UNIVERZA V LJUBLJANI
FAKULTETA ZA MATEMATIKO IN FIZIKO

Matematika – Praktična matematika (VSŠ)

Mateja Lombar

USKLAJEVANJE DVEH PODATKOVNIH BAZ

Diplomska naloga

Ljubljana, 2009

DIPLOMSKA NALOGA :
FAKULTETA ZA MATEMATIKO IN FIZIKO

ZAHVALA

Ob zaključku diplomske naloge se lepo zahvaljujem mentorju mag. Matiji Lokarju ter somentorju Marku Poljancu iz podjetja Spin d.o.o. za strokovno pomoč, nasvete in potrpežljivost v času pripravljanja diplomske naloge.

Diplomsko nalogo posvečam svojim staršem, saj so mi omogočili študij in mi v času pripravljanja diplomske naloge vedno stali ob strani.

DIPLOMSKA NALOGA :
FAKULTETA ZA MATEMATIKO IN FIZIKO

PROGRAM DELA

V diplomski nalogi obdelajte konkretni primer, s katerim pokažete, kako lahko s pomočjo programa v jeziku PL/SQL uskladite dele dveh različic podatkovnih baz. Pri tem predstavite osnovne značilnosti jezika PL/SQL in obravnavajte določene pojme, povezane z bazami podatkov, kot so na primer paketi in prožilci.

mentor

mag. Matija Lokar

somentor

Marko Poljanec

DIPLOMSKA NALOGA :
FAKULTETA ZA MATEMATIKO IN FIZIKO

POVZETEK

V diplomski nalogi sta opisana dva načina izvedbe programa, s katerim nadgradimo oziroma posodobimo strankino podatkovno bazo z novimi in posodobljenimi podatki.

Diplomska naloga predpostavlja osnovno poznavanje programskega jezika java in poznavanje osnov strukturiranega poizvedovalnega jezika SQL. Poleg dveh načinov izvedbe programa bomo predstavili tudi sistem Oracle in jezika SQL in PL/SQL, ki ju ta sistem uporablja. V zaključku diplomske naloge pa bomo razložili, kako bi lahko program še dodatno nadgradili.

Math.Subj.Class(2000): 68N15, 68P15, 68Q32
Computer review Class.system(1998): D.3.0, D.3.1, D.3.3, E.1

Ključne besede: sistem Oracle, jezik SQL, jezik PL/SQL, programske enote, procedure itd.
Keywords: system Oracle, language SQL, language PL/SQL, program units, procedurs, etc.

KAZALO VSEBINE

1 OPIS PROBLEMA 6

2 SISTEM ORACLE 8

2.1 Jezik SQL 8
2.1.1 Združevanje tabel 8

2.2 Jezik PL/SQL 12
2.2.1 Operatorji 12
2.2.2 Spremenljivke 12
2.2.2.1 Podatkovni tipi spremenljivk 13
2.2.3 Vgrajene funkcije 18
2.2.4 Stavek IF 21
2.2.5 Zanke 21
2.2.6 Programske enote 24
2.2.6.1 Podprogrami 24
2.2.6.2 Paket 26
2.2.6.3 Prožilec 28
2.2.7 Paket UTL_FILE 29

3 IDEJA PROGRAMA 31

3.1 Koda programa 34
3.1.1 Glava paketa 34
3.1.2 Telo paketa 34

4 IZBOLJŠAVA PROGRAMA 38

4.1 Slovar podatkov 38

4.2 Paket DBMS_SQL 41

4.3 Ideja programa 46

4.4 Koda programa 49
4.4.1 Glava paketa 49
4.4.2 Telo paketa 50

5 ZAKLJUČEK 57

LITERATURA 58

1 OPIS PROBLEMA

V podjetju Spin d.o.o., v katerem sem opravila delovno prakso, se ukvarjajo z razvijanjem sistema za obračunavanje plač za različna slovenska podjetja (v diplomi jih bomo poimenovali kar stranke). V primeru spremembe zakonodaje, spremembe pravil itd., mora podjetje (v nadaljevanju bomo zanj uporabili izraz razvijalec) pripraviti novo verzijo sistema in jo nato poslati stranki.

Za razvijanje in uporabo sistema sta potrebni dve bazi. Razvijalec, ki razvija sistem, uporablja tako imenovano razvojno bazo, v kateri so določene tabele s pravimi podatki in druge (npr. s podatki o osebah) s testnimi podatki. Stranka, ki sistem uporablja, pa ima svojo bazo, v kateri imajo vse tabele prave podatke. Bazi sta enaki po zgradbi in deloma različni po vsebini.

Tabele v strankini bazi delimo v dve skupini (skupino A in B), glede na to, kdo je zadolžen za nadgradnjo tabel (popravljanje in dodajanje podatkov v tabele). To je bodisi razvijalec bodisi stranka. Oglejmo si primere za posamezno skupino tabel.

- SKUPINA A (tabele, ki jih nadgrajuje stranka):
  - Tabela z osnovnimi podatki o delavcih.
  - Tabela s profili (nazivi) delavcev
  - Tabela s podatki o dopustih …

- SKUPINA B (tabele, ki jih nadgrajuje razvijalec):
  - Tabela s formulami za obračunavanje plač.
  - Tabela z nazivi uradnih oseb.
  - Tabela s prevodi (v primeru strank iz tujine) …

Tudi tabele v razvojni bazi na enak način delimo na skupini A in B, le da tabele skupine A vsebujejo testne podatke (in jih seveda tudi nadgrajuje razvijalec, kar pa običajno sploh ni potrebno). Opisali bomo način, kako zagotoviti, da se na osnovi sprememb tabel skupine B te spremembe opravijo tudi v strankini bazi.

Ko razvijalec pripravlja novo verzijo sistema, lahko v razvojni bazi spremeni ali doda vrednost v tabelah skupine B. Takih tabel je trenutno v bazi 9. Zato je potrebno za pravilno delovanje nove verzije sistema pri stranki poskrbeti za nadgradnjo teh tabel v strankini bazi. S spodnjim primerom si poglejmo, kako to naredimo »ročno«.

PRIMER:

Denimo, da je v zakonodaji prišlo do določenih sprememb, zato smo pripravili novo verzijo sistema za obračun plač.

Preden novo verzijo sistema pošljemo stranki, moramo preveriti, če smo med tem v tabelah skupine B (v razvojni bazi) kakšen podatek spremenili ali dodali. Denimo, da smo dodali neko formulo. V tabeli formul (poimenovana kp1094_formule) je torej prišlo do sprememb. Da lahko opazimo spremembe, imajo vse tabele določen stolpec, v katerem je za vsak zapis (vrstico) shranjen datum zadnje spremembe te vrstice. Stolpec je poimenovan TR0000_DAT_AZUR. Pove nam, kdaj je v določeni vrstici prišlo do spremembe (ali je bil podatek dodan ali popravljen). Ne vemo pa, koliko podatkov v vrstici se je spremenilo in ne kateri med njimi so bili to.

Poiščemo tiste zapise (vrstice) v tej tabeli, ki so se spremenili po datumu, ko je bila končana prejšnja verzija sistema. To smo stranki poslali na primer 20.09.2008. Da izvemo, v kateri vrstici tabele formul je prišlo do spremembe, izvedemo stavek SELECT (poizvedbo). Ta stavek tvorimo z jezikom SQL, ki je na kratko predstavljen v razdelku 2.1.

```
SELECT *
  FROM kp1094_formule
 WHERE tr0000_dat_azur >= TO_DATE('20.09.2008','DD.MM.YYYY');
```

Zgornja poizvedba vrne tabelo s tistimi vrsticami (v našem primeru je to le ena vrstica), v katerih je v tabeli formul prišlo do spremembe. Na Sliki 1in Sliki 2 vidimo, da je vrstica, kjer je vrednost stolpca ID1094_FORMULA enaka 0003, edina, ki se je spremenila po datumu prejšnje verzije sistema za obračun plač. V tej vrstici je bil torej nek podatek spremenjen, oziroma v našem primeru dodan. Da je bil dodan, sklepamo, ker sta datum kreiranja, vsebovan v stolpcu TR0000_DAT_KREIR (datum, ko je bila vrstica ustvarjena) in datum zadnje spremembe enaka.

ID1094_FORMULA	VA1094_SESTAV	DE1094_NAZIV	VA1094_OPIS1	VA1094_OPIS2
0003	VP_KOL * VP_V1	Kolicina * vrednost		

Slika 1 : Del stolpcev tabele formul, ki jo vrne poizvedba.

TR0000_USER_KREIR	TR0000_DAT_KREIR	TR0000_USER_AZUR	TR0000_DAT_AZUR	TR0000_UPORABA
PLACE01	12.1.2009 14:13:01	PLACE01	12.1.2009 14:13:01	

Slika 2 : Nadaljevanje tabele formul - dodatni stolpci.

Ugotovili smo torej, da moramo tudi v strankini bazi v tabelo formul dodati vrstico, ki bo vsebovala vrednosti, ki smo jih dobili s prejšnjim stavkom SELECT. Zato moramo te vrednosti zapisati v stavek INSERT (v nadaljevanju bomo uporabili izraz stavek za nadgradnjo). V njem naštejemo vsa imena stolpcev in njihove vrednosti. Ta stavek za nadgradnjo nato shranimo v datoteko, ki bo vsebovala vse tiste ukaze v jeziku SQL, s katerimi nadgradimo strankino bazo.

```
INSERT INTO kp1094_formule
            (id1094_formula, va1094_sestav, de1094_naziv,
            va1094_opis1, va1094_opis2, tr0000_user_kreir,
            tr0000_dat_kreir,tr0000_user_azur,tr0000_dat_azur,
            tr0000_uporaba)
     VALUES ('0003', 'VP_KOL * VP_V1', 'Količina * vrednost', NULL, NULL,
            'PLACE01', TO_DATE('12.01.2009 14:13:01', 'DD.MM.YYYY
            HH24:MI:SS'), 'PLACE01',
            TO_DATE('12.01.2009 14:13:01', 'DD.MM.YYYY HH24:MI:SS'), NULL
            );
```

Datoteko nato skupaj z novo verzijo sistema za obračun plač pošljemo stranki. Sistem za obračun plač med drugim poskrbi tudi zato, da se pred prvo uporabo sistema pri stranki stavki v tej datoteki izvedejo na strankini bazi. Tako s tem ustrezno popravimo strankino bazo in sistem pri stranki, v skladu s spremenjenimi formulami, deluje pravilno. Spomnimo se še enkrat, da bi brez teh sprememb imela stranka svojo bazo s »starimi« tabelami formul.

V primeru, da pri pripravi nove verzije sistema popravimo in dodamo večje število podatkov v vseh (ali vsaj nekaterih) devetih tabelah skupine B, je »ročno« dodajanje stavkov v datoteko zamudno. Zato sva se z mentorjem v podjetju, Markom Poljancem, odločila, da razvijem program za nadgradnjo strankine baze. Ta naj bi stavke za nadgradnjo avtomatsko sestavil in jih zapisal v datoteko .

Preden pa začnemo opisovati način, kako rešimo problem in razvijemo program, si najprej oglejmo sistem za upravljanje podatkovne baze, ki ga uporablja podjetje Spin.

2 SISTEM ORACLE

Podjetje Spin za upravljanje podatkovnih baz uporablja sistem Oracle. To je sistem za upravljanje relacijske podatkovne baze, angleško imenovan »relational database management system« (RDBMS). To je skupek programov, s katerimi bazo ustvarimo in jo upravljamo.
Trenutna verzija sistema Oracle, ki jo v podjetju uporabljajo, je Oracle11i.

Relacijska podatkovna baza je sestavljena iz množice podatkovnih tabel. To so zbirke podatkov, ki so sestavljene iz vrstic (zapisov) in stolpcev, kjer so v posamezni celici (presečišču tabele in stolpca) shranjeni podatki. Tabele so med seboj povezane in jih je mogoče združevati s pomočjo relacij (ujemanj). Z jezikom SQL ustvarimo tabele in dostopamo do njihovih podatkov.

V bazo lahko shranimo tudi programsko kodo oziroma tako imenovano programsko enoto. To nato po potrebi, na primer v javanskem programu, izvajamo in kličemo. Programske enote uporabljamo za obsežnejše in zahtevnejše upravljanje s podatki, kot jih lahko dosežemo s posameznimi stavki SQL. Pri programskih enotah, shranjenih v bazi, gre predvsem zato, da ni vedno potrebno, ko želimo neko programsko enoto izvesti, bazi posredovati kode enote.
Glede na uporabo programske enote delimo na: podprograme, prožilce in pakete, ki jih bomo podrobneje predstavili v razdelku 2.2.6. Programske enote ustvarimo z jezikom PL/SQL, ki je predstavljen v razdelku 2.2.

2.1 Jezik SQL

Na voljo imamo različne sisteme za upravljanje relacijskih baz podatkov. Na srečo pa obstaja skupni jezik, ki nam v poljubnem RDMBS omogoča upravljanje z bazo. Ta skupni jezik je SQL. SQL je okrajšava za Structured Query Language – strukturirani poizvedovalni jezik. Deli se na jezik za definiranje podatkov (DDL – data definition language), jezik za ravnanje s podatki (DML – data manipulation language) in jezik za nadzor podatkov (DCL – data control language). Oglejmo si najpomembnejše stavke SQL iz vsake skupine.

- Jezik DDL…….stavek CREATE, ALTER, DROP
- Jezik DML……stavek SELECT, INSERT, UPDATE, DELETE
- Jezik DCL…….stavek GRANT, REVOKE

Jezik SQL smo spoznali že med študijem praktične matematike, zato lahko predpostavimo, da ga poznamo.
Podrobneje si bomo ogledali le združevanje tabel, ker bomo pri razvoju našega programa potrebovali določene podrobnosti, ki nastopijo pri združevanju. Tako si bomo osvežili spomin na sam postopek in pogledali razliko v sintaksi ukazov med standardno in Oraclovo različico jezika SQL.

2.1.1 Združevanje tabel

Pri združevanju tabel želimo dobiti kot rezultat neke poizvedbe (stavka SELECT) tabelo, ki vsebuje podatke, ki so vsebovani v dveh ali več tabelah.
Tabele v relacijskih podatkovnih bazah združujemo z relacijami (ujemanji), ki jih izvedemo preko ključev. Ločimo dve vrsti ključev:

- Primarni ključ je stolpec ali kombinacija stolpcev v tabeli, ki enolično določajo vrstico.
- Tuji oziroma zunanji ključ je stolpec (ali kombinacija več stolpcev) tabele, ki je primarni ključ neke druge tabele.

Denimo, da združujemo tabeli Oddelek in Zaposleni (Slika 3).

Zaposleni

PRIIMEK	ODDELEK_ID
Likar	12
Koritnik	13
Sitar	13
Zalar	14

Tuji ključ

Oddelek

ODDELEK_ID	IME_ODDELKA
11	oddelekA
12	oddelekB
13	oddelekC

Primarni ključ

Slika 3 : Tabela Zaposleni in tabela Oddelek.

Združimo ju tako, da v poizvedovalnem stavku (stavek SELECT) uporabimo operacijo za združevanje, ki tabeli združi v eno. To združeno tabelo poizvedovalni stavek vrne kot rezultat. Tega v sistemu Oracle lahko zapišemo na dva načina. Oracle je do različice 9i uporabljal le svojo obliko zapisa združevanja. V njem ni uporabljal v jeziku SQL rezerviranih besed, kot sta INNER JOIN in OUTER JOIN. V različici 9i pa so vpeljali tudi standardni način združevanja tabel, ki jo uporabljajo tudi vse nadaljnje različice sistema Oracle.

Glede na način združevanja lahko dobimo različne združene tabele. Oglejmo si štiri tipe združevanja kar na primeru združevanja tabele Zaposleni in tabele Oddelek, ki sta predstavljeni na Sliki 3. Ob tem si bomo ogledali razliko v sestavi poizvedbe med standardnim načinom (kot določa standard jezika SQL) in posebnim načinom, ki ga uporablja le Oracle.

1. Notranje združevanje:
Tabeli združimo z notranjim združevanjem tako, da k vrstici tabele Zaposleni dodamo tisto vrstico tabele Oddelek, kjer se vrednosti polj v stolpcu OddelekID ujemata (torej kjer se tuji ključ tabele Zaposleni ujema s primarnim ključem tabele Oddelek). Če pripadajoče vrednosti v tabeli Oddelek ni, vrstice v združeni tabeli ni. Oglejmo si sintakso poizvedbe zapisano na dva načina.

Standardni SQL	Oracle SQL
Uporabimo operacijo INNER JOIN.	V delu WHERE združimo tabeli preko stolpca OddelekID.

`SELECT *`
`  FROM Zaposleni`
`  INNER JOIN Oddelek ON`
`  Zaposleni.OddelekID =`
`  Oddelek.OddelekID`
	`SELECT *`
	`  FROM Zaposleni, Oddelek`
	`  WHERE`
	`  Zaposleni.OddelekID =`
	`  Oddelek.OddelekID`

Seveda bi tudi v standardnem jeziku SQL lahko poizvedbo napisali na »Oraclov način«. V vsakem primeru pa kot rezultat dobimo

PRIIMEK	ODDELEK_ID	ODDELEK_ID_1	IME_ODDELKA
Likar	12	12	oddelekB
Koritnik	13	13	oddelekC
Sitar	13	13	oddelekC

Slika 4: Notranje združena tabela.

Kot vidimo na Sliki**Error! Reference source not found.**, zgornji poizvedbi vrneta tabelo s tistimi vrsticami, kjer se vrednosti polj v tabeli Zaposleni in Oddelek v stolpcu OddelekID ujemata.

2. Levo zunanje združevanje:

Tabeli združimo z levim zunanjim združevanjem tako, da najprej določimo glavno tabelo. To je v našem zgledu tabela Zaposleni. Imenujemo jo tudi leva tabela. V združeni tabeli bo vsaka vrstica te leve tabele (torej tabele Zaposleni) dobila vsaj eno pridruženo vrstico tabele Oddelek. To imenujemo desna tabela. Če v tabeli Oddelek za neko vrstico tabele Zaposleni ni ujemajoče se vrstice (take z enakim ključem), ji iz tabele Oddelek pridružimo tako s praznimi polji (vrednosti v stolpcih so ničelne vrednosti). Oglejmo si sintakso poizvedbe zapisano na dva načina.

Standardni SQL	Oracle SQL
Uporabimo operacijo LEFT OUTER JOIN.	V delu WHERE uporabimo operator +, ki ga zapišemo ob izrazu za tisto tabelo, kjer se lahko zgodi, da ujemajoče se vrednosti ni.

`SELECT *`
`  FROM Zaposleni`
`  LEFT OUTER JOIN Oddelek ON`
`  Zaposleni.OddelekID =`
`  Oddelek.OddelekID`
	`SELECT *`
	`  FROM Zaposleni, Oddelek`
	`  WHERE Zaposleni.OddelekID =`
	`  Oddelek.OddelekID(+)`

PRIIMEK	ODDELEK_ID	ODDELEK_ID_1	IME_ODDELKA
Likar	12	12	oddelekB
Koritnik	13	13	oddelekC
Sitar	13	13	oddelekC
Zalar	14		

Slika 5 : Levo zunanje združena tabela.

Kot vidimo na Sliki 5, smo poleg tistih vrstic, ki jih je vrnilo notranje združevanje, dobili tudi dodatno vrstico (k vrstici tabele Zaposleni s tujim ključem OddelekID, ki je enak 14), ki nima ujemajoče se vrstice v tabeli Oddelek.

3. Desno zunanje združevanje:

Tabeli združimo z desnim zunanjim združevanjem na podoben način kot z levim zunanjim združevanjem. Razlika je le v tem, da je tokrat glavna tabela Oddelek (desna tabela). Tako bodo vse vrstice tabele Oddelek dobile vsaj eno pridruženo vrstico iz tabele Zaposleni (leva tabela). Če v tabeli Zaposleni ni ujemajoče se vrstice, vzamemo takšno s praznimi polji. Oglejmo si sintakso poizvedbe zapisano na oba načina.

Standardni SQL	Oracle SQL
Uporabimo operacijo RIGHT OUTER JOIN.	V delu WHERE uporabimo operator +, ki ga zapišemo ob izrazu za tisto tabelo, kjer se lahko zgodi, da ujemajoče se vrednosti ni.

`SELECT *`
`  FROM Zaposleni`
`  RIGHT OUTER JOIN Oddelek ON`
`  Zaposleni.OddelekID =`
`  Oddelek.OddelekID`
	`SELECT *`
	`  FROM Zaposleni, Oddelek`
	`  WHERE Zaposleni.OddelekID(+) =`
	`  Oddelek.OddelekID`

PRIIMEK	ODDELEK_ID	ODDELEK_ID_1	IME_ODDELKA
		11	oddelekA
Likar	12	12	oddelekB
Koritnik	13	13	oddelekC
Sitar	13	13	oddelekC

Slika 6: Desno zunanje združena tabela.

Kot vidimo na Sliki 6, smo poleg tistih vrstic, ki jih je vrnilo notranje združevanje, dobili tudi vrstico k vrstici iz tabele Oddelek s stolpcem Oddelek_ID = 11, ki nima ujemajoče se vrstice v tabeli Zaposleni.

4. Celotno zunanje združevanje:
Je unija levega in desnega zunanjega združevanja. To pomeni, da iščemo ujemajoče vrstice iz tabele Zaposleni (leva tabela) in tabele Oddelek (desna tabela). Če v levi ali desni tabeli ujemajoče vrstice ni, vzamemo tako s praznimi polji. Oglejmo si sintakso poizvedbe zapisano na dva načina.

Standardni SQL	Oracle SQL
Uporabimo operacijo FULL OUTER JOIN.	Uporabimo operacijo UNION, ki združi levo in desno zunanje združevanje.

`SELECT *`
`  FROM Zaposleni`
`  FULL OUTER JOIN Oddelek ON`
`  Zaposleni.OddelekID =`
`  Oddelek.OddelekID`
	`SELECT *`
	`  FROM Zaposleni, Oddelek`
	`  WHERE Zaposleni.OddelekID =`
	`  Oddelek.OddelekID(+)`
	`UNION`
	`SELECT *`
	`  FROM Zaposleni, Oddelek`
	`  WHERE Zaposleni.OddelekID(+) =`
	`  Oddelek.OddelekID`

PRIIMEK	ODDELEK_ID	ODDELEK_ID_1	IME_ODDELKA
		11	oddelekA
Likar	12	12	oddelekB
Koritnik	13	13	oddelekC
Sitar	13	13	oddelekC
Zalar	14		

Slika 7 : Celotno zunanje združena tabela.

Kot vidimo na Sliki 7, smo poleg tistih vrstic, ki jih je vrnilo levo zunanje združevanje, dobili tudi tiste vrstice, ki jih je vrnilo desno zunanje združevanje.

Pri pisanju programov v novejših različicah sistema Oracle lahko uporabljamo oba načina. V nadaljevanju diplomske naloge bomo uporabljali »stari« način (z uporabo operatorja +), saj v podjetju Spin razvijalci še vedno pišejo na tak način.

Sedaj pa si poglejmo še programski jezik PL/SQL.

2.2 Jezik PL/SQL

Jezik PL/SQL je programski jezik, ki predstavlja razširitev jezika SQL. Z jezikom PL/SQL lahko v celoto združimo več stavkov SQL in tako ustvarimo programske enote. Omogoča tudi uporabo vejitev, zank in podobno, ki jih SQL ne pozna. Zato PL/SQL uporabljamo za obsežnejše in zahtevnejše upravljanje s podatki, kot to lahko dosežemo s stavki SQL. Programske enote so podrobneje predstavljene v razdelku 2.2.6.

Programiranje z jezikom PL/SQL omogoča, da program napišemo kot skupek modulov. To pomeni, da program razdelimo na več manjših delov, pri čemer lahko posamezni del vsebuje enega ali več delov. Te dele imenujemo podprogrami, ki jih poimenujemo in uporabimo (kličemo) na različnih mestih v programu. Podprogrami so predstavljeni v razdelku 2.2.6.1.

Sestavni del jezika PL/SQL so operatorji, spremenljivke, stavki SQL, krmilni stavki PL/SQL (zanke, stavki IF itd.), vgrajene funkcije, vgrajeni paketi itd. V nadaljevanju tega razdelka bomo predstavili tiste sestavne dele, ki jih bomo potrebovali pri razvijanju našega programa. Pojme kot so operatorji, spremenljivke, stavki IF, zanke itd. poznamo že iz programskega jezika java. Zato bomo za te pojme opisali le morebitne razlike glede na programski jezik java, sintakso in tam, kjer je potrebno, dodali tudi primere.

Povejmo še, da pri pisanju kode v večini primerov ni pomembno ali uporabljamo velike ali male črke. Le zaradi lepše preglednosti ukaze pišemo z velikimi črkami. Ukazi so v kodi obarvani modro, podatkovni tipi rdeče, imena tabel in programskih enot (tiste, katere so že shranjene v bazi) svetlo zeleno ter komentarji temno zeleno. V nizih pa je v nekaterih primerih pomembna uporaba velikih ali malih črk. Dva takšna primera, bomo opisali v razdelku 4.1 in razdelku 4.2.

2.2.1 Operatorji

Oglejmo si najbolj osnovne operatorje, ki jih uporablja jezik PL/SQL.

Operator	Operacija
:=	Prireditev (spremenljivki dodeli vrednost).
||	Spajanje nizov.
+, -, *, /	Seštevanje, odštevanje, množenje, deljenje.
=, <, >, <=, <>	Primerjanje vrednosti.
AND, OR	Logična IN in ALI.
IS NULL	Preverjanje ničelne vrednosti. Če je neka vrednost ničelna, operator vrne true. V nasprotnem primeru operator vrne false.

Primere uporabe teh operatorjev si bomo ogledali v zgledih v nadaljevanju.

2.2.2 Spremenljivke

Tako kot v jeziku java moramo tudi v jeziku PL/SQL spremenljivke pred prvo uporabo napovedati, oziroma deklarirati. To storimo tako, da jim določimo ime in podatkovni tip (ter začetno vrednost, če to želimo). V jeziku PL/SQL začetno vrednost določimo z operatorjem za prireditev (:=). Če pa želimo, da ima spremenljivka konstantno vrednost (je torej konstanta), pred navedbo tipa uporabimo rezervirano besedo CONSTANT.

SINTAKSA:

```
-- Spremenljivka
ime_spremenljivke TIP := začetna_vrednost;

-- Konstanta
ime_spremenljivke CONSTANT TIP := začetna_vrednost;
```

2.2.2.1 Podatkovni tipi spremenljivk

V tem razdelku si oglejmo sedem podatkovnih tipov, ki jih bomo uporabili pri razvijanju našega programa.

1. Podatkovni tip VARCHAR2
Kot podatkovni tip tistih spremenljivk, v katerih hranimo nize, določimo VARCHAR2. Nize sicer lahko hranimo tudi v spremenljivkah tipa STRING oziroma VARCHAR. Slednji se je uporabljal v starejših verzijah Oracla, kjer tipa VARCHAR2 še ni bilo. Tip VARCHAR se obnaša nekoliko drugače od VARCHAR2 in naj se ne bi uporabljal več.

Niz predstavlja zaporedje znakov, kot so črke, številke, '(', '+' itd. Ob deklaraciji moramo v oklepajih določiti maksimalno možno dolžino niza, saj v nasprotnem primeru prevajalnik javi napako. Največja možna dolžina je 32767 znakov. Nize v kodi označujemo z enojnimi narekovaji. Dvojni narekovaji so lahko le del niza.

PRIMER 1:

Denimo, da bi potrebovali spremenljivko, v kateri so našteta imena tabel. Ta so v nizu ločena z vejicami.

```
imena_tabel VARCHAR2(200)
      := 'KP1199_SKU_SIFRANT,KP1200_SKU_VR_SIFR,KP1093_SPR_FORMUL,
      KP1094_FORMULE,KP1230_JS_VP_SK,KP1229_JS_VP,KP1228_JS_DM,
      KS0027_LABELE,KS0037_BESEDILA';
```

Nizi imajo spremenljivo dolžino. To pomeni, da lahko na primer v spremenljivko `imena_tabel`, shranimo nize z dolžino od 0 do 200 znakov. V primeru, da v spremenljivko želimo shraniti niz dolžine na primer 203, prevajalnik javi napako.

Nize lahko tudi spajamo z operatorjem za spajanje (`||`). Oglejmo si primer.

PRIMER 2:

Denimo, da imamo v spremenljivki zapisan nek niz. Ta niz želimo na začetku in koncu opremiti z narekovaji. Če imamo v tej spremenljivki na primer niz xxx, želimo iz tega narediti niz 'xxx'. V razdelku 3 bomo razložili, zakaj potrebujemo na ta način opremljen niz.

```
formula VARCHAR2(20) := 'VP_KOL * VP_V1';

dodaj_narekovaj VARCHAR2(20):= ''''||formula||'''';
```

"" je niz, ki vsebuje en enojni narekovaj. Prvi ' pomeni začetek niza. Naslednja dva ' v notranjosti niza pomenita en ' ,četrti (zadnji) ' pa zaključuje niz.

Tako iz niza VP_KOL * VP_V1 ustvarimo niz 'VP_KOL * VP_V1'.

2. Podatkovni tip NUMBER

V jeziku java sta za cela in decimalna števila na voljo različna tipa (int in double). V jeziku PL/SQL pa cela in decimalna števila (s fiksno ali premično decimalno piko) deklariramo z istim tipom NUMBER. Sicer lahko uporabimo tudi tipe INTEGER, INT, REAL itd. Slednji so se uporabljali v starejših verzijah Oracla, kjer tipa NUMBER še ni bilo. Vendar naj se tipi INTEGER, INT, REAL ne bi več uporabljali.

Število je lahko sestavljeno iz največ 40 znakov (skupaj z decimalnimi mesti). Morebitna predznak in decimalna pika ne štejeta k temu številu znakov. Če želimo, lahko določimo maksimalno dolžino števila in število decimalnih mest. Tako kot nizi imajo tudi števila spremenljivo dolžino. Oglejmo si primere.

PRIMERI:

```
-- Spremenljivka celo_st lahko shrani trimestno število brez
-- decimalnih mest.
celo_st NUMBER(3) := 10;

-- V spremenljivko decimalno_st lahko shranimo štirimestno število z
-- dvema decimalnima mestoma.
-- V tem primeru se število shrani s fiksno decimalno piko.
decimalno_st NUMBER(4,2) := 88.1;

-- Spremenljivka stevilo lahko vsebuje celo ali decimalno število.
-- Tu je uporabljena premična decimalna pika.
stevilo NUMBER := 88.11;
```

3. Podatkovni tip DATE

Podatkovni tip spremenljivk, v katerih hranimo datume, določimo z DATE. Datum je predstavljen kot zapis s polji za dan, mesec, leto in čas (ure, minute, sekunde). Pri pisanju kode datume običajno vnašamo kot nize, ki predstavljajo parameter funkcije TO_DATE. Tudi, ko želimo nek določen datum vnesti v tabelo uporabimo to funkcijo. Ta je predstavljena v razdelku 2.2.3.

PRIMER:

```
-- Spremenljivka datum vsebuje
-- (datum), ki je zapisan v nizu in smo ga s
-- funkcijo TO_DATE pretvorili v datum.
datum DATE := TO_DATE('04.01.2007 11:20:55','DD.MM.YYYY HH24:MI:SS')
```

4. Podatkovni tip BOOLEAN

Tako kot v jeziku java (z izjemo velikih črk) je ime podatkovnega tipa za hranjenje logičnih vrednosti BOOLEAN. Logične vrednosti imajo vrednost TRUE in FALSE. Kot smo omenili že prej, uporaba velikih oz. malih črk ni pomembna, zato lahko uporabimo tudi obliko true, True …

PRIMER:

```
-- S tem, da spremenljivko postavimo na true, povemo,
-- da je v vrstici neke tabele prišlo do spremembe.
nasli_spremembo BOOLEAN := TRUE;
```

5. Podatkovni tip CURSOR

CURSOR je poseben tip spremenljivke, v kateri hranimo celotno tabelo s podatki. To tabelo nam kot rezultat vrne določena poizvedba (stavek SELECT). V javi je podoben tip Iterator.

Spremenljivko tipa CURSOR deklariramo tako, da ji določimo ime in napišemo poizvedbo, ki vrne ustrezno tabelo. Za pomikanje po tej tabeli uporabimo posebno obliko zanke FOR, s pomočjo katere preberemo vrednosti (vrstice) te tabele. To zanko bomo podrobneje predstavili v razdelku 2.2.5.

Poizvedba je sestavljena statično. To pomeni, da že med pisanjem programa poznamo končno obliko poizvedbe. Ta je sestavljena iz ukazov, imen stolpcev, imen tabel in spremenljivk. Spremenljivke so lahko tipa VARCHAR2, NUMBER, DATE in BOOLEAN. V spremenljivko tipa VARCHAR2 pa ne moremo shraniti imena stolpcev in tabel.

V spodnjem primeru smo v poizvedbi našteli vsa imena stolpcev, čeprav bi lahko uporabili *, s katerim sicer v poizvedbi označimo, da želimo dobiti vse stolpce. Ob uporabi * sistemi za delo z bazami podatkov (RDBMS) ne jamčijo, da bo poizvedba vedno vrnila tabelo z enako razvrščenimi stolpci. Med posamezno izvedbo programa, v katerem je spremenljivka tipa CURSOR deklarirana, se torej lahko spremeni vrstni red stolpcev, ki jih vrača poizvedba. V spodnjem primeru sicer vrstni red stolpcev ni pomemben, saj vrednosti iz spremenljivke tipa CURSOR preberemo s klicem `tek_vrstica.ime_stolpca`. To pomeni, da med pisanjem kode poznamo imena stolpcev. Vrstni red vrnjenih stolpcev pa je lahko pomemben v primeru, ko med pisanjem kode ne poznamo imena stolpcev. Tak primer bomo opisali v razdelku 4.3. V splošnem torej v poizvedbi ni priporočljivo uporabljati *, ampak raje naštejemo vse stolpce.

PRIMER:

Denimo, da bi potrebovali spremenljivko tipa CURSOR, v kateri bi hranili tabelo tistih vrstic, ki bi se spremenile po danem datumu. Slednji je v poizvedbi podan kot spremenljivka.

```
CURSOR cur_formule
      IS
         SELECT id1094_formula, va1094_sestav, de1094_naziv, va1094_opis1,
                va1094_opis2, tr0000_user_kreir, tr0000_dat_kreir,
                tr0000_user_azur, tr0000_dat_azur, tr0000_uporaba
           FROM kp1094_formule
           WHERE tr0000_dat_azur >= datum;

. . .

-- Iz tabele, ki jo hrani cur_formule s pomočjo zanke FOR preberemo vse
-- vrednosti in jih zapišemo v posamezne spremenljivke.
FOR tek_vrstica IN cur_formule LOOP
    id_formule := tek_vrstica.id1094_formula;
    id_sestav := tek_vrstica.va1094_sestav;
    . . .
END LOOP;
```

V nadaljevanju diplomske naloge bomo spremenljivke tipa CURSOR praviloma poimenovali z `cur_imeSpr`. Tako bomo v kodi programa takoj vedeli, da gre za spremenljivko omenjenega tipa.

6. Podatkovni tip RECORD

Podatkovni tip RECORD je sestavljen tip, s katerim lahko določimo nov tip spremenljivke. Tip RECORD je lahko sestavljen iz več spremenljivk enakih ali različnih tipov, kot so tipi: VARCHAR2, NUMBER, DATE, BOOLEAN in RECORD. Te posamezne dele sestavljenega tipa imenujemo atributi (lastnosti), ki jih ločimo z imeni. Novo ustvarjen sestavljen tip moramo poimenovati, da ga lahko uporabimo pri deklaraciji spremenljivk. Poglejmo si primer.

PRIMER 1:

Spremenljivke tipa RECORD ne moremo deklarirati na način, kot je prikazano spodaj.

```
delavec IS RECORD (
    ime VARCHAR2(10),
    priimek VARCHAR2(10),
    starost NUMBER(3) );
```

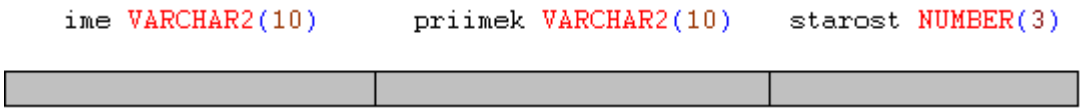
Novo ustvarjen tip RECORD moramo najprej poimenovati in ga nato uporabiti pri deklaraciji spremenljivke.

```
-- Tip osebni_podatki je sestavljen iz dveh atributov tipa VARCHAR2 in
-- atributa tipa NUMBER.
TYPE osebni_podatki IS RECORD (
    ime VARCHAR2(10),
    priimek VARCHAR2(10),
    starost NUMBER(3)
    );

-- Spremenljivka tipa osebni_podatki.
delavec osebni_podatki;
. . .

-- V spremenljivko delavec vstavimo ime, priimek in starost delavca.
delavec.ime := 'Klemen';
delavec.priimek := 'Pevec';
delavec.starost := 50;
```

Za lažjo predstavo si tip RECORD lahko predstavljamo kot zapis oziroma enovrstično tabelo, kot je prikazano na Sliki 8.



Slika 8: Tip RECORD.

PRIMER 2:

Denimo, da bi potrebovali spremenljivko tipa RECORD, v kateri bi hranili imena, tipe in oznake omejitev za posamezni stolpec določene tabele. Oznaki omejitev sta npr. 'P' za primarni ključ in 'R' za tuji ključ. Imena, tipi in oznake so tipa VARCHAR2, zato sestavimo tip RECORD z imenom `opis_stolpca`, sestavljen iz treh atributov tipa VARCHAR2.

```
-- Tip opis_stolpca je sestavljen iz treh atributov tipa VARCHAR2.
TYPE opis_stolpca IS RECORD (
    ime VARCHAR2(20),
    tip VARCHAR2(10),
    omejitev VARCHAR2(1)
    );

-- Spremenljivka tipa opis_stolpca.
stolpec opis_stolpca;

. . .

-- V spremenljivko stolpec vstavimo ime, tip in oznako omejitve stolpca.
stolpec.ime := 'id_1094_formula';
stolpec.tip := 'VARCHAR2';
stolpec.omejitev := 'P';
```

7. Podatkovni tip TABLE

Tudi tip TABLE je sestavljen tip. Vendar pa je tip TABLE z razliko od tipa RECORD sestavljen iz več spremenljivk enakega tipa. Te posamezne dele tudi pri tem tipu imenujemo atributi (lastnosti). Novo ustvarjen tip moramo poimenovati, da ga lahko uporabimo pri deklaraciji spremenljivk. Tak

podatkovni tip poznamo tudi v jeziku java, kjer ga določimo z oglatimi oklepaji, na primer: int[] in nam ga ni potrebno posebej poimenovati.
Do posameznih elementov imamo dostop prek indeksov, ki so lahko cela števila, negativna števila, znaki itd. Pri ustvarjanju tipa moramo navesti, kaj bomo uporabljali kot indekse.
Za indekse običajno uporabljajmo cela števila, tako kot smo navajeni v javi. Takrat kot tip indeksa uporabimo tip BINARY_INTEGER, s katerim tabeli določimo indekse od -2.147.483.647 do +2.147.483.647. Poglejmo si primera.

PRIMER 1:

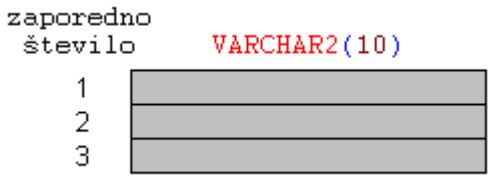
Denimo, da bi potrebovali spremenljivko tipa TABLE, v katero bi shranili imena stolpcev določene tabele.

```
-- Tip imena_stolp je sestavljen iz tipa VARCHAR2.
TYPE imena_stolp IS TABLE OF VARCHAR2(10)

      -- zaporedna števila atributov
      INDEX BY BINARY_INTEGER;

-- Spremenljivka tipa imena_stolp.
stolpci imena_stolp;
```

Tip TABLE si lahko predstavljamo kot tabelo z enim stolpcem, kot je prikazano na Sliki 9.



Slika 9 : Tip TABLE.

Oglejmo si še primer, kjer je tip TABLE sestavljen iz tipa `opis_stolpca`. Slednjega smo predstavili v prejšnji točki 6.

PRIMER 2:

```
-- Tip opis_tabele je sestavljen iz tipa opis_stolpca (tip RECORD iz
-- točke 6.)
TYPE opis_tabele IS TABLE OF  opis_stolpca

      -- zaporedna števila atributov
      INDEX BY BINARY_INTEGER;

-- Spremenljivka tabelaImenTabel je tipa opis_tabele.
tabelaImenTabel opis_tabele;
```

V jeziku PL/SQL nad spremenljivko tipa TABLE lahko uporabljamo različne metode. Tako za štetje »zasedenih« vrstic in brisanje vrednosti iz spremenljivke tipa TABLE uporabimo metodi COUNT in DELETE.

Primer uporabe teh metod ilustrirajmo z naslednjim zgledom. V njem najprej v spremenljivko `tabela` (ki smo jo deklarirali v prejšnjem zgledu) v vrstice z indeksi od 5 do zadnje vrstice v spremenljivki `cur_metapod` (tipa CURSOR) shranimo imena stolpcev, tipe stolpcev in oznake omejitev. Spremenljivko `cur_metapod` bomo podrobneje predstavili v razdelku 4.4.2. Tip

CURSOR smo predstavili v točki 5. Nato z metodo COUNT preštejemo »zasedene« vrstice v spremenljivki `cur_metapod` in na koncu z metodo DELETE iz omenjene spremenljivke izbrišemo vse vrednosti.

```
-- Prvi vrstici v spremenljivki tabela določimo indeks 5.
indeks NUMBER :=5;

-- S pomočjo zanke FOR prenašamo vrednosti iz spremenljivke cur_metapod
-- v spremenljivko tabela. Denimo, da je v spremenljivki cur_metapod 15
-- vrstic z vrednostmi.Te bom dali v tabelo na indekse od 5 do 19.
FOR tek_vrstica IN cur_metapod LOOP
  tabela(indeks).ime := tek_vrstica.column_name;
  tabela(indeks).tip := tek_vrstica.data_type;
  tabela(indeks).oznaka := tek_vrstica.oznaka;
  ...
  indeks := indeks + 1;
END LOOP;

-- Spremenljivka prejme število 15.
st_vrstic := tabela.COUNT;

-- Z DELETE izbrišemo vse vrednosti v tabeli.
tabela.DELETE;

-- v st_vrstic bo sedaj 0.
st_vrstic := tabela.COUNT;
```

Povejmo še, da imamo lahko v spremenljivki tipa TABLE tudi »luknje«. To pomeni, da lahko v spremenljivko `tabela` za zanko dodamo še nekaj vrednosti v del tabele z indeksi npr. od 30 do 40.

2.2.3 Vgrajene funkcije

Podatkovne tipe VARCHAR2, NUMBER in DATE lahko tudi pretvarjamo iz enega tipa v drugega. Pretvarjamo jih z vgrajenimi funkcijami jezika PL/SQL. Poleg teh funkcij bomo v tem razdelku predstavili še dve vgrajeni funkciji, ki ju bomo potrebovali pri razvijanju našega programa.

1. Funkcija INSTR primerja vzorec (niz) na mestu drugega parametra z nizom, ki ga uporabimo kot prvi parameter. Če se vzorec pojavi v prvem parametru, funkcija vrne mesto, kjer se ta prvič pojavi (število od 1 do 32767). Če vzorca v nizu ni, funkcija vrne 0.

PRIMER:

Denimo, da želimo za določeno ime tabele preveriti, če je zapisano v nizu spremenljivke `imena_tabel`. Ta spremenljivka naj ima vrednost, kot smo jo uporabili v zgledu v razdelku 2.2.2.1.

```
-- Funkcija INSRT vrne število 31, saj se vzorec KP1094_FORMULE
-- v spremenljivki imena_tabel pojavi na 31 mestu.
INSTR(imena_tabel,'KP1094_FORMULE')
```

2. Funkcija REPLACE primerja vzorec (niz) na mestu drugega parametra z nizom, ki ga uporabimo kot prvi parameter. Če se vzorec pojavi v nizu prvega parametra, funkcija ta vzorec v prvem parametru zamenja z nizom, ki je podan kot tretji parameter. Funkcija nato ta spremenjen niz vrne. V primeru, če vzorca v nizu ni, funkcija vrne niz v enaki obliki, kot ga prejme.

PRIMER:

Primer uporabe funkcije REPLACE si poglejmo kar na primeru spremenljivke `dodaj_narekovaj` (razdelek 2.2.2.1), kjer smo vrednost spremenljivke `formula` opremili z narekovaji.

```
dodaj_narekovaj1 := ''''||formula||'''';

-- Funkcija REPLACE zamenja dvojni narekovaj z dvema enojnima
-- narekovajema.
dodaj_narekovaj2 := REPLACE('"'||formula||'"', '"', '''');
```

V obe zgornji spremenljivki se shrani enako opremljen niz, kot je npr. niz: 'VP_KOL * VP_V1'.
V funkciji REPLACE z " označimo dva ', ki predstavljata en ' s katerim opremimo vrednost. Tako bomo pri pisanju kode našega programa ločili narekovaje s katerimi bomo označevali nize in narekovaje s katerimi bomo opremili vrednosti. Na ta način bomo dosegli preglednejšo kodo programa.

3. Funkcija TO_DATE pretvori niz, ki ga podamo kot prvi parameter v datum. Z drugim parametrom, ki je tudi niz, povemo, v kakšni obliki je zapisan datum v prvem parametru. Seveda predpostavimo, da prvi niz predstavlja veljaven datum, ne pa npr. 30. februar. V tem primeru funkcija vrne napako.

Parameter, ki poda obliko, je najpogosteje sestavljen iz delov, kot so:

- DD označuje dan v mesecu (01-31)
- MM označuje mesec v letu s števili (01-12)
- YYYY označuje štirimesten zapis leta od leta 4713 p.n.š. do 9999 n.š.
- YY označuje dvomesten zapis leta od leta 2000 do 2099.
- SS označuje sekunde (0-59)
- MI označuje minute (0-59) - v standardnem jeziku SQL uporabljamo MM
- HH24 označuje ure (0-23)

Osnovna oblika, v kateri določamo datume, je : 'DD.MM.YYYY HH24:MI:SS'. Uporabimo pa lahko tudi druge oblike ali pa podamo datum brez določene oblike. Poglejmo si nekaj primerov.

PRIMERI:

```
-- Funkcija vrne datum 10. februar 2009 s časom 00:00:00
-- (saj ga v prvem nizu nismo podali).
TO_DATE('10.02.2009','DD.MM.YYYY HH24:MI:SS')

-- Funkcija vrne datum 10. februar 2009.
TO_DATE('10/02/09','DD/MM/YY')
```

V kolikor ne podamo drugega parametra, funkcija TO_DATE za štirimestni zapis leta v nizu preveri, če je podano leto med letom 4713 p.n.š. in letom 9999 n.š. V kolikor podano leto v nizu ni med omenjenimi leti, prevajalnik javi napako. V primeru dvomestnega zapisa leta v nizu pa funkcija preveri:

- če je dvomesten zapis med 50 in 99, funkcija TO_DATE vrne leto 19xx
- če je dvomesten zapis med 00 in 49, pa funkcija TO_DATE vrne leto 20xx

V nizu prvega parametra moramo najprej podati dan, nato mesec ter leto. V nizu moramo posamezna polja za dan, mesec in leto ločiti z znaki, kot so : ., -, /.

```
-- Funkcija vrne datum 10. februar 2009.
TO_DATE('10.02.2009')

-- Prevajalnik javi napako.
TO_DATE('10.02.99999')

-- Funkcija vrne datum 10. februar 1999.
TO_DATE('10/02/99')

-- Funkcija vrne datum 10. februar 2000.
TO_DATE('10-02-00')
```

4. Funkcija TO_CHAR pretvori vrednost prvega parametra, ki je bodisi število bodisi datum, v niz. Ta je oblikovan tako, kot določa oblika, podana kot drugi parameter. Obliko navedemo kot niz, v katerem uporabimo različne oznake. Tako z 9 povemo, da želimo imeti na tem mestu števko. Glede na to, koliko 9 navedemo, povemo, koliko mestno naj bo število. Če želimo, da ima število decimalna mesta, na ustreznem mestu oznake 9 ločimo s piko. Pri pretvarjanju datumov v nize uporabimo oznake, ki smo jih omenili pri opisu funkcije TO_DATE.

PRIMERI:

```
-- Funkcija vrne niz : 76.
TO_CHAR(76,'999')

-- Funkcija vrne niz : 34.85.
TO_CHAR(34.5767,'999.99')

-- Prevajalnik javi napako.
TO_CHAR(10.2,'9.99')

-- Funkcija vrne niz z datumom, ki ga določimo z
-- vgrajeno funkcijo SYSADTE. Ta vrne današnji
-- datum in trenutni čas.
TO_CHAR(SYSDATE,'DD.MM.YYYY HH24:MI:SS')
```

Seveda lahko nastopijo tudi številne druge kombinacije, v formatnem nizu pa lahko uporabimo tudi drugačne oznake. A v programu, ki ga bomo sestavili, bomo potrebovali le pretvorbe take oblike, kot so navedene v zgornjem zgledu.

5. Funkcija TO_NUMBER pretvori niz, podan kot prvi parameter, v število. Pri tem z drugim parametrom določimo, kako moramo razumeti zapis števila v nizu. Tako kot v funkciji TO_CHAR tudi v tej funkciji navedemo obliko kot niz, v katerem z 9 povemo, da želimo na tem mestu števko.

PRIMERI:

```
-- Funkcija vrne število : 10.
TO_NUMBER('10','9999')

-- Funkcija vrne število : 10.2.
TO_NUMBER('10.2','9999.99')

-- Prevajalnik javi napako.
TO_NUMBER('12.45','99.9')
```

Seveda lahko nastopijo tudi številne druge kombinacije, v formatnem nizu pa lahko uporabimo tudi drugačne oznake. Te funkcije pri razvijanju našega programa ne bomo potrebovali, a smo jo navedli kot »sestrsko« funkcijo k funkciji TO_CHAR.

2.2.4 Stavek IF

Pogojni stavek IF ima v jeziku PL/SQL iste značilnosti kot v jeziku java. Zato si poglejmo le sintaksi gnezdenja stavkov IF v obeh jezikih.

SINTAKSA v jeziku java:

```
if (pogoj1) {
   stavek1;
   stavek2;
   . . .
}
else if (pogoj2) {
   stavek3;
   stavek4;
   . . .
}
else {
   stavek5;
   stavek6;
   . . .
}
```

SINTAKSA v jeziku PL/SQL:

```
IF pogoj1 THEN
   stavek1;
   stavek2;
   . . .

ELSIF pogoj2 THEN
   stavek3;
   stavek4;
   . . .

ELSE
   stavek5;
   stavek6;
   . . .
END IF;
```

Kot vidimo, v jeziku java uporabimo {}, s katerimi obdamo posamezni del stavka if. Z else if določimo gnezdeni stavek. V jeziku PL/SQL z THEN povemo, da bomo v tem delu izvedli določene stavke, z ELSIF pa določimo gnezdeni stavek. Na koncu zaključimo stavek IF z END IF.

2.2.5 Zanke

Iz jezika java poznamo tudi zanki WHILE in FOR.

1. Zanka WHILE
Zanka WHILE, ki jo uporablja jezik PL/SQL, se izvaja enako kot zanka WHILE v javi. Razlika je v bistvu le v zapisu (sintaksi), ki je le malenkost drugačna. Poglejmo si sintaksi zanke WHILE v obeh jezikih.

SINTAKSA v jeziku java:

```
while pogoj {
   stavek1;
   stavek2;
   . . .

   indeks = indeks + 1;
}
```

SINTAKSA v jeziku PL/SQL:

```
WHILE pogoj LOOP
   stavek1;
   stavek2;
   . . .

   indeks := indeks + 1;
END LOOP;
```

Kot vidimo, se sintaksi razlikujeta le v tem, kako določimo telo zanke. V jeziku java telo zanke določimo z {}, v jeziku PL/SQL pa začetek zanke določimo z LOOP, konec zanke pa z END LOOP.

2. Zanka FOR

Tako kot v jeziku java poznamo tudi v jeziku PL/SQL »običajno« in »posebno« obliko zanke FOR.

OBIČAJNA ZANKA FOR:

Običajna zanka FOR, ki jo uporablja jezik PL/SQL, se izvaja podobno kot običajna zanka FOR v javi, kjer uporabimo običajni števec s korakom 1. Poglejmo si sintaksi.

SINTAKSA v jeziku java:

```
//Določiti moramo tip števca, začetno in končno vrednost števca ter korak,
//s katerim ob posameznem prehodu zanke povečujemo števec.
for (int indeks = 1; indeks <= 10; indeks++){
   stavek1;
   stavek2;
   . . .
}
```

SINTAKSA v jeziku PL/SQL:

```
-- Tip spremenljivke indeks se določi
-- avtomatično(cela števila).
FOR indeks IN 1..10 LOOP
  stavek1;
  stavek2;
  . . .

-- Indeks se poveča za 1 s posameznim
-- prehodom zanke.
END LOOP;
```

Kot vidimo, se sintaksi razlikujeta v načinu določitve indeksa. Poleg tega se sintaksi razlikujeta tudi v načinu določitve telesa zanke, tako kot pri zanki WHILE.

POSEBNA ZANKA FOR:

To zanko uporabljamo v našem primeru za branje vrednosti (vrstice) tabele, ki jo hrani spremenljivka tipa CURSOR (v javi je soroden tip Iterator), kot smo opisali v razdelku 2.2.2.1. Posebna oblika zanke FOR deluje tako, da odpre dostop do rezerviranega mesta v pomnilniku, kjer hranimo omenjeno tabelo. Nato vzame posamezno vrstico tabele in zapre dostop do rezerviranega mesta v pomnilniku.
Ta oblika zanke FOR v jeziku java in v jeziku PL/SQL deluje nekoliko drugače. V komentarjih bomo razložili kako delujeta posebni obliki zanke FOR v jeziku java in jeziku PL/SQL.

SINTAKSA v jeziku java:

```
// V spremenljivko it shranimo vse vrednosti, ki so shranjene
// v seznamu.
for (Iterator<tip> it = seznam.iterator(); it.hasNext();) {

  //S funkcijo next() pridemo na naslednjo vrednost v spremenljivki it.
  tip element = it.next();
  stavek1;
  stavek2;
  . . .
}
```

SINTAKSA v jeziku PL/SQL:

```
-- V spremenljivko tek_vrstica (tipa RECORD) shranimo eno vrstico
-- tabele. To tabelo hrani spremenljivka cur_ime(tipa CURSOR).
-- Spremenljivki tek_vrstica se avtomatično priredijo tipi
-- argumentov glede na tipe stolpcev tabele, katero hrani
-- spremenljivka ime_cur.

FOR tek_vrstica IN cur_ime LOOP

  -- Iz tek_vrstice preberemo vrednost iz določenega stolpca.
  element := tek_vrstica.ime_stolpca;
  stavek1;
  stavek2;
  . . .

-- Z naslednjim prehodom zanke v spremenljivko tek_vrstica shranimo
-- naslednjo vrstico spremenljivke cur_ime.
END LOOP;
```

Posebno obliko zanke FOR v jeziku PL/SQL lahko zapišemo tudi tako, kot je prikazano spodaj. Pri razvijanju našega programa bomo zaradi lepše preglednosti spremenljivko tipa CURSOR najprej deklarirali in ji nato uporabili v zanki FOR.

```
FOR indeks IN (SELECT … FROM …) LOOP
  stavek1;
  stavek2;
  . . .
END LOOP;
```

2.2.6 Programske enote

Kot smo razložili v razdelku 2, programske enote delimo na podprograme, prožilce in pakete. V naslednjih razdelkih bomo s primeri predstavili posamezne programske enote.

2.2.6.1 Podprogrami

Podprogrami predstavljajo programsko kodo, ki je lahko samostojno shranjena v podatkovni bazi ali pa je del paketa. Pakete bomo predstavili v razdelku 2.2.6.2. Podprograme delimo na :

- Procedure, ki prejmejo parametre in izvajajo zaporedje stavkov.
- Funkcije, ki prejmejo parametre, vendar z razliko od procedur poleg izvajanja stavkov tudi vračajo rezultat (vrednost).

Podprogrami so sestavljeni iz dveh delov: deklaracijskega dela in stavčnega dela. V deklaracijskem delu deklariramo lokalne spremenljivke in sestavljene tipe, v stavčni del pa zapišemo zaporedje stavkov. Lokalnost spremenljivk pomeni, da jih lahko uporabljajo le tisti podprogrami, v katerih so spremenljivke deklarirane.

Tako kot v jeziku java metode, lahko v jeziku PL/SQL preobtežujemo podprograme. V enem paketu imamo torej lahko več funkcij ali procedur z enakim imenom. Procedure (ali funkcije) se morajo med seboj razlikovati v tipu in/ali v številu parametrov, ki jih sprejmejo.

Poglejmo si, kako napišemo funkcijo. To bomo pokazali kar na primeru, kjer bomo v komentarje zapisali ustrezna pojasnila glede sestavljanja funkcij.

PRIMER 1:

Sestavimo funkcijo `dodaj`, ki za parameter prejme niz in vrne niz 'NULL', če je podana vrednost ničelna. To pomeni, da funkcijo `dodaj` kličemo brez parametra. V nasprotnem primeru naj funkcija vrne niz, ki je opremljen z enojnimi narekovaji.

To funkcijo bomo uporabili pri razvijanju našega programa. V razdelku 3 bomo razložili, zakaj to funkcijo uporabljamo.

```
-- S FUNCTION povemo, da gre za funkcijo, ki prejme parameter
-- tipa VARCHAR2 in vrne rezultat istega tipa.
FUNCTION dodaj (vrednost VARCHAR2)
   RETURN VARCHAR2
-- Z AS označimo začetek deklaracijskega dela funkcije.
-- AS ne smemo spustiti, tudi, če nimamo stavkov za deklaracijski del.
AS
-- Z BEGIN označimo začetek telesa funkcije.
BEGIN

   -- Če je vrednost parametra ničelna, vrnemo niz 'NULL'.
   IF (vrednost IS NULL)
   THEN
      RETURN 'NULL';
```

```
   -- Če vrednost ni ničelna, jo opremimo z ', kot smo razložili
   -- v razdelku 2.2.2.1.
   ELSE
      RETURN REPLACE('"'||vrednost||'"', '"', '''');
   END IF;

-- Z END označimo konec funkcije.
END dodaj;
```

Sestava procedure je podobna, le da ne uporabimo stavka RETURN. Tudi tu si bomo sintakso ogledali kar na primeru.

PRIMER 2:

Sestavimo proceduro `stavek_delete`, ki za parameter prejme datum in v datoteko zapiše ustrezne stavke DELETE. To stori le za tiste vrstice tabele formul, ki so se spremenile po danem datumu. Procedura deluje na podlagi predpostavke, da je v stolpcu `id1094_formula` vrednost tipa VARCHAR2. V proceduri uporabimo tudi vgrajen paket UTL_FILE, ki vsebuje funkcije in procedure s katerimi pišemo v datoteko. Paket UTL_FILE je predstavljen v razdelku 2.2.7.
Tudi to proceduro bomo z nekaj spremembami uporabili v našem programu.

```
-- S PROCEDURE povemo, da gre za proceduro, ki prejme parameter tipa DATE.
PROCEDURE stavek_delete (datum DATE)
-- Tudi v proceduri imamo deklaracijski del.
AS
    -- Lokalne spremenljivke s katerimi določimo: začetni del stavka DELETE,
    -- spremenljivko tipa CURSOR, ki hrani tabelo formul, datotečno
    -- spremenljivko, ime imenika in ime datoteke, kamor
    -- shranimo stavke DELETE.
    stavek_delete VARCHAR2 (1000)
                  := 'DELETE FROM kp1094_formule WHERE id1094_formula = ';

    CURSOR cur_formule
    IS
       SELECT id1094_formula, va1094_sestav, de1094_naziv, va1094_opis1,
                va1094_opis2, tr0000_user_kreir, tr0000_dat_kreir,
                tr0000_user_azur, tr0000_dat_azur, tr0000_uporaba
         FROM kp1094_formule
         WHERE tr0000_dat_azur >= datum;

    dat_zapis UTL_FILE.FILE_TYPE;
    imenik VARCHAR2(20) := 'c:\prenos';
    ime_dat VARCHAR2(20) := 'zap_dat.txt';

BEGIN
   -- Odpremo datoteko, v katero bomo zapisovali stavke DELETE.
   dat_zapis := UTL_FILE.FOPEN(imenik, ime_dat, 'w');

   -- Premikamo se po vrsticah in dopolnjujemo stavke DELETE
   -- z vrednostmi. Ker morajo biti vrednosti opremljene z narekovaji,
   -- uporabimo funkcijo dodaj.
   FOR tek_vrstica IN cur_formule LOOP
     UTL_FILE.PUT_LINE(dat_zapis,
                       stavek_delete || dodaj(tek_vrstica.id1094_formula)
                       ||';' );

     -- Med posameznimi stavki DELETE vstavimo prazno vrstico.
     UTL_FILE.NEW_LINE(dat_zapis,1);
```

```
   END LOOP;
   -- Zaprimo datoteko za zapisovanje.
   UTL_FILE.FCLOSE(dat_zapis);

-- Konec procedure
END stavek_delete;
```

Če želimo ustrezno »samostojno« proceduro shraniti v bazo, moramo izvesti SQL ukaz CREATE OR REPLACE PROCEDURE. Ta je oblike:

```
CREATE OR REPLACE PROCEDURE ime (parametri)
AS
  -- Deklaracijski del.
BEGIN
  -- Stavčni del.
END ime;
```

S tem ustvarimo samostojno proceduro `ime`. Del REPLACE dodamo za primer, če procedura s tem imenom slučajno že obstaja v bazi. S tem jo nadomestimo z novo različico. Če proceduro shranimo v bazo le enkrat, del REPLACE ni nujen. Ker pa procedure običajno večkrat popravljamo, del REPLACE uporabljamo praktično vedno. Z istim ukazom ustvarimo tudi samostojno funkcijo.

Kot vidimo, se samostojna procedura (in funkcija) od običajne razlikuje le v dodanem ukazu CREATE OR REPLACE.

2.2.6.2 Paket

Paket podobno kot knjižnica v javi pomeni način organizacije med sabo povezanih procedur in funkcij v skupine. Paket je sestavljen iz dveh delov: glave in telesa paketa. Oba dela se ločeno shranita v bazo. V glavo paketa zapišemo globalne spremenljivke in sestavljene tipe. Globalnost pomeni, da jih lahko uporabljajo vsi podprogrami, ki so del telesa paketa. V primeru, ko bi za prenos podatkov morali povezati preko parametrov preveč podprogramov, lahko uporabimo globalne spremenljivke.

V glavo paketa zapišemo tudi glave podprogramov, za katere želimo, da so »vidni« (jih torej lahko kličejo) tudi drugim programom. Te imenujemo globalni podprogrami.

Poglejmo si primer, kjer bomo podprograma `stavek_delete` in `dodaj` iz razdelka 2.2.6 shranili v paket `pripravi_dodaj`.

GLAVA PAKETA:

S CREATE OR REPLACE PACKAGE ustvarimo glavo paketa. Del REPLACE dodamo za primer, če v bazi slučajno že obstaja glava paketa s tem imenom.

```
CREATE OR REPLACE PACKAGE pripravi_dodaj
AS
   -- Glavi funkcije in procedure.
   FUNCTION dodaj (vrednost VARCHAR2)
      RETURN VARCHAR2;

   PROCEDURE stavek_delete (datum DATE);
END pripravi_dodaj;
```

TELO PAKETA:

S CREATE OR REPLACE PACKAGE BODY ustvarimo telo paketa. Del REPLACE dodamo za primer, če v bazi, slučajno že obstaja glava paketa s tem imenom.

```
CREATE OR REPLACE PACKAGE BODY pripravi_dodaj
AS
   -- Funkcija, ki vrne niz opremljen z narekovaji na začetku in koncu.
   FUNCTION dodaj (vrednost VARCHAR2)
      RETURN VARCHAR2
   AS
   BEGIN
      -- Če je vrednost parametra ničelna, vrnemo niz 'NULL'.
      IF (vrednost IS NULL)
      THEN
         RETURN 'NULL';

      -- Če vrednost ni ničelna, jo opremimo z ', kot smo razložili
      -- v razdelku 2.2.2.1.
      ELSE
         RETURN REPLACE('"'||vrednost||'"', '"', '''');
      END IF;
   END dodaj;


   -- Procedura, ki sestavi in zapiše stavek DELETE v datoteko.
   PROCEDURE stavek_delete (datum DATE)
   AS
      -- Lokalne spremenljivke s katerimi določimo: začetni del stavka DELETE,
      -- spremenljivko tipa CURSOR, ki hrani tabelo formul, datotečno
      -- spremenljivko, ime imenika in ime datoteke, kamor se
      -- shranijo stavki DELETE.
      stavek_delete VARCHAR2(1000)
                    := 'DELETE FROM kp1094_formule WHERE id1094_formula = ';

      CURSOR cur_formule
      IS
         SELECT id1094_formula, va1094_sestav, de1094_naziv, va1094_opis1,
                va1094_opis2, tr0000_user_kreir, tr0000_dat_kreir,
                tr0000_user_azur, tr0000_dat_azur, tr0000_uporaba
           FROM kp1094_formule
          WHERE tr0000_dat_azur >= datum;

      dat_zapis UTL_FILE.FILE_TYPE;
      imenik VARCHAR2(20) := 'c:\prenos';
      ime_dat VARCHAR2(20) := 'zap_dat.txt';

   BEGIN
      -- Odpremo datoteko, v katero bomo zapisovali stavke DELETE.
      dat_zapis := UTL_FILE.FOPEN(imenik, ime_dat, 'w');

      -- Premikamo se po vrsticah in dopolnjujemo stavke DELETE
      -- z vrednostmi. Ker morajo biti vrednosti opremljene z narekovaji,
      -- uporabimo funkcijo dodaj.
      FOR tek_vrstica IN cur_formule LOOP
         UTL_FILE.PUT_LINE(dat_zapis,
                           stavek_delete || dodaj(tek_vrstica.id1094_formula)
                           || ' ;' );

         -- Med posameznimi stavki DELETE vstavimo prazno vrstico.
         UTL_FILE.NEW_LINE(dat_zapis,1);
```

```
      END LOOP;
      -- Zaprimo datoteko za zapisovanje.
      UTL_FILE.FCLOSE (dat_zapis);

  END stavek_delete;
END pripravi_dodaj;
```

Procedure in funkcije paketa v (na primer) javanskem programu kličemo s `pripravi_dodaj.stavek_delete(datum)`.

2.2.6.3 Prožilec

Tako kot podprogram tudi prožilec predstavlja neko programsko kodo, ki je shranjena v podatkovni bazi. Prožilec se z razliko od podprograma izvede avtomatično, ko izvedemo določen stavek jezika DML ali DDL. Ker bomo v diplomski nalogi (zaključek) uporabljali le prožilce, ki se izvedejo, ko se nad neko tabelo izvedejo stavki DELETE, INSERT ali UPDATE, bomo v tem razdelku govorili le o njih. Z uporabo prožilcev lahko avtomatično izvedemo stavke SQL, stavke PL/SQL ali podprograme (samostojne ali del paketa). Za eno tabelo določimo le en prožilec.

Razvijalec v podjetju uporablja prožilce (za vse tabele) v razvojni in strankini bazi v dva namena. Za primer vzemimo prožilec za tabelo formul.

- Prožilec, ki je namenjen za vodenje evidence po pomoti izbrisanih vrstic table formul. Ta prožilec se izvede, ko se nad tabelo formul izvede stavek DELETE. Prožilec nato v »pomožno« tabelo, ki je namenjena za shranjevanje izbrisanih vrstic, z ukazom INSERT vstavi izbrisano vrstico. Pomožna tabela ima enako strukturo kot tabela formul, torej enako število in enako poimenovane stolpce. V stolpce v pomožni tabeli se shranijo iste vrednosti, kot so v tabeli formul, z izjemo vrednosti dveh stolpcev. Izjema sta stolpec `tr0000_user_azur`, kamor shranimo novo ime uporabnika, ki je to vrstico spreminjal (v našem primeru izbrisal) in stolpec `tr0000_dat_azur`, kamor shranimo nov datum, ko je bila vrstica spremenjena (v našem primeru izbrisana). Ime uporabnika, ki je izbrisal vrstico določimo s funkcijo USER. Ta vrne niz, ki predstavlja ime »trenutnega« uporabnika. Datum, ko je bila vrstica izbrisana pa določimo s funkcijo SYSDATE, ki vrne »trenutni« datum in čas.

- Prožilec, ki se izvede, ko se nad tabelo formul izvede stavek UPDATE ali stavek INSERT. V tem primeru prožilec v določenih stolpcih doda (s stavkom INSERT) ali popravi (s stavkom UPDATE) vrednosti. Ti določeni stolpci, ki so v vseh tabelah so: `tr0000_user_kreir` (ime uporabnika, ki je kreiral vrstico), `tr0000_dat_kreir` (datum, ko je vrstica dodana), `tr0000_user_azur` (ime uporabnika, ki je popravil vrstico) in `tr0000_dat_azur` (datum spremembe vrstice). Ime »trenutnega« uporabnika in »trenutni« datum prav tako določimo s funkcijama USER in SYSDATE.

V tem razdelku bomo predstavili prožilec, ki ga bomo uporabili v razdelku 5.

PRIMER:

Denimo, da bi potrebovali prožilec, ki izvede »glavni« podprogram paketa `pripravi`, ko v tabeli formul izvedemo stavek INSERT ali UPDATE.

```
-- S CREATE OR REPLACE USTVARIMO prožilec.
CREATE OR REPLACE TRIGGER prozilec_formule
   -- Prožilec se izvede po tem ko že popravimo ali dodamo vrstico
   -- tabele formul.
   AFTER INSERT OR UPDATE ON kp1094_formule
BEGIN
   -- Pokličemo "glavno" proceduro paketa pripravi,
   -- ki sestavi in zapiše stavka DELETE in INSERT v datoteko.
   pripravi.zapis_v_dat(SYSDATE, 'KP1094_FORMULE');
END prozilec_formule;
```

Rezervirano besedo AFTER uporabimo, ker želimo, da se prožilec izvede po tem, ko stavek INSERT (ali UPDATE) doda (ali popravi) vrstico v tabelo. Rezervirano besedo BEFORE pa bi uporabili, če bi želeli, da bi se prožilec izvedel preden bi dodali ali popravili vrstico v tabeli.

2.2.7 Paket UTL_FILE

Poleg paketov, ki jih ustvari razvijalec, uporabljamo tudi vgrajene pakete. Ti so sestavni del jezika PL/SQL. Njihovo ime v kodi običajno pišemo z velikimi črkami.

Vgrajen paket UTL_FILE je sestavljen iz podprogramov, ki jih uporabljamo za zapisovanje, dodajanje besedila (nizov) v datoteko in za branje besedila iz datotek. Ogledali si bomo le tiste podprograme, ki jih bomo potrebovali pri pisanju našega programa. Primer njihove uporabe bomo predstavili tako, da bomo podrobneje komentirali proceduro `stavek_delete`, ki smo jo navedli v razdelku 2.2.6.1.

1. Funkcija FOPEN odpre datoteko za pisanje, dodajanje ali branje. Funkcija ima tri parametre. Prvi je ime datoteke, drugi imenik, v kateri je (bo) datoteka, tretji parameter pa je oznaka, ki določa način odpiranja datoteke. Za zapis podatkov v datoteko uporabimo oznako 'w', za dodajanje oznako 'a' in za branje podatkov iz datoteke oznako 'r'.

Funkcija FOPEN vrne logično oznako datoteke. Shranimo jo v spremenljivko tipa UTL_FILE.FILE_TYPE. Na to spremenljivko se nato v programu sklicujemo, ko beremo ali zapisujemo besedilo iz oziroma v datoteko.

```
-- Spremenljivka tipa FILE_TYPE
dat_zapis UTL_FILE.FILE_TYPE;

-- Funkcija FOPEN odpre datoteko Zapis.txt.  V datoteko bomo zapisovali
-- besedilo. Datoteka bo shranjena v imeniku c:\prenos.
dat_zapis := UTL_FILE.FOPEN(imenik, ime_dat, 'w');
```

2. Procedura PUT_LINE zapiše niz, dan kot vrednost drugega parametra v datoteko, katere logična oznaka je prvi parameter procedure.

```
-- Procedura PUT_LINE zapiše vrednost spremenljivke vrednost
-- v datoteko Zapis.txt.
UTL_FILE.PUT_LINE(dat_zapis, vrednost);
```

3. Procedura NEW_LINE v datoteko, katere logična oznaka je prvi parameter, zapiše nekaj praznih vrst. Z vrednostjo drugega parametra povemo, koliko je teh praznih vrstic .

```
-- Procedura NEW_LINE v datoteko Zapis.txt
-- zapiše eno prazno vrstico.
UTL_FILE.NEW_LINE(dat_zapis,1);
```

4. Procedura FCLOSE zapre datoteko. V enem programu lahko uporabimo več datotek za zapis ali branje. Besedilo se nikoli ne zapisuje neposredno v datoteko, ampak v medpomnilnik. Šele, ko je medpomnilnik poln, se vsebina zapiše v datoteko in nato se medpomnilnik sprazni. Z zapiranjem datotek dosežemo, da se vsebina medpomnilnika zapiše v datoteko, kljub temu, da ta ni bil še čisto prazen. Datoteke zapiramo tudi zato, ker s tem sprostimo sistemske vire, ki jih vsaka odprta datoteka (katere ne potrebujemo) troši.

```
-- Procedura zapre datoteko Zapis.txt.
UTL_FILE.FCLOSE(dat_zapis);
```

DIPLOMSKA NALOGA :
FAKULTETA ZA MATEMATIKO IN FIZIKO

3 IDEJA PROGRAMA

V tem razdelku si bomo ogledali, kako smo razvili program, ki zapiše stavke za nadgradnjo baze v datoteko, ki se nato skupaj z novo verzijo sistema pošlje stranki. Ta program je sestavljen iz več podprogramov, ki so shranjeni skupaj v paketu. Paket bo shranjen v razvojni bazi, saj ga bo uporabljal (izvajal) le razvijalec v podjetju.

Predstavimo idejo programa v treh točkah.

1. Branje podatkov

Program mora prebrati tiste vrstice iz tabel skupine B v razvojni bazi, v katerih je po datumu objave prejšnje verzije sistema prišlo do sprememb. Spomnimo se, da so v skupini B tiste tabele, pri katerih za njihovo nadgradnjo skrbi razvijalec v podjetju.

Za branje podatkov iz tabele potrebujemo spremenljivko tipa CURSOR, ki smo jo predstavili v razdelku 2.2.2.1. Ker posamezna spremenljivka tega tipa lahko naenkrat vsebuje le eno tabelo, potrebujemo 9 takšnih spremenljivk, za vsako tabelo eno. Kot primer si oglejmo spremenljivko, ki jo bomo uporabili za rezultate iz tabele formul. Vanjo bomo shranili tiste vrstice tabele formul, kjer bo datum njihove spremembe za vrednostjo spremenljivke `datum`. Slednja predstavlja datum objave prejšnje verzije sistema.

```
CURSOR cur_formule
      IS
         SELECT id1094_formula, va1094_sestav, de1094_naziv, va1094_opis1,
                va1094_opis2, tr0000_user_kreir, tr0000_dat_kreir,
                tr0000_user_azur, tr0000_dat_azur, tr0000_uporaba
           FROM kp1094_formule
           WHERE tr0000_dat_azur >= datum;
```

Ko se ta stavek izvede, imamo preko spremenljivke `cur_formule` dostop do vseh vrstic, ki jih moramo spremeniti tudi v strankini bazi.

Zato bomo stavke, ki jih moramo izvesti na korakih 2 in 3, izvedli v zanki FOR, ki bo pretekla vse vrstice te tabele. V teh dveh korakih bomo stavke DELETE in INSERT sestavili in jih zapisali v datoteko.

```
FOR tek_vrstica IN cur_formule LOOP
        -- Sestavljanje in zapisovanje stavkov DELETE in INSERT v datoteko.
END LOOP;
```

2. Oblikovanje stavkov za nadgradnjo

Kot smo omenili že v razdelku 1, za dodajanje novih vrstic v tabelo uporabimo stavek INSERT. Če želimo podatke v vrstici le popraviti, uporabimo stavek UPDATE. Žal pa se pri stavku UPDATE pojavi težava, saj ne vemo, kateri podatek v določeni vrstici se je spremenil. Datum spremembe (datum ažuriranja) zabeleži le, kdaj se je zgodila sprememba v vrstici, ne pa tudi, kateri podatek se je spremenil. Tako bi morali napisati stavek UPDATE za vsak podatek v vrstici, kar pa ni najboljša rešitev, saj bi jih morali napisati veliko, za vsak stolpec svojega. Zato bomo namesto stavka UPDATE uporabili kar stavek INSERT. Seveda pa bomo morali najprej vse vrstice, ki so bile po datumu zadnje verzije dodane oz. spremenjene, izbrisati. Temu sledi vnos ustrezno popravljenih podatkov v tabelo. Kot prvi ukaz bomo izvedli stavek DELETE in nato stavek INSERT. Tudi v primeru, ko podatka še ni v strankini tabeli (ko želimo torej ustrezno vrstico dodati), bo ta postopek deloval. Če namreč izvedemo stavek DELETE in ustrezne vrstice v tabeli ni, se ne zgodi nič.

Seveda se morata ta stavka DELETE in INSERT sestaviti s pomočjo programske kode, ki bo iz tabele, ki jo hrani spremenljivka tipa CURSOR, pobrala ustrezne vrednosti. Omenjene stavke bomo v programski kodi zapisali v niz. Ta bo sestavljen iz ukazov, imena tabele, imena stolpcev, spremenljivk in nizov, s katerimi moramo opremiti posamezno vrednost.

Za pravilno sestavo stavkov za nadgradnjo moramo poznati imena in tipe stolpcev ter primarni ključ. Primarni ključ potrebujemo za pravilno oblikovanje stavka DELETE, kjer v delu WHERE poiščemo vrstice z določenim primarnim ključem. Pri nekaterih tabelah je primarni ključ sestavljen. Takrat moramo paziti, da zapišemo vse ustrezne stolpce.

Podatki o zgradbi posameznih tabel so spet shranjeni v tabelah. Posamezna tabela opisuje strukturo določene tabele. Primer take tabele je na Sliki 10, kjer vidimo strukturo tabele formul.

Podatki v tabeli formul so lahko tipa VARCHAR2 in DATE, kot vidimo v stolpcu Data Type iz Slike 10. Tabela formul ne uporablja stolpcev, ki bi bili tipa NUMBER. Je pa ta tip uporabljen v drugih tabelah, zato ga moramo pri pisanju programskih stavkov upoštevati.

KP1094_FORMULE:

Column Name	ID	Pk	Null?	Data Type
ID1094_FORMULA	1	1	N	VARCHAR2 (10)
VA1094_SESTAV	2		Y	VARCHAR2 (350)
DE1094_NAZIV	3		N	VARCHAR2 (60)
VA1094_OPIS1	4		Y	VARCHAR2 (200)
VA1094_OPIS2	5		Y	VARCHAR2 (200)
TR0000_USER_KREIR	6		Y	VARCHAR2 (30)
TR0000_DAT_KREIR	7		Y	DATE
TR0000_USER_AZUR	8		Y	VARCHAR2 (30)
TR0000_DAT_AZUR	9		Y	DATE
TR0000_UPORABA	10		Y	VARCHAR2 (1)

imena stolpcev | primarni ključ | tipi stolpcev

Slika 10 : Struktura tabele formul

Kot primer si poglejmo, kako sestavimo ustrezne stavke za nadgradnjo tabele formul.

PRIMER:

Za sestavo stavka DELETE potrebujemo primarni ključ (ključe). Tega razberemo iz tabele, ki predstavlja strukturo tabele formul. Na Sliki 10 vidimo v stolpcu Pk, da primarni ključ predstavlja le stolpec ID1094_FORMULA.
Potrebujemo tudi vrednost primarnega ključa vrstice, v kateri je prišlo do spremembe. To vrednost preberemo iz tabele, ki jo hrani spremenljivka tipa CURSOR. Kot smo omenili, te stavke, katerih sestavljanje opisujemo, izvajamo z zanko FOR. Posamezno vrednost tega stolpca v zanki FOR preberemo s `tek_vrstica.id1094_formula`.

```
stavek_delete := 'DELETE FROM kp1094_formule '
                 'WHERE id1094_formula = '
                 ||dodaj(tek_vrstica.id1094_formula)||
                 ';';
```

Vrednost, ki jo preberemo s `tek_vrstica.id1094_formula`, je tipa VARCHAR2, zato jo moramo opremiti z enojnimi narekovaji. To naredimo s klicem funkcije `dodaj`, ki smo jo opisali v razdelku 2.2.6.1. Na koncu niza DELETE dodamo tudi podpičje, saj se mora stavek zaključiti z njim.

Program stavek DELETE v datoteko zapiše v naslednji obliki:

```
DELETE FROM kp1094_formule WHERE id1094_formula = '0003' ;
```

Za sestavo stavka INSERT uporabimo vsa imena stolpcev iz Slike 10 in vse njihove vrednosti. Tako kot pri stavku DELETE preberemo vrednosti iz tabele, ki jo hrani spremenljivka tipa CURSOR.

```
stavek_insert :=
    'INSERT INTO kp1094_formule
                (id1094_formula, va1094_sestav, de1094_naziv,
                 va1094_opis1, va1094_opis2, tr0000_user_kreir,
                 tr0000_dat_kreir, tr0000_user_azur,tr0000_dat_azur,
                 tr0000_uporaba)
        VALUES('||dodaj(tek_vrstica.id1094_formula)||','
               ||dodaj(tek_vrstica.va1094_sestav)||','
               ||dodaj(tek_vrstica.de1094_naziv)||','
               ||dodaj(tek_vrstica.va1094_opis1)||','
               ||dodaj(tek_vrstica.va1094_opis2)||','
               ||dodaj(tek_vrstica.tr0000_user_kreir)||','
               ||dodaj(tek_vrstica.tr0000_dat_kreir)||','
               ||dodaj(tek_vrstica.tr0000_user_azur)||','
               ||dodaj(tek_vrstica.tr0000_dat_azur)||','
               ||dodaj(tek_vrstica.tr0000_uporaba||') ;' ;
```

V tem stavku INSERT so poleg vrednosti tipa VARCHAR2 tudi vrednosti tipa DATE. To sta vrednosti, ki ju preberemo iz stolpca `tr0000_dat_kreir` in stolpca `tr0000_dat_azur`. Zato potrebujemo poleg funkcije `dodaj`, ki opremi vrednosti tipa VARCHAR2, še eno funkcijo z istim imenom. Ta prejme parameter tipa DATE in vrne niz, ki predstavlja bodisi niz 'NULL' bodisi klic funkcije, ki to vrednost pretvori nazaj v tip DATE. Ne smemo pa pozabiti, da lahko druge tabele vsebujejo tudi vrednosti tipa NUMBER. Zato potrebujemo še tretjo funkcijo `dodaj`, ki prejme parameter tipa NUMBER. Funkcija dodaj bo torej preobtežena, saj se pod njenim imenom »skrivajo« tri funkcije. Funkciji, ki prejmeta parameter tipa DATE in NUMBER, bomo predstavili v razdelku 3.1.

Program stavek INSERT v datoteko zapiše v naslednji obliki:

```
INSERT INTO kp1094_formule
            (id1094_formula, va1094_sestav, de1094_naziv,
            va1094_opis1, va1094_opis2, tr0000_user_kreir,
            tr0000_dat_kreir,tr0000_user_azur,tr0000_dat_azur,
            tr0000_uporaba)
     VALUES ('0003', 'VP_KOL * VP_V1', 'Količina * vrednost', NULL, NULL,
            'PLACE01', TO_DATE('12.01.2009 14:13:01', 'DD.MM.YYYY
            HH24:MI:SS'), 'PLACE01',
            TO_DATE('12.01.2009 14:13:01', 'DD.MM.YYYY HH24:MI:SS'), NULL
            );
```

3. Zapisovanje stavkov za nadgradnjo (stavki DELETE in INSERT) v datoteko

Sedaj, ko vemo, kako morajo biti sestavljeni stavki DELETE in INSERT, si oglejmo, kako jih zapišemo v datoteko.

Za zapisovanje stavkov DELETE in INSERT v datoteko bomo uporabili paket UTL_FILE. Ta vsebuje ustrezne podprograme, ki omogočajo zapisovanje v datoteko. Predstavljen je v razdelku 2.2.7.

Poleg treh funkcij, ki smo jih omenili v točki 2, potrebujemo tudi procedure, ki bodo za posamezno tabelo (za vsako od 9 tabel skupine B) oblikovale stavke za nadgradnjo ter jih zapisale v datoteko.

3.1 Koda programa

Celoten program bomo zapisali v paket. Paket bomo poimenovali `pripravi`. Ta bo sestavljen iz 12 podprogramov (3 funkcij in 9 procedur). Ker pa so si vse procedure zelo podobne, bomo kot primer napisali le tisto, ki poskrbi za pripravo stavkov za ustrezno posodobitev tabele formul. Začnimo z glavo paketa.

3.1.1 Glava paketa

V glavo paketa zapišemo dve spremenljivki in štiri glave podprogramov. Vsi ti elementi so globalni, da jih lahko uporabimo oziroma kličemo tudi iz drugih programov.

```
CREATE OR REPLACE PACKAGE pripravi
AS
   -- Potrebujemo globalni spremenljivki, s katerima povemo, kako se
   -- imenuje datoteka, kamor zapisujemo stavke za nadgradnjo strankine
   -- baze in v katerem imeniku je ta datoteka shranjena.
   ime_dat VARCHAR2(15):= 'zap_dat.txt';
   imenik VARCHAR2(10):= 'c:\prenos';

   -- Glave treh funkcij, ki preverijo, katerega tipa so podatki
   -- in jih ustrezno oblikujejo.
   FUNCTION dodaj (vrednost VARCHAR2)
      RETURN VARCHAR2;

   FUNCTION dodaj (vrednost DATE)
      RETURN VARCHAR2;

   FUNCTION dodaj (vrednost NUMBER)
      RETURN VARCHAR2;

   -- Glava procedure, ki oblikuje in zapisuje stavke za nadgradnjo v
   -- datoteko.
   PROCEDURE pripravi_stavka (datum DATE);

END pripravi;
```

3.1.2 Telo paketa

V telo paketa zapišemo kodo funkcij in kodo procedure. Najprej si oglejmo vse tri funkcije, ki so del telesa paketa.

FUNKCIJE

1. Funkcija `dodaj` s parametrom tipa VARCHAR2.

Uporabimo funkcijo `dodaj`, ki smo jo opisali v razdelku 2.2.6.1.

```
CREATE OR REPLACE PACKAGE BODY pripravi
AS
   -- Funkcija vrne bodisi niz NULL ali z narekovaji opremljen niz v
   -- spremenljivki vrednost.
   FUNCTION dodaj (vrednost VARCHAR2)
      RETURN VARCHAR2
   AS
```

```
   BEGIN
      -- Če je vrednost parametra ničelna, vrnemo niz 'NULL'.
      IF (vrednost IS NULL)
      THEN
         RETURN 'NULL';
      -- Če vrednost ni ničelna jo opremimo z ', kot smo razložili
      -- v razdelku 2.2.2.1.
      ELSE
         RETURN REPLACE ('"'||vrednost||'"', '"', '''');
      END IF;
   END dodaj;
```

Primer podatka, ki ga vrne funkcija : `'VP_KOL * VP_V1'`

2. Funkcija `dodaj` s parametrom tipa DATE.

Funkcija `dodaj` preveri vrednost parametra tipa DATE. Če je vrednost ničelna, vrne niz 'NULL', v nasprotnem primeru pa vrne niz, ki predstavlja klic funkcije TO_DATE, ki to vrednost pretvori nazaj v tip DATE. Funkcija `dodaj` ne more kar vrniti vrednost parametra, saj je ta tipa DATE, funkcija pa mora vrniti niz.

Vrednost moramo najprej s funkcijo TO_CHAR pretvoriti v niz (v obliki datuma s časom), saj bi v nasprotnem primeru iz podane vrednosti prebrali le datum (brez časa). Funkcijo TO_CHAR smo predstavili v razdelku 2.2.3.

```
   -- Funkcija, ki za parameter dobi vrednost tipa DATE in vrne niz,
   -- ki predstavlja bodisi niz 'NULL' bodisi klic funkcije, ki to
   -- vrednost pretvori nazaj v tip DATE.
   FUNCTION dodaj (vrednost DATE)
      RETURN VARCHAR2
   AS
   BEGIN
      -- Če je vrednost parametra ničelna vrnemo niz 'NULL'.
      IF (vrednost IS NULL)
      THEN
         RETURN 'NULL';

      -- Če vrednost ni ničelna vrnemo vrednost, ki predstavlja klic
      -- funkcije TO_DATE, ki to vrednost pretvori nazaj v tip DATE.
      ELSE
         RETURN REPLACE('TO_DATE(
                        "'||TO_CHAR(vrednost, 'DD.MM.YYYY HH24:MI:SS')||'",
                        "DD.MM.YYYY HH24:MI:SS")','"','''');
      END IF;
   END dodaj;
```

Primer podatka, ki ga vrne funkcija :
`'TO_DATE('04.01.2007 14:13:01', 'DD.MM.YYYY HH24:MI:SS')'`

3. Funkcija `dodaj` s parametrom tipa NUMBER.

Funkcija `dodaj` preveri vrednost parametra tipa NUMBER. Če je vrednost ničelna, vrne niz 'NULL', v nasprotnem primeru pa s pomočjo funkcije TO_CHAR število pretvori v niz in ga vrne. Ko število pretvorimo v niz, se namesto decimalne pike zapiše vejica. Kot vemo (glejte razdelek 2.2.2.1) decimalna mesta označujemo s piko in ne z vejico. Da rešimo ta problem, uporabimo funkcijo REPLACE, ki smo jo predstavili v razdelku 2.2.2.1. S funkcijo REPLACE v primeru, da je podano število decimalno, zamenjamo vejico s piko.

```
-- Funkcija podatek pretvori v niz. Če je vrednost parametra ničelna,
-- vrne niz 'NULL'.
FUNCTION dodaj (vrednost NUMBER)
   RETURN VARCHAR2
AS
BEGIN
   -- Če je vrednost parametra ničelna, vrnemo niz 'NULL'.
   IF (vrednost IS NULL)
   THEN
      RETURN 'NULL';
   -- Če vrednost ni ničelna, vrnemo vrednost, v kateri
   -- zamenjamo vejico s piko.
   ELSE
      RETURN REPLACE(TO_CHAR(vrednost,',','.'));
   END IF;
END dodaj;
```

Primer podatka, ki ga vrne funkcija : `'125.33'`

PROCEDURA

Spomnimo se procedure `stavek_delete` iz razdelka 2.2.6.1, ki sestavi in zapisuje stavke DELETE v datoteko. To proceduro le nadgradimo, tako, da procedura poleg stavkov DELETE sestavlja tudi stavke INSERT. Poimenujemo jo `pripravi_stavka`.

```
-- Procedura, ki za parameter dobi datum prejšnje verzije sistema
-- in sestavi ustrezne stavke DELETE in INSERT.
PROCEDURE pripravi_stavka (datum DATE)
AS
   -- Lokalne spremenljivke, s katerimi določimo začetne dele
   -- stavkov DELETE in INSERT, spremenljivka, ki hrani tabelo formul
   -- in datotečno spremenljivko s katero določimo datoteko za zapis
   -- stavkov.
   stavek_insert VARCHAR2(1000)
      := 'INSERT INTO kp1094_formule(id1094_formula,va1094_sestav,
          de1094_naziv,va1094_opis1,va1094_opis2,tr0000_user_kreir,
          tr0000_dat_kreir,tr0000_user_azur,tr0000_dat_azur,
          tr0000_uporaba)VALUES(';

   stavek_delete VARCHAR2(600)
                := 'DELETE FROM kp1094_formule WHERE id1094_formula = ';

   CURSOR cur_formule
   IS
      SELECT id1094_formula, va1094_sestav, de1094_naziv, va1094_opis1,
             va1094_opis2, tr0000_user_kreir, tr0000_dat_kreir,
             tr0000_user_azur, tr0000_dat_azur, tr0000_uporaba
        FROM kp1094_formule
        WHERE tr0000_dat_azur >= datum;

   dat_zapis UTL_FILE.FILE_TYPE;

BEGIN
   -- Odpremo datoteko za zapisovanje stavkov.
   dat_zapis := UTL_FILE.FOPEN(imenik, ime_dat, 'w');

   -- Premikamo se po vrsticah in dopolnjujemo začetne dele stavkov
   -- DELETE in INSERT s podatki. Ker morajo biti podatki ustrezno
```

```
      -- opremljeni, uporabimo funkcije dodaj.
      FOR tek_vrstica IN cur_formule LOOP
         -- Sestavimo stavek DELETE in ga shranimo v datoteko.
         UTL_FILE.PUT_LINE(dat_zapis, stavek_delete
                           ||dodaj(tek_vrstica.id1094_formula)
                           || ';'
                           );

         -- Sestavimo stavek INSERT in ga shranimo v datoteko.
         UTL_FILE.PUT_LINE (dat_zapis, stavek_insert
                            ||dodaj(tek_vrstica.id1094_formula)
                            ||' , '
                            ||dodaj(tek_vrstica.va1094_sestav)
                            ||' , '
                            ||dodaj(tek_vrstica.de1094_naziv)
                            ||' , '
                            |dodaj(tek_vrstica.va1094_opis1)
                            ||' , '
                            ||dodaj(tek_vrstica.va1094_opis2)
                            ||' , '
                            ||dodaj(tek_vrstica.tr0000_user_kreir)
                            ||' , '
                            ||dodaj(tek_vrstica.tr0000_dat_kreir)
                            ||' , '
                            ||dodaj(tek_vrstica.tr0000_user_azur)
                            ||' , '
                            ||dodaj(tek_vrstica.tr0000_dat_azur)
                            ||' , '
                            ||dodaj(tek_vrstica.tr0000_uporaba)
                            ||' ) ;'
                           );

         -- Med posameznimi stavki vstavimo prazno vrstico.
       UTL_FILE.NEW_LINE(dat_zapis,1);
      END LOOP;

      -- Zaprimo datoteko.
      UTL_FILE.FCLOSE(dat_zapis);

   -- Konec procedure.
   END pripravi_stavka;

-- Konec paketa.
END pripravi;
```

Ne smemo pozabiti, da smo na začetku rekli, da je takih tabel 9. Zato tudi za ostale tabele na isti način za vsako napišemo po eno proceduro in jih zapišemo v paket `pripravi`.

4 IZBOLJŠAVA PROGRAMA

S programom, ki smo ga razvili z uporabo statično sestavljenih poizvedb, imamo pri vzdrževanju lahko več dela, kot smo pričakovali. Če se na primer spremeni struktura tabel, moramo popravljati kodo, pri čemer se lahko hitro zmotimo. Struktura tabele se lahko spremeni, če dodamo oziroma odstranimo stolpec, spremenimo tip stolpca ali spremenimo primarni ključ. Lahko se pojavijo tudi nove tabele v skupini B (tabele, ki jih nadgrajuje razvijalec), na primer, da jih v strankino bazo doda drug program. Takrat moramo v naš program dodati nove procedure, kar nam lahko vzame kar nekaj časa. Zato bi bilo boljše, da bi program izboljšali tako, da bi program sam ugotovil ustrezno strukturo baze in tabel ter opravil ustrezne popravke.

Preden bomo pogledali, kako lahko program izboljšamo, si najprej oglejmo nekatere teme v povezavi z bazami podatkov in jezikom PL/SQL, ki smo jih v prejšnjih razdelkih preskočili in nam lahko koristijo pri izboljšavi programa.

4.1 Slovar podatkov

Slovar podatkov je del podatkovne baze, kjer so shranjeni tako imenovani metapodatki. To so podatki, ki opisujejo strukturo tabel, procedur, funkcij itd., shranjenih v bazi. Metapodatki so na primer imena stolpcev in tabel, tipi stolpcev itd. Slovar podatkov se avtomatično spreminja, ko se baza spremeni (ko na primer v tabelo dodamo stolpec).

Metapodatki so razvrščeni in shranjeni v tabelah. Do teh podatkov dostopamo s pomočjo že določenih pogledov[1], ki so del sistema Oracle. Poglejmo si nekaj primerov pogledov, ki jih bomo potrebovali pri izboljšavi programa.

- user_tables : imena tabel
- user_tab_columns : imena tabel, imena stolpcev in tipi stolpcev
- user_cons_columns : imena tabel in njihovih stolpcev
- user_constraint : imena in oznake omejitev, kot sta primarni in tuji ključ. Primarni ključ ima oznako P, tuji ključ pa oznako R.

S predpono user povemo, da želimo dostopati do metapodatkov za tiste tabele v podatkovni bazi, za katere ima »trenutni« uporabnik dovoljenja za izvajanje stavkov SELECT, INSERT, DELETE itd. Če želimo dostopati do metapodatkov za tabele, do katerih nimamo omenjenih dovoljenj, uporabimo predpono dba. Mi bomo uporabili predpono user, saj želimo dostopati le do metapodatkov za tabele, za katere imamo omenjena dovoljenja.

PRIMER:

Denimo, da želimo dobiti metapodatke za imena, tipe in omejitve, ki so določene za posamezni stolpec tabele kp1094_formule. V ta namen potrebujemo zgoraj omenjene poglede, s pomočjo katerih bomo v nadaljevanju zapisali poizvedbo, ki bo vrnila omenjene metapodatke. To poizvedbo bomo potrebovali pri izboljšavi našega programa. Najprej pa si za posamezni pogled poglejmo zapis poizvedbe in njen rezultat za tabelo kp1094_formule.

[1] Pogled (view) : virtualna tabela, ki ne vsebuje podatkov, ampak skrbi za predstavitev podatkov v eni ali več tabelah.

1. user_tab_columns

Ustrezna poizvedba je:

```
SELECT column_name, data_type
  FROM user_tab_columns
 WHERE table_name = 'KP1094_FORMULE';
```

Jezik SQL je le v nekaterih primerih »občutljiv« na male in velike črke. Tak primer so imena tabel, ki jih zapišemo v niz, s katerim nato v poizvedbi poiščemo želene metapodatke. Imena tabel so v slovarju podatkov zapisana z velikimi črkami, zato moramo v takšni obliki zapisati tudi imena v nizu. Če bi imena tabel napisali z malimi črkami, poizvedba ne bi vrnila rezultata.

Poizvedbo lahko zapišemo tudi tako:

```
SELECT column_name
  FROM dba_tab_columns
  WHERE table_name = 'KP1094_FORMULE';
```

Z uporabo predpone dba poizvedba pregleda tudi tista imena tabel, katerih ne potrebujemo. Zato bomo, kot smo že omenili zgoraj, uporabljali le poglede s predpono user.

Obe poizvedbi vrneta:

COLUMN_NAME	DATA_TYPE
ID1094_FORMULA	VARCHAR2
VA1094_SESTAV	VARCHAR2
DE1094_NAZIV	VARCHAR2
VA1094_OPIS1	VARCHAR2
VA1094_OPIS2	VARCHAR2
TR0000_USER_KREIR	VARCHAR2
TR0000_DAT_KREIR	DATE
TR0000_USER_AZUR	VARCHAR2
TR0000_DAT_AZUR	DATE
TR0000_UPORABA	VARCHAR2

Slika 11: Tabela z imeni in tipi, ki so določena za posamezni stolpec tabele formul.

Dobili smo torej tabelo, ki vsebuje imena in tipe, ki so določeni za posamezni stolpec tabele kp1094_formule.

2. user_cons_columns

Sedaj potrebujemo poizvedbo, ki vrne tabelo z imeni stolpcev in imeni omejitev za tabelo kp1094_formule. Rezultat te poizvedbe potrebujemo zato, da bomo v nadaljevanju za posamezno ime stolpca dobili pripadajočo oznako omejitev. Te oznake vrne poizvedba, ki je predstavljena v naslednji točki 3.

Ustrezna poizvedba je:

```
SELECT column_name, constraint_name
  FROM user_cons_columns
  WHERE table_name = 'KP1094_FORMULE';
```

Poizvedba vrne:

COLUMN_NAME	CONSTRAINT_NAME
ID1094_FORMULA	KP1094_PK

Slika 12 : Tabela z imenom stolpca in imenom omejitve za tabelo formul.

3. user_constraints

Potrebujemo še poizvedbo, ki vrne tabelo z imeni omejitev in oznakami omejitev za posamezni stolpec tabele kp1094_formule.

Ustrezna poizvedba je:

```
SELECT constraint_name, constraint_type
  FROM user_constraints
 WHERE table_name = 'KP1094_FORMULE';
```

Poizvedba vrne:

CONSTRAINT_NAME	CONSTRAINT_TYPE
KP1094_PK	P

Slika 13: Tabela z imenom in oznako omejitve za tabelo formul.

Za izboljšavo našega programa bomo omenjene tabele morali združiti. Združimo jih v dveh korakih:

1. Združimo tabeli iz Slike 12 in Slike 13.
Tabeli združimo z notranjim združevanjem. Ta način združevanja uporabimo zato, ker želimo dobiti združeno tabelo samo s tistimi vrsticami, kjer se vrednosti polj v tabelah iz Slike 12 in Slike 13 v stolpcu `constraint_name` (ime omejitve) ujemata.

2. Združimo tabelo iz Slike 11 in združeno tabelo iz točke 1.
Tabeli združimo z levim zunanjim združevanjem, kjer za glavno (levo) tabelo določimo tabelo iz Slike 11. Združena tabela tako vsebuje vrstice, ki jih je vrnilo notranje združevanje in vrstice, ki nimajo ujemajoče vrstice v tabeli iz Slike 11. Ta način združevanja uporabimo zato, ker želimo v združeni tabeli dobiti vsa imena, tipe in oznake omejitev za posamezni stolpec tabele kp1094_formule. Tam kjer omejitve ni, vrstice iz leve tabele (točka 1) dobijo zraven »prazno« vrednost. Če bi uporabili notranje združevanje bi v našem primeru dobili tabelo z eno vrstico. Tisto, ki ima oznako omejitev.

Poglejmo si zapis poizvedbe in tabelo, ki jo omenjena poizvedba vrne.

```
SELECT column_name, data_type, omejitve.oznaka
   FROM user_tab_columns tab,
        (SELECT stolpec.column_name ime, omejitev.constraint_type oznaka
            FROM user_constraints omejitev, user_cons_columns stolpec
            WHERE stolpec.table_name = 'KP1094_FORMULE'
               AND omejitev.constraint_name = stolpec.constraint_name
        ) omejitve
   WHERE table_name = 'KP1094_FORMULE'
         AND tab.column_name = omejitve.ime(+);
```

Stavek SELECT smo napisali tako, da smo rezultat »notranjega« stavka SELECT poimenovali `omejitve`. »Zunanji« stavek SELECT tako vrne imena stolpcev in tipe stolpcev iz tabele poimenovane `tab` ter oznake omejitev, ki jih vrne stavek poimenovan `omejitve`.

Rezultat zgornje poizvedbe:

COLUMN_NAME	DATA_TYPE	OZNAKA
DE1094_NAZIV	VARCHAR2	
ID1094_FORMULA	VARCHAR2	P
TR0000_DAT_AZUR	DATE	
TR0000_DAT_KREIR	DATE	
TR0000_UPORABA	VARCHAR2	
TR0000_USER_AZUR	VARCHAR2	
TR0000_USER_KREIR	VARCHAR2	
VA1094_OPIS1	VARCHAR2	
VA1094_OPIS2	VARCHAR2	
VA1094_SESTAV	VARCHAR2	

Slika 14 : Združena tabela z metapodatki za tabelo formul.

Ta združena tabela vedno prikaže ažurne metapodatke za imena, tipe in omejitve, ki so določene za posamezni stolpec tabele kp1094_formule. S pomočjo te tabele bomo pri izboljšavi našega programa lahko ustrezno sestavili stavke DELETE in INSERT za tabelo kp1094_formule.

4.2 Paket DBMS_SQL

Vgrajen paket DBMS_SQL sestavljajo funkcije in procedure, katere uporabljamo za izvedbo stavkov jezika DDL in DML. Ti stavki so zapisani v nizu. Pri izboljšavi programa bomo uporabili le poizvedbe, katere tvorimo z jezikom DML. Te poizvedbe se sestavljajo med izvajanjem programa. To pomeni, da med pisanjem programa ne poznamo oblike poizvedbe oziroma vseh delov poizvedbe, kot jo poznamo pri statično sestavljenih poizvedbah. Zato rečemo, da so te poizvedbe sestavljene dinamično.

Sestavni del poizvedbe so lahko tudi spremenljivke. Te spremenljivke z razliko od spremenljivk v statično sestavljeni poizvedbi lahko vsebujejo tudi ukaze, imena stolpcev in imena tabel.

Spremenljivke v poizvedbi lahko obravnavamo na dva načina:

- Spremenljivke, v katerih so shranjeni ukazi, imena stolpcev in tabel, spojimo s preostalim nizom poizvedbe z operatorjem za spajanje (`||`). Te spremenljivke so lahko le tipa VARCHAR2.
- Spremenljivke, v katerih so shranjene vrednosti s katerimi primerjamo določene podatke v tabeli, pa v nizu poizvedbe označimo tako, da pred njihovim imenom uporabimo dvopičje (:). S tem napovemo, da se bo na to mesto v poizvedbi kasneje zapisala vrednost spremenljivke. Te spremenljivke so lahko tipa VARCHAR2, NUMBER in DATE.

Tako je niz

```
stavek_select VARCHAR2(100)
          := 'SELECT tr0000_dat_azur FROM '||
             ime_tabele||
             ' WHERE tr0000_dat_azur >= :dat';
```

sestavljen iz treh delov. Prvi del (SELECT … FROM) je niz, ki predstavlja »statični« del poizvedbe, saj smo v njem navedli le imena stolpcev (le ime stolpca `tr0000_dat_azur`). Drugi del niza dobimo iz spremenljivke `ime_tabele`, kjer kot niz shranimo ime neke tabele, v tretjem delu (WHERE) pa nastopa tudi ime spremenljivke tipa DATE (`dat`). Da povemo, da je na to mesto potrebno dati vrednost te spremenljivke, spredaj dodamo :.

Če je torej v spremenljivki `ime_tabele` shranjen niz: `'kp1094_formule'` in v spremenljivki `dat` datum: `TO_DATE('01.01.2009', 'DD.MM.YYYY')` je vrednost zgornje spremenljivke `stavek_select` torej

```
SELECT tr0000_dat_azur FROM kp1094_formule
     WHERE tr0000_dat_azur >= TO_DATE('01.01.2009','DD.MM.YYYY')
```

Niz s poizvedbo pa lahko napišemo tudi tako:

```
stavek_select VARCHAR2(100)
          := 'SELECT tr0000_dat_azur FROM '||
             ime_tabele||
             ' WHERE tr0000_dat_azur >= '||dat;
```

Poizvedbo, ki je zapisana na prvi način lahko npr. v nekem podprogramu (kjer je zgornja poizvedba deklarirana) sestavimo 1x in izvršimo večkrat. Pri tem lahko večkrat zamenjamo vrednost spremenljivke `dat` in poizvedbo ponovno izvedemo. To lahko storimo zato, ker imamo na voljo proceduro BIND_VARIABLE (sestavni del paketa DBMS_SQL), ki to omogoča. Te procedure pa ne moremo uporabiti pri poizvedbi, ki je zapisana na drugi način. Zato moramo to poizvedbo (drugi način) v primeru, da jo želimo izvesti večkrat v istem podprogramu (z drugimi vrednostmi spremenljivke `dat`) tudi večkrat sestaviti. Zaradi tega omenjeni podprogram deluje počasneje, kot če bi uporabili prvi način. Ker se v splošnem uporablja prvi način, bomo ta način uporabili tudi pri izboljšavi našega programa, kjer bi sicer lahko uporabili oba načina. Za sestavljanje in izvedbo poizvedbe uporabljamo podprograme paketa DBMS_SQL, ki jih bomo predstavili v nadaljevanju.

Preden pa si ogledamo primer uporabe dinamično sestavljene poizvedbe, predstavimo še en podatkovni tip, ki ga v razdelku 2.2.2.1 nismo omenili. Tudi spremenljivka tega tipa je lahko del poizvedbe.

Podatkovni tip ROWID:

Vsaka na novo dodana vrstica v tabelo avtomatično dobi enoličen naslov. Ta naslov je podan v obliki niza, na primer 'AAADnWAAbAAAIfSAAb'. S pomočjo vrednosti tipa ROWID lahko pridemo do

določene vrstice brez navajanja primarnega ključa v poizvedbi. V primeru, da je primarni ključ sestavljen, z uporabo tipa ROWID lažje (hitreje) pridemo do želene vrstice. Poglejmo si primer zapisa poizvedbe in tabelo, ki jo poizvedba vrne.

PRIMER:

Denimo, da bi imela tabela formul sestavljen primarni ključ. Poglejmo si dva načina zapisa poizvedbe, ki vrneta isto tabelo. Prva poizvedba vrne tabelo glede na primarni ključ, druga pa glede na vrednost tipa ROWID.

```
-- Spremenljivka tipa ROWID z enoličnim naslovom določene vrstice
-- tabele formul.
vred_rowid ROWID := 'AAADnWAAbAAAIfSAAb';

-- Poizvedba s pomočjo primarnega ključa vrne določeno vrstico.
SELECT id1094_formula, va1094_sestav, de1094_naziv
  FROM kp1094_formule
  WHERE id1094_formula = '0003' AND id1094_xxx = 110 AND ...;

-- Poizvedba s pomočjo rowid vrne določeno vrstico.
SELECT id1094_formula, va1094_sestav, de1094_naziv
  FROM kp1094_formule
  WHERE ROWID = vred_rowid;
```

Zgornji poizvedbi vrneta tabelo, ki je prikazana na Sliki 15.

ID1094_FORMULA	VA1094_SESTAV	DE1094_NAZIV
0003	VP_KOL * VP_V1	Kolicina * vrednost

Slika 15 : Vrstica tabele formul glede na vrednost tipa ROWID.

V nizu, ki predstavlja enoličen naslov so pomembne velike in male črke. V primeru, da bi v spremenljivko `vred_rowid` shranili niz: 'AAADNWAABAAAIFSAAB' (male črke zamenjamo z velikimi), potem zgornja poizvedba ne bi vrnila rezultata ali pa bi vrnila drugo vrstico tabele formul.

Vidimo, da ROWID označuje dve stvari. Enkrat označuje tip, drugič pa se v poizvedbah (v stavku SELECT) z njim sklicujemo na določeno vrstico.

Kot lahko opazimo, je veliko preprosteje priti do določene vrstice z uporabo vrednosti tipa ROWID. Zavedati pa se moramo, da je enoličen naslov iste vrstice v razvojni in strankini bazi različen. Zato ne moremo za nadgradnjo v strankini bazi uporabiti kar stavka

```
DELETE kp1094_formule WHERE ROWID = ' AAADnWAAbAAAIfSAAb';
```

saj ima vrstica, ki jo v strankini bazi želimo izbrisati, drugačen enoličen naslov, kot ista (ekvivalentna) vrstica v razvojni bazi.

Sedaj pa si oglejmo niz s poizvedbo, v kateri sta del niza spremenljivka z datumom in spremenljivka z imenom tabele. Ta poizvedba vrne enolične naslove vrstic podane tabele, ki jo bomo uporabili na spodnjem primeru. V poizvedbi moramo stolpec enoličnih naslovov poimenovati, drugače poizvedba ne vrne rezultata.

```
stavek_select VARCHAR2(100)
         := 'SELECT rowid id_vrstice FROM '||ime_tabele
            ||' WHERE tr0000_dat_azur >= :dat';
```

V nadaljevanju bomo predstavili podprograme paketa DBMS_SQL kar na primeru, kjer bomo v komentarje zapisali ustrezna pojasnila.

PRIMER:

Denimo, da bi potrebovali proceduro `delete_insert`, ki bi prejela dva parametra. S prvim določimo datum, z drugim pa ime tabele. Procedura sestavi in izvede poizvedbo, ki vrne vrednost tipa ROWID, torej enoličen naslov posamezne vrstice določene tabele. Tako poizvedbo smo predstavili zgoraj. Procedura nato s pomočjo funkcij `vrni_delete` in `vrni_insert`, zapiše stavke za nadgradnjo v datoteko. Omenjeni funkciji bomo predstavili v razdelku 4.4.2.

```
PROCEDURE delete_insert (datum DATE, ime_tabele VARCHAR2)
AS
   -- Spremenljivka z nizom poizvedbe.
   poizvedba VARCHAR2 (100)
         := 'SELECT rowid id_vrstice FROM '||ime_tabele
            ||' WHERE tr0000_dat_azur >= :dat';

   -- Spremenljivka v katero shranimo niz s stavkom DELETE.
   stavek_delete VARCHAR2(600);

   -- Spremenljivka v katero shranimo niz s stavkom INSERT.
   stavek_insert VARCHAR2(1000);

   -- Spremenljivka v katero shranimo vrednosti, ki jo vrne poizvedba.
   vred_rowid ROWID;

   -- Spremenljivka, v katero shranimo zaporedno številko prostora
   -- v pomnilniku, kamor bomo shranili rezultat poizvedbe. To
   -- spremenljivko bomo v nalogi označevali s predpono c.
   c_rowid NUMBER;

   -- Spremenljivka v katero shranimo število vrstic tabele,
   -- ki jo vrne poizvedba.
   izvrsitev NUMBER;

BEGIN
   -- Funkcija OPEN_CURSOR rezervira prostor v pomnilniku, kamor se
   -- shrani rezultat poizvedbe. Funkcija vrne zaporedno številko
   -- s katero je ta prostor v pomnilniku poimenovan in jo shrani
   -- v spremenljivko tipa NUMBER. S to spremenljivko (kot prvi parameter)
   -- se skličemo na rezultat poizvedbe v vseh nadaljnih podprogramih
   -- paketa DBMS_SQL
   c_rowid := DBMS_SQL.OPEN_CURSOR;

   -- Procedura PARSE preveri pravilnost poizvedbe, ki jo poda drugi
   -- parameter. V primeru, da je poizvedba sestavljena napačno,
   -- prevajalnik javi napako.
   -- Tretji parameter avtomatsko določi verzijo sistema Oracle, ki jo
   -- uporabljamo, saj se vsaka verzija sistema Oracle(6i, 7i itd.)
   -- obnaša nekoliko drugače. Od verzije Oracle 9i naprej se uporablja
   -- parameter DBMS_SQL.NATIVE. V primeru starejših verzij sistema Oracle
   -- prav tako lahko uporabljamo omenjeni parameter, zaradi katerega se
   -- nam ni potrebno ozirati na verzijo sistema.
   -- Tako ni potrebno npr. za verzijo Oracle 6i določiti parameter
   -- DBMS_SQL.V6, za verzijo Oracle 7i parameter DBMS_SQL.V7 itd.
   DBMS_SQL.PARSE(c_rowid, poizvedba, DBMS_SQL.NATIVE);

   -- Procedura BIND_VARIABLE poveže vrednost spremenljivke datum z mestom
   -- v poizvedbi, ki je označeno z :dat. Rezultat (tabela) poizvedbe v
```

```
    -- pomnilniku, kamor kaže c_rowid, se tako spreminja glede na podano
    -- vrednost spremenljivke datum.
    DBMS_SQL.BIND_VARIABLE(c_rowid, 'dat', datum);

    -- S proceduro DEFINE_COLUMN določimo stolpcu na mestu 1 (v tabeli,
    -- katero vrne poizvedba) tip spremenljivke, v katero bomo v
    -- nadaljevanju shranili vrednost stolpca 1.
    -- Tip stolpca in tip spremenljivke morata biti torej enaka.
    -- Ni pa nujno, da vrednost stolpca shranimo v spremenljivko vred_rowid.
    -- Lahko jo shranimo tudi v drugo spremenljivko pod pogojem, da je ta
    -- tipa VARCHAR2. Kako nam to lahko pomaga, bomo opisali v razdelku
    -- 4.4.2. Če stolpcu določimo napačen tip spremenljivke, se lahko zgodi,
    -- da imamo kasneje na mestu kjer bi moral biti npr. datum niz.
    -- Zadnji parameter dodamo, če obravnavamo stolpec tipa VARCHAR2
    -- ali ROWID. S tem določimo maksimalno dolžino niza, ki ga lahko
    -- sprejme spremenljivka.
    DBMS_SQL.DEFINE_COLUMN(c_rowid, 1, vred_rowid, 30);

    -- Funkcija EXECUTE izvrši poizvedbo in vrne število
    -- vrstic tabele, katero vrne poizvedba.
    izvrsitev := DBMS_SQL.EXECUTE(c_rowid);

    -- Procedura FETCH_ROWS "ujame" posamezno vrstico tabele, ki jo
    -- vrne poizvedba. Iz te vrstice nato v nadaljevanju programa beremo
    -- vrednosti. V našem primeru zanko while izvajamo, dokler ne pregledamo
    -- vseh vrstic tabele.
    WHILE DBMS_SQL.FETCH_ROWS(c_rowid) <> 0 LOOP

       -- Procedura COLUMN_VALUE shrani prebrano vrednost iz 1. stolpca
       -- (ujete vrstice) v spremenljivko vred_rowid.
       DBMS_SQL.COLUMN_VALUE(c_rowid, 1, vred_rowid);

       -- Pokličemo funkcijo vrni_delete, ki vrne ustrezno sestavljen
       -- stavek DELETE.
       stavek_delete := vrni_delete(ime_tabele, vred_rowid);

       -- Vrnjen stavek DELETE zapišemo v datoteko.
       UTL_FILE.PUT_LINE(dat_zapis, stavek_delete);

       -- Pokličemo funkcijo vrni_delete, ki vrne ustrezno sestavljen
       -- stavek INSERT.
       stavek_insert := vrni_insert(ime_tabele, vred_rowid);

       -- Vrnjen stavek INSERT zapišemo v datoteko.
       UTL_FILE.PUT_LINE(dat_zapis, stavek_insert);

       -- Med posamezne stavke DELETE in INSERT vstavimo prazno vrstico.
       UTL_FILE.NEW_LINE(dat_zapis);
    END LOOP;

    -- Procedura CLOSE_CURSOR zapre dostop do dela v pomnilniku, kjer je
    -- shranjena tabela, ki jo vrne poizvedba.
    DBMS_SQL.CLOSE_CURSOR(c_rowid);

END delete_insert;
```

DIPLOMSKA NALOGA :
FAKULTETA ZA MATEMATIKO IN FIZIKO

4.3 Ideja programa

Program za nadgradnjo moramo popraviti tako, da bo deloval za vse tabele iz skupine B. Predstavimo idejo za izboljšavo programa v naslednjih treh točkah.

1. Obravnavanje vseh tabel skupine B.

Za »glavni« podprogram bomo določili proceduro, ki bo prejela datum prejšnje verzije sistema in za posamezno ime tabele izvedla določene podprograme (točka 2). Imena tabel bomo shranili v spremenljivko, kjer bodo posamezna imena zapisana v nizu in ločena z vejicami. To spremenljivko smo predstavili v razdelku 2.2.2.1.

Potrebovali bomo tudi spremenljivko tipa CURSOR, ki bo hranila tabelo, ki je rezultat spodnje poizvedbe. Ta poišče tista imena tabel iz slovarja podatkov, ki so našteta v nizu. Tako bo potrebno za vzdrževanje programa skrbeti le za pravilen zapis niza z imeni tabel.

```
imena_tabel VARCHAR2(200)
      := 'KP1199_SKU_SIFRANT,KP1200_SKU_VR_SIFR,KP1093_SPR_FORMUL,
      KP1094_FORMULE,KP1230_JS_VP_SK,KP1229_JS_VP,KP1228_JS_DM,
      KS0027_LABELE,KS0037_BESEDILA';

CURSOR cur_tabele
      IS
         SELECT table_name
           FROM user_tables
           -- Izberemo tista imena tabel v bazi, ki so
           -- v spremenljivki imena_tabel.
           WHERE INSTR (imena_tabel, table_name) > 0 ;
```

V poizvedbi uporabimo funkcijo INSTR, ki smo jo predstavili v 2.2.3. Ta funkcija za posamezno ime tabele preveri, če je shranjeno v spremenljivki `imena_tabel`.

Po izvedbi ukaza je v spremenljivki `cur_tabele` tabela, ki je prikazana na Sliki 16. Po tej tabeli se bomo sprehodili z zanko FOR, kjer bomo klicali ustrezne »pomožne« podprograme. Ti bodo sestavljali in zapisovali stavke DELETE in INSERT v datoteko.

TABLE_NAME
KP1094_FORMULE
KP1199_SKU_SIFRANT
KP1200_SKU_VR_SIFR
KP1093_SPR_FORMUL
KP1230_JS_VP_SK
KP1229_JS_VP
KP1228_JS_DM
KS0027_LABELE
KS0037_BESEDILA

Slika 16 : Tabela z imeni tabel skupine B.

2. Sestavljanje »začetnih« delov stavka DELETE in INSERT.

V naslednjem koraku moramo razmisliti, kako sestaviti »začetni« del stavka DELETE in INSERT. Poglejmo si primer začetnih delov stavkov:

```
DELETE FROM kp1094_formule WHERE id1094_formula = PRIMARNI_KLJUČ;

INSERT INTO kp1094_formule
            (id1094_formula, va1094_sestav, de1094_naziv,
            va1094 opis1, va1094 opis2, tr0000 user kreir,
            tr0000_dat_kreir,tr0000_user_azur,tr0000_dat_azur,
            tr0000_uporaba) VALUES ( . . .VREDNOSTI. . .);
```

Tu smo s PRIMARNI_KLJUČ označili vrednost stolpca `id1094_formula`, ki predstavlja primarni ključ tabele formul. Glede na podano vrednost stavek DELETE izbriše določeno vrstico iz strankine tabele formul. Z …VREDNOSTI… pa smo označili vrednosti vseh naštetih stolpcev tabele formul v stavku INSERT. Ob izvedbi stavka INSERT v strankino tabelo formul vstavimo vrstico s posodobljenimi vrednostmi.

Poleg ukazov in imen tabel bomo za oblikovanje stavkov potrebovali še imena stolpcev, tipe stolpcev in oznake omejitev. Te podatke prav tako kot imena tabel preberemo iz slovarja podatkov. Oznake omejitev bomo potrebovali zato, da bomo vedeli, katero ime stolpca predstavlja primarni ključ. Tega bomo potrebovali za pravilno obliko stavka DELETE. Tipe stolpcev pa bomo potrebovali za pravilno oblikovane vrednosti. Slednje bomo v naslednjem koraku dodali začetnim delom stavkov DELETE in INSERT.

Imena stolpcev, tipe stolpcev in oznake omejitev bomo shranili v spremenljivko sestavljenega tipa, ki smo jo predstavili že v razdelku 2.2.2.1. Poizvedbo, ki vrne te podatke, pa smo predstavili v razdelku 4.1. To spremenljivko bomo napolnili večkrat, za vsako tabelo s svojimi podatki. Zato bomo to spremenljivko sproti praznili z metodo DELETE (razdelek 2.2.2.1). Potrebovali bomo tudi proceduro, ki bo za posamezno ime tabele najprej izpraznila in nato napolnila spremenljivko tipa TABLE.

Stavka DELETE in INSERT sta oblikovana različno, zato ju bomo oblikovali vsakega posebej v svojem podprogramu. Ta dva podprograma bo uporabila procedura `delete_insert`, ki smo jo predstavili v razdelku 4.2. Procedura za parameter prejme ime tabele in datum ter izvede dinamično sestavljeno poizvedbo, ki vrne enolične naslove vrstic. To stori le za tiste vrstice, ki so se po danem datumu spremenile. Še enkrat si poglejmo to poizvedbo in tabelo, ki jo vrne. Denimo, da je podano ime tabele : `kp1094_formule` in datum : `TO_DATE('01.01.2009', 'DD.MM.YYYY')`.

```
stavek_select VARCHAR2(100)
          := 'SELECT rowid id_vrstice FROM '||ime_tabele
             ||' WHERE tr0000_dat_azur >= :dat';
```

Zgornja poizvedba vrne tabelo:

ID_VRSTICE
AAADVGAABAAAIFSAAA
AAADVGAABAAAIFSAAB
AAADVGAABAAAIFSAAC
AAADVGAABAAAIFSAAD
AAADVGAABAAAIFSAAG

Slika 17: Delna tabela z enoličnimi naslovi vrstic tabele formul.

V proceduri bomo v zanki WHILE za posamezen enoličen naslov vrstice izvedli prej omenjena podprograma, ki bosta sestavila stavka DELETE in INSERT.

Te enolične naslove bomo uporabili le za branje vrednosti in ne kot del stavka DELETE. Kot smo razložili že v razdelku 4.2, so enolični naslovi vrstic v razvojni bazi različni od tistih v strankini bazi.

3. Dopolnjevanje stavkov DELETE in INSERT z opremljenimi vrednostmi.

Podprograma, ki bosta oblikovala stavka DELETE in INSERT, bosta za parameter poleg ime tabele prejela tudi vrednost tipa ROWID, torej enoličen naslov posamezne vrstice. Podprograma bosta za pravilno oblikovanje stavkov poleg »začetnih« delov stavkov potrebovala tudi podatke prebrane iz vrstic določene tabele. Zato bomo v obeh podprogramih potrebovali dinamično sestavljeno poizvedbo. Ta bo za dano ime tabele in vrednost tipa ROWID vrnila eno vrstico določene tabele.

V dinamično sestavljeni poizvedbi ne moremo uporabiti *, saj pri branju podatkov iz tabele, ki jo poizvedba vrne, ne poznamo vrstnega reda stolpcev, niti imena stolpcev. Vrednosti stolpcev torej ne moremo brati, kot jih lahko preberemo iz tabele, katero vrne statično sestavljena poizvedba (glejte razdelek 2.2.2.1). Takrat vnaprej poznamo imena stolpcev. Rezultate dinamično sestavljene poizvedbe beremo v takšnem vrstnem redu, kot ga določi poizvedba sama. Tako se lahko v našem primeru zgodi, da se stolpci v »začetnem« delu stavka INSERT INTO (imena_stolpcev) ne bodo ujemali s pripadajočimi vrednostmi stolpcev v delu VALUES (vrednosti). Kot smo omenili že v razdelku 2.2.2.1, ob uporabi * sistemi za delo z bazami podatkov (RDBMS) ne jamčijo, da bo poizvedba vedno (ob različnih izvedbah programa) vrnila tabelo z enako razvrščenimi stolpci. Zato moramo v dinamično sestavljeni poizvedbi našteti vsa imena stolpcev. Ta imena bomo v poizvedbi našteli tako, da jih bomo prebrali iz spremenljivke sestavljenega tipa (točka 2.) in shranili v spremenljivko `imena_stolpcev`.

Poglejmo si primer poizvedbe in vrstico tabele, ki jo ta poizvedba vrne.

```
st_select := REPLACE('SELECT '||imena_stolpcev||' FROM '||ime_tabele
                     ||' WHERE rowid = '''||vrednost_rowid||'''','''','''''') ;
```

Denimo, da je podano ime tabele : `kp1094_formule` in vrednost tipa ROWID: `'AAADnWAAbAAAIfSAAb'`. Potem zgornja poizvedba vrne tabelo:

ID1094_FORMULA	VA1094_SESTAV	DE1094_NAZIV	• • •
0003	VP_KOL * VP_V1	Kolicina * vrednost	• • •

Slika 18: Del stolpcev tabele formul, ki jo vrne poizvedba.

Za sestavljanje poizvedb bomo sestavili funkcijo. Ta bo s pomočjo podatkov zapisanih v spremenljivki tipa TABLE in enoličnega naslova sestavila in vrnila niz s poizvedbo.

Da bodo podatki v stavku DELETE in INSERT pravilno oblikovani, moramo za posamezen stolpec preveriti njegov tip. Tudi tega preberemo iz spremenljivke tipa TABLE. Glede na tip stolpca nato lahko pokličemo funkcije `dodaj`, ki smo jih predstavili že v prejšnji verziji programa.

4.4 Koda programa

Celoten program bomo zapisali v paket `pripravi`. S tem bomo zamenjali prejšnjo verzijo programa. Ker je ta paket že shranjen v bazi, se ime paketa v kodi nove različice programa obarva svetlo zeleno. Nova različica paketa bo sestavljena iz 11 podprogramov (7 funkcij in 4 procedur). Začnimo z glavo paketa.

4.4.1 Glava paketa

V glavo paketa zapišemo dve spremenljivki in štiri glave podprogramov. Ti so globalni, da jih lahko kličemo tudi iz drugih programov.

```
CREATE OR REPLACE PACKAGE pripravi
AS
   -- Potrebujemo globalno spremenljivko tipa TABLE, ki smo jo predstavili
   -- v razdelku 2.2.2.1. To spremenljivko bomo za posamezno ime tabele
   -- napolnili z imeni stolpcev, tipi stolpcev in oznakami omejitev.
   -- To spremenljivko bo uporabljalo več podprogramov.

   TYPE opis_stolpca IS RECORD
   (
      ime VARCHAR2(20),
      tip VARCHAR2(10),
      omejitev VARCHAR2 (1)
   );

   TYPE opis_tabele IS TABLE OF opis_stolpca
   INDEX BY BINARY_INTEGER;

   tab opis_tabele;

   -- Sklic na datoteko, kjer bomo zapisali stavke DELETE in INSERT.
   dat_zapis UTL_FILE.FILE_TYPE;

   -- Glave podprogramov.

   -- "Glavna" procedura, ki z izvedbo spodnjih podprogramov
   -- zapisuje stavke DELETE in INSERT v datoteko.
   PROCEDURE zapis_v_dat (datum DATE);

   -- Procedura, ki napolni tabelo tab z imeni stolpcev,
   -- tipi stolpcev in oznakami omejitev za določeno tabelo.
   PROCEDURE napolni_tab (ime_tabele VARCHAR2);

   -- Procedura, ki v datoteko zapiše stavke DELETE in INSERT.
   PROCEDURE delete_insert (datum DATE, ime_tabele VARCHAR2);

   -- Funkcija, ki vrne niz sestavljenega stavka DELETE.
   FUNCTION vrni_delete (ime_tabele VARCHAR2, id_vrstice ROWID)
      RETURN VARCHAR2;

   -- Funkcija, ki vrne niz sestavljenega stavka INSERT.
   FUNCTION vrni_insert (ime_tabele VARCHAR2, id_vrstice ROWID)
      RETURN VARCHAR2;

   -- Funkcija, ki sestavi in vrne poizvedbo. Ta vrne eno vrstico
   -- določene tabele.
```

```
   FUNCTION vrni_poizvedbo (ime_tabele VARCHAR2, id_vrstice ROWID)
      RETURN VARCHAR2;

   -- Procedura, ki glede na tipe stolpcev v tabeli (to vrne
   -- poizvedba)določi tipe spremenljivk.
   PROCEDURE definiranje_stolpcev (cur NUMBER);

   -- Funkcija, ki vrne ustrezno opremljene vrednosti trenutno
   -- obravnavanega stolpca (indeks).
   FUNCTION vrednost_stolpca (cur NUMBER, indeks NUMBER)
      RETURN VARCHAR2;

   -- Funkcije, ki ustrezno opremijo in vrnejo vrednosti.
   FUNCTION dodaj (vrednost VARCHAR2)
      RETURN VARCHAR2;

   FUNCTION dodaj (vrednost DATE)
      RETURN VARCHAR2;

   FUNCTION dodaj (vrednost NUMBER)
      RETURN VARCHAR2;
END pripravi;
```

4.4.2 Telo paketa

V telo paketa zapišemo kodo funkcij in procedur. Začnimo z »glavno« proceduro.

1. Predstavimo proceduro `zapis_v_dat`, ki za parameter prejme datum objave prejšnje verzije sistema. Procedura potrebuje spremenljivko tipa CURSOR, ki dobi tabelo z imeni tabel, kot smo razložili v razdelku 4.3. Procedura nato za posamezno tabelo odpre datoteko v načinu za pisanje in v zanki WHILE pokliče proceduri `napolni_tab` in `delete_insert`. S proceduro `napolni_tab` napolnimo globalno spremenljivko `tab` z imeni stolpcev, tipi stolpcev in omejitvami za posamezno tabelo. Omenjeno spremenljivko za svoje delovanje potrebuje več podprogramov. Proceduro `delete_insert`, ki sestavi in zapiše stavke INSERT in DELETE v datoteko, smo predstavili že v razdelku 4.2. Proceduro `napolni_tab` pa bomo predstavili v nadaljevanju.

```
CREATE OR REPLACE PACKAGE BODY pripravi
AS
   PROCEDURE zapis_v_dat (datum DATE)
   AS
      -- Spremenljivka z imeni tabel iz skupine B.
      imena_tabel VARCHAR2(200)
           := 'KP1199_SKU_SIFRANT,KP1200_SKU_VR_SIFR,KP1093_SPR_FORMUL,
              KP1094_FORMULE,KP1230_JS_VP_SK,KP1229_JS_VP,KP1228_JS_DM,
              KS0027_LABELE,KS0037_BESEDILA';

      -- Ustvarimo tabelo z imeni tistih tabel iz baze, ki so tudi v
      -- nizu imena_tabel.
      CURSOR cur_tabele
      IS
         SELECT table_name
           FROM user_tables
           WHERE INSTR(imena_tabel, table_name) > 0;
   BEGIN
      dat_zapis := UTL_FILE.FOPEN('c:\prenos', 'zap_dat.txt', 'w');

      -- Za posamezno ime tabele v datoteko zapišemo stavke DELETE in
      -- INSERT.
```

```
    FOR tek_vrstica IN cur_tabele
    LOOP
       -- Napolnimo spremenljivko tab z imeni stolpcev, tipi stolpcev
       -- in omejitvami.
       napolni_tab(tek_vrstica.table_name);

       -- V datoteko zap_dat.txt zapišemo stavke DELETE in INSERT.
       delete_insert(datum, tek_vrstica.table_name);
    END LOOP;

 UTL_FILE.FCLOSE(dat_zapis);
END zapis_v_dat;
```

2. Predstavimo sedaj proceduro `napolni_tab`, ki za parameter prejme ime tabele. Procedura nato napolni globalno spremenljivko `tab` z imeni stolpcev, tipi stolpcev in oznakami omejitev. Spremenljivko `tab` moramo najprej izprazniti z metodo DELETE. Izprazniti jo moramo zato, ker jo bomo ob vsakem klicu te procedure napolnili z drugimi podatki. Te podatke preberemo iz tabele, ki jo vrne poizvedba. Slednjo smo predstavili v razdelku 4.1.

```
PROCEDURE napolni_tab(ime_tabele VARCHAR2)
AS
    -- Potrebujemo spremenljivko tipa CURSOR, ki hrani tabelo z
    -- imeni stolpcev, tipi stolpcev in oznakami omejitev ter
    -- spremenljivko, s katero določimo posamezno vrstico tabele tab.
    CURSOR cur_metapod
    IS
       SELECT column_name, data_type, omejitve.oznaka oznaka
         FROM user_tab_columns tab,
                (SELECT stolpec.column_name ime,
                        omejitev.constraint_type oznaka
                   FROM user_constraints omejitev,
                        user_cons_columns stolpec
                  WHERE stolpec.table_name = ime_tabele
                        AND omejitev.constraint_name =
                            stolpec.constraint_name
                 ) omejitve
          WHERE table_name = ime_tabele
                AND tab.column_name = omejitve.ime(+);

    indeks NUMBER := 1;
BEGIN
    -- Izpraznimo spremenljivko tab preden jo napolnimo z imeni stolpcev,
    -- tipi stolpcev in omejitvami za naslednjo tabelo.
    tab.DELETE;

    -- Prenesemo vse vrednosti iz spremenljivke cur_metapod
    -- v spremenljivko tab.
    FOR tek_vrstica IN cur_metapod
    LOOP
       tab(indeks).ime := tek_vrstica.column_name;
       tab(indeks).tip := tek_vrstica.data_type;
       tab(indeks).omejitev := tek_vrstica.oznaka;

       indeks:= indeks + 1;
    END LOOP;
END napolni_tab;
```

3. V razdelku 4.2 smo že opisali proceduro `delete_insert`, ki s pomočjo funkcij `vrni_delete` in `vrni_insert` v datoteko zapiše ustrezne stavke DELETE in INSERT. Spomnimo se, da s temi stavki popravimo strankino bazo tako, da ima posodobljene in nove podatke v bazi. Omenjeni funkciji bomo predstavili v nadaljevanju.

4. Oglejmo si funkcijo `vrni_delete`, ki prejme parameter z imenom tabele in parameter tipa ROWID. Funkcija sestavi stavek DELETE, s katerim v strankini bazi najprej izbrišemo vrstico, katero želimo posodobiti.
Funkcija s pomočjo spremenljivke `tab` sestavi »začetni« del stavka DELETE (glejte razdelek 4.3). Poleg tega potrebujemo še funkcijo `vrni_poizvedbo`, ki sestavi in vrne niz s poizvedbo. Ta vrne posamezno vrstico določene tabele iz katere preberemo podatke in jih dodamo »začetnemu« delu stavka DELETE. Funkcija `vrni_delete` potrebuje še proceduro `definiranje_stolpcev`, ki glede na tip stolpcev v tabeli (katero vrne poizvedba) določi tipe spremenljivk. V spremenljivko določenega tipa nato funkcija `vrednost_stolpca` shrani vrednost za posamezni stolpec in ga ustrezno opremljenega vrne. Funkcijo `vrni_poizvedbo` in `vrednost_stolpca` ter proceduro `definiranje_stolpcev` bomo podrobneje predstavili v nadaljevanju.

```
FUNCTION vrni_delete (ime_tabele VARCHAR2, id_vrstice ROWID)
   RETURN VARCHAR2
AS
   -- Spremenljivka, v katero shranimo poizvedbo, ki vrne eno vrstico
   -- določene tabele.
   poizvedba VARCHAR2(600);

   -- Spremenljivka, v katero shranimo ustrezno opremljeno vrednost.
   vred VARCHAR2(500);

   -- Spremenljivka, v katero shranimo imena prim. ključev in
   -- pripadajoče vrednosti. Za začetno vrednost uporabimo
   -- prazen niz (zakaj potrebujemo prazen niz, je razloženo v
   -- nadaljevanju).
   prim_ključ_z_vred VARCHAR2(1000) := ' ';

   -- Spremenljivka v katero shranimo zaporedno številko mesta
   -- v pomnilniku, kamor bomo shranili rezultat poizvedbe.
   c_delete NUMBER;

   -- Spremenljivka v katero shranimo število vrstic tabele,
   -- katero vrne poizvedba.
   izvrsitev NUMBER;

BEGIN
   -- Pokličemo funkcijo, ki glede na podano vrednost rowid
   -- vrne poizvedbo. Ta vrne eno vrstico določene tabele.
   poizvedba := vrni_poizvedbo(ime_tabele, id_vrstice);

   -- Rezerviramo prostor v pomnilniku, kamor bomo shranili tabelo,
   -- katero vrne poizvedba.
   c_delete := DBMS_SQL.OPEN_CURSOR;

   -- Preverimo pravilnost poizvedbe.
   DBMS_SQL.PARSE(c_delete, poizvedba, DBMS_SQL.NATIVE);

   -- Pokličemo proceduro, ki določi tipe spremenljivk posameznega
   -- stolpca tabele. Slednja je shranjena v pomnilniku kamor kaže
   -- c_delete.
   definiranje_stolpcev(c_delete);
```

```
    -- Izvršimo poizvedbo in ujamemo prvo vrstico omenjene tabele.
    -- To funkcijo paketa DBMS_SQL v razdelku 4.2 nismo
    -- omenili. To funkcijo lahko uporabimo zato, ker vemo, da poizvedba
    -- vrača le eno vrstico določene tabele (zaradi enoličnega naslova).
    izvrsitev := DBMS_SQL.EXECUTE_AND_FETCH(c_delete);

    -- Z zanko FOR iz spremenljivke tab postopoma preberemo imena
    -- stolpcev,  tipe stolpcev in omejitve. Te vrednosti potrebujemo
    -- za pravilno oblikovan stavek DELETE.
    FOR indeks IN 1 .. tab.COUNT LOOP

      -- Pokličemo funkcijo, ki vrne ustrezno opremljeno vrednost.
      -- To vrednost nato shranimo v spremenljivko vred. Vrednost te
      -- spremenljivke v nadaljevanju programa spojimo s preostalim nizom
      -- stavka DELETE.
      vred := vrednost_stolpca(c_delete, indeks);

      -- Vzamemo le tiste vrednosti stolpcev v ujeti vrstici,
      -- ki so prim. ključi.
      -- Če ključ ni sestavljen (ali je prvi) ne dodamo ukaza AND.
      -- To preverimo s praznim nizom.
      IF (tab(indeks).omejitev = 'P')
         AND (prim_ključ_z_vred = ' ')
      THEN
         prim_ključ_z_vred:= tab(indeks).ime ||' = '||vred;
      ELSIF (tab(indeks).omejitev = 'P')
      THEN
         prim_ključ_z_vred:= prim_ključ_z_vred || ' AND '
                             ||tab(indeks).ime
                             ||' = '||vred;
      END IF;
    END LOOP;

    DBMS_SQL.CLOSE_CURSOR(c_delete);

    -- Vrnemo sestavljen stavek DELETE.
    RETURN 'DELETE FROM '||ime_tabele||' WHERE '
                         ||prim_ključ_z_vred||' ;';
END vrni_delete;
```

5. Sestavimo funkcijo `vrni_insert`, ki je podobna funkciji `vrni_delete`. Razlikujeta se le v tem, da funkcija `vrni_insert` sestavi in vrne niz s stavkom INSERT. Ta stavek zapiše v določeno tabelo strankine baze vrstico s posodobljenimi podatki. To vrstico smo najprej s stavkom DELETE izbrisali.

```
FUNCTION vrni_insert (ime_tabele VARCHAR2, id_vrstice ROWID)
   RETURN VARCHAR2
AS
   -- Spremenljivka, v katero shranimo poizvedbo, ki vrne eno vrstico.
   poizvedba VARCHAR2(600);

   -- Spremenljivka, v katero shranimo opremljeno vrednost.
   vred VARCHAR2(500);

   -- Spremenljivka v katero shranimo vrednosti "ujete" vrstice določene
   -- tabele.
   vrednosti VARCHAR2(1000);

   -- Spremenljivka v katero shranimo imena stolpcev določene tabele.
   imena_stolp VARCHAR2(1000);
```

```
    -- Spremenljivka v katero shranimo zaporedno številko mesta
    -- v pomnilniku, kamor bomo shranili rezultat poizvedbe.
    c_insert NUMBER;

    -- Spremenljivka v katero shranimo število vrstic tabele,
    -- ki jo vrne poizvedba.
    izvrsitev NUMBER;

BEGIN
    -- Pokličemo funkcijo, ki glede na podano vrednost rowid
    -- vrne poizvedbo. Ta vrne eno vrstico tabele (npr. tabele formul).
    poizvedba := vrni_poizvedbo(ime_tabele, id_vrstice);

    -- Rezerviramo prostor v pomnilniku kamor bomo shranili tabelo,
    -- ki jo vrne poizvedba.
    c_insert := DBMS_SQL.OPEN_CURSOR;

    -- Preverimo še pravilnost poizvedbe.
    DBMS_SQL.PARSE(c_insert, poizvedba, DBMS_SQL.NATIVE);

    -- Pokličemo proceduro, ki določi tipe spremenljivk posameznega
    -- stolpca tabele. Slednja je shranjena v pomnilniku kamor kaže
    -- c_insert.
    definiranje_stolpcev(c_insert);

    -- Izvršimo poizvedbo in ujamemo prvo vrstico omenjene tabele.
    -- To funkcijo lahko uporabimo zato, ker vemo, da
    -- poizvedba vrača le eno vrstico določene tabele
    -- (zaradi enoličnega naslova).
    izvrsitev := DBMS_SQL.EXECUTE_AND_FETCH(c_insert);

    -- Z zanko FOR iz spremenljivke tab postopoma preberemo imena
    -- stolpcev, tipe stolpcev in omejitve. Te vrednosti potrebujemo
    -- za pravilno oblikovan stavek INSERT.
    FOR indeks IN 1 .. tab.COUNT LOOP

      -- Pokličemo funkcijo, ki vrne ustrezno opremljeno vrednost.
      -- To shranimo v spremenljivko vred.
      -- V nadaljevanju vrednost te spremenljivke spojimo s preostalim
      -- nizom stavka INSERT.
      vred := vrednost_stolpca(c_insert, indeks);

      -- Če je trenutno obravnavan stolpec prvi ali edini, ne
      -- dodamo vejice.
      IF (indeks = 1)
      THEN
         imena_stolp := tab(indeks).ime;
         vrednosti := vred;
      ELSE
         imena_stolp := imena_stolp ||','|| tab(indeks).ime;
         vrednosti := vrednosti ||','|| vred;
      END IF;
    END LOOP;

    DBMS_SQL.CLOSE_CURSOR(c_insert);

    -- Vrnemo sestavljen stavek INSERT.
    RETURN 'INSERT INTO '||ime_tabele||' ( '||imena_stolp
           ||' ) VALUES ( '||vrednosti||' );';
END vrni_insert;
```

6. Sestavimo funkcijo `vrni_poizvedbo`, ki prejme parameter z imenom tabele in parameter tipa ROWID. Funkcija iz vrednosti spremenljivke `tab` sestavi in vrne niz, ki predstavlja poizvedbo. Ta glede na enoličen naslov obravnava eno vrstico določene tabele.

```
FUNCTION vrni_poizvedbo(ime_tabele VARCHAR2, id_vrstice ROWID)
   RETURN VARCHAR2
AS
   -- Spremenljivka v katero shranimo imena stolpcev
   -- za določeno tabelo.
   imena_stolp VARCHAR2(600);
BEGIN
   -- Z zanko FOR se sprehodimo preko vseh vrstic
   -- spremenljivke tab in sestavimo niz s poizvedbo. Ta bo ob
   -- izvedbi (v funkciji vrni_delete in vrni_insert) vrnila
   -- eno vrstico določene tabele.
   FOR indeks IN 1 .. tab.COUNT
   LOOP
      IF indeks = 1
      THEN
         imena_stolp:= tab(indeks).ime;
      ELSE
         imena_stolp:= imena_stolp ||','
                       ||tab (indeks).ime;
      END IF;
   END LOOP;

   -- Vrnemo sestavljeno poizvedbo.
   RETURN  REPLACE('SELECT '||imena_stolp||' FROM '
                   ||ime_tabele||' WHERE  rowid = "'
                   ||id_vrstice|| '"','"',''');
END vrni_poizvedbo;
```

7. Sestavimo proceduro `definiranje_stolpcev`, ki glede na tip stolpcev v tabeli (katero vrne poizvedba) določi tipe spremenljivk. Kot smo razložili že v razdelku 4.3 ni potrebno v spremenljivke, ki jih določimo s proceduro DEFINE_COLUMN shraniti vrednosti stolpcev (določene tabele) v iste spremenljivke, katere uporabimo v proceduri COLUMN_VALUE. Vrednosti stolpcev s COLUMN_VALUE v ustrezno spremenljivko shranimo v funkciji `vrednost_stolpca`, ki jo bomo podrobneje predstavili v nadaljevanju. V obeh omenjenih podprogramih uporabimo tri lokalne spremenljivke tipa VARCHAR2, NUMBER in DATE. Spremenljivke omenjenih tipov uporabimo zato, ker so lahko stolpci v tabelah takšnega tipa (glejte razdelek 3).

```
PROCEDURE definiranje_stolpcev (cur NUMBER)
AS
   -- Spremenljivke s katerimi za posamezni stolpec tabele, ki je
   -- shranjena v pomnilniku kamor kaže parameter cur, določimo tipe
   -- spremenljivk. V te spremenljivke, bomo nato v funkciji
   -- vrednost stolpca shranili vrednost določenega stolpca.
   niz VARCHAR2(400);
   st  NUMBER;
   dat DATE;

BEGIN
   -- Za vse stolpce določene tabele določimo tipe spremenljivk.
   -- V spremenljivke določenega tipa, bo nato funkcija vrednost_stolpca
   -- shranila posamezno vrednost stolpca.
   FOR indeks IN 1 .. tab.COUNT
   LOOP
      -- Če je obravnavani stolpec tipa NUMBER, določimo temu
      -- stolpcu tip spremenljivke st.
```

```
        -- Isto preverimo za stolpce tipa VARCHAR2 in DATE. Spremenljivki
        -- tipa VARCHAR2 določimo še maksimalno možno dolžino niza.
        IF tab(indeks).tip = 'NUMBER'
        THEN
           DBMS_SQL.DEFINE_COLUMN(cur, indeks, st);
        ELSIF tab(indeks).tip = 'VARCHAR2'
        THEN
           DBMS_SQL.DEFINE_COLUMN(cur, indeks, niz,400);
        ELSIF tab(indeks).tip = 'DATE'
        THEN
           DBMS_SQL.DEFINE_COLUMN(cur, indeks, dat);
        END IF;
     END LOOP;
  END definiranje_stolpcev;
```

8. Sestavimo funkcijo `vrednost_stolpca`, ki glede na tip spremenljivke (določene v proceduri `definiranje_stolpcev`) v spremenljivko ustreznega tipa shrani vrednost posameznega stolpca določene tabele. Nato s pomočjo funkcij `dodaj` oblikuje in vrne ustrezno oblikovano vrednost. Funkcije `dodaj` smo uporabili že v prejšnji verziji programa.

```
FUNCTION vrednost_stolpca (cur NUMBER, indeks NUMBER)
      RETURN VARCHAR2
   AS
      -- Spremenljivke v katere bomo shranili vrednosti prebrane iz
      -- "ujete" vrstice določene tabele. To vrstico ujamemo v
      -- funkciji vrni_delete in vrni_insert.
      niz VARCHAR2(400);
      st  NUMBER;
      dat DATE;

   BEGIN
      -- Če je obravnavani stolpec "ujete" vrstice tipa NUMBER potem s
      -- COLUMN_VALUE shranimo vrednost stolpca v spremenljivko st.
      -- Da je vrednost pravilno oblikovana pokličemo še funkcije dodaj,
      -- ki ustrezno opremijo vrednost.
      -- Isti postopek velja za stolpec tipa VARCHAR2 in DATE.
      IF tab(indeks).tip = 'NUMBER'
      THEN
         DBMS_SQL.COLUMN_VALUE(cur, indeks, st);
         RETURN dodaj(st);
      ELSIF tab(indeks).tip = 'VARCHAR2'
      THEN
         DBMS_SQL.COLUMN_VALUE(cur, indeks, niz);
         RETURN dodaj(niz);
      ELSIF tab(indeks).tip = 'DATE'
      THEN
         DBMS_SQL.COLUMN_VALUE(cur, indeks, dat);
         RETURN dodaj(dat);
      END IF;
   END vrednost_stolpca;

   -- Funkcije dodaj.

END pripravi;
```

5 ZAKLJUČEK

V diplomski nalogi smo predstavili dva načina izvedbe programa za nadgradnjo strankine podatkovne baze. Da smo lahko opisali izvedbo programa, smo predstavili še sistem Oracle ter jezika SQL in PL/SQL, ki ju ta sistem uporablja.
Trenutna različica programa zahteva le, da moramo skrbeti za pravilen zapis niza, v katerem so zapisana imena tabel, ki jih popravljajo razvijalci in se morajo ažurirati v strankini bazi (tabele skupine B). Program pa moramo še vedno izvesti sami in mu določiti datum objave prejšnje verzije sistema. Zato bi lahko ta program »avtomatizirali«. To pomeni, da bi se program izvedel sam, če bi se v razvojni bazi spremenila, ali bi bila dodana nova vrstica v tabelo skupine B. Za avtomatično izvedbo programov skrbi prožilec. Prožilce smo predstavili v razdelku 2.2.6.3
Zato bi za vsako tabelo iz skupine B napisali prožilec, ki bi izvedel naš program. Imena tabel tako ne bi več hranili v nizu, ampak bi prožilec programu posredoval ime tabele (za katero pišemo prožilec). Prihodnjo različico našega programa bi torej lahko popravili tako, da bi »glavni« podprogram za parameter prejel ime tabele in datum (določimo ga s funkcijo SYSDATE). Spomnimo, da se v primeru, ko izvedemo stavek UPDATE ali INSERT izvede prožilec, ki posodobi (doda) datum spremembe (`tr0000_dat_azur`), ime uporabnika spremembe (`tr0000_user_azur`) itd. (glejte razdelek 2.2.6.3). Ko bi se torej prožilec nad določeno tabelo izvedel, bi program preveril datum, ki bi ga podali za parameter ter datum iz stolpca `tr0000_dat_azur`. Program bi zapisal stavke za nadgradnjo, če bi veljalo : datum_spremembe >= SYSDATE. Ker bi bila oba datuma podana s funkcijo SYSDATE, bi program našel le vrstico, ki bi jo nazadnje posodobili (ali dodali). Tudi v primeru, da bi v enem dnevu dodali ali spremenili več vrstic, bi program še vedno našel pravo vrstico, saj s funkcijo SYSDATE lahko določimo tudi sekunde. Primer takšnega prožilca (za tabelo formul) smo predstavili v razdelku 2.2.6.3.

Z uporabo prožilcev bi se tako pred vsako novo verzijo sistema za obračun plač izognili ročnemu izvajanju našega programa, saj bi se le-ta izvedel sam.

LITERATURA

1. Urman, S. *Oracle8 PL/SQL Programing.* United States of America: McGraw-Hill Companies, 1997. The instant notes series. ISBN 0-07-882305-6.

2. Owens, K. T. *Builnig Intelligent Databases with Oracle PL/SQL, Triggers and stored procedures.* 2st. ed. New Jersey: A Simon & Schuster Company,1998. The instant notes series. ISBN 0-13-794314-8.

3. Nagavalli, P. and Priya, N. *Oracle9i: Program with PL/SQL.* United States of America: Oracle Corporation, 1999-2001.

Spletni viri

1. Lokar M. *Združevanje tabel*. Ljubljana: Fakulteta za matematiko in fiziko, Pridobljeno 18.6.2009 iz http://ucilnica0809.fmf.uni-lj.si/login/index.php.

2. Oracle/PLSQL Topics: Built-In Functions (By Category), Pridobljeno 27.09.2009 iz http://www.techonthenet.com/oracle/functions/index.php.

3. O'Reilly & Associates, Oracle PL/SQL Language Pocket Reference, Pridobljeno 27.09.2009 iz http://hell.org.ua/Docs/oreilly/oracle/langpkt/ch01_12.htm.

4. Oracle Corporation, Using PL/SQL Subprograms, Pridobljeno 27.09.2009 iz http://download-west.oracle.com/docs/cd/B28359_01/appdev.111/b28370/subprograms.htm.

5. BC praetoriate Oracle Sopport, Pridobljeno 29.09.2009 iz http://www.praetoriate.com/t_high_perform_define_column.htm.