Predavanje 3

Sadrzaj

Mehanizmi RSUBP 1
Implementacija skupa šema relacija 1
Implementacija ograničenja šeme BP 2
Implementacija pravila poslovanja koja ne rezultuju ograničenjima šeme BP 2
Deklarativni mehanizmi 3
CREATE DOMAIN 3
SQL klauzula CONSTRAINT 3
Create Assertion 10
Proceduralni mehanizmi 10
Specifikacija trigera 10
Procedure i funkcije 13
Paketi 15

Mehanizmi RSUBP

Namenjeni su za implementaciju skupa šema relacija, ograničenja šeme BP i ostalih pravila poslovanja koja se obično odnose na unapred definisani redosled, obaveze i uslovljenosti izvođenja operacija nad BP, šta to znači ? To znači kad imaš „te i te“ operacije moraš da ih izvršiš u „tom i tom“ redosledu. Recimo možeš da pristupiš operaciji u nekoj proceduri samo ako si uspešno prethodno izvršio neku drugu operaciju. Recimo hoćeš da prijaviš ispit ali nemaš dovoljno novca na računu, i onda se dogodi da hoćeš da prijaviš ispit ali ne možeš a posle dolazi do uslovljenosti da ne možeš posle da upišeš ocenu nad BP. Ovde je cilj da vidimo da možemo i takve vrste pravila poslovanja da ugradimo u šemu baze podataka odnosno u integritentnu komponentu.

Implementacija skupa šema relacija

Klauzula **alter table** uspeva sigurno kada je nasa tabela prazna, ali kada nasa tabela ima podatke, klauzula mora da proveri citavu semu relacije (tabelu). U slucaju da su sve torke zadovoljile ono sto klauzula pokusava da uradi, operacija ce biti uspesna, ukoliko ne, operacija ce biti neuspesna.

- kreiranje, modifikovanje i brisanje korisnički definisanog domena
  - CREATE DOMAIN, ALTER DOMAIN, DROP DOMAIN
- kreiranje, modifikovanje i brisanje složenog tipa podatka
  - CREATE TYPE, DROP TYPE
- kreiranje, modifikovanje i brisanje tabele (šeme relacije)
  - CREATE TABLE, ALTER TABLE, DROP TABLE
- dodavanje, modifikovanje i brisanje kolone tabele (obeležja šeme relacije)
  - ALTER TABLE / ADD, MODIFY, DROP

Implementacija ograničenja šeme BP

- **deklarativni mehanizmi**
  - aktivnosti provere važenja ograničenja i očuvanja konzistentnosti se, većim delom, podrazumevaju
    - SQL klauzula CONSTRAINT
    - CREATE DOMAIN, CREATE ASSERTION
- **proceduralni mehanizmi**
  - aktivnosti provere važenja ograničenja i očuvanja konzistentnosti se, većim delom, programiraju
    - putem proceduralnog jezika
    - CREATE TRIGGER
    - CREATE PROCEDURE, CREATE FUNCTION
    - CREATE PACKAGE, CREATE PACKAGE BODY

Za deklarativne mehanizme cemo koristiti one mehanizme kod kojih se specifikacija i provera ograničenja u najvecem delu podrazumevaju, što znači da mi puno toga pod klauzulom CONSTRAINT smo u stanju da podrazumevamo i da ce sistem svakako raditi i da ne moramo dodatno nista da specificiramo.

Proceduralni mehanizmi se programiraju putem proceduralnog jezika, zato mi na vežbama učimo PL/SQL.

Za implementaciju ograničenja kod deklarativnih mehanizama mi implementiramo na najvećim delom podrazumevan način, dok kod proceduralnih mehanizama mi moramo da primenjujemo i tehnike programiranja u jeziku treće generacije.

Implementacija pravila poslovanja koja ne rezultuju ograničenjima šeme BP

Za njih imamo najčešće samo proceduralne mehanizme u igri, i onda moramo da praktično definišemo uslove i redosled izvođenja operacija koje definišu pravilo poslovanja i to se većim delom programira i za tu svrhu korsitimo opet trigere procedure pakete itd...

Deklarativni mehanizmi

Deklarativni mehanizam – veci deo ponasanja mehanizma se podrazumeva u odnosu na deklaraciju koja je data

Proceduralni mehanizmi – mehanizam kod koga se kompletan ili vecina procesa programira

CREATE DOMAIN

Nacin na koji mozemo kreirati domene SUBP-om onda kada je ova komanda podrzana u istom.

```
CREATE DOMAIN Naziv_domena
  [AS] Tip_podatka[(Dužina)]
  [DEFAULT {Konstanta | Funkcija | NULL}]
  [CHECK (LogičkiIzraz)]
```

SQL klauzula CONSTRAINT

Mozemo dodavati ogranicenja koja sadrzaj tabele mora zadovoljavati.

- deklarativno definisanje ograničenja, različitih tipova
- predstavlja sastavni deo naredbe CREATE TABLE, ili ALTER TABLE
  - CREATE TABLE (..., CONSTRAINT ...)
  - ALTER TABLE ADD CONSTRAINT ...
  - ALTER TABLE DROP CONSTRAINT ...

Ove komande uspevaju onda kada trenutni sadrzaj tabele zadovoljava ta ogranicenja.

```
[CONSTRAINT NazivOgr] SpecifikacijaTipaOgraničenja
  [INITIALLY {DEFERRED | IMMEDIATE}
    [ [NOT] DEFERRABLE] ]
```

Sluzbena rec **CONSTRAINT** sa nazivom ogranicenja kao sto vidimo jeste opciona (kasnije ce biti objasnjeno kada se to ipak mora staviti)

- *SpecifikacijaTipaOgraničenja*
  - NOT NULL- ograničenje nula vrednosti
  - PRIMARY KEY ... - ograničenje primarnog ključa
  - UNIQUE ... - ograničenje jedinstvenosti
  - CHECK ... - ograničenje torke
  - FOREIGN KEY ... - ograničenje stranog ključa

Specifikacija trenutka provere ogranicenja

Kada koristimo mehanizam **constraint** klauzole u create ili alter table naredbi, odnosno kada smo deklarisali ogranicenje, treba znati da u zavisnosti od tipa ogranicenja koji je iskoriscen dalje, ogranicenje

ce biti pokretano za kontrolu onoga sto mi zelimo da postignemo u smislu validnosti podataka na odredjene dogadjaje odnosno na odredjene operacije. Ono sto je pitanje jeste **u kom trenutku** mi znamo da hocemo da se izvrsi provera vazenja naseg ogranicenja ?

Odnosno, pitanje je kada ce on(SUBP) odreagovati sa svojom proverom. Postoje dve opcije. **Momentalno[not deferrable]** okidanje provere vazenja ogranicenja i **odlozeno[deferrable]** okidanje provere vazenja ogranicenja. Klauzula **not deferrable[momentalno]** govori da je ne moguce odloziti kontrolu ogranicenja, tacnije *izvodi se u momentu izvodjenja* ***kriticne operacije***.

Recimo, ako je kriticna operacija insert a imamo **not deferrable**, to znaci da u paketu sa izvodjenjem operacije insert bice sprovedena i kontrola ovog ogranicenja. A u slucaju **deferrable** klauzuole, to znaci da je trenutak provere vazenja ogranicenja **odloziv**.

Klauzola **initally** oznacava kakvo je podrazumevano ponasanje. Ako je nas contstraint **not deferrable**, onda zadavanje klauzole **initally** nema nikakvu ulogu. Samo ako kazemo da je ogranicenje **defferable** onda sa sluzbenom recju **initally** mozemo da kazemo da li je po *defaultu* kontrola ogranicenja **immediate**(**momentalana**) ili **deferred** (**odlozena**). To znaci da je inicijalno tako, a kako ce biti kasnije, mi mozemo da menjamo na nivou transkacije odnosno na nivou sesije.

Deklarativni mehanizam radi tako da **initally immediate** kaze da se ogranicenje kontrolise i dalje **momentalno** na izvodjenje date operacije a **initally deferred** govori da je inicijalno i odlozeno izvodjenje kontrole vazenja ogranicenja i to za *commit.* (To je u deklarativnom mehanizmu tako, samo mozemo da kazemo I da se tako podrazumeva, tj. da se podrazumeva da je odlozeno za *kraj transakcije* odnosno za *komit* tj. za trenutak potvrdjenja transakcije).

Takodje mozemo da definisemo ponasanje i to na *nivou sesije.* Tipa ALTER SESSION ali moze i ALTER SYSTEM pa:

```
SET CONSTRAINT { ListaNazivaOgr | ALL}
                { DEFERRED | IMMEDIATE }
```

Bitno razumeti kod ovoga jeste da ne menjamo da li je **deferrable** ili **not deferrable** nego kakvo ce sad biti konkretno ogranicenje, kao sto smo pomocu klauzole **initially** to postavljali po defaultu, tako i sad definisemo za zeljena ogranicenja.

U nastavku slede tipovi ogranicenja specifikacije, ***not null, primary key, unique, check, foreign key.***

Not Null

Uvek se zadaje na *nivou obelezja* seme relacije (kolone tabele). Forma:

```
– CREATE TABLE
    (...,
        Kolona Tip(Dužina) NOT NULL,
    ...)
```

```
– CREATE TABLE
    (...,
        Kolona Tip(Dužina) CONSTRAINT Naziv NOT NULL,
    ...)
```

U prvoj formi vidimo *skracenu* sintaksu u kojoj je izostavljena sluzbena rec **constraint**. Problem sa prvom je sto se naziv sistemski generise, te posle ako zelimo da obrisemo neko ogranicenje po nazivu, imamo poteskoce dok saznamo koje je to ogranicenje.

Proverava se prilikom svakog pokusaja **insert**, **update** operacije (upisa nove vrednosti obelezja ili modifikacije postojece vrednosti obelezja). U slucaju pokusaja narusavanja ogranicenja, jedina moguca aktivnost sprecavanja operacije je **no action**.

Bitno zapazanje vezano za *sprecavanje operacija* je da *ili celokupna naredba uspeva ili ne*. To znaci da ako 5000 torki zadovolji operaciju a 5001 ne, imamo situaciju da celokupna naredba nije uspesna! Odnosno, ponistavaju se izmene na svih 5001 torki a to se naziva **implicitni rollback**.

Implicitni rollback je rollback samo jedne naredbe(DLM naredbe). Nasa transakcija moze da ima puno drugih naredbi, i to onih kojih je vec realizovala, ako se desio implicitni rollback na neku od njih, mi mozemo obraditi taj *izuzetak* odnosno mozemo odluciti sta cemo uraditi ako se neka naredba nije uspesno izvrsila (rukovanje greskama). Podrazumevano ponasanje pada jedne naredbe implicira da **NECE** pasti transakcija! To znaci da je na programeru da rukuje izuzetcima odnosno greskama.

Primary Key

PRIMARY KEY [(*Lista_obeležja*)]

Zadaje se na *nivou obelezja* seme relacije koje jedino predstavlja *primarni kljuc*, bez navodjenja liste obelezja. Takodje se zadaje na *nivou celokupne seme relacije*(tabele), sa navodjenjem liste obelezja (uobicajeno i opstije resenje). Na *nivou obelezja(kolone)*:

```
» CREATE TABLE
   (...,
        Kolona Tip(Dužina)
                    CONSTRAINT Naziv PRIMARY KEY,
   ...)
```

```
» CREATE TABLE
   (...,
        Kolona Tip(Dužina) PRIMARY KEY,
   ...)
```

Na *nivou seme relacije(tabele)*:

```
» CREATE TABLE
   (lista specifikacija kolona tabele,
    ...
    CONSTRAINT Naziv PRIMARY KEY (Lista_obeležja),
   ...)
```

Znaci, prvo se nabroje sve *specifikacije svih kolona tabele*, a potom *specifikacije constraint klauzola*.

Podrazumeva se, bez posebnog deklarisanja, da je svako obelezje u *lista_oblezja* deklarisano kao **not null** a proverava se prilikom svakog pokusaja upisa nove vrednosti obelezja kljuca (*insert)* ili modifikacije postojece vrednosti obelezja kljuca(*update)*. U slucaju pokusaja narusavanja ogranicenja, jedina moguca aktivnost je sprecavanje operacije (**no action**).

Samo jedan kljuc je moguce ovako definisati, sve ostale moramo na neki od sledecih mehanizma da definisemo.

Pokretanje ove klauzule automatski izaziva *kreiranje "unique" indeksa* (B+ stabla) nad *lista_obelezja*.

Unique

UNIQUE [(*Lista_obeležja*)]

Zadaje se na *nivou obelezja* seme relacije koje jedino zadovoljava *ogranicenje jedinstvenosti* i to bez navodjenja liste obelezja. A takodje se zadaje i *na nivou celokupne seme relacije*(tabele) sa navodjenjem liste obelezja koja zadovoljava svojstvo jedinstvenosti (uobicajeno i opstije resenje). Zadavanje na *nivou obelezja:*

```
» CREATE TABLE
   (...,
    Kolona Tip(Dužina) CONSTRAINT Naziv UNIQUE,
   ...)
```

```
» CREATE TABLE
   (...,
    Kolona Tip(Dužina) UNIQUE,
   ...)
```

Zadavanje na *nivou celokupne seme relacije*:

```
» CREATE TABLE
   (lista specifikacija kolona tabele,
    ...
    CONSTRAINT Naziv UNIQUE (Lista_obeležja),
   ...)
```

Obelezja u *lista_obelezja* mogu biti deklarisana kao **not null** a ne moraju, a kada se deklarisu, moraju se eksplicitno deklarisati. Provera se vrsi prilikom svakog pokusaja upisa nove vrednosti obelezja iz liste (*inserta)* ili modifikacije postojece vrednosti obelezja iz liste (*update*). U slucaju pokusaja narusavanja ogranicenja, jedina moguca aktivnost je sprecavanje operacije (**no action**).

Unique constraint nam takoreci sluzi za definisanje **ekvivalentnih kljuceva** a takodje cemo deklarisati I svaki zaisti unique constrain koji imamo u projektu seme baze podataka. Posto nemamo nista od mehanizama za ostale kljuceve a **primary key** mozemo iskljucivo za primarni kljuc, ostaje nam **unique** za ostale kljuceve.

Redosled obelezja u *lista_obelezja* je bitan. Redosled obelezja u **primary key** klauzoli je vazan jer se nad primary key-em kreira B+ stablo. Mi obezbedjujemo redosled elemenata u svakom cvoru B stabla koji je rastuci ili opadajuci, u ovom slucaju defaultno rastuci, prvo po prvoj koloni, pa zatim po drugoj koloni, pa zatim po trecoj itd. Znaci, redosled je vrlo vazan! Jer ce to biti redosled kako ce se uredjivati sadrzaj svakog cvora, odnosno kako ce elementi biti poredjani u svakom cvoru. Znaci ako imam obelezja (A, B, C) I imamo upit tipa pretrazivanja nad (A, B) ili nad (A), znaci nad levim podnizovima ovog naseg niza (A, B, C) onda treba tako organizovati redosled da sto vise upita koji obuhvataju leve podnizove moze biti realizovan putem B+ stabla, znaci, ako imam upit koji ukljucuje B i C on nece moci najverovatnije da bude realizovan putem ovog B+ stabla (A, B, C) a ako imamo upit nad A,B ili nad A on hoce, a ako imamo ceste upite nad B, C onda bi trebalo stablo organizovati kao (B, C, A). Ukratko, *prvo stavljamo obelezja po kojima ce biti* ***frekventnije pretrage*** ili pak *zahtevniji upiti*. Ideja je da probamo da pokrijemo sto vise upita jednim indeksom, odnosno da smanjimo sto vise broj indeksa a da sa njima pokrijemo sto vise upita, stoga, redosled obelezja je od jako velikog znacaja

[Podsetnik] ***Trazenje*** *je algoritam koji iskljucivo na ulazu prihvata argument koji predstavlja vrednost iz domena kljuca. Rezultat trazenja je pre svega indikator uspesnosti (true,false). Odnosno, ima li torke sa vrednoscu kljuca ili nema, pa zatim mozda sadrzaj tog sloga a na kraju mozda i adresa na kom se trazenje zaustavilo.*

[Podsetnik] ***Pretrazivanje*** *je nalazenje torke koja zadovoljava odredjeni logicki uslov (koji nije povezan sa vrednostima kljuca).*

Check

CHECK (*LogičkiIzraz*)

Zadaje se na nivou *obelezja* seme relacije koje je jedino upotrebljeno u *LogickomIzrazu*, takodje se zadaje i na nivou *celokupne seme* relacije(tabele) i to obavezno, kada *logicki izraz* obuhvata vise od jednog obelezja seme relacije (uobicajeno i opstije resennje).

Ovo proverava **ogranicenje torke** a moze i da sluzi za **ogranicenja domena** (kada nema **create domain** mehanizma, a prakticno ga cesto nema).

Zadavanje na nivou obelezja:

```
» CREATE TABLE
   (...,
      Kolona Tip(Dužina)
            CONSTRAINT Naziv CHECK (LogičkiIzraz),
   ...)
```

```
» CREATE TABLE
   (...,
      Kolona Tip(Dužina) CHECK (LogičkiIzraz),
   ...)
```

Zadavanje na *nivou seme relacije*:

```
» CREATE TABLE
   (lista specifikacija kolona tabele,
    ...
    CONSTRAINT Naziv CHECK (LogičkiIzraz),
   ...)
```

Obelezja upotrebljena u *LogickomIzrazu* mogu biti deklarisana kao **not null** a ne moraju a mogu i pripadati skupu obelezja date seme relacije(uobicajeno) a ne moraju. Dozvoljeno je da se u *LogickomIzrazu* vrse pozivi prethodno isprogramiranih funkcija, a u tim tim funkcijama se mogu koristiti obelezja drugih sema relacija(sto vecina DBMS-ova u praksi ne podrzava).

LogickiIzraz mora biti izracunljiv za svaku torku relacije nad datom semom. Moguce vrednosti izracunatog izraza su: *true, false* ili *null*. Na true i null je recimo dozvoljena modifikacija i upis a na false nije. Na null je dozvoljeno zato sto nije prekrseno (nije false), stoga je dozvoljeno kad je null.

Proverava se prilikom svakog pokusaja upisa nove torke u relaciju ili modifikacije postojece vrednosti obelezja, obuhvacenog zadatim logickim izrazom. U slucaju pokusaja narusavanja ogranicenja, jedina moguca aktivnost je sprecavanje operacije (**no action**).

U *LogickomIzrazu* ne mozemo da imamo tipa Exists(Select * itd..) iz razloga jer taj select koji bi radili nad nekom tabelom je deterministicki (promenljiv u prostoru), sto nas dovodi do toga da bi morali da

proveravamo check constraint I svaki put kad bi se azurirala ta tabela u okviru selecta, sto dovodi do jako puno iscrpljenja sistema (te iz tog razloga, vecina DBMS-ova to zabranjuje).

Foreign Key

```
FOREIGN KEY [(ListaObeležja)]
  REFERENCES NazivRefŠR [(ListaRefObeležja)]
  [MATCH { FULL | PARTIAL }]
  [ON DELETE {NO ACTION | CASCADE |
                          SET DEFAULT | SET NULL} ]
  [ON UPDATE {NO ACTION | CASCADE |
                          SET DEFAULT | SET NULL} ]
```

Posle **references** ide naziv seme relacije koja je **referencirana** + njena lista obelezja. Klauzola **match**, govori kako cemo tretirati foreign key constraint u slucaju da delovi stranog kljuca ili ceo strani kljuc mogu imati *nulla vrednosti*, pa onda imamo tri vrste reference *full, partial i default*.

Foreign key se deklarise u *referencirajucoj* semi relacije(onoj sa leve strane). Zadaje se na nivou *obelezja* kada imamo jedno jedino obelezje koje cini strani kljuc (retko ali moze sintaksno). Ono sto je uobicajeno je da se zadaje na nivou celokupne seme relacije (tabele).

*Naziv**Ref**ŠR* je naziv *referencirane* seme relacije, *ListaObelezja* je lista obelezja stranog kljuca u *referencirajucoj* semi relacije, dok je *Lista**Ref**Obelezja* lista obelezja u referenciranoj semi relacije (moze se izostaviti kada se navodi primarni kljuc).

– zadavanje na nivou obeležja

» CREATE TABLE
(...,
Kolona Tip(Dužina)
CONSTRAINT Naziv FOREIGN KEY
REFERENCES *ŠemaRel(Obeležje)*,
...)

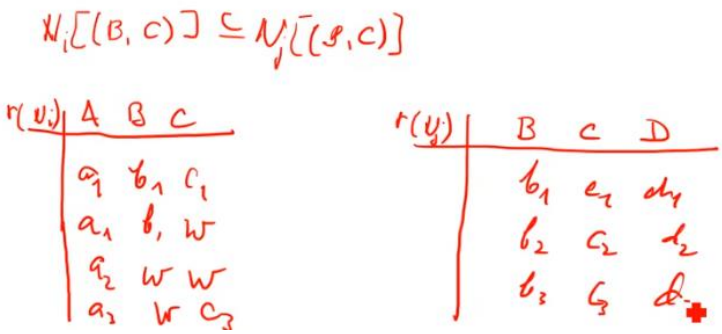
– zadavanje na nivou šeme relacije

» CREATE TABLE
(*lista specifikacija kolona tabele*,
...
CONSTRAINT Naziv FOREIGN KEY (*Lista_obeležja*)
REFERENCES *NazivRefŠR* (*ListaRefObeležja*),
...)

Obelezja u *ListaObelezja* mogu biti deklarisana kao **not null** a ne moraju, obelezja u *Lista**Ref**Obelezja* mogu biti deklarisana kao **not null**, a ne moraju. *ListaObelezja* definise strani kljuc u *referencirajucoj* semi relacije. *Lista**Ref**Obelezja* definise niz obelezja Y koji u referenciranoj semi relacije moze predstavljati *primarni, alternativni kljuc*, *skup obelezja s definisanim ogranicenjem jedinstvenosti*(Unique(Nj,Y)) ili bilo koji niz obelezja(domenski kompatibilan sa nizom obelezja *ListaObelezja*).

Ogranicenje se proverava saglasno opstim pravilima za proveru vazenja *zavisnosti sadrzavanja* i specifikaciji klauzule **MATCH**.

- MATCH PARTIAL
  - delimično referenciranje
  - $(\forall u \in r)(\exists v \in s)(\forall l \in \{1,\ldots, n\})(u[A_l] = \omega \vee u[A_l] = v[B_l])$



Prema onome sto smo do sad naucili, u ovom primeru zadovoljeno je ogranicenje. Vidimo da b1,c1 iz Ni imaju svoj par u Nj, a b1,w ne moramo proveravati jer *nisu not null* obe vrednosti obelezja(pa se podrazumeva da je to zadovoljeno), isto tako ne moramo proveravati ni w,w a ni w,c3 (jer nisu sva obelezja not null). I to je po defaultu.

A u slucaju **match partial** moramo proveriti b1,w ali i w,c3 dok w,w ne moramo proveriti. **Match partial** znaci da ukoliko je deo stranog kljuca sa nula vrednostima, proverava se ostatak na referenciranje. U gornjem primeru vidimo da **default**ni nije narusen dok bi **partial** bio narusen ako ne bi bilo poslednje torke u Nj. Odnosno, ako je ne nula vrednost, za nju mora da postoji torka u drugoj tabeli, ako je nema, onda je narusen **match partial**, a ako ima takve torke onda je sve uredu.

- MATCH FULL
  - potpuno referenciranje
  - $(\forall u \in r)(u[X] = \omega \vee (u[X] \neq \omega \wedge (\exists v \in s)(u[X] = v[Y])))$

Match full partial uglavnom nisu ni implementirani kod DBMS-ova (ali da znamo da postoje generalno). U **potpunom referenciranju** nije dozvoljeno uopste ni da postoje torke tipa druge i cetvrte(u gornjem primeru).

Posto DBMS-ovi uglavnom ne podrzavaju **match full partial** mi mozemo sa **check constraint**om da to obezbedimo. Tako sto cemo reci da je ceo strani kljuc sa nula vrednosti ili je ceo strani kljuc bez nula vrednosti. Napravimo takav chech constraint sa takvim xorom za svaki atribut, ili su svi atributi not null ili su svi atributi null (mozemo preko xor logicke operacije). A za **match partial** nema brzog znalazenja kao sa ovaj match fulll, ali bi moglo preko trigera proceduralnim mehanizmima.

Ogranicenje se proverava prilikom svakog pokusaja upisa nove torke u *referencirajucu* relaciju, *modifikacije vrednosti stranog kljuca* (datog putem *ListaObelezja*), *brisanja* postojece torke iz *referencirane* relacije, *modifikacije vrednosti obelezja*(sadrzanih u *ListaRefObelezja)*. Kod pokusaja upisa nove torke u *referencirajucu* relaciju i modifikacije *vrednosti stranog kljuca* (datog putem *ListaObelezja*) jedina moguca aktivnost ocuvanja konzistentnosti je **no action**. Medjutim, kod brisanja torke iz *referencirane* relacije, imamo klauzolu **on delete{ no action | cascade | set default | set null }** pri cemu je podrazumevana aktivnost **no action**. Kada zelimo modifikovati *vrednosti obelezja*, sadrzanih u *Lista**Ref**Obelezja* imamo klauzulu **on update { no action | cascade | set default | set null }** pri cemu je

podrazumevana aktivnost **no action**. Ono sto nece podrzati DBMS-ovi, mi mozemo nadoknaditi **proceduralnim mehanizmima**.

[***Napomena***]: Umesto sluzbene reci **no action** u vecini DBMS-ova imamo sluzbenu rec **RESTRICT.**

Create Assertion

```
CREATE ASSERTION Naziv_ograničenja
CHECK (LogičkiIzraz)
```

Dobijamo kreiranje *viserelacionog, medjurelacionog* ogranicenja (pri cemu SUBP-ovi obicno ne podrzavaju ovaj mehanizam).

Proceduralni mehanizmi

Imamo **okidace(trigere)**, **procedure i funkcije** baze podataka kao i **pakete** baze podataka. Ovi proceduralni mehanizmi nam omogucavaju da isprogramiramo ponasanje mehanizma onako kako mi to zelimo. Sada vise *ne postoji standardni jezik*. SUBP-ovi imaju svoje jezike i koncepte (koji su slicni), tipa Oracle ima **PL/SQL** a Microsoft SQL server ima **T-SQL**, oni jesu slicni ali nisu skroz isti.

Predavanje 4

Specifikacija trigera

Triger je mehanizam koji se aktivira samostalno, mimo zelje korisnika. **Specifikaciju** trigera cine **oblast aktiviranja**(tabela(ili pogled) nad kojom se definise)**, specifikacija operacija** koje ga pokrecu(insert, update, delete)**, uslovi** pod kojima se triger **aktivira**, **vreme aktiviranja**(neposredno pre ili posle same operacije), **frekvencija aktiviranja**(jednom za celu operaciju ili za svaku torku koja je predmet operacije, pojedinacno), **aktivnost(procedura)** koju triger treba da realizuje.

Sintaksa za definisanje trigera (PL/SQL)

Sintaksu treba znati napamet (jako cesto na usmenom).

```
CREATE [OR REPLACE] TRIGGER NazivTrigera
  BEFORE | AFTER | INSTEAD OF
    INSERT | DELETE | UPDATE [OF ListaObeležja]
    [ OR INSERT | DELETE | UPDATE [ OF ListaObeležja ] ... ]
  ON NazivTabele
  [ FOR EACH ROW [WHEN (LogičkiUslovPokretanjaTrigera)]
  [ REFERENCING OLD AS NazivOld NEW AS NazivNew ] ]
  [ DECLARE
    Deklarativni deo - lokalne deklaracije
  ]
  BEGIN
    Izvršni deo - proceduralni deo, specifikacija aktivnosti
  [ EXCEPTION
    Deo za obradu izuzetaka ]
  END NazivTrigera
```

Opcija **before|after** nam kaze da li ce prvo biti izveden operacija pa triger opcija je **after**, il ce prvo biti izveden triger pa onda operacija naredba je **before**.
Opcija **instead of** je rezervisana iskljucivo za poglede. Umesto da pustimo da parser sql naredbi razresi kako ce nad datim pogledom sprovesti naredbu *insert, update, delete* mi definisemo da se umesto neke od tih naredbe okine ovaj triger, koji ce onda sam sprovesti ono sto u telu trigera bude pisalo. Ako ne zadamo *ListaObelezja* onda se to odnosi na citavu torku (odnosno na bilo koje obelezje iz te torke). Jedan insert moze da se koristi za vise dogadja, zato i imamo opciju *OR.*

Ako izostavimo **for each row** klauzulu, onda su to tkzv **statement level trigeri** koji se okidaju *jednom za naredbu*, ukoliko uvodimo tu klauzulu onda se nas triger naziva **low lever triger** i okida se onoliko puta *koliko je torki bilo predmet nase* insert, update, delete *operacije*. Klauzula **when** je podopcija od **for each row** i nju mozemo definisati samo kad imamo **low level** trigere i onda mozemo dodati logicki uslov pokretanja trigera. **Referencing** opcija ima smisla samo u **for each row** trigerima (iako to sintaksno nije tako predstavljeno). Kod **for each row** trigera ako je kontekst trigera tekuci red, imamo mogucnost preuzimanja vrednosti tog tekuceg reda a cak mozemo da menjamo vrednosti tog tekuceg reda. To ima smisla raditi u before trigerima jer pre izvodjenja naredbe mi mozemo da zadamo neke vrednosti koje mi hocemo.

U PL/SQL se tim vrednostima pristupa (u proceduralnom delu trigera) tako sto kazemo **old.naziv_kolone** (preuzima vrednost obelezja/kolone *pre* operacije(insert, update, delete) nad torkom) i **new.naziv_kolone** (preuzima vrednost obelezja/kolone *posle* operacije).

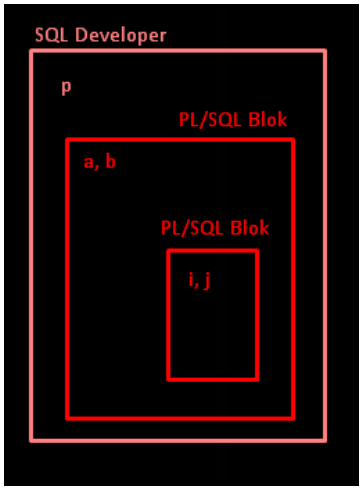
Referencing klauzula nam sluzi da redefnisemo naziv **old**-a i **new**-a ako tu uopste ima potrebe (jako retko se koristi).

Od **declare** pa do **end**-a imamo ugradjeni PL/SQL blok(tekst koji je ugradjen u jednu SQL naredbu). Voditi racuna da od **create** do **end**-a mi imamo jednu SQL naredbu a time takodje definisemo u data dictionary

objekat tipa *triger*. Napomena je da SQL engine odredjuje citavu SQL naredbu (znaci od **create** pa do **end**), ali kada dodje do **declare** dela, onda taj deo preuzima PL/SQL engine.

Uvek se mora znati sta je **context** jedne PL/SQL jedinice. Mi smo radili oblik **anonimnog pl/sql bloka** sa **declare begin** i **end**, kontekst izvrsenja jednog anonimnog pl/sql bloka je citav SQL Developer (ili neki drugi tipa SQL+, u zavisnosti sa cim radimo). Za spoljni kontekst je taj anonimni pl/sql blok kao bilo koja sql naredba (tako ga i vidi).

Promenljivoj koja se nalzi u kontekstu engin-a (na slici ispod, promenljiva p) se naziva **host promenljiva** i njoj se pristupa unutar PL/SQL Bloka tako sto se navede **:naziv_promenljive**(u primeru dole, rekli bi :p, odnosno ako zelimo da stavimo vrednost promenljive na 5, to bi ucinili sa :p=5).



Slika 1. Primer host promenljive

Kod ostalih promenljivih to nije potrebno, odnosno u bloku gde se nalaze promenljive i,j jednostavno mozemo da kazemo a=5 ili b=5, ali moramo dodati dvotacku ako zelimo da pristupimo **host** promenljivoj iz konteksta engina.

Znaci, PL/SQL blok moze biti deo trigera, voditi racuna da se u tom trenutku pl/sql blok vise ne izvrsava kao anononimni pl/sql blok nego kao pl/sql blok ciji je kontekst njegov triger u kom se nalazi (posledicno tabela). Ako je to **each row triger** onda je to prakticno i onaj red nad kojim se taj triger trenutno okida.

Aktiviranje trigera

Trigeri se aktiviraju **automatski** mimo volje korisnika, a oni se ponasaju kao produzeci *insert, update, delete* naredbi i to onda kada su zadovoljeni specificirani uslovi za aktiviranje trigera(klauzula **when**).

[**SIGURNO PITANJE**] Ako imamo recimo naredbu update i BEFORE|AFTER STL|RWL CONSTRAINT, redosled izvrsavanja je bitan i mora da se zna napamet!! Redosled je sledeci:

1. before statement level
2. before row level (ovde i u prethodnom koraku mozemo da 'izokidamo' neke provere, cak mozemo da uticemo i na vrednosti koje ce biti updateovane

3. update …
4. after row level (sluzi da vidimo da li je sve uredu, u after trigerima ne mozemo vise menjati ono sto je update naredba promenila, dok u before mozemo menjati tekuce redove)
5. ***constraint***
6. after statement level (ovde jos imamo sansu da ponistimo celu sekvencu, *sekvenca* je sve oko naredbe, znaci sve od tacke 1 do 6 predstavlja sekvencu, koja u celosti uspeva ili ne)

Ako padne bilo koja od stavki, desava se *implicitni rollback* i pada cela naredba(sve sto je uradjeno u svakoj od tacaka).

Procedure i funkcije

Predstavljaju proceduralno specificirane programe, definisane na nivou SUBP-a i pozivaju se po potrebi (ne aktiviraju se automatski). Cuvaju se u izvornom i kompajliranom, optimizovanom obliku a pozivaju se iz trigera, ili direktno iz korisnickih programa(spoljnog konteksta).

Glavna razlika izmedju procedure i funkcije je u tom sto funkcija *vraca vrednosti* a procedura *ne.*

Sintaksa za kreiranje procedure (PL/SQL)

```
CREATE [OR REPLACE] PROCEDURE NazivProcedure
  [ (ListaFormalnihParametara) ]
  AS | IS
    Deklarativni deo - lokalne deklaracije procedure
              - tipovi podataka
              - konstante i promenljive
              - procedure i funkcije
              - kursorska područja
              - izuzeci
  BEGIN
    Izvršni deo - proceduralni deo, specifikacija aktivnosti
  [ EXCEPTION
    Deo za obradu izuzetaka ]
  END NazivProcedure
```

Moramo da znamo da je ovo SQL naredba. Klauzule **is, as** se razlikuju samo od standrda do standrda (neko koristi **is** a neko **as**). U deklariativnom delu pocinje tekst PL/SQL naredbe ali nema sluzbene reci *declare*(jer se podrazumeva).

Sintaksa za kreiranje funkcije (PL/SQL)

```
CREATE [OR REPLACE] FUNCTION NazivFunkcije
  [ (ListaFormalnihParametara) ]
  RETURN TipPodatkaPovratneVrednostiFunkcije
  AS | IS
    Deklarativni deo - lokalne deklaracije funkcije
  BEGIN
    Izvršni deo - proceduralni deo, specifikacija aktivnosti
      /* Zahteva pojavljivanje naredbe oblika RETURN Izraz */
  [ EXCEPTION
    Deo za obradu izuzetaka ]
  END NazivFunkcije
```

Ovo je takodje SQL naredba u kojoj je dalje kao tekst ugradjen tekst kao PL/SQL blok koji zapocinje posle sluzbene reci **as | is**. Funkcije za razliku od procedura mogu da se koriste u *bilo kom izrazu*, zato sto imaju povratnu vrednost. Tada funkcija ima ulogu jednog operanda.

ListaFormalnihParametara

- *Parametar* [, *Parametar*...]

Parametar

- *NazivParametra* [IN | OUT | IN OUT] *TipParametra*
  - IN - ulazni parametar
  - OUT - izlazni parametar
  - IN OUT - ulazno-izlazni parametar
- *TipParametra*
  - predefinisani, ili
  - prethodno deklarisan (korisnički definisan)

Napomena: Kod funkcija kao i kod procedura mi možemo definisati i IN, OUT, IN OUT parametre. Kod funkcije je bitno da definišemo samo isključivo IN parametre! Ako nam trebaju OUT ili IN OUT parametri onda treba da napravimo proceduru. Bitno je da funkcija nema bočne efekte!

Izuzetak

Predstavlja dogadjaj, cije nastupanje izaziva prekid normalnog toka izvrsenja programa (definisanog proceduralnim delom funkcije ili procedure) . Vrste:

```
predefinisani          - ugrađen u definiciju jezika
korisnički definisani  - EXCEPTION NazivIzuzetka
korisnički definisani, povezan sa greškom SUBP
  » EXCEPTION NazivIzuzetka
  » PRAGMA EXCEPTION_INIT (NazivIzuzetka, -KodGreške)
```

Obrada izuzetaka ide na sledeci nacin:

```
WHEN {NazivIzuzetka [OR NazivIzuzetka]... | OTHERS}
    THEN Procedura za obradu izuzetka
```

Kada predjemo u deo kod sluzbene reci **then** to znaci da se desio izuzetak i da cemo uraditi ono sto se nalazi u tom delu. U tom trenutku smo izasli iz glavnog toka programa i presli u taj alternativni tok. Bitno je napomenuti da sada vise ***nema vracanja*** na glavni tok programa! Voditi racuna da **when others** stavljamo na kraj (jer ono ce biti zadovoljeno za svaki izuzetak).

Sa sluzbenom recju **RAISE** *izuzetak* mi okidamo izuzetak.

Paketi

Predstavljaju biblioteke deklaracija i programa definisanih na nivou SUBP-a a koji se cuvaju u izvornom I kompajliranom, optimizovanom obliku. Oni sadrze javni i privatni deo(koncept *ucaurenja*). Koristi se za tematsko organizovanje softvera na nivou SUBP-a. Takodje, podrzavaju perzistenciju podataka na nivou sesije i preklapanje(overloading) procedura i funkcija.

Sintaksa za kreiranje paketa i tela paketa

```
CREATE [OR REPLACE] PACKAGE NazivPaketa
  AS | IS
    Deklarativni deo – javne deklaracije paketa
            - tipovi podataka
            - konstante i promenljive
            - zaglavlja procedura i funkcija
            - kursorska područja
            - izuzeci
END NazivPaketa
```

```
CREATE [OR REPLACE] PACKAGE BODY NazivPaketa
  AS | IS
    Deklarativni deo – privatne deklaracije paketa
            - tipovi podataka
            - konstante i promenljive
            - lokalne procedure i funkcije
            - razrada javnih procedura i funkcija
            - kursorska područja
            - izuzeci
  [ BEGIN
    Deo za inicijalizaciju - proceduralni, specifikacija aktivnosti
  ]
END NazivPaketa
```

Kreiranje tela paketa je opciono. Ako imamo proceduru koju definisemo iznad paketa, za nju je vidljivo samo ono sto se nalazi unutar javnog paketa. U javnom delu paketa idu potpisi (tipa *procedure p;)* a u privatnom idu implementacije.