
   empno NUMBER (4) NOT NULL PRIMARY KEY,
   ename VARCHAR2 (20) NOT NULL,
   mgr   NUMBER (4) REFERENCES emp (empno)
)
/
```

Želi se da baza spriječi petlju u tablici "emp", odnosno želi se onemogućiti da se dobiju podaci u kojima bi neki djelatnik bio menadžer drugom djelatniku, a drugi djelatnik bi (direktno ili indirektno) bio menadžer prvom djelatniku.

Prvo se tablicu "emp" puni sa 7 redaka. Djelatnik sa brojem 1 bit će "glavni šef", djelatnici sa brojevima 2 i 3 bit će "šefovi" (podređeni "glavnom šefu"), djelatnici 4 i 5, odnosno 6 i 7, bit će podređeni "šefu 2", odnosno "šefu 3":

```
INSERT INTO emp (empno, ename, mgr) VALUES (1, 'EMP 1', NULL);

INSERT INTO emp (empno, ename, mgr) VALUES (2, 'EMP 2', 1);

INSERT INTO emp (empno, ename, mgr) VALUES (3, 'EMP 3', 1);

INSERT INTO emp (empno, ename, mgr) VALUES (4, 'EMP 4', 2);

INSERT INTO emp (empno, ename, mgr) VALUES (5, 'EMP 5', 2);

INSERT INTO emp (empno, ename, mgr) VALUES (6, 'EMP 6', 3);

INSERT INTO emp (empno, ename, mgr) VALUES (7, 'EMP 7', 3);
```



Slika 1. Grafički prikaz početnih podataka u tablici EMP

Ovaj zahtjev je, očito, relativno lako definirati, a dosta je čest u praksi. Međutim, nije ga lako realizirati (isključivo) u bazi podataka, tj. bez pomoći klijentske strane. Ovdje se prikazuje rješenje tog zahtjeva u Oracle bazi podataka, bez pomoći programa na klijentu ili aplikacijskom serveru (dakle, rješenje je u cijelosti na strani baze).

Rješenje u jednokorisničkom radu je relativno jednostavno. Zapravo, rješenje bi bilo vrlo jednostavno kad ne bi dolazilo do jednog problema - problema mutirajućih tablica (mutating tables). Mutirajuća tablica je ona tablica koja se trenutačno modificira pomoću naredbe INSERT, UPDATE ili DELETE, ili ona tablica koja bi trebala biti ažurirana zbog efekta DELETE CASCADE deklarativnog ograničenja. Oracle ne dozvoljava da se mutirajuće tablice čitaju (niti ažuriraju) u "row" okidačima baze (database triggers), tj. okidačima koji se okidaju za svaki redak tablice, jer bi se kao rezultat (čitanja) mogla dobiti neka neočekivana vrijednost. Pojednostavljeno rečeno, Oracle ne dozvoljava da se čita tablicu dok traje proces njene izmjene u istoj sesiji baze.

Međutim, za razliku od "row" okidača, čitanje se može raditi u "statement" okidačima, tj. okidačima koji se okidaju jedanput za svaku naredbu INSERT, UPDATE ili DELETE. Klasično rješenje problema mutirajućih tablica jeste da se u "row" okidaču zapamti (npr. u PL/SQL memorijsku tablicu) koji su redovi ažurirani, a onda se u "after statement" okidačima čita PL/SQL tablica i primjenjuje se provjera poslovnog pravila na retke koji su zapamćeni u PL/SQL tablici. Obično se želi da okidači sadrže što manje programskog koda, tako da okidači najčešće samo pozivaju pohranjene (a najčešće i pakirane) procedure ili funkcije.

Okidač "bus_emp" ("before update statement" nad tablicom "emp" - okida se jedanput prije naredbe UPDATE) poziva (pakiranu) proceduru za čišćenje PL/SQL tablice (napomena: okidači će privremeno biti invalidni, dok se ne propusti paket koji pozivaju):

```
CREATE OR REPLACE TRIGGER bus_emp
   BEFORE UPDATE ON emp
BEGIN
   emp_closed_loop.clear_plsql_tab;
END;
/
```

Okidač "bir_emp" ("before insert row" - okida se jedanput za svaki uneseni redak) provjerava da li su u stupcima "empno" i "mgr" različite vrijednosti (inače javlja grešku):

```
CREATE OR REPLACE TRIGGER bir_emp
   BEFORE INSERT ON emp
   FOR EACH ROW
BEGIN
   IF :NEW.empno = :NEW.mgr THEN
      RAISE_APPLICATION_ERROR
        (-20002, 'Djelatnik ne može biti nadređen samome sebi!');
   END IF;
END;
/
```

Okidač "bur_emp" ("before update row") zabranjuje mijenjanje šifra djelatnika, zabranjuje da djelatnik bude nadređen samome sebi i poziva proceduru koja pamti redak u PL/SQL tablicu:

```
CREATE OR REPLACE TRIGGER bur_emp
   BEFORE UPDATE ON emp
   FOR EACH ROW
BEGIN
   IF :NEW.empno <> :OLD.empno THEN
      RAISE_APPLICATION_ERROR (-20001, 'EMPNO se ne može mijenati');
   END IF;

   IF :NEW.empno = :NEW.mgr THEN
      RAISE_APPLICATION_ERROR
         (-20002, 'Djelatnik ne može biti nadređen samome sebi!');
   END IF;
```

```
IF :NEW.mgr IS NOT NULL
      AND :NEW.mgr <> NVL (:OLD.mgr, 0)
   THEN
      emp_closed_loop.write_plsql_tab (
         p_empno => :OLD.empno,
         p_mgr   => :NEW.mgr);
   END IF;
END;
/
```

Okidač "aus_emp" ("after update statement" nad tablicom "emp" - okida se jedanput nakon naredbe UPDATE) poziva (pakiranu) proceduru za provjeru poslovnog pravila (ta je procedura krucijalni dio programskog koda):

```
CREATE OR REPLACE TRIGGER aus_emp
   AFTER UPDATE ON emp
BEGIN
   emp_closed_loop.test;
END;
/
```

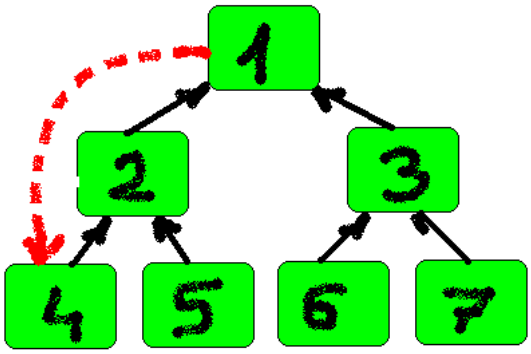
Slijedi paket "emp_closed_loop", sa tri (već navedene) procedure:

```
CREATE OR REPLACE PACKAGE emp_closed_loop IS
   PROCEDURE clear_plsql_tab;
   PROCEDURE write_plsql_tab (
      p_empno emp.empno%TYPE,
      p_mgr   emp.mgr%TYPE);
   PROCEDURE test;
END;
/

CREATE OR REPLACE PACKAGE BODY emp_closed_loop IS
   TYPE rec_t IS RECORD (
      empno emp.empno%TYPE,
      mgr   emp.mgr%TYPE);

   TYPE plsql_tab_t IS TABLE OF rec_t
      INDEX BY BINARY_INTEGER;

   m_plsql_tab plsql_tab_t;
   m_rows      BINARY_INTEGER;

   PROCEDURE clear_plsql_tab IS
   BEGIN
      m_rows := 0;
   END;


   PROCEDURE write_plsql_tab (
      p_empno emp.empno%TYPE,
      p_mgr   emp.mgr%TYPE)
   IS
   BEGIN
      m_rows := m_rows + 1;
      m_plsql_tab (m_rows).empno := p_empno;
      m_plsql_tab (m_rows).mgr   := p_mgr;
   END;
```

```
   PROCEDURE test IS
      l_mgr   emp.mgr%TYPE;
      l_empno emp.empno%TYPE;
   BEGIN
      FOR i IN 1..m_rows LOOP
         l_empno := m_plsql_tab (i).empno;
         l_mgr   := m_plsql_tab (i).mgr;

         WHILE l_mgr IS NOT NULL LOOP
            SELECT mgr INTO l_mgr
              FROM emp
             WHERE empno = l_mgr;

            IF l_mgr = l_empno THEN
               RAISE_APPLICATION_ERROR
                  (-20003, 'Greška - zatvorena petlja!');
            END IF;
         END LOOP;
      END LOOP;
   END;

END emp_closed_loop;
/
```

Vidljivo je da procedura "test" čita tablicu "emp", pa tu proceduru nije moguće pozvati u "row" okidaču "bur_emp", već se to može napraviti samo u "statement" okidaču "aus_emp". Procedura "test" mogla se napisati i drugačije (konciznije), tako da se koristi klauzula CONNECT BY naredbe SELECT. Međutim, u nastavku će se dograđivati postojeća verzija bez klauzule CONNECT BY. Sada se može testirati rješenje. Ako se nad početnim podacima primijeni sljedeća UPDATE naredbu, dobija se poruka o grešci (i to je u redu):

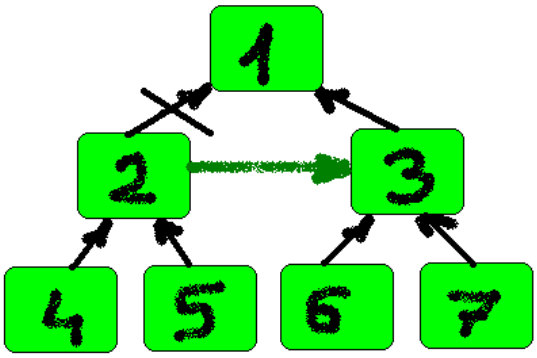
```
UPDATE emp SET mgr = 4 WHERE empno = 1;
ERROR at line 1: ORA-20003: Greška - zatvorena petlja! …
```



Slika 2. Rješenje radi dobro u jednokorisničkom radu – ne dopušta petlju

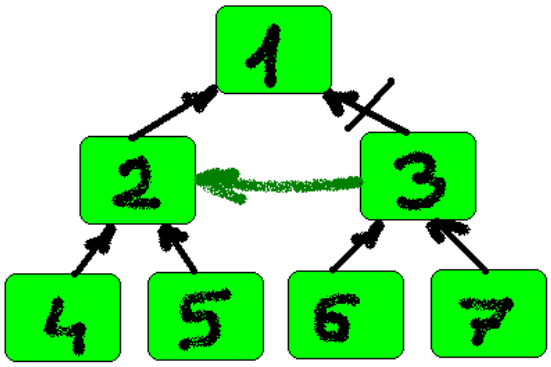
Nažalost, navedeno rješenje radi dobro samo u jednokorisničkom radu! U višekorisničkome radu (ili, što je isto, u jednokorisničkom radu u kojem korisnik ima više sesija baze), može se desiti greška, kao u sljedećem primjeru:

```
-- 1. SESIJA
UPDATE emp SET mgr = 3 WHERE empno = 2;
```



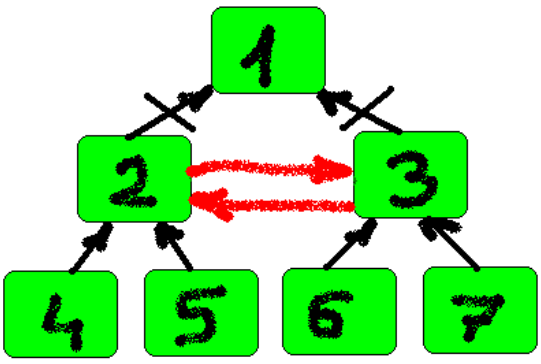
Slika 3. Naredba u prvoj sesiji je prošla i to je u redu

```
-- 2. SESIJA
UPDATE emp SET mgr = 2 WHERE empno = 3;
```



Slika 4. Prošla je i naredba u drugoj sesiji – to nije u redu

Dakle, obje naredbe su prošle i dobila se zatvorena petlja!



Slika 5. Dobila se zatvorena petlja

4. SPREČAVANJA POJAVE PETLJE U HIJERARHIJSKOJ STRUKTURI PODATAKA U VIŠEKORISNIČKOM RADU, POMOĆU AUTONOMNE TRANSAKCIJE I POMOĆU SIMULACIJE "ROLLBACK TO SAVEPOINT" U OKIDAČU BAZE

Glavna ideja za sprečavanje pojave petlje u višekorisničkom radu je da se, istovremeno dok se provjerava da li je došlo do petlje, gleda da li je tekući redak (tj. redak koji se trenutačno provjerava) zaključan. Ako je zaključan, može se pretpostaviti da bi moglo doći do zatvorene petlje, te javiti grešku. Međutim, kako provjeriti da li je redak (koji provjeravamo) zaključan? Ako se za tu namjenu koristi SELECT FOR UPDATE, zaključat će se redak sve do kraja transakcije, zato što se u okidaču Oracle baze ne može koristiti naredbu ROLLBACK TO SAVEPOINT (napomena: ovo ograničenje, kao ni ograničenje vezano za mutirajuće tablice, nije mana Oracle baze, već prednost, jer ta ograničenja sprečavaju da dođe do programskih grešaka koje bi se vrlo teško mogle otkriti). No, ako redak ostane zaključan sve do kraja transakcije, to će spriječiti druge da rade sa takvim retkom, što je neprihvatljivo.

Budući da od verzije 8i Oracle baza podržava autonomnu transakciju, može se razmišljati da se ona primijeni za rješenje problema zaključavanja. Naime, u autonomnoj transakciji može se koristiti ROLLBACK (zapravo, autonomna transakcija i mora na kraju imati ROLLBACK ili COMMIT). U tijelo paketa "emp_closed_loop" dodat će se nova autonomna procedura "test_lock":

```
PROCEDURE test_lock (p_mgr emp.mgr%TYPE) IS
   PRAGMA AUTONOMOUS_TRANSACTION;
   l_dummy NUMBER;
BEGIN
   SELECT 1 INTO l_dummy
     FROM emp
    WHERE empno = p_mgr FOR UPDATE NOWAIT;
   ROLLBACK;
EXCEPTION
   WHEN OTHERS THEN
      IF SQLCODE = -54 THEN
         RAISE_APPLICATION_ERROR
           (-20004, 'Greška - moguća zatvorena petlja!');
      ELSE
         RAISE;
      END IF;
END;
```

Nova procedura pozvat će se iz mijenjane procedure "test":

```
PROCEDURE test IS
   l_mgr   emp.mgr%TYPE;
   l_empno emp.empno%TYPE;
BEGIN
   FOR i IN 1..m_rows LOOP
      l_empno := m_plsql_tab (i).empno;
      l_mgr   := m_plsql_tab (i).mgr;

      WHILE l_mgr IS NOT NULL LOOP
         test_lock (l_mgr);

         SELECT mgr INTO l_mgr
           FROM emp
          WHERE empno = l_mgr;

         IF l_mgr = l_empno THEN
            RAISE_APPLICATION_ERROR
               (-20003, 'Greška - zatvorena petlja!');
         END IF;
      END LOOP;
   END LOOP;
END;
```

Naredbe koje su uzrokovale grešku u prethodnoj točci sada neće uspjeti, jer će baza upozoriti da bi moglo doći do petlje ("moguća greška" a ne "sigurna greška", jer to što je redak zaključan ne znači da bi do greške sigurno došlo):

```
-- 1. SESIJA
UPDATE emp SET mgr = 3 WHERE empno = 2;

-- 2. SESIJA
UPDATE emp SET mgr = 2 WHERE empno = 3;
ERROR at line 1:
ORA-20004: Greška - moguća zatvorena petlja! …
```

Nažalost, rješenje sa autonomnom transakcijom može javiti dosta lažnih grešaka (false negative), zato što su autonomnoj transakciji (baš zato što je autonomna, tj. nezavisna od "glavne" transakcije) zaključani oni redovi koje je zaključala "glavna" transakcija. Zato autonomna procedura "zaključuje" da je došlo do zatvorene petlje i onda kad je očito da nije došlo do zatvorene petlje. Evo takvog slučaja, u kojem autonomna transakcija "pogrešno zaključuje":

```
-- dvije UPDATE naredbe u istoj sesiji
UPDATE emp SET mgr = 2 WHERE empno = 6;

UPDATE emp SET mgr = 6 WHERE empno = 7;
ERROR at line 1:
ORA-20004: Greška - moguća zatvorena petlja! …
```

Iako bi bilo sasvim u redu da djelatnik broj 6 postane (direktno) nadređen djelatniku 7, druga naredba UPDATE javlja grešku zato jer autonomna procedura "test_lock" nalazi da je djelatnik 6 zaključan (zaključala ga je "glavna" transakcija, kroz prvu naredbu UPDATE).

Međutim, našli smo da je u Oracle bazi moguće simulirati SAVEPOINT / ROLLBACK TO SAVEPOINT naredbe u okidaču baze, primjenom trik rješenja. Rješenje se temelji na sljedećem: ako se poziva udaljena procedura (pomoću database linka) i ako se u njoj desi neobrađena greška, njeni se efekti u cijelosti poništavaju (za razliku od lokalne procedure). Istina, udaljena procedura nije potrebna, ali zato se radi kvazi-udaljena procedura, koristeći "lokalni" database link (link baze na sebe samu):

```
CREATE DATABASE LINK local_db_link
   CONNECT TO scott IDENTIFIED BY tiger
   USING 'local_alias' -- alias na lokalnu bazu
/
```

Ova simulacija se još u nečemu ponaša kao "pravi" ROLLBACK TO SAVEPOINT. Naime, ako druga transakcija pokuša zaključati redak koji je već zaključala prva transakcija, i ako prva transakcija otključa taj redak sa ROLLBACK TO SAVEPOINT, redak i dalje ostaje zaključan za drugu transakciju (međutim, neka treća transakcija bi sad mogla bez problema zaključati otključani redak).

Sada se može mijenjati paket "emp_closed_loop". U odnosu na prethodnu verziju, paket sada ima proceduru "test_lock" navedenu (i) u specifikaciji, zato jer se procedura "test_lock" poziva iz procedure "test" kao udaljena procedura. Procedura "test_lock" koristi naredbu "RAISE_APPLICATION_ERROR (-20999, ...)" (koju procedura "test" ignorira, tj. ne smatra ju greškom), da bi otključala redak koji je prethodno zaključala (sa SELECT ... FOR UPDATE):

```
CREATE OR REPLACE PACKAGE emp_closed_loop IS
   PROCEDURE clear_plsql_tab;
   PROCEDURE write_plsql_tab (
      p_empno emp.empno%TYPE,
      p_mgr   emp.mgr%TYPE);
```

```
   PROCEDURE test;
   PROCEDURE test_lock (p_mgr emp.mgr%TYPE);
END emp_closed_loop;
/

CREATE OR REPLACE PACKAGE BODY emp_closed_loop IS

... kao prije, osim procedura TEST i TEST_LOCK ...

   PROCEDURE test IS
      l_mgr   emp.mgr%TYPE;
      l_empno emp.empno%TYPE;
   BEGIN
      FOR i IN 1..m_rows LOOP
         l_empno := m_plsql_tab (i).empno;
         l_mgr   := m_plsql_tab (i).mgr;

         WHILE l_mgr IS NOT NULL LOOP
            BEGIN
               emp_closed_loop.test_lock@local_db_link (l_mgr);
            EXCEPTION
               WHEN OTHERS THEN
                  IF SQLCODE = -20999 THEN NULL;
                  ELSE RAISE;
                  END IF;
            END;

            SELECT mgr INTO l_mgr
              FROM emp
             WHERE empno = l_mgr;

            IF l_mgr = l_empno THEN
               RAISE_APPLICATION_ERROR
                  (-20004, 'Greška - zatvorena petlja!');
            END IF;
         END LOOP;
      END LOOP;
   END;

   PROCEDURE test_lock (p_mgr emp.mgr%TYPE) IS
      l_dummy NUMBER;
   BEGIN
      SELECT 1 INTO l_dummy
        FROM emp
       WHERE empno = p_mgr FOR UPDATE NOWAIT;

      RAISE_APPLICATION_ERROR (-20999, 'Nije važno');
   EXCEPTION
      WHEN OTHERS THEN
      -- resource busy and acquire with NOWAIT specified
         IF SQLCODE = -54 THEN
            RAISE_APPLICATION_ERROR
               (-20004,'Greška - moguća zatvorena petlja!');
         ELSE
            RAISE;
         END IF;
   END;
END emp_closed_loop;
/
```

No, mora se reći da i ovo rješenje ponekad može javiti "lažnu uzbunu", tj. javiti da je (možda)

došlo do petlje, iako do toga nije došlo, kao npr. u primjeru:

```
-- 1.SESIJA
UPDATE emp SET mgr = 6 WHERE empno = 2;

-- 2.SESIJA
UPDATE emp SET mgr = 5 WHERE empno = 7;
ERROR at line 1:
ORA-20004: Greška - moguća zatvorena petlja!…
```

Nažalost, ovakva "lažna uzbuna" ne može se spriječiti, jer sesija baze ne može točno "znati" što druge sesije baze rade.

5. TESTIRANJE KONKURENTNIH TRANSAKCIJA POMOĆU JAVA EXECUTORA

Za potrebe testiranja, tablicu EMP ćemo napuniti sa 1000 redaka, na ovaj način:

- 1 korijenski redak (šifra 0)

- njegovih 9 podređenih

- njihovih 90 podređenih (po 10 za svakog)

- njihovih 900 podređenih (po 10 za svakog).

```
-- 0
insert into emp (empno, ename, mgr) values (0, 'EMP 0', null);
-- 1-9
declare
   ename varchar2(20);
begin
   for j in 1..9 loop
      ename := 'EMP ' || j;
      insert into emp (empno, ename, mgr) values (j, ename, 0);
   end loop;
end;
/

-- 10-99
declare
   empno number(4);
   ename varchar2(20);
begin
   for i in 1..9 loop
      for j in 0..9 loop
         empno := i * 10 + j;
         ename := 'EMP ' || empno;
         insert into emp (empno, ename, mgr) values (empno, ename, i);
      end loop;
   end loop;
end;
/

-- 100-999
declare
   empno number(4);
   ename varchar2(20);
begin
   for i in 10..99 loop
      for j in 0..9 loop
```

```
         empno := i * 10 + j;
         ename := 'EMP ' || empno;
         insert into emp (empno, ename, mgr) values (empno, ename, i);
      end loop;
   end loop;
end;
/

commit;
```

Testiranje ćemo raditi pomoću Java programa (u nastavku) koji ima sljedeće ulazne parametre:

- brDretvi: broj Java dretvi (default je 10)

- cekanje: vrijeme namjernog čekanja u pojedinoj dretvi (default je 1 sekunda), kako bi se simuliralo kašnjenje kod rada korisnika

- brIteracija: broj ponavljanja testa (default je 1).

Glavna metoda **main** poziva metodu **testiraj**, u kojoj se kreira objekt (connectionTask) anonimne klase (podklasa od Runnable). U (nadjačanoj) metodi **run** dvaput se poziva metoda **slucajni_broj**, koja generira slučajne emp i mgr (brojeve između 200 i 300). Na temelju toga se radi UPDATE jednog retka i COMMIT. Kreira se fiskni broj executora sa **newFixedThreadPool(brDretvi)** i svakom se daje njegov zadatak sa **executorService.submit(connectionTask)**.

```
import java.sql.*;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import static java.util.concurrent.TimeUnit.MINUTES;


public class TestEmp {

static int brDretvi = 10;
static int cekanje = 1; // 1 sekunda
static int brIteracija = 1;

public static void main(String[] args)  {
   if (args.length > 0) {
      brDretvi = Integer.parseInt(args[0]);
   };

   if (args.length > 1) {
      cekanje = Integer.parseInt(args[1]) * 1000;
   };

   if (args.length > 2) {
      brIteracija = Integer.parseInt(args[2]);
   };

   for (int i = 1; i <= brIteracija; i++) {
      testiraj();
   };
};
```

```
private static void testiraj() {
   String url = "jdbc:oracle:thin:emp/emp@localhost:1521:ORCL";

   Runnable connectionTask = new Runnable() {
      public void run() {
         try (Connection con = DriverManager.getConnection(url);) {
           con.setAutoCommit(false);
           String query;
           PreparedStatement stm;

           // ORA-01436: CONNECT BY loop in user data
           query =
              "select distinct 0 from emp connect by prior empno = mgr";
           stm = con.prepareStatement(query);
           stm.executeQuery();

           // s donjim postavkama dolazi do zatvorene petlje
           // kod nesigurne verzije paketa
           int mgr   = slucajni_broj (200, 300);
           // nije pogodno: slucajni_broj (0, 999);
           int empno = slucajni_broj (200, 300);

           query = "update emp set mgr = ? where empno = ?";
           stm = con.prepareStatement(query);
           stm.setInt(1, mgr);
           stm.setInt(2, empno);
           stm.execute();
           Thread.sleep(cekanje);
           con.commit();
           System.out.println("Empno:" + empno + " Mgr:" + mgr);
        } catch (Exception e) {
           // System.out.println(e.toString().substring(34, 67));
           System.out.println(e.toString());
        }
     }
   };

   try {
      ExecutorService executorService =
         Executors.newFixedThreadPool(brDretvi);
      for (int j = 1; j <= brDretvi; j++) {
         executorService.submit(connectionTask);
      }
      executorService.shutdown();
      executorService.awaitTermination(30, MINUTES);
   } catch (Exception e) {
      System.out.println(e.toString());
   }
};

private static int slucajni_broj (int min_p, int max_p) {
   /*
   Generira slučajni cijeli broj od min (uključujući) do max (uključujući).
   Math.random vraća vrijednost od 0 (uključujući) do 1 (isključujući).
   Zato se množi s max_p i zbraja min_p. Vraća se u cijeli broj.
   */
   int broj = min_p + (int) (Math.random() * (max_p - min_p + 1));
   return broj;
}

}
```

Testiranje možemo raditi npr. ovako – pokreće se 100 Java dretvi (time i 100 paralelnih transakcija), sa vremenom čekanja od 2 sekunde u transakciji, u 5 iteracija testiranja:

```
java –cp ”*”; TestEmp 100 2 5
```

Pokazuje se da nesigurna varijanta paketa **emp_closed_loop** vrlo brzo dovodi do greške. Sigurne (dvije) varijante tog paketa ne dovode do greške niti nakon puno ponavljanja. Naravno, to nije matematički dokaz da su to zaista sigurna varijante. Kao i uvijek, testiranjem se može dokazati da program ne radi dobro, ali se testiranjem ne može dokazati da program (uvijek) radi dobro.

6. TESTIRANJE KONKURENTNIH TRANSAKCIJA POMOĆU JOBOVA NA BAZI

PL/SQL paket (u nastavku) ima javne (public) procedure:

- pokreni: parametar je broj jobova (default je 10)

- zadatak: procedura je javna zato jer se poziva iz joba, pa mora biti navedena u specifikaciji paketa (inače bi se javilo: ORA-06576: not a valid function or procedure name).

U proceduri **pokreni** kreira se i pokreće zadani broj jobova. Kod poziva DBMS_SCHEDULER.RUN_JOB parametar **use_current_session** postavlja se na FALSE (default je TRUE), kako bi svaki job radio u posebnoj sesiji baze. Procedura **zadatak** (slično ekvivalentnom dijelu koda u Java programu) postavlja upit kojim se utvrđuje da li je došlo do petlje, a onda (ako nije došlo do petlje) slučajno generira šifru za mgr i empno, radi UPDATE, čeka jednu sekundu, i daje COMMIT (ovdje nema parametra za broj iteracija testiranja).

```
-- kao SYS
GRANT CREATE JOB TO emp
/

CREATE OR REPLACE PACKAGE test_emp IS
   PROCEDURE pokreni (br_jobova_p NUMBER DEFAULT 10);
   PROCEDURE zadatak;
END;
/

CREATE OR REPLACE PACKAGE BODY test_emp IS
   PROCEDURE pokreni (br_jobova_p NUMBER DEFAULT 10) IS
   BEGIN
      FOR j IN 1 .. br_jobova_p LOOP
         DBMS_SCHEDULER.CREATE_JOB (
            job_name   => 'TEST_EMP_' || j,
            job_type   => 'STORED_PROCEDURE',
            job_action => 'TEST_EMP.ZADATAK',
            enabled    => TRUE,
            auto_drop  => TRUE -- default
         );

         DBMS_SCHEDULER.RUN_JOB (
            job_name => 'TEST_EMP_' || j,
            use_current_session => FALSE
         );
      END LOOP;
   END;
```

```
   PROCEDURE zadatak IS
      mgr_l   NUMBER (4);
      empno_l NUMBER (4);
      dummy_l NUMBER (1);
   BEGIN
      -- može se desiti ORA-01436: CONNECT BY loop in user data
      SELECT DISTINCT 0 INTO dummy_l
        FROM emp
      CONNECT BY PRIOR empno = mgr;

      mgr_l   := TRUNC (DBMS_RANDOM.VALUE (200, 301));
      empno_l := TRUNC (DBMS_RANDOM.VALUE (200, 301));

      UPDATE emp
         SET mgr = mgr_l
       WHERE empno = empno_l;

      DBMS_LOCK.SLEEP (1);
      COMMIT;
   END;
END;
/

-- pokretanje testiranja
exec test_emp.pokreni (50)

-- provjera da li su jobovi završili
select job_name
  from user_scheduler_jobs
 order by 1;

-- da li je došlo do petlje
select distinct 0
  from emp.emp
connect by prior empno = mgr;

-- detalji rada jobova – prikaz grešaka
select error#, count(*)
  from USER_SCHEDULER_JOB_RUN_DETAILS
 where job_name like 'TEST_EMP_%'
 group by error#
 order by 1;
```

7. ZAKLJUČAK

Prvo smo se ukratko podsjetili na dvije varijante SQL hijerarhijskih upita – Oracle varijantu (koju je Oracle baza imala od početka) i ANSI varijantu (moguća od Oracle baze 11.2). Zatim smo ponovili prezentaciju sa CASE 16 (2004.), koja prikazuje tri rješenja za sprečavanje zatvorene petlje u hijerarhijskoj strukturi (tablica u bazi podataka). Zapravo, prvo rješenje je nesigurno - radi dobro samo u jednokorisničkom radu. Za preostala dva rješenja vjerujemo (nemamo formalnog dokaza) da rade dobro i u višekorisničkom radu.

Na tom primjeru prikazali smo testiranje konkurentnih transakcija pomoću Java Executora Pokazalo se da nesigurna varijanta vrlo brzo uzrokuje petlju u podacima. Ostale dvije varijante su dobro radile. Zatim smo prikazali testiranje konkurentnih transakcija pomoću jobova na bazi (DBMS_SCHEDULER paketa). To je testiranje dalo iste rezultate kao i prethodno, što je i očekivano, jer oba testiranja rade na sličan način, samo se za paralelno izvršavanje u prvom slučaju koriste Java dretve, a drugom slučaju jobovi na bazi.

Naravno, postoje i bolji načini testiranja konkurentnih transakcija – pomoću gotovih alata namijenjenih tome, od kojih su neki (možda) i besplatni. No, nije loše, barem iz edukativnih razloga, probati testiranje (i) pomoću vlastitih, jednostavnih programa.

LITERATURA:

[1] Boyarsky, J., Selikoff, S. (2015): OCP:
Oracle Certified Professional Java SE 8 Programmer II Study Guide: Exam 1Z0-809, Sybex

[2] Kyte, T. (2009): Expert Oracle Database Architecture, Apress

[3] Oracle priručnik (2010): Oracle Database Administrator's Guide 11g Release 2

[4] Oracle priručnik (2010): Oracle Database Advanced Application Developer's Guide 11g Release 2

[5] Oracle priručnik (2013): Oracle Database Development Guide 12c Release 1

[6] Oracle priručnik (2013): Oracle Database JDBC Developer’s Guide 12c Release 1

[7] Sierra K., Bates B. (2015):
OCA/OCP Java SE 7 Programmer I & II Study Guide, McGraw-Hill Education

Zlatko Sirotić, univ.spec.inf.
ISTRA TECH d.o.o., Pula
e-mail: zlatko.sirotic@istratech.hr

Autor radi oko 35 godina na informatičkim poslovima, uglavnom u poduzeću ISTRA TECH d.o.o., Pula (ISTRA TECH je novo ime poduzeća Istra informatički inženjering, osnovanog prije 28 godina). Oracle softverske alate (baza, Designer CASE, Forms 4GL, Reports, JDeveloper IDE, Java) koristi više od 20 godina. Objavljivao je stručne radove na kongresima / konferencijama CASE, KOM, HrOUG, JavaCro, "Hotelska kuća", u časopisima "Mreža", "InfoTrend" i "Ugostiteljstvo i turizam", a neka njegova programska rješenja objavljivana su na web stranicama firmi Oracle i Quest (danas dio firme Dell). Na Fakultetu informatike u Puli sudjeluje (od početka osnivanja, 2011.) kao vanjski suradnik, uglavnom na kolegijima Baze podataka 2 i Informatički praktikum 1.