

Pohranjene procedure/funkcije

Primjer 1:

	CHAR (11)	CHAR (30)
osoba	oib	prez
	22039796228	Horvat
	1712 128712	Kolar
	27079867362	Še5fer
	03AB621.228	Novak

- smatra se da su ispravne one vrijednosti atributa OIB u kojima postoji točno 11 znamenaka
- smatra se da su ispravna ona prezimena u kojima ne postoji niti jedna znamenka
- ispisati podatke o osobama s neispravnim OIB-om ili prezimenom
- **kad bi barem postojala SQL funkcija CountDigits(nizZnakova)**

```
SELECT * FROM osoba
WHERE CountDigits(oib) <> 11
      OR CountDigits(prez) > 0;
```

Pohranjene procedure (pohranjene funkcije)

- Pohranjena procedura ili pohranjena funkcija je potprogram koji je pohranjen u rječniku podataka i koji se izvršava u kontekstu sustava za upravljanje bazama podataka
 - može se promatrati kao procedura ili funkcija kojom se proširuje skup SQL funkcija ugrađenih u SUBP
 - procedura je potprogram koji u pozivajući program ne vraća rezultat
 - funkcija je potprogram koji u pozivajući program vraća rezultat

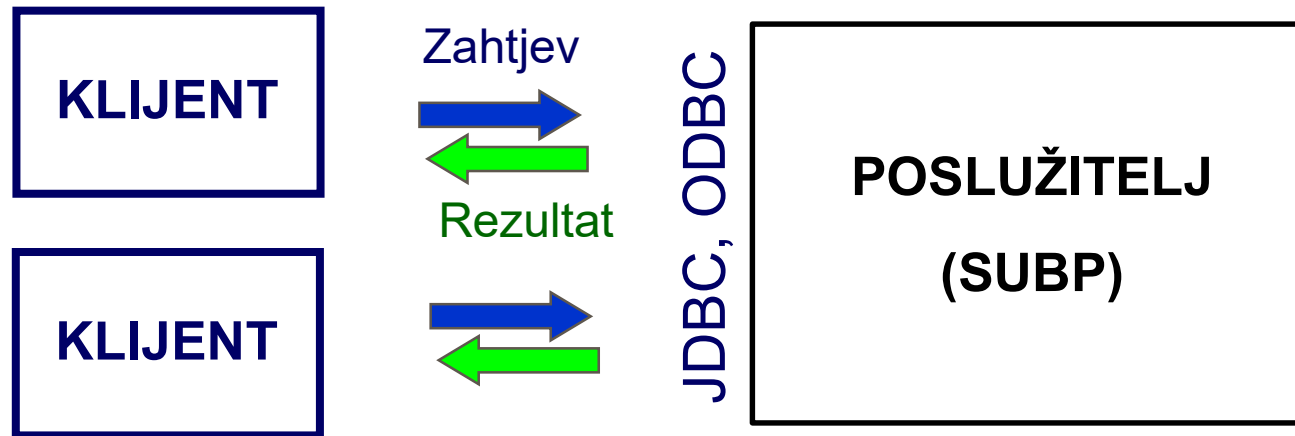
SPL (*Stored Procedure Language*)

- Proizvođači SUBP koriste vlastite inačice jezika za definiranje pohranjenih funkcija/procedura (standard postoji, ali je rijetko gdje implementiran)
 - IBM Informix: SPL (Stored Procedure Language)
 - Oracle: PL/SQL (Procedural Language/Structured Query Language)
 - Microsoft SQL Server: Transact-SQL
 - PostgreSQL: PL/pgSQL
- Navedeni jezici proširuju mogućnosti SQL jezika proceduralnim elementima koji se koriste u strukturiranim jezicima (C, Java, ...). Osim SQL naredbi, pohranjene procedure omogućuju korištenje
 - varijabli
 - naredbi za kontrolu toka programa (*if, for, while, ...*)
 - naredbi za rukovanje iznimkama (*exception handling*)

Prednosti uporabe pohranjenih procedura

- proširenje mogućnosti SQL jezika
- omogućena je zaštita podataka na razini funkcije (a ne samo objekta)
- omogućena je uporaba klijent-poslužitelj arhitekture oslonjene na poslužitelj:
 - postiže se veća učinkovitost SUBP
 - SUBP ne mora ponavljati prevođenje i optimiranje SQL upita
 - postiže se veća produktivnost programera i smanjuje mogućnost pogreške
 - programski kôd potreban za obavljanje nekog postupka koji čini logičku cjelinu implementira se i testira na samo jednom mjestu
 - **veće opterećenje poslužitelja (usko grlo)**

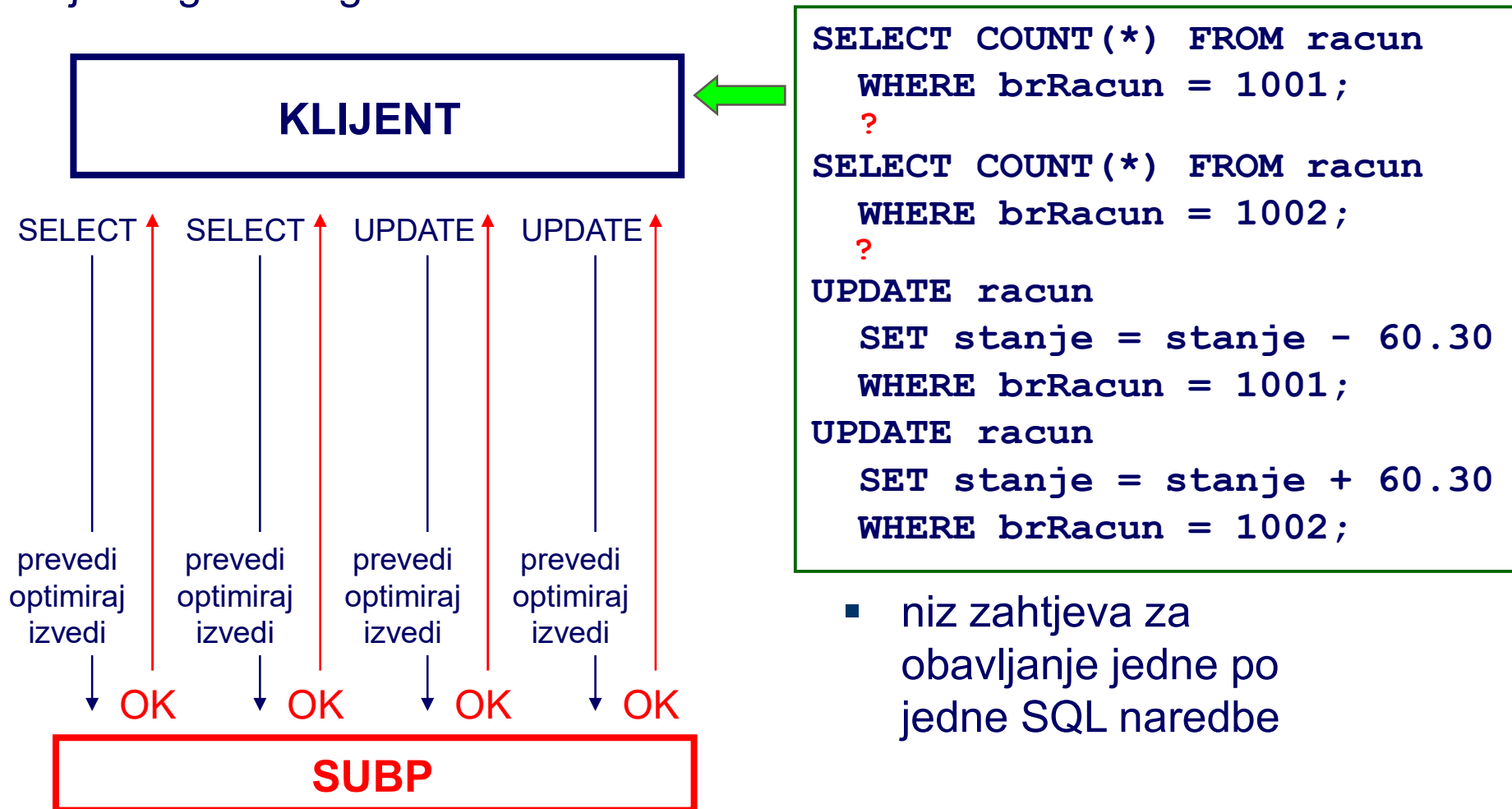
(Klijent-poslužitelj arhitektura)



- sustav obuhvaća dvije komponente
 - klijent i poslužitelj (*client-server*)
- koncept zahtjev-odgovor (*request-response*): klijent postavlja zahtjev, poslužitelj odgovara
- komunikacija između klijenta i poslužitelja se odvija preko dobro definiranih, standardnih programskih sučelja: npr. ODBC (*Open Database Connectivity*), JDBC (*Java Database Connectivity*)

(Klijent-poslužitelj arhitektura - oslonjena na klijenta)

- provjeri postoje li zadani brojevi računa, ako postoje, prebaci iznos s jednog na drugi račun



(Klijent-poslužitelj arhitektura - oslonjena na poslužitelj)

- provjeri postoje li zadani brojevi računa, ako postoje, prebaci iznos s jednog na drugi račun



EXECUTE

izvedi

OK



- U rječnik podataka se pohranjuje:
 - izvorni kôd procedure
 - prevedeni i optimirani kôd procedure

```
CREATE PROCEDURE prebaci (...)  
  DEFINE ...  
  SELECT ...  
  SELECT ...  
  UPDATE ...  
  UPDATE ...  
END PROCEDURE;
```

Struktura PL/pgSQL

- *Block-structured*
- *Block:*

```
[ <<label>> ]  
[ DECLARE  
    declarations ]  
BEGIN  
    statements  
END [ label ];
```

- Naredbe se odvajaju s ;
- Moguće je gnijezditi blokove
 - tada **END** mora imati ;
 - Prekrivanje varijabli (*scope*), kao u C-u
- Ignore case, komentari -- i /* */
- **label**(oznaka) se koristi samo:
 - Ako se hoće referencirati blok, npr. varijabla iz bloka

Digresija – dollar quoting

- `$$` - *dollar quoting*
 - elegantniji način pisanja string konstanti kada sadrže navodnike
 - Nema (posebne) veze s funkcijama

```
SELECT 'Age is an issue of mind over matter.  
        If you don't mind, it doesn't matter.'  
  
SELECT $$Age is an issue of mind over matter.  
        If you don't mind, it doesn't matter.$$  
  
SELECT $x$Age is an issue of mind over matter.  
        If you don't mind, it doesn't matter.$x$
```

"imenovani" navodnici za referencu
(ako "gnijezdimo" \$\$-ove, ovako znamo gdje su granice niza)

Funkcije, povratna vrijednost, primjer

- Tip povratne vrijednosti funkcije se definira s RETURNS
- Tijelo funkcije je string konstanta

```
CREATE FUNCTION gimme_quote() RETURNS TEXT AS
$$
    BEGIN
        RETURN $x$Age is an issue of mind over matter.
            If you don't mind, it doesn't matter.$x$;
    END;
$$ LANGUAGE plpgsql;
```

Definicija varijabli

- Sve varijable koje se koriste u bloku moraju biti prethodno definirane
 - Iznimka: FOR loop varijabla za iteriranje po cijelim brojevima se automatski definira, kao i za iteriranje po kursoru (kasnije)
 - Može biti bilo kojeg SQL tipa
- Sintaksa:

```
name [ CONSTANT ] type [ NOT NULL ]  
    [ { DEFAULT | := | = } expression ] ;
```

- Npr.

```
brojac SMALLINT;  
url TEXT = 'http://www.fer.hr';
```

Definicija varijabli (2)

- DECLARE blok nalazi se na početku tijela funkcije, prije ključne riječi BEGIN
- funkcijski argumenti su također "definirane varijable"
- Npr.

```
CREATE FUNCTION brojZnamenki(niz TEXT) RETURNS INT AS
$$
    DECLARE
        brojac SMALLINT;
    BEGIN
        . . . .
```

- Definicija varijable koja je jednakog tipa kao atribut relacije:

```
. . . .
DECLARE
    sRacunaBr racun.brRacun%TYPE;
. . . .
```

Definicija argumenata funkcije

- Funkcijski argumenti se mogu referencirati kao \$1, \$2, ... ali puno je bolje (čitljivije) dodijeliti im ime u samom zaglavlju funkcije

```
CREATE FUNCTION pdv(iznos REAL) RETURNS real AS
$$
    BEGIN
        RETURN iznos * 0.25;  -- ili $1 * 0.25
    END;
$$ LANGUAGE plpgsql;
```

- Funkciju pozivamo uz pomoć ključne riječi SELECT

```
SELECT pdv(10);
```

pdv
2.5

Primjer 1 (nastavak):

```
-- broji koliko ima znakova koji su znamenke (broji znakove iz intervala '0' ... '9')
CREATE FUNCTION brojZnamenki (niz TEXT) RETURNS SMALLINT AS
$$
    DECLARE
        brojac SMALLINT;
    BEGIN
        brojac = 0;
        IF (niz IS NULL) THEN
            RETURN 0;
        END IF;
        FOR i IN 1..CHAR_LENGTH(niz) LOOP
            IF SUBSTRING(niz FROM i FOR 1) BETWEEN '0' AND '9' THEN
                brojac = brojac + 1;
            END IF;
        END LOOP;
        RETURN brojac;
    END;
$$ LANGUAGE plpgsql;
```

- Funkcija je napravljena u PUBLIC shemi, svi (PUBLIC) imaju pravo pokretanja
- Naravno, moguće je:
 - REVOKE EXECUTE ON FUNCTION brojZnamenki FROM PUBLIC;
 - te potom npr.: GRANT EXECUTE ON FUNCTION brojZnamenki TO horvat;

Primjer 1 (nastavak):

	CHAR (11)	CHAR (30)
osoba	oib	prez
	22039796228	Horvat
	1712 128712	Kolar
	27079867362	Še5fer
	03AB621.228	Novak

- funkcija brojZnamenki se može iskoristiti za ispis onih osoba u čijem OIB-u nema točno 11 znamenaka ili u prezimenu postoje znamenke

```
SELECT *, brojZnamenki(oib) AS br1, brojZnamenki(prez) AS br2
FROM osoba
WHERE brojZnamenki(oib) <> 11
      OR brojZnamenki(prez) > 0;
```

oib	prez	br1	br2
1712 128712	Kolar	12	0
27079867362	Še5fer	13	1
03AB621.228	Novak	10	0

Primjer 2:

- Korisnik novak je službenik u banci kojem je potrebno omogućiti obavljanje **isključivo** jedne vrste bankovne transakcije: prebacivanje iznosa s jednog na drugi račun

racun	brRacun	stanje
	1001	1250.15
	1002	-300.00
	1003	10.25

- Zadatak se **ne može** riješiti dodjelom dozvole za obavljanje operacije UPDATE nad relacijom racun korisniku novak (**zašto?**)

Dozvole za funkcije

- SQL naredbe za dodjeljivanje i ukidanje dozvola za izvršavanje procedura

```
GRANT EXECUTE ON {  
    {FUNCTION} function_name([atype [, ...] ] )}  
    TO role_specification [, ...] [ WITH GRANT OPTION ]
```

```
REVOKE [ GRANT OPTION FOR ] EXECUTE ON {  
    {FUNCTION} function_name([atype [, ...] ] )}  
    FROM { [ GROUP ] role_name | PUBLIC }  
    [ CASCADE | RESTRICT ]
```

Primjer 2 (nastavak):

```
CREATE FUNCTION prebaci(sRacunaBr racun.brRacun%TYPE
                        , naRacunBr  racun.brRacun%TYPE
                        , iznos       racun.stanje%TYPE) RETURNS VOID AS
-- vraća VOID, pa se može smatrati procedurom
-- CREATE PROCEDURE prebaci(...) AS
$$
BEGIN
    UPDATE racun SET stanje = stanje - iznos
        WHERE brRacun = sRacunaBr;
    UPDATE racun SET stanje = stanje + iznos
        WHERE brRacun = naRacunBr;
END;
$$ LANGUAGE plpgsql;

REVOKE EXECUTE ON prebaci(int, int, decimal) FROM PUBLIC;
GRANT EXECUTE ON prebaci(int, int, decimal) TO novak;
```

- Općenito, bolje je nešto vratiti, npr. ovdje bi se mogao vratiti BOOLEAN u smislu uspješnosti obavljanja funkcije

Primjer 2 (nastavak):

racun	brRacun	stanje
	1001	1250.15
	1002	-300.00
	1003	10.25

novak `UPDATE racun SET stanje = stanje - 60.30
WHERE brRacun = 1001;`

[Error] ERROR: permission denied for relation racun

novak `SELECT prebaci (1001, 1002, 60.30);`

ERROR: permission denied for relation racun

**CONTEXT: SQL statement "UPDATE racun SET stanje = stanje - iznos
WHERE brRacun = sRacunaBr"**

**PL/pgSQL function prebaci(integer,integer,numeric) line 3 at SQL
statement**

- S čijim dozvolama se obavlja funkcija?

Dozvole prilikom obavljanja funkcije

```
CREATE FUNCTION prebaci (...) RETURNS VOID AS  
$$  
    BEGIN  
        ...  
    END;  
$$ LANGUAGE plpgsql SECURITY INVOKER;
```

DEFAULT

Funkcija se obavlja s dozvolama korisnika koji ju je pokrenuo. Preddefinirano ponašanje

```
CREATE FUNCTION prebaci (...) RETURNS VOID AS  
$$  
    BEGIN  
        ...  
    END;  
$$ LANGUAGE plpgsql SECURITY DEFINER;
```

Funkcija se obavlja s dozvolama korisnika koji ju je napravio.

Primjer 2 (nastavak):

racun	brRacun	stanje
	1001	1250.15
	1002	-300.00
	1003	10.25

- Konačno, ako smo ponovo napravili funkciju sa SECURITY DEFINER i dali EXECUTE dozvolu:

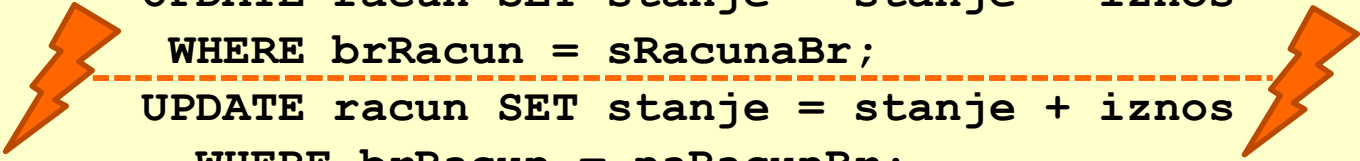
novak

```
SELECT prebaci (1001, 1002, 60.30);
```

racun	brRacun	stanje
	1001	1189.85
	1002	-239.70
	1003	10.25

Problem – kvar za vrijeme obavljanja funkcije?

```
CREATE FUNCTION prebaci(sRacunaBr  racun.brRacun%TYPE
                        , naRacunBr  racun.brRacun%TYPE
                        , iznos       racun.stanje%TYPE) RETURNS VOID AS
$$
  BEGIN
    UPDATE racun SET stanje = stanje - iznos
    WHERE brRacun = sRacunaBr;
    UPDATE racun SET stanje = stanje + iznos
    WHERE brRacun = naRacunBr;
  END;
$$ LANGUAGE plpgsql;
```



- Ako se završetak transakcije (COMMIT ili ROLLBACK) ne regulira eksplicitno u funkciji/proceduri onda se naredbe funkcije obavljaju unutar transakcije ustanovljene vanjskim (pozivajućim) upitom

Primjer 2 (nastavak):

racun	brRacun	stanje
	1001	1250.15
	1002	-300.00
	1003	10.25

- Problem: što će se dogoditi ako korisnik pri pozivu procedure kao broj prvog računa zada postojeći, a kao broj drugog računa zada nepostojeći broj računa?

```
SELECT prebaci(1001, 1005, 30.15);
```

Iznimke (*Exceptions*)

- ukoliko SUBP tijekom obavljanja operacije utvrdi da se dogodila pogreška (*error condition*), obavljanje operacije se prekida, a stanje pogreške se signalizira iznimkom (*exception*)

```
SELECT (stanje/(stanje-10.25)) FROM racun;
```

[Error] An attempt was made to divide by zero.

```
SELECT * FROM ispit;
```

[Error] No SELECT permission.

- pogreške koje SUBP nije u stanju prepoznati (jer ih ne smatra pogreškama), mogu se signalizirati naredbom **RAISE EXCEPTION**. Npr, u poboljšanoj proceduri **prebaci** signalizira se pogreška u slučaju kad ne postoji neki od zadanih brojeva računa

```
SELECT prebaci(1001, 1005, 30.15);
```

[Error] Ne postoji drugi račun

Dojavljivanje pogrešaka

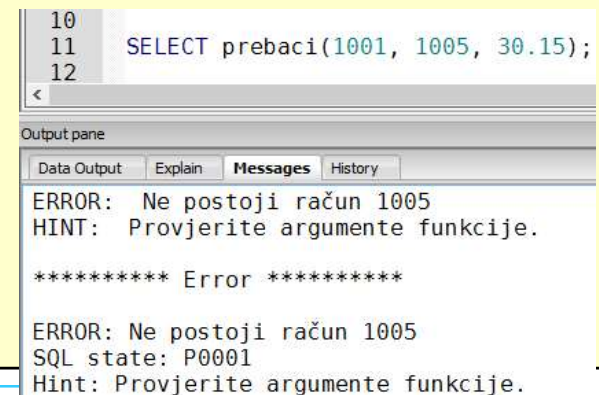
```
-- prikazan je samo dio sintakse  
RAISE [ level ] 'format' [, expression [, ... ]]  
      [ USING option = expression [, ... ] ];
```

- Šest razina (level):
 - DEBUG, LOG, INFO, NOTICE, WARNING
 - EXCEPTION (*default*) – signalizira grešku, što uobičajeno otkazuje transakciju

```
RAISE EXCEPTION 'Ne postoji račun %', sRacunaBr  
      USING HINT = 'Provjerite argumente funkcije.';
```

Primjer 2 (nastavak):

```
CREATE FUNCTION prebaci (sRacunaBr  racun.brRacun%TYPE
                        , naRacunBr  racun.brRacun%TYPE
                        , iznos       racun.stanje%TYPE) RETURNS VOID AS
$$
BEGIN
    -- provjeri postoje li zadani brojevi računa
    IF (SELECT COUNT(*) FROM racun
        WHERE brRacun = sRacunaBr) = 0 THEN
        RAISE EXCEPTION 'Ne postoji račun %', sRacunaBr
        USING HINT = 'Provjerite argumente funkcije.';
    END IF;
    IF (SELECT COUNT(*) FROM racun
        WHERE brRacun = naRacunBr) = 0 THEN
        RAISE EXCEPTION 'Ne postoji račun %', naRacunBr
        USING HINT = 'Provjerite argumente funkcije.';
    END IF;
    UPDATE racun SET stanje = stanje - iznos
        WHERE brRacun = sRacunaBr;
    UPDATE racun SET stanje = stanje + iznos
        WHERE brRacun = naRacunBr;
END;
$$ LANGUAGE plpgsql SECURITY DEFINER;
```



```
10
11 SELECT prebaci(1001, 1005, 30.15);
12
Output pane
Data Output Explain Messages History
ERROR: Ne postoji račun 1005
HINT: Provjerite argumente funkcije.

***** Error *****

ERROR: Ne postoji račun 1005
SQL state: P0001
Hint: Provjerite argumente funkcije.
```

Okidači

Primjer 3:

racun	brRac	sifKlijent	stanje
	1001	98281	216.80
	1002	89734	134.99
	1003	23232	2750.00
	1004	63443	849.50

uplataIsplata	brRac	vrijeme	iznos
	1001	7.8.2007 08:20	15.00
	1002	9.4.2006 12:31	-100.21
	1001	6.5.2007 14:15	452.15
	1004	5.5.2007 16:42	1200.00
	1004	9.9.2005 10:15	-350.50
	1002	7.2.2007 15:01	235.20
	1003	1.4.2005 12:44	2750.00
	1001	1.9.2007 12:19	-250.35

- u relaciju **uplataIsplata** upisuju se promjene na računima
- tijekom godina evidentiran je vrlo veliki broj uplata i isplata
- stanje na određenom računu moglo bi se izračunati zbrajanjem iznosa u relaciji **uplataIsplata**, koji se odnose na dotični račun
- u ovom primjeru, uz svaki račun se redundantno pohranjuje trenutno stanje računa, koje u svakom trenutku mora odgovarati stanju koje bi se dobilo zbrajanjem iznosa u relaciji **uplataIsplata**
- kako osigurati da se pri svakoj relevantnoj promjeni podataka (unos, brisanje, izmjena iznosa) u relaciji **uplataIsplata** izmijeni i odgovarajuće stanje u relaciji **racun**?

Aktivne baze podataka

- konvencionalni SUBP je **pasivan**
 - operacije nad podacima se izvršavaju isključivo na temelju eksplicitnog zahtjeva korisnika/aplikacije
- **aktivni** SUBP i aktivne baze podataka
 - aktivni SUBP autonomno reagira na određene događaje (*events*)
 - u aktivnim bazama podataka neke operacije nad podacima se izvršavaju automatski, reakcijom na određeni događaj ili stanje
- željeno ponašanje sustava postiže se definiranjem aktivnih pravila (*active rules*)
- najčešće korištena paradigma za opisivanje aktivnih pravila u današnjim SUBP je događaj-uvjet-akcija (*ECA: Event-Condition-Action*)
 - okidači (*triggers*)

ECA

on *event*

if *condition* **then** *action*

- događaj (*event*): ako se dogodi, izračunava se uvjet
 - općenito, događaji mogu biti:
 - unos, izmjena ili brisanje podatka
 - čitanje podatka
 - uspostavljanje SQL-sjednice
 - protok određene količine vremena, dostizanje trenutka u vremenu, ...
- uvjet (*condition*): ako je rezultat izračunavanja uvjeta istina, obavljaju se akcije
 - zadaje se u obliku predikata (slično kao u WHERE dijelu SQL naredbi)
- akcije (*action*): niz operacija, najčešće operacije nad podacima
 - SQL naredbe INSERT, UPDATE, DELETE, poziv procedure, ...

Primjer 3 (nastavak):

- kako osigurati da se pri svakoj relevantnoj promjeni podataka (unos, brisanje, izmjena iznosa) u relaciji **uplataIsplata** izmijeni i odgovarajuće stanje u relaciji **racun**?

racun	brRac	sifKlijent	stanje
	1001	98281	216.80
	1002	89734	134.99

uplataIsplata	brRac	vrijeme	iznos
	1001	7.8.2007 08:20	15.00
	1002	9.4.2006 12:31	-100.21
	1001	6.5.2007 14:15	452.15

- potrebno je utvrditi koji događaji mogu uzrokovati neispravnu vrijednost atributa stanje u relaciji racun, te pod kojim uvjetima treba obaviti koje akcije kako bi se očuvao integritet podataka, npr.
- događaj: obavljanje operacije INSERT nad relacijom uplataIsplata
- uvjet: iznos \neq 0.00
- akcija: pribrojiti vrijednost atributa iznos unesene n-torke u odgovarajuće stanje

Primjer 3 (nastavak), INSERT:

- događaj: obavljanje operacije INSERT nad relacijom uplataisplata
- uvjet: iznos \neq 0.00
- akcija: pribrojiti vrijednost atributa iznos unesene n-torke u odgovarajuće stanje

```
CREATE FUNCTION sync_racun_insert() RETURNS trigger AS
$$
BEGIN
    UPDATE racun
        SET stanje = stanje + NEW.iznos
        WHERE brRac = NEW.brRac;
    RETURN NEW; -- obavezno!
END;
$$ LANGUAGE plpgsql;
```

```
CREATE TRIGGER ins_sync_racun
    BEFORE INSERT ON uplataisplata
    FOR EACH ROW
    WHEN (NEW.iznos  $\neq$  0)
    EXECUTE PROCEDURE sync_racun_insert();
```

Predstavlja ntorku.
**Ako se vrati null
akcija se otkazuje!**

Novoj ntorki se
pristupa putem
varijable NEW.

U PostgreSQL-u, akcija
mora biti definirana
putem funkcije

Primjer 3 (nastavak), DELETE:

- događaj: brisanje n-torke iz relacije uplataisplata
- uvjet: iznos \neq 0.00
- akcija: oduzeti vrijednost atributa iznos unesene n-torke od odgovarajućeg stanja

```
CREATE FUNCTION sync_racun_delete() RETURNS trigger AS
$$
BEGIN
    UPDATE racun
        SET stanje = stanje - OLD.iznos
        WHERE brRac = OLD.brRac;
    RETURN OLD; -- obavezno!
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER del_sync_racun
    BEFORE DELETE ON uplataisplata
    FOR EACH ROW
    WHEN (OLD.iznos  $\neq$  0)
    EXECUTE PROCEDURE sync_racun_delete();
```

Staroj ntorki se pristupa putem varijable OLD.

Primjer 3 (nastavak), UPDATE:

- događaj: izmjena vrijednosti atributa iznos u relaciji uplataispata
- uvjet: nova vrijednost iznosa \neq stara vrijednost iznosa
- akcija: u odgovarajuće stanje pribrojiti razliku između nove i stare vrijednosti atributa iznos

```
CREATE FUNCTION sync_racun_update() RETURNS trigger AS
$$
BEGIN
    UPDATE racun
        SET stanje = stanje + (NEW.iznos - OLD.iznos)
        WHERE brRac = OLD.brRac;
    RETURN NEW; -- obavezno!
END;
$$ LANGUAGE plpgsql;
CREATE TRIGGER upd_sync_racun
    BEFORE UPDATE OF iznos ON uplataispata
    FOR EACH ROW
    WHEN (OLD.iznos <> NEW.iznos)
    EXECUTE PROCEDURE sync_racun_update();
```

Kod UPDATE su definirane i NEW i OLD.

- UPDATE OF column_name1 [, column_name2 ...]
- UPDATE OF tblName – izmjena bilo kojeg atributa

Naredba CREATE TRIGGER

- oblik naredbe za kreiranje okidača propisan je SQL standardom, ali SUBP koriste uglavnom vlastite inačice
- jedna od važnijih mogućnosti koje su na raspolaganju pri definiciji okidača:
 1. moguće je specificirati da li se akcije navedene u okidaču obavljaju:
 - a) FOR EACH ROW: po jednom za svaku n-torku na koju je djelovala operacija koja je aktivirala okidač (operacija koja je uzrokovala događaj)
 - b) FOR EACH STATEMENT: jednom, za naredbu koja je aktivirala okidač (npr. INSERT)
 2. Prije ili poslije:
 - a) AFTER INSERT, AFTER UPDATE, AFTER DELETE
 - b) BEFORE INSERT, BEFORE UPDATE, BEFORE DELETE
- Ne implementiraju svi SUBP-ovi sve kombinacije (PostgreSQL da!)

PgSQL CREATE TRIGGER pojednostavljena sintaksa

```
CREATE TRIGGER name { BEFORE | AFTER | INSTEAD OF }  
  { event [ OR ... ] }  
ON table_name  
  [ FOR [ EACH ] { ROW | STATEMENT } ]  
  [ WHEN ( condition ) ]  
EXECUTE PROCEDURE f_name ( arguments )
```

Events:

- INSERT
- DELETE
- UPDATE

PgSQL CREATE TRIGGER

- Okidač se definira pomoću funkcije koja se formalno prijavljuje bez argumenata (iako ih je moguće „poslati”)
- Funkcija ima na raspolaganju niz sistemskih varijabli koje dodatno opisuju kontekst okidača (nisu sve prikazane):

Varijabla	Opis
NEW	Nova vrijednost ntorke, ta ntorke će biti u konačnici zapisana (moguće ju je mijenjati). NULL, ako je DELETE ili STATEMENT LVL
OLD	Stara vrijednost ntorke, NULL ako je INSERT ili STATEMENT LVL. (podsjetnik: UPD = INS + DEL)
TG_NAME	Ime okidača
TG_WHEN	BEFORE, AFTER ili INSTEAD OF
TG_LEVEL	ROW ili STATEMENT
TG_OP	INSERT, UPDATE, DELETE ili TRUNCATE

PgSQL CREATE TRIGGER

- Moguće je koristiti jednu funkciju za više okidača
 - Za vježbu probajte napisati jednu funkciju za INSERT, UPDATE i DELETE za tablicu `uplataIsplata` iz prethodnog primjera
- Moguće je definirati više okidača za isti događaj
 - Obavljaju se abecednim redom
- **Povratne vrijednosti okidača:**
 - Za ROW LEVEL AFTER i STATEMENT LEVEL se ignoriraju (ali ti okidači mogu svejedno RAISE EXCEPTION)
 - Inače, okidač mora vratiti NULL ili n-torku; ako vrati:
 - NULL: **otkazuju** se daljnje akcije (redak se npr. ne unosi i potencijalni sljedeći okidači se ne okidaju!)
 - n-torku: daljnje operacije se nastavljaju s tom vrijednošću n-torke (npr. biva unesena i/ili proslijeđena drugim okidačima).

Važno, paziti!

Primjena okidača

- implementacija integritetskih ograničenja
 - okidače treba koristiti onda kada integritetska ograničenja nije moguće opisati na drugi način (PRIMARY KEY, FOREIGN KEY, UNIQUE, CHECK, ...)
 - obavljanjem korektivne akcije koja bazu podataka dovodi u konzistentno stanje (primjer 3)
 - odbijanjem operacije koja narušava integritetsko ograničenje (primjer 4)
- praćenje rada korisnika (primjer 5)
- sustavi obavješćavanja (primjer 6)
- itd.

Primjer 4:

- u relaciji ispit osigurati integritetsko ograničenje prema kojem je promjena ocjena dopuštena samo ako se mijenja na višu ocjenu, npr.
 - nije dopušteno ocjenu izvrstan promijeniti u dobar
 - dopušteno je ocjenu dovoljan promijeniti u vrlo dobar

ispit	matBr	sifPred	datIspr	ocj	sifNast
	100	1001	29.06.2006	1	1111
	100	1001	05.02.2006	3	3333
	101	1002	27.06.2006	2	2222
	102	1001	29.01.2006	1	2222

- očito je da ne postoji korektivna akcija koja bi bazu podataka mogla dovesti u konzistentno stanje nakon što korisnik obavi naredbu:

```
UPDATE ispit SET ocjena = 1
WHERE ocjena = 2;
```

- jedini način na koji se može osigurati navedeno integritetsko ograničenje jest: odbiti izvršavanje takve naredbe

Primjer 4 (nastavak):

```
CREATE FUNCTION trg_raise_exception() RETURNS trigger AS
$$
BEGIN
    RAISE EXCEPTION 'Error: ocjena se ne smije smanjiti!';
    RETURN NULL;
END;
$$ LANGUAGE plpgsql;
CREATE TRIGGER upd_ocjena_ispit
BEFORE UPDATE OF ocjena ON ispit FOR EACH ROW
WHEN (NEW.ocjena < OLD.ocjena)
EXECUTE PROCEDURE trg_raise_exception();
```

- što se događa pri izvršavanju naredbe
- nakon promjene prve n-torke, akcije iz okidača se neće obaviti jer uvjet za obavljanje akcije nije ispunjen
- nakon promjene druge n-torke, aktivirat će se akcija iz okidača
 - poziva se procedura
 - procedura signalizira pogrešku
 - budući da se naredba mora obaviti u cijelosti ili uopće ne, sustav poništava i promjenu prve n-torke, a korisniku prikazuje opis pogreške

```
UPDATE ispit SET ocj = 2
WHERE matBr = 100;
```

Primjer 5 (*auditing*):

- Pretpostavi li se da je izmjena podataka u relaciji **emp** naročito osjetljiva operacija - potrebno je pratiti rad korisnika (*audit trail*)
 - Primjer preuzeti iz PostgreSQL dokumentacije
<https://www.postgresql.org/docs/9.6/static/plpgsql-trigger.html>

```
CREATE TABLE emp (  
    empname          text NOT NULL,  
    salary           integer  
);  
  
CREATE TABLE emp_audit(  
    operation        char(1)  NOT NULL,  
    stamp            timestamp NOT NULL,  
    userid           text     NOT NULL,  
    empname          text     NOT NULL,  
    salary integer  
);
```

Primjer 5 (nastavak)

```
CREATE OR REPLACE FUNCTION process_emp_audit() RETURNS TRIGGER AS
$$
BEGIN
    IF (TG_OP = 'DELETE') THEN
        INSERT INTO emp_audit
        SELECT 'D', CURRENT_TIMESTAMP, current_user, OLD.*;
        RETURN OLD;
    ELSIF (TG_OP = 'UPDATE') THEN
        INSERT INTO emp_audit
        SELECT 'U', CURRENT_TIMESTAMP, current_user, NEW.*;
        RETURN NEW;
    ELSIF (TG_OP = 'INSERT') THEN
        INSERT INTO emp_audit
        SELECT 'I', CURRENT_TIMESTAMP, current_user, NEW.*;
        RETURN NEW;
    END IF;
    RETURN NULL; -- result is ignored since this is an AFTER trigger
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER emp_audit
AFTER INSERT OR UPDATE OR DELETE ON emp
FOR EACH ROW EXECUTE PROCEDURE process_emp_audit();
```

Primjer 5 (nastavak):

- što se dešava obavljanjem naredbe

postgres

```
UPDATE emp
SET salary = 1.1 * salary;
```

	empname text	salary integer
1	Boris	1000
2	Tibor	2000

- osim promjene u relaciji emp, u relaciju emp_audit bit će dodane dvije n-torke:

emp

	empname text	salary integer
1	Boris	1100
2	Tibor	2200

emp_audit

	operation character(1)	stamp timestamp without time zone	userid text	empname text	salary integer
1	U	03.06.2018 17:20:47.3394	postgres	Boris	1100
2	U	03.06.2018 17:20:47.3394	postgres	Tibor	2200

Primjer 6 (izmjena ntorke):

- Želimo svakom zapisu održavati vrijeme zadnje izmjene i korisnika koji je obavio izmjenu (nalik *user/date modified* u datotečnom sustavu)
 - Primjer preuzeti iz PostgreSQL dokumentacije
<https://www.postgresql.org/docs/9.6/static/plpgsql-trigger.html>

```
-- (svakoj) tablici dodajemo:  
-- last_timestamp i last_user attribute  
  
CREATE TABLE emp (  
    empname          TEXT          NOT NULL,  
    salary            DECIMAL(10, 2) NOT NULL,  
    last_timestamp    TIMESTAMP,  
    last_user         TEXT  
);
```

Primjer 6 (izmjena ntorke, nastavak)

```
CREATE FUNCTION emp_log () RETURNS trigger
AS $emp_log$
BEGIN
    -- Usput provjeravamo i je li plaća pozitivna
    -- (mogli smo ovo implementirati i u posebnom okidaču ili kao CHECK)
    IF NEW.salary < 0 THEN
        RAISE EXCEPTION '% cannot have a negative salary', NEW.empname;
    END IF;

    -- Postavljamo varijable koje pratimo:
    NEW.last_timestamp := CURRENT_TIMESTAMP;
    NEW.last_user := CURRENT_USER;

    RETURN NEW; -- OBAVEZNO vratiti izmijenjeni redak!!
END;
$emp_log$ LANGUAGE plpgsql;

CREATE TRIGGER emp_log_trigg BEFORE INSERT OR UPDATE ON emp
FOR EACH ROW EXECUTE PROCEDURE emp_log();
```


Primjer 7:

- postoji pohranjena procedura `saljiPostu(adresa, tekst)`
- u relaciji `artikl` nalaze se podaci o artiklima na skladištu. Za svaki artikl prati se trenutno stanje (količina) artikla
- kada stanje artikla padne ispod optimalne količine, potrebno je na e-mail adresu djelatnika zaduženog za nabavu tog artikla poslati poruku

artikl	sifArt	stanje	optKol	adresaZaduzenog
	1001	250	150	pero@tvrtka.hr
	1002	400	200	joza@tvrtka.hr
	1003	450	350	jura@tvrtka.hr

```
CREATE TRIGGER updArtikl
BEFORE UPDATE OF stanje ON artikl
FOR EACH ROW
WHEN (OLD.stanje >= NEW.optKol AND
      NEW.stanje < NEW.optKol)
(EXECUTE PROCEDURE saljiPostu(OLD.adresaZaduzenog
                              , 'Nabavi artikl: ' || OLD.sifArt));
```

Primjer 7 (nastavak):

artikl	sifArt	stanje	optKol	adresaZaduzenog
	1001	250	150	pero@tvrka.hr
	1002	400	200	joza@tvrka.hr
	1003	450	350	jura@tvrka.hr

- rezultat obavljanja naredbe

```
UPDATE artikl  
SET stanje = stanje - 150;
```



artikl	sifArt	stanje	optKol	adresaZaduzenog
	1001	100	150	pero@tvrka.hr
	1002	250	200	joza@tvrka.hr
	1003	300	350	jura@tvrka.hr

- + dvije poruke

pero@tvrka.hr: Nabavi artikl: 1001

jura@tvrka.hr: Nabavi artikl: 1003

- ako se nakon toga obavi naredba

```
UPDATE artikl  
SET stanje = stanje - 100;
```



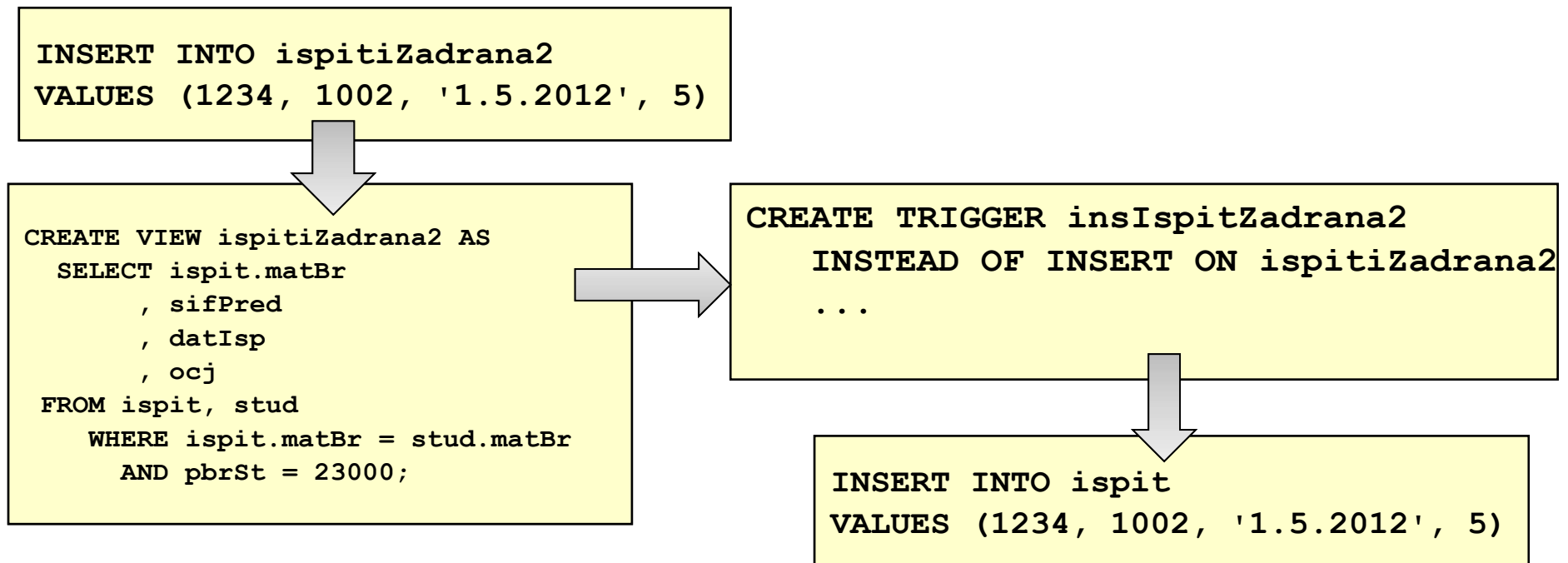
artikl	sifArt	stanje	optKol	adresaZaduzenog
	1001	0	150	pero@tvrka.hr
	1002	150	200	joza@tvrka.hr
	1003	200	350	jura@tvrka.hr

- + poruka

joza@tvrka.hr: Nabavi artikl: 1002


Za one koji žele znati više ... INSTEAD OF okidači

- Mehanizam kojim se svaka virtualna relacija može učiniti izmjenjivom
 - definira se koje operacije treba napraviti u temeljnim relacijama umjesto zadanih operacija nad virtualnom relacijom
- Nije u skladu sa SQL standardom, ali je podržan u mnogim sustavima
- Npr. za neizmjenjivu virtualnu relaciju `ispitiZadrana2`:



INSTEAD OF okidači

- Djelovanje okidača mora biti u skladu s definicijom virtualne relacije
- Npr. je li prije unosa n-torke s prethodne strane obavljena provjera da li je student 1234 iz Zadra?



```
INSERT INTO ispit  
VALUES (1234, 1002, '1.5.2012', 5)
```

- Više nema „WITH CHECK OPTION“, odgovornost je na programeru

PostgreSQL primjer, INSTEAD OF okidači

```
CREATE OR REPLACE FUNCTION update_ispiti_zadrana() RETURNS TRIGGER AS $$
BEGIN
    -- Prvo za sve operacije provjerimo radi li se o studentu iz Zadra
    IF (SELECT COUNT(*) FROM stud WHERE mbr = OLD.mbr AND pbrStan = 23000) = 0 THEN
        RETURN NULL; -- signal da nije bilo izmjena i da se dodatni okidači otkazuju
    END IF;
    IF (TG_OP = 'DELETE') THEN
        DELETE FROM ispit
            WHERE matBr = OLD.matBr AND sifPred = OLD.sifPred AND datIsp = OLD.datIsp;
        IF NOT FOUND THEN RETURN NULL; END IF;
        RETURN OLD;
    ELSIF (TG_OP = 'UPDATE') THEN
        UPDATE ispit SET ocj = NEW.ocj
            WHERE matBr = OLD.matBr AND sifPred = OLD.sifPred AND datIsp = OLD.datIsp;
        IF NOT FOUND THEN RETURN NULL; END IF;
        RETURN NEW;
    ELSIF (TG_OP = 'INSERT') THEN
        INSERT INTO ispit(matBr, sifPred, datIsp, ocj)
            VALUES(NEW.matBr, NEW.sifPred, NEW.datIsp, NEW.ocj);
        RETURN NEW;
    END IF;
END;
$$ LANGUAGE plpgsql;
```

```
CREATE TRIGGER trig_ispiti_zadrana2
INSTEAD OF INSERT OR UPDATE OR DELETE ON ispitiZadrana2
FOR EACH ROW EXECUTE PROCEDURE update_ispiti_zadrana();
```