UNIVERZITET U BEOGRADU
FAKULTET ORGANIZACIONIH NAUKA

Skripta iz predmeta

- Baze podataka 2 -

Autori:

prof. dr Zoran Marjanović
prof. dr Nenad Aničić
doc. dr Slađan Babarogić
mr Marija Janković
Miroslav Ljubičić
Srđa Bjeladinović
Elena Milovanović

Beograd, 2013.

Sadržaj

1. Denormalizacija

Denormalizacija je postupak poboljšanja performansi baze podataka dodavanjem redundantnih podataka, uz narušavanje normalnih formi. Drugačije rečeno, denormalizacija predstavlja postupak "spuštanja" relacija iz više u nižu normalnu formu.

Denormalizacija spada u fizičko projektovanje baze podataka na logičkom nivou, pri čemu se kao polazna tačka uzima potpuno normalizovani konceptualni model kreiran u fazi logičkog projektovanja. Nad tim, potpuno normalizovanim modelom, potom se vrši denormalizacija, pri čemu se narušavaju normalne forme, uvodi redundansa podataka i narušava integritet podataka. Zbog uvedene redundanse i narušavanja integriteta podataka, neophodno je kroz aplikacioni kôd kompenzovati posledice denormalizacije, tj. omogućiti očuvanje integriteta i konzistentnosti redundantnih podataka (npr. upotrebom trigera nad denormalizovanim tabelama u bazi podataka).

U nastavku su dati primeri denormalizacije 2NF i 3NF, ilustrovani upotrebom PMOV – a i relacionog modela. Za svaki primer, tj. za svaku denormalizaciju, neophodno je implementirati i odgovarajuće trigere kojima se održava integritet denormalizovanih i redundantnih podataka. Pre same implementacije trigera, potrebno je dati njihovu specifikaciju u vidu tabele. U okviru **tabele specifikacije trigera** navode se svi neophodni trigeri i daje se kratak opis akcija koje ti trigeri izvršavaju kao odgovore na DML operacije nad tabelama koje su učestvovale u denormalizaciji. Opšta struktura tabele specifikacije trigera je:

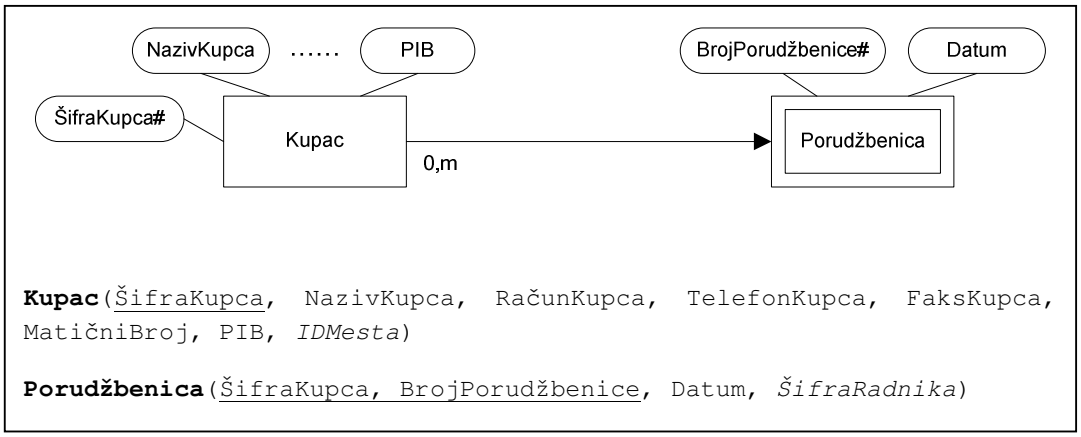
Tabela	Tip trigera	Kolona	Potreban	Šta treba da uradi?
Naziv_tabele1	Insert		DA	*Opis_logike_akcije (2-3 rečenice)*
	Update		NE	
	Delete	*Naziv_kolone*	NE	
Naziv_tabele2	Insert		NE	
	Update	*Naziv_kolone1*	DA	*Opis_logike_akcije*
		Naziv_kolone2	NE	
		Naziv_kolone3	DA	*Opis_logike_akcije*
	Delete		DA	*Opis_logike_akcije*

Tabela 1 - Opšta struktura tabele specifikacije trigera

Kao što se može videti, u prvoj koloni navodi se naziv tabele nad kojom se definiše triger. Potom se za tri osnovne DML naredbe i, ukoliko je potrebno, konkretne kolone u okviru tabele, daje opis akcije koju izvršava triger kao odgovor na odgovarajuću DML naredbu nad tabelom, odnosno, kolonom tabele.

1.1. Primer denormalizacije 2NF

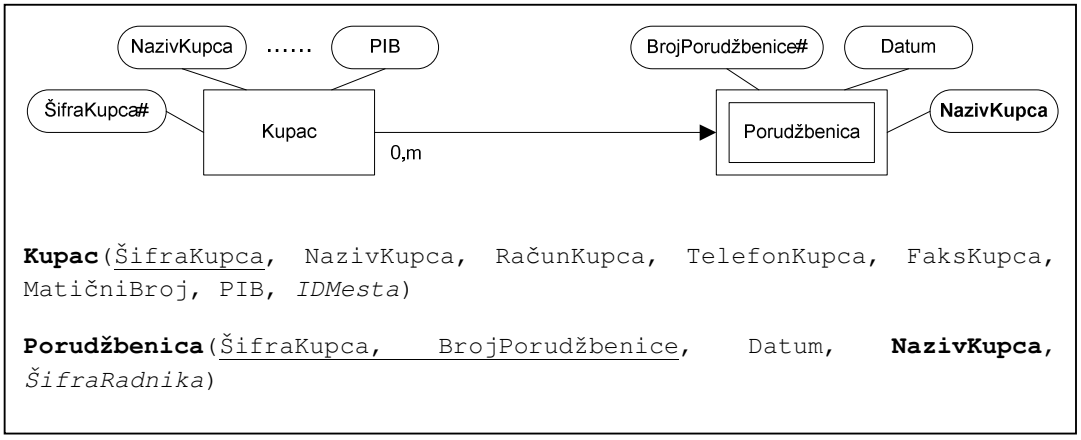
Pretpostavimo da je dat sledeći **normalizovani model**[1]:



Kupac(ŠifraKupca, NazivKupca, RačunKupca, TelefonKupca, FaksKupca, MatičniBroj, PIB, *IDMesta*)

Porudžbenica(ŠifraKupca, BrojPorudžbenice, Datum, *ŠifraRadnika*)

Očigledno je da će prilikom rada sa porudžbenicom, pored šifre kupca, veoma često biti neophodno prikazivanje i naziva kupca. Ukoliko bi model ostao normalizovan, prikazivanje naziva kupca u okviru porudžbenice bi zahtevalo spajanje tabela (JOIN klauzula) u okviru odgovarajućih SQL upita, što je vremenski zahtevna operacija. Kako bi se izbegla konstantna upotreba JOIN klauzule, moguće je izvršiti denormalizaciju relacija, pri čemu se atribut NazivKupca relacije Kupac „prebacuje" i u relaciju Porudžbenica (tj. u entitet, ukoliko se posmatra PMOV). **Denormalizovani model** bi sada izgledao:



Kupac(ŠifraKupca, NazivKupca, RačunKupca, TelefonKupca, FaksKupca, MatičniBroj, PIB, *IDMesta*)

Porudžbenica(ŠifraKupca, BrojPorudžbenice, Datum, **NazivKupca**, *ŠifraRadnika*)

Kao što se može videti, posle izvršene denormalizacije, relacija Porudžbenica sadrži i redundantan atribut NazivKupca, koji se takođe nalazi i u relaciji Kupac. Očigledno je da uvođenje redundanse može lako narušiti integritet baze podataka, ukoliko redundantni podaci nisu kontrolisani (npr. ukoliko je dozvoljeno da vrednosti atributa NazivKupca u relacijama Kupac i Porudžbenica, za istu šifru kupca, tj. za istog kupca, budu različite). Zbog toga je neophodno

[1] Zbog preglednosti, na PMOV – u neće biti prikazivani svi atributi entiteta niti sve njihove veze.

definisati i implementirati odgovarajuće trigere, koji će održavati integritet i konzistentnost podataka u bazi podataka prilikom izvršavanja DML naredbi koje mogu dovesti do njihovog narušavanja.

U odnosu na prethodni denormalizovani model, akcije koje mogu narušiti integritet podataka su:

- Izmena vrednosti atributa NazivKupca u relaciji Kupac. Neophodno je da se ista izmena izvrši i u relaciji Porudžbenica.
- Dodavanje nove porudžbenice. Ne sme se dozvoliti proizvoljni unos vrednosti atributa NazivKupca u relaciji Porudžbenica, već se njegova vrednost postavlja na osnovu unete vrednosti atributa ŠifraKupca u okviru iste relacije.
- Izmena vrednosti atributa NazivKupca u relaciji Porudžbenica. Ova izmena ne sme biti dozvoljena, obzirom da bi dovela do nekonzistentnosti sa ispravnom vrednošću atributa NazivKupca u relaciji Kupac.
- Izmena vrednosti atributa ŠifraKupca u relaciji Porudžbenica, ukoliko je dozvoljena. Pri izmeni vrednosti atributa ŠifraKupca u relaciji Porudžbenica, odnosno, pri izmeni kupca koji je poslao porudžbenicu, mora se izvršiti ažuriranje vrednosti atributa NazivKupca u istoj relaciji, kako bi odgovarao nazivu novoizabranog kupca.

O očuvanju integriteta baze podataka pri ovim akcijama brinu trigeri čija se specifikacija daje u vidu tabele. **Tabela specifikacije trigera** vezanih za navedeni primer denormalizacije data je ispod. Specifikacija trigera daje se za sve slučajeve u kojima može doći do narušavanja integriteta baze podataka, a pri čemu ti slučajevi nisu već obuhvaćeni referencijalnim integritetom (npr. brisanje kupca za kojeg postoji bar jedna porudžbenica je obuhvaćeno referencijalnim integritetom, pa zbog toga nije navedeno kao akcija koja će biti obuhvaćena trigerom).

Tabela	Tip trigera	Kolona	Potreban	Šta treba da uradi?
Kupac	Insert		NE	
	Update	NazivKupca	DA	Prilikom izmene vrednosti kolone NazivKupca u tabeli Kupac, pokreće se triger koji izmenjenu vrednost ažurira u tabeli Porudžbenica.
	Delete		NE	
Porudžbenica	Insert		DA	Triger ažurira vrednost kolone NazivKupca na osnovu unete vrednosti kolone ŠifraKupca.
	Update	ŠifraKupca	DA	Triger vrši ažuriranje kolone NazivKupca na osnovu izmenjene vrednosti kolone ŠifraKupca.*
		NazivKupca	DA	Zabraniti direktno ažuriranje ove kolone.
	Delete		NE	

** Ukoliko je uopšte dozvoljeno ažuriranje kolone ŠifraKupca*

Vrsta denormalizacije u kojoj se neključni atribut iz referencirane relacije (u ovom primeru, Kupac) dodaje u referencirajuću relaciju (u ovom primeru, Porudžbenica), preko spoljnog ključa u referencirajućoj relaciji ili dela primarnog ključa referencirajuće relacije, naziva se ***Pre – joining*****.** U ovom primeru, izvršena je Pre – joining denormalizacija oslanjanjem na deo primarnog ključa relacije Porudžbenica (atribut ŠifraKupca, koji je istovremeno i primarni ključ relacije Kupac).

Takođe, u ovom slučaju izvršena je denormalizacija Druge normalne forme. Naime, za **normalizovanu** relaciju Porudžbenica važila je sledeća funkcionalna zavisnost:

```
ŠifraKupca, BrojPorudžbenice -> Datum, ŠifraRadnika
```

Posle denormalizacije, za **denormalizovanu** relaciju Porudžbenica važe sledeće funkcionalne zavisnosti:

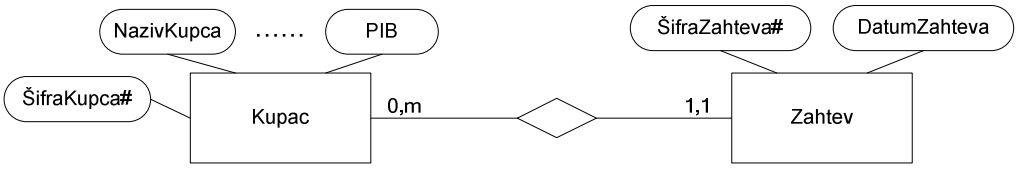
```
ŠifraKupca, BrojPorudžbenice -> Datum, NazivKupca, ŠifraRadnika

ŠifraKupca -> NazivKupca
```

Očigledno je da atribut NazivKupca, dodat u relaciju Porudžbenica tokom denormalizacije, **nepotpuno funkcionalno zavisi** od složenog atributa ŠifraKupca, BrojPorudžbenice, jer je funkcionalno zavisan i od njega i od njegovog dela (ŠifraKupca), što se može videti po drugoj funkcionalnoj zavisnosti koja se javila posle denormalizacije relacije Porudžbenica. Samim tim, relacija Porudžbenica više nije u Drugoj normalnoj formi.

1.2. Primeri denormalizacije 3NF (Pre-joining)

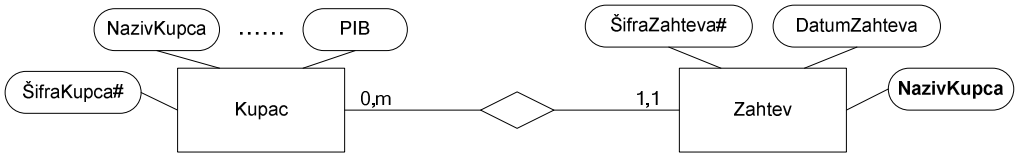
Prethodno navedena, Pre – joining vrsta denormalizacije, može se koristiti i za denormalizaciju Treće normalne forme, kada se redundantni atribut dodaje u relaciju na osnovu postojećeg spoljnog ključa. Pretpostavimo da je dat sledeći **normalizovani model**:



KUPAC(ŠifraKupca, NazivKupca, PIB, DržavaKupca, MestoKupca, AdresaKupca,TelefonKupca,FaxKupca)

ZAHTEV(ŠifraZahteva, DatumZahteva, *ŠifraValute, ŠifraMetodaIsporuke, ŠifraNačinaPlaćanja, ŠifraKupca*)

Kao što se može videti, povezanost relacije Zahtev i relacije Kupac u relacionom modelu iskazana je spoljnim ključem ŠifraKupca u relaciji Zahtev. Slično prethodnom primeru, može se očekivati da će u radu sa zahtevom, pored same ŠifreKupca, često biti potreban i njegov naziv, tj. vrednost atributa NazivKupca iz relacije Kupac, što bi u SQL upitima opet zahtevalo upotrebu JOIN klauzule. Analogno prethodnom primeru, i ovde bi se mogla izvršiti Pre – joining denormalizacija, pri čemu bi se atribut NazivKupca dodao u relaciju Zahtev. **Denormalizovani model** bi sada izgledao:



KUPAC(ŠifraKupca, NazivKupca, PIB, DržavaKupca, MestoKupca, AdresaKupca,TelefonKupca,FaxKupca)

ZAHTEV(ŠifraZahteva, DatumZahteva, *ŠifraValute, ŠifraMetodaIsporuke, ŠifraNačinaPlaćanja, ŠifraKupca,* **NazivKupca**)

Tabela specifikacije trigera za prethodni primer denormalizacije bi bila:

Tabela	Tip trigera	Kolona	Potreban	Šta treba da uradi?
Kupac	Insert		NE	
	Update	NazivKupca	DA	Prilikom izmene vrednosti polja NazivKupca u tabeli Kupac, pokreće se triger koji ažurira vrednost u tabeli Zahtev.
	Delete		NE	
Zahtev	Insert		DA	Triger ažurira vrednost kolone NazivKupca na osnovu unete vrednosti atributa ŠifraKupca.
	Update	ŠifraKupca	DA	Triger vrši ažuriranje kolone NazivKupca na osnovu izmenjene vrednosti kolone ŠirfaKupca.
		NazivKupca	DA	Zabraniti direktno ažuriranje ove kolone.
	Delete		NE	

Iako je i u ovom slučaju korišćena Pre – joining vrsta denormalizacije, za razliku od prethodnog primera, ovde je izvršena denormalizacija Treće normalne forme. Naime, za **normalizovanu** relaciju Zahtev važila je sledeća funkcionalna zavisnost:

```
ŠifraZahteva -> DatumZahteva, ŠifraValute, ŠifraMetodaIsporuke,
ŠifraNačinaPlaćanja, ŠifraKupca
```

Posle denormalizacije, za **denormalizovanu** relaciju Zahtev važe sledeće funkcionalne zavisnosti:

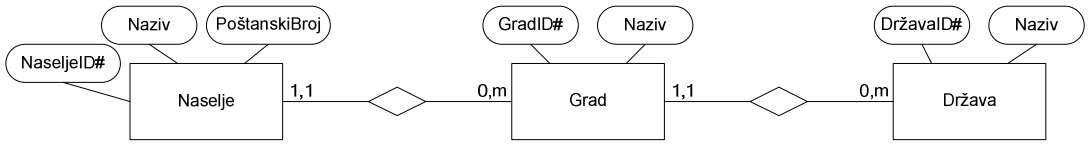
```
ŠifraZahteva -> DatumZahteva, ŠifraValute, ŠifraMetodaIsporuke,
ŠifraNačinaPlaćanja, ŠifraKupca, NazivKupca

ŠifraKupca -> NazivKupca
```

Može se primetiti da sada atribut NazivKupca **tranzitivno funkcionalno zavisi** od atributa ŠifraZahteva, preko atributa ŠifraKupca. Samim tim, relacija Zahtev više nije u Trećoj normalnoj formi.

1.3. Primeri denormalizacije 3NF (Short – circuit keys)

U prethodna dva primera korišćena je Pre – joining vrsta denormalizacije kojom je ilustrovana denormalizacija Druge i Treće normalne forme. Ovde će biti prikazana još jedna vrsta denormalizacije, ***Short – circuit keys*** denormalizacija, kojom se vrši denormalizacija Treće normalne forme. Pretpostavimo da je dat sledeći **normalizovani model**:

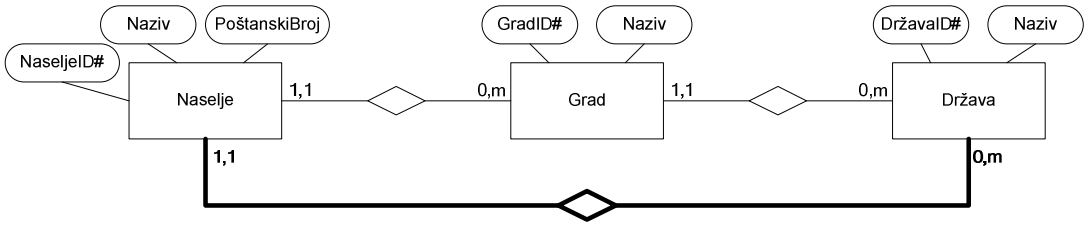


Naselje(NaseljeID, PoštanskiBroj, Naziv, *GradID*)

Grad(GradID, Naziv, *DržavaID*)

Drzava(DržavaID, Naziv)

Može se pretpostaviti da će, prilikom rada se naseljem, često biti potrebni i podaci o državi u kojoj se naselje nalazi. Drugačije rečeno, biće neophodna višestruka upotreba JOIN klauzule, tj. višestruko spajanje tabela, obzirom da bi u SQL upitima bilo neophodno izvršiti spajanje tabela Naselje i Grad, a potom i spajanje sa tabelom Država. Kako bi se broj spajanja tabela smanjio, vrši se Short – circuit keys denormalizacija, koja bi podrazumevala dodavanje spoljnog ključa DržavaID u relaciju Naselje, odnosno, dodavanje veze između Naselja i Države. Takav **denormalizovani model** bi sada izgledao:



Naselje(NaseljeID, PoštanskiBroj, Naziv, *GradID*, ***DržavaID***)

Grad(GradID, Naziv, *DržavaID*)

Drzava(DržavaID, Naziv)

Ovom denormalizacijom broj potrebnih spajanja tabela se smanjuje, obzirom da je sada dovoljno izvršiti spajanje samo tabela Naselje i Država bez učestvovanja tabele Grad, kada je pored podataka o naselju potrebno prikazati i podatke o državi.

Tabela specifikacije trigera za ovaj primer denormalizacije bi bila:

Tabela	Tip trigera	Kolona	Potreban	Šta treba da uradi?
Grad	Insert		NE	
	Update	DrzavaID	DA	Ukoliko postoji Naselje za Grad, ažurira DrzavaID u tabeli Naselje.
	Delete		NE	
Naselje	Insert		DA	U kolonu DrzavaID upisuje odgovarajuću vrednost za unetu vrednost GradID.
	Update	GradID	DA	U kolonu DrzavaID upisuje odgovarajuću vrednost na osnovu nove vrednosti za GradID
		DrzavaID	DA	Sprečava direktnu izmenu.
	Delete		NE	

Sa poslednjeg dijagrama očigledno je da se novom vezom u model uvodi tranzitivnost, odnosno da se narušava Treća normalna forma. Naime, za **normalizovanu** relaciju Naselje važi sledeća funkcionalna zavisnost:

```
NaseljeID -> PoštanskiBroj, Naziv, GradID
```

Posle denormalizacije, za **denormalizovanu** relaciju Naselje važe sledeće funkcionalne zavisnosti:

```
NaseljeID -> PoštanskiBroj, Naziv, GradID, DržavaID

GradID -> DržavaID
```

Može se primetiti da sada atribut DržavaID **tranzitivno funkcionalno zavisi** od atributa NaseljeID, preko atributa GradID. Samim tim, relacija Naselje više nije u Trećoj normalnoj formi.

Reference

Denormalizacija se često obrađuje u različitim knjigama koje se bave bazama podataka. Ispod su navedene neke reference u kojima se, u različitoj meri, objašnjava problem denormalizacije.

1. Whitehorn M., Marklyn B., *„Inside Relational Databases with Examples in Access“*, Springer, 2007.
2. Mullins C., *„Database Administration: The Complete Guide to Practices and Procedures“*, Addison Wesley, 2002.
3. Connoly T., Begg C., *„Database Solutions: A step-by-step guide to building databases“*, Pearson Education Limited, 2004.
4. Lightstone S., Teorey T., Nadeau T., *„Physical Database Design“*, Morgan Kaufmann, 2007.
5. Buxton S., et al., *„Database Design: Know it all“*, Morgan Kaufmann, 2009.
6. Powell G., *„Beginning Database Design“*, Wiley Publishing, 2006.
7. Hoffer J., Prescott M., McFadden F., *„Modern Database Management“*, Pearson Education, 2007.
8. Hoberman S., *„Data Modeler's Workbench“*, John Wiley & Sons, 2002.

2. Trigeri

Triger se može definisati kao proceduralni kod koji se automatski izvršava svaki put kada se desi definisani događaj nad određenom tabelom ili pogledom. Trigeri su specifična vrsta ECA (Event – condition – action) pravila. Ova pravila se još zovu i produkciona pravila, a mogu se definisati na sledeći način:

ON događaj **IF** uslov	„situacija“
DO akcija	“(re)akcija”

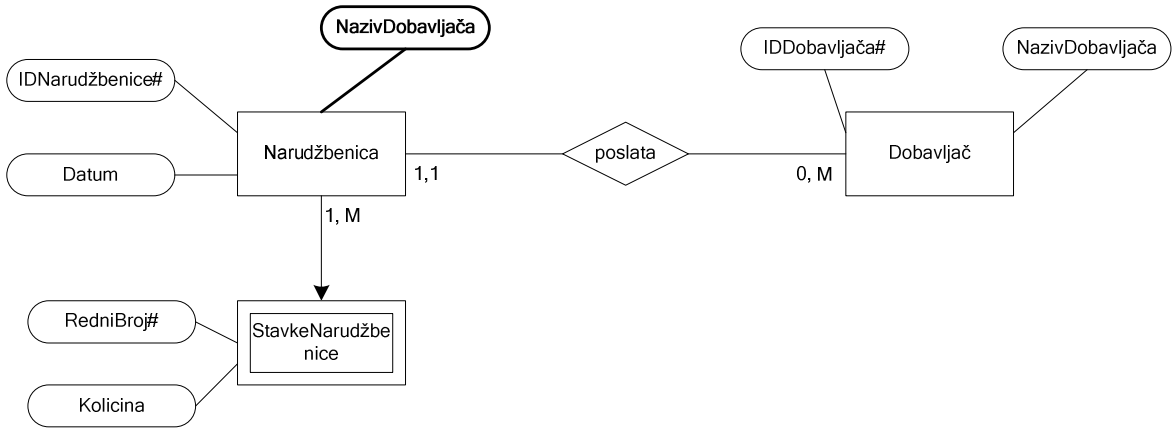
Događaj je operacija ažuriranja baze podataka, uslov je proizvoljni SQL predikat, a akcija je sekvenca SQL naredbi. Pod ažuriranjem se podrazumevaju operacije INSERT, UPDATE ili DELETE. Gore pomenuti trigeri se nazivaju DML (Data Manipulation Language) trigeri jer su definisani kao deo DML naredbi i okidaju se prilikom manipulacije podacima. Neki sistemi takođe podržavaju i drugu vrstu trigera koji se okidaju kao posledica izvršenja DDL (Data Definition Language) naredbi, kao što je kreiranje tabele, ili usled događaja kao što su commit i rollback transakcije. Ovakvi DDL trigeri se mogu koristiti u bazi podataka za potrebe revizije.

Dve osnovne karakteristike trigera su:

- trigeri ne primaju parametre ili argumente
- trigeri ne mogu izvršiti commit ili rollback neke transakcije

2.1. Čemu služe, u kojim slučajevima ih koristimo

Trigeri se pre svega koriste za učuvanje integriteta baze podataka. Na taj način se mogu implementirati pravila integriteta, koja se dele na dve grupe, a to su: pravila integriteta modela i poslovna pravila integriteta (za detaljnije objašnjenje pogledati). Na primer, u bazi imamo tabelu sa dobavljačima i tabelu sa narudžbenicama koje su poslate određenim dobavljačima. Konkretna narudžbenica može biti poslata jednom i samo jednom dobavljaču. Na osnovu pravila prevođenja PMOV-a u relacioni model, znamo da će narudžbenica na sebi imati spoljni ključ, koji je spušten sa dobavljača. Ono što je jako zgodno, i često potrebno, jeste da na narudžbenici imamo i naziv dobavljača, a ne samo njegovu šifru jer time zaobilazimo nepotrebno spajanje tabela.



Na ovom demonstrativnom primeru, jasno vidimo ulogu trigera. Na ovako denormalizovanom modelu, problem bi predstavljala konzistentnost podataka u slučaju ažuriranja NazivaDobavljača. Jasno je da izmena Naziva u tabeli Dobavljač, za sobom povlači i ažuriranje tog istog polja u tabeli Narudžbenica. Upravo tu na scenu stupa triger, koji će se okinuti svaki put kada se izmena desi.

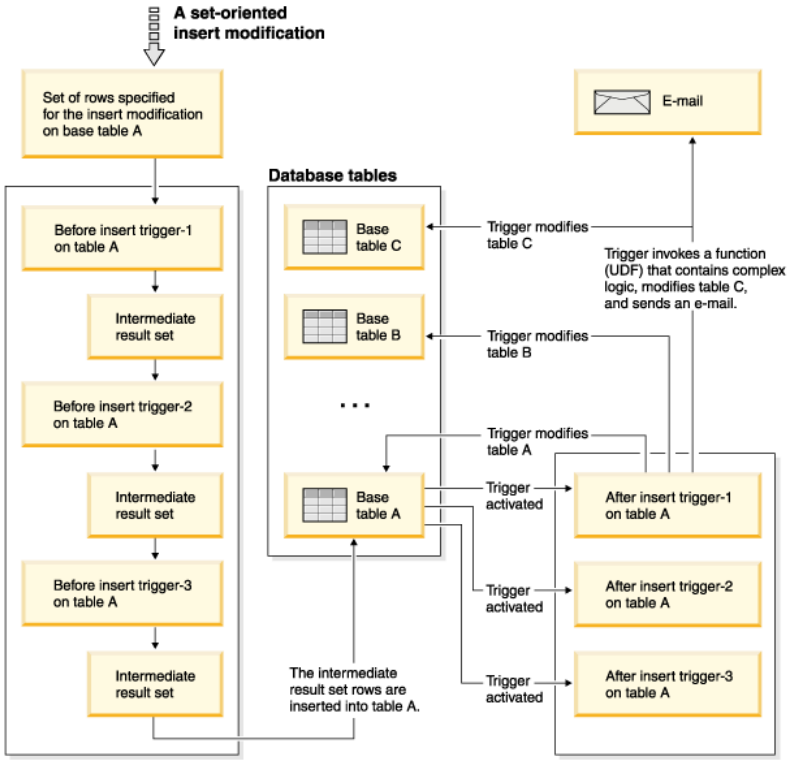
Trigeri se, takođe, mogu koristiti za validaciju podataka, koji se unose i to definisanjem seta pravila napisanih korišćenjem T-SQL jezika. Trigeri omogućavaju administratorima baze podataka da uspostave dodatne veze između odvojenih baza podataka.

Drugi primer korišćenja trigera je podizanje novca sa bankovnog računa. Tom prilikom okidaju se trigeri u dva slučaja:

- Ukoliko nema dovoljno novca na računu, okida se triger koji će o tome obavestiti korisnika

- U slučaju da ima dovoljno sredstava na računu, koirsnik će podići željeni iznos, a tom prilikom će se okinuti triger koji će pozvati uskladištenu proceduru za ažuriranje stanja na računu korisnika

2.2. Princip rada trigera



For a given table with both before and AFTER triggers, and a modifying event that is associated with these triggers, all the BEFORE triggers are activated first. The first activated BEFORE trigger for a given event operates on the set of rows targeted by the operation and makes any update modifications to the set that its logic prescribes. The output of this BEFORE trigger is accepted as input by the next before-trigger. When all of the BEFORE triggers that are activated by the event have been fired, the intermediate result set, the result of the BEFORE trigger modifications to the rows targeted by the trigger event operation, is applied to the table. Then each AFTER trigger associated with the event is fired. The AFTER triggers might modify the same table, another table, or perform an action external to the database.

The different activation times of triggers reflect different purposes of triggers. Basically, BEFORE triggers are an extension to the constraint subsystem of the database management system. Therefore, you generally use them to:

- Perform validation of input data
- Automatically generate values for newly inserted rows
- Read from other tables for cross-referencing purposes

BEFORE triggers are not used for further modifying the database because they are activated before the trigger event is applied to the database.

Consequently, they are activated before integrity constraints are checked.Conversely, you can view AFTER triggers as a module of application logic that runs in the database every time a specific event occurs. As a part of an application, AFTER triggers always see the database in a consistent state. Note that they are run after the integrity constraint validations. Consequently, you can use them mostly to perform operations that an application can also perform. For example:

- Perform follow on modify operations in the database.
- Perform actions outside the database, for example, to support alerts. Note that
- actions performed outside the database are not rolled back if the trigger is rolled back.

In contrast, you can view an INSTEAD OF trigger as a description of the inverse operation of the view it is defined on. For example, if the select list in the view contains an expression over a table, the INSERT statement in the body of its INSTEAD OF INSERT trigger will contain the reverse expression.

Preuzeto sa [http://pic.dhe.ibm.com]

2.3. Podela trigera

Trigeri se mogu pokrenuti jednom za celu operaciju ažuriranja ili po jednom za svaki red koji je ažuriran. U zavisnosti od toga, razlikujemo dve vrste trigera:

- **Trigeri koji se pokreću na nivou naredbe (statement-level trigger)**
- **Trigeri koji se pokreću na nivou reda (row-level trigger)**

Sintaksa kojom se specificira ova osobina je opciona, a prema SQL: 1999 default vrednost je FOR EACH STATEMENT.

Ukoliko koirstimo row-level triger, možemo koristiti old i new ključne reči koje zapravo imaju na referencu na red koji je nov, izmenjen ili obrisan. Ukoliko triger definišemo kao statement level, ova opcija nije dostupna.

U zavisnosti od trenutka aktivacije razlikujemo sledeće vrste trigera:

- **BEFORE**

  Okinute akcije se aktiviraju za svaki red pre nego što se događaj uopšte izvrši. Tabela na koju se ova akcija odnosi će biti izmenjena tek nakon što je BEFORE triger izvršio definisane akcije za svaki red. Iz prethodno navedenog, dolazimo do zaključka da BEFORE treigeri moraju biti definisani na nivou reda (FOR EACH ROW).

- **AFTER**

  Okinute akcije se aktiviraju za svaki red ili jednom na nivou naredbe, u zavisnosti od definicije samog trigera. Ove izmene se dešavaju nakon završenog događaja koji je izazvao okidanje trigera i nakon provere svih ograničenja, kao što je referencijalni integritet, koja mogu biti narušena usled definisanog događaja. Ovde možemo primetiti da se AFTER triger može definisati i na nivou reda i na nivou naredbe (FOR EACH ROW ili FOR EACH STATEMENT)

- **INSTEAD OF**

  INSTEAD OF trigeri moraju imati granularnost FOR EACH ROW, a mogu se definisati isključivo nad pogledom. Ni jedan drugi triger, sem ovog, se ne može definisati nad pogledom.

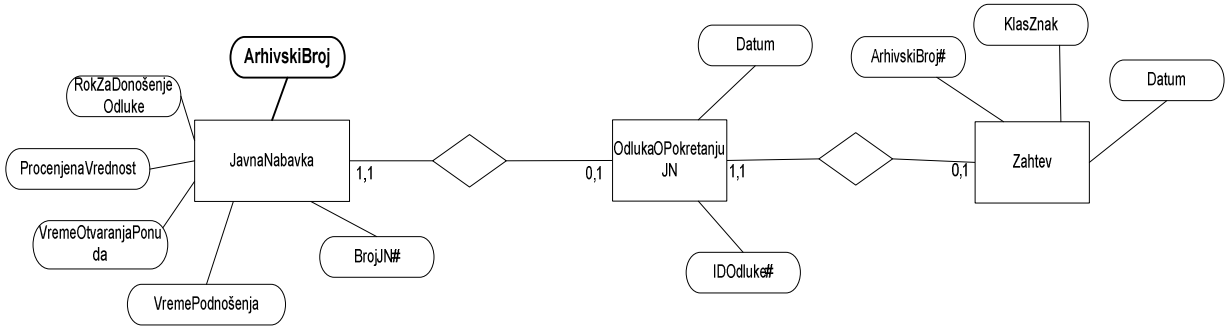
U prethodnoj podeli pomenuto je korišćenje old i new referenci. Ovaj način referenciranja nije dostupan ukoliko se definiše AFTER triger. Kao što je opisano u prethodnom tekstu, kod AFTER trigera akcije se dešavaju tek nakon izvršenog događaja. Upravo ovo je razlog zašto pomenuti način referenciranja nije moguć kod ove vrste trigera.

Trigeri se mogu podeliti i prema događaju koji izaziva njihovo pokretanje. U zavisnosti od tipa operacije ažuriranja, razlikujemo sledeće trigere:

- **INSERT**
- **UPDATE**
- **DELETE**

2.3.1. Trigeri u Oracle SUBP-u

Što se tiče ovog SUBP-a, on omogućava realizaciju svih pomenutih tipova trigera. Ukoliko želimo da definišemo for each statement triger, samo ćemo iz definicije trigera izostaviti ključnu reč for each row. Kao što je već pomenuto, u tom slučaju, prema SQL: 1999 standardu, podrazumeva se statement level triger.



Na navedenom primeru je denormalizovana Treća normalna forma i to korišćenjem Short – circuit keys tehnike.

Na modelu su prikazana tri jaka objekta. Zahtev, Odluka i JavnaNabavka. Na objekat JavnaNabavka spušten je atribut ArhivskiBroj na osnovu vrednosti spoljnog ključa IDOdluke.

Tabela specifikacije trigera za prethodni primer denormalizacije bi bila:

Tabela	Tip trigera	Kolona	Potreban	Šta treba da uradi?
OdlukaOPokretanjuJN	Insert		NE	
	Update	ArhivskiBroj	DA	Ukoliko postoji JavnaNabavka za OdlukuOPokretanjuJN, ažurira ArhivskiBroj u tabeli JavnaNabavka
	Delete		NE	
JavnaNabavka	Insert		DA	U kolonu ArhivskiBroj upisuje odgovarajuću vrednost za unetu vrednost IDOdluke
	Update	IDOdluke	DA	U kolonu ArhivskiBroj upisuje odgovarajuću vrednost na osnovu nove vrednosti za IDOdluke
		ArhivskiBroj	DA	Sprečava direktnu izmenu
	Delete		NE	

U daljem tekstu biće prikazani neki od navedenih trigera koji će se realizovati korišćenjem Oracle SUBP-a.

U slučaju ažuriranja ArhivskogBroja u tabeli OdlukaOPokretanjuJN, pokreće se triger koji treba tu vrednost da izmeni u tabeli JavnaNabavka. Ovaj triger biće realizovan kao AFTER UPDATE triger, a što se tiče granularnosti definisan je kao FOR EACH ROW triger (Primer 2). Kao što smo već objasnili, ukoliko koristimo AFTER klauzulu, nije moguće referenciranje novih i starih vrednosti korišćenjem new i old ključnih reči. Dakle, u ovom slučaju prvo se izvrši ažuriranje ArhivskogBroja u tabeli OdlukaOPokretanjuJN. Tek kada se ova izmena potvrdi (AFTER), pokreće se akcija čiji je cilj da tu izmenjenu vrednost ArhivskogBroja, postavi u tabeli JavnaNabavka i to za svaki red (FOR EACH ROW), koji ima onu vrednost spoljnog ključa IDOdluke koja odgovara izmenjenom redu u tabeli OdlukaOPokretanjuJN. Kako su izmene u tabeli OdlukaOPokretanjuJN već izvršene, SUBP više nema evidenciju o podacima iz reda koji je izmenjen. Da bismo omogućili da se baš ta nova vrednost ArhivskogBroja postavi u svim javnim nabavkama, koje su vezane za tu konkretnu, izmenjenu odluku, neophodno je da definišemo još jedan BEFORE triger, koji će sve neophodne vrednosti smestiti u jedan paket unutar globalnih promenljivih (Primer 1).

Paket se u Oracle-u definiše na sledeći način:

```
CREATE OR REPLACE PACKAGE "PAKET" AS
ODLUKA NUMBER:=0;
SIFRA NUMBER:=0;
IMEPREZIME VARCHAR2(20);
IDZAP NUMBER:=0;
END;
```

Kreirani paket će se koristiti za realizaciju primera korišćenjem Oracle SUBP-a.

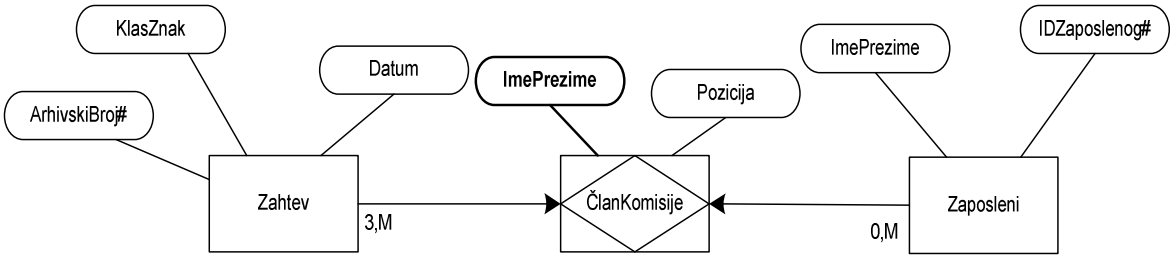
```
CREATE OR REPLACE TRIGGER "ODLUKAJNPRE"
BEFORE INSERT OR UPDATE OR DELETE ON ODLUKAJN
FOR EACH ROW
BEGIN
        IF (INSERTING OR UPDATING)
        THEN
                BEGIN
    PAKET.SIFRA:=:NEW.ARHIVSKIBROJ;
    PAKET.ODLUKA:=:NEW.IDODLUKEOPOKRETANJU;
    PAKET.STARASIFRA:=:OLD.ARHIVSKIBROJ;
    PAKET.STARAODLUKA:=:OLD.IDODLUKEOPOKRETANJU;END;
        ELSE
                BEGIN
    PAKET.SIFRA:=:OLD.ARHIVSKIBROJ;
    PAKET.ODLUKA:=:OLD.IDODLUKEOPOKRETANJU;END;
        END IF;
END;
```

Primer 1 Oracle - BEFORE INSERT, UPDATE OR DELETE triger

```
CREATE OR REPLACE TRIGGER "ODLUKAJN"
AFTER UPDATE OF ARHIVSKIBROJ
ON ODLUKAOPOKRETANJUPOSTUPKA
FOR EACH ROW
DECLARE
PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN
EXECUTE IMMEDIATE 'ALTER TRIGGER NEMENJAJARHIVSKIBROJ DISABLE';
UPDATE JAVNANABAVKA
```

```
SET ARHIVSKIBROJ = PAKET.SIFRA
WHERE IDODLUKEOPOKRETANJU = PAKET.ODLUKA;
COMMIT;
BEGIN
EXECUTE IMMEDIATE 'ALTER TRIGGER NEMENJAJARHIVSKIBROJ ENABLE';
END;
END;
```

Primer 2 Oracle - AFTER UPDATE triger



Na navedenom primeru je denormalizovana Druga normalna forma i to korišćenjem Pre-joining tehnike.

Na modelu su prikazana tri jaka objekta. Zahtev, Zaposleni i ČlanKomisije. Na objekat ČlanKomisije spušten je atribut ImePrezime na osnovu vrednosti ključa IDZaposlenog.

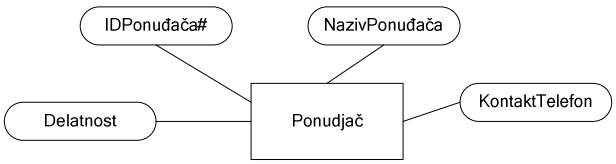
Tabela specifikacije trigera za prethodni primer denormalizacije bi bila:

Tabela	Tip trigera	Kolona	Potreban	Šta treba da uradi?
Zaposleni	Insert		NE	
	Update	ImePrezime	DA	Prilikom izmene vrednosti kolone ImePrezime u tabeli Zaposleni, pokreće se triger koji izmenjenu vrednost ažurira u tabeli ČlanKomisije.
	Delete		NE	
ČlanKomisije	Insert		DA	U kolonu ImePrezime upisuje odgovarajuću vrednost za unetu vrednost IDZaposlenog.
	Update	ImePrezime	DA	Sprečava direktnu izmenu.
	Delete		NE	

Od identifikovanih trigera biće kreiran onaj koji se okida prilikom unošenja novog člana komisije. Dati triger će biti definisan kao AFTER INSERT sa granularnošću FOR EACH ROW. Priča je ista kao i u primeru 2. Zbog tipa datog trigera, moramo kreirati još jedan, koji će pre unošenja podataka o novom članu, ubaciti u paket IDZaposlenog koji se odnosi na novokreiranog člana. Pokreće se akcija koja za cilj ima uzimanje imena i prezimena zaposlenog iz tabele Zaposleni, koji odgovaraju vrednosti promenljive IdZap iz paketa. Dobijeno ime i prezime se smešta nazad u paket, i to u globalnu promenljivu ImePrezime preko koje se postavlja vrednost datog atributa za novog člana komisije.

```
CREATE OR REPLACE TRIGGER "CLANOVI"
    AFTER INSERT
        ON CLANKOMISIJE
        FOR EACH ROW
    DECLARE
        V_IMEPREZIME VARCHAR2(20);
    BEGIN
        SELECT IMEPREZIME INTO V_IMEPREZIME
        FROM ZAPOSLENI
        WHERE IDZAPOSLENOG = PAKET.IDZAP;
        PAKET.IMEPREZIME := V_IMEPREZIME;
END;
```

Primer 3 Oracle - AFTER INSERT triger



Na slici iznad prikazan je objekat koji će poslužiti kao primer za definisanje AFTER DELETE trigera (Primer 4). Nad ovom tabelom je izvršeno i vertikalno particionisanje. To znači da je data tabela zapravo podeljena na dve; jednu u kojoj će biti dati osnovni podaci o Ponuđaču i jednu u kojoj će se čuvati dodatni podaci o datim Ponuđačima.

```
CREATE OR REPLACE TRIGGER NEBRISI
   AFTER DELETE  ON PONUDJAC
   BEGIN
      RAISE_APPLICATION_ERROR (
         NUM => -20000,
         MSG => 'NE SMETE DA BRISETE POSTOJECE PONUDJACE');
   END;
```

Primer 4 Oracle - AFTER DELETE triger

U Oracle SUBP-u ukoliko želimo da definišemo FOR EACH STATEMENT triger, samo je potrebno izostaviti FOR EACH ROW. STATEMENT ključna reč NE POSTOJI u Oracle-u.

Kao što smo već rekli, nad prethodnim primerom je izvršeno vertikalno particionisanje. Kako bi se objedinili svi podaci iz ove dve navedene tabele definisaćemo pogled Sve_O_Ponudjacu. U ovom

slučaju je neophodno definisati jednu posebnu vrstu trigera koja je specijalizovana za rad sa pogledima. U ovu svrhu se koriste INSTEAD OF trigeri koji omogućavaju unošenje podataka o novom ponuđaču i to preko pogleda. Moramo naglasiti da je u ovom slučaju korišćenje INSTEAD OF trigera nezaobilazno jer je potrebno spustiti podatke u dve različite tabele – Ponudjac i PonudjacDetalji.

```
Ponudjac (IDPonudjaca, NazivPunudjaca)
PonudjacDetalji (IDPonudjaca, Delatnost, KontaktTelefon)
```

Kreiranje tabele Ponudjac

```
CREATE TABLE PONUDJAC (
    IDPONUDJJACA NUMBER(7) PRIMARY KEY,
    NAZIVPONJUDJACA VARCHAR(20));
```

Kreiranje tabele PonudjacDetalji

```
CREATE TABLE PONUDJACDETALJI (
    IDPONUDJJACA NUMBER(7) PRIMARY KEY,
    DELATNOST VARCHAR(20),
    KONTAKTTELEFON VARCHAR(20));
```

Kreiranje pogleda Sve_O_Ponudjacu

```
CREATE OR REPLACE VIEW SVE_O_PONUDJACU
AS
SELECT P.IDPONUDJACA, P.NAZIVPONUDJACA, D.DELATNOST, D.KONTAKTTELEFON
FROM PONUDJAC P, PONUDJACDETALJI D
WHERE P.IDPONUDJACA = D.IDPONUDJACA;
```

Kreiranje trigera POGLEDTRIGER pomoću kojeg će se omogućiti unošenje novog ponuđača preko pogleda je prikazano primerom 5.

```
CREATE OR REPLACE TRIGGER POGLEDTRIGER
INSTEAD OF INSERT ON SVE_O_PONUDJACU
REFERENCING NEW AS NOVI
FOR EACH ROW
BEGIN
INSERT INTO PONUDJAC (IDPONUDJACA, NAZIVPONUDJACA) VALUES
(:NOVI.IDPONUDJACA, : NOVI.NAZIVPONUDJACA);
INSERT INTO PONUDJACDETALJI (IDPONUDJACA, DELATNOST, KONTAKTTELEFON) VALUES
(:NOVI.IDPONUDJACA, : NOVI.DELATNOST, : NOVI.KONTAKTTELEFON);
END;
```

Primer 5 Oracle - INSTEAD OF triger

Kreiranje novog ponuđača preko pogleda `SVE_O_PONUDJACU`:

```
INSERT INTO SVE_O_PONUDJACU VALUES (1,'NAZIV','DELATNOST','KONTAKT');
```

2.3.2. Trigeri u Microsoft SQL Server SUBP-u

U prethodnom poglavlju su već date sve podele trigera. Kao što je prikazano na primerima, Oracle omogućava implementaciju svih tipova trigera. Međutim, situacija sa SQL Serverom nije takva. SQL Server ne podržava FOR EACH ROW trigere, već ovde postoje samo oni koji se izvršavaju na vivou naredbe (statement level triggers). Upravo zbog rada sa većim brojem redova, a ne samo sa jednom, možemo reći da su trigeri u datum SUBP-u kompleksniji. Ako se vratimo na objašnjenja svakog tipa trigera iz četvrtog poglavlja, takođe, možemo videti da BEFORE trigeri moraju biti definisani kao FOR EACH ROW. Odavde dolazimo do zaključka da ovaj tip trigera nije podržan u SQL Serveru. Podržani su sledeće vrste DML (eng. Data manipulation language) trigera:

- AFTER trigeri su bili jedini tip trigera u verzijama pre SQL Server 2000. Takođe su poznati pod nazivom „FOR trigeri“, a često je korišćen samo termin trigeri s obzirom da je navedeni tip bio jedini. Razlika u odnosu na Oracle je što se ovi trigeri ne mogu definisati na nivou reda, već samo na nivou naredbe. Ostale karakteristike su iste. Operacije ažuriranja mogu biti INSERT, UPDATE ili DELETE, a koriste se isključivo za rad sa tabelama.
- INSTEAD OF trigeri imaju istu ulogu kao i u Oracle-u, a to je rad sa pogledima, ali se ovde mogu definisati i nad tabelama.
- CLR su novi u odnosu na ostale vrste SUBP-ova, a mogu biti definisani kao AFTER ili kao INSTEAD OF. Takođe mogu imati ulogu DDL (eng. Data definition language) trigera. U tom slučaju ovi trigeri umesto izvršenja uskladištene procedure, izvršavaju jednu ili više metoda koje su definisane unutar assembly-a kreiranog korišćenjem .NET framework-a.

Svi primeri koji su prikazani korišćenjem Oracle-a, sada će biti realizovani uz pomoć SQL server-a.

Ono što još moramo napomenuti kada je ovaj SUBP u pitanju jeste referenciranje starih i novih vrednosti. U tu svrhu se koriste logičke tabele inserted i deleted. Ove specijalne tabele se automatski kreiraju. One se mogu koristiti za postavljanje uslova za akcije u trigerima. Direktno menjanje podataka nije moguće.

Inserted i deleted tabele se pre svega koriste u upitima i time omogućavaju sledeće:

- Proširenje referencijalnog integriteta među tabelama
- Ubacivanje ili ažuriranje podataka u baznim tabelama
- Provera grešaka i sprovođenje akcija u skladu sa nastalom greškom
- Identifikovanje razlika u stanju tabele pre i posle menjanja podataka i peduzimanje odgovarajućih akcija

Tabela deleted čuva kopije redova izmenjenih prilikom DELETE ili UPDATE naredbe. Prilikom izvršenja neke od ovih naredbi, odgovarajući redovi se brišu iz tabele nad kojom je definisan triger i premeštaju se u deleted tabelu. Ova i bazna tabela nemaju zajedničke redove.

Inserted tabela skladišti redove koji su obuhvaćeni INSERT ili UPDATE naredbom. Prilikom izvršenja jedne od ovih naredbi, novi redovi se istovremeno dodaju i u inserted tabelu i u tabelu nad kojom je dati triger definisan. Redovi u inserted tabeli su zapravo kopije istih iz bazne tabele.

Operacija ažuriranja se može definisati kao operacija brisanja, nakon koje je izvršeno unošenje novih redova. Stari redovi se nalaze u deleted tabeli, a novi redovi se ubacuju u inserted tabelu.

Primeri koji su prethodno prikazani u Oracle-u, sada će se iskoristiti i za njih će se prikazati način implamentacije korišćenjem SQL Server SUBP-a. Naravno, pri tom uzimamo u obzir dostupne vrste trigera.

```
CREATE TRIGGER "CLANOVI"
        ON [DBO].[CLANKOMISIJE]
AFTER INSERT AS
    BEGIN
      UPDATE [DBO].[CLANKOMISIJE]
SET IMEPREZIME =
    (
    SELECT [IMEPREZIME1] FROM [DBO].[ZAPOSLENI]
    WHERE [IDZAPOSLENOG] = (SELECT [IDZAPOSLENOG] FROM INSERTED)
    )
FROM [DBO].[CLANKOMISIJE] Y
JOIN INSERTED I ON Y.IDZAPOSLENOG = I.IDZAPOSLENOG
WHERE I.IMEPREZIME IS NULL
END
```

Primer 6 MS SQL Server - AFTER INSERT triger

```
CREATE TRIGGER NEBRISI
   ON [DBO].[PONUDJAC]
AFTER DELETE AS
   BEGIN
     RAISERROR('NE SME SE BRISATI!', 16, -1)
     ROLLBACK TRAN
     RETURN
   END
```

Primer 7 MS SQL Server - AFTER DELETE triger

```
CREATE TRIGGER [dbo].[ODLUKAJN] ON [dbo].[ODLUKAOPOKRETANJUPOSTUPKA]
AFTER UPDATE AS
IF UPDATE (ARHIVSKIBROJ)
BEGIN
ALTER TABLE [dbo].[JAVNANABAVKA] DISABLE TRIGGER [NEMENJAJARHIVSKIBROJ]
UPDATE [DBO].[JAVNANABAVKA]
SET [ARHIVSKIBROJ] = (SELECT [ARHIVSKIBROJ] FROM INSERTED)
WHERE IDODLUKEOPOKRETANJU = (SELECT [IDODLUKE] FROM INSERTED)
ALTER TABLE [dbo].[JAVNANABAVKA] ENABLE TRIGGER [NEMENJAJARHIVSKIBROJ]
END;
```

Primer 8 MS SQL Server - AFTER UPDATE triger

Kreiranje trigera POGLEDTRIGER pomoću kojeg će se omogućiti unošenje novog ponuđača preko pogleda

```
CREATE TRIGGER POGLEDTRIGER ON SVE_O_PONUDJACU
INSTEAD OF INSERT
AS
BEGIN
SET NOCOUNT ON
IF (NOT EXISTS (SELECT P.IDPONUDJACA
      FROM PONUDJAC P, INSERTED I
      WHERE P.IDPONUDJACA = I.IDPONUDJACA))
   INSERT INTO PONUDJAC
      SELECT IDPONUDJACA, NAZIVPONUDJACA
      FROM INSERTED

IF (NOT EXISTS (SELECT E.IDPONUDJACA
      FROM PONUDJACDETALJI E, INSERTED
      WHERE E.IDPONUDJACA = INSERTED.IDPONUDJACA))
   INSERT INTO PONUDJACDETALJI
      SELECT IDPONUDJACA, DELATNOST, KONTAKTTELEFON
      FROM INSERTED
ELSE
   UPDATE PONUDJACDETALJI
      SET IDPONUDJACA = I.IDPONUDJACA,
          DELATNOST = I.DELATNOST,
          KONTAKTTELEFON = I.KONTAKTTELEFON
   FROM PONUDJACDETALJI E, INSERTED I
   WHERE E.IDPONUDJACA = I.IDPONUDJACA
END
```

Primer 9 MS SQL Server - INSTEAF OF triger

Kreiranje novog ponuđača preko pogleda SVE_O_PONUDJACU:

```
INSERT INTO [DBO].[SVE_O_PONUDJACU] VALUES (3,'PON3','DEL3','KON3');
```

```
CREATE TRIGGER NEMENJAJARHIVSKIBROJ
   ON [DBO].[JAVNANABAVKA]
AFTER UPDATE AS
   BEGIN
      IF UPDATE (ARHIVSKIBROJ)
     RAISERROR('NE SME SE MENJATI!', 16, -1)
   END
```

Primer 10 MS SQL Server – Zabrana ažuriranja

2.3.3. Trigeri u PostgreSQL SUBP-u

PostgreSQL SUBP omogućava implementaciju svih trigera koji su definisani u četvrtom poglavlju. Međutim, ovde se uvodi par novina. Za razliku od Oracle SUBP-a, ovde se INSTEAD OF trigeri mogu definisati i nad tabelama, a ne samo nad pogledima. Ista situacija je bila i kod SQL Server-a. Još jedna novina jeste i mogućnost definisanja BEFORE i AFTER trigera nad pogledom, ali u tom slučaju on mora biti definisan na nivou naredbe. Pored standardnih operacija ažuriranja, pokretanje trigera može izazvati i TRUNCATE naredba, ali u tom slučaju on mora biti definisan na nivou naredbe.

U nastavku je data tabela koja prikazuje postojeće tipove trigera kao i to da li mogu biti definisani nad tabelom ili pogledom.

Kada	Događaj	Row-level	Statement-level
BEFORE	INSERT/UPDATE/DELETE	Tabele	Tabele i pogledi
	TRUNCATE	-	Tabele
AFTER	INSERT/UPDATE/DELETE	Tabele	Tabele i pogledi
	TRUNCATE	-	Tabele
INSTEAD OF	INSERT/UPDATE/DELETE	Pogledi	-
	TRUNCATE	-	-

Tabela 2 - Implementirani tipovi trigera unutar PostgreSQL-a

Reference

1. Lazarević B., Marjanović Z., Aničić N., Babarogić S., „Baze podataka“, Beograd, 2006.
2. „Types of DML Triggers“,

   http://technet.microsoft.com/en-us/library/ms178134%28v=sql.105%29.aspx
3. „Programming CLR Triggers“,

   http://technet.microsoft.com/en-us/library/ms179562%28v=sql.105%29.aspx
4. „PostgreSQL 9.1.9 Documentation“, http://www.postgresql.org/docs/9.1/static/sql-createtrigger.html

3. Objektno-relacioni model

Dva najznačajnija modela podataka na kojima se zasnivaju SUBP-ovi su relacioni i objektni model. Relacije (tabele) koje čine konvencionalnu relacionu bazu podataka su normalizovane (ravne) strukture, sa jednostavnim predefinisanim tipovima podataka nad kojima su definisani njihovi atributi. Ovakva struktura relacionih tabela značajno ograničava i usložnjava kompleksnije upite i kompleksniju obradu podataka. Da bi se ovaj nedostatak otklonio, relacioni model se proširuje idejama i konceptima objektnog modela. Tako dolazimo do objektno-relacionog modela i objektno-relacionih sistema koji nastoje da integrišu najbolje karakteristike objektne i relacione tehnologije. Konceptualni model objektno-relacionih baza podataka sastavljen je od skupa međusobno povezanih relacija, ali domeni atributa relacija ne moraju biti samo osnovni predefinisani tipovi podataka, već i korisnički definisani osnovni i složeni tipovi podataka koji mogu biti organizovani u genralizacione hijerarhije tipova.

Bitna karakteristika objektno-relacionih sistema je mogućnost definisanja metoda. Njima se definišu operacije nad korisničkim tipovima. Neke od njih se automatski generišu (metode za kreiranje pojavljivanja novog tipa, metode za pristup i izmenu). Ostale korisnik sam definiše, ukoliko ima potrebe za njima. Navedene objektne karakteristike objektno-relacionih sistema standardizovane su SQL:1999 standardom, putem specifikacije **korisnički definisanih tipova** i **konstruisanih tipova podataka**.

3.1. Korisnički definisani tipovi

Korisnički definisani tipovi su tipovi koje definiše korisnik i koji se koriste na isti način kao i predefinisani (ugrađeni) tipovi. Uvođenjem korisnički definisanih tipova pojednostavljuje se razvoj i održavanje aplikacija, jer definisane tipove može da koristi više aplikacija. Korisnički definisani tip može biti **distinkt tip** ili **struktuirani tip**.

3.1.1. Struktuirani tip

Struktuirani tip omogućava definisanje perzistentnih, imenovanih tipova, koji mogu imati jedan ili više atributa. Atributi mogu biti bilo kog tipa, uključujući druge struktuirane tipove, nizove itd, tako da mogu predstavljati strukture proizvoljne složenosti. Pored strukture definisane atributima, struktuirani tipovi imaju i ponašanje, definisano metodama, koje čine sastavni deo specifikacije struktuiranog tipa. Struktuirani tipovi podržavaju nasleđivanje, što znači da se pri definisanju novog tipa mogu naslediti svi atributi i metode nekog postojećeg tipa. Struktuirani tip se definiše sledećom sintaksom:

```
CREATE TYPE <naziv struktuiranog tipa>
[UNDER <naziv nadtipa>]
AS (<predefinisai tip>  <tip atributa>, ...)
[[NOT] INSTANTIABLE]
NOT FINAL
[<specifikacija referenciranja>]
[<specifikacija metode>, ...]
```

Sledi primer korisnički definisanog tipa Adresa, koji sadrži atribute naziv ulice, broj zgrade i dodatak broju, kreiranog u Oracle-u:

```
CREATE OR REPLACE
TYPE obj_adresa_br AS OBJECT
( ulica varchar2(50),
  broj  number,
  dodatakbroju VARCHAR2(10),
MEMBER FUNCTION get_ulica RETURN varchar2,
MEMBER FUNCTION get_broj RETURN number,
MEMBER FUNCTION get_dodatakbroju RETURN varchar2
)
INSTANTIABLE NOT FINAL;

CREATE OR REPLACE
TYPE BODY obj_adresa_br AS
      MEMBER FUNCTION get_ulica RETURN varchar2 IS
      BEGIN
        RETURN SELF.ulica;
      END;
      MEMBER FUNCTION get_broj RETURN number IS
      BEGIN
       RETURN SELF.broj;
      END;
      MEMBER FUNCTION get_dodatakbroju RETURN varchar2 IS
      BEGIN
        RETURN SELF.dodatakbroju;
      END;
END;
```

Naredba za kreiranje tabele adresa, koja će sadržati kolonu sa definisanim tipom je:

```
CREATE TABLE adresa ( sifra_adr number NOT NULL,
                      adresa obj_adresa_br,
                      CONSTRAINT adresa_pk PRIMARY KEY (sifra_adr));
```

Naredba za unos podataka u tabelu adresa, koja sadrži korisnički definisani tip je:

```
INSERT INTO adresa
       VALUES (1,obj_adresa_br('Jove Ilica',154,'A'));
```

Naredba za prikaz podataka iz tabele adresa, koja sadrži korisnički definisani tip je:

```
SELECT
       a.sifra_adr "Sifra adrese",
```

```
        a.adresa.get_ulica() "Ime ulice",
        a.adresa.get_broj() "Broj",
        a.adresa.dodatakbroju.get_dodatakbroju()"Dodatak broju"
FROM adresa a;
```

Naredba za kreiranje opisanog korsiničkog tipa u Microsoft okruženju data je u nastavku:

```
using System;
using System.Data;
using System.Data.SqlClient;
using System.Data.SqlTypes;
using Microsoft.SqlServer.Server;
using System.IO;

[Serializable]
[Microsoft.SqlServer.Server.SqlUserDefinedType(Format.UserDefined,
MaxByteSize = 800)]

public struct Adresa : INullable, IBinarySerialize
{
    public bool IsNull
    {
        get
        {
            // Put your code here
            return m_Null;
        }
    }
    public static Adresa Null
    {
        get
        {
            Adresa h = new Adresa();
            h.m_Null = true;
            return h;
        }
    }
    private string nazivUlice;
    private int broj;
    private string dodatakBroju;
    private bool m_Null;

    public Adresa(string nazivUlice, int broj, string dodatakBroju)
    {
        this.nazivUlice = nazivUlice;
        this.broj = broj;
        this.dodatakBroju = dodatakBroju;
        m_Null = false;
    }
    public override string ToString()
    {
        if (this.IsNull)
                return "null";
        else
        {
                string delim = new string((new char[] { ';' }));
                return(this.nazivUlice  +  delim  +  this.broj  +  delim  +
                        this.dodatakBroju);
        }
    }
```

```
    public static Adresa Parse(SqlString s)
    {
        if (s.IsNull)
                return Null;
        else
                {
                        Adresa addr = new Adresa();
                        string str = Convert.ToString(s);
                        string[] a = null;
                        a = str.Split(new char[] { ';' });
                        addr.nazivUlice = a[0] == null ? string.Empty :
a[0];
                        int broj = Convert.ToInt32(a[1]);
                        ValidateBroj(broj);
                        addr.broj = broj;
                        if (a.Length == 3)
                        {
                                addr.dodatakBroju = a[2];
                        }
                        else
                        {
                                addr.dodatakBroju = string.Empty;
                        }
                        addr.m_Null = false;
                        return (addr);
                }
        }

        private static void ValidateBroj(int broj)
        {
                if (broj < 0)
                {
                        throw new ArgumentOutOfRangeException("Broj ne može
biti manji od 0");
                }
        }

        public string NazivUlice
        {
                get
                {
                        return (this.nazivUlice);
                }
                set
                {
                        this.nazivUlice = value;
                        this.m_Null = false;
                }
        }

        public int Broj
        {
                get
                {
                        return (this.broj);
                }
                set
                {
                        this.broj = value;
```

```
                        this.m_Null = false;
                }
        }
        public string DodatakBroju
        {
                get
                {
                        return (this.dodatakBroju);
                }
                set
                {
                        if (!string.IsNullOrEmpty(value))
                        {
                                this.dodatakBroju = value;
                                this.m_Null = false;
                        }
                }
        }

        public override bool Equals(object other)
        {
                return this.CompareTo(other) == 0;
        }

        public override int GetHashCode()
        {
                if (this.IsNull)
                        return 0;

                return this.ToString().GetHashCode();
        }

        public int CompareTo(object other)
        {
                if (other == null)
                        return 1; //by definition


                Adresa addr = (Adresa)other;

                if (addr.Equals(null))
                        throw new ArgumentException("the argument to compare
is not a adresa");

                if (this.IsNull)
                {
                        if (addr.IsNull)
                                return 0;

                        return -1;
                }

                if (addr.IsNull)
                        return 1;

                return this.ToString().CompareTo(addr.ToString());
        }

        public void Write(System.IO.BinaryWriter w)
        {
```

```
                byte header = (byte)(this.IsNull ? 1 : 0);

                w.Write(header);
                if (header == 1)
                {
                        return;
                }

                w.Write(this.NazivUlice);
                ValidateBroj(this.Broj);
                w.Write(this.Broj);
                w.Write(this.DodatakBroju);
        }

        public void Read(System.IO.BinaryReader r)
        {
                byte header = r.ReadByte();

                if (header == 1)
                {
                        this.m_Null = true;
                        return;
                }

                this.m_Null = false;
                this.nazivUlice = r.ReadString();
                int broj = r.ReadInt32();
                ValidateBroj(broj);
                this.broj = broj;
                this.dodatakBroju = r.ReadString();
        }
}
```

3.1.2. Distinct tip

Distinkt tip je jednostavan, perzistentni, imenovani korisnički definisani tip, čijim uvođenjem je podržano strogo tipiziranje. Za distinkt tip se može reći da je preimenovani predefinisani SQL tip sa drugačijim ponašanjem u odnosu na izvorni. Distinkt tip se definiše sledećom sintaksom:

```
CREATE TYPE <naziv distinct tipa>
AS <predefinisai tip> FINAL
[<cast opcije prevođenja>]
```

Distinkt tipovi su uvek konačni (FINAL), što znači da ne mogu imati podtipove, odnosno za njih nije podržano nasleđivanje. Distinkt tip i njegov izvorni predefinisani tip nisu direktno uporedivi. Opcije prevođenja omogućuju konverziju podataka iz distinkt tipa u predefinisani i obratno. Može se koristiti CAST naredba za prevođenje iz distinkt tipa u izovrni predefinisani tip i obratno. Na taj način se obezbeđuje automatska (implicitna) konverzija kada se atributi definisani nad distinkt tipovima koriste u izrazima. Dozvoljeno je i kreiranje proizvoljnog broja metoda, funkcija i procedura, čijim pažljivim izborom se može obezdediti željena funkcionalnost. Iako po SQL:1999 standardu distinct predstavlja korisnički definisani tip, u PL/SQL-u, koji koristi Oracle, nije eksplicitno podržan ovaj tip.

Struktuirani i distinct tip iz SQL:1999 standarda se u Oracle-u implementiraju preko Object tipa. Naime, u Oracle-u, Object tip sa samo jednim atributom odgovara distinct tipu po SQL:1999 standardu, dok Object tip sa dva ili više atributa odgovara struktuiranom tipu po SQL:1999 standardu. Dat je primer korisničkog tipa dinar, koji ima atribut vrednost tipa number(9,2).

```
CREATE OR REPLACE TYPE  "dinar" AS OBJECT (vrednost number(9,2),
MEMBER FUNCTION get_vrednost RETURN number)
INSTANTIABLE NOT FINAL;
/
CREATE OR REPLACE TYPE BODY  "DINAR" AS
      MEMBER FUNCTION get_vrednost RETURN number IS
      BEGIN
        RETURN SELF.vrednost;
      END;
END;
```

Kreirani tip je iskorišćen u tabeli Stavka_Cenovnika, kao tip kolone cena. Relacioni model Stavke_Cenonika je dat u nastavku.

STAVKA_CENOVNIKA(SifraCenovnika, RedniBroj, Cena, SifraProizvoda)

Naredba za kreiranje Stavke_Cenovika je:

```
CREATE TABLE stavka_cenovnika(
SifraCenovnika number,
RedniBroj number,
Cena dinar,
SifraProizvoda number,
CONSTRAINT scfk1 FOREIGN KEY (ŠifraCenovnika) REFERENCES
cenovnik(ŠifraCenovnika),
CONSTRAINT scfk2 FOREIGN KEY (ŠifraProizvoda) REFERENCES
proizvod(ŠifraProizvoda),
CONSTRAINT scpk PRIMARY KEY (ŠifraCenovnika, RedniBroj));
```

Naredba za unos podataka u tabelu sa korisniči definisanim tipom (stavka_cenovnika):

```
INSERT INTO stavka_cenovnika VALUES(1,1,dinar(1000),1)
```

Naredba za prikaz podataka tabele sa korisniči definisanim tipom (stavka_cenovnika):

```
SELECT
sc.sifracenovnika,
sc.rednibroj,
sc.cena.get_vrednost(),
sc.sifraproizvoda
FROM stavka_cenovnika sc
WHERE sifraproizvoda=1;
```

Naredba za kreiranje opisanog korsiničkog tipa u Microsoft okruženju data je u nastavku:

```
CREATE TYPE [dbo].[dinar] FROM [decimal](9, 2);
```

3.2. Konstruisani tipovi

Po SQL:1999 standardu, konstruisani tipovi su referentni tipovi (reference), tipovi vrste i kolekcije. Pored navedenih tipova na kraju će biti ukratko prikazan i nested table tip. Iako vodeći proizvođači podržavaju navedene tipove (poput Oracle-a), ovi tipovi se daju samo u vidu kratkog prikaza i neće biti zastupljeni u projektnom radu.

3.2.1. Referentni tip

Tabele definisane direktno nad struktuiranim tipovima mogu imati referentnu kolonu, koja služi kao identifikator n-torki. Takva kolona može biti primarni ključ ili kolona sa jedinstvenim vrednostima koje automatski generiše sistem za upravljanje bazom podataka. Sintaksa REF tipa je data kroz sledeći primer:

```
ATRIBUT REF(TIP) SCOPE RELACIJA
```

Ukoliko je TIP iz primera neki struktuirani tip, onda je REF(TIP) referentni tip, odnosno tip reference na n-torku tipa TIP. Referenciranju može biti određen opseg (SCOPE), koji se specificira navođenjem naziva relacije čije se n-torke referenciraju (u primeru je to RELACIJA).

3.2.2. Tip vrsta

Ovaj tip se definiše kao par (<naziv podatka>, <tip podatka>). Od standrada SQL:1999 uvedeno je mogućnost definisanja promenljivih i parametara koji su tipa vrsta, odnosno definisanje kolone u tabeli koja će imati kompleksnu strukturu. Primer kreiranja raw tipa dat je u nastavku.

```
create table kolekcija_boja (boja raw(16));
```

Primer za unos vrednosti u tabelu koja sadrži raw tip:

```
insert into kolekcija_boja values ('FF0000');
```

Prikazaivanje vrednosti iz tabele sa raw tipom podataka:

```
select * from kolekcija_boja where utl_raw.substr(boja, 1, 2) = 'FF';
```

3.2.3. Varrays

Array predstavlja numerisani niz čiji elementi su podaci. Svi elementi unutar array-a su istog tipa. Svaki element ima svoj indeks, koji označava mesto elementa u nizu. Broj elementa u nizu definisan je veličinom array-a i maskimalna veličina se mora odrediti prilikom kreiranja arrays-a. Oracle dozvoljava upotrebu array-a promenljive dužine, pa je varrays skraćenica od variable arrays. Veličina varrays-a se može menjati. Sintaksa za kreiranje varrays:

```
CREATE OR REPLACE TYPE name-of-type IS VARRAY(nn)of type
```

Konkretan primer kreiranja varrays-a *phones* je dat u nastavku:

```
CREATE TYPE phones AS VARRAY(10) OF varchar2(10);
```

Sledećom naredbom kreirani tip se koristi u tabeli:

```
reate table suppliers (supcode number(5),
             Company varchar2(20),
             ph   phones);
```

Naredba za unos podataka u tabelu sa varrays tipom:

```
insert into suppliers values (101,'Interface Computers',
          Phones('64199705','55136663'));
insert into suppliers values (102,'Western Engg. Corp',
          Phones('23203945','23203749','9396577727'));
```

Naredba za prikaz podataka tabele sa varrays tipom:

```
Select * from suppliers;
```

3.2.4. Nested table

Nested table je tip koji podržava viševrednosne atribute, tj. podržava kolone koje mogu sadržati kompletne druge tabele. Ovj vid kolekcije narušava prvu normalnu formu i neće biti korišćen u projektnom radu. Primer za kreiranje ovog korisničkog tipa(tabela Department koristi CourseList):

```
CREATE OR REPLACE TYPE CourseList AS TABLE OF VARCHAR2(64);
SELECT type, text
FROM user_source
WHERE name = 'COURSELIST';
CREATE TABLE department (
name     VARCHAR2(20),
```

```
director VARCHAR2(20),
office   VARCHAR2(20),
courses  CourseList)
NESTED TABLE courses STORE AS courses_tab;
```

Naredba za prikaz podataka tabele sa nested table tipom:

```
SELECT column_name, data_type, data_length
FROM user_tab_cols
WHERE table_name = 'DEPARTMENT';

SELECT table_name, table_type_owner, table_type_name,
parent_table_column
FROM user_nested_tables;
```

Naredba za unos podataka u tabelu sa nested table tipom:

```
SELECT cardinality(courses)
FROM department;

INSERT INTO department
(name, director, office, courses)
VALUES
('English', 'Lynn Saunders', 'Breakstone Hall 205', CourseList(
'Expository Writing',
'Film and Literature',
'Modern Science Fiction',
'Discursive Writing',
'Modern English Grammar',
'Introduction to Shakespeare',
'Modern Drama',
'The Short Story',
'The American Novel'));

SELECT * FROM department;

SELECT cardinality(courses)
FROM department;
```

Reference

U nastavku su navedene reference korišćene za poglavlje o objekto-rleacionom modelu. Njaveći deo poglavlja je preuzet iz reference [1], koja predstavlja i osnov navedenog poglavlja (uopšteno i objektno-relacionom modelu, korisnički definisanim i konstruisanim tipovima). Ostale reference su korišćene kao uzor za kreiranje autentičnih primera [7 - 9] i za proširenje teoretskog osnova spomenutog objektno-relacionog modela [2 - 6].

1. Lazarević B, Marjanović Z, Aničić N, Babarogić S *„Baze podataka“* FON, 2006.
2. Connoly T., Begg C., *„Database Solutions: A step-by-step guide to building databases“*, Pearson Education Limited, 2004.
3. Buxton S., et al., *„Database Design: Know it all“*, Morgan Kaufmann, 2009.
4. Powell G., *„Beginning Database Design“*, Wiley Publishing, 2006.
5. Hoffer J., Prescott M., McFadden F., *„Modern Database Management“*, Pearson Education, 2007.
6. Hoberman S., *„Data Modeler's Workbench“*, John Wiley & Sons, 2002.
7. Feuerstein S, Pribyl B *„Oracle PL/SQL Programming“* O'Reilly, 2009.
8. Rosenzweig B, Rakhimov E *„Oracle PL/SQL by Example“* Prentice Hall, 2009.
9. Whitehorn M, Marklyn B, *„Inside Relational Databases with Examples in Access“*, Springer, 2007.

4. Optimizacija

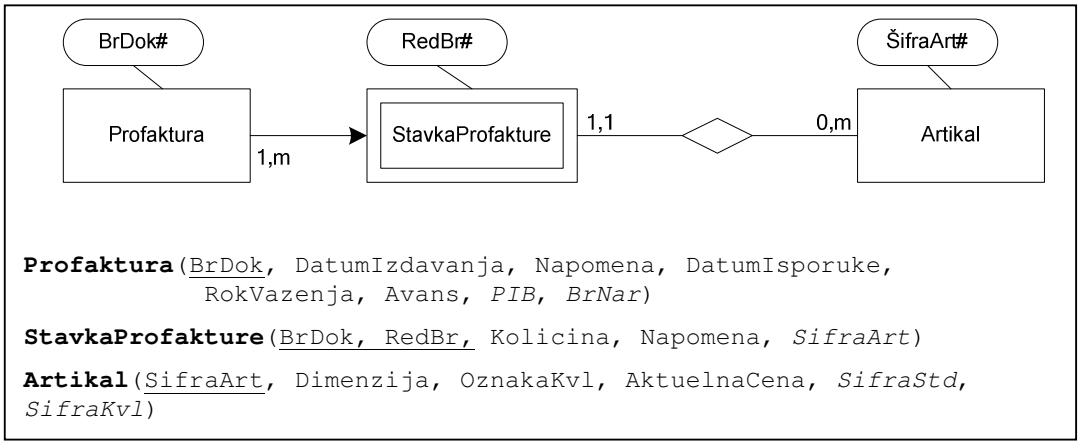
Optimizacija baze podataka obuhvata različite tehnike kojima se žele poboljšati performanse rada baze podataka. U ovom delu biće opisani: tehnike optimizacije zasnovane na upotrebi izvedenih vrednosti, indeksi i vertikalno particionisanje.

4.1. Tehnike optimizacije zasnovane na izvedenim vrednostima

Ovde će biti ukratko opisane tehnike optimizacije zasnovane na upotrebi izvedenih vrednosti. Osnovna ideja optimizacije upotrebom izvedenih vrednosti slična je ideji denormalizacije. I u slučaju optimizacije upotrebom izvedenih vrednosti polazi se od normalizovanog konceptualnog modela u koji se uvodi redundansa podataka, sa ciljem poboljšanja performansi baze podataka. Ipak, bitna razlika između optimizacije upotrebom izvedenih vrednosti i denormalizacije je u tome što se u slučaju optimizacije ne narušavaju normalne forme, već se redundansa podataka uvodi kako bi se izbegla potreba za čestim obračunavanjem izvedenih vrednosti. Pod izvedenom vrednošću podrazumeva se vrednost koja se određuje ili izračunava na osnovu jedne ili više drugih vrednosti (npr. ukupna vrednost računa je izvedena vrednost, dobijena sumom iznosa svih stavki računa). U nastavku će biti opisane različite tehnike optimizacije zasnovane na izvedenim vrednostima.

4.1.1. Storing Derivable Values tehnika optimizacije

Ukoliko se u upitima nad bazom podataka često javlja potreba za određenim izračunavanjem vrednosti, pogodno bi bilo čuvati tu izvedenu vrednost, rezultat navedenog izračunavanja. Kako bi se omogućilo čuvanje izvedene vrednosti, uvodi se reduntantna kolona odgovarajuće tabele u koju se smešta izračunata izvedena vrednost. Pretpostavimo da je dat sledeći konceptualni model[2]:



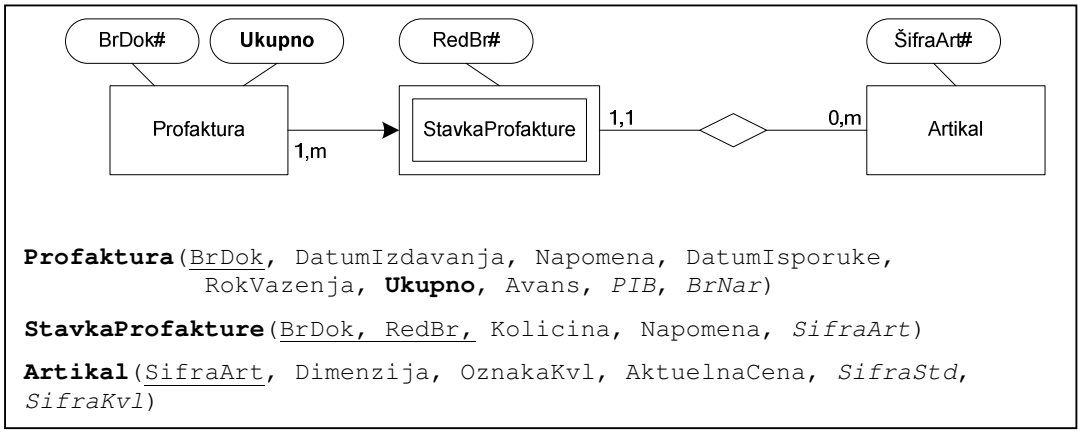
Profaktura(BrDok, DatumIzdavanja, Napomena, DatumIsporuke, RokVazenja, Avans, *PIB*, *BrNar*)

StavkaProfakture(BrDok, RedBr, Kolicina, Napomena, *SifraArt*)

Artikal(SifraArt, Dimenzija, OznakaKvl, AktuelnaCena, *SifraStd*, *SifraKvl*)

[2] Zbog preglednosti, na PMOV – u neće biti prikazivani svi atributi entiteta niti sve njihove veze.

Može se očekivati da će pri radu sa profakturom biti potrebno obračunavanje njene ukupne vrednosti, koja se dobija sumom iznosa stavki profakture (koji se dobija množenjem količine artikla i njegove aktuelne cene; atribut `Kolicina` relacije `StavkaProfakture` i atribut `AktuelnaCena` relacije `Artikal`). Kako bi se izbeglo obračunavanje ukupne vrednosti profakture pri svakom upitu, u relaciju `Profaktura` dodaje se atribut `Ukupno`, u kojem će se čuvati ukupna vrednost profakture. Prethodni konceptualni model sada bi izgledao:



Profaktura(BrDok, DatumIzdavanja, Napomena, DatumIsporuke, RokVazenja, **Ukupno**, Avans, *PIB*, *BrNar*)

StavkaProfakture(BrDok, RedBr, Kolicina, Napomena, *SifraArt*)

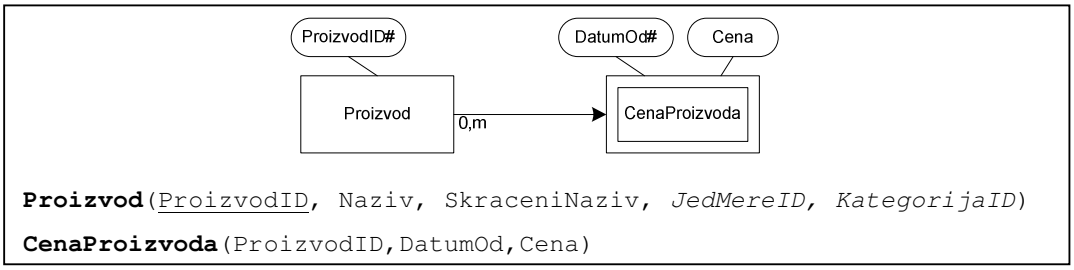
Artikal(SifraArt, Dimenzija, OznakaKvl, AktuelnaCena, *SifraStd*, *SifraKvl*)

Očigledno je da se na ovaj način izbegava obračunavanje ukupne vrednosti profakture pri svakom upitu, čime se optimizuje vreme njegovog izvršavanja. Naravno, uvođenje izvedene kolone potencijalno može dovesti do nekonzistentnosti podataka, pa je neophodno pisanje aplikacionog kôda koji će ponovo izračunavati izvedenu vrednost pri svakoj DML operaciji koja može dovesti do narušavanja konzistentnosti. Tako je, na primer, neophodno ponovo izračunati ukupnu vrednost pri svakom dodavanju nove stavke profakture. Na sličan način kao i u delu o denormalizaciji, neophodno je specificirati trigere koji će se „okidati" pri svakoj DML operaciji od uticaja na konzistentnost podataka i ponovo izračunavati ukupnu vrednost profakture[3].

4.1.1.1. End Date Column tehnika optimizacije

End Date Column tehnika optimizacije predstavlja poseban vid *Storing Derivable Values* tehnike. Ova tehnika optimizacije koristi se kada u bazi podataka postoji tabela čiji redovi predstavljaju uzastopne vremenske periode. Pretpostavimo da je dat sledeći konceptualni model:

[3] U okviru projektnog rada, zahteva sa da se aplikacioni kôd koji vrši izračunavanje izvedene vrednosti izdvoji u uskladištenu proceduru, kao i da se specificira tabela trigera koji će pozivati posmatranu uskladištenu proceduru kada je to potrebno. Detaljnije o ovome u delu o uskladištenim procedurama.



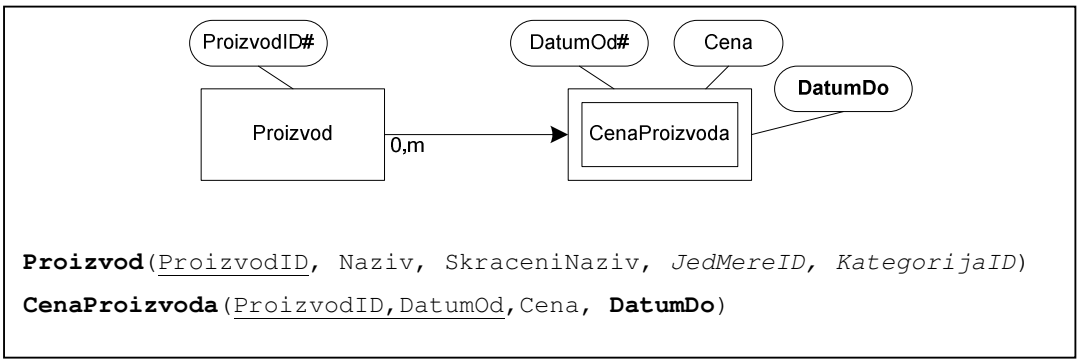
Proizvod(ProizvodID, Naziv, SkraceniNaziv, *JedMereID, KategorijaID*)
CenaProizvoda(ProizvodID,DatumOd,Cena)

Ukoliko bismo želeli da dobijemo trenutnu cenu proizvoda, odgovarajući upit bi zahtevao upotrebu podupita kojim bismo pronašli maksimalnu vrednost DatumOd manju od trenutnog datuma, kao što je prikazano ispod.

```
...WHERE proizvodid = ...
AND   datumod = (SELECT max(datumod)
                 FROM cenaproizvoda
                 WHERE datumod <= sysdate
                 AND proizvodid = ...)
```

Na sličan način bi se realizovao i upit kojim bismo želeli da pronađemo cenu proizvoda koja je važila određenog datuma (umesto sysdate koristio bi se željeni datum).

Kako bi se izbegla upotreba podupita, može se izvršiti optimizacija dodavanjem atributa `DatumDo` u relaciju `CenaProizvoda`, kao što je prikazano na sledećem konceptualnom modelu:



Proizvod(ProizvodID, Naziv, SkraceniNaziv, *JedMereID, KategorijaID*)
CenaProizvoda(ProizvodID,DatumOd,Cena, **DatumDo**)

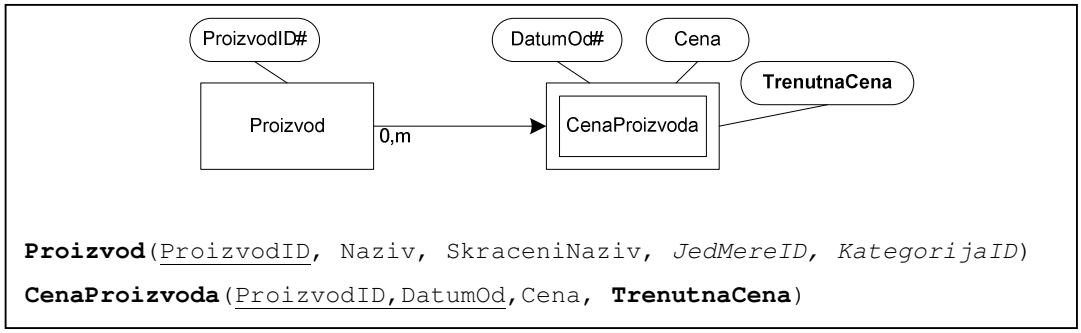
Na ovaj način, umesto upotrebe podupita, za pronalaženje trenutne cene proizvoda može se koristiti between klauzula, kao što je prikazano ispod.

```
...WHERE prozvodid = ...
AND sysdate between datumod and nvl(datumdo, sysdate)...
```

I ovde je neophodno implementirati aplikacioni kôd kojim se, na određene DML naredbe, postavlja odgovarajuća vrednost za `DatumDo`. Na primer, pri dodavanju nove cene proizvoda neophodno je prethodnoj ceni, za vrednost kolone `DatumDo`, postaviti vrednost kolone `DatumOd` nove cene.

4.1.1.2. Current Indicator Column tehnika optimizacije

Current Indicator Column tehnika optimizacije predstavlja poseban vid *Storing Derivable Values* tehnike optimizacije. Sâma tehnika koristi se u sličnim situacijama kao i prethodno opisana *End Date Column* tehnika. Razlika je u tome što se umesto atributa koji predstavlja krajnji datum, kod *Current Indicator Column* tehnike optimizacije koristi atribut koji ima ulogu indikatora da li je posmatrana cena istovremeno i trenutno aktuelna cena proizvoda. Ukoliko se posmatra početni konceptualni model naveden u prethodnom delu o *End Date Column* tehnici, *Current Indicator Column* tehnika optimizacije bi se mogla sprovesti kao što je prikazano na sledećem konceptualnom modelu:



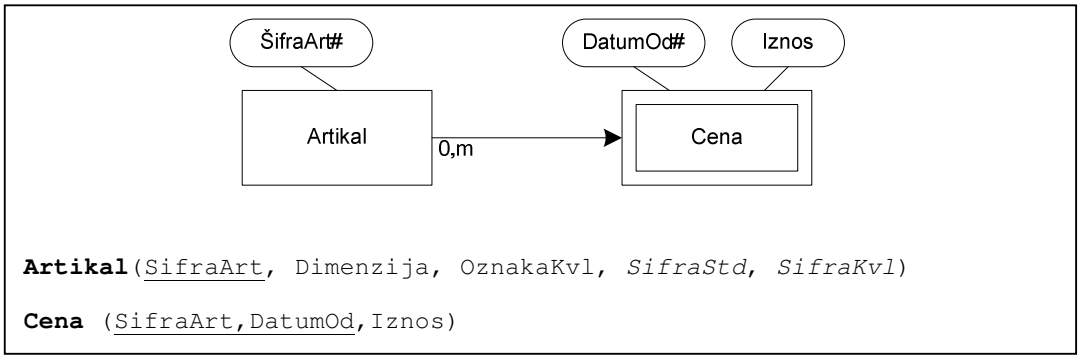
Kao što se može videti, relaciji `CenaProizvoda` dodat je atribut `TrenutnaCena` kojim se za svaku cenu definiše da li je trenutno aktuelna ili ne. U ovom slučaju, umesto ranije navedenog podupita, koristi se samo uslov definsan nad dodatim atributom, kao što je dato u nastavku.

```
...WHERE proizvodid = ...
AND trenutnacena = 'da'
```

Naravno, i ovde je neophodan aplikacioni kôd kojim se čuva konzistentnost podataka pri određenim DML operacijama (npr. pri dodavanju nove cene potrebno je ažurirati vrednosti kolone `TrenutnaCena` u novododatoj ceni i ceni koja je prethodno bila aktuelna).

4.1.2. Repeating Single Detail with Master tehnika optimizacije

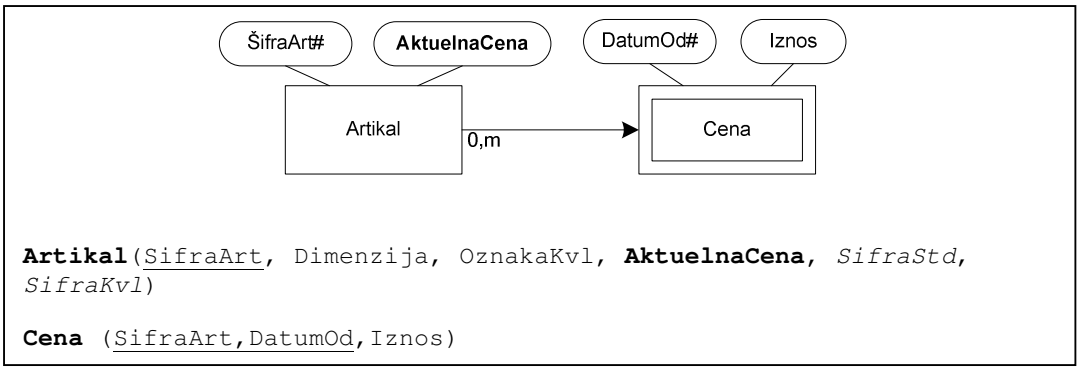
Pri čuvanju evidencije promene određenog podatka kroz vreme, čest je slučaj da će u upitima biti potrebna samo trenutna vrednost. Posmatrajmo sledeći konceptualni model u kom se za artikal čuvaju sve cene koje je imao kroz vreme:



Artikal(SifraArt, Dimenzija, OznakaKvl, *SifraStd*, *SifraKvl*)

Cena (SifraArt,DatumOd,Iznos)

Može se pretpostaviti da će u upitima najčešće biti potrebna aktuelna cena proizvoda. U dve prethodno navedene tehnike optimizacije, ovaj problem rešavan je upotrebom krajnjeg datuma važenja cene (*End Date Column* tehnika) i upotrebom indikatora trenutne cene na samoj ceni (*Current Indicator Column* tehnika). U slučaju *Repeating Single Detail with Master* tehnike optimizacije, redundansa se uvodi u master tabelu. Drugačije rečeno, u ovom primeru, redundantan atribut dodaje se u relaciju `Artikal`, kao što je prikazano na sledećem konceptualnom modelu.



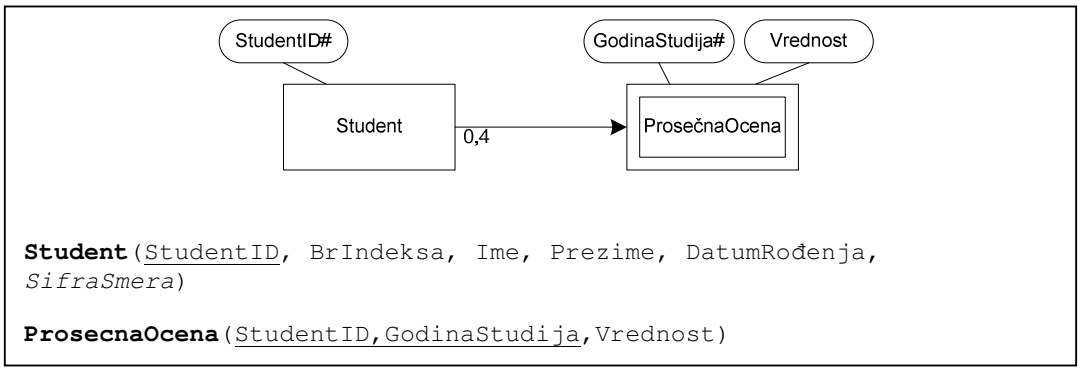
Artikal(SifraArt, Dimenzija, OznakaKvl, **AktuelnaCena**, *SifraStd*, *SifraKvl*)

Cena (SifraArt,DatumOd,Iznos)

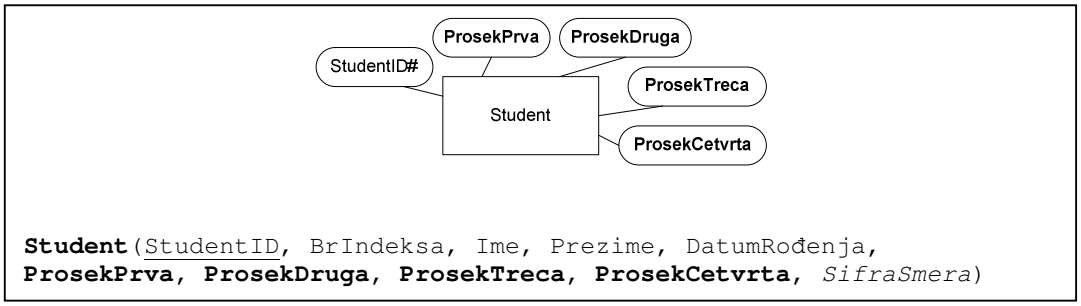
Relaciji `Artikal` dodaje se atribut `AktuelnaCena` koji će u sebi sadržati trenutno aktuelnu cenu proizvoda. Kao i u prethodnim slučajevima, neophodno je aplikacionim kôdom čuvati konzistentnost podataka (na primer, pri svakom dodavanju nove cene artikla, neophodno je ažurirati vrednost aktuelne cene na artiklu kako bi odgovarala vrednosti nove cene).

4.1.3. Keeping Details with Master tehnika optimizacije

U slučaju kada za svaki red master tabele postoji fiksan broj redova detail tabele, moguće je primeniti *Keeping Details with Master* tehniku optimizacije koja podrazumeva „prebacivanje“ kolona detail tabele u master tabelu. Pretpostavimo da je dat sledeći konceptualni model:



Kao što se može videti, za svakog studenta vodi se evidencija prosečnih ocena koje je imao u svakoj godini studija. Očigledno je da svaki student može imati maksimalno četiri prosečne ocene. Usled toga, moguće je u relaciju student dodati četiri atributa, po jedan za svaku godinu studija, kao što je prikazano na sledećem konceptualnom modelu.

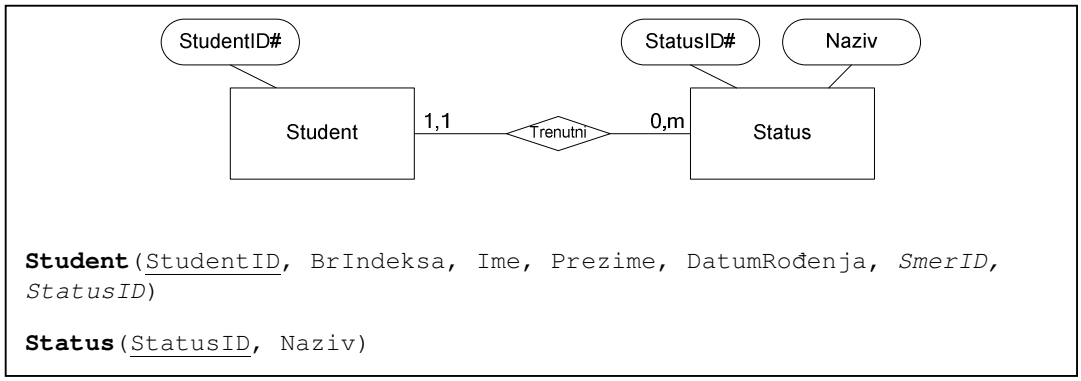


Dakle, master tabeli (u ovom primeru Student) dodaju se nove kolone koje reprezentuju moguće redove detail tabele (u ovom primeru ProsecnaOcena), pri čemu detail tabela nestaje, obzirom da više nema potrebe za njom.

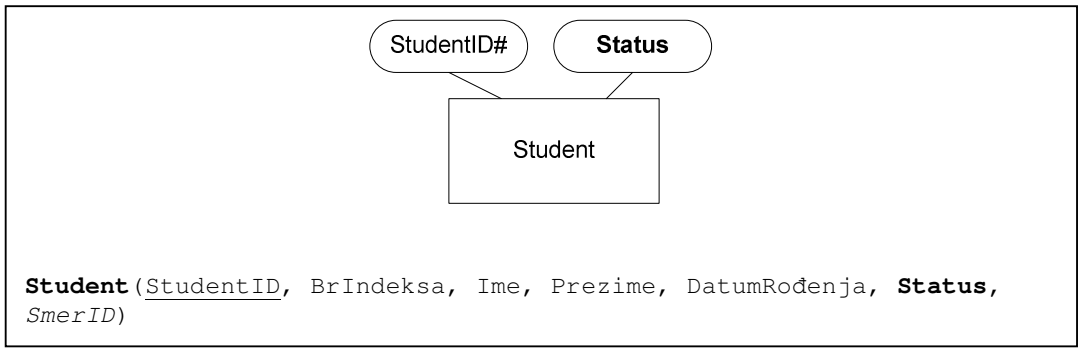
4.1.4. Hard – Coded Values tehnika optimizacije

U slučaju kada u bazi podataka postoji tabela, referencirana od strane neke druge tabele, pri čemu referencirana tabela u sebi sadrži relativno konstantan broj redova, moguće je primeniti *Hard – Coded Values* tehniku optimizacije. Ova tehnika podrazumeva da se vrednosti referencirane tabele izraze upotrebom aplikacionog kôda, pri čemu referencirana tabela nestaje, dok se u referencirajuću

tabelu dodaje atribut koji zamenjuje referenciranu tabelu. Pretpostavimo da je dat sledeći konceptualni model:



Ako se uzme u obzir da relacija `Status` ima samo dve n – torke, obzirom da student može biti u statusu samofinansirajućeg ili budžetskog studenta, može se izvršiti *Hard – Coded Values* optimizacija, pri čemu se dobija sledeći konceptualni model:



Kao što se može videti **relacija** `Status` je nestala, pri čemu je relaciji `Student` dodat novi **atribut** `Status` koji će se koristiti za čuvanje statusa u kom se student trenutno nalazi. Sâme vrednosti koje je sadržala relacija `Status` (samofinansirajući i budžetski), sada se nalaze u aplikacionom kôdu. Na primer, mogu se definisati kao CHECK ograničenje nad dodatim atributom `Status` relacije `Student`.

4.2. Indeksi

Indeksi predstavljaju strukture podataka koje omogućavaju efikasan pristup podacima koji se nalaze u bazi podataka. Dva osnovna razloga upotrebe indeksa su:

- Poboljšanje performansi – indeksi pružaju način za brzo pristupanje podacima.
- Osiguranje jedinstvenosti – jedinstveni indeks je struktura koja na efikasan način osigurava da se vrednosti u kolonama koje su indeksirane ne ponavljaju. U skladu sa tim, većina SUBP – a automatski kreira indekse za sve primarne ključeve.

Pored primarnih ključeva koji se najčešće automatski indeksiraju, kolone kandidati za kreiranje indeksa nad njima bile bi:

- Kolone koje se u upitima često koriste u JOIN klauzuli
- Kolone koje sadrže širok opseg vrednosti
- Kolone koje se često koriste u WHERE klauzuli
- Kolone koje se često koriste u ORDER BY klauzuli

Kolone koje nisu dobri kandidati za indeksiranje su:

- Kolone koje se retko koriste u WHERE ili ORDER BY klauzuli
- Kolone koje pripadaju tabelama male veličine
- Kolone koje sadrže veliki broj null vrednosti, u slučaju kada se traže redovi koji sadrže upravo null vrednosti

Pretpostavimo da je data sledeća relacija:

```
Racun (SifraRacuna, Datum_izdavanja_racuna, Mesto_izdavanja,
Sifra_radnika, Sifra_valute, Sifra_metoda_isporuke)
```

Može se pretpostaviti da će se u upitima kao kriterijum pretrage često koristiti atribut `Datum_izdavanja_racuna`. Zbog toga je nad kolonom `Datum_izdavanja_racuna` tabele `Racun` potrebno kreirati indeks kako bi se ubrzalo izvršavanje upita nad tom tabelom.

Opšta sintaksa za kreranje indeksa je:

```
create [unique] index <naziv_indeksa>
on <naziv_tabele> (<naziv_kolone1> [,<naziv_kolone2>, …])
```

Navedena sintaksa za kreiranje indeksa može se koristiti u različitim SUBP, pri čemu svaki od SUBP definiše i dodatne klauzule za definisanje različitih vrsta indeksa ili indeksa sa različitim karakteristikama, što ovde neće biti razmatrano.

U skladu sa prethodno navedenom opštom sintaksom, kreiranje indeksa nad kolonom `Datum_izdavanja_racuna` kreiralo bi se naredbom:

```
create index racun_ind on racun (datum_izdavanja_racuna);
```

4.3. Vertikalno particionisanje

Particionisanje tabele podrazumeva podelu jedne tabele u dve ili više manjih tabela kako bi se smanjila količina podataka koju SUBP mora da obradi pri izvršavanju upita. Vertikalno particionisanje tabele podrazumeva podelu kolona posmatrane tabele u dve tabele, pri čemu se u jednu tabelu izdvajaju kolone koje se češće koriste u upitima, dok se preostale kolone, kolone koje se ređe koriste, smeštaju u drugu tabelu. Podela kolona može se izvršiti i u više od dve tabele, pri čemu se tada kolone grupišu po stepenu njihove upotrebe u upitima. U svakoj od tabela dobijenih vertikalnim particionisanjem mora se naći primarni ključ originalne tabele, obzirom da će on predstavljati „vezu“ između redova particionisanih tabela.

Pretpostavimo da je data sledeća relacija:

Kupac(ŠifraKupca, NazivKupca, RačunKupca, TelefonKupca, FaksKupca, MatičniBroj, PIB, Adresa, *IDMesta*)

Može se pretpostaviti da se u upitima, pored primarnog ključa `ŠifraKupca`, često koriste atributi `NazivKupca` i `Adresa`, pa se oni mogu izdvojiti u jednu tabelu, dok se ostali atributi izdvajaju u drugu tabelu. Tako se dobijaju sledeće dve relacije:

Kupac(ŠifraKupca, NazivKupca, Adresa)

Kupac_Detalji(ŠifraKupca, RačunKupca, TelefonKupca, FaksKupca, MatičniBroj, PIB, *IDMesta*)

Za tabele dobijene vertikalnim particionisanjem potom se definiše pogled kojim se podaci iz tih tabela objedinjuju, faktički kreirajući strukturu početne tabele. Takođe, upotrebom pogleda od krajnjeg korisnika se “skriva” izvršeno vertikalno particionisanje i potreba za spajanjem dobijenih tabela kako bi se dobila početna tabela. Pogled kojim se spajaju podaci iz tabela `Kupac` i `Kupac_Detalji` i stvara privid originalne tabele `Kupac` dat je u nastavku:

```
CREATE OR REPLACE VIEW KUPAC_VIEW
AS
SELECT     K.SifraKupca,     K.NazivKupca,     K.Adresa,     KD.RacunKupca,
KD.TelefonKupca, KD.FaksKupca, KD.MaticniBroj, KD.PIB, KD.IDMesta
FROM KUPAC K, KUPAC_DETALJI KD
WHERE K.SifraKupca = KD.SifraKupca;
```

Nad ovako definisanim pogledom potom se može definisati INSTEAD OF triger kojim se dodavanje reda nad pogledom pretvara u dodavanje redova u tabele dobijene vertikalnim particionisanjem, pri čemu se u svaku od tabela dodaju vrednosti koje odgovaraju kolonama koje se nalaze u tim tabelama (u prethodnom primeru, dodavanje reda u pogled `Kupac_view` pretvara se u dodavanje redova u tabele `Kupac` i `Kupac_Detalji`, pri čemu se u tabelu `Kupac` insertuju vrednosti kolona `ŠifraKupca`, `Nazivkupca` i `Adresa`, dok se ostale vrednosti insertuju u tabelu `Kupac_Detalji`).

4.4. Uskladištene Procedure

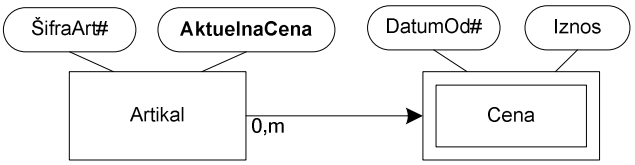
Modularizacija je proces kojim se veliki blokovi kôda razbijaju na manje delove (module). pri ćemu moduli mogu koristiti druge module. Pod procedurom se podrazumeva modul koji može izvršiti jednu ili više aktivnosti. Poziv procedure je nezavisna izvršna naredba u SQL-u i zbog toga SQL blok može sadržati samo poziv procedure. Procedure su glavni delovi modularnog kôda, koji omogućavaju konsolidaciju i ponovnu upotrebu programske logike. Procedure uskladištene u bazi nazivaju se uskladištene (stored) procedure. Procedure se definišu sledećom sintaksom:

```
CREATE [OR REPLACE] PROCEDURE [schema.] naziv_procedure [(parametar [,
parametar ...])]
AS
[deklaracija_lokalnih_promenljivih]
BEGIN
izvršne_naredbe
[EXCEPTION excption_handlers]
END [naziv_procedure];
```

Procedura ne mora imati parametre, a može ih imati neograničeno mnogo (ulazni - IN, izlazni - OUT ili ulazno/izlazni IN/OUT). Svaka procedura se sastoji iz sledećih delova: zaglavlja, koje dolazi pre rezervisane reči AS (umesto AS moguće je koristiti i rezervisanu reč IS sa istim efektom); ključne reči PROCEDURE iza koje sledi naziv procedure i (opciono) spisak parametara i telo procedure, koje čine sve reči nakon rezervisane reči AS. Reč REPLACE je opciona i omogućava da se postojeća procedura sa određenim nazivom izmeni, tj. da se promeni kôd postojeće procedure. Podrazumevana šema je šema trenutnog korisnika. Ukoliko se navede naziv druge šeme, ta šema će sadržati kreiranu proceduru. Da bi korisnik bio u mogućnosti da kreira porceduru, nephodno je da ima privilegije kreiranja procedure u navedenoj šemi. Nakon rezervisane reči AS sledi deo za deklarisanje lokalnih promenljivih. Ukoliko ih procedura ne koristi, iza rezervisane reči AS sledi rezervisana reč BEGIN. Nakon ključne reči BEGIN sledi obavezni deo procedure: izvršne naredbe, koje će biti izvršene svaki put kada je procedura pozvana. Mora postojati barem jedna naredba. Iza izvršne naredbe opciono sledi rezervisana reč EXCEPTION nakon koje se definiše način upravljanja izuzecima. Na kraju tela procedure sledi rezervisana reč END iza koje se, opciono, navodi naziv procedure.

4.4.1. Primer uskladištene procedure: Aktuelna cena

U ovom delu, biće prikazana i detaljno opisana dva primera uskladištenih procedura. Pogledajmo, još jednom, dati primer za Repeating Single Detail with Master tehniku optimizacije. U slučaju *Repeating Single Detail with Master* tehnike optimizacije, redundansa se uvodi u master tabelu. U ovom primeru, redundantan atribut dodaje se u relaciju `Artikal`, kao što je prikazano na sledećem konceptualnom modelu.



Artikal(SifraArt, Dimenzija, OznakaKvl, **AktuelnaCena**, *SifraStd*, *SifraKvl*)

Cena (SifraArt,DatumOd,Iznos)

Relaciji `Artikal` dodaje se atribut `AktuelnaCena` koji će u sebi sadržati trenutno aktuelnu cenu proizvoda. Neophodno je aplikacionim kôdom čuvati konzistentnost podataka (na primer, pri svakom dodavanju nove cene artikla, neophodno je ažurirati vrednost aktuelne cene na artiklu kako bi odgovarala vrednosti nove cene). Tabela za specifikaciju trigera koji će pozivati uskladištenu proceduru sledi:

Tabela	Tip trigera	Kolona	Potreban	Šta treba da uradi?
Proizvod	Insert		NE	
	Update	AktuelnaCena	DA	Zabrana direktnog ažuriranja polja AktuelnaCena u tabeli Proizvod
	Delete		NE	
CenaProizvoda	Insert		DA	Prilikom unosa nove CeneProizvoda, okida se triger koji poziva proceduru za ažuriranje polja AktuelnaCena na Proizvodu
	Update	Cena	DA	Prilikom ažuriranja CeneProizvoda, okida se triger koji poziva proceduru za ažuriranje polja AktuelnaCena na Proizvodu
	Delete		DA	Prilikom brisanja CeneProizvoda, okida se triger koji poziva proceduru za ažuriranje polja AktuelnaCena na Proizvodu

Prikazanom tabelom se specificiraju potrebni trigeri za navedeni primer uskladištene procedure. Biće potrebno kreirati trigere koji će se aktivirati kada je izdata naredba ažuriranja rekorda u tabeli Proizvod. Implementacija ovog trigera omogućava da se uvede zabrana ažuriranja polja AktuelnaCena u samom proizvodu. Vrednost ove kolone je, samim tim, moguće menjati isključivo pozivom procedure za postavljanje vrednosti AktuelneCene. Pored ovog, potrebno je kreirati trigere koji će se okidati kada se vrši unos novog rekorda tabele CenaProizvoda, vrši brisanje postojećeg rekorda tabele CenaProizvoda, ili ažurira postojeći rekord. U aplikacijskom kôdu, identifikovani trigeri mogu biti zasebni, a isto tako funkcionalnost ovih trigera može biti integrisana u kôd jednog trigera. U nastavku sledi kôd trigera korišćenih za poziv procedure za određivanje i postavljanje aktuelne cene. Prvi triger određuje šifru proizvoda i postavlja je u promenljivu:

```
CREATE OR REPLACE TRIGGER  "AKTUELNAC"
BEFORE INSERT OR UPDATE OR DELETE ON cenaProizvoda
FOR EACH ROW
BEGIN
 IF (INSERTING OR UPDATING) THEN
      BEGIN
         cenap.sifra := :NEW.sifraproizvoda;
      END;
 ELSE
     BEGIN
         cenap.sifra := :OLD.sifraproizvoda;
     END;
 END IF;
END;
```

Triger za pozivanje procedure:

```
CREATE OR REPLACE TRIGGER  "AKTUELNAC2"
AFTER INSERT OR UPDATE OR DELETE ON cenaProizvoda
DECLARE
   s NUMBER:=cenaP.sifra;
BEGIN
   aktuelnacena(s);
END;
```

U ovom primeru je dat kôd trigera koji se okida i za unos i za ažuriranje i za brisanje nad tabelom cenaProizvoda. U prvom trigeru se šifra proizvoda, za koji se vrši unos, ažuriranje ili brisanje, postavlja u globalnu promenljivu, pre izvršenja samih naredbni unosa, ažuriranja ili brisanja. Nakon izvršenja date naredbe, okida se drugi triger koji poziva proceduru aktuelnacena(s) i prosleđuje joj s kao konkretnu šifru proizvoda. U nastavku sledi kôd procedure za određivanje aktuelne cene:

```
CREATE OR REPLACE PROCEDURE  "AKTUELNACENA" (SifraPro IN NUMBER) AS
  aktCena PROIZVOD.AKTUELNACENA%type;
BEGIN
 aktCena:=0;
SELECT cena INTO aktCena FROM CenaProizvoda
```

```
  WHERE sifraProizvoda=SifraPro and datumOd=(select max(DATUMOD) from
cenaProizvoda where sifraProizvoda=SifraPro and datumOd<=sysdate);

  UPDATE Proizvod
  SET AktuelnaCena = aktCena
  WHERE SifraProizvoda=SifraPro;

END;
```

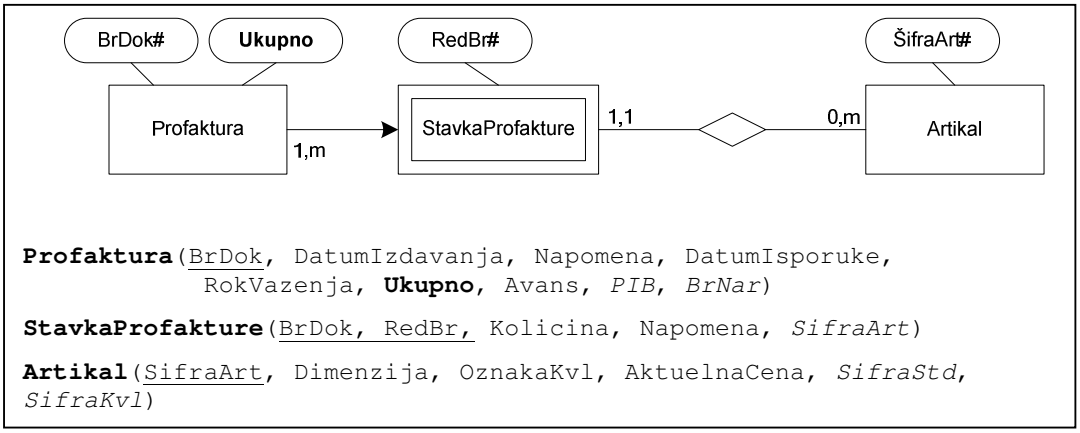
Prikazana procedura prima kao ulazni (IN) parametar šifru proizvoda (tipa NUMBER) za koji se računa aktuelna cena. Vrednost aktuelne cene se postavlja u lokalnu promenljivu aktCena. U ovoj proceduri koristi se SQL funkcija MAX(DATUMOD) koja određuje „najnoviji“ datum. Za taj datum se uzima vrednost cene i postavlja u promenljivu. U uslovu se proverava da li je šifra prozivoda ista kao i u ulaznom parametru i da li je datum od kada važi cena manji od trenutnog. Ovim se ne uzimaju u obzir buduće cene, tj. cene čiji datum početka važenja još nije nastupio. Na samom kraju procedure vrši se ažuriranje tabele Proizvod, preciznije samo kolone sa aktuelnom cenom. U nastavku je prikazana procedura iste funkcionalnosti, kreirana u Microsoft SQL Sever-u (T/SQL):

```
CREATE PROCEDURE [dbo].[sp_AktuelnaCena]
@SifraPro int = 0
AS
BEGIN
        SET NOCOUNT ON;
declare @AktuelnaCena int;
        SELECT @AktuelnaCena = Iznos FROM Cena
  WHERE SifraPro=@SifraPro and DatumOd=(select max(DATUMOD) from cena where
SifraPro=@SifraPro and datumod<=getdate());

UPDATE Arikal
  SET AktuelnaCena = @AktuelnaCena
  WHERE SifraPro=@SifraPro;
END
GO
```

4.4.2. Primer uskladištene procedure: Ukupan iznos profakture

Pogledajmo, još jednom, dati primer za Storing Derivable Values tehniku optimizacije. Može se očekivati da će pri radu sa profakturom biti potrebno obračunavanje njene ukupne vrednosti, koja se dobija sumiranjem iznosa stavki profakture (množenjem količine artikla i njegove aktuelne cene; `Kolicina` je iz relacije `StavkaProfakture`, a `AktuelnaCena` iz relacije `Artikal`). Kako bi se izbeglo obračunavanje ukupne vrednosti profakture pri svakom upitu, u relaciju `Profaktura` dodaje se atribut `Ukupno`, u kojem će se čuvati ukupna vrednost profakture. Početni konceptualni model nakon optimizacije izgleda:



Profaktura(BrDok, DatumIzdavanja, Napomena, DatumIsporuke, RokVazenja, **Ukupno**, Avans, *PIB*, *BrNar*)

StavkaProfakture(BrDok, RedBr, Kolicina, Napomena, *SifraArt*)

Artikal(SifraArt, Dimenzija, OznakaKvl, AktuelnaCena, *SifraStd*, *SifraKvl*)

Očigledno je da se na ovaj način izbegava obračunavanje ukupne vrednosti profakture pri svakom upitu, čime se optimizuje vreme njegovog izvršavanja. Naravno, uvođenje izvedene kolone potencijalno može dovesti do nekonzistentnosti podataka, pa je neophodno pisanje aplikacionog kôda koji će ponovo izračunavati izvedenu vrednost pri svakoj DML operaciji koja može dovesti do narušavanja konzistentnosti. Tako je, na primer, neophodno ponovo izračunati ukupnu vrednost pri svakom dodavanju nove stavke profakture. Na sličan način kao i u delu o denormalizaciji, neophodno je specificirati trigere koji će se „okidati" pri svakoj DML operaciji od uticaja na konzistentnost podataka i ponovo izračunavati ukupnu vrednost profakture. Tabela za specifikaciju trigera koji će pozivati uskladištenu proceduru sledi:

Tabela	Tip trigera	Kolona	Potreban	Šta treba da uradi?
Profaktura	Insert		NE	
	Update	Ukupno	DA	Zabrana direktnog ažuriranja polja Ukupno
	Delete		NE	
StavkagaProfakture	Insert		DA	Prilikom unosa nove StavkeProfakture, okida se triger koji poziva proceduru za ažuriranje polja Ukupno na Profakturi
	Update	Kolicina	DA	Prilikom ažuriranja nove StavkeProfakture, okida se triger koji poziva proceduru za ažuriranje polja Ukupno na Profakturi
	Delete		DA	Prilikom brisanja

				StavkeProfakture, okida se triger koji poziva proceduru za ažuriranje polja Ukupno na Profakturi

Prikazanom tabelom se specificiraju potrebni trigeri za navedeni primer uskladištene procedure. Biće potrebno kreirati trigere koji će se aktivirati kada je izdata naredba ažuriranja rekorda u tabeli Profaktura. Implementacija ovog trigera omogućava da se uvede zabrana ažuriranja polja Ukupno u samoj Profakturi. Vrednost ove kolone je, samim tim, moguće menjati isključivo pozivom procedure za postavljanje vrednosti Ukupno na Profakturi. Pored navedenog, potrebno je kreirati trigere koji će se okidati kada se vrši unos novog rekorda tabele StavkaProfakture, vrši brisanje postojećeg rekorda tabele StavkaProfakture, ili ažurira postojeći rekord. U aplikacijskom kôdu, identifikovani trgeri mogu biti zasebni, a isto tako funkcionalnost ovih trigera može biti integrisana u kôd jednog trigera. U nastavku sledi kôd trigera korišćenih za poziv procedure za određivanje i postavljanje ukupnog iznosa profakture. Prvi triger određuje šifru profakture i postavlja je u promenljivu:

```
CREATE OR REPLACE TRIGGER  "SUMASP"
BEFORE INSERT OR UPDATE OR DELETE ON stavkaProfakture
FOR EACH ROW
BEGIN
   IF (INSERTING OR UPDATING) then
      BEGIN
         stavkap.brdok := :NEW. brdok;
      END;
   ELSE
      BEGIN
         stavkap.brdok:= :OLD. brdok;
      END;
   END IF;
END;
```

Triger za pozivanje procedure:

```
CREATE OR REPLACE TRIGGER  "SUMASP2"
AFTER INSERT OR UPDATE OR DELETE ON stavkaProfakture
DECLARE
   dok NUMBER:=stavkap.brdok;
BEGIN
   sumaStavkiProfakture(dok);
END;
```

U ovom primeru je dat kôd trigera koji se okida za unos, za ažuriranje i za brisanje nad tabelom stavkaProfakture. U prvom trigeru se BrDok (šifra profakture), za koji se vrši unos, ažuriranje ili brisanje, postavlja u globalnu promenljivu brdok, pre izvršenja samih naredbi unosa, ažuriranja ili brisanja. Nakon izvršenja date naredbe, okida se drugi triger koji poziva proceduru

sumaStavkiProfakture(dok) i prosleđuje joj dok kao konkretnu šifru profakture. U nastavku sledi kôd procedure za određivanje ukupnog iznosa profakture:

```
CREATE OR REPLACE PROCEDURE  "SUMASTAVKIPROFAKTURE" (sifDok IN NUMBER) AS
  suma NUMBER:=0;
BEGIN
  SELECT SUM(p.aktuelnacena*sp.kolicina) INTO suma
  FROM      proizvod      p      join      stavkaprofakture      sp      on
(p.sifraproizvoda=sp.sifraproizvoda)
  WHERE brdok=SifDok;
UPDATE Profaktura
SET Ukupno=suma
WHERE brdok=SifDok;
END;
```

Prikazana procedura prima kao ulazni (IN) parametar šifru profakture (tipa NUMBER) za koji se računa ukupan iznos vrednosti na stavkama. Suma pomnoženih vrednosti količine tabele stavkaProfakture i aktuelneCene tabele Proizvod smešta se u promenljivu suma. Zatim, vrši se ažuriranje tabele Profaktura, tj. njene kolone Ukupno sa vrednošću iz promenljive suma. U uslovu se proverava da li je šifra pofakture ista kao i u ulaznom parametru. U nastavku je prikazana procedura iste funkcionalnosti, kreirana u Microsoft SQL Sever-u (T/SQL):

```
CREATE PROCEDURE [dbo].[sp_SumaStavkiProfakture]

@BrDok int = 0

AS
BEGIN
        SET NOCOUNT ON;

        declare @ukupno numeric(18,0);
        SELECT @ukupno =
SUM(p.aktuelnacena*sp.kolicina)
FROM Arikal p inner join stavkaprofakture sp on p.sifrapro=sp.sifrapro
WHERE brdok=@BrDok;

UPDATE Profaktura
SET Ukupno = @ukupno
WHERE Profaktura.BrDok = @BrDok
END
```

Reference

Tehnike optimizacije zasnovane na izvedenim i redundantnim podacima obrađuju se u [1], [2] i [3]. Vertikalno particionisanje objašnjava se u [2], [4], [5] i [6], dok se indeksi u užem ili širem obimu obrađuju u svim navedenim referencama. Procedure su objašnjenjene u [9 - 14].

1. Whitehorn M., Marklyn B., *„Inside Relational Databases with Examples in Access“*, Springer, 2007.
2. Mullins C., *„Database Administration: The Complete Guide to Practices and Procedures“*, Addison Wesley, 2002.
3. Connoly T., Begg C., *„Database Solutions: A step-by-step guide to building databases“*, Pearson Education Limited, 2004.
4. Harrington J., *„Relational Database Design and Implementation“*, Morgan Kaufmann, 2009.
5. Lightstone S., Teorey T., Nadeau T., *„Physical Database Design“*, Morgan Kaufmann, 2007.
6. Buxton S., et al., *„Database Design: Know it all“*, Morgan Kaufmann, 2009.
7. Powell G., *„Beginning Database Design“*, Wiley Publishing, 2006.
8. Hoffer J., Prescott M., McFadden F., *„Modern Database Management“*, Pearson Education, 2007.
9. Feuerstein S, Pribyl B *„Oracle PL/SQL Programming“* O'Reilly, 2009.
10. Rosenzweig B, Rakhimov E *„Oracle PL/SQL by Example“* Prentice Hall, 2009.
11. Whitehorn M, Marklyn B, *„Inside Relational Databases with Examples in Access“*, Springer, 2007.
12. Mullins C., *„Database Administration: The Complete Guide to Practices and Procedures“*, Addison Wesley, 2002.
13. Lightstone S., Teorey T., Nadeau T., *„Physical Database Design“*, Morgan Kaufmann, 2007.
14. Buxton S., et al., *„Database Design: Know it all“*, Morgan Kaufmann, 2009.