

Programiranje baza podataka

Beleške za predavanja

Vesna Marinković

email: vesnap@matf.bg.ac.rs

URL: www.matf.bg.ac.rs/~vesnap

Matematički fakultet, Beograd

©2019

Autor:

dr Vesna Marinković, docent Matematičkog fakulteta u Beogradu

PROGRAMIRANJE BAZA PODATAKA

Sva prava zadržana. Nijedan deo ovog materijala ne može biti reprodukovan niti smešten u sistem za pretraživanje ili transmitovanje u bilo kom obliku, elektronski, mehanički, fotokopiranjem, smanjenjem ili na drugi način, bez prethodne pismene dozvole autora.

Sadržaj

Sadržaj	3
1 Uvod	5
1.1 Logistika kursa	5
2 Načini korišćenja baza podataka u višim programskim jezicima	7
2.1 Aplikativni (ugrađeni) SQL	9
2.2 Dinamički SQL	21
2.3 Direktni pozivi funkcija SUBP (DB2 CLI)	31
2.4 Upotreba ODBC standarda	39
2.5 Zapamćene procedure	48
3 Transakcije	59
3.1 Transakcija i integritet	59
3.2 Konkurentnost	60
3.3 Svojstva transakcija i nivoi izolovanosti transakcija	74
3.4 Oporavak	84
4 Objektno-relaciono preslikavanje i alat Hibernate	93
4.1 Objektno-relaciono preslikavanje	94
4.2 Hibernate	104
5 Administracija baza podataka	151
5.1 Kreiranje baze podataka	151
5.2 Premeštanje podataka	158
5.3 Pravljenje rezervnih kopija i oporavak	162
5.4 Aktivnosti na održavanju baze podataka	165

Glava 1

Uvod

1.1 Logistika kursa

Cilj kursa:

- usvojiti napredne koncepte i tehnike baza podataka,
- savladati aplikativni SQL (SQL/C),
- savladati ODBC standard rada sa bazom podataka (JDBC),
- ovladati tehnikama programiranja transakcija (upravljanje transakcijama, pad i oporavak; kontrola konkurentnosti),
- shvatiti osnove objektno-relacionog preslikavanja (ORM Hibernate),
- dobiti pregled operacija potrebnih za administraciju i održavanje baze podataka

Literatura:

- G. Pavlović - Lažetić: Osnove relacionih baza podataka, Matematički fakultet, Beograd, 1999.
- H. Garcia-Molina, J. D. Ullman, J. Widom: Database Systems: The Complete Book, International Version, 2nd edition, Pearson Education, 2008.
- C.J.Date: An Introduction to Database Systems, VIII ed, Addison Wesley Inc, 2004.
- S. Malkov: Programiranje baza podataka, skripta
- DB2 uputstva i materijali

Glava 2

Načini korišćenja baza podataka u višim programskim jezicima

Većina SQL proizvoda podržava dva vida izvršavanja SQL naredbi:

- **interaktivni**, koji podrazumeva izvršavanje samostalnih SQL naredbi preko on-line terminala,
- **aplikativni**, koji podrazumeva izvršavanje SQL naredbi umetnutih u program na višem programskom jeziku opšte namene, pri čemu se SQL naredbe mogu izvršavati naizmenično sa naredbama programskog jezika.

Program na aplikativnom SQL-u prolazi kroz pretprocesiranje kao prvu fazu svoje obrade. Aplikativni SQL može da uključi dve vrste SQL naredbi:

- **statičke SQL naredbe** koje već u fazi pisanja programa imaju precizan oblik (strukturu) i čija se analiza i priprema za izvršenje mogu u potpunosti obaviti u fazi pretprocesiranja; na primer, moraju se u vreme preprocesiranja u potpunosti znati svi nazivi kolona i tabela na koje se referiše;
- **dinamičke SQL naredbe** koje se zadaju u obliku niske karaktera koja se dodeljuje promenljivoj u programu; analiza i priprema ovakve vrste naredbi može se obaviti tek u fazi izvršavanja programa kada je poznat njen precizni oblik; na primer, kada aplikacija u vreme izvršavanja očekuje od korisnika da unese deo SQL naredbe, na primer nazive tabela i kolona nad kojima se upit izvršava

Pored ovakvog načina korišćenja baze podataka u programu na programskom jeziku opšte namene, postoje i druge mogućnosti ugradnje blokova za komunikaciju sa sistemom za upravljanje bazama podataka (SUBP) u program. U slučaju sistema DB2, to su:

- **upotreba ODBC standarda** (eng. Open DataBase Connectivity) i
- **direktni pozivi DB2 funkcija – CLI** (eng. Call Level Interface).

ODBC standard podrazumeva sledeći model komunikacije: korisnik komunicira sa ODBC međusistemom, koji dalje komunicira sa konkretnim SUBP-om, potpuno nezavisno od korisnika. U ovom modelu podrazumeva se standardizovanje komunikacije korisnika sa ODBC međusistemom, nezavisno od SUBP kome će međusistem proslediti zahtev korisnika. Ovaj pristup omogućava korišćenje različitih SUBP-a, čak i u okviru jednog programa. Dakle, na ovaj način mi u stvari imamo zajednički interfejs za programiranje baza podataka, i jedino je potrebno uključiti drajver za određeni tip baze podataka. Na ovaj način je korisniku olakšan posao jer piše program na isti način bez obzira na to nad kojim SUBP-om radi, ali nije moguće iskoristiti specifičnosti pojedinog sistema (u ovom slučaju sistema DB2). Svi upiti upućeni sistemu DB2 putem ODBC standarda izvršavaju se dinamički.

S druge strane, direktni pozivi DB2 funkcija su specifični za sistem DB2 i koriste sve njegove mogućnosti (kao i aplikativni SQL), s tim da je ipak ovaj pristup izgrađen tako da ima najveći mogući zajednički skup funkcija i principa sa ODBC pristupom, čime je korisnicima omogućeno da ukoliko je potrebno jednostavnije prilagođavaju programe jednom ili drugom obliku. Prednosti ovakvog načina komunikacije sa bazom u odnosu na ugrađeni SQL jeste portabilnost aplikacija tako što je eliminisana zavisnost od preprocesora, dakle, aplikacije se distribuiraju ne kao ugnježdeni SQL izvorni kod koji se mora preprocesirati za svaki proizvod baze podataka, već kao prevedena aplikacija ili izvršna biblioteka. Takođe, moguće je u jednoj istoj aplikaciji povezati se na veći broj baza podataka, uključujući veći broj konekcija na istu bazu podataka.

Ugrađeni SQL je po mogućnostima sličan direktnim pozivima DB2 funkcija, jedina bitna razlika je u načinu kodiranja SQL naredbi. Dok se u CLI pristupu podrazumeva upotreba funkcija kojima se kao argumenti zadaju SQL naredbe, u ugrađenom SQL-u se koriste makroi za kodiranje SQL naredbi. Ugrađeni SQL se prevodi, od strane pretprocesora, u program koji ne sadrži makroe, već direktne pozive DB2 funkcija.

Za razliku od programa na aplikativnom SQL-u koji mora da se pretprocesira, prevede, poveže sa bazom i najzad izvrši, DB2 CLI aplikacija ne mora ni da se pretprocesira ni da se povezuje sa bazom; ona koristi standardni skup funkcija za izvršavanje SQL naredbi u vreme izvršavanja programa. Osnovna prednost aplikativnog SQL-a u odnosu na direktne pozive DB2 funkcija je u tome što može da koristi statički SQL i što je u velikoj meri standardizovan pa se može koristiti (u manjoj ili većoj meri) i među različitim SUBP-platformama.

U slučaju da aplikacija zahteva prednosti oba mehanizma (direktnih poziva DB2 funkcija i aplikativnog SQL-a), moguće je koristiti statički SQL unutar DB2 CLI aplikacije kreiranjem **zapamćenih procedura** (eng. stored procedures) napisanih korišćenjem statičkog SQL-a. Zapamćena procedura se poziva iz DB2 CLI aplikacije i izvršava se na serveru. Jednom napisanu zapamćenu proceduru može da pozove bilo koja DB2 CLI ili ODBC aplikacija. Naime, aplikacija može biti isprojektovana tako da se izvršava u dva dela: jedan deo na klijentu, a drugi na serveru, pri čemu je zapamćena procedura deo koji se izvršava na serveru.

2.1 Aplikativni (ugrađeni) SQL

Umesto ugradnje SQL naredbi u programski jezik opšte namene, može se razmatrati mogućnost korišćenja samo jednog jezika: da se kompletan posao odradi u SQL-u ili da zaboravimo SQL i sve programiramo u nekom višem programskom jeziku. Međutim, potreba za umetanjem SQL-a u viši programski jezik javlja se zbog odsustva kontrolnih struktura u upitnom jeziku, često neophodnih pri obradi podataka iz baze podataka. S druge strane, SQL značajno pomaže programeru u izvršavanju operacija nad bazom podataka: na ovaj način operacije nad bazom se efikasno izvršavaju i, dodatno, izražene su na višem nivou. Umetanjem (tzv. ugnježdenjem) SQL-a (eng. Embedded SQL) u viši programski jezik, tzv. **matični jezik** (eng. host language), kao što su C, C++, Java, FORTRAN, COBOL, dobija se **aplikativni SQL**, na kome se mogu pisati kompleksni programi za najraznovrsnije obrade. Osnovni princip svih aplikativnih SQL jezika je **princip dualnosti**, prema kome se svaka interaktivna SQL naredba može izraziti i u aplikativnom SQL-u, ali obratno ne važi (što i opravdava ovakva umetanja).

Osnovni problem pri povezivanju SQL naredbi sa naredbama standardnog programskog jezika jeste činjenica da se model podataka u SQL-u razlikuje od modela podataka u drugim jezicima. Kao što je poznato, SQL koristi relacioni model podataka, dok recimo C i slični jezici koriste model podataka koji uključuje cele brojeve, razlomljene brojeve, karaktere, pokazivače, strukture podataka, nizove, itd. Skupovi nisu direktno podržani u jeziku C, dok SQL ne koristi pokazivače, petlje, grananja i mnoge druge uobičajene programske konstrukcije. Dakle, razmena podataka između SQL-a i drugih programskih jezika nije pravolinijska i potrebno je definisati mehanizam kojim bi se omogućio razvoj aplikacija koje koriste i SQL i neki drugi programski jezik.

Svi dalji koncepti aplikativnog SQL-a biće ilustrovani na primeru tabele Knjiga koja je definisana na sledeći način:

```
create table Knjiga
(k_sifra integer not null,
 naziv char(50) not null,
 izdovac char(30),
 god_izdavanja smallint,
 primary key (k_sifra))
```

Povezivanje SQL-a sa matičnim jezikom

Kada želimo da koristimo SQL naredbu u programu na matičnom jeziku, upozoravamo pretprocesor da nailazi SQL kôd navođenjem ključnih reči EXEC SQL ispred naredbe. Razmena informacija između baze podataka, kojoj se pristupa samo putem SQL naredbi, i programa na matičnom jeziku vrši se kroz **host (matične) promenljive** (eng. host variables), za koje je dozvoljeno da se pojave i u naredbama matičnog jezika i u SQL naredbama. Host promenljive imaju kao prefiks dvotačku u naredbama SQL-a, dok se u naredbama matičnog jezika javljaju bez dvotačke.

Program koji sadrži izvršne SQL naredbe komunicira sa sistemom DB2 preko memorijskog prostora koji se zove **SQL prostor za komunikaciju** (eng. SQL Communication Area, SQLCA). SQLCA je struktura podataka koja se

ažurira posle izvršavanja svake SQL naredbe. U ovu strukturu upisuju se informacije o izvršenoj SQL naredbi.

SQLCA prostor se u program na matičnom jeziku uključuje naredbom:

```
EXEC SQL INCLUDE SQLCA;
```

Ovde je u pitanju SQL naredba INCLUDE. Ona služi za uključivanje datoteka i definicija koje su potrebne za aplikacije u ugnježdenom SQL-u. Razlikuje se od standardne pretprocesorske direktive #include po tome što je obrađuje SQL preprocesor, dok #include se izvršava tek kasnije, pri prevođenju programa na matičnom jeziku (i SQL preprocesor ignoriše ovakve naredbe). Ova naredba se mora navesti pre prve izvršne SQL naredbe. Jedan od efekata ove naredbe jeste deklaracija SQLCA strukture, koja se koristi za obradu greške.

Najčešće korišćena promenljiva strukture SQLCA je promenljiva SQLCODE. Ona predstavlja indikator uspešnosti izvršenja SQL naredbe i može da ima sledeće vrednosti:

- 0, ako je izvršavanje prošlo uspešno (ako je pritom vrednost promenljive SQLWARN0="" onda je sve u redu, a ako je SQLWARN0='W' znači da se javilo neko upozorenje),
- pozitivnu vrednost, ako se pri uspešnom izvršavanju dogodilo nešto izuzetno (npr. +100 ako je rezultujuća tabela prazna),
- negativnu vrednost, ako se zbog nastale greške naredba nije izvršila uspešno.

Provera statusa izvršenja SQL naredbe, sačuvane u promenljivoj SQLCODE, zadaje se direktivom pretprocesoru da umetne odgovarajuće IF - THEN blokove u program. Direktiva je deklarativni iskaz oblika:

```
EXEC SQL WHENEVER <uslov> <akcija>;
```

pri čemu <uslov> može biti:

- NOT FOUND (nije nađen - drugi zapis za SQLCODE = 100),
- SQLERROR (indikator greške - drugi zapis za SQLCODE < 0) ili
- SQLWARNING (indikator upozorenja - drugi zapis za SQLWARN0 = 'W' OR (SQLCODE > 0 AND SQLCODE <> 100));

a <akcija> je:

- CONTINUE - program nastavlja sa izvršavanjem ili
- GOTO obeležje - skok na deo programa u kome se nastavlja obrada, npr. izveštavanje o uzroku nastalog uslova.

Dakle, deklarativni iskaz WHENEVER omogućuje programeru da zada način na koji će se proveravati vrednost promenljive SQLCODE nakon svake izvršene SQL naredbe. Ovakvih naredbi može biti veći broj u programu. Svaki WHENEVER iskaz odnosi se na sve SQL naredbe (u redosledu njihovog zapisa, a ne izvršenja) sve do navođenja sledećeg WHENEVER iskaza. Na primer, iskaz oblika:

```
EXEC SQL WHENEVER SQLERROR CONTINUE;
```

obezbediće nastavak izvršavanja programa i u slučaju da je pri izvršavanju SQL naredbe došlo do greške. To znači da program neće imati prinudni završetak, ali, sa druge strane, to obavezuje programera da posle svake SQL naredbe u programu eksplicitno ispituje vrednost promenljive `SQLCODE`; u slučaju greške kontrolu upravljanja treba preneti na deo kojim se greška obrađuje. Ako ne postoji odgovarajući `WHENEVER` iskaz, podrazumevana akcija je `CONTINUE`.

Osim promenljive `SQLCODE`, `SQLCA` struktura uključuje i veći broj drugih promenljivih. Najvažnija među njima je promenljiva `SQLSTATE` koja je tipa `char[5]` i slično promenljivoj `SQLCODE` sadrži kôd rezultata izvršavanja SQL naredbe (npr. prva dva karaktera '01' promenljive `SQLSTATE` predstavljaju grupu kodova za upozorenja i ukazuju, slično `SQLCODE > 0`, da je došlo do pojave koja rezultuje upozorenjem).

Među ostalim zanimljivijim promenljivama `SQLCA` strukture je skup promenljivih `SQLWARN0` – `SQLWARN9` koje su tipa `char` i sadrže informacije o uzroku upozorenja. Na primer, značenje nekih od ovih elemenata `SQLCA` strukture je sledeće:

- `SQLWARN0` ima vrednost 'W' ako bar jedna od preostalih `SQLWARNi` promenljivih sadrži upozorenje ('W'); tako uslov `SQLWARNING` u direktivi `WHENEVER` predstavlja disjunkciju uslova:
(`SQLCODE > 0 AND SQLCODE <> 100`) OR `SQLWARN0 = 'W'`;
- `SQLWARN1` sadrži 'W' ako je pri dodeli promenljivoj matičnog jezika odsečena vrednost kolone koja je tipa niska karaktera (`CHAR(m)`);
- `SQLWARN2` sadrži 'W' ako je došlo do eliminacije `NULL` vrednosti pri primeni agregatne funkcije.

`SQLCODE` i `SQLWARN0`,...`SQLWARN9` su makroi, koji ukazuju na odgovarajuće delove `SQLCA` strukture, tj. na polja `sqlcode` i delove niza karaktera `sqlwarn`. Na primer, naredna funkcija bi se mogla koristiti za proveru da li je došlo do greške:

```
int is_error( char error_string[], struct sqlca *sqlca_pointer){
    if (sqlca_pointer->sqlcode < 0) {
        printf ("ERROR occured : %s.\n SQLCODE : %ld\n",
                error_string, sqlca_pointer->sqlcode);
        return 1;
    }
    return 0;
}
```

Njena upotreba bi se mogla olakšati uvođenjem narednog makroa:

```
#define CHECKERR(message) if(is_error(message,&sqlca) != 0) return 1;
```

Komentari u okviru SQL naredbi

U okviru SQL naredbi mogu se koristiti standardni komentari na matičnom jeziku, kao i SQL komentari, koji počinju sa dve crtice `--` i traju do kraja reda, jedino što njima ne sme biti prekinut par ključnih reči `EXEC SQL`. Dakle, naredna naredba ne bi bila ispravna:

```
EXEC -- ovo nije dobar komentar
SQL create view Knjige_2016
as select * from Knjige
where god_izdavanja = 2016;
```

Povezivanje na bazu podataka

Da bi program radio sa podacima iz neke baze podataka, neophodno je da se prethodno poveže na tu bazu. Ukoliko se u sam program ne ugradi povezivanje na bazu podataka, može doći do nepredvidivih posledica ukoliko se program pokrene u okviru neke druge sheme iste baze podataka. Stoga je preporučljivo da se program poveže na bazu podataka pre početka obrade i da raskine vezu po završetku obrade. Ovo se postiže SQL naredbama `CONNECT` i `CONNECT RESET`.

Faze prevođenja

Program koji u sebi sadrži ugrađeni SQL prevodi se u nekoliko koraka. Pre prevođenja neophodno je prvo izvršiti pretprocesiranje ugrađenog SQL-a od strane DB2 preprocesora. Kao što smo već pomenuli, kao rezultat faze pretprocesiranja dobićemo program koji u sebi ne sadrži SQL makroe već direktne pozive DB2 funkcija i tzv. veznu datoteku koja je potrebna za ugradnju paketa u bazu. Paket čini program koji koristi bazu podataka i podaci o tome kako to čini (proizvodi se plan pristupa za SQL naredbe, npr nivo izolovanosti i oni se čuvaju zajedno u vidu paketa unutar baze podataka). Da bi se program mogao izvršavati potrebno je ugraditi paket u bazu, pri čemu se vrši optimizacija svih pristupa bazi od strane programa. Moguće je ponavljati ugradnju paketa u cilju optimizacije pristupa bazi u skladu sa najnovijim statističkim podacima.

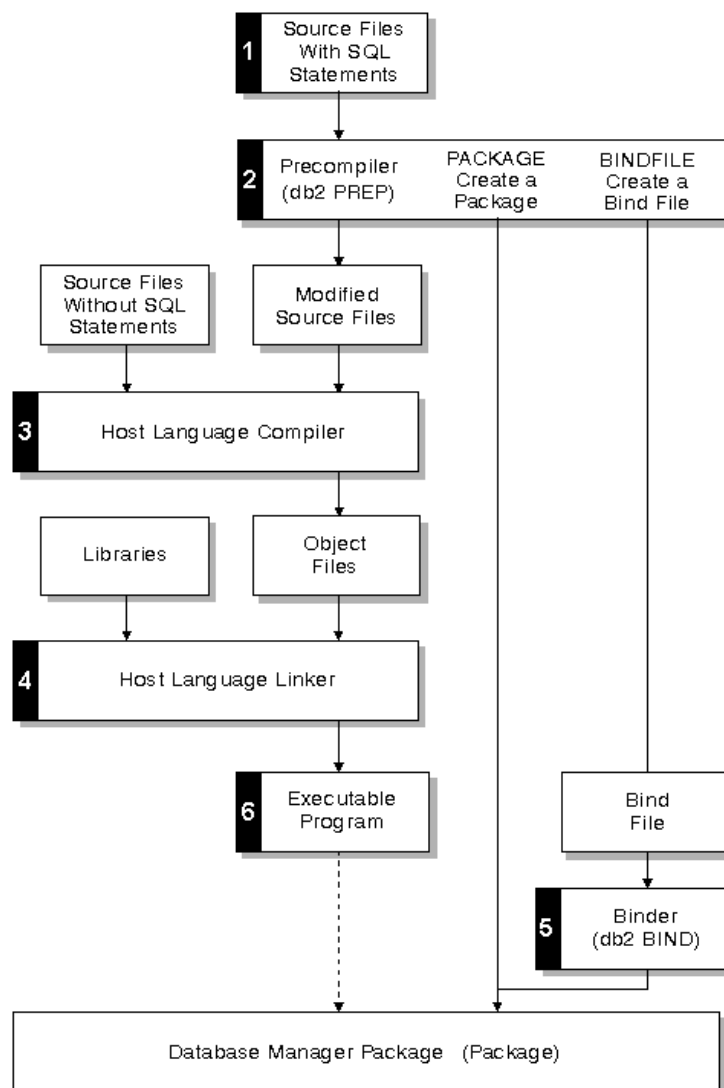
Pretprocesiranje se vrši DB2 naredbom `PREP` (ili `PRECOMPILE`):

```
DB2 PREP datoteka.sqc BINDFILE
```

Kao rezultat dobijaju se dve datoteke sa istim nazivom ali sa različitim ekstenzijama `.c` i `.bnd`. Opcijom `BINDFILE` zadaje se da će ugradnja paketa biti odložena za kasnije (u principu moguće je ugradnju paketa uraditi i tokom pretprocesiranja od strane SQL preprocesora). Nakon faze pretprocesiranja moguće je vršiti ugradnju paketa u bazu naredbom `BIND`:

```
DB2 BIND datoteka.bnd
```

Nakon ovoga potrebno je još prevesti dobijenu C datoteku, a zatim i izvršiti linkovanje sa odgovarajućom bibliotekom db2 funkcija.



Slika 2.1: Prevođenje programa sa ugrađenim SQL-om

Sekcija za deklaraciju host promenljivih

Host promenljive se deklariraju tako što se njihove deklaracije postavljaju između dve ugrađene SQL naredbe:

```
EXEC SQL BEGIN DECLARE SECTION;
...
EXEC SQL END DECLARE SECTION;
```

Ova sekcija se naziva **sekcijom za deklaraciju host promenljivih**. Format deklaracije promenljive treba da odgovara formatu matičnog jezika. Ima smisla deklarirati promenljive samo onih tipova sa kojima se može raditi i u

matičnom jeziku i u SQL-u, kao što su recimo celi brojevi, niske karaktera i slično. Host promenljiva i atribut mogu imati isti naziv.

U narednoj tabeli dat je pregled osnovnih SQL tipova i odgovarajućih C tipova.

SQL tip	C tip
SMALLINT	short
INTEGER	long
DOUBLE	double
CHAR	char
CHAR(n)	char[n+1]
VARCHAR(n)	char[n+1]
DATE	char[11]
TIME	char[9]

Na primer, ukoliko bismo hteli da u programu koristimo host promenljive koje će sadržati identifikator knjige, njen naziv i godinu izdavanja, njih bismo mogli da deklariramo na sledeći način:

```
EXEC SQL BEGIN DECLARE SECTION;
    long k_sifra;
    char naziv[51];
    short godina_izdavanja;
EXEC SQL END DECLARE SECTION;
```

Korišćenje host promenljivih

Host promenljiva se može koristiti u SQL naredbama na mestima gde se može naći konstanta. Podsetimo se da host promenljivoj prethodi dvotačka kada se koristi u SQL naredbi. Ukoliko se pokazivačke promenljive koriste u SQL naredbama tada se moraju upotrebljavati sa tačno onoliko simbola '*' sa koliko su definisane.

U nastavku je prikazan fragment programa u kome se od korisnika zahteva da unese podatke o knjizi: šifru, njen naziv i godinu izdavanja, zatim se ti podaci učitavaju i u tabelu Knjiga se dodaju informacije o novoj knjizi.

```
EXEC SQL BEGIN DECLARE SECTION;
    long uneta_sifra;
    char naziv[51];
    short god_izdavanja;
EXEC SQL END DECLARE SECTION;

/* ovde bi islo ispisavanje poruke korisniku da unese
   sifru knjige, njen naziv i godinu izdavanja, a zatim
   učitavanje podataka koje je korisnik uneo */

EXEC SQL INSERT INTO Knjiga(k_sifra,naziv,god_izdavanja)
    VALUES (:uneta_sifra,:naziv,:god_izdavanja);
```

Poslednje dve linije kôda sadrže ugrađenu SQL naredbu INSERT. Ovoj naredbi prethodi par ključnih reči EXEC SQL da bi se naznačilo da je ovo ugrađena SQL naredba. Vrednosti koje se unose u ovim linijama nisu eksplicitne konstante, već host promenljive čije tekuće vrednosti postaju komponente n -torke koja se dodaje u tabelu.

Svaka SQL naredba koja ne vraća rezultat (tj. koja nije upit) se može ugraditi u program na matičnom jeziku dodavanjem ključnih reči EXEC SQL ispred naredbe. Primeri naredbi koje se ovako mogu ugraditi su INSERT, UPDATE, DELETE i naredbe kojim se prave, menjaju ili brišu elementi sheme kao što su tabele ili pogledi.

Upiti tipa SELECT-FROM-WHERE se ne mogu direktno ugraditi u matični jezik zbog razlike u modelu podataka. Upiti kao rezultat proizvode skup n -torki, dok nijedan od glavnih matičnih jezika ne podržava direktno skupovni tip podataka. Stoga, ugrađeni SQL mora da koristi jedan od dva mehanizma za povezivanje rezultata upita sa programom u matičnom jeziku, a to su:

1. SELECT naredba koja vraća tačno jedan red: ovakav upit može svoj rezultat da sačuva u host promenljivama, za svaku komponentu n -torke.
2. kursori: upiti koji proizvode više od jedne n -torke u rezultatu se mogu izvršavati ako se deklarise kursor za taj upit; opseg kursora čine sve n -torke rezultujuće relacije i svaka pojedinačna n -torka se može prihvatiti u host promenljivama i obrađivati programom u matičnom jeziku.

Upiti koji vraćaju tačno jedan red

Forma naredbe SELECT koja vraća tačno jedan red je ekvivalentna formi SELECT-FROM-WHERE naredbe, osim što stavku SELECT prati ključna reč INTO i lista host promenljivih. Ove host promenljive se pišu sa dvotačkom ispred naziva, kao što je slučaj i sa svim host promenljivama u okviru SQL naredbe. Ako je rezultat upita tačno jedna n -torka, onda komponente ove n -torke postaju vrednosti datih promenljivih. Ako je rezultat prazan skup ili više od jedne n -torke, onda se ne vrši nikakva dodela host promenljivama i odgovarajući kôd za grešku se upisuje u promenljivu SQLSTATE.

```
EXEC SQL SELECT naziv, god_izdavanja  
INTO :naziv, :godina_izdavanja  
FROM Knjiga  
WHERE k_sifra = :uneta_sifra;
```

S obzirom na to da je atribut `k_sifra` primarni ključ tabele `Knjiga`, rezultat prethodnog upita je najviše jedna n -torka, tj. tabela sa jednom vrstom čije se vrednosti atributa `naziv` i `god_izdavanja` upisuju, redom, u host promenljive `naziv` i `godina_izdavanja`. Ovde je dat primer kada su isto imenovani atribut tabele i odgovarajuća host promenljiva.

U slučaju da je u navedenom primeru vrednost atributa `god_izdavanja` bila NULL, došlo bi do greške pri izvršenju SELECT naredbe, jer host promenljiva `godina_izdavanja` ne može da sačuva NULL vrednost. Iz ovog razloga se uz host promenljive koje mogu imati nedefinisanu (NULL) vrednost uvode i **indikatorske promenljive** koje na neki način mogu da sačuvaju informaciju da li je vrednost odgovarajućeg atributa NULL. Stoga, obraćanje

host promenljivoj biće sada oblika :promenljiva:ind_promenljiva (nazivu indikatorske promenljive može da prethodi ključna reč INDICATOR). Indikatorske promenljive su tipa koji odgovara SQL tipu SMALLINT (u C-u je to short).

U slučaju naredbe SELECT INTO, ako je vrednost atributa u SELECT liniji NULL, vrednost host promenljive neće se promeniti, a odgovarajuća indikatorska promenljiva dobiće negativnu vrednost. Stoga bi prethodni kôd bio oblika:

```
EXEC SQL SELECT naziv, god_izdavanja
INTO :naziv, :godina_izdavanja:ind_godina_izdavanja
FROM Knjiga
WHERE k_sifra = :uneta_sifra;
```

```
if (ind_godina_izdavanja < 0)
    printf("Godina izdavanja date knjige nije poznata");
```

Za promenljivu naziv nije potrebno uvoditi odgovarajuću indikatorsku promenljivu jer je atribut naziv definisan sa opcijom NOT NULL te ne može imati nedefinisano vrednost.

Kursori

Najčešći način za povezivanje upita iz SQL-a sa matičnim jezikom jeste posredstvom **kursora** koji prolazi redom kroz sve n -torke relacije. Ova relacija može biti neka od postojećih tabela ili može biti dobijena kao rezultat upita. Kursori predstavljaju jedan vid "pokazivača" na jedan red rezultata upita i mogu se postepeno pomerati unapred po rezultujućoj tabeli.

Da bismo napravili i koristili kursor, potrebni su nam naredni koraci:

1. deklaracija kursora, koja ima formu:

```
EXEC SQL DECLARE <naziv_kursora> CURSOR FOR <upit>;
```

Opseg kursora čine n -torke relacije dobijene upitom.

2. otvaranje kursora, koje ima formu:

```
EXEC SQL OPEN <naziv_kursora>;
```

Ovim se izvršava upit kojim je kursor definisan i kursor se dovodi u stanje u kojem je spreman da prihvati prvu n -torku relacije nad kojom je deklarisan.

3. jedna ili više naredbi čitanja podataka, kojom se dobija naredna n -torka relacije nad kojom je kursor deklarisan. Ova naredba ima formu:

```
EXEC SQL FETCH <naziv_kursora> INTO <lista_promenljivih>;
```


U listi promenljivih mora da postoji po jedna promenljiva odgovarajućeg tipa za svaki atribut relacije. Ako postoji n -torka relacije koja se može pročitati, navedene promenljive dobijaju vrednosti odgovarajućih komponenti te n -torke. Ako u relaciji više nema n -torki (iscrpili smo celu relaciju), onda se ne vraća nijedna n -torka, vrednost promenljive SQLCODE postaje 100, a vrednost promenljive SQLSTATE se postavlja na '02000' što znači da nije pronađena nijedna n -torka.

4. zatvaranje kursora, koje ima formu:

```
EXEC SQL CLOSE <naziv_kursora>;
```

Kada se kursor zatvori, onda on više nema vrednost neke n -torke relacije nad kojom je definisan. Ipak, on se može opet otvoriti i onda će opet uzimati vrednosti iz skupa n -torki relacije.

FETCH naredba je jedina naredba kojom se kursor može pomerati. Pošto, u opštem slučaju, rezultujuća tabela sadrži veći broj vrsta, pretraživanje vrste iz rezultujuće tabele obično se stavlja u petlju (WHILE, na primer, u C-u). Petlja se ponavlja sve dok ima vrsta u rezultujućoj tabeli. Po izlasku iz petlje kursor se zatvara.

Jedan program može imati veći broj deklaracija (različitih) kursora.

U nastavku je dat fragment koda kojim se korišćenjem kursora ispisuju red po red informacije o šifri, nazivu i izdavaču svih knjiga izdatih u 2016. godini.

...

```
/* funkcija za obradu greske */
int iserror(char err[]){
    if (SQLCODE < 0){
        printf("SQLCODE %ld %s\n\n", SQLCODE, err);
        return 1;
    }
    return 0;
}
```

```
/* za poziv funkcije greske koristice se makro */
#define CHECKERR(s) if (iserror(s)) exit(1);
```

....

```
/* deklaracija kursora */
EXEC SQL DECLARE kursor_o_knjigama CURSOR FOR
select k_sifra, naziv, izdavac
from Knjiga
where god_izdavanja = 2016;
CHECKERR("Declare kursor_o_knjigama");
```

```
/* otvaranje kursora */
EXEC SQL OPEN kursor_o_knjigama;
CHECKERR("Open kursor_o_knjigama");
```

```

/* sve dok ima jos vrsta i FETCH se uspesno izvrsava */
WHILE (1){

    /* cita se sledeca vrsta */
    EXEC SQL FETCH kursor_o_knjigama
        INTO :k_sifra, :naziv, :izdavac:ind_izdavac;
    CHECKERR("Fetch kursor_o_knjigama");

    /* ako smo obradili poslednju n-torku relacije
       izlazimo iz while petlje */
    if (SQLCODE == 100)
        break;

    /* obrada te vrste */
    printf("Sifra: %ld, naziv knjige: %s ", k_sifra, naziv);

    if (ind_izdavac < 0)
        printf("izdavac: nepoznat\n");
    else
        printf("izdavac: %s\n", izdavac);
}

EXEC SQL CLOSE kursor_o_knjigama;
CHECKERR("Close kursor_o_knjigama");
...

```

Deklarisanje kursora samo za čitanje

Kursor može biti deklarisan samo za čitanje, za čitanje i brisanje, ili za čitanje, brisanje i menjanje podataka. U nekim slučajevima nije očigledno da li je kursor deklarisan samo za čitanje ili je njim dopušteno i brisanje, te je poželjno eksplicitno zadati tip kursora.

Postoje određene situacije kada je zgodno deklarirati kursor samo za čitanje. Ako kursor deklariramo sa opcijom `FOR READ ONLY` (`FOR FETCH ONLY`) (ova opcija se navodi na kraju deklaracije kursora), onda SUBP može biti siguran da relacija nad kojom je kursor deklarisan neće biti izmenjena iako joj dati kursor pristupa. Na ovaj način se omogućava racionalniji pristup podacima većem broju korisnika jer SUBP zna da će podaci biti samo čitani od strane kursora i da se ne zahteva ekskluzivni pristup podacima.

Na primer, prethodno definisani kursor `kursor_o_knjigama` nam je služio samo za čitanje, te smo ga mogli deklarirati i na sledeći način:

```

EXEC SQL DECLARE kursor_o_knjigama CURSOR FOR
select k_sifra, naziv, izdavac
from Knjiga
where god_izdavanja = 2016
FOR READ ONLY;

```

Izmene posredstvom kursora

Kada domen kursora čine n -torke neke bazne tabele (relacije koja se čuva u bazi podataka), onda se posredstvom kursora mogu ne samo čitati vrednosti tekuće n -torke, već se ona može i obrisati ili joj se može izmeniti vrednost. Sintaksa naredbi UPDATE i DELETE jednaka je standardnoj sintaksi ovih naredbi, sa izuzetkom WHERE stavke koja ovde ima oblik:

```
WHERE CURRENT OF <naziv_kursora>
```

Moguće je u programu na matičnom jeziku za datu n -torku ispitivati da li zadovoljava sve željene uslove pre nego što odlučimo da je obrisemo ili izmenimo. Pritom, ako želimo da podatke menjamo putem kursora, potrebno je u deklaraciji kursora nakon upita navesti stavku:

```
FOR UPDATE OF <lista_atributa>
```

gde <lista_atributa> označava listu naziva atributa, međusobno razdvojenih zarezima, koje je dozvoljeno menjati korišćenjem kursora. Ukoliko se navede stavka FOR UPDATE bez navođenja naziva kolona, onda je putem kursora moguće menjati sve kolone tabele ili pogleda navedenih u stavci FROM spoljasnje naredbe SELECT. Nije dobra praksa navoditi veći broj kolona nego što je zaista potrebno u stavci FOR UPDATE jer u nekim slučajevima to može učiniti DB2 manje efikasnim prilikom pristupanja podacima. Za brisanje tekućeg reda putem kursora ne zahteva se navođenje stavke FOR UPDATE.

Posredstvom kursora može se jednom naredbom izmeniti, odnosno obrisati samo jedan red i to onaj koji je poslednji pročitani. Neposredno nakon brisanja poslednjeg pročitanoog reda, kursor se može upotrebljavati samo za čitanje narednog reda.

Na primer, mogli bismo definisati kursor kojim je moguće izmeniti nazive knjiga čiji je izdavač Prosveta. Da li će naziv biti izmenjen, zavisi od interakcije sa korisnikom.

```
/* deklaracija kursora */
EXEC SQL DECLARE promeni_naziv CURSOR FOR
SELECT k_sifra, naziv
FROM Knjiga
WHERE izdavac = 'Prosveta'
FOR UPDATE OF naziv;
CHECKERR("Declare promeni_naziv");

/* otvaranje kursora */
EXEC SQL OPEN promeni_naziv;
CHECKERR("Open promeni_naziv");

/* sve dok ima jos vrsta i FETCH se uspesno izvrsava */
WHILE (1){

    /* cita se sledeca vrsta */
    EXEC SQL FETCH promeni_naziv INTO :k_sifra, :naziv;
    CHECKERR("Fetch promeni_naziv");
```

```
/* ako je doslo do neke greske ili smo obradili poslednju
n-torku relacije, izlazimo iz while petlje */
if (SQLCODE != 0)
    break;

/* interakcija sa korisnikom */
printf("Da li zelite da izmenite naziv knjige sa sifrom %ld
      ciji je naziv %s? Uneti d ili n.\n", k_sifra, naziv);
c = getchar();
/* jos jedan getchar() da procitamo i znak za novi red */
getchar();

if (c == 'd' || c == 'D'){

    /* obrada te vrste */
    printf("Uneti novi naziv knjige\n");
    scanf("%s", novi_naziv);

    EXEC SQL UPDATE Knjiga
        SET naziv = :novi_naziv
        WHERE CURRENT OF promeni_naziv;
    CHECKERR("Update Knjiga");

}

EXEC SQL CLOSE promeni_naziv;
CHECKERR("Close promeni_naziv");
```

Kursor se implicitno ograničava samo za čitanje ako SELECT naredba u deklaraciji kursora uključuje neku od sledećih konstrukcija¹:

- dve ili više tabela u FROM liniji ili pogled koji se ne može ažurirati;
- GROUP BY ili HAVING stavku u spoljašnjoj SELECT naredbi;
- kolonsku funkciju u spoljašnjoj SELECT liniji;
- skupovnu (UNION, INTERSECT, EXCEPT) operaciju osim UNION ALL;
- DISTINCT opciju u spoljašnjoj SELECT liniji;
- agregatnu funkciju u spoljašnjoj SELECT liniji;
- ORDER BY stavku.

¹Lista navedenih uslova nije iscrpna.

2.2 Dinamički SQL

Kao što smo već napomenuli, svaka ugnježdjena statička SQL naredba je u fazi pretprocesiranja sasvim precizno određena u smislu da je poznat njen tip (zna se da je u pitanju recimo naredba `SELECT` ili naredba `UPDATE` ili naredba `INSERT`) i poznata su imena tabela i kolona koje u njoj učestvuju. SQL naredba se zamenjuje pozivom (u matičnom jeziku) odgovarajuće CLI procedure, ali se izvršavanjem programa taj poziv (pa dakle i sama SQL naredba) ne menja.

Za razliku od ovog pristupa, u dinamičkom SQL-u naredbe nisu poznate u vreme pisanja programa i preciziraju se tek u fazi izvršavanja programa. Takve SQL naredbe, umesto da se eksplicitno navode u programu, zadaju se u obliku niske karaktera koja se dodeljuje host promenljivoj. SQL naredba se izgrađuje na osnovu podataka iz programa (npr. vrednost odgovarajuće host promenljive može se uneti sa standardnog ulaza). Dinamički programi nude veću slobodu prilikom pisanja jer se delovi naredbe, čak i kompletne naredbe, mogu formirati na osnovu informacija koje dostavlja korisnik. Međutim, ovako pisani programi se moraju svaki put prilikom izvršavanja iznova analizirati i optimizovati od strane sistema DB2.

Najčešći razlozi za korišćenje dinamičkog SQL-a su sledeći:

- deo ili kompletna SQL naredba će biti generisana tek u vreme izvršavanja;
- objekti nad kojima je SQL naredba formulisana ne postoje u fazi preprocesiranja;
- želimo da se za izvršavanje uvek koristi optimalan plan pristupa podacima, zasnovan na tekućim statistikama baze podataka.

Kako vrednost host promenljive može da se menja u toku izvršavanja programa, tako i SQL naredba dodeljena toj host promenljivoj takođe može da se menja u toku izvršavanja programa.

Da bi dinamička SQL naredba mogla da se izvrši, potrebno je pripremiti je za izvršenje ugnježđenom (statičkom) naredbom `PREPARE`. Nakon toga se dinamička naredba može izvršiti ugnježđenom (statičkom) naredbom `EXECUTE`.

Aplikacija koja koristi dinamički SQL se najčešće sastoji iz narednih koraka:

- SQL naredba se formira na osnovu ulaznih podataka;
- SQL naredba se priprema za izvršavanje i zahteva se opisivanje rezultujuće relacije (ako postoji);
- ako je u pitanju naredba `SELECT` obezbeđuje se dovoljno prostora za smeštanje podataka koje dohvatamo;
- naredba se izvršava (ako nije u pitanju naredba `SELECT`) ili se hvata jedan red podataka (ako je u pitanju naredba `SELECT`);
- vrši se obrada dobijenih informacija.

Razmotrimo naredni primer. Neka je potrebno u biblioteci održavati broj knjiga izdatih u 2019. i 2020. godini i neka se te informacije čuvaju u tabelama 'Izdate2019' i 'Izdate2020'. Dakle, nakon uzimanja knjige iz biblioteke, ako

je godina izdavanja knjige jedna od pomenutih, potrebno je u odgovarajućoj tabeli broj kopija date knjige smanjiti za 1.

```
strcpy(naredba,"UPDATE ");

/* od korisnika učitavamo sifru knjige
   i smestamo je u host promenljivu sifra_knjige */

EXEC SQL SELECT god_izdavanja
INTO :god_izdavanja FROM Knjiga
WHERE k_sifra = :sifra_knjige;

if (god_izdavanja == 2019)
    strcpy(tabela,"Izdate2019");
else if (god_izdavanja == 2020)
    strcpy(tabela,"Izdate2020");

if (god_izdavanja == 2019 || god_izdavanja == 2020){
    strcat(naredba,tabela);
    strcat(naredba, " SET br_knjiga = br_knjiga - 1 WHERE id = ");
    strcat(naredba, k_sifra);

    EXEC SQL PREPARE azuriraj_stanje FROM :naredba;
    EXEC SQL EXECUTE azuriraj_stanje;
}
```

Naredbu UPDATE iz ovog primera nije bilo moguće izvršiti statički jer nismo unapred mogli da znamo u kojoj tabeli treba vršiti izmene, već je to zavisilo od informacija koje smo dobili od korisnika sa standardnog ulaza.

Tipovi dinamičkih naredbi

U zavisnosti od tipa naredbe koja se dinamički izvršava zavisi i odgovarajući kôd. Naredba SELECT se mora razmatrati na drugačiji način od ostalih naredbi jer ona vraća podatke i program mora biti spreman da prihvati te podatke. S obzirom na to da kolone koje se izdvajaju mogu biti nepoznate u vreme pisanja programa, prostor za smeštanje podataka se mora alocirati u vreme izvršavanja. Za naredbe koje nisu tipa SELECT ovo nije neophodno.

Pored toga, naredbe koje sadrže parametre zahtevaju složenije tehnike kodiranja od gotovih naredbi jer se parametrima mora dodeliti vrednost pre izvršavanja naredbe.

Postoje, naravno, i situacije u kojima ne znamo unapred koji će se tip naredbe izvršavati (na primer, korisnik sa standardnog ulaza unosi proizvoljnu naredbu koju treba dinamički izvršiti).

Dakle, razlikujemo sledeće tipove naredbi:

- naredba koje nija SELECT
 - potpuna
 - parametrizovana

- SELECT naredba
 - sa fiksnom listom kolona koje se izdvajaju
 - sa promenljivom listom kolona koje se izdvajaju
- nepoznata naredba

Naredba PREPARE

Naredba PREPARE prevodi tekstualnu formu SQL naredbe u izvršni oblik, dodeljuje pripremljenoj naredbi naziv i eventualno upisuje potrebne informacije u SQLDA strukturu (eng. SQL Description Area) o kojoj će biti više reči kasnije. Njena sintaksa je:

```
PREPARE <naziv_naredbe>
[[[OUTPUT] INTO <ime_rezultujeceg_deskriptora>]
[INPUT INTO <ime_ulaznog_deskriptora>]
FROM <host_prom>
```

gde je <naziv_naredbe> naziv koji se koristi za identifikovanje pripremljene naredbe, a <host_prom> niska karaktera koja sadrži tekst naredbe. Ako se u okviru naredbe navede neka od INTO stavki i naredba se uspešno izvrši onda se navedene deskriptorske strukture popunjavaju informacijama kao u slučaju naredbe DESCRIBE o kojoj će biti reči malo kasnije (u rezultujuću deskriptorsku strukturu se upisuju informacije o izlaznim oznakama parametara, a u ulaznu deskriptorsku strukturu informacije o ulaznim oznakama parametara). U okviru naredbe koja se priprema ne sme se pojavljivati niz ključnih reči EXEC SQL, znak za kraj naredbe ';' (osim u slučaju naredbe CREATE TRIGGER), host promenljive, niti komentari.

Ukoliko naziv naredbe referiše na već postojeću pripremljenu naredbu, onda se ta prethodno pripremljena naredba uništava.

Neka je npr. V1 host promenljiva koja je u programu dobila vrednost – nisku karaktera neke izvršne SQL naredbe. Tada naredba:

```
EXEC SQL PREPARE S1 FROM :V1;
```

na osnovu niske karaktera iz host promenljive V1 kreira izvršnu SQL naredbu čije je ime S1. Ovim se u stvari analizira SQL naredba i utvrđuje se najefikasniji način da se izdvoje traženi podaci. Pitanja na koja treba dati odgovor su:

- kako se upit može prezapisati tako da se može lakše optimizovati?
- koji indeks (ili kombinaciju indeksa) je najbolje iskoristiti?
- za upite kojima se spajaju tabele, u kom redosledu se one trebaju spojiti da bi se minimizovao broj pristupa disku i slično?

Metod na osnovu koga sistem DB2 bira kako pristupiti podacima se naziva plan pristupa (eng. access plan).

Pripremljenu naredbu moguće je koristiti u narednim naredbama:

- DESCRIBE (proizvoljna naredba);

- EXECUTE (ne sme biti naredba SELECT);
- DECLARE CURSOR (samo naredba SELECT).

Dakle, nakon provere uspešnosti izvršenja naredbe PREPARE, pripremljena naredba se može veći broj puta izvršavati korišćenjem naredbe EXECUTE.

Naredba PREPARE koja koristi stavku INTO:

```
EXEC SQL PREPARE S1 INTO :sqlda FROM :V1;
```

ekvivalentna je narednom bloku naredbi:

```
EXEC SQL PREPARE S1 FROM :V1;
EXEC SQL DESCRIBE S1 INTO :sqlda;
```

Kao što je već pomenuto, dinamička SQL naredba ne sme da sadrži obraćanja host promenljivama. Umesto toga mogu se koristiti **oznake parametara** koje mogu biti neimenovane i imenovane. Neimenovana oznaka parametra se najčešće označava znakom pitanja '?' i navodi se na mestu na kome bi se navela host promenljiva kada bi naredba bila statička. Postoje dve vrste neimenovanih oznaka parametara: sa opisom tipa i bez opisa tipa. Oznake parametara bez tipa se označavaju samo znakom pitanja, a sa tipom izrazom CAST (? AS <tip>). Pritom, ova notacija ne označava poziv funkcije, već na neki način daje obećanje da će tip vrednosti koja će se koristiti umesto oznake parametra odgovarati navedenom tipu.

Oznake parametara bez opisa tipa mogu se koristiti na mestima gde se u fazi pripreme može na nedvosmislen način na osnovu konteksta odrediti njihov tip. Na primer, ispravna dinamička SQL naredba bi mogla imati oblik:

```
INSERT INTO Knjiga VALUES (?, ?, ?, ?)
```

jer se iz same naredbe mogu odrediti tipovi parametara (oni redom odgovaraju tipovima kolona navedenim u definiciji tabele Knjiga). Ako se tip parametra ne može odrediti iz konteksta same naredbe, neophodno je koristiti oznaku parametra sa opisom tipa. Na primer, naredba:

```
SELECT ? FROM Knjiga
```

ne bi bila validna jer nije poznat tip rezultujuće kolone. Dakle, u ovom slučaju neophodno je eksplicitno zadati tip parametra:

```
SELECT CAST(? AS INTEGER) FROM Knjiga
```

Imenovane oznake parametara se navode dvotačkom iza koje sledi identifikator oznake parametra, npr. :parametar. Iako ovakva sintaksa nalikuje korišćenju host promenljivih, postoje razlike. Naime, host promenljiva sadrži vrednost u memoriji i koristi se direktno u statičkoj SQL naredbi. Imenovana oznaka parametra predstavlja zamenu za vrednost u dinamičkoj SQL naredbi i njena vrednost se navodi prilikom izvršavanja pripremljene naredbe.

U vreme izvršavanja pripremljene SQL naredbe, oznake parametara se zamjenjuju tekućim vrednostima host promenljivih (ili strukture) koje se navode u EXECUTE naredbi.

Naredba EXECUTE

Naredba EXECUTE se može upotrebljavati u dva oblika:

```
EXECUTE <naziv_naredbe> [USING <lista_host_prom>]
```

ili:

```
EXECUTE <naziv_naredbe> [USING DESCRIPTOR <ime_deskriptora>]
```

Ako pripremljena naredba sadrži oznake parametara, onda se prilikom navođenja EXECUTE naredbe mora navesti neka od USING stavki. U slučaju da se navodi lista host promenljivih, njihov broj mora odgovarati broju oznaka parametara u naredbi, a tipovi host promenljivih redom tipovima parametara.

Na primer, ako je vrednost host promenljive V1 niska karaktera "INSERT INTO Knjiga VALUES (?, ?, ?, ?)", posle izvršavanja naredbe:

```
EXEC SQL PREPARE k_insert FROM :V1;
```

može se izvršiti naredba:

```
EXEC SQL EXECUTE k_insert
      USING :k_sifra, :k_naziv, :k_izdavac, :k_god;
```

pri čemu su k_sifra, k_naziv, k_izdavac i k_god host promenljive koje odgovaraju shemi tabele Knjiga.

Ako se upotrebljava druga vrsta USING stavke, njen argument je SQLDA struktura koja mora biti popunjena podacima o parametrima na odgovarajući način. Naredna polja SQLDA strukture moraju biti popunjena na sledeći način:

- polje SQLN sadrži broj alociranih SQLVAR struktura,
- polje SQLDABC sadrži ukupan broj bajtova alociranih za SQLDA strukturu,
- polje SQLD sadrži broj promenljivih koji se upotrebljava pri obradi naredbe,
- polja SQLVAR sadrže opise pojedinačnih promenljivih.

Najpre je potrebno alocirati dovoljno veliku SQLDA strukturu koja može da primi sve potrebne podatke. Zaglavlje SQLDA strukture je fiksne veličine od 16 bajtova, dok svaka SQLVAR struktura zauzima 44 bajta na 32-bitnim, odnosno 56 bajtova na 64-bitnim sistmima. Dakle, potrebno je alocirati prostor veličine $16 + (\text{SQLD} * \text{sizeof}(\text{struct sqlvar}))$ bajtova. Najčešće se za izračunavanje broja bajtova koji je potreban za smeštanje svih podataka koristi se makro $\text{SQLDASIZE}(n)$, gde je n broj potrebnih SQLVAR struktura, odnosno vrednost polja SQLD.

Pre nego što počne izvršavanje naredbe svaka oznaka parametra zamenjuje se odgovarajućom vrednošću parametra.

Naredbu INSERT iz prethodnog primera možemo izvršiti korišćenjem SQLDA strukture na sledeći način:

```

...
strcpy (k_niska,"INSERT INTO Knjiga VALUES(?,?,?,?)");
EXEC SQL PREPARE k_insert FROM :k_niska;
...
/* priprema SQLDA strukture mojDA */
...
EXEC SQL EXECUTE k_insert USING DESCRIPTOR :mojDA;
...

```

Treba obratiti pažnju da je identifikator deskriptorske SQLDA strukture takođe host promenljiva, te se u SQL naredbama koristi sa dvotačkom.

Dinamički se mogu pripremiti i izvršiti samo izvršne SQL naredbe, i to ne baš sve. Neke od naredbi koje se mogu izvršiti (date u abecednom redosledu) su: ALTER, CREATE, DELETE, DROP, INSERT, SELECT, UPDATE. Među izvršnim SQL naredbama koje ne mogu da se izvršavaju dinamički nalaze se naredbe: CLOSE, DECLARE, EXECUTE, EXECUTE IMMEDIATE, FETCH, OPEN, PREPARE.

Naredba EXECUTE IMMEDIATE

Moguće je koristiti i ugnježdenu (statičku) naredbu EXECUTE IMMEDIATE da bi se u jednom koraku pripremila i izvršila dinamička SQL naredba. Sintaksa ove naredbe je:

```
EXECUTE IMMEDIATE <host_prom>
```

gde je <host_prom> host promenljiva koja sadrži nisku karaktera sa tekstom SQL naredbe. U okviru naredbe se ne sme pojavljivati niz ključnih reči EXEC SQL, znak za kraj naredbe ';', host promenljive, oznake parametara, niti komentari.

Na ovaj način se može izvršiti svaka naredba koja nije SELECT, ako nije parametrizovana, tj. ako ne sadrži oznake parametara. Ako SQL naredba nije validna, naredba se neće izvršiti i putem SQLCA strukture može se analizirati uslov koji je doveo do greške. Ako je SQL naredba validna, ali njeno izvršavanje dovodi do greške, i o toj grešci se možemo informisati putem SQLCA strukture.

Kod ove naredbe se cena pripreme naredbe plaća svaki put kada se ona izvršava, te se ova naredba obično primenjuje kada je dinamičku SQL naredbu potrebno izvršiti samo jednom.

Ako je, na primer, vrednost host promenljive V1 niska karaktera:

"DELETE FROM Knjiga WHERE k_sifra = 101", onda se naredbom:

```
EXEC SQL EXECUTE IMMEDIATE :V1;
```

priprema i izvršava naredba DELETE nad tabelom Knjiga, čime se briše knjiga sa šifrom 101, ako takva postoji.

Naredba DESCRIBE

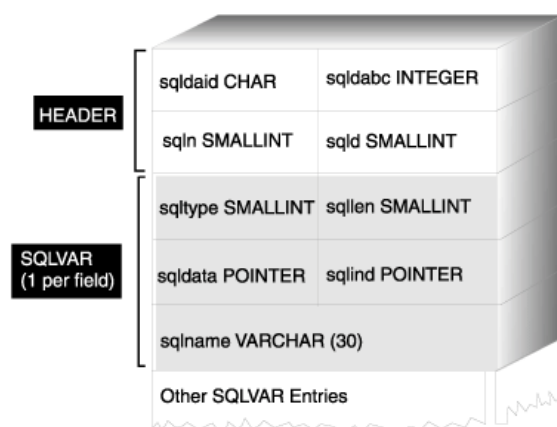
Naredba DESCRIBE upisuje potrebne informacije o naredbi u SQLDA strukturu, pod pretpostavkom da je već izvršena njena priprema. Da bi se SQLDA struktura mogla upotrebljavati neophodno je uključiti odgovarajuće zaglavlje naredbom:

```
EXEC SQL INCLUDE SQLDA;
```

Naredba DESCRIBE se najčešće upotrebljava u formi:

```
DESCRIBE [OUTPUT] <naziv_naredbe> INTO <ime_deskriptora>
```

pri čemu mora postojati dinamička SQL naredba pripremljena pod datim nazivom i mora biti alocirana odgovarajuća SQLDA deskriptorska struktura. Bitno je da je unapred popunjeno polje SQLN strukture SQLDA, koje sadrži maksimalan broj promenljivih čije je opise SQLDA struktura u stanju da sačuva, a može biti i nula. Opciona ključna reč OUTPUT ukazuje na to da će naredbom DESCRIBE biti vraćene informacije o kolonama u pripremljenoj SELECT naredbi.



Slika 2.2: Struktura SQLDA

Nakon izvršavanja naredbe DESCRIBE, SQLDA struktura se popunjava na sledeći način:

- polje SQLDABC sadrži veličinu SQLDA strukture u bajtovima;
- polje SQLD sadrži broj kolona u rezultujućoj relaciji, ako je u pitanju naredba SELECT, ili broj oznaka parametara;
- polje SQLVAR sadrži podatke o pojedinačnim parametrima: ako je SQLD nula, ili je veće od SQLN, ne dodeljuje se ništa SQLVAR elementima.

Svaka SQLVAR struktura sadrži sledeće komponente:

- polje SQLTYPE koje čuva oznaku tipa parametra i oznaku da li može ili ne sadržati NULL vrednost;
- polje SQLLEN koje sadrži veličinu parametra u bajtovima;
- polje SQLNAME čija je vrednost naziv kolone ili tekstom zapisan broj koji opisuje originalnu poziciju izvedene kolone.

Oznake tipova se uglavnom dobijaju tako što se na nisku SQL_TYP_ nadoveže naziv odgovarajućeg SQL tipa, na primer SQL_TYP_SMALLINT (sa izuzetkom za tipove TIMESTAMP, LONG VARCHAR, REAL). Ukoliko parametar može imati NULL vrednost, dodaje se slovo N pre naziva tipa (u prethodnom slučaju SQL_TYP_NSMALLINT).

Korišćenje SQLDA strukture pruža veću fleksibilnost od korišćenja liste host promenljivih. Na primer, korišćenjem SQLDA strukture moguće je preneti podatke koji nemaju nativni ekvivalent u matičnom jeziku (npr tip DECIMAL u programskom jeziku C).

Korisnik je dužan da nakon izvršavanja naredbe DESCRIBE odgovarajućim podacima popuni preostala polja SQLVAR strukture:

- polje SQLDATA mora sadržati pokazivač na prostor predviđen za parametar (bilo za čitanje ili pisanje parametra);
- polje SQLIND mora sadržati pokazivač na prostor predviđen za indikator, ako parametar može imati NULL vrednost.

Alociranje SQLDA strukture može se vršiti na dva načina: alokacijom sa dovoljno mesta (postavljanjem polja SQLN na dovoljno veliku vrednost) ili u dva koraka: u prvom koraku bi se vršila alokacija SQLDA strukture bez ijednog SQLVAR elementa i izvršavala naredba DESCRIBE u cilju dobijanja pravih informacija (polje SQLD sadržaće broj kolona u rezultatu), a u drugom koraku bi se ponovljenom alokacijom (postavljanjem polja SQLN na vrednost veću ili jednaku od vrednosti polja SQLD) obezbeđivalo dovoljno prostora.

Na primer, SQLDA strukturu možemo formirati i popuniti jednom promenljivom u pokretnom zarezu koja je praćena indikatorom na sledeći način:

```
...
struct sqlda *mojDA = (struct sqlda*)malloc(SQLDASIZE(1));
double prom;
short prom_ind;
mojDA->sqln = mojDA->sqld = 1;
mojDA->sqlvar[0].sqltype = SQL_TYP_NFLOAT;
mojDA->sqlvar[0].sqllen = sizeof(double);
mojDA->sqlvar[0].sqldata = &prom;
mojDA->sqlvar[0].sqlind = &prom_ind;
...
```

Pored ove forme naredbe DESCRIBE, postoji i analogna naredba DESCRIBE INPUT kojom se dohvataju informacije o oznakama ulaznih parametara u pripremljenoj naredbi.

Dinamički kursori

Dinamički pripremljena SELECT naredba se može izvršiti korišćenjem kursora. Samo rukovanje dinamičkim kursorom je gotovo identično rukovanju statičkim kursorom, jedina razlika je u tome što je umesto statičkog upita potrebno navesti naziv pripremljenog dinamičkog upita.

Dakle, prilikom rada sa dinamičkim kursorima potrebno je da se izvrše naredni koraci:

- priprema naredbe,
- deklarisanje kursora pod datim nazivom,
- otvaranje kursora,
- dohvaćanje vrsta iz rezultujuće relacije,
- zatvaranje kursora.

Deklaracija dinamičkog kursora izvodi se naredbom:

```
DECLARE <ime_kursora> CURSOR FOR <naziv_naredbe>
```

Kako se u dinamičkim upitima mogu upotrebljavati oznake parametara, prilikom otvaranja kursora potrebno je navesti odgovarajuće vrednosti parametara. To se postiže na isti način kao i kod EXECUTE naredbe, u jednoj od dve forme:

```
OPEN <ime_kursora> [USING <lista_host_prom>]
```

```
OPEN <ime_kursora> [USING DECSRIPTOR <ime_deskriptora>]
```

Primetimo da se pri otvaranju kursora navedeni parametri koriste samo za čitanje vrednosti radi formiranja upita.

Pri čitanju redova pomoću kursora može se koristiti uobičajena forma:

```
FETCH <ime_kursora> INTO <lista_host_prom>
```

ili se može upotrebljavati i deskriptorska struktura:

```
FETCH <ime_kursora> USING DECSRIPTOR <ime_deskriptora>
```

Primetimo da se prilikom čitanja redova pomoću kursora navedeni parametri koriste samo za upisivanje vrednosti koje predstavljaju rezultat upita.

Pogledajmo na primeru dinamičkog kursora kako se mogu ispisati one knjige koje su izdate u godini koja se unosi sa standardnog ulaza.

```
...
/* deklaracija host promenljivih */
EXEC SQL BEGIN DECLARE SECTION;
    long sifra; /* sifra knjige */
    char naziv[51]; /* naziv knjige */
    char izdavac[31]; /* izdavac */
    short ind_izdavac; /* indikatorska promenljiva za izdavaca */
    short godina; /* godina izdavanja */
    char naredba[255]; /* promenljiva u kojoj ce biti naredba */
EXEC SQL END DECLARE SECTION;
...

/* definisanje naredbe sa parametrom */
sprintf(naredba, "%s", "SELECT k_sifra, naziv, izdavac
                        FROM Knjiga WHERE god_izdavanja = ?");
```

```
/* prevodjenje naredbe iz tekstualnog u izvrsni oblik i
   njeno imenovanje */
EXEC SQL PREPARE select_k FROM :naredba;
CHECKERR("Prepare select_k");

/* definisanje kursora */
EXEC SQL DECLARE izdate_knjige CURSOR FOR select_k;
CHECKERR("Declare izdate_knjige");

printf("Unesite godinu izdavanja: ");
scanf("%hd",&godina);

/* otvaranje kursora i navodjenje vrednosti parametra */
EXEC SQL OPEN izdate_knjige USING :godina;
CHECKERR("Open izdate_knjige");

/* sve dok ima redova u rezultatu ... */
while(1){

    /* cita se red po red */
    EXEC SQL FETCH izdate_knjige
        INTO :sifra, :naziv, :izdavac:ind_izdavac;
    CHECKERR("Fetch izdate_knjige");

    /* ako su procitani svi redovi izlazi se iz petlje */
    if (SQLCODE==100) break;

    /* inace, stampa se red rezultata */
    printf("Sifra: %ld Naziv: %s \n", sifra, naziv);

    if (ind_izdavac<0)
        printf("Izdavac: nepoznat\n");
    else
        printf("Izdavac: %s\n", izdavac);
}

/* zatvara se kursor */
EXEC SQL CLOSE izdate_knjige;
CHECKERR("Close izdate_knjige");
...
```

Dakle, ukoliko je u vreme pripreme naredbe poznat broj kolona kojima će se pristupati i njihovi tipovi, rad sa parametrizovanom SELECT naredbom nije nimalo složeniji od rada sa statičkom naredbom. Ovaj scenario odgovara situaciji iz prethodnog primera kada se zna o kojoj se naredbi i nad kojim kolonama radi, te se mogu koristiti host promenljive i može se unapred alocirati prostor za njihovo smeštanje. Ukoliko se u vreme pisanja programa ne zna kojim kolonama će se pristupati, prostor za smeštanje host promenljivih se ne može unapred alocirati, već se to mora raditi u vreme

izvršavanja, korišćenjem SQLDA strukture.

2.3 Direktni pozivi funkcija SUBP (DB2 CLI)

DB2 CLI je IBM-ov C i C++ aplikacioni programski interfejs (API) za pristup relacionim bazama podataka. On koristi pozive funkcija čiji su argumenti SQL naredbe i dinamički ih izvršava.

Program koji koristi ugrađeni SQL je potrebno pretprocesirati da bi se SQL naredbe transformisale u programski kôd, koji se zatim kompajlira, vezuje sa bazu podataka i izvršava. DB2 CLI program ne zahteva pretprocesiranje niti vezivanje za bazu podataka, već koristi standardni skup funkcija za izvršavanje SQL naredbi i odgovarajućih servisa u vreme izvršavanja.

Ova razlika je značajna jer su pretprocesori, tradicionalno, specifični za neki konkretan proizvod koji radi nad bazom podataka, što efektivno vezuje razvijenu aplikaciju za taj proizvod. DB2 CLI omogućava pisanje portabilnih aplikacija koje su nezavisne od konkretnog proizvoda baza podataka. Ova nezavisnost znači da DB2 CLI aplikaciju nema potrebe ponovo pretprocesirati ili iznova vezivati da bi se pristupilo drugom proizvodu baza podataka, već se ona povezuje sa odgovarajućom bazom podataka u vreme izvršavanja programa.

DB2 CLI aplikacije se mogu povezati na veći broj baza podataka, uključujući i veći broj konekcija na istu bazu podataka. Svaka od konekcija ima svoj zasebni prostor.

DB2 CLI takođe eliminiše potrebu za globalnim oblastima za podatke koje kontrolišu aplikacije, kao što su SQLCA i SQLDA, koje se koriste u ugrađenom SQL-u. Umesto toga, DB2 CLI sam alokira i kontroliše potrebne strukture podataka i obezbeđuje mehanizme kojima aplikacija može na njih da referiše.

Slogovi

Program u programskom jeziku C koji koristi direktne pozive DB2 funkcija mora da uključi zaglavlje `sqlcli.h` u kome su deklarirane funkcije, tipovi podataka, strukture, kao i simboličke konstante potrebne za rad u ovom pristupu.

U programu se mogu kreirati i može se raditi sa četiri tipa **slogova** (struktura u C-u):

- *okruženjima* – slog ovog tipa se pravi od strane aplikacije kao priprema za jednu ili više konekcija sa bazom podataka;
- *konekcijama* – slog ovog tipa se pravi da bi se aplikacija povezala sa bazom podataka; svaka konekcija postoji unutar nekog okruženja; aplikacija može biti povezana sa više različitih baza podataka u isto vreme, a može uspostaviti i više različitih konekcija sa istom bazom podataka (maksimalni dozvoljeni broj aktivnih konekcija po slogu okruženja je 512);
- *naredbama* – u aplikaciji se može kreirati jedan ili više slogova naredbi. Svaki od njih čuva informaciju o pojedinačnoj SQL naredbi, koja uključuje i podrazumevani kursor ako je naredba upit. U različito vreme ista CLI naredba može da predstavlja različite SQL naredbe;

- Navedenim slogovima se u programu rukuje putem “pokazivača” na slog (eng. handle). Korišćenjem slogova se izbegava potreba da se alociraju i da se upravlja globalnim promenljivama i strukturama podataka kao što je to slučaj sa strukturama SQLDA ili SQLCA u ugrađenom SQL-u. U narednom tekstu nećemo praviti razliku između sloga i pokazivača na njega.

Slogovi se prave korišćenjem funkcije:

Argumenti ove funkcije su:

- SQLAllocHandle vraća vrednost tipa SQLRETURN (celobrojnu vrednost). Ova vrednost je jednaka SQL_SUCCESS ako je izvršavanje naredbe prošlo uspešno, bez greške, a jednaka je SQL_ERROR ako je izvršavanje dovelo go greške. Svaka funkcija u CLI pristupu vraća osnovnu informaciju o uspešnosti izvršavanja naredbe putem povratne vrednosti (pošto nemamo na raspolaganju promenljivu SQLCODE kao u ugrađenom SQL-u). U slučaju greške prilikom izvršavanja, za detaljniju dijagnostiku potrebno je pozvati neku od funkcija SQLGetDiagRec ili SQLGetDiagField.

[illegible]


```

if(kodGreske1 == SQL_SUCCESS) {
    kodGreske2 = SQLAllocHandle(SQL_HANDLE_DBC,
                               okruzenje, &konekcija);
    if(kodGreske2 == SQL_SUCCESS)
        kodGreske3 = SQLAllocHandle(SQL_HANDLE_STMT,
                                     konekcija, &naredba);
}

```

U prethodnom kodu deklarirani su slogovi okruženja, konekcije i naredbe, redom. Nakon toga se poziva funkcija `SQLAllocHandle` kojom se zahteva slog okruženja (kao drugi argument prosleđuje se `NULL` slog) a dobijeni slog smešta se na adresu koja je prosleđena kao treći argument ove funkcije. Ako je ova funkcija završila sa uspehom, u okviru ovog okruženja se pravi slog konekcije, a zatim se (ako je uspešno dobijena konekcija) u okviru ove konekcije kreira slog naredbe.

Nakon što slogovi više nisu potrebni, treba ih osloboditi. To se postiže pozivom funkcije

```
SQLRETURN SQLFreeHandle(SQLSMALLINT hType, SQLHANDLE h)
```

koja kao argumente prima redom tip i naziv sloga koji je potrebno osloboditi. Pre oslobađanja sloga okruženja potrebno je osloboditi sve slogove konekcija za to okruženje, inače će doći do greške i biće vraćena vrednost `SQL_ERROR`. Pre oslobađanja sloga konekcije potrebno je raskinuti vezu sa svim konekcijama koje postoje u tom slogu konekcije. Na ovaj način moguće je osloboditi i slog naredbe čime se oslobađaju svi resursi koji su zauzeti alokacijom naredbe. Moguće je, takođe, osloboditi i slog opisa.

Povezivanje na bazu podataka

U okviru kôda potrebno je povezati se na bazu podataka i to se postiže naredbom:

```

SQLRETURN SQLConnect(SQLHDBC hdbc,
                     SQLCHAR db_name, SQLSMALLINT db_name_len,
                     SQLCHAR user_name, SQLSMALLINT user_name_len,
                     SQLCHAR password, SQLSMALLINT password_len);

```

pri čemu su argumenti ove funkcije redom:

1. `hdbc` – slog konekcije; funkcija `SQLAllocHandle(SQL_HANDLE_DBC, ..., &hdbc)` mora biti pozvana pre ove funkcije
2. `db_name` – naziv baze podataka
3. `db_name_len` – dužina naziva baze podataka
4. `user_name` – identifikator korisnika
5. `user_name_len` – dužina identifikatora korisnika
6. `password` – šifra

7. password_len – dužina šifre

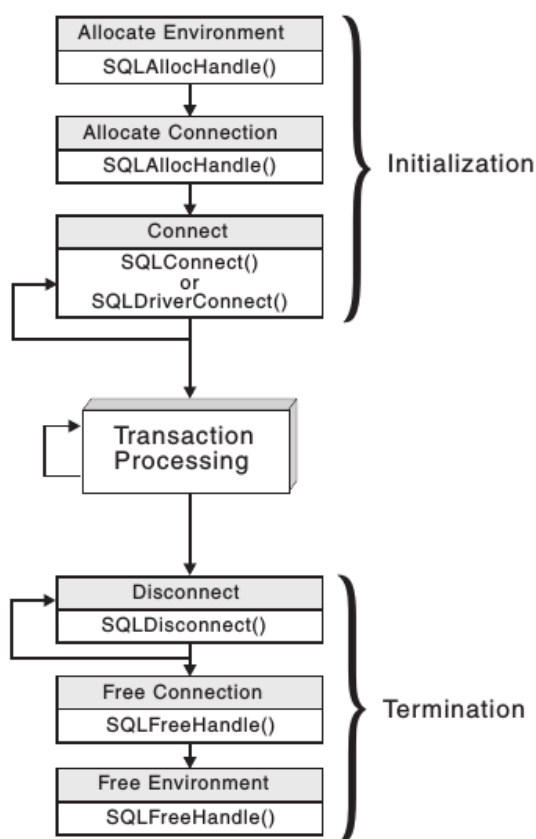
Umesto dužine niske može se proslediti i vrednost SQL_NTS ukoliko je niska terminisana (onda je za određivanje njene dužine dovoljno da se niska skenira dok se ne naiđe na znak '\0'). Ukoliko se na mestu identifikatora korisnika ili šifre (ili na oba ova mesta) prosledi vrednost NULL, konekcija se startuje u ime korisnika koji pokreće CLI aplikaciju. Kao povratna vrednost funkcije vraća se informacija o uspešnosti povezivanja na bazu podataka kao vrednost tipa SQLRETURN (SQL_SUCCESS ako je sve prošlo kako treba).

Naredbom:

```
SQLRETURN SQLDisconnect(SQLHDBC hdbc);
```

raskida se konekcija sa bazom, pri čemu je hdbc slog konekcije. Uspešnost izvršenja ove operacije može se proveriti analizom povratne vrednosti koja je takođe tipa SQLRETURN.

Tipičan redosled koraka pri radu sa slogovima prikazan je na slici 2.3.



Slika 2.3: Faza inicijalizacije i završetka CLI aplikacije

Obrada naredbi

Proces pridruživanja i izvršavanja SQL naredbe je analogan pristupu koji se koristi u dinamičkom SQL-u: tamo smo pridruživali tekst SQL naredbe host promenljivoj korišćenjem naredbe PREPARE i onda je izvršavali korišćenjem naredbe EXECUTE. Ako na slog naredbe gledamo kao na host promenljivu, situacija sa CLI pristupom je prilično slična. Postoji funkcija:

```
SQLRETURN SQLPrepare(SQLHSTMT sh, SQLCHAR st, SQLINTEGER si)
```

čiji su argumenti:

1. sh – slog naredbe
2. st – niska karaktera koja sadrži SQL naredbu
3. si – dužina niske karaktera st. Ukoliko dužina niske nije poznata, a niska je terminisana, može se proslediti konstanta SQL_NTS koja govori funkciji SQLPrepare da sama izračuna dužinu niske.

Efekat ove funkcije je da slog naredbe sh poveže sa konkretnom naredbom st. Funkcijom:

```
SQLRETURN SQLExecute(SQLHSTMT sh)
```

izvršava se naredba na koju referiše slog naredbe sh. I naredba SQLPrepare i naredba SQLExecute vraćaju vrednost tipa SQLRETURN. Za mnoge forme SQL naredbi, kao što su, na primer, umetanja, ažuriranja i brisanja, efekat izvršavanja naredbe SQLExecute je očigledan. Za njih je nakon ove naredbe moguće pozvati naredbu:

```
SQLRETURN SQLRowCount(SQLHSTMT sh, SQLLEN *vr)
```

kojom se vraća broj redova tabele na koje je naredba imala uticaj. Prvi argument ove funkcije je slog naredbe, a drugi pokazivač na lokaciju gde će biti sačuvan rezultat naredbe. Povratna vrednost funkcije je tipa SQLRETURN i vraća informaciju o tome da li je naredba uspešno izvršena.

Manje je očigledno šta je efekat ove naredbe ako je SQL naredba na koju se referiše upit. Kao što ćemo u nastavku videti, deo sloga naredbe je i implicitni kursor. Naredba se izvršava tako što se sve *n*-torke u rezultatu upita negde smeštaju i čekaju da im se pristupi. Korišćenjem implicitnog kursora možemo čitati jednu po jednu *n*-torku, na način sličan onome kako smo radili sa kursorima u ugnježđenom SQL-u.

Na primer, slogu naredba iz prethodnog primera može se pridružiti naredba "SELECT god_izdavanja FROM Knjiga", a zatim se ona može izvršiti na sledeći način:

```
SQLPrepare(naredba, "SELECT god_izdavanja FROM Knjiga", SQL_NTS);
SQLExecute(naredba);
```

ili

```
char *tekst_naredbe = "SELECT god_izdavanja FROM Knjiga";
SQLPrepare(naredba, (SQLCHAR *)tekst_naredbe, strlen(tekst_naredbe));
SQLEExecute(naredba);
```

Priprema i izvršavanje naredbe mogu se objediniti u jedan korak korišćenjem funkcije `SQLExecDirect`, nalik funkciji `EXECUTE IMMEDIATE` u ugrađenom SQL-u. Ovu naredbu ima smisla koristiti u situacijama kada se naredba izvršava samo jednom.

Na primer, prethodne dve naredbe mogli bismo zapisati i na sledeći način:

```
SQLExecDirect(naredba,
              "SELECT god_izdavanja FROM Knjiga", SQL_NTS);
```

Za razliku od dinamičkog ugnježdenog SQL-a, ovde je dozvoljeno da SQL naredba koja se izvršava korišćenjem naredbe `SQLExecDirect` sadrži oznake parametara.

Čitanje podataka iz rezultata upita

Kursor se otvara kada se dinamička `SELECT` naredba uspešno izvrši pozivom naredbe `SQLExecute` ili `SQLExecDirect`.

Funkcija u CLI pristupu koja odgovara naredbi `FETCH` u ugrađenom SQL-u jeste:

```
SQLRETURN SQLFetch(SQLHSTMT sh)
```

gde je `sh` slog naredbe. Preduslov za pozivanje ove funkcije je da je naredba na koju referiše slog `sh` već izvršena; ukoliko nije, čitanje podataka izazvaće grešku. Funkcija `SQLFetch` kao i sve CLI funkcije vraća vrednost tipa `SQLRETURN` koja predstavlja indikator da li je naredba uspešno izvršena. Ukoliko nema više n -torki u rezultatu upita, funkcija vraća vrednost `SQL_NO_DATA`. Kao i u slučaju ugrađenog SQL-a, ova vrednost će se koristiti da bismo izašli iz petlje u kojoj čitamo nove n -torke.

Dakle, u slučaju naredbe `SELECT` poziv funkcije `SQLExecute` biće praćen sa jednim ili više poziva funkcije `SQLFetch`. Komponente n -torke koja se trenutno čita smeštaju se u jedan od opisnih slogova koji su pridruženi naredbi čiji se slog javlja u pozivu funkcije `SQLFetch`. Pre nego što krenemo da čitamo podatke, komponenta n -torke se može povezati sa promenljivom u matičnom jeziku, pozivom funkcije:

```
SQLRETURN SQLBindCol(SQLHSTMT sh,
                     SQLUSMALLINT colNo, SQLSMALLINT colType,
                     SQLPOINTER pVar, SQLLEN varSize,
                     SQLLEN *varInfo)
```

sa narednom listom argumenata:

1. `sh` je slog naredbe,
2. `colNo` je redni broj komponente (u okviru n -torke) čije vrednosti čitamo,

3. `colType` je kôd tipa promenljive u koju se smešta vrednost komponente (kodovi su definisani u zaglavlju `sqlcli.h` i imaju vrednosti npr. `SQL_CHAR` za niske karaktera ili `SQL_INTEGER` za cele brojeve),
4. `pVar` je pokazivač na promenljivu u koju se smešta vrednost,
5. `varSize` je veličina u bajtovima promenljive na koju pokazuje `pVar`,
6. `varInfo` je pokazivač na celobrojnu vrednost koja se može koristiti da bi se obezbedile neke dodatne informacije; na primer poziv `SQLFetch` vraća vrednost `SQL_NULL_DATA` na ovom mestu argumenta, ako je vrednost kolone `NULL`

Napišimo funkciju kojom bi se utvrđivalo da li je više knjiga izdato u 2000. godini ili u 2010. godini. Da bi kôd bio što sažetiji preskačemo sve provere greški, osim testiranja da li je funkcija `SQLFetch` vratila da nema više *n*-torki.

```
#include "sqlcli.h"

void vise_knjiga(){
    int br_2000 = 0, br_2010 = 0;
    SQLHENV okruzenje;
    SQLHDBC konekcija;
    SQLHSTMT naredba;
    SQLSMALLINT godina, godinaInfo;

    SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE, &okruzenje);
    SQLAllocHandle(SQL_HANDLE_DBC, okruzenje, &konekcija);
    SQLAllocHandle(SQL_HANDLE_STMT, konekcija, &naredba);
    SQLPrepare(naredba, "SELECT god_izdavanja FROM Knjiga", SQL_NTS);
    SQLExecute(naredba);
    SQLBindCol(naredba, 1, SQL_SMALLINT,
               &godina, sizeof(godina), &godinaInfo);
    while(SQLFetch(naredba) != SQL_NO_DATA) {
        if (godina == 2000)
            br_2000++;
        else if (godina == 2010)
            br_2010++;
    }
    if (br_2000 > br_2010)
        printf("Vise knjiga je izdato u 2000. godini\n");
    else
        printf("Vise knjiga je izdato u 2010. godini\n");
    SQLFreeHandle(SQL_HANDLE_STMT, naredba);
    SQLFreeHandle(SQL_HANDLE_DBC, konekcija);
    SQLFreeHandle(SQL_HANDLE_ENV, okruzenje);
}
```

U ovom primeru deklarirali smo promenljivu `godina` koja odgovara host promenljivoj u ugrađenom SQL-u i promenljivu `godinaInfo` koje je potrebna zbog poziva funkcije `SQLBindCol`, ali se u kodu nigde dalje ne koristi.

Prosleđivanje parametara upitu

Ugrađeni SQL daje mogućnost izvršavanja SQL naredbe, pri čemu se deo naredbe sastoji od vrednosti koje su određene tekućim vrednostima host promenljivih. Sličan mehanizam postoji i u CLI pristupu, ali je malo komplikovaniji. Koraci koji su potrebni da bi se postigao isti efekat su:

1. iskoristiti funkciju `SQLPrepare` za pripremu naredbe u kojoj su neki delovi koje nazivamo parametrima zamenjeni znakom pitanja. *i*-ti znak pitanja predstavlja *i*-ti parametar;
2. iskoristiti funkciju `SQLBindParameter` za vezivanje vrednosti za mesta na kojima se nalaze znakovi pitanja;
3. izvršiti upit korišćenjem ovih veza, pozivom funkcije `SQLExecute`. Primećimo da ako promenimo vrednost jednog ili više parametara, moramo ponovo da pozovemo ovu funkciju sa novim vrednostima parametara.

Za vezivanje vrednosti za oznake parametra koristimo funkciju

```
SQLRETURN SQLBindParameter(SQLHSTMT sh, SQLUSMALLINT parNo,
                             SQLSMALLINT inputOutputType, SQLSMALLINT valueType,
                             SQLSMALLINT parType, SQLULEN colSize,
                             SQLSMALLINT decimalDigits,
                             SQLPOINTER parValPtr, SQLLEN bufferLen,
                             SQLLEN *indPtr)
```

koja ima narednih 10 argumenata:

1. `sh` je slog naredbe,
2. `parNo` je redni broj parametra (počevši od 1)
3. `inputOutputType` je tip parametra: `SQL_PARAM_INPUT` – za SQL naredbe koje nisu zapamćene procedure, `SQL_PARAM_OUTPUT` – izlazni parametar zapamćene procedure, `SQL_PARAM_INPUT_OUTPUT` – ulazno-izlazni parametar zapamćene procedure,
4. `valueType` je C tip parametra, npr. `SQL_C_LONG`,
5. `parType` je SQL tip parametra, npr. `SQL_INTEGER`,
6. `colSize` je preciznost odgovarajućeg parametra, npr. maksimalna dužina niske
7. `decimalDigits` je broj decimala, ako je tip parametra `SQL_DECIMAL`
8. `parValPtr` je pokazivač na bafer za smeštanje parametra
9. `bufferLen` je veličina bafera za smeštanje parametra
10. `indPtr` je pokazivač na lokaciju koja sadrži dužinu parametra koja se čuva u `parValPtr`; služi npr. za zadavanje NULL vrednosti parametra tako što se vrednost ovog polja postavi na vrednost `SQL_NULL_DATA`

Naredni primer ilustruje ovaj proces i ukazuje na najznačajnije argumente funkcije `SQLBindParameter`.

Ideja je da se vrednosti šifre knjige i njenog naziva unose sa standardnog ulaza i da se onda unete vrednosti koriste kao deo n -torke koju dodajemo u tabelu `Knjiga`.

```
...
/* učitavamo vrednosti promenljivih sifra_knjige i naziv_knjige */
scanf("%ld",&sifra_knjige);
scanf("%s",naziv_knjige);

SQLPrepare(naredba,
    "INSERT INTO Knjiga(k_sifra, naziv) VALUES (?, ?)", SQL_NTS);
SQLBindParameter(naredba, 1, SQL_PARAM_INPUT, SQL_C_LONG, SQL_INTEGER,
    0, 0, &sifra_knjige, 0, NULL);
SQLBindParameter(naredba, 2, SQL_PARAM_INPUT, SQL_C_CHAR, SQL_CHAR,
    50, 0, naziv_knjige, 50, NULL);
SQLExecute(naredba);
...
```

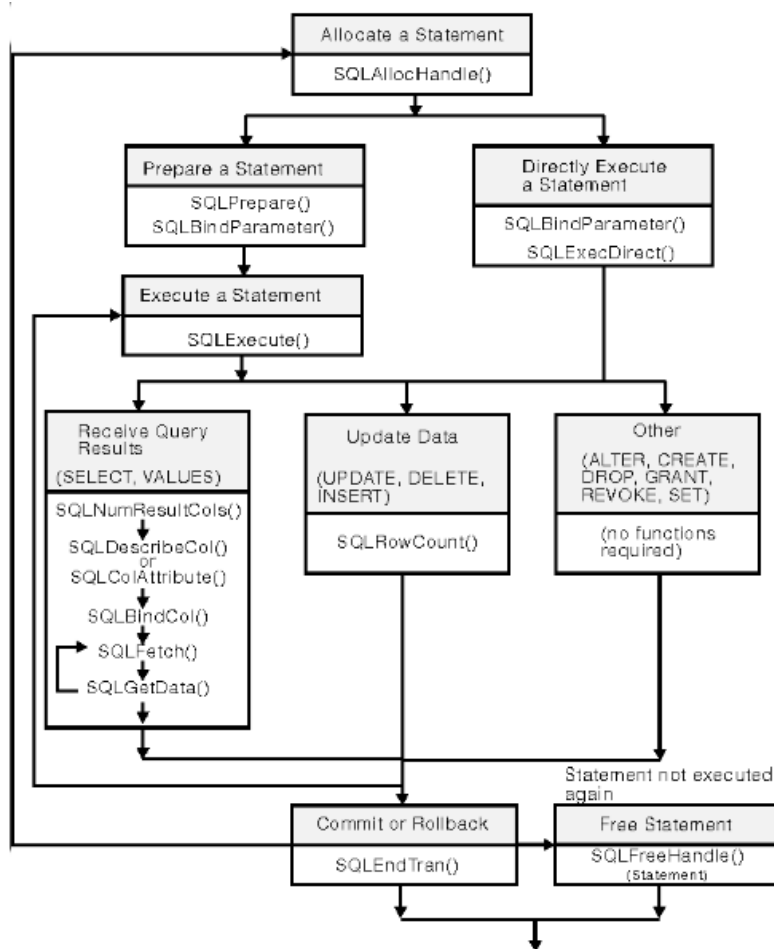
Uobičajen redosled poziva funkcija u DB2 CLI aplikacijama prikazan je na slici 2.4.

Da sumiramo, važi da DB2 CLI može da obradi proizvoljnu naredbu koja se može dinamički pripremiti u ugrađenom SQL-u, ali postoje i razlike:

- u DB2 CLI nije neophodno eksplicitno definisati kursore, već ih sam CLI generiše kada su mu potrebni. Aplikacija može da iskoristi generisani kursor u uobičajenom scenariju – da njime dohvata red po red podataka, kao i za ažuriranje i brisanje tekućeg reda kursora;
- u DB2 CLI nije potrebno otvoriti kursor; umesto toga, obrada naredbe `SELECT` automatski otvara kursor; analogno, kursor nije potrebno ni zatvoriti;
- za razliku od ugrađenog SQL-a, DB2 CLI dozvoljava korišćenje oznaka parametara u naredbi `SQLExecDirect` (ekvivalentu naredbe `EXECUTE IMMEDIATE` u ugrađenom SQL-u);
- informacijama vezanim za naredbu se upravlja od strane aplikacije i obezbeđuje se slog naredbe da bi se referisalo na objekat; na ovaj način se eliminiše potreba da aplikacija koristi strukture podataka specifične za sam proizvod;
- nalik slogu naredbe, slogovi okruženja i konekcije daju mogućnost za referisanje na sve globalne promenljive okruženja i na informacije koje se odnose na samu konekciju.

2.4 Upotreba ODBC standarda

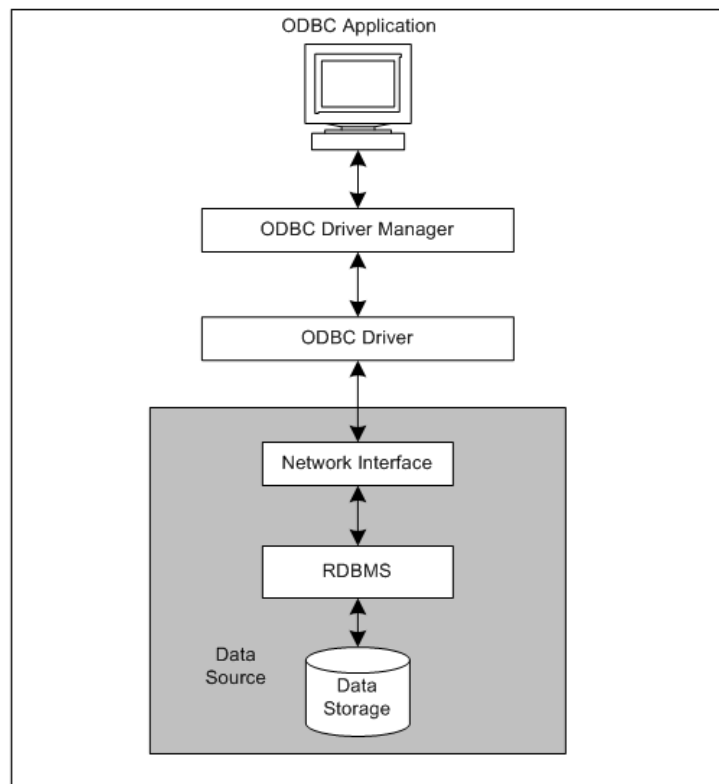
Kao što smo već napomenuli, cilj razvoja CLI interfejsa bio je da se poveća portabilnost aplikacija tako što bi im se omogućilo da postanu nezavisne od konkretnog korisničkog programskog interfejsa. Microsoft je početkom devedesetih godina prošlog veka razvio SQL aplikacioni programski interfejs



Slika 2.4: Obrada transakcije u DB2 CLI

(API) koji se naziva Open Database Connectivity (ODBC) za Microsoft operativne sisteme, koji je zasnovan na preliminarnoj verziji CLI pristupa. ODBC specifikacija uključuje operativno okruženje u kome se ODBC drajveri koji su specifični za određenu bazu podataka dinamički učitavaju u vreme izvršavanja. Aplikacija se direktno linkuje sa bibliotekom menadžera drajvera umesto sa bibliotekom za svaki SUBP. Menadžer drajvera posreduje između ODBC aplikacije i odgovarajućeg ODBC drajvera. Naime, on prihvata pozive funkcija ODBC-a i prosleđuje ih odgovarajućem ODBC drajveru koji je specifičan za odgovarajući SUBP na dalju obradu; on, takođe, prihvata rezultate od drajvera i prosleđuje ih aplikaciji. S obzirom na to da su ODBC menadžeru drajvera poznate samo ODBC funkcije, funkcije koje su specifične za neki SUBP se ne mogu koristiti u ovom okruženju.

ODBC više nije ograničen na Microsoft operativne sisteme; postoje implementacije koje su dostupne na drugim platformama. Pored nezavisnosti od konkretnog sistema za upravljanje bazama podataka, on je nezavisan i od programskog jezika u kome se razvija aplikacija.



Slika 2.5: Arhitektura ODBC okruženja

ODBC aplikacije se mogu relativno jednostavno preneti u CLI pristup i obratno.

JDBC

JDBC API (koji je sličan ODBC API-ju) obezbeđuje Java programerima standardizovan način za pristup i komunikaciju sa bazom podataka, nezavisno od drajvera i proizvoda baza podataka. Java kôd prosleđuje SQL naredbe kao argumente DB2 funkcija JDBC drajveru i drajver ih dalje obrađuje.

Iako sami koncepti u JDBC-u liče na one u CLI pristupu, u JDBC-u je očigledno prisustvo objektno-orijentisane prirode programskog jezika Java.

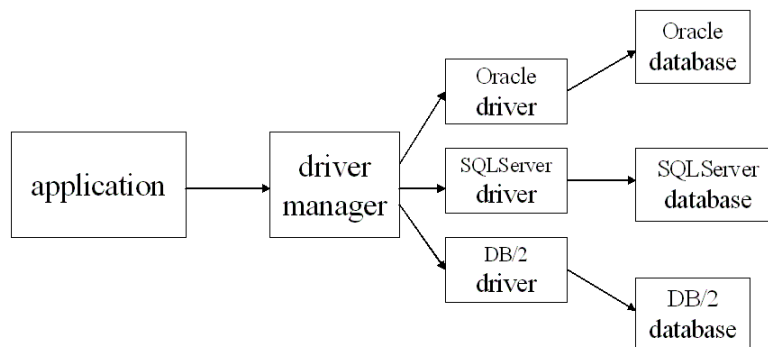
Uvod u JDBC

Da bismo u aplikaciji pisanoj na programskom jeziku Java koristili JDBC potrebno je da:

- uključimo liniju:

```
import java.sql.*;
```

Na ovaj način programu pisanom u programskom jeziku Java su na raspolaganju JDBC klase,



Slika 2.6: Arhitektura JDBC okruženja

- učitamo drajver za sistem za upravljanje bazom podataka koji želimo da koristimo. Drajver koji nam treba zavisi od toga koji SUBP nam je na raspolaganju i on se uključuje naredbom:

```
Class.forName(<ime_drajvera>);
```

Na primer, uključivanje drajvera za SUBP DB2 može se izvesti naredbom:

```
Class.forName("com.ibm.db2.jcc.DB2Driver");
```

dok recimo drajver za SUBP mySQL uključujemo naredbom:

```
Class.forName("com.mysql.jdbc.Driver");
```

Efekat metoda `Class.forName()` je da se odgovarajuća klasa učitava dinamički (u vreme izvršavanja programa). Od ovog momenta aplikaciji je na raspolaganju klasa `DriverManager`. Na ovaj način se može učitati i registrovati proizvoljan broj drajvera. Klasa `DriverManager` je po mnogo čemu slična okruženju čiji slog pravimo kao prvi korak u CLI pristupu.

- uspostavimo konekciju sa bazom podataka. Promenljiva klase `Connection` se pravi primenom metode `getConnection` na objekat klase `DriverManager`.

Metod klase `DriverManager` u Javi kojom se pravi konekcija je:

```
Connection getConnection(String URL,
                          String username, String password);
```

Svaki SUBP ima svoj način zadavanja URL adrese u metodu `getConnection`. Na primer, ako želimo da se povežemo na DB2 bazu podataka, forma URL adrese je:

```
"jdbc:db2://<server_name>:<port_number>/<database_name>"
```

Dakle, ako bismo hteli da se povežemo na bazu podataka bazaKnjiga, pod korisničkim imenom korisnik i šifrom sifra, to bismo mogli da uradimo na sledeći način:

```
Connection konekcija = DriverManager
    .getConnection("jdbc:db2://localhost:50001/bazaKnjiga",
                  "korisnik","sifra");
```

JDBC objekat klase Connection je u velikoj meri sličan slogu konekcije koji koristimo u CLI pristupu i služi istoj svrsi. Primenom odgovarajućih metoda na konekciju konekcija, mogu se napraviti objekti naredbe, smeštati konkretne SQL naredbe u ovaj objekat, vezivati vrednosti za parametre SQL naredbe, izvršavati SQL naredbe, i, ukoliko je u pitanju naredba SELECT pregledati rezultujuće *n*-torke.

Konekcija se raskida pozivom metoda

```
void close()
```

na objekat klase Connection. Poželjno je eksplicitno raskinuti sve konekcije sa bazom kada više nisu potrebne.

Kao i u svakoj aplikaciji pisanoj u programskom jeziku Java, izuzecima se rukuje korišćenjem try-catch blokova. DB2 aplikacija baca izuzetak tipa SQLException uvek kada dođe do greške prilikom izvršavanja SQL naredbe, odnosno SQLWarning kada se prilikom izvršavanja naredbe javi upozorenje. Objekat klase SQLException sadrži informacije o nastaloj grešci koje je moguće dobiti narednim metodama:

- getMessage() – vraća tekstualni opis koda greške,
- getSQLState() – vraća nisku SQLSTATE,
- getErrorCode() – vraća celobrojnu vrednost koja ukazuje na tip greške.

Dakle, obradu izuzetaka možemo raditi na sledeći način:

```
try {
    ...
}
catch (SQLException e) {
    e.printStackTrace();

    System.out.println("SQLCODE: " + e.getErrorCode() + "\n"
                      + "SQLSTATE: " + e.getSQLState() + "\n"
                      + "PORUKA: " + e.getMessage());
    ...
}
```

Pravljenje naredbi u JDBC-u

Naredbu je moguće napraviti korišćenjem dve različite metode od kojih se svaka primenjuje na objekat klase `Connection`:

1. metodom `createStatement()` koja vraća objekat klase `Statement`. Ova metoda se može koristiti za naredbe SQL-a koje ne sadrže parametre. Objektu naredbe kreiranom na ovaj način još uvek nije dodeljena nijedna SQL naredba, te se on može smatrati analognim pozivu funkcije `SQLAllocHandle` u CLI pristupu,
2. metodom `prepareStatement(Q)`, gde je `Q` niska koja sadrži naredbu SQL-a i vraća objekat klase `PreparedStatement`. Ovo odgovara primeni dva koraka u CLI pristupu kojima se prvo dobija pokazivač na slog naredbe korišćenjem funkcije `SQLAllocHandle`, a nakon toga na taj slog i upit `Q` primenjuje funkcija `SQLPrepare`. Ukoliko SQL naredba sadrži oznake parametra, onda je neophodno naredbu kreirati (i pripremiti) na ovaj način. Podsetimo se da priprema naredbe podrazumeva utvrđivanje najefikasnijeg načina da se izdvoje podaci od interesa.

Postoje četiri različita načina na koje se može izvršiti naredba SQL-a. Oni se međusobno razlikuju po tome da li primaju SQL naredbu kao argument ili ne, a, takođe, prave razliku i među SQL naredbama – da li je u pitanju upit ili neka druga naredba (naredba izmene). Pod izmenom podrazumevamo i umetanje novih redova, brisanje postojećih redova, kao i sve naredbe koje rade nad shemom baze podataka, kao što je recimo `CREATE TABLE`. Četiri metode za izvršavanje naredbe su:

1. `executeQuery(Q)` koja kao argument prima nisku `Q` koja sadrži upit u SQL-u i primenjuje je na objekat klase `Statement`. Ovaj metod vraća objekat klase `ResultSet`, koji je skup n -torki u rezultatu upita `Q`.
2. `executeQuery()` koja se primenjuje na objekat klase `PreparedStatement`. S obzirom na to da već pripremljena naredba ima pridružen upit, ova funkcija nema argumenata. I ona vraća objekat klase `ResultSet`.
3. `executeUpdate(U)` koja prima kao argument nisku `U` koja sadrži SQL naredbu koja nije upit i kada se primeni na objekat klase `Statement` izvršava SQL naredbu `U`. Efekat ove operacije se može uočiti samo na samoj bazi podataka; ne vraća se objekat klase `ResultSet` već vraća broj redova na koje je ta naredba uticala, odnosno 0 ako je u pitanju naredba DDL-a.
4. `executeUpdate()` koja nema argument i primenjuje se na već pripremljenu naredbu. U ovom slučaju izvršava se SQL naredba koja je pridružena naredbi. SQL naredba ne sme biti upit.

Pretpostavimo da imamo objekat konekcija klase `Connection` i da želimo da izvršimo upit:

```
SELECT god_izdavanja, izdavac FROM Knjiga
```

Jedan način da to uradimo jeste da napravimo objekat `upit1` klase `Statement` i da ga iskoristimo da bismo direktno izvršili upit:

```
Statement upit1 = konekcija.createStatement();
ResultSet godine_i_izdavaci = upit1.executeQuery(
    "SELECT god_izdavanja, izdavac FROM Knjiga");
```

Rezultat upita je objekat klase `ResultSet` pod nazivom `godine_i_izdavaci`. Videćemo u nastavku kako možemo da izdvojimo n -torke iz ovog objekta i kako da ih obradimo.

S druge strane, možemo prvo pripremiti upit, pa ga naknadno izvršiti. Ovaj pristup je pogodniji ako želimo da isti upit izvršimo veći broj puta. U tom slučaju ima smisla jednom pripremiti upit, a onda ga više puta izvršavati, umesto da SUBP svaki put prilikom izvršavanja iznova priprema isti upit. Koraci u JDBC-u kojima se ovo postiže su sledeći:

```
PreparedStatement upit2 = konekcija.prepareStatement(
    "SELECT god_izdavanja, izdavac FROM Knjiga");
ResultSet godine_i_izdavaci = upit2.executeQuery();
```

Ako pak želimo da izvršimo naredbu koja nije upit i koja nema parametre, možemo da izvedemo analogne korake. Ipak, u ovom slučaju ne postoji rezultujući skup. Pretpostavimo da želimo da dodamo novu knjigu u tabelu `Knjiga`. Možemo napraviti naredbu i izvršiti je kao što smo to radili u slučaju upita:

```
Statement dodaj1 = konekcija.createStatement();
int brRedova = dodaj1.executeUpdate("INSERT INTO Knjiga
    VALUES(111,'Na Drini cuprija','Zavod za udzbenike',2009)");
System.out.println("Broj dodatih redova: " + brRedova);
```

ili:

```
PreparedStatement dodaj2 = konekcija.prepareStatement(
    "INSERT INTO Knjiga VALUES(111,'Na Drini cuprija', " +
    " 'Zavod za udzbenike',2009)");
int brRedova = dodaj2.executeUpdate();
System.out.println("Broj dodatih redova: " + brRedova);
```

Primitimo da je u drugoj naredbi iskorišćen Java operator `+` za konkate-naciju stringova. Stoga smo u mogućnosti da naredbu SQL-a pišemo u više redova ako je potrebno.

Kada naredba više nije potrebna dobra praksa je zatvoriti je pozivom metoda `close()`. Isto važi i za objekte klase `ResultSet`. Ovim se oslobađaju resursi koje su ovi objekti zahtevali.

Operacije nad kursorima u JDBC-u

Kada izvršavamo upit i dobijemo skup redova u rezultatu, možemo pokrenuti kursor kroz n -torke u skupu rezultata. Klasa `ResultSet` raspolaže narednim metodama:

- `next()` – kada se ovaj metod primeni na objekat klase `ResultSet`, uzrokuje da se implicitni kursor pomeri na sledeću n -torke (odnosno na prvu ako se prvi put primenjuje). Ova metoda vraća `FALSE` ako nema naredne n -torke u rezultatu upita;
- `getString(i)`, `getInt(i)`, `getFloat(i)` i analogne metode za druge tipove podataka koje SQL vrednosti mogu da imaju – svaka od ovih metoda vraća i -tu komponentu n -torke na koju kursor trenutno ukazuje; pritom se mora koristiti metod koji je odgovarajući za tip i -te komponente;
- `close()` – zatvara se objekat klase `ResultSet`; ovim se oslobađaju resursi baze podataka i JDBC resursi koje je ovaj objekat zauzimao; skup rezultata se automatski zatvara ako se zatvori naredba za koju je vezan;
- `wasNull()` – proverava da li je poslednja pročitana vrednost iz rezultujućeg skupa bila `NULL` – na ovaj način se u JDBC pristupu rukuje nedostajućim vrednostima (nalik indikatorskim promenljivama u ugrađenom SQL-u);
- `first()` i `last()` – kursor se pozicionira na prvi, odnosno poslednji red u rezultatu; vraća `FALSE` ako nema redova u rezultujućem skupu.

Ako smo u prethodnom primeru dobili rezultujući skup `godine_i_izdavaci`, moguće je redom pristupiti njenim n -torkama i izvršiti odgovarajuću obradu. Podsetimo se da se ove n -torke sastoje od dve komponente od kojih je jedna celobrojnog tipa, a druga niska.

U narednoj tabeli dati su preporučeni tipovi podataka programskog jezika Java za najčešće korišćene SQL tipove podataka:

- `SMALLINT` – `short`
- `INTEGER` – `int`
- `DOUBLE` – `double`
- `CHAR(n)`, `VARCHAR(n)` – `java.lang.String`
- `DATE` – `java.sql.Date`
- `TIME` – `java.sql.Time`

Format petlje kojom bismo prošli kroz rezultujući skup `godine_i_izdavaci` bio bi sledeći:

```

while(godine_i_izdavaci.next()){

    int godina = godine_i_izdavaci.getInt(1);
    System.out.println("godina " + godina);

    String izdavac = godine_i_izdavaci.getString(2);
    if (godine_i_izdavaci.isNull())
        System.out.println("izdavac je nepoznat");
    else
        System.out.println("izdavac " + izdavac);
}

godine_i_izdavaci.close();

```

Podrazumevano, kursorom se u JDBC pristupu može kretati samo unapred. Međutim, mogu se definisati i kursori kojima se može kretati i unapred i unazad, navođenjem odgovarajućih vrednosti kao argumenata funkcija `prepareStatement()` ili `createStatement()`. Pored definisanja da li se po rezultujućem skupu može kretati samo u jednom smeru ili u oba smera, postoji i svojstvo koje nazivamo osetljivošću na izmene. Njime se definiše da li su napravljene izmene u bazi podataka vidljive dok prolazimo skupom redova: ako je osetljiv onda su izmene odmah vidljive, a ako nije onda one neće biti vidljive nakon što je otvoren rezultujući skup. Naredne tri konstante se koriste za eksplicitno zadavanje dozvoljenih smerova kojim je moguće kretati se po skupu redova u rezultatu:

- `TYPE_FORWARD_ONLY`: moguće je kretati se samo unapred po skupu redova u rezultatu;
- `TYPE_SCROLL_SENSITIVE`: moguće je kretati se i unapred i unazad i izmene nad bazom podataka napravljene od drugih transakcija i naredbi u istoj transakciji odmah su vidljive rezultujućem skupu;
- `TYPE_SCROLL_INSENSITIVE`: moguće je kretati se i unapred i unazad i izmene nad bazom podataka napravljene od drugih transakcija i naredbi u istoj transakciji nisu odmah vidljive rezultujućem skupu.

Na primer:

```

Statement naredba = konekcija.createStatement(
    ResultSet.TYPE_SCROLL_INSENSITIVE,
    ResultSet.CONCUR_READ_ONLY);

```

Ovde je dodatno postavljeno da se kursor može koristiti samo za čitanje.

Ako se kursorom može kretati i unazad, moguće je, recimo, umesto primene metoda `next()`, metodom `afterLast()` primenjenim na `ResultSet` pozicionirati se nakon svih redova u rezultatu upita, pa napredovati ka početku pozivom metoda `previous()`. Kursor se može definisati tako da služi i za ažuriranje, navođenjem odgovarajućih parametara u okviru naredbe `prepareStatement()`.

Prosleđivanje parametara

Kao i u CLI pristupu, moguće je unutar teksta SQL naredbe navesti znak pitanja kojim se označavaju parametri upita, i onda naknadno vezati vrednosti za ove parametre. Da bismo ovo uradili u JDBC-u, potrebno je napraviti pripremljenu naredbu, a zatim pre izvršavanja naredbe za parametre vezati vrednosti. To se može postići primenom metoda kao što je recimo `setString(i, v)` ili `setInt(i, v)` koji vezuje vrednost `v` za `i`-ti parametar upita; pritom vrednost `v` mora biti odgovarajućeg tipa u odnosu na primenjeni metod za `i`-ti parametar upita. Na ovaj način moguće je proslediti i NULL vrednost parametra, npr. sa `setString(1, null)`;

Pokušajmo da oponašamo CLI kôd koji smo imali u prethodnom poglavlju u kome smo pripremili naredbu za umetanje nove knjige u relaciju `Knjiga`, sa parametrima za šifru i naziv knjige. Odgovarajući Java kôd sledi u nastavku. Pritom, možemo jednom pripremljenu naredbu više puta izvršavati sa različitim vrednostima parametara.

```
PreparedStatement naredba_insert_param =
    konekcija.prepareStatement(
        "INSERT INTO Knjiga(k_sifra,naziv) VALUES(?,?)");

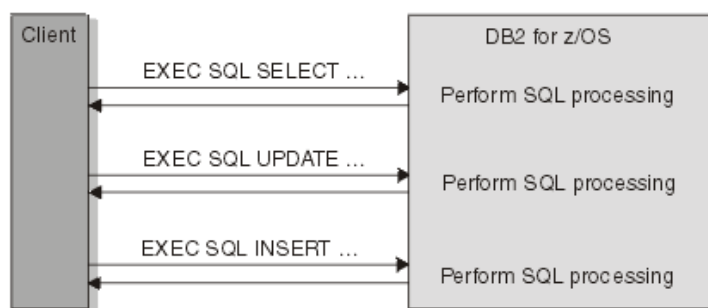
naredba_insert_param.setInt(1,10765);
naredba_insert_param.setString(2,"Na Drini cuprija");
naredba_insert_param.executeUpdate();

naredba_insert_param.setInt(1,21345);
naredba_insert_param.setString(2,"Gospodjica");
naredba_insert_param.executeUpdate();
```

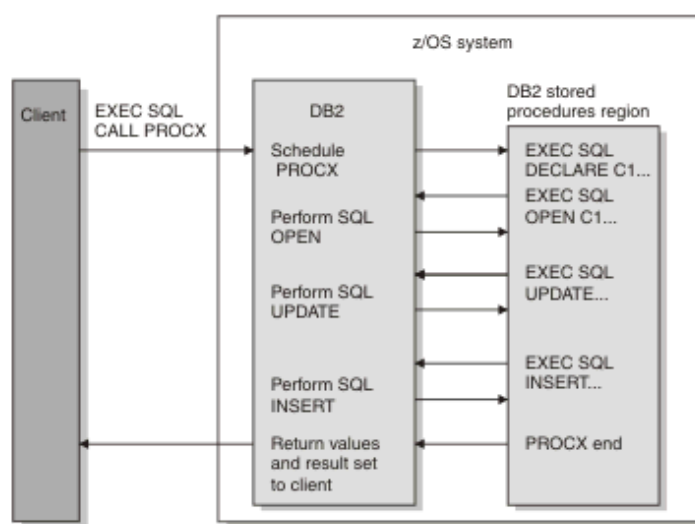
2.5 Zapamćene procedure

U ovom poglavlju upoznaćemo se sa pojmom **zapamćenih procedura** (eng. stored procedures). One omogućavaju pisanje procedura u jednostavnom jeziku opšte namene i njihovo pamćenje u bazi podataka, kao dela sheme. Tako definisane procedure se mogu koristiti u SQL upitima i drugim naredbama za izračunavanja koja se ne mogu uraditi samo pomoću SQL-a. Ovde ćemo opisati standard SQL/PSM koji predstavlja proširenje SQL-a konstruktima iz proceduralnih jezika i koji se prevashodno koristi u zapamćenim procedurama. SQL/PSM standardizuje sintaksu i semantiku naredbi kontrole toka, rukovanja izuzecima (koje nazivamo uslovima), lokalnih promenljivih, dodele izraza promenljivama i parametara i slično. IBM-ov SQL PL je jedan od prvih proizvoda koji je zvanično implementirao SQL/PSM.

Zapamćena procedura je programski blok koji se poziva iz aplikacije na klijentu a izvršava se na serveru baza podataka. Piše se u odgovarajućim proširenjima SQL-a, kompilira i pamti u biblioteci odgovarajućeg SUBP. Tako se primenom zapamćenih procedura deo logike aplikacije prenosi sa klijenta na server, i povećava se funkcionalnost servera. Najčešći razlog za korišćenje zapamćenih procedura je intenzivna obrada podataka iz baze podataka koja proizvodi malu količinu rezultujućih podataka, ili činjenica da je skup operacija (koje se izdvajaju u zapamćenu proceduru) zajednički za više aplikacija.



Slika 2.7: Obrada podataka bez zapamćenih procedura



Slika 2.8: Obrada podataka sa zapamćenim procedurama

Zapamćene procedure ostvaruju mnoge prednosti:

1. koriste prednosti moćnih servera;
2. donose poboljšanja performansi statičkom SQL-u;
3. smanjuju mrežni saobraćaj;
4. poboljšavaju integritet podataka dopuštanjem raznim aplikacijama da pristupe istom programskom kodu.

Zapamćene procedure je moguće pozvati i iz aplikacije napisane u ugrađenom SQL-u, i iz aplikacije koja koristi direktne pozive DB2 funkcija, kao i iz ODBC aplikacija.

Definisanje funkcija i procedura

U SQL/PSM-u definišu se *moduli* koji predstavljaju kolekcije definicija procedura i funkcija, deklaracija privremenih relacija i još nekih opcionih deklaracija.

Procedura se definiše na sledeći način:

```
CREATE PROCEDURE <naziv>(<parametri>)  
<lokalne_deklaracije>  
<telo_procedure>;
```

Funkcija se definiše na sličan način, osim što se koristi ključna reč FUNCTION i što postoji povratna vrednost koja se mora zadati.

```
CREATE FUNCTION <naziv>(<parametri>) RETURNS <tip>  
<lokalne_deklaracije>  
<telo_funkcije>;
```

Parametri procedure se ne zadaju samo svojim nazivom i tipom, već im prethodi i naziv *moda*, koji može biti IN za parametre koji su samo ulazni, OUT za parametre koji su samo izlazni, odnosno INOUT za parametre koji su ulazni-izlazni. Podrazumevana vrednost je IN i može se izostaviti. S druge strane, parametri funkcije mogu biti samo ulazni. To znači da se zabranjuju bočni efekti funkcija, te je jedini način da dobijemo informaciju od funkcije putem njene povratne vrednosti. Nećemo navoditi IN mod za parametre funkcija, a hoćemo u procedurama.

U okviru tela procedure i funkcije može se naći veliki broj različitih naredbi, između ostalog i SQL naredbe. Od upita može se naći onaj koji vraća tačno jedan red u rezultatu, a ako upit vraća potencijalno više redova u rezultatu onda se mora pristupati putem kursora.

Prikažimo proceduru koja kao argumente prima dva naziva izdavača: stari i novi i zamenjuje sve pojave starog naziva izdavača u tabeli Knjiga novim nazivom.

```
CREATE PROCEDURE Zameni(IN stari_izdavac char(30),  
                        IN novi_izdavac char(30))  
  
UPDATE Knjiga  
SET izdavac = novi_izdavac  
WHERE izdavac = stari_izdavac;
```

Ova procedura ima dva parametra, oba su niske dužine 30 karaktera, što je u skladu sa tipom atributa izdavac u tabeli Knjiga. Imena parametara se mogu koristiti kao da su konstante. Ispred naziva parametara i lokalnih promenljivih procedura i funkcija se ne navodi dvotačka.

Neke jednostavne forme naredbi

Nabrojimo neke jednostavne naredbe:

1. *naredba poziva*: Forma poziva procedure je:

```
CALL <naziv_procedure>(<lista_argumenata>);
```

Ovaj poziv se može javiti kao deo programa na matičnom jeziku. Recimo prethodnu proceduru bismo mogli da pozovemo sa:

```
EXEC SQL CALL Zameni('Prosveta','Nova prosveta');
```

ili

```
EXEC SQL CALL Zameni('Prosveta',:x);
```

Poziv zapamćene procedure se može naći i kao deo neke druge funkcije ili procedure. Primetimo da funkciju nije moguće pozvati na ovaj način već se funkcija poziva, kao i u C-u, navođenjem njenog naziva i odgovarajuće liste argumenata u okviru nekog izraza.

2. *naredba vraćanja vrednosti*: njena forma je:

```
RETURN <izraz>;
```

Ova naredba se može javiti samo u funkciji. Ona izračunava vrednost izraza, postavlja povratnu vrednost funkcije na tu vrednost i prekida izvršavanje funkcije.

3. *deklaracija lokalnih promenljivih*: naredba oblika:

```
DECLARE <naziv> <tip>;
```

deklariše promenljivu datog tipa sa datim nazivom. Ova promenljiva je lokalna i njena vrednost se ne čuva od strane SUBP-a nakon izvršavanja funkcije ili procedure. Deklaracije moraju da prethode izvršnim naredbama u telu funkcije ili procedure.

4. *naredbe dodele*: Dodela ima naredni oblik:

```
SET <promenljiva> = <izraz>;
```

Osim početne reči SET dodela je nalik dodeli u drugim jezicima. Izračunava se vrednost izraza sa desne strane jednakosti i ta vrednost se dodeljuje promenljivoj sa leve strane znaka jednakosti. Dozvoljeno je da vrednost izraza bude NULL. Šta više, izraz sa desne strane znaka jednakosti može biti i upit ukoliko on vraća tačno jednu vrednost.

5. *grupe naredbi*: Možemo formirati listu naredbi koje se završavaju tačkom i zarezom i koje su ograđene ključnim rečima BEGIN i END. Ovaj konstrukt se tretira kao jedinstvena naredba i može se javiti u zapamćenoj proceduri svuda gde se može javiti jedna naredba. Na primer, pošto se za telo procedure ili funkcije očekuje jedinstvena naredba, moguće je u telo staviti proizvoljan niz naredbi i ograditi ih ključnim rečima BEGIN i END.
6. *imenovanje naredbi*: naredbu možemo imenovati tako što joj kao prefiks navedemo ime i stavimo dvotačku.

Naredbe grananja

U zapamćenim procedurama podržano je grananje korišćenjem naredbe IF. Ona se razlikuje od istoimene naredbe u programskom jeziku C u narednim tačkama:

- naredba se završava ključnim rečima `END IF`;
- naredba IF koja je umetnuta u ELSE klauzu se uvodi jedinstvenom ključnom rečju `ELSEIF`.

Stoga je opšta forma naredbe IF sledeća:

```
IF <uslov> THEN
    <lista_naredbi>
ELSEIF <uslov> THEN
    <lista_naredbi>
ELSEIF
    ...
[ELSE
    <lista_naredbi>]
END IF;
```

Uslov je proizvoljni logički izraz. Poslednji ELSE i njegova lista naredbi su opciona.

U nastavku je dat kôd funkcije koja kao argumente prima naziv izdavača *i* i godinu *g* i vraća logičku vrednost TRUE ako *i* samo ako je izdavač *i* izdao bar jednu knjigu u godini *g* ili ako u toku te godine nije izdata nijedna knjiga.

```
CREATE FUNCTION IzdaoUGod(i CHAR(30), g SMALLINT) RETURNS BOOLEAN
IF NOT EXISTS(
    SELECT * FROM Knjiga WHERE god_izdavanja = g)
THEN RETURN TRUE;
ELSEIF 1 <=
    (SELECT COUNT(*) FROM Knjiga WHERE god_izdavanja = g AND
     izdavac = i)
THEN RETURN TRUE;
ELSE RETURN FALSE;
END IF;
```

Upiti

Postoji nekoliko načina na koje se u zapamćenim procedurama mogu koristiti upiti:

1. upiti se mogu koristiti u uslovima, odnosno u opštem slučaju na svakom mestu gde se u SQL-u može naći podupit (u prethodnom kôdu videli smo dva primera upotrebe podupita);
2. upiti koji vraćaju jednu vrednost mogu se koristiti sa desne strane naredbe dodele;

3. upit koji vraća tačno jedan red je legalna naredba u zapamćenoj proceduri. Podsetimo se da ova naredba ima INTO stavku kojom se zadaju promenljive u koje se smeštaju komponente te jedinstvene n -torke koja se vraća. Ove promenljive mogu biti lokalne promenljive ili parametri procedure;
4. nad upitom možemo deklarirati i koristiti kursor, isto kao kod ugrađenog SQL-a. Jedina razlika je u tome što se ne javlja EXEC SQL na početku naredbe i promenljive ne koriste dvotačku kao prefiks.

Na primer, možemo napisati zapamćenu proceduru koja sadrži naredbu SELECT INTO na sledeći način:

```
CREATE PROCEDURE Obrada_knjige(IN sifra_knjige INTEGER)

DECLARE naziv_knjige CHAR(50);

BEGIN
    SELECT naziv INTO naziv_knjige
    FROM Knjiga
    WHERE k_sifra = sifra_knjige;
    ...
    /* ovde ide dalja obrada naziva knjige */
    ...
END
```

Isti efekat bismo mogli da postignemo i na drugačiji način:

```
CREATE PROCEDURE Obrada_knjige(IN sifra_knjige INTEGER)

DECLARE naziv_knjige CHAR(50);

BEGIN
    SET naziv = (SELECT naziv FROM Knjiga
                  WHERE k_sifra = sifra_knjige);
    ...
    /* ovde ide dalja obrada naziva knjige*/
    ...
END
```

Petlje

U zapamćenim procedurama moguće je koristiti i petlje. Osnovna konstrukcija za petlju je oblika:

```
LOOP
<lista_naredbi>
END LOOP;
```

Ovim se definiše beskonačna petlja iz koje se može izaći samo nekom naredbom za transfer kontrole toka. Ova vrsta petlje se često imenuje da bi se iz nje moglo izaći korišćenjem naredbe:

```
LEAVE <naziv_petlje>;
```

Najčešće se u petlji putem kursora čitaju n -torke u rezultatu upita i iz petlje je potrebno izaći kada više nema n -torki u rezultatu upita. Korisno je uvesti naziv za uslov na vrednost promenljive SQLSTATE koji ukazuje da nijedna n -torka nije pročitana (za vrednost '02000'). To je moguće uraditi na sledeći način:

```
DECLARE Not_Found CONDITION FOR SQLSTATE '02000';
```

U opštem slučaju možemo deklarirati uslov sa proizvoljnim imenom koji odgovara proizvoljnoj vrednosti promenljive SQLSTATE:

```
DECLARE <naziv> CONDITION FOR SQLSTATE <vrednost>;
```

Napišimo sad proceduru koja ima tri argumenta: jedan ulazni argument i koji označava naziv izdavača i dva izlazna argumenta $sr_vrednost$ i $varijansa$ koji na kraju izvršenja procedure treba da sadrže srednju vrednost i varijansu godina izdavanja svih knjiga koje je izdao izdavač i .

```
CREATE PROCEDURE SrVredVar(IN i CHAR(30),
                           OUT sr_vrednost REAL,
                           OUT varijansa REAL)

DECLARE Not_Found CONDITION FOR SQLSTATE '02000';
DECLARE KnjigeKursor CURSOR FOR
  SELECT god_izdavanja FROM Knjige WHERE izdavac = i;
DECLARE nova_god SMALLINT;
DECLARE broj_knjiga INTEGER;

BEGIN
  SET sr_vrednost = 0.0;
  SET varijansa = 0.0;
  SET broj_knjiga = 0;

  OPEN KnjigeKursor;

  knjigePetlja: LOOP
    FETCH KnjigeKursor INTO nova_god;
    IF Not_Found THEN LEAVE knjigePetlja END IF;
    SET broj_knjiga = broj_knjiga + 1;
    SET sr_vrednost = sr_vrednost + nova_god;
    SET varijansa = varijansa + nova_god * nova_god;
  END LOOP;

  SET sr_vrednost = sr_vrednost/broj_knjiga;
  SET varijansa = varijansa/broj_knjiga - sr_vrednost * sr_vrednost;
  CLOSE KnjigeKursor;
END;
```

For petlja

Pored petlje LOOP, postoji i FOR petlja, ali se ona koristi samo za iteriranje kroz kursor. Naredba FOR petlje ima sledeći oblik:

```
FOR [<naziv_petlje> AS] [<naziv_kursora> CURSOR FOR]
  <upit>
DO
  <lista_naredbi>
END FOR;
```

Ova naredba ne samo da deklarise kursor već nas oslobađa velikog broja tehničkih detalja: otvaranja i zatvaranja kursora, čitanja podataka i provere da li više nema n -torki za čitanje. Ipak, s obzirom na to da ne čitamo sami podatke, ne možemo zadati promenljive u koje treba smestiti komponente n -torke. Stoga se imena koja se koriste za atribut u rezultatu upita tretiraju kao lokalne promenljive istog tipa. Ukoliko se ne zada naziv kursora, DB2 će automatski generisati jedinstveni naziv kursora za interne potrebe. Na kursor koji je deklarisan u FOR petlji se ne može referisati van petlje: dakle poziv naredbe OPEN, FETCH ili CLOSE rezultovaće greškom. U okviru FOR petlje može se javiti naredba LEAVE, ali se onda kompletna petlja mora imenovati.

Zapišimo prethodni primer korišćenjem FOR petlje. Mnoge stvari se ne menjaju: deklaracija procedure je ista, kao i deklaracija lokalne promenljive. Ipak, nije potrebno da deklarishemo kursor u sekciji za deklaraciju procedure niti da definišemo uslov kad podaci nisu nađeni. Primetimo da se podatku god_izdavanja pristupa putem naziva atributa, a ne putem neke nove lokalne promenljive.

```
CREATE PROCEDURE SrVredVar(IN i CHAR(30),
  OUT st_vrednost REAL, OUT varijansa REAL)

DECLARE br_knjiga INTEGER;

BEGIN
  SET sr_vrednost = 0.0;
  SET varijansa = 0.0;
  SET br_knjiga = 0;

  FOR knjigePetlja AS KnjigeKursor CURSOR FOR
    SELECT god_izdavanja FROM Knjige WHERE izdavac = i;
  DO
    SET br_knjiga = br_knjiga + 1;
    SET sr_vrednost = sr_vrednost + god_izdavanja;
    SET varijansa = varijansa + god_izdavanja * god_izdavanja;
  END FOR;

  SET sr_vrednost = sr_vrednost/br_knjiga;
  SET varijansa = varijansa/br_knjiga - sr_vrednost * sr_vrednost;
END;
```

While petlja

U zapamćenim procedurama moguće je koristiti i WHILE petlju, čija je sintaksa:

```
WHILE <uslov> DO  
    <lista_naredbi>  
END WHILE
```

Repeat-until petlja

Pored navedenih vrsta petlji postoji podrška i za zadavanje REPEAT-UNTIL petlju, čije se telo izvršava barem jednom i iz koje se izlazi čim dati uslov postane tačan. Njena sintaksa je:

```
REPEAT  
    <lista_naredbi>  
UNTIL <uslov>  
END REPEAT
```

Na primer:

```
REPEAT  
    DELETE FROM Knjiga WHERE god_izdavanja = x;  
    SET x = x + 1;  
    UNTIL x > 2000  
END REPEAT
```

Izuzeci

SQL sistem nam ukazuje na potencijalne greške postavljanjem nenula niza vrednosti u nisku SQLSTATE. Videli smo primer ovog kôda: kôd '02000' ukazuje na to da nije pronađena naredna n -torka u rezultatu upita, dok recimo kôd '21000' ukazuje da je SELECT koji vraća jedan red vratio više od jednog reda.

U zapamćenoj proceduri moguće je da deklarišemo deo koda, koji nazivamo *hvatač izuzetka* (eng. exception handler) koji se poziva uvek kada promenljiva SQLSTATE dobije kao vrednost jedan od kodova iz date liste kodova. Svaki hvatač izuzetaka je vezan za neki blok koda, ograđenim ključnim rečima BEGIN i END. Hvatač se javlja u okviru ovog bloka i primenjuje se samo na naredbe u ovom bloku. Njegove komponente su:

1. lista kodova izuzetaka za koje se poziva hvatač,
2. kôd koji se izvršava kada se uhvati neki od pridruženih izuzetaka,
3. naznaka gde treba ići nakon što je hvatač završio posao.

Deklaracija hvatača izuzetaka ima sledeći oblik:

```
DECLARE <gde_ici_nakon> HANDLER FOR <lista_uslova>  
    <naredba>
```

Podržane su naredne vrednosti za to gde ići nakon obrade izuzetka:

1. CONTINUE – označava da se nakon izvršavanja naredbe iz tela deklaracije hvatača, izvršava naredba nakon one koja je proizvela izuzetak,
2. EXIT – označava da se nakon izvršavanja naredbe iz tela deklaracije hvatača izlazi iz bloka označenog sa BEGIN i END i izvršava se prva naredba nakon ovog bloka,
3. UNDO – ima isto značenje kao EXIT, osim što se poništavaju sve izmene na bazi podataka ili lokalnim promenljivama koje su izvele naredbe iz ovog bloka.

U listi uslova se mogu naći ili deklarirani uslovi (npr. Not_found) ili izrazi koji uključuju promenljivu SQLSTATE i niske dužine 5.

Pogledajmo primer funkcije koja kao argument prima naziv knjige i vraća godinu u kojoj je knjiga izdata. Ako ne postoji knjiga sa datim nazivom ili postoji više knjiga sa datim nazivom, potrebno je vratiti NULL.

```
CREATE FUNCTION IzracunajGodinu(n CHAR(50)) RETURNS SMALLINT

DECLARE Not_Found CONDITION FOR SQLSTATE '02000';
DECLARE Too_Many CONDITION FOR SQLSTATE '21000';

BEGIN
    DECLARE EXIT HANDLER FOR Not_Found, Too_Many
        RETURN NULL;
    RETURN (SELECT god_izdavanja FROM Knjiga WHERE naziv = n);
END;
```

U PSM procedurama postoji podrška za dve vrste komentara: linijske koji se označavaju sa `--` i važe do kraja reda i blokove koje kojima se komentar ograđuje oznakama `/* i */`.

Glava 3

Transakcije

3.1 Transakcija i integritet

Transakcija je logička jedinica posla pri radu sa podacima. Ona predstavlja niz radnji koji ne narušava uslove integriteta. Sa stanovišta korisnika, izvršavanje transakcije je atomično.

Po izvršenju kompletne transakcije stanje baze treba da bude *konzistentno*, tj. takvo da su ispunjeni uslovi integriteta. Dakle, posmatrana kao jedinica posla, transakcija prevodi jedno konzistentno stanje baze podataka u drugo takvo stanje baze, dok u međukoracima transakcije konzistentnost podataka može biti i narušena. Transakcija na taj način predstavlja i bezbedno sredstvo za interakciju korisnika sa bazom.

U sistemu DB2, za sadržaj transakcije i proveru uslova integriteta odgovoran je programer, tj. sama aplikacija treba da obezbedi važenje uslova integriteta. Uobičajeno je da u radu sa transakcijama SUBP koristi usluge drugog sistema – *upravljača transakcijâ* (eng. transaction manager). Sistem za upravljanje transakcijama obezbeđuje da se, bez greške, upisuju u bazu podataka ili efekti izvršenja svih radnji od kojih se transakcija sastoji, ili nijedne radnje. Na ovaj način je omogućen jedan aspekt kontrole konzistentnosti baze.

RSUBP DB2 koristi dve *radnje* upravljača transakcijâ: to su COMMIT i ROLLBACK (to su ujedno i naredbe SQL-a), od kojih svaka označava i kraj transakcije: COMMIT označava uspešan kraj transakcije i trajan upis efekata svih radnji transakcije u bazu podataka, dok ROLLBACK označava neuspešan kraj transakcije i poništenje svih efekata koje su proizvele radnje te transakcije.

U sistemu DB2 transakcija je deo aplikacije od početka programa do prve radnje COMMIT ili ROLLBACK, odnosno između dve susedne takve radnje u programu, odnosno od poslednje takve radnje do kraja programa. Ako se nijedna od radnji COMMIT i ROLLBACK ne uključi u program, onda čitav program predstavlja jednu transakciju; na kraju programa se sistemski generiše jedna naredba COMMIT ako je kraj programa uspešan, odnosno jedna naredba ROLLBACK ako je kraj programa neuspešan. Dakle, transakcija može biti čitav program, deo programa ili pojedinačna naredba. Ugnježdene transakcije u sistemu DB2 nisu moguće.

Razmotrimo kao primer deo programa u ugrađenom SQL-u kojim se realizuje transakcija u sistemu DB2 kojom se 100 dinara prebacuje sa računa sa šifrom R102 na račun sa šifrom R203 i kojom se održava dati uslov integriteta nad relacijom Racun.

```
EXEC SQL WHENEVER SQLERROR GOTO ponisti;
```

```
EXEC SQL UPDATE Racun  
  SET Stanje = Stanje - 100  
  WHERE R_sifra = 'R102';
```

```
EXEC SQL UPDATE Racun  
  SET Stanje = Stanje + 100  
  WHERE R_sifra = 'R203';
```

```
EXEC SQL COMMIT;  
goto završena;
```

```
ponisti: EXEC SQL ROLLBACK;
```

```
završena: return;
```

3.2 Konkurentnost

Kako bazu podataka istovremeno koristi veći broj korisnika, nad bazom podataka se istovremeno može izvršavati veći broj transakcija. Prednosti konkurentnog (istovremenog) rada transakcija jeste kraće vreme odziva, kao i veća propusnost.

Mada svaka od transakcija može ispunjavati uslove integriteta baze podataka, ponekad njihovo istovremeno izvršavanje može da naruši te uslove. Pod izvršenjem skupa transakcija ovde se podrazumeva niz radnji od kojih se te transakcije sastoje. U tom nizu redosled radnji iz svake transakcije je nepromenjen, ali dve uzastopne radnje ne moraju pripadati istoj transakciji. Kao osnovne komponente transakcije posmatraćemo objekte (npr. slogove) i dve radnje: *čitanje* i *upis* (tj. ažuriranje). Dve radnje u ovom modelu su *konfliktne* ako se obavljaju nad istim objektom a jedna od njih je radnja upisa. To znači da parovi konfliktnih radnji mogu biti samo parovi (čitanje, upis) i (upis, upis).

Problem kojim ćemo se u nastavku baviti jeste kako obezbediti da konkurentne transakcije ne "smetaju" jedna drugoj i kako obezbediti da njihovo konkurentno izvršavanje ostavi bazu podataka u konzistentnom stanju.

Problemi pri konkurentnom radu

Pri konkurentnom izvršavanju skupa transakcija mogu nastati sledeći problemi:

1. *problem izgubljenih ažuriranja* (eng. lost update problem)
2. *problem zavisnosti od nepotvrđenih podataka* (eng. uncommitted dependency problem)

3. *problem nekonzistentne analize* (eng. inconsistent analysis problem)**Problem izgubljenih ažuriranja**

Neka se svaka od transakcija A i B sastoji od čitanja i upisa istog sloga datoteke, pri čemu transakcije A i B najpre čitaju podatke, a zatim vrše upis u istom redosledu. Na primer, pretpostavimo da imamo tabelu koja za svaki proizvod čuva njegov id, naziv i količinu tog proizvoda na stanju. Ona se koristi kao deo onlajn sistema koji prikazuje koliko proizvoda nekog tipa ima na stanju i potrebno ju je ažurirati svaki put kada se izvrši prodaja. Pretpostavimo da na stanju postoji 10 laptopova odgovarajuće marke. Neka npr. transakcija A čita slog koji se odnosi na tu marku laptopa, a zatim transakcija B čita isti slog (obe transakcije čitaju iste podatke; konkretno one čitaju vrednost 10). Posle toga transakcija A želi da kupi dva laptopa te marke i menja broj laptopova na stanju na 8 i završava sa radom, npr. zatvara datoteku, a zatim transakcija B želi da kupi 3 laptopa te marke i menja broj laptopova na vrednost za 3 manju od pročitano broja laptopova ($10 - 3 = 7$) i završava sa radom (npr. zatvara svoju verziju iste datoteke). Kakav je efekat rada ove dve transakcije? Ostaju zapamćene samo promene transakcije B koja je poslednja izvršila upis i zatvorila datoteku, a transakciji A se bez obaveštenja poništava efekat upisa, tj. gubi se informacija o prodaji 2 laptopa putem transakcije A.

T	Transakcija A	Transakcija B
...
t1	FETCH R	—
t2	—	FETCH R
t3	UPDATE R	—
t4	—	UPDATE R

Tabela 3.1: Problem izgubljenih ažuriranja

Problem zavisnosti od nepotvrđenih podataka

Pretpostavimo da rad transakcije A uključuje čitanje nekog sloga R, a da rad transakcije B uključuje ažuriranje istog sloga. Neka pri konkurentnom izvršavanju ovih transakcija najpre transakcija B ažurira slog R, a zatim transakcija A pročita slog R i završi svoje izvršavanje. U nekom kasnijem trenutku, iz nekog razloga, transakcija B poništi sve efekte koje je do tog trenutka proizvela na bazu podataka, te poništi i ažuriranje sloga R. U tom slučaju transakcija A pročitala je ažuriranu vrednost sloga R koja nije ni trebalo da bude upisana u bazu podataka.

Slično, neka rad transakcije A uključuje ažuriranje sloga R, a neka se transakcija B sastoji od istih radnji kao u prethodnom slučaju. Pri istom redosledu izvršavanja radnji i poništenju transakcije B izgubilo bi se, bez obaveštenja, ažuriranje transakcije A, jer bi poništavanjem efekata transakcije B vrednost sloga R bila vraćena na vrednost pre ažuriranja (od strane transakcije A).

T	Transakcija A	Transakcija B
...
t1	---	UPDATE R
t2	FETCH R	---
t3	---	ROLLBACK

Tabela 3.2: Problem zavisnosti od nepotvrđenih podataka (slučaj (a))

T	Transakcija A	Transakcija B
...
t1	---	UPDATE R
t2	UPDATE R	---
t3	---	ROLLBACK

Tabela 3.3: Problem zavisnosti od nepotvrđenih podataka (slučaj (b))

Problem nekonzistentne analize

Problem inkonzistentne analize se javlja u situaciji kada jedna transakcija čita nekoliko vrednosti, a druga transakcija ažurira neke od tih vrednosti tokom izvršavanja prve transakcije. Recimo u slučaju kada prva transakcija računa neku agregiranu vrednost skupa vrednosti dok druga transakcija ažurira neke od ovih vrednosti, može se desiti da agregatna funkcija iskoristi neke od vrednosti pre nego što su ažurirane, a neke druge nakon ažuriranja i da dobijena vrednost ne oslikava tačnu vrednost agregatne funkcije.

Pretpostavimo da imamo dve transakcije A i B koje rade nad tri bankovna računa: transakcija A sabira stanje na sva tri računa, dok transakcija B prenosi 1000 dinara sa trećeg računa na prvi. Na računu Rac1 nalazi se 4000 dinara, na računu Rac2 nalazi se 5000 dinara, a na računu Rac3 3000 dinara. Ukoliko se transakcija B u potpunosti izvrši između radnji čitanja stanja na računu Rac1 i čitanja stanja na računu Rac3 transakcije A, baza će nakon izvršenja ove dve transakcije biti u nekonzistentnom stanju (u navedenom slučaju transakcija A bi dobila kao zbir stanja na sva tri računa 11000 dinara umesto 12000 dinara).

T	Transakcija A	Transakcija B
...
t1	FETCH Rac1 (4000), sum=4000	---
t2	FETCH Rac2 (5000), sum=9000	---
t3	---	FETCH Rac3 (3000)
t4	---	UPDATE Rac3 (3000→2000)
t5	---	FETCH Rac1 (4000)
t6	---	UPDATE Rac1 (4000→5000)
t7	---	COMMIT
t8	FETCH Rac3 (2000), sum=11000	---

Tabela 3.4: Problem nekonzistentne analize

Problemi izgubljenih ažuriranja i nekonzistentne analize vezani su za redosled konkurentnog izvršavanja radnji dve ili više transakcija. Problem zavisnosti od nepotvrđenih podataka je, osim za konkurentnost, vezan i za oporavak baze podataka nakon poništavanja transakcije pri konkurentnom izvršenju. Do poništenja transakcije može doći, na primer, zbog greške u transakciji ili zbog pada sistema.

Sva tri navedena problema biće rešavana korišćenjem mehanizma *zaključavanja* (eng. locking). Mehanizam zaključavanja obezbeđuje konkurentno izvršenje skupa transakcija koje je ekvivalentno *serijskom izvršenju* transakcija – kada se transakcije izvršavaju jedna za drugom: sve radnje svake transakcije se izvršavaju jedna za drugom, bez preplitanja sa radnjama druge transakcije (svaka transakcija izvršava COMMIT pre nego što je drugoj transakciji dozvoljeno da krene sa izvršavanjem). Za serijsko izvršavanje važi da ne narušava integritet i konzistentnost baze ako to isto važi za svaku pojedinačnu transakciju u posmatranom skupu transakcija.

Kažemo da je izvršavanje datog skupa transakcija korektno ako je *serijalizabilno*, tj. ako proizvodi isti rezultat kao i serijsko izvršavanje istog skupa transakcija. Na primer, ako dve transakcije samo čitaju neki slog, redosled u kome one to rade je nebitan. Takođe, ako transakcija A čita i menja slog S, a transakcija B čita i menja slog T, onda se radnje ove dve transakcije mogu izvršavati u proizvoljnom redosledu.

Zaključavanje

Zaključavanje podrazumeva postavljanje katanaca na objekat. Ideja zaključavanja je sledeća: kada transakcija želi da radi sa nekim resursom ona zahteva zaključavanje tog objekta (postavlja *katanac* nad tim objektom), a kada se transakcija završi taj resurs se oslobađa.

Svaki katanac karakterišu naredna svojstva:

- *vrsta, mod, režim* – određuje načine pristupa koji su dozvoljeni vlasniku katanca, kao i načine pristupa koji su dozvoljeni konkurentnim korisnicima zaključanog objekta,
- *objekat* – resurs koji se zaključava; objekat koji se zaključava takođe određuje *granularnost* (opseg) katanca, a on može biti prostor tabela, tabela, red,
- *trajanje* – vremenski interval tokom koga se katanac drži, on zavisi od nivoa izolovanosti transakcije (o kojima će biti više reči u nastavku).

Pretpostavimo da sistem podržava bar dve vrste katanaca:

- *deljivi* (S – Share) katanac i
- *privatni* ili *ekskluzivni* (X – eXclusive) katanac.

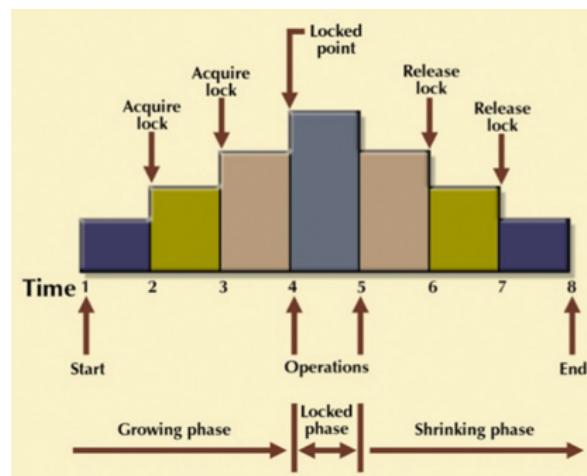
S katanac je potreban za čitanje, a X katanac za ažuriranje. Ako je transakcija A postavila X katanac nad vrstom *t* tada zahtev bilo koje druge transakcije B za postavljanje bilo kog katanca nad *t* biva odbijen. Ako je transakcija A postavila S katanac nad vrstom *t* tada zahtev bilo koje druge transakcije B za postavljanje S katanca nad *t* može biti ispunjen, dok zahtev za postavljanje X katanca nad *t* biva odbijen.

		Transakcija A		
		X	S	-
Transakcija B	X	N	N	D
	S	N	D	D
	-	D	D	D

Tabela 3.5: Matrica kompatibilnosti katanaca; X označava ekskluzivni katanac, S deljivi, a '-' da ne postoji katanac; D označava da transakcija može dobiti željeni katanac, a N da ne može.

Dvofazni protokol zaključavanja (eng. two-phase locking: 2PL) podrazumeva da se izvršavanje svake transakcije sastoji iz dve faze: faze zaključavanja objekta i faze otključavanja objekta i te dve faze se izvršavaju jedna za drugom (serijski). U fazi zaključavanja nema nijedne radnje otključavanja objekta, dok u fazi otključavanja više nema radnji zaključavanja (ali naravno obe ove faze u sebi uključuju i druge operacije kao što su čitanje, upis, itd). Faza otključavanja objekata započinje prvom radnjom otključavanja.

Dakle, kada transakcija izvrši željenu radnju (čitanja, pisanja), ona može da oslobodi katanac na tom slogu, ali nakon što oslobodi neki katanac ne može da zahteva nove katanace.



Slika 3.1: Ilustracija dvofaznog protokola zaključavanja

Razmotrimo naredni primer: transakcija A poštuje dvofazni protokol zaključavanja: svi potrebni katanci se traže (i dobijaju) pre nego što se oslobađa neki od njih. Transakcija B ne poštuje dvofazni protokol zaključavanja: ona oslobađa svoj katanac na slogu X i nakon toga zahteva katanac na slogu Y.

A	B
read-lock(X)	read-lock(X)
Read(X)	Read(X)
write-lock(Y)	unlock(X)
unlock(X)	write-lock(Y)

Read(Y)	Read(Y)
$Y = Y + X$	$Y = Y + X$
Write(Y)	Write(Y)
unlock(Y)	unlock(Y)

Poštovanje dvofaznog protokola zaključavanja garantuje da se navedeni problemi konkurentnosti neće javiti.

Striktni dvofazni protokol zaključavanja se pridržava dvofaznog protokola zaključavanja i, dodatno, u ovom protokolu se ekskluzivni katanaci oslobađaju tek nakon kraja transakcije. Dakle, transakcije ne mogu da čitaju niti menjaju neki slog sve dok poslednja transakcija koja je menjala X nije izvršila COMMIT ili ROLLBACK. Na ovaj način se garantuje da su svi objekti izmenjeni od strane transakcije koja još nije potvrdila svoje izmene, zaključani ekskluzivnim katancom sve dok transakcija ne uradi COMMIT, onemogućavajući druge transakcije da pročitaju izmenjene podatke koji nisu potvrđeni.

Dakle, striktni dvofazni protokol se sastoji od narednih koraka:

1. Transakcija koja želi da pročita neku vrstu mora prvo da nad njom postavi deljivi (S) katanac
2. Transakcija koja želi da ažurira neku vrstu mora prvo da nad njom postavi ekskluzivni (X) katanac. Alternativno, ako već drži S katanac nad tom vrstom, transakcija mora da zahteva unapređenje S katanca u X katanac
3. Ako je zahtev za postavljanjem katanca od strane transakcije B odbijen jer je u konfliktu sa već postavljenim katancom od strane transakcije A, transakcija B ide u stanje čekanja. Transakcija B čeka dok transakcija A ne oslobodi katanac (sistem mora da garantuje da transakcija B neće zauvek da ostane u stanju čekanja)
4. Ekskluzivni katanac se zadržava do kraja transakcije (naredbe COMMIT ili ROLLBACK). Deljivi katanac se, uobičajeno, zadržava (najduže) do kraja transakcije.

Efekat zaključavanja jeste da isforsira serijalizabilnost transakcija. Važi sledeća teorema: ako sve transakcije primenjuju dvofazni protokol zaključavanja, tada su svi mogući rasporedi izvršavanja transakcija serijalizabilni.

Rešenje navedenih problema konkurentnosti

Videćemo sada kako problemi koje smo naveli da su mogući prilikom konkurentnog rada transakcija mogu biti rešeni korišćenjem dvofaznog protokola zaključavanja.

Rešenje problema izgubljenih ažuriranja

U trenutku t_1 transakcija A postavlja S katanac nad R, a u trenutku t_2 transakcija B postavlja S katanac nad R. U trenutku t_3 transakcija A ne može da postavi X katanac nad R i ide u stanje čekanja. U trenutku t_4 transakcija B ne može da postavi X katanac nad R i takođe ide u stanje čekanja. Dakle, ni jedna transakcija ne može da nastavi rad i obe čekaju jedna na drugu, odnosno dolazi do *mrtve petlje* (eng. deadlock).

T	Transakcija A	Transakcija B
...
t1	FETCH R (zahteva S katanac nad R)	---
t2	---	FETCH R (zahteva S katanac nad R)
t3	UPDATE R (zahteva X katanac nad R)	---
t4	čeka	UPDATE R (zahteva X katanac nad R)
t5	čeka	čeka

Tabela 3.6: Problem izgubljenih ažuriranja - rešenje

Rešenje problema zavisnosti od nepotvrđenih podataka

Vratimo se na primer transakcija A i B u primeru problema zavisnosti od nepotvrđenih podataka i pogledajmo koji se niz radnji izvršava pod pretpostavkom dvofaznog protokola zaključavanja.

T	Transakcija A	Transakcija B
...
t1	---	UPDATE R (zahteva X nad R)
t2	FETCH R (zahteva S nad R)	---
t3	čeka	ROLLBACK (tačka sinhronizacije, oslobađa se X sa R)
t4	FETCH R (ponovo se izvršava i postavlja S nad R)	---

Tabela 3.7: Problem zavisnosti od nepotvrđenih podataka - rešenje (slučaj a))

U trenutku t2 transakcija A je sprečena da pročita nepotvrđene promene transakcije B. Kada transakcija B uradi ROLLBACK oslobađa se ekskluzivni katanac koji je ona držala. Nakon toga, u trenutku t4, transakcija A ponovo pokušava da dobije deljeni katanac nad R i ovog puta ga dobija.

Pogledajmo sada drugi scenario u kome transakcija A želi da menja podatke koje je druga transakcija već izmenila, ali svoje izmene još uvek nije potvrdila.

T	Transakcija A	Transakcija B
...
t1	---	UPDATE R (zahteva X nad R)
t2	UPDATE R (zahteva X nad R)	---
t3	čeka	ROLLBACK (tačka sinhronizacije, oslobađa se X sa R)
t4	UPDATE R (ponovo se izvršava i postavlja X nad R)	---

Tabela 3.8: Problem zavisnosti od nepotvrđenih podataka - rešenje (slučaj b))

Pod pretpostavkom dvofaznog protokola zaključavanja, u trenutku t2 transakcija A biće sprečena da ažurira nepotvrđene promene. Nakon što transakcija B uradi ROLLBACK oslobađa se ekskluzivni katanac nad R i transakcija A ponovo pokušava da dobije ekskluzivni katanac nad R i ovaj put ga dobija.

Rešenje problema nekonzistentne analize

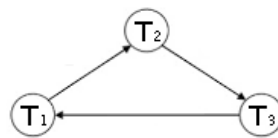
Vratimo se na primer dve transakcije koje rade nad bankovnim računima. Korišćenjem mehanizma katanaca nekonzistentna analiza je sprečena, ali se u trenutku t8 javila mrtva petlja.

T	Transakcija A	Transakcija B
...
t1	FETCH Rac1 (40), sum=40 (postavlja S na Rac1)	---
t2	FETCH Rac2 (50), sum=90 (postavlja S na Rac2)	---
t3	---	FETCH Rac3 (30) (postavlja S na Rac3)
t4	---	UPDATE Rac3 (30→20) (postavlja X na Rac3)
t5	---	FETCH Rac1 (40) (postavlja S na Rac1)
t6	---	UPDATE Rac1 (40→50) (zahteva X na Rac1)
t7	---	čeka
t8	FETCH Rac3 (20) (zahteva S na Rac3)	čeka
t9	čeka	čeka

Tabela 3.9: Problem nekonzistentne analize - rešenje

Mrtva petlja

Kod problema izgubljenih ažuriranja i nekonzistentne analize, rešavanje samog problema dovelo je do novog problema – uzajamnog blokiranja transakcija. Mrtva petlja je situacija kada dve ili više transakcija imaju postavljen katanac nad resursom koji je potreban drugoj transakciji tako da nijedna transakcija ne može da nastavi sa radom. Mrtva petlja se otkriva korišćenjem *grafa čekanja*. To je usmereni graf koji ima po jedan čvor za svaku transakciju, dok grana (T_i, T_j) označava da transakcija T_j drži katanac nad resursom koje je potreban transakciji T_i , te transakcija T_i čeka da transakcija T_j oslobodi katanac na tom resursu. U ovako zadatom grafu mrtva petlja odgovara usmerenom ciklusu u grafu. Primer grafa čekanja dat je na slici 3.2.



Slika 3.2: Primer grafa čekanja u slučaju da transakcija T_1 čeka na transakciju T_2 , transakcija T_2 na transakciju T_3 , a transakcija T_3 na transakciju T_1

Problem uzajamnog blokiranja transakcija nije potrebno sprečiti, već je potrebno efikasno detektovati mrtve petlje. Nakon što se mrtva petlja pronađe, neka od transakcija koja učestvuje u mrtvoj petlji bira se za "žrtvu", njeni efekti se poništavaju i na taj način se oslobađaju svi katanaci koje je ona držala.

Zaključavanje sa namerom

U sistemima za upravljanje bazama podataka često se koristi zaključavanje sa više nivoa granularnosti.

Prilikom zaključavanja reda tabele, tabela koja sadrži taj red se takođe zaključava. Ovo *zaključavanje sa namerom* (eng. intent) ukazuje na plan koji transakcija ima u pristupu podataka: namera da se vrši zaključavanje sa finijom granularnošću. U ovom slučaju, katanac na tabelu je jednog od narednih tipova: IS (intent share), IX (intent exclusive), SIX (share with intent exclusive). Ideja uvođenja ove vrste katanaca je da se konflikt između zahteva transakcija otkrije već na nivou tabele, pre nego na nivou pojedinačnih redova tabele.

Tipovi katanaca i njihovi efekti su izlistani u rastućem redosledu kontrole nad resursima.

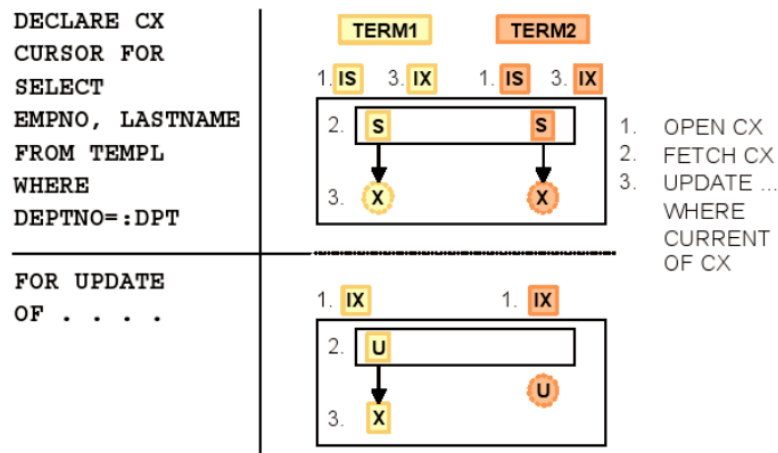
- IS katanac: transakcija koja drži ovu vrstu katanca namerava da postavi S katanac nad pojedinačnim n -torkama tabele, radi garantovanja stabilnosti tih n -torki koje namerava da obrađuje. Za svaki red koji se namerava čitati potrebno je dobiti S katanac nad tim redom.
- IX katanac: transakcija koja drži ovu vrstu katanca može da čita i da menja pojedinačne n -torke tabele, ali pre toga mora da dobije odgovarajući katanac na pojedinačnim redovima koje želi da čita/menja.
- S katanac: transakcija toleriše konkurentno čitanje, ali ne i konkurentno ažuriranje. Ne traže se katanci na pojedinačnim redovima koje transakcija želi da čita. Transakcija nema nameru da ažurira bilo koju n -torku iz tabele.
- SIX katanac: ovo je kombinacija S i IX katanca, odnosno transakcija može da toleriše konkurentno čitanje, ali transakcija namerava da ažurira pojedinačne n -torke tabele u kojem će slučaju postaviti X katanac na te n -torke u tabeli.
- X katanac: transakcija ne toleriše bilo kakav konkurentan pristup tabeli. Samo ona može, ali i ne mora da ažurira n -torke ove tabele i pritom ne postavlja katance na pojedinačne redove tabele.

IS katanac je u konfliktu sa X katancom, dok je IX katanac konfliktan sa S i X katancom. Dakle, veći broj transakcija može istovremeno da drži IX katanac na jednoj tabeli, jer on samo označava nameru da se izmene neki (potencijalno različiti) redovi u tabeli. Dakle, namera IX katanaca je da se blokiraju samo operacije na čitavoj tabeli.

		Transakcija A					
		X	SIX	IX	S	IS	-
Transakcija B	X	N	N	N	N	N	Y
	SIX	N	N	N	N	Y	Y
	IX	N	N	Y	N	Y	Y
	S	N	N	N	Y	Y	Y
	IS	N	Y	Y	Y	Y	Y
	-	Y	Y	Y	Y	Y	Y

Tabela 3.10: Matrica kompatibilnosti katanaca.

U katanac



Slika 3.3: Korišćenje U katanca

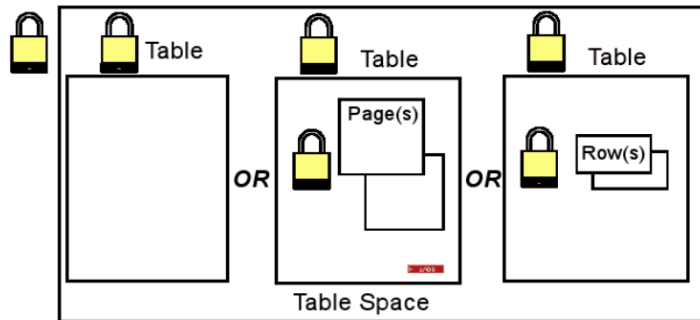
Razmotrimo sledeći scenario: dve aplikacije žele da pozove istu transakciju. U gornjem delu slike 3.3 aplikacija 1 traži da otvori kursor i čita prvi red. Ona dobija IS katanac na tabeli i S katanac na redu. Aplikacija 2 traži da otvori kursor i čita prvi red. Dobija IS katanac na tabeli i S katanac na istom redu. Aplikacija 1 pokušava da ažurira pročitani red. Dobija IX katanac na tabeli, ali joj treba i X katanac na redu. Ona ne može da dobije X katanac na redu zato što aplikacija 2 drži S katanac na redu, te stoga čeka. Aplikacija 2 odlučuje da ažurira red dok ga čeka. Ona dobija IX katanac na tabeli kroz konverziju katanca. Međutim, da bi menjala željeni red, ona mora da dobije X katanac na tom redu, a ne može da ga dobije jer aplikacija 1 na njemu drži S katanac. Ovim dolazimo do mrtve petlje.

Update (U) katanac predstavlja hibrid deljenog i ekskluzivnog katanca. On se zahteva u situaciji kada transakcija želi da čita ali će naknadno želeći i da menja dati red (pre nego što krene da menja tražiće ekskluzivni katanac nad tim redom). Korišćenje U katanca onemogućava da dva poziva iste transakcije naprave mrtvu petlju na istom redu. Namera da se menja onemogućava da dva poziva izvode operacije na istom redu u isto vreme, s obzirom na to da su oba poziva kroz definiciju kursora naglasila moguću želju da dobiju X katanac.

U donjem delu slike 3.3 prikazan je isti scenario u slučaju da se koristi U katanac. Aplikacija 1 otvara kursor i čita prvi red; ona pritom dobija IX katanac na tabeli, a U katanac na redu. Nakon toga aplikacija 2 otvara kursor i, takođe, želi da pročita prvi red. Ona dobija IX katanac na tabeli, ali čeka na U katanac na redu. Aplikacija 1 ažurira red koji čita i promoviše svoj katanac u X katanac na redu. Aplikacija 2 čeka dok aplikacija 1 ne izvrši COMMIT ili dok ne istekne maksimalno dozvoljeno vreme čekanja na katanac. Na ovaj način je dakle, uvođenje U katanaca ograničilo stepen konkurentnosti transakcija, ali je i sprečilo pojavu mrtve petlje.

Zaključavanje u sistemu DB2

DB2 *uvek* zahteva katanac na tabeli pre nego što se dozvoli pristup podacima. DB2 može takođe da zahteva katanac na prostoru tabela, ali ova vrsta katanca se u principu ne odnosi na programera aplikacije, te neće biti diskutovana u nastavku teksta.



Slika 3.4: DB2 koristi ili striktno zaključavanje tabele ili zaključavanje tabele zajedno sa zaključavanjem reda/stranice

DB2 može da zaključa samo tabelu (ovo se nekada naziva *striktno zaključavanje tabele*) ili može da zaključa i tabelu i redove ili stranice tabele (slika 3.4). Ukoliko strategija zaključavanja ne uključuje zaključavanje reda ili stranice, tabela se zaključava u modu koji se odnosi na čitavu tabelu i cela tabela biće zaključana na isti način. Ukoliko strategija zaključavanja uključuje zaključavanje redova ili stranica, tabela se zaključava pre zaključavanja zahtevanih redova, odnosno stranica. Tabela se u ovom pristupu zaključava na nižem nivou restrikcije u odnosu na strategiju koja zaključava samo tabelu.

U sistemu DB2 može se postaviti da se pri svakom pristupu tabeli zahtevaju katanci na tabeli na sledeći način:

```
ALTER [TABLESPACE|TABLE] naziv LOCKSIZE TABLE
```

Na slici 3.5 prikazane su neke vrste katanaca koje sistem DB2 koristi na nivou tabela (proširenje prethodno razmatranog skupa katanaca):

- IN (intent none) – vlasnik katanca može da čita sve podatke u tabeli uključujući i nepotvrđene podatke, ali ne može da ih menja. Druge konkurentne aplikacije mogu da čitaju ili da ažuriraju tabelu. Ne postavljaju se nikakvi katanci na redovima
- IS (intent share) – vlasnik katanca može da čita proizvoljan podatak u tabeli ako se S katanac može dobiti na redovima ili stranicama od interesa
- IX (intent exclusive) – vlasnik katanca može da čita ili menja proizvoljni podatak u tabeli ako se (a) X katanac može dobiti na redovima ili stranicama koje želimo da menjamo i (b) S ili U katanac može dobiti na redovima koje želimo da čitamo

IN	Intent None
IS	Intention Share
IX	Intention eXclusive
SIX	Share with Intention eXclusive
S	Share
U	Update
X	eXclusive
Z	superexclusive

Row Locking also used

Strict Table Locking

Slika 3.5: Modovi zaključavanja tabela

- SIX (share with intention exclusive) – vlasnik katanca može da čita sve podatke iz tabele i da menja redove ako može da dobije X katanac na tim redovima. Za čitanje se ne dobijaju katanci na redovima. Druge konkurentne aplikacije mogu da čitaju iz tabele. SIX katanac se dobija ako aplikacija ima IX katanac na tabeli i onda se zahteva S katanac ili obratno.
- S (share) – vlasnik katanca može da čita proizvoljan podatak u tabeli i neće dobiti katance na redovima ili stranicama
- U (update) – vlasnik katanca može da čita proizvoljan podatak u tabeli i može da menja podatke ako se na tabeli može dobiti X katanac. Pritom se ne dobijaju nikakvi katanci na redovima ili stranicama
- X (exclusive) – vlasnik katanca može da čita i da menja proizvoljan podatak iz tabele. Pritom se ne dobijaju nikakvi katanci na redovima ili stranicama
- Z (super exclusive) – ovaj katanac se zahteva na tabeli u posebnim prilikama, kao što su recimo menjanje strukture tabele ili njeno brisanje. Nijedna druga aplikacija (ni ona sa UR nivoom izolovanosti) ne može da čita niti da ažurira podatke u tabeli.

Modovi IS i IX se koriste na nivou tabela da daju podršku katanima na redovima ili stranicama. Oni omogućavaju zaključavanje na nivou redova i stranica i pritom sprečavaju još neki ekskluzivni katanac na tabeli od strane druge transakcije. Ako transakcija ima IS katanac na tabeli, ona može da dobije katanac na redu ili stranici samo za čitanje; druga transakcija može takođe da čita taj isti red ili stranicu, a može, takođe, i da menja podatak u nekom drugom redu tabele. Ako transakcija ima IX katanac, ona može da dobije katanac na redu/stranici za menjanje, a i druga transakcija može da dobije katanac za čitanje ili menjanje na drugim redovima/stranicama tabele.

Modovi S, U i X se koriste na nivou tabela da nametnu striktnu strategiju zaključavanja tabela. Za transakcije koje koriste ovaj vid zaključavanja ne

koristi se nikakvo zaključavanje na nivou redova ili stranica. Transakcija koja ima S katanac na tabeli može da čita proizvoljan podatak iz tabele; na taj način druge transakcije mogu da dobiju katance koji podržavaju zahteve samo za čitanjem nekog podatka iz tabele; nijedna druga transakcija ne može da menja nijedan podatak iz tabele sve dok se S katanac ne oslobodi. Ako transakcija ima U katanac na tabeli ona može da čita proizvoljan podatak iz tabele i eventualno da menja podatke iz tabele tako što dobija X katanac; druge transakcije mogu samo da čitaju podatke iz tabele. Ako transakcija ima X katanac na tabeli ona može da čita i menja proizvoljan podatak iz tabele, a nijedna druga transakcija ne može da pristupi nijednom podatku iz tabele, niti za čitanje niti za menjanje (osim onih sa UR nivoom izolovanosti kada je moguće da pristupi izmenjenim redovima ili stranicama). Na slici 3.6 prikazana je tabela kompatibilnosti katanaca koje je moguće dobiti na tabeli.

MODE OF LOCK A	MODE OF LOCK B							
	IN	IS	S	IX	SIX	U	X	Z
IN	YES	YES	YES	YES	YES	YES	YES	NO
IS	YES	YES	YES	YES	YES	YES	NO	NO
S	YES	YES	YES	NO	NO	YES	NO	NO
IX	YES	YES	NO	YES	NO	NO	NO	NO
SIX	YES	YES	NO	NO	NO	NO	NO	NO
U	YES	YES	YES	NO	NO	NO	NO	NO
X	YES	NO	NO	NO	NO	NO	NO	NO
Z	NO	NO	NO	NO	NO	NO	NO	NO

Slika 3.6: Kompatibilnost katanaca koji se mogu dobiti na tabeli

Na slici 3.7 je prikazan stepen konkurentnosti koji je dozvoljen u različitim strategijama zaključavanja. Dve aplikacije prikazane sa leve strane slike ne mogu konkurentno da pristupe željenoj tabeli zbog globalne prirode katanca koji ima prva aplikacija.

Striktni katanac na tabelu se može zahtevati naredbom:

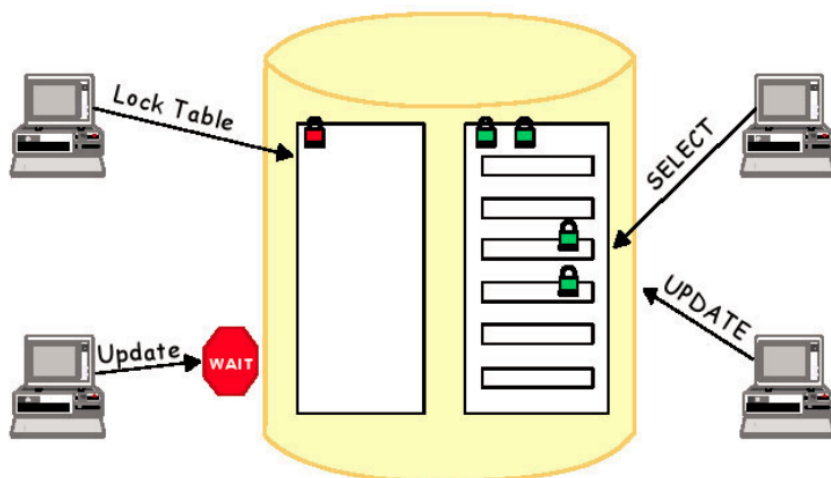
```
EXEC SQL LOCK TABLE <naziv_tabele> IN SHARE|EXCLUSIVE MODE;
```

Ako se u okviru naredbe navede ključna reč SHARE traži se S katanac, a ako se navede EXCLUSIVE traži se X katanac.

Aplikacije prikazane na desnoj strani iste slike konkurentno pristupaju ciljnoj tabeli. Strategije sa namerom koje se koriste (korišćenjem IX i IS katanaca) dozvoljavaju ovakav vid konkurentnosti. U ovoj strategiji se svaki konflikt u zaključavanju javlja na nivou reda ili stranice.

Efekti traženja katanaca

U nekim situacijama katanci koji su potrebni da bi se izvršila neka naredba jedne aplikacije su u konfliktu sa katancima koje već imaju druge konkurentne aplikacije. Takmičenje među aplikacijama za katance može da izazove čekanje



Slika 3.7: Zaključavanje tabele vs. zaključavanje redova/stranica

na katanace, isticanje maksimalnog dozvoljenog čekanja na katanac, mrtve petlje i eskalaciju katanaca.

Da bi se uslužio što veći broj aplikacija, menadžer baza podataka omogućava funkciju *ekskalacije katanaca*. Ovaj proces objedinjuje proces dobijanja katanca na tabeli i oslobađanja katanaca na redovima. Željeni efekat je da se smanje ukupni zahtevi skladištenja za katanace od strane menadžera baza podataka. Ovo će omogućiti drugim aplikacijama da dobiju željene katanace.

Dva konfiguraciona parametra baze podataka imaju direktan uticaj na proces eskalacije katanaca:

- **locklist**: to je prostor u globalnoj memoriji baze podataka koji se koristi za skladištenje katanaca,
- **maxlocks**: to je procenat ukupne liste katanaca koji je dozvoljeno da drži jedna aplikacija.

Oba parametra su konfigurabilna.

Ekskalacija katanaca se dešava u dva slučaja:

- aplikacija zahteva katanac koji bi proizveo da ona prevaziđe procenat ukupne veličine liste katanaca, koji je definisan parametrom **maxlocks**. U ovoj situaciji menadžer baze podataka će pokušati da oslobodi memorijski prostor dobijanjem jednog katanca na tabelu, a oslobađanjem katanaca na redove te tabele.
- aplikacija ne može da dobije katanac jer je lista katanaca puna. Menadžer baze podataka će pokušati da oslobodi memorijski prostor tako što će dobiti katanac na tabelu, a osloboditi katanace na redove te tabele. Primećimo da aplikacija koja pokreće eskalaciju katanaca može ali i ne mora da drži značajan broj katanaca.

Ekskalacija katanaca može da ne uspe. Ako se desi greška, aplikacija koja je pokrenula ekskalaciju dobiće vrednost SQLCODE -912. Ovaj kôd bi trebalo da bude obrađen programski od strane aplikacije.

Detekcija *prekoračenog vremena čekanja na katanac* (eng. locktimeout) je svojstvo menadžera baza podataka kojim se sprečava da aplikacija beskonačno dugo čeka na katanac.

Na primer, transakcija može da čeka na katanac koji drži aplikacija drugog korisnika, ali drugi korisnik je napustio radno mesto bez toga da je omogućio aplikaciji da uspešno završi svoj posao, čime bi se oslobodio katanac. Da bi se izbegao zastoj aplikacije u ovom slučaju, može se postaviti konfiguracioni parametar locktimeout na maksimalno vreme koje proizvoljna aplikacija može da čeka da bi dobila katanac. Postavljanje ovog parametra pomaže da se izbegnu globalne mrtve petlje. Na primer, ako aplikacija 1 pokuša da dobije katanac koji već drži aplikacija 2, aplikacija 1 dobija SQLCODE -911 (SQLSTATE 40001) with reason code 68 ako je prekoračeno vreme čekanja na katanac. Podrazumevana vrednost parametra locktimeout je -1 što označava da je ova opcija isključena.

Mrtva petlja se obrađuje procesom u pozadini koji se naziva detektor mrtve petlje. Ako se uoči mrtva petlja, određuje se "žrtva", zatim se za nju automatski poziva ROLLBACK i vraća se SQLCODE -911 with reason code 2. Poništanjem radnji žrtve oslobađaju se katanci i to bi trebalo da omogući drugim procesima da nastave sa radom.

Parametar dlchktime definiše frekvenciju sa kojom se proverava da li je došlo do mrtve petlje između aplikacija koje su povezane na bazu podataka. Ovaj parametar je konfigurabilan i vrednost mu je izražena u milisekundama. Ako je ova vrednost postavljena na neku veliku vrednost, onda se povećava vreme koje će aplikacije čekati dok se ne otkrije mrtva petlja, ali se šteti cena izvršavanja detektora mrtve petlje. S druge strane, ako se ova vrednost postavi na neku malu vrednost, onda će se mrtve petlje brzo otkrivati, ali ukupne performanse sistema mogu biti smanjene jer se često rade provere. Podrazumevana vrednost ovog parametra je 10s.

Vrednosti svih ovih parametara (i mnogih drugih) mogu se pročitati naredbom:

```
GET DATABASE CONFIGURATION FOR <naziv_baze_podataka>
```

Vrednost nekog parametra moguće je izmeniti naredbom:

```
UPDATE DATABASE CONFIGURATION FOR <naziv_baze_podataka>
    <parametar> <vrednost>
```

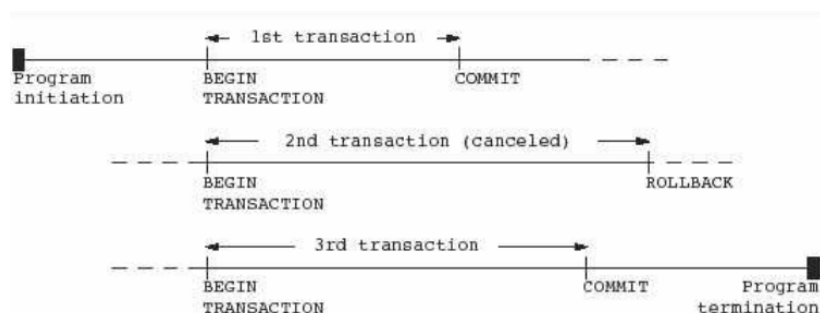
Na primer, vrednost parametra locktimeout moguće je izmeniti za bazu podataka BPKnjiga na sledeći način:

```
UPDATE DATABASE CONFIGURATION FOR BPKnjiga LOCKTIMEOUT 30
```

3.3 Svojstva transakcija i nivoi izolovanosti transakcija

Jedan aplikativni program može se sastojati od većeg broja transakcija (slika 3.8). Operacije COMMIT i ROLLBACK završavaju transakciju, ali ne prekidaju izvršavanje programa.

3.3. SVOJSTVA TRANSAKCIJA I NIVOI IZOLOVANOSTI TRANSAKCIJA 75



Slika 3.8: Ilustracija programa koji se sastoji od tri transakcije (u sistemu DB2 operacija `BEGIN TRANSACTION` se izvršava implicitno.)

Izvršavanje jedne transakcije može da se završi *planirano* ili *neplanirano*. Do planiranog završetka dolazi izvršavanjem operacije `COMMIT` kojom se transakcija uspešno završava, ili eksplicitne `ROLLBACK` operacije, koja se izvršava kada dođe do greške za koju postoji programska provera (tada se radnje te transakcije poništavaju, a program nastavlja sa radom izvršavanjem sledeće transakcije). Neplanirani završetak izvršenja transakcije događa se kada dođe do greške za koju ne postoji programska provera; tada se izvršava implicitna (sistemska) `ROLLBACK` operacija, radnje te transakcije se poništavaju a program prekida sa radom. I operacija `COMMIT` se može implicitno izvršiti, pri normalnom završetku programa.

Izvršavanjem operacije `COMMIT`, efekti svih ažuriranja te transakcije postaju trajni, odnosno više se ne mogu poništiti procedurom oporavka. U log datoteku se upisuje odgovarajući slog o kompletiranju transakcije (`COMMIT` slog), a svi katanci koje je transakcija držala nad objektima se oslobađaju. Izvršenje `COMMIT` operacije ne podrazumeva nužno fizički upis svih ažuriranih podataka u bazu (neki od njih mogu biti u baferu podataka, pri čemu se, efikasnosti radi, sa fizičkim upisom u bazu čeka dok se baferi ne napune). Činjenica da efekti ažuriranja postaju trajni znači da se može garantovati da će se upis u bazu sigurno dogoditi u nekom narednom trenutku.

Pada transakcije nastaje kada transakcija ne završi svoje izvršenje planirano. Tada sistem izvršava implicitnu – prinudnu `ROLLBACK` operaciju, tj. sprovodi aktivnost oporavka od pada transakcije.

Pojedinačne radnje ažuriranja u okviru jedne transakcije, kao i operacija početka transakcije (u sistemu DB2 se operacija `BEGIN TRANSACTION` izvršava implicitno), izvršavaju se na način koji omogućuje oporavak baze u slučaju pada transakcije. Oporavak podataka u bazi i vraćanje baze u konzistentno stanje omogućuje se upisom svih informacija o ovim radnjama i operacijama u log datoteku. Pri izvršavanju operacija ažuriranja (u užem smislu), osim što se ažurira sadržaj baze podataka, u log datoteci se beleže vrednosti svakog ažuriranog objekta pre i posle ažuriranja.

Izvršavanjem operacije `BEGIN TRANSACTION` (bilo da je eksplicitna ili implicitna) u log datoteku se upisuje slog početka transakcije.

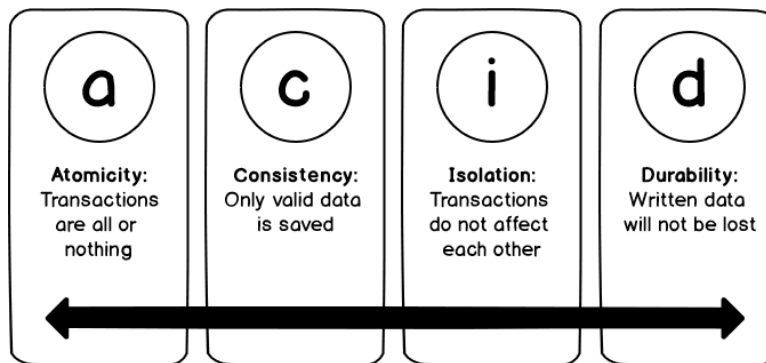
Operacija `ROLLBACK`, bilo eksplicitna ili implicitna, sastoji se od poništavanja učinjenih promena nad bazom. Izvršava se čitanjem unazad svih slogova iz log datoteke koji pripadaju toj transakciji, do sloga `BEGIN TRANSACTION`. Za

svaki slog, promena se poništava primenom odgovarajuće UNDO-operacije.

ACID svojstva transakcije

Transakcija se karakteriše sledećim važnim svojstvima (poznatim kao ACID svojstva):

- *atomičnost* (eng. Atomicity): transakcija se izvršava u celosti ili se ne izvršava ni jedna njena radnja;
- *konzistentnost* (eng. Consistency): transakcija prevodi jedno konzistentno stanje baze podataka u drugo konzistentno stanje;
- *izolacija* (eng. Isolation): efekti izvršenja jedne transakcije su nepoznati drugim transakcijama sve dok ona ne završi uspešno svoj rad;
- *trajnost* (eng. Durability): svi efekti uspešno završene transakcije su trajni, odnosno mogu se poništiti samo drugom transakcijom. Dakle, nakon potvrde transakcije, promene ostaju upisane u bazi podataka, čak i u slučaju pada sistema.



Slika 3.9: ACID svojstva transakcije

Nivoi izolovanosti

Zbog povećanja konkurentnosti, svojstvo izolovanosti transakcije (I u ACID) realizuje se u nekoliko nivoa koji se među sobom razlikuju po stepenu u kome operacije jedne transakcije mogu da utiču na izvršenje operacija konkurentnih transakcija, odnosno stepen u kome operacije drugih konkurentnih transakcija mogu da utiču na izvršavanje operacija posmatrane transakcije. Odnosno, na ovaj način se opisuje stepen ometanja koji tekuća transakcija može da podnese pri konkurentnom izvršavanju. Ove nivoe nazivamo *nivoima izolovanosti transakcije* (tj. celokupne aplikacije). Ako su transakcije serijalizabilne stepen ometanja ne postoji, tj. nivo izolovanosti je maksimalan. Realni sistemi iz različitih razloga (npr. iz razloga povećanja performansi) dopuštaju rad sa nivoima izolovanosti koji su manji od maksimalnog. Važi da što je veći nivo izolovanosti manje su dopuštene smetnje i obratno.

SQL standard definiše naredne nivoe izolovanosti (koji su navedeni u redosledu od najjačeg do najslabijeg):

3.3. SVOJSTVA TRANSAKCIJA I NIVOI IZOLOVANOSTI TRANSAKCIJA7

- SERIALIZABLE
- REPEATABLE READ
- READ COMMITTED
- READ UNCOMMITTED

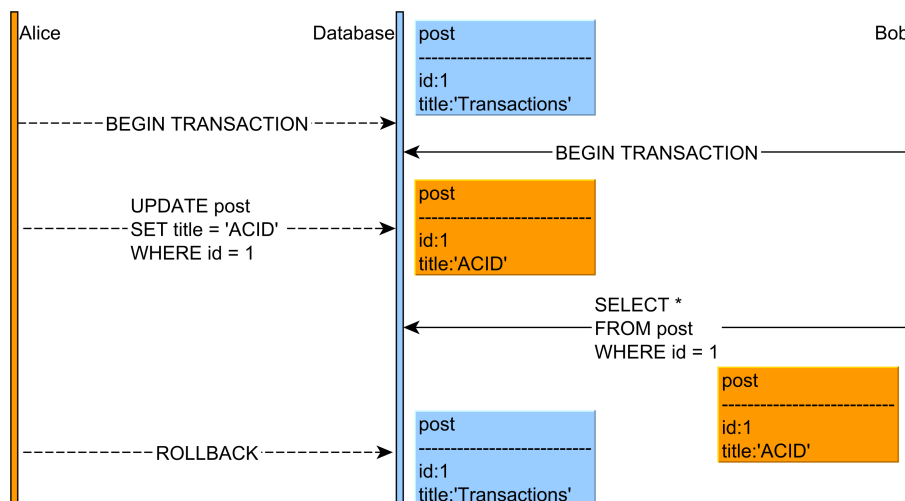
Prvi nivo izolovanosti SERIALIZABLE je najviši nivo izolovanosti i garantuje serijalizabilnost izvršenja. Standard definiše i tri načina na koja serijalizabilnost može da bude narušena, ukoliko se ne radi sa najvišim nivoom izolovanosti i to su:

- *prljavo čitanje* (eng. dirty read), odgovara pojmu zavisnosti od nepotvrđenih podataka. Na primer, transakcija *A* ažurira neki slog, nakon toga transakcija *B* čita taj slog, a zatim transakcija *A* poništi promene. Ono što se desilo jeste da je transakcija *B* pročitala slog koji ne postoji.

Primer:

1. transakcija *A* započinje sa radom
2. transakcija *A*: UPDATE Knjiga SET god_izdavanja = 1990 WHERE k.sifra=123;
3. transakcija *B* započinje sa radom
4. transakcija *B*: SELECT * FROM Knjiga;
5. transakcija *A*: ROLLBACK;

Efekat ovog niza radnji jeste da transakcija *B* vidi podatke koje je izmenila transakcija *A*, ali ove izmene još uvek nisu potvrđene.



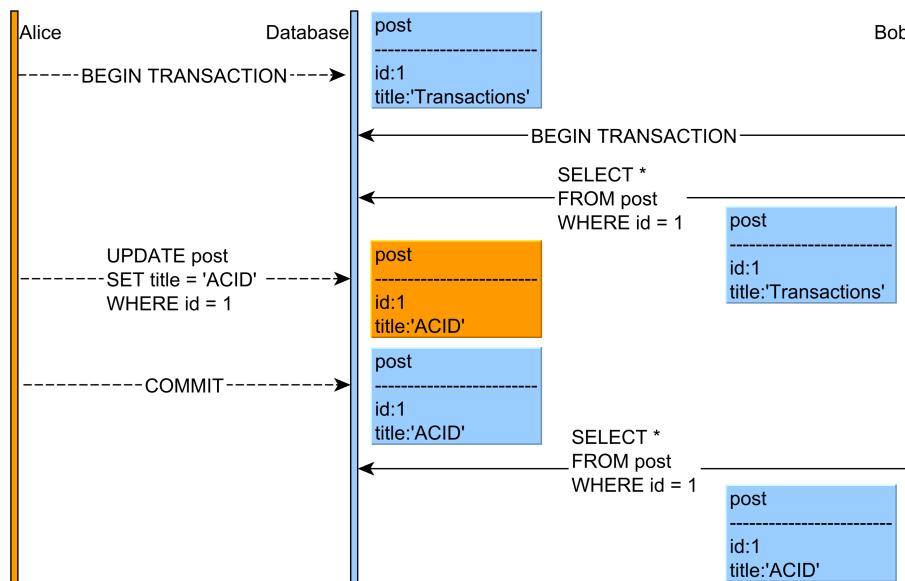
Slika 3.10: Ilustracija efekta prljavog čitanja (izvor: <https://vladmihalcea.com/dirty-read/>)

- *neponovljivo čitanje* (eng. nonrepeatable read). Recimo transakcija *A* čita slog, zatim transakcija *B* ažurira taj slog i posle toga transakcija *A* ponovo pokušava da pročita "isti" slog.

Primer:

1. transakcija *A* započinje sa radom
2. transakcija *A*: `SELECT * FROM Knjiga WHERE k_sifra = 123;`
3. transakcija *B* započinje sa radom
4. transakcija *B*: `UPDATE Knjiga SET god_izdavanja = 1990 WHERE k_sifra = 123;`
5. transakcija *B*: `COMMIT;`
6. transakcija *A*: `SELECT * FROM Knjiga WHERE k_sifra = 123;`
7. transakcija *A*: `COMMIT;`

Efekat ovog niza radnji jeste da transakcija *A* ponovnim čitanjem istih podataka dobija drugačiji rezultat.



Slika 3.11: Ilustracija efekta neponovljivog čitanja (izvor: <https://vladmihalcea.com/non-repeatable-read/>)

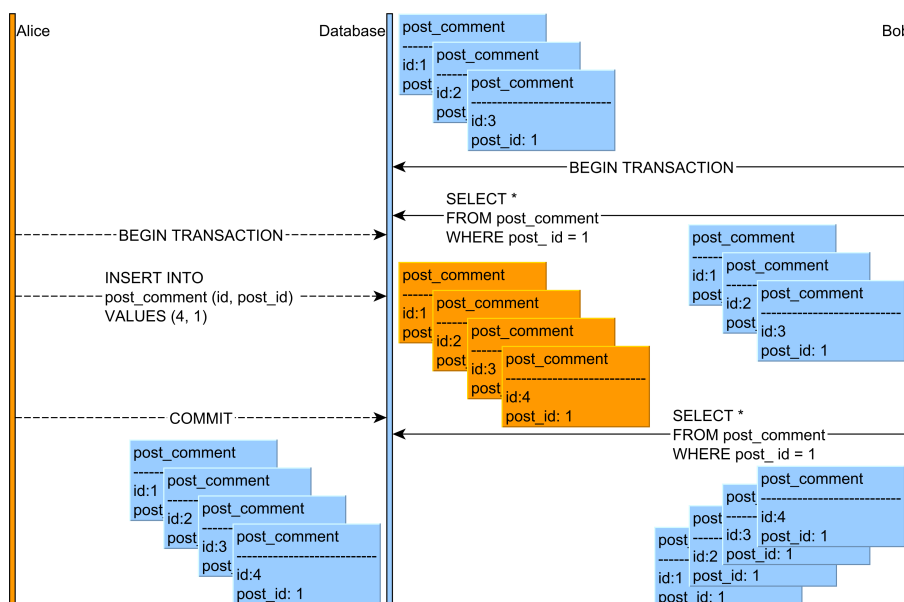
- *fantomsko čitanje* (eng. phantom read). Neka transakcija *A* čita skup slogova koji zadovoljavaju neki uslov, a zatim transakcija *B* unese novi slog koji zadovoljava isti uslov. Ako transakcija *A* sada ponovi zahtev, tj. pročita ponovo podatke po istom uslovu, videće slog koji ranije nije postojao - tzv. fantomski slog.

3.3. SVOJSTVA TRANSAKCIJA I NIVOI IZOLOVANOSTI TRANSAKCIJA

Primer:

1. transakcija *A* započinje sa radom
2. transakcija *A*: `SELECT * FROM Knjiga WHERE god.izdavanja > 1990;`
3. transakcija *B* započinje sa radom
4. transakcija *B*: `INSERT INTO Knjiga (124, 'Na Drini cuprija', 'Prosveta', 2000);`
5. transakcija *B*: `COMMIT;`
6. transakcija *A*: `SELECT * FROM Knjiga WHERE god.izdavanja > 1990;`
7. transakcija *A*: `COMMIT;`

Efekat ovog niza radnji jeste da transakcija *B* umeće novi red koji zadovoljava uslov upita kojim transakcija *A* čita podatke i pri ponovnom čitanju dobijaju se novi redovi u rezultatu koji prethodno nisu postojali.



Slika 3.12: Ilustracija efekta fantomskog čitanja (izvor: <https://vladmihalcea.com/phantom-read/>)

U tabeli 3.11 za svaki nivo izolovanosti prikazano je koji se od nabrojanih problema mogu desiti.

Nivoi izolovanosti u sistemu DB2

U sistemu DB2 postoje naredni nivoi izolovanosti (koji su navedeni u redosledu od najjačeg do najslabijeg):

- REPEATABLE READ (RR)
- READ STABILITY (RS)

Nivo izolovanosti	Priljavo čitanje	Neponovljivo čitanje	Fantomsko čitanje
SERIALIZABLE	N	N	N
REPEATABLE READ	N	N	Y
READ COMMITTED	N	Y	Y
READ UNCOMMITTED	Y	Y	Y

Tabela 3.11: Mogući problemi koji se mogu javiti pri radu sa određenim nivoom izolovanosti.

- CURSOR STABILITY (CS)
 - CURRENTLY COMMITTED (CC)
- UNCOMMITTED READ (UR)

Oni odgovaraju nivoima izolovanosti definisanim SQL standardom, ali su drugačije imenovani.

Najviši nivo izolovanosti aplikacije (i njenih transakcija) u sistemu DB2 je *nivo ponovljivog čitanja* – RR (eng. repeatable read). Ovaj nivo obezbeđuje zaključavanje svih vrsta kojima se transakcija *obraća* (npr. cele tabele), a ne samo onih vrsta koje zadovoljavaju uslov upita. Tako se rezultat čitanja transakcijom T ne može promeniti od strane druge transakcije pre nego što se transakcija T završi. Na primer, ako transakcija T sa nivoom izolovanosti RR želi da pročita knjige izdate nakon 1990. godine naredbom `SELECT * FROM Knjiga WHERE god_izdavanja > 1990`, ona zaključava sve redove tabele Knjiga (a ne npr. samo one koji zadovoljavaju uslov upita). Katanci za čitanje reda se drže do naredne naredbe COMMIT ili ROLLBACK. S druge strane, ovaj nivo izolovanosti obezbeđuje i da transakcija nema uvid u nepotvrđene promene drugih transakcija.

Sledeći nivo izolovanosti jeste *nivo stabilnosti čitanja* – RS (eng. read stability). Za razliku od prethodnog, ovaj nivo izolovanosti obezbeđuje zaključavanje samo onih vrsta koje *zadovoljavaju uslov upita*, tj. ne dopušta promenu (od strane drugih transakcija) vrste koju je pročitala transakcija T, sve do završetka transakcije T. Mada će, pri ovom nivou izolovanosti, vrsta koju je pročitala transakcija T ostati nepromenjena od strane drugih transakcija sve do završetka transakcije T, rezultati ponovljenih izvršenja istog upita od strane transakcije T mogu da se razlikuju (može da dođe do dodavanja, od strane drugih transakcija, vrsta koje će zadovoljiti taj upit, tj. do efekta fantomskih redova). Na primer, ako transakcija T sa nivoom izolovanosti RS želi da pročita knjige izdate nakon 1990. godine naredbom `SELECT * FROM Knjiga WHERE god_izdavanja > 1990`, ona zaključava samo one koji zadovoljavaju uslov upita, te druga transakcija može u međuvremenu dodati nove redove u tabelu Knjiga koji zadovoljavaju ovaj uslov i na taj način može doći do efekta fantomskog čitanja, ako transakcija T ponovo pročita redove tabele Knjiga po istom uslovu. Katanci za čitanje reda se drže do naredne naredbe COMMIT ili ROLLBACK.

Nivo stabilnosti kursora – CS (eng. cursor stability) obezbeđuje zaključavanje samo one vrste koju transakcija T trenutno čita. Sve vrste koje je transakcija T prethodno pročitala (ali ne i menjala) otključane su i druge transakcije ih mogu menjati i pre okončanja transakcije T. Na primer, ako transakcija T sa nivoom izolovanosti CS želi da pročita knjige izdate nakon 1990.

3.3. SVOJSTVA TRANSAKCIJA I NIVOI IZOLOVANOSTI TRANSAKCIJA 81

godine odgovarajućim kursorom, ona zaključava samo aktivni red kursora, ali se prelaskom na naredni red ovaj red otključava. Rad sa ovim nivoom izolovanosti može dovesti do efekta fantomskog čitanja, kao i neponovljivog čitanja jer neka druga transakcija može ažurirati neki slog koji je transakcija T ranije pročitala (i nakon toga pustila katanac na tom slogu), te se prilikom ponovnog zahteva transakcije T za čitanjem istih podataka može dobiti drugačiji rezultat. Transakcija T sa ovim nivoom izolovanosti i dalje nema uvid u promene drugih transakcija pre okončanja tih transakcija. Ovaj nivo izolovanosti je podrazumevani u sistemu DB2.

U novim verzijama DB2 (počev od verzije 9.7) podrazumeva se *semantika trenutno potvrđenih izmena* – CC (eng. currently committed). Transakcijama sa nivoom izolovanosti CS koje implementiraju ovu semantiku se čitaju samo potvrđeni podaci. Na primer, ako je transakcija T menjala neki podatak, a neka druga transakcija želi da te podatke čita, ona ne mora da čeka da transakcija T oslobodi katanac na tom redu. Umesto toga, ona može da čita podatke, ali se vraćaju vrednosti koje se zasnivaju na trenutno potvrđenim izmenama, tj. podaci koji su bili pre početka operacije ažuriranja transakcijom T.

Najslabiji nivo izolovanosti transakcija jeste *nivo nepotvrđenog čitanja* – UR (eng. uncommitted read). Transakcija T sa ovim nivoom izolovanosti čita podatke bez toga da zahteva (i dobije) deljivi katanac na tom objektu. Usled prethodnog, ovaj nivo omogućuje transakciji T da pročita nepotvrđene promene drugih transakcija, kao i drugim transakcijama da pristupe podacima koje transakcija T upravo čita. Ovaj nivo izolovanosti je dobar za pristupanje tabelama koje je moguće samo čitati (koje su definisane kao “read-only”).

X katanci se ne oslobađaju sve do naredbe COMMIT ili ROLLBACK, bez obzira na nivo izolovanosti transakcije.

Izabrani nivo izolovanosti je važeći tokom date jedinice posla. Za neku naredbu SQL-a, nivo izolovanosti se utvrđuje na sledeći način:

- za statički SQL
 - ako je za samu tu naredbu eksplicitno postavljen nivo izolovanosti, koristi se ta vrednost,
 - ako za tu naredbu nije eksplicitno postavljen nivo izolovanosti, koristi se nivo izolovanosti koji je zadat prilikom ugradnje paketa u bazu
- za dinamički SQL
 - ako je za samu tu naredbu eksplicitno postavljen nivo izolovanosti, koristi se ta vrednost,
 - ako za tu naredbu nije eksplicitno postavljen nivo izolovanosti i ako je naredbom SET CURRENT ISOLATION postavljena vrednost nivoa izolovanosti u tekućoj sesiji, koristi se ta vrednost,
 - ako za naredbu nije postavljen nivo izolovanosti, niti je naredbom SET CURRENT ISOLATION postavljena vrednost nivoa izolovanosti za tekuću sesiju, koristi se nivo izolovanosti koji je zadat prilikom ugradnje paketa u bazu.

Pojedinačnoj naredbi se nivo izolovanosti postavlja navođenjem stavke `WITH <nivo_isolovanosti>` nakon teksta naredbe, pri čemu `<nivo_isolovanosti>` ima jednu od vrednosti `RR`, `RS`, `CS`, `UR`. Ovo je moguće uraditi samo za neke naredbe kao što su recimo `DECLARE CURSOR`, `INSERT`, `SELECT`, `SELECT INTO`, itd. Na primer:

```
UPDATE Knjiga SET izdavac = 'Laguna' WHERE id_knjige = 20153 WITH RR
```

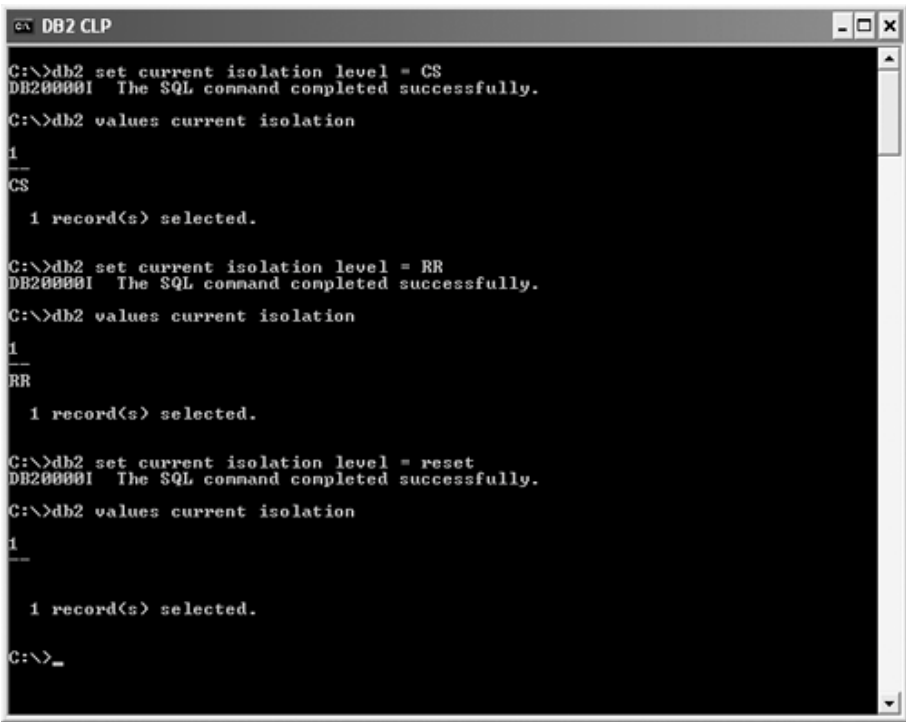
Nivo izolovanosti se može postaviti prilikom ugradnje paketa u bazu podataka na sledeći način:

```
BIND <paket> ISOLATION (RR|RS|CS|UR)
```

Za dinamičke SQL naredbe unutar jedne sesije moguće je postaviti nivo izolovanosti naredbom:

```
SET CURRENT ISOLATION = {RR|RS|CS|UR}
```

Ovaj nivo izolovanosti važi za sve dinamičke naredbe unutar date sesije, bez obzira na to kom paketu naredba pripada.



```

C:\>db2 set current isolation level = CS
DB20000I The SQL command completed successfully.
C:\>db2 values current isolation
1
--
CS
1 record(s) selected.

C:\>db2 set current isolation level = RR
DB20000I The SQL command completed successfully.
C:\>db2 values current isolation
1
--
RR
1 record(s) selected.

C:\>db2 set current isolation level = reset
DB20000I The SQL command completed successfully.
C:\>db2 values current isolation
1
--
1 record(s) selected.

C:\>_

```

Slika 3.13: Postavljanje nivoa izolovanosti za celu sesiju

U okviru DB2 CLI pristupa, nivo izolovanosti transakcije se može izmeniti putem funkcije `SQLSetConnectAttr` kojom se konekciji mogu postaviti vrednosti atributa. Naime, u ovom slučaju potrebno je atribut `SQL_ATTR_TXN_ISOLATION` postaviti na jednu od vrednosti:

3.3. SVOJSTVA TRANSAKCIJA I NIVOI IZOLOVANOSTI TRANSAKCIJA 83

- SQL_TXN_SERIALIZABLE,
- SQL_TXN_REPEATABLE_READ,
- SQL_TXN_READ_COMMITTED (podrazumevani),
- SQL_TXN_READ_UNCOMMITTED.

Na primer:

```
SQLSetConnectAttr(konekcija, SQL_ATTR_TXN_ISOLATION,  
                  (void *)SQL_TXN_READ_COMMITTED, 0);
```

U JDBC-u je moguće postaviti nivo izolovanosti transakcije pozivom metoda `setTransactionIsolation(isolation)` na objekat konekcije. Vrednost koja se prosleđuje je jedna od gore navedenih vrednosti za nivo izolovanosti. Pritom, kada se kreira konekcija u JDBC-u, ona se nalazi u modu auto-commit, odnosno svaka pojedinačna SQL naredba se razmatra kao transakcija i automatski se vrši operacija COMMIT. Stoga je pre postavljanja nivoa izolovanosti potrebno pozvati metod `setAutoCommit(false)` na objekat konekcije.

Kursori deklarisan sa opcijom WITH HOLD

U principu, izvršavanjem naredbi za kraj transakcije (COMMIT i ROLLBACK) zatvaraju se otvoreni kursori i oslobađaju katanci. S obzirom na to da je nekada zgodno da nakon uspešnog kraja transakcije otvoreni kursor i dalje ostane otvoren, moguće je navesti opciju WITH HOLD u deklaraciji kursora čime se postiže da on ostane otvoren i nakon uspešnog kraja transakcije.

Kursori u ugrađenom SQL-u mogu biti deklarisan korišćenjem opcije WITH HOLD na sledeći način:

```
DECLARE <naziv> CURSOR WITH HOLD FOR  
      <upit>
```

Na ovaj način kursor ostaje otvoren sve dok se eksplicitno ne zatvori, ili se ne izvrši naredba ROLLBACK. Prilikom izvršavanja naredbe COMMIT, otvoreni kursori koji su deklarisan sa opcijom WITH HOLD ostaju otvoreni i pridruženi katanci na tabelama se zadržavaju. Na ovaj način nakon izvršavanja naredbe COMMIT naredni red u rezultatu se može pročitati naredbom FETCH.

Dakle, izvršavanje naredbe COMMIT:

- zatvara sve kursori koji nisu definisani sa opcijom WITH HOLD,
- oslobađa sve katance osim na tekućem redu kursora koji su definisani sa opcijom WITH HOLD,
- transakcija se potvrđuje i sve promene se trajno zapisuju u bazi podataka u smislu da postaju dostupne svim konkurentnim ili narednim transakcijama.

Izvršavanje naredbe ROLLBACK:

- zatvara sve kursori,

- oslobađa sve katance,
- poništava sve promene nastale tokom transakcije.

U JDBC-u se kao argument funkcija `createStatement()` i `prepareStatement()` može navesti vrednost `HOLD_CURSORS_OVER_COMMIT` (odnosno vrednost `CLOSE_CURSORS_AT_COMMIT`) kojom se postavlja da će objekti rezultujućeg skupa ostati otvoreni (odnosno biti zatvoreni) kada transakcija uspešno završi rad pozivom metoda `commit()`.

U DB2 CLI pristupu se odgovarajući efekat može postići tako što se za datu naredbu postavi atribut `SQL_ATTR_CURSOR_HOLD` na vrednost `SQL_CURSOR_HOLD_ON` (podrazumevana vrednost) ili `SQL_CURSOR_HOLD_OFF`. Funkcija kojom se postavlja vrednosti atributa nekoj naredbi je `SQLSetStmtAttr`. Na primer:

```
SQLSetStmtAttr(naredba,
               SQL_ATTR_CURSOR_HOLD,
               (void *)SQL_CURSOR_HOLD_OFF,
               0);
```

3.4 Oporavak

1

Oporavak (eng. recovery) podrazumeva aktivnost koju sistem preduzima u slučaju da se, u toku izvršenja jedne ili više transakcija, otkrije neki razlog koji onemogućava njihov uspešan završetak. Taj razlog može biti:

- u samoj transakciji, kao što je prekoračenje neke od dozvoljenih vrednosti – *pad transakcije*,
- u sistemu, npr. prestanak električnog napajanja – *pad sistema*, ili
- u disku na kome je baza podataka – *pad medijuma*.

Kako je transakcija logička jedinica posla, njene radnje moraju ili sve uspešno da se izvrše ili nijedna, pa je u slučaju nemogućnosti uspešnog završetka transakcija neophodno poništiti efekte parcijalno izvršenih transakcija (eng. rollback). Sa druge strane, može se dogoditi da u trenutku pada sistema neki efekti uspešno kompletiranih transakcija još nisu upisani u bazu podataka (npr. ažurirani slogovi su još uvek u unutrašnjoj memoriji u baferu podataka čiji se sadržaj pri padu sistema gubi); zato može postojati potreba za ponovnim (delimičnim) izvršenjem nekih prethodno uspešno kompletiranih transakcija (eng. rollforward). To poništavanje odnosno ponovno izvršavanje radnji transakcija u cilju uspostavljanja konzistentnog stanja baze podataka predstavlja sistemsku aktivnost oporavka od pada transakcije ili sistema.

U slučaju pada medijuma, oporavak uključuje pre svega prepisivanje arhivirane kopije baze na ispravan medijum, a zatim, eventualno, ponovno izvršenje transakcija kompletiranih posle poslednjeg arhiviranja a pre pada medijuma.

Ovako koncipiran oporavak nužno se zasniva na postojanju ponovljenih (dupliranih) podataka i informacija na različitim mestima ili medijumima,

¹Materijal iz ovog poglavlja je nastao na osnovu knjige prof. Gordane Pavlović Lažetić: Baze podataka

tj. na mogućnosti rekonstrukcije informacije na osnovu druge informacije, ponovljeno smeštene na drugom mestu u sistemu. Kao “drugo” mesto u sistemu, na koje se (osim u bazu) upisuju informacije o izvršenim radnjama, obično se koristi *log datoteka* (koja se nekad naziva i sistemski log).

Naredbom `GET DATABASE CONFIGURATION` možemo dobiti informacije o lokaciji log datoteke na sistemu, kao i o njenoj maksimalnoj veličini (parametrom `LOGFILSIZ` se postavlja koliko stranica (veličine 4KB) sadrži log datoteka).

Komponenta SUBP odgovorna za oporavak baze podataka od pada transakcije, sistema ili medijuma je *upravljač oporavka*. Njegova aktivnost se sastoji iz sledećeg niza radnji:

- periodično prepisuje (eng. dump) celu bazu podataka na medijum za arhiviranje;
- pri svakoj promeni baze podataka, upisuje slog promene u log datoteku, sa tipom promene i sa:
 - novom i starom vrednošću pri ažuriranju,
 - novom vrednošću pri unošenju u bazu,
 - starom vrednošću pri brisanju iz baze;
- za svaku naredbu DDL-a, upisuje odgovarajući slog u log datoteku;
- u slučaju pada transakcije ili sistema, stanje baze podataka može biti nekonzistentno; upravljač oporavka koristi informacije iz log datoteke da poništi dejstva parcijalno izvršenih transakcija odnosno da ponovo izvrši neke kompletirane transakcije; arhivirana kopija baze podataka se u ovom slučaju ne koristi;
- u slučaju pada medijuma (npr. diska na kome je baza podataka), “najsvežija” arhivirana kopija baze podataka se prepisuje na ispravan medijum (disk), a zatim se koriste informacije iz log datoteke za ponovno izvršenje transakcija kompletiranih posle poslednjeg arhiviranja a pre pada medijuma.

Da bi se omogućilo upravljaču oporavka da poništi odnosno da ponovo izvrši transakcije, potrebno je obezbediti procedure kojima se poništavaju odnosno ponovo izvršavaju pojedinačne radnje tih transakcija kojima se menja baza podataka. Baza podataka menja se operacijama ažuriranja, unošenja ili brisanja podataka, pa pored procedura za izvršenje tih operacija, SUBP mora biti snabdeven i odgovarajućim procedurama za poništavanje odnosno ponovno izvršavanje tih operacija, na osnovu starih odnosno novih vrednosti zapamćenih u log datoteci. Drugim rečima, SUBP poseduje, osim tzv. DO logike (“uradi”), i tzv. UNDO (“poništi”) i REDO (“ponovo uradi”) logiku.

Pad sistema ili medijuma može se dogoditi i u fazi oporavka od prethodnog pada. Zato može doći do ponovnog poništavanja već poništenih radnji, odnosno do ponovnog izvršavanja već izvršenih radnji. Ova mogućnost zahteva da UNDO i REDO logika imaju svojstvo idempotentnosti, tj. da je:

$$UNDO(UNDO(x)) \equiv UNDO(x)$$

$$REDO(REDO(x)) \equiv REDO(x)$$

za svaku radnju x . U log datoteci pokazivačima su povezani slogovi koji se odnose na jednu transakciju. Pretpostavlja se da log datoteka nikada, ili veoma retko, "pada", tj. da je, zahvaljujući sopstvenom dupliranju, tripliranju, itd na raznim medijumima, ona uvek dostupna.

Oporavak od pada transakcije

Kao što smo već pomenuli, izvršenje jedne transakcije može da se završi planirano ili neplanirano. Neplanirani završetak izvršenja transakcije događa se kada dođe do greške za koju ne postoji programska provera; tada se izvršava implicitna (sistemska) ROLLBACK operacija, radnje transakcije se poništavaju a program prekida sa radom.

Oporavak podataka u bazi i vraćanje baze u konzistentno stanje omogućuje se upisom svih informacija o ovim radnjama i operacijama u log datoteku. Izvršavanjem operacije BEGIN TRANSACTION (bilo da je eksplicitna ili implicitna) u log datoteku se upisuje slog početka transakcije. Operacija ROLLBACK, bilo eksplicitna ili implicitna, sastoji se od poništavanja učinjenih promena nad bazom podataka. Izvršava se čitanjem unazad svih slogova iz log datoteke koji pripadaju toj transakciji, do sloga početka transakcije. Za svaki slog, promena se poništava primenom odgovarajuće UNDO-operacije. Aktivnost oporavka od pada transakcije ne uključuje REDO logiku.

Oporavak od pada sistema

U slučaju pada sistema, sadržaj unutrašnje memorije je izgubljen. Zato se, po ponovnom startovanju sistema, za oporavak koriste podaci iz log datoteke da bi se poništili efekti transakcija koje su bile u toku u trenutku pada sistema. Ovakvih transakcija može biti više i one se mogu identifikovati čitanjem log datoteke unazad, kao one transakcije za koje postoji BEGIN TRANSACTION slog ali ne postoji COMMIT slog.

Procedura oporavka od pada sistema kod ranijih SUBP

Da bi se smanjila količina posla potrebna za poništavanje transakcija, uvode se, u pravilnim intervalima, obično posle određenog broja upisanih slogova u log datoteku, tzv. *tačke preseka stanja* (eng. checkpoint). U momentu koji određuje tačku preseka stanja, fizički se upisuju podaci i informacije iz log bafera i bafera podataka (koji su u unutrašnjoj memoriji) u log datoteku i u bazu podataka, redom, i u log datoteku se upisuje slog tačke preseka stanja. Ovaj slog sadrži informaciju o svim aktivnim transakcijama u momentu tačke preseka stanja, adrese poslednjih slogova tih transakcija u log datoteci, a može sadržati i niz drugih informacija o stanju baze podataka u tom trenutku. Adresa sloga tačke preseka stanja upisuje se u *datoteku ponovnog startovanja* (eng. restart file).

Fizički upis u tački preseka stanja znači da će se sadržaj bafera prepisati na spoljni medijum bez obzira da li su baferi puni ili ne. To dalje znači da se fizički upis svih ažuriranih podataka uspešno završene transakcije može garantovati tek u tački preseka stanja koja sledi za uspešnim završetkom transakcije. Dolazimo do zaključka da su završetak transakcije (upis COMMIT sloga

u log datoteku) i definitivni upis svih ažuriranja te transakcije u bazu podataka – dve odvojene radnje, za koje se ne sme dogoditi da se jedna izvrši a druga ne. Mehanizam koji to obezbeđuje je *protokol upisivanja u log unapred* (eng. WAL – Write Ahead Log). Prema ovom protokolu, pri izvršenju operacije COMMIT najpre se odgovarajući slog fizički upisuje u log datoteku, pa se zatim podaci iz bafera podataka prepisuju u bazu podataka. Ako dođe do pada sistema posle upisa COMMIT sloga u log datoteku a pre nego što je sadržaj bafera podataka prepisan u bazu, taj se sadržaj može restaurirati iz log datoteke REDO logikom.

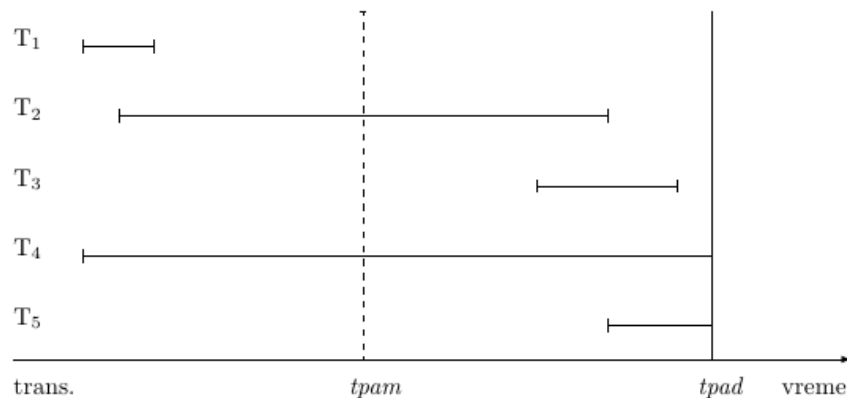


Slika 3.14: Protokol upisivanja u log unapred

Pri ponovnom startovanju sistema, posle pada sistema, moguće je prema sadržaju log datoteke identifikovati neuspele transakcije – kandidate za poništavanje, i uspele transakcije – kandidate za ponovno izvršavanje. Oporavak baze podataka tada se vrši prema protokolu koji se može opisati sledećim pseudokodom:

```
BEGIN
  naci slog poslednje tacke preseka stanja u log datoteci
    (iz datoteke ponovnog startovanja);
  IF u poslednjoj tacki preseka stanja nema aktivnih transakcija
    i slog tacke preseka stanja je poslednji slog u log datoteci,
    oporavak je završen
  ELSE
    BEGIN
      formirati dve prazne liste transakcija: uspele i neuspele;
      sve transakcije aktivne u poslednjoj tacki preseka stanja
        staviti u listu neuspelih;
      citati redom log datoteku od tacke preseka stanja do kraja:
        kada se naidje na slog '‘pocetak transakcije’',
          dodati transakciju listi neuspelih;
        kada se naidje na slog '‘kraj transakcije’',
          dodati (premestiti) tu transakciju u listu uspelih;
      citati unazad log datoteku (od kraja)
        i ponistiti akcije i efekte neuspelih transakcija;
      citati log datoteku od poslednje tacke preseka stanja unapred
        i ponovo izvršiti uspele transakcije
    END
  END.
```

Razmotrimo izvršavanje skupa transakcija koje su prikazane na slici 3.15. Na vremenskoj osi označene su dve tačke: $tpam$ – tačka preseka stanja, i $tpad$ – tačka pada sistema. Transakcija T_1 završila je rad pre poslednje tačke preseka stanja, pa su njena ažuriranja trajno upisana u bazu u momentu koji odgovara tački preseka stanja, i zato ona ne ulazi u proces oporavka. Transakcije T_2 i T_3 završile su sa radom posle momenta $tpam$ a pre pada sistema. U momentu pada sistema aktivne su transakcije T_4 i T_5 . Zbog toga prethodni protokol svrstava transakcije T_4 i T_5 u listu neuspelih (čija dejstva poništava), a transakcije T_2 i T_3 u listu uspehlih (koje zatim ponovo izvršava od tačke $tpam$). Time je završena aktivnost oporavka od pada sistema.



Slika 3.15: Pad sistema pri izvršavanju skupa transakcija

Poboljšanja procedure oporavka od pada sistema

Postoji niz poboljšanja i modifikacija izloženog postupka oporavka od pada sistema.

Jedno evidentno poboljšanje protokola oporavka od pada sistema odnosi se na aktivnosti vezane za tačku preseka stanja. Naime, prethodno opisani protokol poništava efekte neuspelih transakcija koji možda nisu ni bili ubeleženi u bazu (već samo u bafer podataka ako su se dogodili posle tačke preseka stanja), odnosno ponovo izvršava radnje uspehlih transakcija od tačke preseka stanja, iako je moguće da su efekti tih radnji, zbog popunjenosti bafera podataka, upisani u bazu podataka pre pada sistema.

Moguće je eliminisati fizičko upisivanje bafera podataka u bazu podataka u tački preseka stanja, a u fazi oporavka od pada sistema poništavati samo one radnje neuspelih transakcija čiji su efekti zaista upisani u bazu, odnosno ponovo izvršavati samo one radnje uspehlih transakcija čiji efekti nisu upisani u bazu. U tom cilju uvodi se, u svaki slog log datoteke, u vreme njegovog upisa u log datoteku, jedinstvena oznaka, tzv. *serijski broj u logu* (eng. LSN – Log Sequence Number). Serijski brojevi su u rastućem poretku. Osim toga, kadgod se ažurirana stranica fizički upisuje u bazu podataka, u stranicu se upisuje i LSN sloga log datoteke koji odgovara tom ažuriranju. U log datoteci može biti više slogova koji odgovaraju ažuriranju iste stranice baze podataka. Ako je serijski broj upisan u stranicu P veći ili jednak od serijskog broja sloga

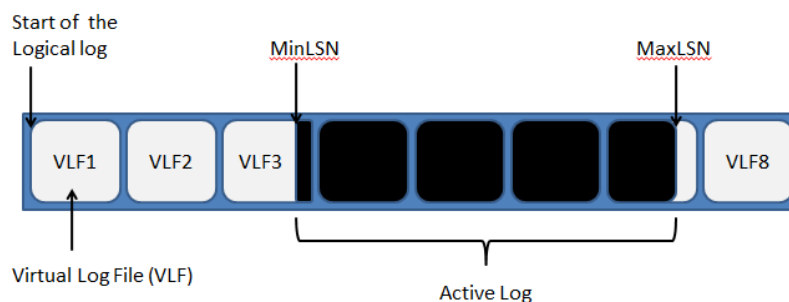
log datoteke, efekat ažuriranja kome odgovara taj slog fizički je upisan u bazu; ako je manji, efekat nije upisan.

Da bi se slog iz baze podataka ažurirao, potrebno je pročitati (iz baze podataka) stranicu na kojoj se slog nalazi. Posle ažuriranja sloga, stranica je (u baferu podataka) spremna za upis u bazu, pri čemu i dalje nosi serijski broj svog prethodnog upisa u bazu.

Tačnije, za slog log datoteke uspele transakcije gde stranica baze podataka ima LSN vrednost veću ili jednaku od LSN vrednosti sloga log datoteke, onda ništa ne treba raditi. Efekat ovog sloga log datoteke je već trajno sačuvan na stranici na disku. Za slog log datoteke uspele transakcije u situaciji kada stranica baze podataka ima LSN vrednost manju od LSN vrednosti sloga log datoteke, potrebno je ponovo izvršiti da bi se osiguralo da efekti transakcije budu trajno sačuvani.

Za slog log datoteke neuspele transakcije gde stranica baze podataka ima LSN vrednost veću ili jednaku od LSN vrednosti tog sloga log datoteke, potrebno je poništiti efekte tog sloga da bi se obezbedilo da efekti transakcije ne budu trajno sačuvani. Za slog log datoteke neuspele transakcije u situaciji kada stranica baze podataka ima LSN vrednost manju od LSN vrednosti sloga log datoteke, nije potrebno ništa raditi. Efekat ovog sloga nije trajno sačuvan na stranici na disku i kao takav nije ga potrebno poništavati.

Sada se u proceduru registrovanja tačke preseka stanja, osim eliminisanja fizičkog upisa bafera podataka u bazu, može dodati upis, u slog tačke preseka stanja, vrednosti LSN m najstarije stranice bafera podataka, tj. najmanjeg LSN stranicâ iz bafera podataka, koje su ažurirane ali još nisu upisane u bazu. Slog log datoteke sa serijskim brojem m odgovara tački u log datoteci od koje treba, umesto od tačke preseka stanja, pri oporavku od pada sistema ponovo izvršavati uspele transakcije. Kako tačka m prethodi tački preseka stanja, može se desiti da neka transakcija koja je uspešno kompletirana pre tačke preseka stanja, "upadne" u skup aktivnih (uspelih) transakcija (transakcija T_1 , slika 3.17). Na takvu transakciju primeniće se procedura oporavka, mada se to ne bi dogodilo u prethodnoj varijanti oporavka; to je "cena" smanjenog broja poništavanja i smanjenog broja ponovnog izvršavanja i fizičkog upisa koje obezbeđuje ovaj postupak.



Slika 3.16: Log datoteka – LSN m najstarije stranice bafera podataka

Procedura oporavka od pada sistema sada ima sledeći izgled:

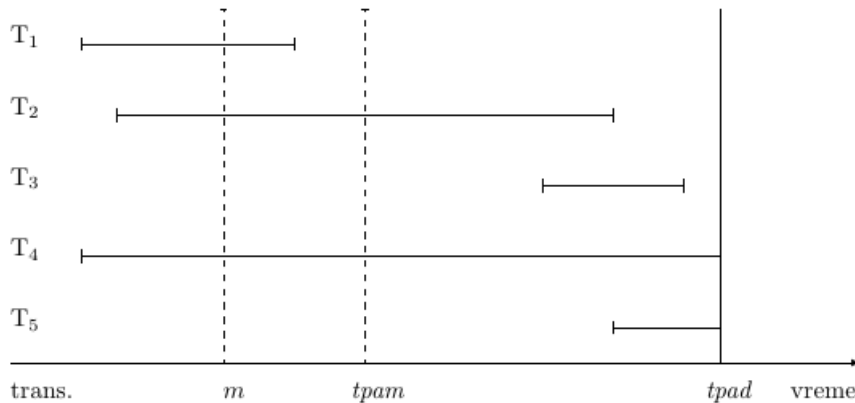
BEGIN

formirati dve prazne liste transakcija: uspele i neuspele;

```

sve transakcije aktivne u tacki m staviti u listu neuspelih;
citati redom log datoteku od tacke m do kraja:
    kada se naidje na slog 'pocetak transakcije',
        dodati transakciju listi neuspelih;
    kada se naidje na slog 'kraj transakcije',
        dodati (premestiti) tu transakciju u listu uspelih;
citati unazad log datoteku (od kraja)
    i ponistiti efekte onih radnji neuspelih transakcija
    za koje je  $p \geq r$ , gde je  $r$  - LSN tekuceg sloga
    a  $p$  - LSN odgovarajuće stranice baze podataka;
citati log datoteku od tacke m unapred
    i ponovo izvršiti one radnje uspelih transakcija
    za koje je  $p < r$  ( $p$ ,  $r$  - kao u prethodnom koraku)
END.

```



Slika 3.17: Oporavak od pada sistema korišćenjem serijskih brojeva

Moguće je i sva upisivanja radnje jedne transakcije u bazu podataka ostaviti za trenutak izvršenja COMMIT operacije te transakcije, čime se eliminiše potreba za UNDO logikom.

Novi SUBP imaju podršku za *tačke pamćenja* (eng. savepoint) koje omogućavaju da ako tokom izvršavanja dođe do greške da se ne poništavaju izmene kompletne transakcije, već samo izmene koje su se desile tokom trajanja transakcije od trenutka kada je tačka pamćenja počela do trenutka u kom je zahtevano poništavanje efekata. Na ovaj način omogućeno je da se jedna velika transakcija podeli u nekoliko delova ("podtransakcija") tako da svaki od njih ima svoju definisanu tačku pamćenja.

Na ovim idejama zasnovan je i algoritam oporavka ARIES (Algorithm for Recovery and Isolation Exploiting Semantics) koji koriste razni SUBP, između ostalih i SUBP DB2.

SQL podrška za tačke pamćenja

Tačka pamćenja unutar neke transakcije se pravi naredbom:

SAVEPOINT <naziv>

Radi povećanja jasnoće kôda predlaže se da se tačkama pamćenja daju informativna imena. Dodatne opcije koje se mogu zadati prilikom pravljenja tačke pamćenja na kraju naredbe su:

- UNIQUE - isti naziv tačke pamćenja se neće javiti ponovo sve dok je tačka pamćenja aktivna,
- ON ROLLBACK RETAIN CURSORS - kad god je moguće operacija ROLLBACK TO SAVEPOINT neće uticati na otvorene kursore,
- ON ROLLBACK RETAIN LOCKS - katanci koji su dobijeni nakon tačke pamćenja se ne oslobađaju nakon operacije ROLLBACK TO SAVEPOINT.

Ukoliko neka tačka pamćenja više nije potrebna, ona se može osloboditi. Oslobađanje tačke pamćenja vrši se naredbom:

RELEASE SAVEPOINT <naziv>

Ako se tačka pamćenja eksplicitno ne oslobodi, ona će biti oslobođena na kraju transakcije.

Da bi se izvršio povratak na neku tačku pamćenja potrebno je pozvati naredbu:

ROLLBACK TO SAVEPOINT [<naziv>]

Ovom naredbom se sve izmene nad bazom podataka koje su vršene nakon te tačke pamćenja poništavaju. Ukoliko se ne navede naziv tačke pamćenja izvršava se povratak na poslednju aktivnu tačku pamćenja. Sve tačke pamćenja koje su postavljene nakon te tačke pamćenja se oslobađaju, dok se tačka pamćenja na koju je izvršen povratak ne oslobađa.

Na primer, možemo imati niz operacija oblika:

```
...  
SAVEPOINT s1 ON ROLLBACK RETAIN CURSORS;  
INSERT ...  
INSERT ...  
IF ...  
    ROLLBACK TO SAVEPOINT s1;  
SAVEPOINT s2 ON ROLLBACK RETAIN CURSORS;  
UPDATE ...  
DELETE ...  
COMMIT;
```

Ukoliko je tačka pamćenja već oslobođena, nije moguće izvršiti povratak na nju naredbom ROLLBACK TO SAVEPOINT.

U sistemu DB2 dozvoljeno je imati naredbe DDL-a unutar tačke pamćenja. Ako aplikacija uspešno oslobodi tačku pamćenja (naredbom RELEASE), ona može da nastavi da koristi SQL objekte koji su kreirani naredbama DDL-a unutar te tačke pamćenja. Ipak, ako je aplikacija izvršila naredbu ROLLBACK TO SAVEPOINT za tačku pamćenja koja izvršava naredbe DDL-a, DB2 označava neispravnim sve kursore koji su zavisni od ovih naredbi.

Glava 4

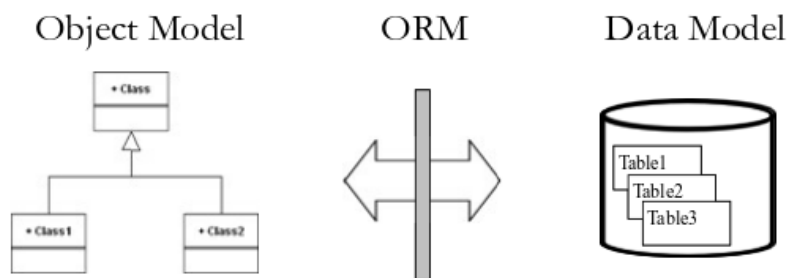
Objektno-relaciono preslikavanje i alat Hibernate

Danas se za većinu poslovnih primena koriste relacione baze podataka. U pogledu programskih jezika, objektno-orijentisano programiranje je postalo standard: jezici kao što su Java, C#, C++, pa čak i objektno-orijentisani skript jezici postaju uobičajen izbor za razvoj aplikacija. U objektno-orijentisanim aplikacijama, manipulacija podacima se izvršava nad objektima koji gotovo nikada nisu skalarne vrednosti. Međutim, mnogi popularni proizvodi baza podataka kao što su sistemi za upravljanje SQL bazama podataka mogu da čuvaju i obrađuju samo skalarne vrednosti kao što su celi brojevi i stringovi koji su organizovani unutar tabela. Na programeru je da ili konvertuje vrednosti objekta u grupe jednostavnijih vrednosti koje je moguće sačuvati u bazi (i da ih konvertuje nazad kada mu budu potrebni) ili da se u programu ograniči na korišćenje jednostavnih skalarnih vrednosti. Većina aplikacija koja koristi relacione baze podataka i objektno-orijentisane jezike se svodi na pisanje koda kojim se relacioni model preslikava na objektno-orijentisani model.

Prilikom dizajniranja aplikacija koje koriste baze podataka za skladištenje podataka, potrebno je obezbediti mogućnost preuzimanja podataka sa višeg sloja hijerarhije aplikacije i njihovo smeštanje u bazu podataka i obrnuto – preuzimanje podataka iz baze podataka i njihovo prosleđivanje višem sloju aplikacije. Direktna komunikacija sa bazom podataka ima svoje nedostatke, kao što su: zavisnost aplikacije od konkretnog SUBP-a, zavisnost aplikacije od izmena u bazi podataka, itd. Dakle, poželjno je definisati neki vid preslikavanja objekata iz objektno-orijentisanih aplikacija u svet relacionih baza podataka. Za ove potrebe je moguće koristiti i nerelaciona rešenja, kao što su objektno baze podataka dizajnirane za rad sa objektno-orijentisanim vrednostima. Korišćenje objektno-orijentisanih sistema za upravljanje bazama podataka eliminiše potrebu da se podaci konvertuju u svoju SQL formu i obratno. Ipak, za većinu aplikacija se podrazumeva da se podaci čuvaju u relacionim bazama podataka. Jedan od razloga za to je to što se velika količina informacija već čuva u relacionim bazama podataka i ako aplikacija želi da im pristupa mora biti u stanju da čita i piše u relacionom formatu. Često je razlog za korišćenje relacionih baza podataka i taj što se baza podataka koristi i deli između više različitih aplikacija (od kojih neke ne moraju biti objektno-orijentisane).

Rešenje ovog problema pružaju alati za *objektno-relaciono preslikavanje* (eng. Object Relation Mapping - ORM) koji vrše automatsko preslikavanje relacionog modela baze podataka na objektni model aplikacije i obratno. Oni omogućavaju da objektno-orijentisane aplikacije ne rade direktno sa tabelarnom reprezentacijom podataka već da imaju svoj objektno-orijentisani model. Iako je i dalje neophodno preslikati relacioni model na objektno-orijentisani model podataka, preslikavanje je na neki način odvojeno od logike aplikacije koja se razvija. Jednom kada se napravi odgovarajuće preslikavanje za datu klasu, instance ove klase se mogu koristiti svuda u samoj aplikaciji. Objektno-relaciono preslikavanje ćemo ilustrovati na primeru alata *Hibernate* koji se koristi za objektno-relaciono preslikavanje u programskom jeziku Java i koji je do danas stekao veliku popularnost među Java programerima.

Kao što je i očekivano, prilikom pokušaja preslikavanja jedne paradigme u drugu, javljaju se različiti problemi i u tom slučaju govorimo o objektno-relacionom neslaganju. U ovom poglavlju biće izloženi ovi problemi i nakon toga biće prikazano kako se korišćenjem alata *Hibernate* možemo izboriti sa njima.



Slika 4.1: Objektno relaciono preslikavanje

4.1 Objektno-relaciono preslikavanje

Trajnost ili *perzistentnost podataka* (eng. data persistence) jedan je od osnovnih koncepata koji se podrazumevaju prilikom razvoja aplikacije. Ukoliko informacioni sistem nije sačuvao podatke prilikom isključivanja, on teško može biti od neke praktične koristi. Trajnost objekata podrazumeva da pojedinačni objekti mogu da nadžive izvršavanje aplikacije koja ih je kreirala – objekti se mogu sačuvati u skladištu podataka i možemo im iznova pristupiti u nekom kasnijem trenutku. Nije neophodno trajno sačuvati sve objekte, ali je potrebno trajno sačuvati objekte koji su od ključne važnosti za poslovnu logiku aplikacije. Kada se govori o pojmu trajnosti u programskom jeziku Java, obično se misli na preslikavanje i čuvanje instanci objekata u bazi podataka korišćenjem SQL-a.

Relaciona tehnologija je danas najzastupljenija zbog svojih dobrih osobina, između ostalog zbog svog fleksibilnog i robusnog pristupa upravljanju podacima. Zbog veoma dobro proučenih teorijskih osnova relacionog modela podataka, relacione baze podataka garantuju i čuvaju integritet podataka.

Naravno, relacioni sistemi za upravljanje bazama podataka nisu specifični za programski jezik Java, niti je SQL baza podataka specifična za neku određenu aplikaciju. Ovaj važan princip je poznat kao *nezavisnost podataka*. Drugim rečima, podaci žive duže od svake aplikacije, a relaciona tehnologija daje mogućnost deljenja podataka između različitih aplikacija ili među različitim delovima istog sistema. Kaže se i da je relaciona tehnologija “zajednički delilac” mnogih različitih sistema i platformi.

Korišćenje SQL-a u programskom jeziku Java

Kada radimo sa SQL bazom podataka u Java aplikaciji, pozivamo SQL naredbe nad bazom podataka korišćenjem JDBC API-a. JDBC API se koristi za vezivanje argumenata prilikom pripreme parametara upita, za izvršavanja upita, kretanje kroz rezultat upita, izdvajanje vrednosti iz rezultujućeg skupa itd. Svi ovi zadaci kojima se pristupa podacima su niskog nivoa. Ono što bi bilo poželjno jeste napisati kôd koji čuva i izdvaja instance naših klasa, oslobađajući nas na taj način napornog posla niskog nivoa. S obzirom na to da navedeni zadaci pristupa podacima mogu biti mukotrpniji, pitanje je da li je relacioni model podataka i posebno SQL pravi izbor za obezbeđivanje trajnosti u objektno-orijentisanim aplikacijama. Pokazuje se da u većini situacija to jeste dobar izbor.

Objektni svet je napravljen sa ciljem rešavanja specifičnih problema kao što su rezervacija leta, transfer novca sa jednog na drugi račun, kupovina knjige, itd. Tehnička rešenja ovih problema se dizajniraju imajući na umu *model domena* (eng. domain model). Model domena definiše objekte koji predstavljaju probleme iz realnog sveta i kojima se objedinjuju ponašanje i podaci. Rezervacija leta, bankovni račun ili potraga za knjigom su primeri objekata iz domena. Za ove domenske objekte se očekuje da budu sačuvani u bazi podataka i naknadno izdvojeni iz baze podataka kada bude bilo potrebe. Mnoge aplikacije iz realnog sveta ne bi mogle da postoje bez neke forme trajnosti. Zamislimo da smo dobili informaciju od banke da su izgubili naš novac jer nisu sačuvali naš depozit u trajnom prostoru za skladištenje. Takođe, bili bismo veoma ljuti na aviokompaniju ako bismo kupili kartu a zatim otkrili da nemamo rezervaciju za let na željeni dan.

U okviru ovog poglavlja, razmatraćemo probleme čuvanja podataka i njihovog deljenja u kontekstu objektno-orijentisane aplikacije koja koristi model domena. Umesto da se direktno radi sa redovima i kolonama rezultujućeg skupa `java.sql.ResultSet`, poslovna logika aplikacije rukuje objektno-orijentisanim modelom domena koji je specifičan za aplikaciju. Npr, ako SQL shema baze podataka sadrži tabelu `Knjiga`, Java aplikacija definiše klasu `Knjiga`. Umesto da čita i piše vrednosti pojedinačnog reda i kolone, aplikacija učitava i čuva instance klase `Knjiga`.

Naravno, nisu sve aplikacije u programskom jeziku Java dizajnirane na ovaj način, niti bi trebalo da budu. Jednostavne aplikacije mogu raditi mnogo bolje bez modela domena. Treba koristiti JDBC `ResultSet` ako je to sve što nam je potrebno. Ali u slučaju aplikacija sa netrivialnom poslovnom logikom, pristup modela domena pomaže da se značajno popravi ponovno korišćenje kôda i njegovo održavanje.

Objektno-relaciono neslaganje

Već desetinama godina, programeri govore o neskladu koji postoji između različitih paradigmi. Ovaj nesklad objašnjava zašto se u svaki projekat ulaže dosta napora u brigu o trajnosti podataka.

Dok objektno-orijentisana i relaciona paradigma dele neke zajedničke karakteristike (atributi objekta su konceptualno slični kolonama entiteta), postoje i razlike koje uzrokuju da neprimetna integracija ove dve paradigme nije jednostavno. Osnovna razlika je ta da model podataka prikazuje podatke (kroz kolone vrednosti), dok objektni model sakriva podatke (enkapsulirajući ih iza javnih interfejsa).

Kombinacija objektno i relacione tehnologije može da dovede do konceptualnih i tehničkih problema. Ti problemi su poznati pod nazivom *objektno-relaciono neslaganje* (eng. object-relational impedance mismatch). Među najvažnije probleme spadaju:

- neslaganje između nivoa granularnosti u objektno-orijentisanom modelu i odgovarajućem relacionom modelu;
- neslaganje hijerarhijskog uređenja podataka – ne postoji mogućnost izgradnje hijerarhije između tabela u relacionoj bazi podataka, dok se klase mogu organizovati hijerarhijski;
- neslaganje pojma istovetnosti – npr. dva reda tabele koji sadrže iste podatke smatraju se istim, te iz tog razloga nije dozvoljeno unošenje dva ista reda u tabelu, dok objekti koji sadrže iste podatke mogu biti različiti jer imaju različite memorijske adrese;
- neslaganje u načinima na koji se realizuju veze;
- neslaganje u načinu na koji se pristupa podacima u programskom jeziku Java i u relacionim bazama podataka.

Svaki od ovih problema ilustrovaćemo na jednom primeru.

Pretpostavimo da je zadatak da se dizajnira i implementira aplikacija za onlajn elektronsku trgovinu. U ovoj aplikaciji potrebna je klasa kojom se predstavljaju informacije o korisniku sistema i druga klasa za predstavljanje informacija o računima korisnika, kao što je prikazano na slici 4.2.



Slika 4.2: Jednostavni UML dijagram entiteta Korisnik i Račun

Sa ovog dijagrama jasno se očitava da jedan korisnik može da ima više računa, dok je svaki račun vezan za tačno jednog korisnika. Iz svake klase može se pristupiti drugom kraju veze. Ovo se realizuje tako što se svakoj od ove dve klase dodaje odgovarajuća instanca druge klase. Klase kojima se predstavljaju ovi entiteti mogu biti vrlo jednostavne:


```
public class Korisnik{

    String korisnicko_ime;
    String adresa;
    Set<Racun> racuni;
    // metode za pristup (get/set), itd.

}

public class Racun{

    String racun;
    String naziv_banke;
    Korisnik korisnik;
    // metode za pristup (get/set), itd.

}
```

U ovom slučaju nije teško doći do dizajna odgovarajuće sheme SQL baze podataka:

```
create table Korisnici (
    korisnicko_ime varchar(15) not null primary key,
    adresa varchar(255) not null
);

create table Racun (
    racun varchar(15) not null primary key,
    naziv_banke varchar(255) not null,
    korisnicko_ime varchar(15) not null,
    foreign key (korisnicko_ime) references Korisnici
);
```

Kolona `korisnicko_ime` u tabeli `Racun` sa koje pravimo strani ključ predstavlja odnos između dva entiteta. Za ovaj jednostavan model domena, objektno-relaciono neslaganje se skoro i ne primećuje. Pravolinijski je napisati odgovarajuću JDBC aplikaciju za umetanje, ažuriranje i brisanje informacija o korisnicima i računima.

Međutim, neslaganje ove dve paradigme biće vidljivije kada se aplikaciji doda još entiteta i odnosa između entiteta. Ilustrovaćemo sada sve pomenute vidove neslaganja na ovom primeru.

Problem granularnosti

U prethodnom primeru `adresa` je bila vrednost tipa `String`. U mnogim primenama potrebno je posebno sačuvati informaciju o ulici, gradu i poštanskom kodu. Moguće je, naravno, dodati sve ove podatke direktno u klasu `Korisnik` ali s obzirom na to da je moguće da druge klase takođe čuvaju informaciju o adresi, ima više smisla napraviti zasebnu klasu `Adresa`. Na slici 4.3 prikazan je dopunjeni model.



Slika 4.3: Model u kome korisnik ima informaciju o adresi

Da li bi, takođe, trebalo dodati i zasebnu tabelu Adresa? To nije neophodno, već je uobičajeno čuvati informaciju o adresi u tabeli Korisnici u posebnim kolonama. U ovom scenariju nije potrebno spajati tabele ako je potrebno dobiti korisnika i adresu u jednom upitu. Možda bi najbolje rešenje bilo da se napravi novi SQL tip podataka za predstavljanje adrese i da se tabeli Korisnici doda jedna kolona novodefinisanog tipa umesto više njih. Dakle, imamo izbor da tabeli Korisnici dodamo više kolona ili jednu kolonu (novog SQL tipa podataka). Ovde se očito javlja *problem granularnosti*. Granularnost se odnosi na relativne veličine tipova sa kojima radimo. Vratimo se prethodnom primeru i dodajmo tabeli Korisnici kolonu novodefinisanog SQL tipa address:

```

create table Korisnici (
    korisnicko_ime varchar(15) not null primary key,
    adresa address not null
);

```

Novouvedena klasa Adresa u Java aplikaciji i novi SQL tip podataka address bi trebalo da omoguće kompatibilnost. Međutim, ukoliko proverimo kakva je podrška za korisnički definisane tipove podataka u današnjim sistemima za upravljanje bazama podataka susrećemo se sa mnogim problemima, stoga korišćenje korisnički definisanih tipova nije česta praksa u industrijskim aplikacijama. Pragmatično rešenje ovog problema se sastoji u dodavanju nekoliko kolona koje su osnovnih SQL tipova podataka:

```

create table Korisnici (
    korisnicko_ime varchar(15) not null primary key,
    adresa_ulica varchar(255) not null,
    adresa_kod varchar(5) not null,
    adresa_grad varchar(255) not null
);

```

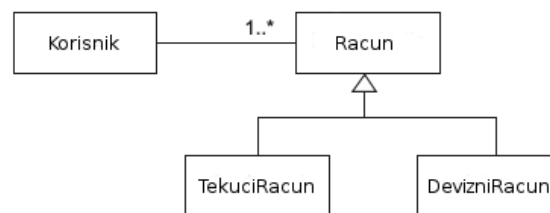
Nekad će objektni model imati veći broj klasa nego što je broj odgovarajućih tabela u bazi podataka (kažemo da je objektni model granularniji od relacionog modela).

Kada razmotrimo model domena, možemo uočiti razliku između klasa: neke od klasa deluju važnije od drugih, predstavljajući objekte prve klase. Primer je klasa Korisnik: ona odgovara entitetu u realnom svetu koji želimo da predstavimo. Neki drugi tipovi pak mogu delovati manje važno. Na primer, postoje jednostavne klase kao što bi recimo bila klasa PostanskiKod koja nasleđuje klasu ApstraktniPostanskiKod.

S druge strane, u SQL bazama podataka postoje samo dva nivoa granularnosti: nivo relacija koje mi kreiramo, kao što su recimo relacije Korisnici i Racun i ugrađeni tipovi podataka kao što su VARCHAR, BIGINT, ...

Problem podtipova

U programskom jeziku Java se nasleđivanje implementira korišćenjem mehanizma natklasa i potklasa. Dodajmo prethodnom primeru mogućnost da aplikacija može da prihvati ne samo naplatu sa računa u banci, već i sa kreditnih i debitnih kartica. Prirodan način da se ova promena izvede jeste korišćenjem natklase *Racun*, zajedno sa nekoliko konkretnih potklasa: *TekuciRacun*, *DevizniRacun*, ... Svaka od ovih potklasa definiše neznatno izmenjene podatke (i potpuno drugačije funkcionalnosti koje se izvode na tim podacima). UML klasni dijagram prikazan na slici 4.4 ilustruje ovaj model.



Slika 4.4: Korišćenje mehanizma nasleđivanja za različite strategije plaćanja

Koje izmene u bazi podataka treba izvesti da bi podržali izmenjene Java klasne strukture? SQL baza podataka u opštem slučaju ne implementira nasleđivanje tabela (čak ni nasleđivanje tipova podataka). Takođe, čim se uvede pojam nasleđivanja, postoji mogućnost polimorfizma. Klasa *Korisnik* ima definisanu vezu ka natklasi *Racun*. Ovo je polimorfna veza što znači da u vreme izvršavanja instanca klase *Korisnik* može da referiše na instancu bilo koje potklase klase *Racun*. Slično, želimo da omogućimo da pišemo *polimorfne upite* koji bi referisali na klasu *Racun*, a kojim bi se vratile i instance svih potklasa te klase.

SQL bazama podataka nedostaje način za predstavljanje polimorfne veze. Ograničenje stranog ključa se odnosi na tačno jednu ciljnu tabelu i nije pravolinijski definisati strani ključ koji referiše na više tabela. U te svrhe morali bismo napisati proceduralno ograničenje koje bi nametalo ovu vrstu pravila integriteta.

Neophodno je da struktura nasleđivanja u modelu bude očuvana u SQL bazi podataka koja ne nudi mehanizam nasleđivanja.

Problem istovetnosti

Na problem istovetnosti se nailazi u situaciji kada je potrebno proveriti da li su dve instance identične. Naime, postoje tri načina na koja se ovaj problem može adresirati: dva u kontekstu programskog jezika Java i jedan u kontekstu SQL baze podataka. U programskom jeziku Java definišu se dva različita pojma istovetnosti:

- identičnost instanci, koja se odnosi na ekvivalentnost memorijskih lokacija koja se proverava uslovom $a == b$;
- jednakost instanci, koja se utvrđuje implementacijom metoda `equals()` – na ovu vrstu jednakosti se često referiše kao na *jednakost prema vrednosti*.

S druge strane, istovetnost redova baze podataka se izražava kao poređenje vrednosti primarnih ključeva tih redova. Međutim, niti korišćenje metoda `equals()` niti operatora `==` nije uvek ekvivalentno poređenju vrednosti primarnog ključa. Uobičajeno je da nekoliko neidentičnih instanci u programskom jeziku Java istovremeno predstavljaju isti red u bazi podataka – na primer, u nitima aplikacije koje se konkurentno izvršavaju.

Razmotrimo još jedan problem koji se javlja u vezi sa pojmom istovetnosti u bazama podataka. Kolona `korisnicko_ime` je primarni ključ tabele `Korisnik`. Nažalost, ova odluka otežava promenu korisničkog imena; potrebno je ažurirati ne samo odgovarajući red u tabeli `Korisnik`, već i vrednosti stranog ključa u (potencijalno mnogim) redovima tabele `Racun`. Da bi se rešio ovaj problem, često se savetuje korišćenje *surogat ključeva*. Pod surogat ključem podrazumevamo kolonu koja čini primarni ključ i koja nema nikakvo značenje samom korisniku aplikacije – drugim rečima ključ koji se ne prikazuje korisniku aplikacije. Njegova jedina svrha jeste identifikacija podataka unutar aplikacije. Na primer, možemo izmeniti definicije tabela na sledeći način:

```
create table Korisnici(
    ID bigint not null primary key,
    korisnicko_ime varchar(15) not null unique,
    ...
);

create table Racun(
    ID bigint not null primary key,
    racun varchar(15) not null,
    naziv_banke varchar(255) not null,
    ID_korisnika bigint not null,
    foreign key (ID_korisnika) references Korisnici
);
```

Kolone `ID` sadrže vrednosti koje su sistemski generisane i koje su uvedene isključivo zbog pojednostavljenja modela podataka, tako da ostaje pitanje kako (ako uopšte) treba da budu prikazane u modelu domena programskog jezika Java.

Problemi asocijacija (veza)

U modelu domena, asocijacije (odnosno veze) predstavljaju odnose između entiteta. Klase `Korisnik`, `Adresa` i `Racun` su međusobno povezane. Ali za razliku od klase `Adresa`, klasa `Racun` je samostalna: instance klase `Racun` se čuvaju u svojoj vlastitoj tabeli. Preslikavanje veza i upravljanje vezama između entiteta su centralni koncepti u svakom rešenju problema obezbeđivanja trajnosti objekata.

Objektno-orijentisani jezici predstavljaju veze korišćenjem referenci na objekte; u relacionom svetu, veza se predstavlja kolonom stranog ključa koja sadrži kopiju vrednosti primarnog ključa. Postoje značajne razlike između ova dva mehanizma. Reference na objekte su suštinski usmerene; veza “ide” od jedne instance ka drugoj. One su pokazivači. Ako bi veza između instanci trebalo da bude dvosmerna, veza bi se morala definisati dva puta, po jednom u svakoj od klasa koje treba povezati.

```
public class Korisnik {
    Set<Racun> racuni;
}
```

```
public class Racun {
    Korisnik korisnik;
}
```

Jednosmerna veza u relacionom modelu podataka nema smisla jer je moguće napraviti proizvoljnu vezu između podataka korišćenjem operatora spajanja i operatora projekcije.

Veze u programskom jeziku Java mogu da budu tipa više ka više. Na primer, prethodne klase mogu imati naredni oblik:

```
public class Korisnik {
    Set<Racun> racuni;
}
```

```
public class Racun {
    Set<Korisnik> korisnici;
}
```

Deklaracija stranog ključa u tabeli Racun predstavlja vezu tipa “više ka jedan”: svaki račun u banci vezan je za jednog korisnika, dok svaki korisnik može da poseduje više bankovnih računa. Ako želimo da u SQL bazama podataka predstavimo vezu tipa “više ka više” potrebno je uvesti novu tabelu, koju najčešće nazivamo *veznom tabelom*. U većini slučajeva ova tabela se ne javlja nigde u modelu domena.

Za ovaj primer, ako bismo razmatrali da je veza između korisnika i računa tipa “više ka više”, definisali bismo veznu tabelu na sledeći način:

```
create table Korisnik_Racun (
    ID_korisnika bigint,
    ID_racuna bigint,
    primary key (ID_korisnika, ID_racuna),
    foreign key (ID_korisnika) references Korisnici,
    foreign key (ID_racuna) references Racun
);
```

U ovoj situaciji u tabeli Racun više nije potrebno imati kolonu stranog ključa i odgovarajuće ograničenje stranog ključa ; ova dodatna tabela sada kontroliše vezu između dva entiteta.

Problem navigacije podataka

Podacima se u programskom jeziku Java pristupa na drugačiji način u odnosu na relacione baze podataka. Informacijama o računima nekog korisnika se u programskom jeziku Java može pristupiti pozivom oblika:

```
nekiKorisnik.getRacun().iterator().next()
```

Ovo je najprirodniji način da se pristupi podacima u objektno-orijentisanoj paradigmi. Na ovaj način kreće se od jedne instance ka drugoj, prateći pripremljene pokazivače između klasa. Nažalost, ovo nije efikasan način za dobijanje podataka iz SQL baze podataka.

Jedan od načina na koji se pristup podacima može učiniti efikasnijim jeste da se minimizuje broj zahteva nad bazom podataka. Očigledan način da se to uradi jeste da se minimizuje broj SQL upita. Stoga se za efikasan pristup relacionim podacima putem SQL-a obično zahteva spajanje tabela od interesa. Broj tabela koje su uključene u spajanje određuje dubinu mreže objekata kojom se krećemo po memoriji. Na primer, ako hoćemo da izdvojimo korisnika, a nisu nam od značaja informacije vezane za korisnikove račune, možemo željene informacije dobiti korišćenjem narednog upita:

```
select * from Korisnici k
where k.ID = 123
```

S druge strane, ako je potrebno izdvojiti korisnika, a onda naknadno posetiti svaku od njemu pridruženih instanci klase *Racun* (recimo da izlistamo sve bankovne račune korisnika) pisali bismo drugačiji upit:

```
select * from Korisnici k
  left outer join Racun r
    on r.ID_korisnika = k.ID
where k.ID = 123
```

Kao što možemo videti, da bismo efikasno koristili operaciju spajanja potrebno je znati kom delu mreže objekta planiramo da pristupamo. Ukoliko izdvojimo previše podataka, rasipamo memoriju na nivou aplikacije.

SQL bazu podataka možemo, takođe, opteretiti i velikim dekartovskim proizvodima. Zamislimo da u jednom upitu ne izdvajamo samo korisnike i bankovne račune, već takođe i sve narudžbine plaćene sa svakog od računa i sve proizvode u svakoj od narudžbina itd.

Svako rešenje koje uključuje trajnost objekata obezbeđuje funkcionalnost za izdvajanje podataka iz povezanih instanci tek kada se toj vezi prvo pristupi u Java programskom kodu. Na primer ako korisnik ima više računa, mi ne želimo da prilikom preuzimanja informacija o korisniku preuzmемо i sve informacije o svim njegovim računima. Ovaj efekat je poznat kao *lenjo učitavanje* (eng. lazy loading, on-demand loading), odnosno preuzimanje podataka (tj. materijalizacija slogova baze podataka u objekte programa) vrši se samo na zahtev, u trenutku kada su oni potrebni (kada im se pristupi u programskom kodu).

Ovakav stil pristupanja jednom po jednom podatku je neefikasan u kontekstu SQL baza podataka, jer zahteva izvršavanje jedne naredbe za svaki od čvorova ili kolekciju objekata mreže kojoj se pristupa. Ovo neslaganje u načinu na koji se pristupa podacima u programskom jeziku Java i relacionim bazama podataka je možda i najčešći uzrok problema vezanih za performanse informacionih sistema implementiranih u programskom jeziku Java.

Alat Hibernate obezbeđuje sofisticirane metode za efikasno i transparentno dohvaćanje mreža objekata iz baze podataka do aplikacije koja im pristupa.

Objektno-relaciono preslikavanje i Java Persistence API

U suštini, objektno-relaciono preslikavanje (ORP) je automatsko (i transparentno) trajno čuvanje objekata iz Java aplikacije u tabelama SQL baze podataka, korišćenjem metapodataka koji opisuju preslikavanja između klasa aplikacije i sheme SQL baze podataka. Ovo preslikavanje funkcioniše tako što transformiše podatke iz jedne reprezentacije u drugu.

Hibernate je jedan od alata kojima se može izvesti objektno-relaciono preslikavanje u programskom jeziku Java. Razmotrimo neke prednosti korišćenja alata Hibernate:

- produktivnost – korišćenjem alata Hibernate programer se oslobađa većine zamornog posla niskog nivoa i omogućava mu se da se skoncentriše na poslovnu logiku problema. Bez obzira na to koja se strategija razvoja aplikacije koristi: odozgo-naniže (počev od modela domena) ili odozdo-naviše (počev od postojeće sheme baze podataka), upotreba alata Hibernate sa drugim odgovarajućim alatima značajno smanjuje vreme razvoja.
- pogodnost održavanja – automatsko izvršavanje objektno-relacionog preslikavanja smanjuje broj linija koda, čineći na taj način sistem razumljivijim i jednostavnijim za refaktorisanje. Alat Hibernate predstavlja “bafer” između modela domena i SQL sheme, izolujući na taj način svaki od ovih modela od manjih izmena drugog.
- performanse – iako ručno kodiranje trajnosti može biti brže u istom smislu kao što je kôd u assembleru brži od Java programskog kôda, automatska rešenja kao što je Hibernate omogućavaju korišćenje raznih vidova optimizacije. Ovo znači da programeri mogu da potroše više energije na ručno optimizovanje nekoliko preostalih uskih grla, umesto da prerano sve optimizuju.
- nezavisnost od proizvođača – alat Hibernate može da doprinese smanjivanju rizika koji su povezani sa situacijom da korisnik postane zavisan od proizvođača nekog proizvoda ili usluge, bez mogućnosti da koristi usluge drugog proizvođača, bez značajnijih troškova. Čak iako se nikada ne planira izmena SUBP proizvoda, ORP alati koji podržavaju različite SUBP-ove omogućavaju određeni nivo portabilnosti. Dodatno, nezavisnost od SUBP-a pomaže u scenariju gde se tokom razvoja koristi jednostavna lokalna baza, a testiranje i produkcija rade na drugačijem sistemu.

Hibernate pristup obezbeđivanju trajnosti podataka je dobro primljen od strane Java programera i napravljen je standardizovani *Java Persistence API (JPA)*. JPA specifikacijom se definiše:

- mogućnost zadavanja metapodataka preslikavanja – kako se trajne klase i njihova svojstva odnose na shemu baze podataka. JPA se u velikoj meri oslanja na Java anotacije u klasama modela domena, ali se, takođe, preslikavanja mogu zadavati korišćenjem XML datoteka,

- API za izvođenje osnovnih *CRUD operacija* (Create, Read, Update, Delete) na instancama trajnih klasa, pre svega `javax.persistence.EntityManager` za čuvanje i učitavanje podataka,
- jezik i API za zadavanje upita koji se odnose na klase i svojstva klasa. Ovaj jezik naziva se *Java Persistence Query Language (JPQL)* i nalik je SQL-u. Standardizovani API omogućava programsko kreiranje kriterijuma upita bez manipulacije nad stringovima.
- kako mehanizam trajnosti interaguje sa mehanizmom transakcija.

Alat Hibernate implementira JPA i pruža podršku za sva standardna preslikavanja, upite i programske interfeje.

Dakle, objektno-relaciono preslikavanje je nastalo kao trenutno najbolje rešenje problema objektno-relacionog neslaganja. Na primer, mreža objekata ne može biti sačuvana u tabeli baze podataka: ona mora biti rastavljena na kolone portabilnih SQL tipova podataka. Zadatak objektno-relacionog preslikavanja je da programera oslobodi od 95% posla obezbeđivanja trajnosti objekata, kao što su pisanje složenih SQL upita koji uključuju mnoga spajanja tabela i kopiranje vrednosti iz JDBC rezultujućeg skupa u objekte ili grafove objekata.

4.2 Hibernate

Razmotrimo Java aplikaciju `SkladisteKnjiga` čiji je zadatak da sačuva knjigu u bazi podataka i da traženu knjigu pronađe i izdvoji iz baze podataka. U ove svrhe koristićemo bazu podataka koja sadrži tabelu `Knjige`, prikazanu na slici 4.1.

k.sifra	naziv	izdavac	god_izdavanja
105	Na Drini Cuprija	Prosveta	1978
107	Prokleta Avlija	Nova knjiga	1990
203	Beograd veciti grad	Skordisk	2010
207	Norveska suma	Geopoetika	2000

Tabela 4.1: Tabela Knjige

Svaki red tabele `Knjige` predstavljaće jednu instancu klase `Knjiga` u aplikaciji `SkladisteKnjiga`. Želimo da u korisničkoj klasi omogućimo pozivanje metoda `sacuvajKnjigu()` kojim se omogućava čuvanje date knjige. Da bismo postigli ovaj cilj, definisaćemo klasu `Knjiga` i za svaku knjigu koju treba da bude sačuvana u bazi podataka, napravićemo novi objekat ove klase.

```
public class Knjiga{
    private int id_knjige = 0;
    private String naziv = null;
    private String izdavac = null;
    private int god_izdavanja = 0;
    // get/set metode
    ...
}
```



```
}
```

Ono što je dalje potrebno jeste omogućiti da se ovaj objekat trajno sačuva u tabeli Knjige baze podataka, preslikavanjem objektnog modela (objekta tipa Knjiga) u relacioni model (red tabele Knjige). Kreirajmo klasu KnjigaTrajno koja treba da uradi ovaj posao:

```
public class KnjigaTrajno {
    public void sacuvajKnjigu(Knjiga knjiga){
        // ovde ide kod kojim se knjiga trajno cuva
    }
    public Knjiga izdvoji(String naziv) {
        // ovde ide kod kojim se knjiga izdvaja na osnovu naziva
    }
    ...
}
```

Iako metodama još uvek nismo dodali potrebnu funkcionalnost, pogledajmo kako je moguće trajno sačuvati knjigu korišćenjem klase KnjigaTrajno:

```
KnjigaTrajno knjigaTrajno = new KnjigaTrajno();
Knjiga knjiga = new Knjiga();
knjiga.setId_knjige(101);
knjiga.setNaziv("Norveska suma");
knjiga.setIzdavac("Geopoetika");
knjiga.setGod_izdavanja(2007);
knjigaTrajno.sacuvajKnjigu(knjiga);
```

Da bismo metodama `sacuvajKnjigu` i `izdvoji` dodali potrebnu funkcionalnost, neophodno je da se najpre povežemo na bazu podataka, a zatim nam je potreban mehanizam kojim se objekat konvertuje u red tabele (kao i preslikavanje polja objekata u kolone tabele). Naravno, mogli bismo da sami osmislimo ovakav mehanizam konverzije, međutim ovo je moguće jednostavno uraditi i korišćenjem okruženja Hibernate na sledeći način:

```
public class SkladisteKnjiga{
    private void sacuvajKnjigu(Knjiga knjiga) {
        Session sesija = fabrika.openSession();
        ...
        sesija.save(knjiga);
        ...
        sesija.close();
    }
    ...
}
```

Primetimo da se instanca `knjiga` klase `Knjiga` trajno skladišti u bazi podataka izvršavanjem jedne jedine linije koda: `sesija.save(knjiga)`. Ovo je moguće zato što API klase okruženja Hibernate sadrži metode za jednostavno manipulisanje Java objektima. Naravno, pre toga potrebno je alatu Hibernate dostaviti konfiguraciju konekcije na bazu podataka, kao i preslikavanje kojim se zadaju naše namere.

Korišćenje Hibernate okruženja

Standardni koraci koje izvršavamo prilikom pravljenja Hibernate aplikacije su:

1. konfigurisanje konekcije na bazu podataka,
2. kreiranje definicija preslikavanja,
3. trajno čuvanje klasa.

U nastavku je dat spisak uobičajenih koraka koje treba primeniti u razvoju Hibernate verzije Java aplikacije *SkladisteKnjiga*:

1. kreirati objekat sa domenom *Knjige*,
2. konfigurisati okruženje Hibernate,
3. kreirati klijentsku aplikaciju koja izvršava razne operacije nad knjigama (dodaje/ažurira/briše/traži knjigu).

Glavni zadatak Hibernate aplikacije jeste njeno konfigurisanje. Postoje dva dela konfiguracije koja se zahtevaju u svakoj Hibernate aplikaciji: jedan deo je zadužen za konfigurisanje konekcije na bazu podataka, a drugim se zadaju preslikavanja objekata na tabele: njime se definiše koja se polja objekata preslikavaju u koje kolone tabele.

Konfigurisanje konekcije na bazu podataka

Da bi se povezali na bazu podataka, Hibernate mora da zna detalje o bazi podataka, tabelama, klasama itd. Ove informacije se mogu proslediti posredstvom XML datoteke (koja se podrazumevano naziva `hibernate.cfg.xml`) ili u vidu jednostavne tekstualne datoteke koja sadrži parove oblika `ime/vrednost` (čiji je podrazumevani naziv `hibernate.properties`). Ukoliko se koriste podrazumevani nazivi datoteka, okruženje će automatski učitati ove datoteke. Moguće je koristiti i drugačija imena datoteka od ovih podrazumevanih, ali je onda neophodno eksplicitno zadati naziv odgovarajuće datoteke kao argument funkcije za konfigurisanje (što ćemo videti malo kasnije).

U nastavku je prikazano kako je moguće ove informacije proslediti putem odgovarajuće XML datoteke.

```
<hibernate-configuration>
  <session-factory>
    <property name = "dialect">
      org.hibernate.dialect.DB2Dialect
    </property>
    <property name = "connection.driver_class">
      com.ibm.db2.jcc.DB2Driver
    </property>
    <!-- Pretpostavimo da je ime baze podataka test -->
    <property name = "connection.url">
      jdbc:db2://localhost:50001/test
    </property>
```

```

    <!-- Pretpostavimo da je korisnicko ime korisnicko_ime,
         a sifra korisnik_sifra -->
    <property name = "connection.username">
        korisnicko_ime
    </property>
    <property name = "connection.password">
        korisnik_sifra
    </property>
    <!-- Ovdje ide lista XML datoteka koje sadrže preslikavanja -->
    <mapping resource = "Knjiga.hbm.xml" />
    <mapping resource = "Biblioteka.hbm.xml" />
</session-factory>
</hibernate-configuration>

```

Element `<property>` služi za zadavanje vrednosti pojedinačnih svojstava, kao što su SUBP koji se koristi, naziv baze podataka i slično.

Ove informacije je moguće zadati i putem tekstualne datoteke koja sadrži parove oblika `ime = vrednost`. Na primer, jedan deo datoteke `hibernate.properties` bi izgledao ovako:

```

hibernate.dialect = org.hibernate.dialect.DB2Dialect
hibernate.connection.driver_class = com.ibm.db2.jcc.DB2Driver
hibernate.connection.url = jdbc:db2://localhost:50001/test

```

Kao što možemo da primetimo ako informacije o bazi podataka zadajemo na ovaj način sva navedena svojstva imaju prefiks "hibernate".

U okviru ovih konfiguracionih datoteka, svojstvo `connection.url` ukazuje na URL adresu baze podataka na koju se treba povezati, `driver_class` na relevantnu klasu drajvera potrebnu za pravljenje konekcije, a `dialect` na dijalekt baze podataka koji koristimo čime se automatski postavljaju neka svojstva tog dijalekta (u okviru kursa izučava se DB2 dijalekt).

Pored konfiguracionih svojstava potrebno je uključiti i datoteke koje sadrže preslikavanja i njihove lokacije. Ova preslikavanja se zadaju u posebnim datotekama sa sufiksom `.hbm.xml`. Njih je u konfiguracionoj datoteci potrebno navesti kao vrednost atributa `resource` elementa `mapping` (u prikazanom primeru, datoteka `Knjiga.hbm.xml` sadrži detalje o tome kako se objekat `Knjiga` preslikava na tabelu `Knjige`). Ono što je važno napomenuti jeste da ako se konfiguracija prosleđuje putem datoteke `hibernate.properties`, tada nije moguće direktno zadati preslikavanja klasa (za šta se ovde koristi element `<mapping>`), već je ovaj pristup moguće koristiti kada programski konfiguriramo okruženje.

Oba razmatrana metoda (konfigurisanje konekcije na bazu podataka putem XML datoteke ili putem tekstualne datoteke koja sadrži parove svojstvo/vrednost) su deklarativna. Hibernate, takođe, podržava i programsko konfigurisanje, prikazano narednim fragmentom koda:

```

Configuration konf = new Configuration();
konf.setProperty("hibernate.dialect",
    "org.hibernate.dialect.DB2Dialect");
konf.setProperty("hibernate.connection.username", korisnicko_ime);
konf.setProperty("hibernate.connection.password", korisnik_sifra);

```

```
konf.setProperty("hibernate.connection.url",
    "jdbc:db2://localhost:50001/test");
```

Alternativno, moguće je programski konfigurirati okruženje Hibernate korišćenjem odgovarajuće konfiguracione datoteke:

```
Configuration konf = new Configuration();
konf.configure("hibernate.cfg.xml");
```

Kao argument funkcije `configure()` moguće je navesti putanju do konfiguracione datoteke snimljene pod proizvoljnim imenom.

Sesija i transakcija

Hibernate okruženje na osnovu prosleđenih informacija o konfiguraciji pravi *kreitora sesija (fabriku sesija)* `SessionFactory`. To je klasa koja omogućava kreiranje sesija i nju može na bezbedan način koristiti više niti u isto vreme (eng. *thread-safe*). U idealnom slučaju potrebno je kreirati jednog kreitora sesija i deliti ga među aplikacijama. Međutim, možemo primetiti da je on definisan za jednu i samo jednu bazu podataka. Stoga, ako imamo potrebu za još jednom bazom podataka potrebno je definisati drugu konfiguraciju u drugoj datoteci (npr `hibernate2.hbm.xml`) da bismo kreirali posebnog kreitora sesija za tu bazu podataka. Kao što i sam naziv kaže, cilj kreitora sesije je kreiranje objekata sesija. Sesija predstavlja “prolaz” do baze podataka. Zadatak sesije jeste da vodi računa o svim operacijama nad bazom podataka kao što su čuvanje, učitavanje i izdvajanje podataka iz relevantnih tabela. Ona predstavlja i posrednika u rukovođenju transakcijama u okviru aplikacije. Operacije koje uključuju pristup bazi podataka su “upakovane” u jedinstvenu celinu posla – jednu transakciju. Sve operacije u okviru transakcije se ili uspešno završavaju ili se poništavaju.

Pomenimo da objekat tipa `Session` nije bezbedno koristiti od strane više niti u isto vreme (kažemo i da nije *thread-safe*). Stoga objekat sesije ne bi trebalo da bude otvoren neki duži vremenski period.

Sve operacije nad bazom podataka se izvršavaju u okviru sesija. Da bismo mogli da radimo sa bazom podataka, potrebno je da otvorimo novu sesiju, što možemo uraditi pozivom `fabrika.openSession()`, gde je `fabrika` objekat tipa `SessionFactory`. Kada završimo sa radom, potrebno je da sesiju zatvorimo pozivom metoda `close()` nad objektom klase `Session`.

Kada imamo objekat sesije možemo izvoditi operacije nad bazom podataka u okviru transakcije. Sesija i transakcija idu “ruku pod ruku”. Transakcije se u Hibernate okruženju predstavljaju klasom `Transaction`. Transakcija se započinje pozivom metoda `beginTransaction()` nad objektom sesije čime se kreira novi objekat transakcije i vraća referenca na njega. Transakcija traje sve dok se ona ne potvrdi ili poništi. U trenutku potvrđivanja transakcije, objekti se trajno pamte u bazi podataka. Ukoliko pritom dođe do greške, biće izbačen izuzetak koji treba hvatati i potrebno je poništiti transakciju.

Hibernate aplikacija se može izvršavati u kontrolisanom i u nekontrolisanom okruženju. Na osnovu ovog razlikujemo dva pristupa radu sa transakcijama:

- kontrolisano okruženje: CMT transakcije (eng. *Container Managed Transactions*) kod kojih kontejner može da kreira i upravlja radom aplikacije u svojim transakcijama,

- nekontrolisano okruženje: JDBC transakcije kod kojih mi možemo da upravljamo mehanizmom transakcija.

U prvom slučaju nema potrebe da brinemo o semantici transakcija kao što je potvrđivanje i poništavanje transakcija – o ovome se stara sam kontejner. Ovim se smanjuje broj redova koda koji pišemo.

U drugom slučaju, potrebno je kodirati početak transakcije i njen uspešan završetak (commit) ili neuspešan završetak (rollback). Mi ćemo se u nastavku oslanjati na drugi od pomenutih mehanizama. Uobičajeni scenario u ovom pristupu ima sledeću formu:

```
private void sacuvajKnjigu() {  
    // otvaramo sesiju  
    Session sesija = fabrika.openSession();  
  
    Transaction tr = null;  
    try {  
        // kreiramo instancu transakcije  
        tr = sesija.beginTransaction();  
        // dodajemo zeljenu funkcionalnost  
        ...  
        // uspesan kraj  
        tr.commit();  
    } catch (HibernateException he) {  
        // imamo problem, ponistavamo transakciju  
        if (tr != null)  
            tr.rollback();  
        throw he;  
    } finally{  
        // zatvaramo transakciju  
        sesija.close();  
    }  
}
```

Transakciju započinjemo pozivom metoda `sesija.beginTransaction()`, koji kreira novi objekat transakcije i vraća referencu na njega. Ona je povezana sa sesijom i otvorena sve dok transakcija ne potvrdi svoje izmene ili ne bude poništena. Nakon toga izvodimo zahtevani zadatak unutar transakcije, a zatim zahtevamo potvrdu ove transakcije. U ovom momentu se entiteti trajno skladište u bazi podataka. Ukoliko tokom procesa pamćenja ovih podataka dođe do neke greške, biće izbačen `HibernateException` izuzetak. Potrebno je taj izuzetak uhvatiti i poništiti transakciju. Moguće je izbaciti dodatne izuzetke nazad klijentu u informativne shrhe ili u svrhe debugovanja.

Preslikavanje klasa na tabele

Postoje dva načina na koja je moguće zadati preslikavanje trajnih klasa na tabele baze podataka:

1. korišćenjem XML datoteka preslikavanja
2. korišćenjem anotacija

XML datoteke preslikavanja

Najjednostavniji pristup za preslikavanje trajnih klasa na tabele baze podataka je onaj u kome se preslikavanje vrši van kôda, korišćenjem datoteka preslikavanja. Datoteke preslikavanja su u XML formatu i imaju odgovarajuću strukturu.

Prikažimo sadržaj datoteke `Knjiga.hbm.xml` kojom se zadaje preslikavanje objekta `Knjiga` na tabelu `Knjige`:

```
<hibernate-mapping>
  <class name = "Knjiga" table = "Knjige">
    <id name = "id_knjige" column = "k_sifra">
      <generator class = "native"/>
    </id>
    <property name = "naziv" column = "naziv"/>
    <property name = "izdavac" column = "izdavac"/>
    <property name = "god_izdavanja" column = "god_izdavanja"/>
  </class>
</hibernate-mapping>
```

Element `<hibernate-mapping>` sadrži definicije preslikavanja klasa na tabele. Pojedinačna preslikavanja klasa zadaju se elementom `<class>`. Atribut `name` etikete `<class>` referiše na domensku klasu `Knjiga`, dok atribut `table` referiše na tabelu `Knjige` u kojoj se čuvaju objekti. U okviru elementa `<class>` može se naći veći broj elemenata `<property>` kojima se zadaju preslikavanja polja objekta na kolone tabele (npr. polje `id_knjige` objekta se preslikava u kolonu `k_sifra` tabele). Svaki objekat mora imati *jedinstveni identifikator* – nalik primarnom ključu tabele i on se preslikava u primarni ključ odgovarajuće tabele. Jedinstveni identifikator postavlja se korišćenjem etikete `<id>` kojoj treba postaviti vrednosti atributa `name` – polje objekta i `column` – kolona tabele koja predstavlja primarni ključ. U okviru elementa `<id>` potrebno je navesti element `<generator>` kojim se zadaje strategija kojom se generišu ovi identifikatori. Atribut `class` elementa `<generator>` određuje strategiju generisanja identifikatora. Neke od mogućih vrednosti su:

- `increment` – vrednost kolone celobrojnog tipa se automatski inkrementira; ako tabela ne sadrži podatke, onda se počinje od vrednosti 1, a ako već sadrži podatke razmatra se maksimalna postojeća vrednost identifikatora i vrši se uvećavanje te vrednosti za 1;
- `identity` – vrednost kolone celobrojnog tipa se automatski inkrementira na osnovu vrednosti koju odredi baza podataka;
- `sequence` – vrednost kolone celobrojnog tipa se postavlja tako što se konsultuje baza podataka za narednu vrednost odgovarajućeg niza; moguće je postaviti korak inkrementiranja; prilikom traženja naredne vrednosti, moguće je dohvatiti više njih i zapamtiti ih i imati ih za naredne operacije `INSERT`, te je stoga ova strategija efikasnija od `identity` strategije;
- `native` – bira se `identity` ili `sequence` strategija u zavisnosti od toga šta baza podataka podržava;

- `assigned` – ne vrši se automatska dodela vrednosti, već je potrebno da aplikacija postavi jedinstveni identifikator; ovo je podrazumevana strategija ukoliko se ne navede `element generator`.

Svaki od navedenih naziva generatora predstavlja skraćenicu određene klase. Na primer, `assigned` je skraćenicu za klasu `org.hibernate.id.Assigned`.

Generatorima možemo zadati potrebne argumente korišćenjem elemenata `param`. Na primer, prilikom korišćenja strtegije `sequence`, možemo zadati niz prema kome se vrši postavljanje naredne vrednosti na sledeći način:

```
<id name = "id_knjige" column = "k_sifra">
  <generator class = "sequence">
    <param name = "sequence">Knjige_niz</param>
  </generator>
</id>
```

Pretpostavka je da je ovaj niz kreiran u bazi podataka naredbom:

```
create sequence Knjige_niz start with 5 increment by 3;
```

Definicije preslikavanja ukazuju na to koji će objekti biti trajno sačuvani i u kojoj tabeli. Podrazumevano se sva nestatička nekonstantna polja entiteta trajno čuvaju u odgovarajućoj tabeli. Element `<property>` služi za definisanje kolona u kojima će biti sačuvana polja (atributi, svojstva) objekta: na primer, polje `izdavac` objekta `Knjiga` biće sačuvano u koloni `izdavac` tabele `Knjige`. Ovaj element je samozatvarajući, odnosno zadaje se etiketom oblika `< ... />`. Ukoliko je ime kolone isto kao i ime polja objekta, nije neophodno navoditi atribut `column`. Stoga bi prethodni primer mogao da ima naredni oblik:

```
<hibernate-mapping>
  <class name = "Knjiga" table = "Knjige">
    <id name = "id_knjige" column = "k_sifra">
      <generator class = "native"/>
    </id>
    <property name = "naziv"/>
    <property name = "izdavac"/>
    <property name = "god_izdavanja"/>
  </class>
</hibernate-mapping>
```

U prethodnom fragmentu kôda, nismo navodili kog su tipa atributi. Na koji način Hibernate treba da zna da je atribut `naziv` tipa `String`, a atribut `god_izdavanja` celobrojnog tipa? U opštem slučaju potrebno je eksplicitno navesti tipove, korišćenjem elementa `type`, na sledeći način:

```
<hibernate-mapping>
  <class name = "Knjiga" table = "Knjige">
    ...
    <property name = "naziv" type = "string"/>
    <property name = "izdavac" type = "string"/>
```

```
        <property name = "god_izdavanja" type = "integer"/>
    </class>
</hibernate-mapping>
```

Primitimo da tipovi koji se navode u datoteci preslikavanja nisu Java tipovi, niti SQL tipovi, već su to tipovi koje koristi Hibernate i koji omogućavaju jednostavno prevođenje iz Java u SQL tip podataka i obratno. U principu dozvoljeno je izostaviti tipove i ostaviti da Hibernate sam zaključi kog su tipa promenljive, međutim dobra je praksa eksplicitno navesti tipove, da Hibernate na to ne bi trošio vreme ili ne bi zaključio da je tip drugačiji od očekivanog (npr. Hibernate ne može da zna da li polje tipa `java.util.Date` treba da se preslika u kolonu tipa `date`, `timestamp` ili `time`). Detaljan spisak Hibernate tipova može se naći na adresi: https://docs.jboss.org/hibernate/orm/5.4/userguide/html_single/Hibernate_User_Guide.html#basic-provided.

Anotacije

Pristup u kome se koriste XML datoteke za zadavanje konfiguracije i preslikavanja ima i svoje prednosti i mane. Prednost bi bila njihova jednostavnost i čitljivost, ali ih karakteriše i opširnost. Preslikavanje objekata na tabele moguće je zadati i putem *anotacija* (eng. annotations), koje su uvedene u verziji Java 5 i koje su ubrzo prihvaćene od strane mnogih okruženja pa i od strane okruženja Hibernate. Anotacije predstavljaju metapodatke o preslikavanju objekata na tabele i one se dodaju klasi na nivou izvornog kôda. Pišu se ispred elementa klase na koji se odnose. Svaka anotacija počinje karakterom `@` i ima definisan svoj skup atributa koji je moguće koristiti. Atributi se anotaciji dodaju nakon navođenja naziva anotacije unutar male zagrade u obliku `atribut = vrednost` i, ako ih ima više, međusobno su razdvojene zapetama. Anotacije ne menjaju, niti utiču na to kako radi izvorni kôd. Za razliku od XML datoteka, one su prilično koncizne.

Hibernate koristi Java Persistence API (JPA) anotacije, te se prethodne anotacije učitavaju iz paketa `javax.persistence`.

U nastavku je dat jednostavan primer trajne klase `Knjiga` koja je definisana bez korišćenja anotacija:

```
public class Knjiga {
    private int id_knjige = 0;
    private String naziv = null;
    private String izdavac = null;
    private int god_izdavanja = 0;
    // get/set metode
    ...
}
```

Sve klase koje hoćemo da budu trajno zapamćene je potrebno definisati kao entitet navođenjem anotacije `@Entity` ispred definicije klase. Anotacijom `@Table` zadaje se naziv tabele baze podataka u kojoj će se čuvati ovi entiteti. Ako je ime klase isto kao i naziv odgovarajuće tabele, onda nije neophodno navoditi anotaciju `@Table`. U našem slučaju ovi nazivi se razlikuju pa je ovu anotaciju neophodno navesti.


```

@Entity
@Table(name = "Knjige")
public class Knjiga{
    private int id_knjige = 0;
    ...
}

```

Ako se ime polja razlikuje od imena odgovarajuće kolone tabele potrebno je anotacijom `@Column` zadati ime kolone u koju se polje slika. Ovde je to slučaj sa poljem `id_knjige` i kolonom `k_sifra`, te za polje `id_knjige` dodajemo anotaciju `@Column`, dok za ostala polja to nije neophodno uraditi.

```

@Entity
@Table(name = "Knjige")
public class Knjiga {
    @Column(name = "k_sifra")
    private int id_knjige = 0;
    private String naziv = null;
    private String izdavac = null;
    private int god_izdavanja = 0;
    ...
}

```

Putem atributa anotacije `@Column` moguće je koloni postaviti i neka dodatna svojstva. Na primer, ako kolona ne može da sadrži NULL vrednost, treba postaviti `nullable = false`. Slično, ako se kolona treba generisati sa ograničenjem jedinstvenosti treba postaviti `unique = true`. Na primer:

```

@Entity
@Table(name = "Knjige")
public class Knjiga {
    @Column(name = "k_sifra", nullable = false, unique = true)
    private int id_knjige = 0;
    ...
}

```

Ukoliko neko polje klase koje nije konstanta niti statička članica ne treba trajno pamtiti u tabeli, to je moguće uraditi korišćenjem anotacije `@Transient`.

Naredni korak jeste definisanje jedinstvenog identifikatora instance objekta – to postizemo dodavanjem anotacije `@Id` ispred polja koja predstavlja identifikator. U ovom primeru, primarnom ključu tabele `Knjige` odgovara polje `id_knjige` te njemu dodajemo ovu anotaciju.

```

@Entity
@Table(name = "Knjige")
public class Knjiga {
    @Id
    @Column(name = "k_sifra", nullable = false, unique = true)
    private int id_knjige = 0;
    private String naziv = null;
    ...
}

```

Napomenimo i to da je moguće ne anotirati polja objekta već odgovarajuće `get` metode. Ukoliko se anotacijom `@Id` anotira polje klase koje odgovara primarnom ključu, onda će Hibernate pristupati svojstvima objekta direktno kroz polja, dok ako anotiramo `get` metodu anotacijom `@Id`, onda se omogućava pristup svojstvima objekta kroz `get` i `set` metode. Mi nećemo ulaziti u detaljnu diskusiju koje su prednosti, a koje mane ovih pristupa, ali ćemo istaći da se uobičajeno anotiraju polja objekta.

Napomenimo i to da nekada u okviru baza podataka nije dozvoljeno koristiti polje `id` jer je to rezervisana reč, pa u toj situaciji treba promeniti ime u npr. `id_knjige`, kako je to ovde slučaj.

Hibernate podržava različite načine generisanja identifikatora, kao što smo videli i u slučaju XML preslikavanja. U prethodnom primeru strategija nije bila eksplicitno navedena, čime se podrazumevala `AUTO` strategija, kojom se oslanjamo na bazu podataka u generisanju ključeva. Različite strategije se mogu zadati postavljanjem anotacije `@GeneratedValue` na polje koje predstavlja identifikator i postavljanjem atributa `strategy` na jednu od narednih vrednosti:

- `GenerationType.AUTO` – strategija generisanja identifikatora zavisi od konkretnog dijalekta baze podataka koji se koristi; u najvećem broju slučajeva koristi se `GenerationType.SEQUENCE`
- `GenerationType.IDENTITY` – strategija se oslanja na automatsko inkrementiranje kolone baze podataka; ostavlja se bazi podataka da generiše novu vrednost za svaku novu operaciju umetanja; proces inkrementiranja se dešava izvan tekuće transakcije, te eventualno poništavanje transakcije može dovesti do odbacivanja već dodeljenih vrednosti (može doći do prekida u nizu vrednosti).
- `GenerationType.SEQUENCE` – naredna vrednost identifikatora se izdvaja na osnovu datog niza vrednosti definisanog u okviru baze podataka. Ukoliko niz nije eksplicitno zadat, Hibernate će razmatrati narednu vrednost iz svog podrazumevanog niza. Ukoliko želimo da eksplicitno zadamo niz, anotaciji `@GeneratedValue` kao vrednost atributa `generator` navodimo naziv generatora niza, zasebno zadatog anotacijom `@SequenceGenerator`. Ova anotacija podržava atribut `name` kojim se zadaje naziv generatora, `initialValue` kojom se postavlja inicijalna vrednost od koje kreću vrednosti koje se generišu (podrazumevana vrednost je 1), `allocationSize` kojom se zadaje vrednost za koju se vrši uvećavanje prilikom generisanja identifikatora (podrazumevana vrednost je 50). Ovaj niz je globalan i može se koristiti za više polja u jednoj ili većem broju klasa.
- `GenerationType.TABLE` – ova strategija je slična prethodnoj i ne koristi se često. Ona simulira niz čuvanjem i ažuriranjem njegove tekuće vrednosti u tabeli baze podataka, čime se zahteva korišćenje katanaca koje nameću svim transakcija sekvencijalni poredak. Ovim se narušavaju performanse same aplikacije.

Na primer:

```
@Entity
```

```
@Table(name = "Knjige")
public class Knjiga {
    @Id
    @Column(name = "k_sifra", nullable = false, unique = true)
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id_knjige = 0;
    ...
}
```

ili ako u bazi podataka imamo kreiran niz naredbom:

```
CREATE SEQUENCE dvostrukiNiz START WITH 20 INCREMENT BY 5;
```

```
@Entity
@Table(name = "Knjige")
public class Knjiga {
    @Id
    // definisemo niz, ova definicija moze biti i u drugoj klasi
    @SequenceGenerator(name = "nizId", sequenceName = "dvostrukiNiz",
        initialValue = 20, allocationSize = 100)
    // koristimo gore definisan niz
    @GeneratedValue(strategy = GenerationType.SEQUENCE, generator = "nizId")
    @Column(name = "k_sifra", nullable = false, unique = true)
    private int id_knjige = 0;
    ...
}
```

Ne postoji anotacija koja je odgovarajuća strategiji `assigned` u XML definicijama preslikavanja. Ukoliko želimo da se u aplikaciji postavljaju vrednosti jedinstvenog identifikatora, onda ne treba navesti anotaciju `@GeneratedValue` uz anotaciju `@Id`.

Atributom `generator` moguće je izmeniti npr. niz koji se koristi u strategiji `SEQUENCE` i slično.

Ovako anotiranu klasu možemo uključiti programski u okruženje Hibernate korišćenjem funkcije `addAnnotatedClass()` na sledeći način:

```
Configuration konf = new Configuration()
    .configure("anotacije/hibernate.cfg.xml")
    .addAnnotatedClass(Zaposleni.class);
```

Trajno čuvanje objekata

Napravimo sada klijenta koji trajno čuva objekte uz pomoć Hibernate alata. Najpre je potrebno kreirati instancu klase `SessionFactory` pomoću koje kreiramo objekat sesije. U nastavku je prikazano kako to možemo postići:

```
public class SkladisteKnjiga {
    private static SessionFactory fabrikaSesija = null;

    static{
        // gradimo servisni registar koriscenjem konfiguracionih svojstava
        StandardServiceRegistry servReg =
```

```

        new StandardServiceRegistryBuilder().configure().build();
    // pravimo fabriku sesija
    fabrikaSesija =
        new MetadataSources(servReg).buildMetadata().buildSessionFactory();
    ...
}

```

Kao što smo već pomenuli, ukoliko kao argument funkcije `configure` nije naveden naziv konfiguracione datoteke, podrazumeva se da se u putanji nalaze datoteke sa podrazumevanim nazivima kao što su: `hibernate.cfg.xml` ili `hibernate.properties`. Ako imena ovih datoteka nisu podrazumevana, onda se moraju navesti kao argumenti odgovarajućih metoda.

Kompletirajmo sada metod `sacuvajKnjigu` u okviru klase `SkladisteKnjiga` kojim je moguće trajno sačuvati knjigu. On poziva metod `save` nad objektom sesije dobijenim pozivom metoda `openSession()`. Ovaj mehanizam (bez try-catch bloka) prikazan je u okviru sledećeg koda:

```

public class SkladisteKnjiga{
    private void sacuvajKnjigu(Knjiga knjiga) {
        Session sesija = fabrikaSesija.openSession();
        Transaction tr = sesija.beginTransaction();
        sesija.save(knjiga);
        tr.commit();
        sesija.close();
    }
}

```

Postoje dva različita načina na koja je moguće testirati trajnost podataka: prvi je pokretanjem SQL upita na bazi podataka, a drugi korišćenjem klijenta za testiranje. Drugi od pomenutih načina je moguće implementirati dodavanjem još jednog metoda u okviru klijenta (nazovimo ga `nadjiKnjigu`). Ovaj metod poziva metod `load` klase `Session` da bi izdvojio odgovarajući red iz baze podataka.

```

public class SkladisteKnjiga{
    ...
    private void nadjiKnjigu(int id_knjige) {
        Session sesija = fabrikaSesija.openSession();
        Transaction tr = sesija.beginTransaction();
        Knjiga knjiga = (Knjiga)sesija.load(Knjiga.class, id_knjige);

        System.out.println("Knjiga:" + knjiga);
        tr.commit();
        sesija.close();
    }
}

```

Metod `load` izdvaja odgovarajući objekat klase `Knjiga` za dati jedinstveni identifikator koji se prosleđuje. Ovo funkcioniše tako što Hibernate u pozadini pokreće naredbu `SELECT` u ove svrhe.

Ukoliko želimo da izdvojimo sve knjige iz tabele, potrebno je kreirati upit kao objekat tipa Query pozivom metoda `createQuery` klase `Session` koji kao argument prima tekst upita "FROM Knjige" i izvršiti ga. Primenom metoda `list` na upit dobija se lista knjiga na sledeći način:

```
public class SkladisteKnjiga{
    // izdvajamo sve knjige
    private void nadjiSve() {
        Session sesija = fabrikaSesija.openSession();
        Transaction tr = sesija.beginTransaction();
        List<Knjiga> knjige = sesija.createQuery("FROM Knjige").list();
        tr.commit();
        System.out.println("Sve knjige:" + knjige);
        sesija.close();
    }
    ...
}
```

Složeni identifikatori

Ne važi uvek da jedna kolona na jedinstven način identifikuje red tabele. Nekada se kombinacija kolona uzima za primarni ključ tabele i takav ključ naziva se *složeni primarni ključ* (eng. compound primary key). U situaciji kada tabela ima složeni primarni ključ, jedinstveni identifikator objekta moramo zadati na drugačiji način.

Razmotrimo naredni primer: tabela `Kurs` ima složeni primarni ključ koji se sastoji od dva atributa `profesor` i `naziv`. U tom slučaju potrebno je definisati klasu `KursPK` koja sadrži dve promenljive: `profesor` i `naziv`.

```
@Embeddable
public class KursPK implements Serializable{

    private String profesor = null;
    private String naziv = null;

    // Podrazumevani konstruktor
    public KursPK() {
    }

    // metoda hashCode
    @Override
    public int hashCode(){
        ...
    }

    // metoda equals
    @Override
    public boolean equals(Object o){
        ...
    }
}
```

Ova klasa mora biti označena anotacijom `@Embeddable` čime se postiže da se složeni primarni ključ posmatra kao jedinstveno polje, potrebno je da implementira interfejs `Serializable` i da ima definisan podrazumevani konstruktor. Takođe, neophodno je da ima implementirane metode `hashCode` i `equals` koji pomažu da Hibernate proveri da li je došlo do kolizije na vrednostima primarnog ključa. Ove dve metode su označene anotacijom `@Override` koja nije neophodna, ali je poželjno navesti je zbog čitljivosti koda i zbog jednostavnijeg testiranja (ako napravimo neku grešku u nazivu metoda, tipovima argumenata i slično, dobićemo grešku u vreme kompilacije).

Naredni korak jeste da se ova klasa umetne u polje trajne klase, na primer `id`. Ovo radimo korišćenjem jednostavne `@Id` anotacije:

```
@Entity
@Table(name = "Kurs_annotacije")
public class Kurs {

    @Id
    private KursPK id = null;
    private int ukupnoStudenata = 0;
    private int registrovanihStudenata = 0;

    public KursPK getId() {
        return id;
    }

    public void setId(KursPK id) {
        this.id = id;
    }
    ...
}
```

Istestirajmo sada na koji način se složeni ključ formira u tabeli:

```
private void sacuvaj() {
    ...
    Kurs kurs = new Kurs();
    KursPK kursPk = new KursPK();
    kursPk.setNaziv("Racunarstvo i informatika");
    kursPk.setProfesor("Prof. Petar Petrovic");
    kurs.setId(kursPk);
    kurs.setUkupnoStudenata(20);
    kurs.setRegistrovanihStudenata(12);
    sesija.save(kurs);
    ...
}
```

Napomenimo da postoje i drugi načini za zadavanje složenih ključeva, koje nećemo ovde razmatrati.

Trajno čuvanje kolekcija

U kontekstu programiranja u programskom jeziku Java, rad sa kolekcijama je neizbežan. Hibernate podržava Java kolekcije kao što su: `List`, `Set`, `Array`, `Map`, ...

U programskom jeziku Java postoje naredni interfejsi kolekcija:

- `java.util.Collection` (koji je roditelj svih interfejsa kolekcija osim `mapa`),
- `java.util.List` (za rad sa listama),
- `java.util.Set` (za rad sa skupovima),
- `java.util.Map` (za rad sa mapama, tj. preslikavanjima oblika ključ/vrednost),
...

Interfejs `List` je implementiran tako da čuva uređenu listu elemenata. Mi koristimo njegove konkretne implementacije kao što su `ArrayList` ili `LinkedList`, ali za potrebe razvoja Hibernate aplikacije listu je potrebno dizajnirati tako da implementira interfejs `List`, a ne njegovu konkretnu implementaciju. Dakle, potrebno je da uvek koristimo interfejs prilikom definisanja promenljivih koje odgovaraju kolekcijama i umesto:

```
ArrayList<String> knjige = new ArrayList<String>();
```

iskoristiti:

```
List<String> knjige = new ArrayList<String>();
```

Razmotrićemo najpre kako trajno čuvati kolekcije korišćenjem XML datoteka preslikavanja.

Trajno čuvanje kolekcija korišćenjem XML datoteka preslikavanja

Trajno čuvanje liste Liste predstavljaju jednostavnu strukturu podataka za čuvanje objekata u uređenom poretku. One, takođe, čuvaju informaciju o poziciji elementa, posredstvom indeksa. Element možemo umetnuti na proizvoljno mesto u listi, a možemo i izdvojiti proizvoljan element iz liste na osnovu njegovog indeksa. Na primer, pretpostavimo da smo trajno sačuvali listu automobila korišćenjem programa u programskom jeziku Java. Očekujemo da iz tabele izdvojimo podatke o automobilima u onom poretku u kom su bili umetani (pri čemu to naravno ne mora da odgovara redosledu u kome su redovi zapamćeni u tabeli). Dakle, prilikom pokretanja upita `lista.get(n)`, očekujemo da bude vraćen n -ti element liste. Da bi ovaj zahtev bio zadovoljen, Hibernate održava još jednu tabelu sa indeksima automobila. Prilikom izdvajanja automobila iz glavne tabele, izdvaja se indeksirani poredak elemenata iz dodatne tabele. Nakon toga vrši se povezivanje odgovarajućih elemenata da bi se odredio njihov ispravan poredak. Na ovaj način se omogućava izdvajanje automobila u zadatom poretku.

Razmotrimo primer salona automobila koji raspolaže određenim brojem automobila koje mušterije mogu da razgledaju i potencijalno kupe. Automobile ćemo modelovati korišćenjem interfejsa `java.util.List` na sledeći način:

```
public class Salon {
    private int id = 0;
    private String menadzer = null;
    private String lokacija = null;
    private List<Automobil> automobili = null;
    ...
    // set i get metode
}

public class Automobil {
    private int id;
    private String proizvođjac = null;
    private String boja = null;
    ...
    // set i get metode
}
```

Obratimo pažnju da klasa Salon sadrži polje koje odgovara listi automobila. Definicije preslikavanja ove dve klase (Salon i Automobil) na odgovarajuće tabele (Salon_lista i Automobil_lista) bile bi oblika:

```
<hibernate-mapping>
  <class name = "Salon" table = "Salon_lista">
    <id name = "id" column = "ID_salona">
      <generator class = "native"/>
    </id>
    <property name = "menadzer" column = "menadzer"/>
    <property name = "lokacija" column = "lokacija"/>
    <list name = "automobili" cascade = "all"
      table = "Automobil_lista">
      <key column = "ID_salona"/>
      <list-index column = "automobil_index"/>
      <one-to-many class = "Automobil"/>
    </list>
  </class>

  <class name = "Automobil" table = "Automobil_lista">
    <id name = "id" column = "ID_automobila">
      <generator class = "native"/>
    </id>
    <property name = "proizvođjac" column = "proizvođjac"/>
    <property name = "boja" column = "boja"/>
  </class>
</hibernate-mapping>
```

Lista se preslikava korišćenjem elementa <list>. Ovaj element definiše preslikavanje polja automobili klase Salon na tabelu Automobil_lista. Atribut name elementa <list> ukazuje na ime polja klase, a atribut table na tabelu u kojoj se pamte elementi liste.

Atribut cascade elementa <list> se koristi da bi se postavilo da li je pri likom trajnog čuvanja/brisanja/menjanja tog objekta potrebno da se automatski

pamte/brišu/izmene i svi njemu pridruženi objekti (npr. pamćenjem podataka o salonu automobila, pamte se i podaci o automobilima u odgovarajućoj tabeli). Ukoliko je ona postavljena na `cascade = "all"` sve pomenute operacije se kaskadno izvršavaju na svim pridruženim entitetima. Podrazumevana vrednost ovog atributa je `cascade = "none"`, čime se ignorišu sve asocijacije. Druge interesantne vrednosti ovog atributa su `cascade = "save-update"` čime se operacije trajnog pamćenja ili ažuriranja kaskadno izvršavaju na svim pridruženim entitetima (a ostale operacije ne), dok opcija `cascade = "delete"` omogućava kaskadno izvršavanje brisanja na svim pridruženim entitetima (dok to ne važi za trajno čuvanje i ažuriranje).

Element `<list>` sadrži elemente:

- `key` – njime se zadaje kolona tabele `Automobil_lista` koja predstavlja strani ključ ka tabeli `Salon_lista`;
- `list-index` – njime se zadaje u koju kolonu tabele `Automobil_lista` se preslikava pozicija elementa u uređenoj kolekciji; prvi element ima podrazumevanu vrednost 0, koja se može izmeniti postavljanjem vrednosti atributu `base`;
- `one-to-many` – ovim se postavlja tip veze između klasa `Salon` i `Automobil`: ovde označava da se jedan objekat klase `Salon` povezuje sa potencijalno više objekata klase `Automobil`; pored ovog tipa moguće je zadati i druge tipove veza elementima `<one-to-one>`, `<many-to-one>` i `<many-to-many>` o kojima će biti više reči kasnije.

Glavna tabela, `Salon_lista` biće kreirana i popunjena kao što je i očekivano, ali će nam pored nje biti potrebna i dodatna tabela `Automobil_lista`. Pored kolona `ID_automobila`, `proizvodjac` i `boja` koja su direktno deklarirana na samom objektu, Hibernate tabeli `Automobil_lista` dodaje još dve kolone. Jedna od njih je strani ključ `ID_salona` na tabelu `Salon_lista` dok je druga `automobil_index` koja treba da sadrži indekse liste i koja je zadana korišćenjem etikete `list-index` (sva preslikavanja kolekcija, osim `set` i `bag` zahtevaju kolonu indeksa u tabeli kolekcije i to je kolona koja se preslikava u indeks niza, indeks liste ili ključ mape). Kolona `automobil_index` se popunjava pozicijom elementa liste i koristi se za kasniju rekonstrukciju elemenata liste na originalnim pozicijama. Prilikom izdvajanja automobila u vreme izvršavanja Hibernate preuređuje redove u skladu sa njihovim indeksom datim u koloni `automobil_index`.

Pokrenimo jednostavnog klijenta zarad testiranja funkcionalnosti trajnog čuvanja liste:

```
private void sacuvajListu() {

    // pravimo objekat salon
    Salon salon = new Salon();
    salon.setLokacija("Tosin bunar, Beograd");
    salon.setMenadzer("Petar Petrovic");

    // pravimo listu automobila
    List<Automobil> automobili = new ArrayList<Automobil>();
```

```

    automobili.add(new Automobil("Tojota", "Plava"));
    automobili.add(new Automobil("Tojota", "Plava"));
    automobili.add(new Automobil("Opel", "Bela"));
    automobili.add(new Automobil("BMW", "Crna"));
    automobili.add(new Automobil("Mercedes", "Srebrna"));
    ...

    // povezimo automobile sa salonom
    salon.setAutomobili(automobili);

    // pamtimo podatke o salonu
    sesija.save(salon);
}

```

Prilikom štamapanja informacija o datom salonu dobija se sledeći izlaz (primetimo duplirani Tojota automobil u izlazu):

```

Salon{id = 6, menadzer = Petar Petrovic,
lokacija = Tosin bunar, Beograd,
automobili = [Automobil{id = 15, proizvođjac = Tojota, boja = Plava},
Automobil{id = 16, proizvođjac = Tojota, boja = Plava},
Automobil{id = 17, proizvođjac = Opel, boja = Bela},
Automobil{id = 18, proizvođjac = BMW, boja = Crna},
Automobil{id = 19, proizvođjac = Mercedes, boja = Srebrna}]}

```

Kao što se može primetiti, Hibernate uzima u obzir redosled umetanja automobila u listu.

Trajno čuvanje skupova `java.util.Set` predstavlja neuređenu strukturu podataka u kojoj nije dozvoljeno pojavljivanje duplikata. Izmenimo prethodni primer tako da je kolekcija automobila koja pripada salonu automobila modelovana kao skup. Koristimo `HashSet` za konkretnu implementaciju `Set` interfejsa.

```

public class Salon {
    private int id = 0;
    private String menadzer = null;
    private String lokacija = null;
    private Set<Automobil> automobili = null;
    // get i set metode
    ...
}

```

U ovom slučaju preslikavanje skupa vrši se korišćenjem etikete `set`:

```

<hibernate-mapping>
    <class name = "Salon" table = "Salon_skup">
        <id name = "id" column = "ID_salona">
            <generator class = "native"/>
        </id>

```

```

    ...
    <set name = "automobili" cascade = "all"
        table = "Automobil_skup">
        <key column = "ID_salona"/>
        <one-to-many class = "Automobil"/>
    </set>
</class>

<class name = "Automobil" table = "Automobil_skup">
    <id name = "id" column = "ID_automobila">
        <generator class = "native"/>
    </id>
    <property name = "proizvodjac" column = "proizvodjac"/>
    <property name = "boja" column = "boja"/>
</class>
</hibernate-mapping>

```

Instanca klase Salon se preslikava u red tabele Salon_skup, dok se polje automobili koje je predstavljeno skupom preslikava u tabelu Automobil_skup. Element <key> označava kolonu tabele Automobil_skup koja sadrži strani ključ ka tabeli Salon_skup. Kao što smo već pomenuli, Hibernate automatski dodaje ovaj strani ključ tabeli Automobil_skup. Stoga će tabela Automobil_skup, koju Hibernate kreira i njome upravlja, imati dodatni strani ključ ID_salona kojim se povezuju ove dve tabele. Skup ne predstavlja uređenu kolekciju, te se u ovom slučaju ne dodaje kolona indeksa.

Možemo kreirati klijenta koji će nam služiti za testiranje:

```

private void sacuvajSkup() {

    // Kreiramo i popunjavamo salon
    Salon salon = new Salon();
    salon.setLokacija("Tosin bunar, Beograd");
    salon.setMenadzer("Petar Petrovic");

    // Kreiramo i popunjavamo skup automobila
    Set<Automobil> automobili = new HashSet<Automobil>();
    automobili.add(new Automobil("Tojota", "Plava"));
    automobili.add(new Automobil("Opel", "Bela"));
    automobili.add(new Automobil("BMW", "Crna"));
    automobili.add(new Automobil("BMW", "Crna"));
    ...

    // povezimo automobile sa salonom
    salon.setAutomobili(automobili);

    // pamtimo podatke o salonu automobila
    sesija.save(salon);
}

```

U prikazanom primeru kreirali smo objekat tipa `Salon` i dodali mu tri automobila. Pokušaj dodavanja istog automobila ne uspeva jer se utvrđuje da su ova dva automobila identična i duplikat se ne pamti u kolekciji.

Prilikom rada sa skupovima zbog potrebe da se ne čuvaju duplikati, potrebno je u klasi `Automobil` definisati metode `equals` i `hashCode` u klasi `Automobil`. Kao što znamo skup ne može da sadrži dva ista elementa i ova dva metoda pomažu da se ispuni ovaj zahtev.

Trajno čuvanje mapa Kada postoji zahtev za predstavljanjem parova oblika ključ/vrednost, pogodno je koristiti mape. Struktura podataka `Map` je kao rečnik u kome postoji ključ (reč) i odgovarajuća vrednost (značenje). Razmotrimo dalje primer salona automobila i dodajmo mu mogućnost da on sadrži potencijalne rezervacije mušterija za testnu vožnju automobila. Ovu funkcionalnost možemo implementirati korišćenjem mapa, povezivanjem mušterija sa rezervacijama automobila:

```
public class Salon {
    private int id = 0;
    private String menadzer = null;
    private String lokacija = null;
    private Map<String, Automobil> automobili = null;
    // get i set metode
}
```

Svaki automobil je rezervisan od strane neke mušterije, i svi automobili pripadaju salonu automobila. Svaka mušterija može rezervisati samo jedan automobil. Mušteriju ćemo predstaviti stringom koji odgovara imenu, te ćemo koristiti tip podataka `Map<String, Automobil>`. Ključna stvar jeste kako zadatai odgovarajuće preslikavanje:

```
<hibernate-mapping>
  <class name = "Salon" table = "Salon_mapa">
    <id name = "id" column = "ID_salona">
      <generator class = "native"/>
    </id>
    <property column = "menadzer" name = "menadzer"/>
    ...
    <map name = "automobili" cascade = "all"
      table = "Automobil_mapa">
        <key column = "ID_salona"/>
        <map-key column = "ime_musterije" type = "string" />
        <one-to-many class = "Automobil"/>
      </map>
    </class>

  <class name = "Automobil" table = "Automobil_mapa">
    <id column = "ID_automobila" name = "id">
      <generator class = "native"/>
    </id>
    <property name = "proizvodjac" column = "proizvodjac" />
```

```

        <property name = "boja" column = "boja" />
    </class>
</hibernate-mapping>

```

Kao što je i očekivano, polje `automobili` klase `Salon` je predstavljeno elementom `map` koji se odnosi na tabelu `Automobil_mapa` iz definicije preslikavanja. Elementom `<key>` se zadaje kolona stranog ključa ka tabeli `Salon_mapa` (u ovom slučaju `ID_salona`). Element `<map-key>` definiše ključ mape – u ovom slučaju to je `ime_musterije`.

Preslikavanje klase `Automobil` je pravolinijsko. Naglasimo i to da Hibernate dodaje još nekoliko kolona tabeli `Automobil_mapa`: to su `ID_salona` i `ime_musterije`. Ovako definisana preslikavanja možemo testirati korišćenjem narednog fragmenta kôda:

```

private void sacuvajMapu() {
    Salon salon = new Salon();
    salon.setLokacija("Tosin bunar, Beograd");
    salon.setMenadzer("Petar Petrovic");
    Map<String, Automobil> automobili = new HashMap<String, Automobil>();
    automobili.put("Marko", new Automobil("Tojota", "Zelena"));
    automobili.put("Milos", new Automobil("Opel", "Bela"));
    automobili.put("Milan", new Automobil("BMW", "Crna"));
    salon.setAutomobili(automobili);

    // pamtimo podatke o salonu
    sesija.save(salon);
}

```

Primitimo da je u slučaju sve tri pomenute kolekcije preslikavanje klase `Automobil` bilo nepromenjeno, dok se preslikavanje klase `Salon` razlikuje samo u zadavanju odgovarajućeg elementa kojim je kolekcija modelovana.

Trajno čuvanje kolekcija korišćenjem anotacija

Videli smo kako je kolekcije moguće trajno sačuvati korišćenjem XML datoteka za zadavanje preslikavanja. Odgovarajuća preslikavanja je moguće zadati i korišćenjem anotacija. Pre nego što kodu koji sadrži kolekcije dodamo anotacije, potrebno ga je pripremiti na jedan od dva načina:

- korišćenjem stranog ključa,
- korišćenjem spojne tabele.

Razmotrimo najpre metod pripreme zasnovan na *korišćenju stranog ključa*. S obzirom na to da za dati salon možemo dobiti informaciju koji automobili pripadaju tom salonu, a da za automobil ne možemo ustanoviti kom salonu pripada, ova veza je jednosmerna. Svaki salon sadrži potencijalno više automobila, što predstavlja vezu tipa "jedan prema više". Dakle, entitet `Salon` sadrži kolekciju automobila. Automobili, sa druge strane, pripadaju salonu; stoga su modelovani da imaju odnos stranog ključa ka salonu.

Razmotrimo najpre entitet `Salon`:

```
@Entity
@Table(name = "Salon_lista_an")
public class Salon {
    @Id
    @Column(name = "ID_salona")
    @GeneratedValue(strategy = GenerationType.AUTO)
    private int id = 0;

    @OneToMany
    @JoinColumn(name = "ID_salona")
    @Cascade(CascadeType.ALL)
    private List<Automobil> automobili = null;

    // ostala polja
    private String menadzer = null;
    private String lokacija = null;
}
```

Klasa Salon je deklarirana kao trajni entitet, koji se preslikava u tabelu pod nazivom Salon_lista_an. Identifikator ove klase se postavlja korišćenjem automatske strategije.

Osvrnimo se na važno svojstvo klase Salon: kolekcija automobila je predstavljena poljem automobili i ono je tipa lista: java.util.List. Ovo polje je označeno anotacijom @OneToMany jer svaki salon sadrži potencijalno više automobila, a svaki automobil pripada tačno jednom salonu. U ovom pristupu kolekcija automobila treba da ima svoju tabelu sa stranim ključem koji referiše na primarni ključ tabele Salon (u ovom slučaju to je ID_salona). Da bi Hibernate znao za ovu zavisnost, polju automobili dodajemo anotaciju @JoinColumn kojom se definiše strani ključ. Da bismo pokupili listu automobila iz tabele automobila potrebno je proslediti vrednost kolone ID_salona. Anotacijom @Cascade(CascadeType.ALL) zadaje se da Hibernate prilikom čuvanja objekta/brisanja/ažuriranja trajno sačuva/obriše/ažurira i kolekcije povezane sa glavnim instancom. Ukoliko je potrebno ovo ponašanje omogućiti samo za čuvanje objekta, potrebno je postaviti opciju @Cascade(CascadeType.PERSIST), a ako želimo da samo obezbedimo kaskadno brisanje svih pridruženih objekata potrebno je postaviti opciju @Cascade(CascadeType.REMOVE). Napomenimo da postoje i druge moguće vrednosti anotacije @Cascade.

U entitetu Automobil nema nikakvih dodatnih anotacija:

```
@Entity
@Table(name = "Automobil_lista_an")
public class Automobil {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    @Column(name = "ID_automobila")
    private int id;
    private String proizvođač = null;
    private String boja = null;
    ...
}
```

Ovakva preslikavanja odgovaraju narednim definicijama tabela `Salon_lista_an` i `Automobil_lista_an`.

```
CREATE TABLE Salon_lista_an (
    ID_salona integer NOT NULL,
    lokacija varchar(255) DEFAULT NULL,
    menadzer varchar(255) DEFAULT NULL,
    PRIMARY KEY (ID_salona)
)

CREATE TABLE Automobil_lista_an (
    ID_automobila integer NOT NULL,
    boja varchar(255) DEFAULT NULL,
    proizvođjac varchar(255) DEFAULT NULL,
    ID_salona integer DEFAULT NULL,
    PRIMARY KEY (ID_automobila),
    FOREIGN KEY (ID_salona) REFERENCES Salon_lista_an (ID_salona)
)
```

Tabela `Automobil_lista_an` ima primarni ključ `ID_automobila` i strani ključ `ID_salona` koji referiše na tabelu `Salon_lista_an`.

Podsetimo se da je anotirane klase potrebno dodati konfiguraciji okruženja na sledeći način:

```
StandardServiceRegistry registry = new StandardServiceRegistryBuilder()
    .configure("kolekcije/hibernate.cfg.xml").build();

SessionFactory = new MetadataSources(registry)
    .addAnnotatedClass(Salon.class)
    .addAnnotatedClass(Automobil.class)
    .buildMetadata()
    .buildSessionFactory();
```

Sam mehanizam trajnog čuvanja kolekcije se ne bi ni u čemu razlikovao od prethodnog slučaja kada smo razmatrali XML datoteke preslikavanja.

Sada, kada smo videli kako da sačuvamo kolekcije korišćenjem stranog ključa, razmotrimo i drugi metod: *korišćenjem spojne tabele*. Prilikom korišćenja ove strategije, potrebno je da postoji spojna tabela preslikavanja koja sadrži primarne ključeve obe tabele.

U primeru salona automobila potrebno je napraviti neke izmene u anotiranju:

```
@Entity
@Table(name = "Salon_spojnatabela")
public class Salon {
    @Id
    @Column(name = "ID_salona")
    @GeneratedValue(strategy = GenerationType.AUTO)
    private int id = 0;

    @OneToMany
```

```
@JoinTable
(name = "Salon_automobil_spojnatabela",
 joinColumns = @JoinColumn(name = "ID_salona")
)
@Cascade(CascadeType.ALL)
private Set<Automobil> automobili = null;

private String menadzer = null;
private String lokacija = null;
...
}
```

Anotacija `@JoinTable` u prethodnom fragmentu koda ukazuje na to da će biti kreirana spojna tabela pod nazivom `Salon_automobil_spojnatabela`. Atribut `joinColumns` anotacije `@JoinTable` se postavljaju kolone stranog ključa spojne tabele koje referišu na primarni ključ entiteta koji je odgovoran za održavanje asocijacije (ovde je to `Salon_spojnatabele`). Spojna tabela koja se automatski kreira ima sledeću formu:

```
CREATE TABLE Salon_automobil_spojnatabela (
ID_salona integer NOT NULL,
ID_automobila integer NOT NULL,
PRIMARY KEY (ID_salona, ID_automobila),
FOREIGN KEY (ID_salona) REFERENCES Salon_spojnatabela (ID_salona),
FOREIGN KEY (ID_automobila) REFERENCES Automobil_spojnatabela (ID_automobila)
)
```

Primarni ključ ove tabele čini kombinacija kolona `ID_salona` i `ID_automobila`.

Ukoliko se ne navede na koji način treba uraditi fizičko preslikavanje (`@JoinColumn` ili `@JoinTable`) koristi se pristup zasnovan na spojnoj tabeli čiji se podrazumevani naziv dobija konkatenacijom naziva glavne tabele, znaka `_` i kolone primarnog ključa glavne tabele.

Rad sa ostalim vrstama kolekcija je sličan. Ukoliko radimo sa uređenim kolekcijama, uređenost postizemo dodavanjem anotacije `@OrderColumn` polju kolekcijskog tipa. Ova anotacija ima atribut `name` kojim se zadaje naziv kolone koja će sadržati vrednost indeksa. Ova kolona se dodaje tabeli `Automobil`.

U slučaju mapa, potrebno je zadati gde se čuva ključ mape: ključ može biti neko od postojećih polja samog entiteta ili može postojati namenska kolona za čuvanje vrednosti ključa.

Vratimo se primeru klase `Salon` kod koga se koriste mape. Potrebno je definisati da će se za ključ mape koristiti polje `ime_musterije` klase `Automobil`.

```
@Entity
@Table(name = "Salon_mapa")
public class Salon {
    @Id
    @Column(name = "ID_salona")
    @GeneratedValue(strategy = GenerationType.AUTO)
    private int id = 0;
    private String menadzer = null;
```



```

    private String lokacija = null;

    @OneToMany
    @MapKey(name = "ime_musterije")
    @Cascade(CascadeType.ALL)
    private Map<String, Automobil> automobili = null;
    // get i set metode
}

```

Druga mogućnost jeste da se koristi spojna tabela i tada je pored anotacije @JoinTable potrebno navesti i anotaciju @MapKeyColumn kojom se zadaje ključ te mape. Dakle spojna tabela će od kolona sadržati primarne ključeve obe tabele i ovu kolonu ključa.

```

@Entity
@Table(name = "Salon_mapa")
public class Salon {
    @Id
    @Column(name = "ID_salona")
    @GeneratedValue(strategy = GenerationType.AUTO)
    private int id = 0;
    private String menadzer = null;
    private String lokacija = null;

    @OneToMany
    @JoinTable(name = "Salon_automobil_mapa")
    @MapKeyColumn(name = "ime_musterije")
    @Cascade(CascadeType.ALL)
    private Map<String, Automobil> automobili = null;
    // get i set metode
}

```

Asocijacije (veze)

U programskom jeziku Java veze se modeluju na jednostavan način: u te svrhe se koriste polja klase. Razmotrimo naredni primer:

```

public class Automobil {
    private int id;
    // Automobil ima motor
    private Motor motor;
    ...
}

public class Motor {
    private int id = 0;
    private String proizvođjac = null;
    ...
}

```

Iz ovog koda možemo zaključiti da je klasa `Automobil` povezana sa klasom `Motor`.

Pod *asocijacijom (vezom)* podrazumevaćemo veze između tabela relacione baze podataka. One se najčešće realizuju korišćenjem mehanizama primarnih i stranih ključeva. Veze karakterišu dva svojstva: *višestrukost* i *usmerenost*. Termin višestrukost se odnosi na to koliki broj objekata je povezan sa koliko ciljnih objekata (odnosno, koliki broj objekata je sa svake strane veze). U prethodnom primeru mogli bismo zaključiti da svaki automobil ima jedan motor, a svaki motor pripada jednom automobilu. Ovo je tip veze “jedan prema jedan”. Pored ovog tipa veze postoje i tipovi “više prema više” i “jedan prema više”.

Drugo bitno svojstvo veze je njena usmerenost. Asocijacije mogu biti *jednosmerne* ili *dvosmerne*. U primeru automobila i motora, ispitivanjem atributa automobila možemo da ustanovimo koji je motor u pitanju, dok na osnovu motora ne možemo dobiti detalje o automobilu. Veza između automobila i motora je jednosmerna.

S druge strane, ako se možemo kretati od polaznog do ciljnog objekta, kao i u suprotnom smeru, za vezu se kaže da je dvosmerna. Za dvosmernu vezu se uobičajeno definiše jedna strana koja je odgovorna za održavanje te veze (kažemo i da ova strana vlasnik veze) i druga – pridružena ili inverzna strana veze koja nije odgovorna za održavanje te veze.

```
public class Vlasnik {
    private int id = 0;
    // mogli bismo imati listu automobila koje vlasnik poseduje
    // ali jednostavnosti radi neka bude samo jedan automobil
    private Automobil automobil = null;
    ...
}

public class Automobil {
    private int id = 0;
    private Vlasnik vlasnik = null;
    ...
}
```

U slučaju klasa `Automobil` i `Vlasnik` važi:

- za svaki automobil možemo odrediti vlasnika, i
- za svakog vlasnika možemo dobiti informaciju o njegovom automobilu.

Ovo je primer dvosmerne veze tipa “jedan prema jedan”.

Vezu tipa “više prema više” možemo uočiti u odnosu studenta i kursa. Naime, možemo za datog studenta utvrditi koje kurseve on pohađa, a, takođe, za dati kurs zaključiti koji studenti pohađaju taj kurs.

```
public class Student {
    private int id = 0;
    private List<Kurs> kursevi = null;
    ...
}
```

```

}

public class Kurs {
    private int id = 0;
    private List<Student> studenti = null;
    ...
}

```

Na ovaj način se ostvaruje veza tipa “više prema više”.

Preslikavanje putem XML datoteka preslikavanja

Modelovanje veze tipa “jedan prema jedan” Postoje dva načina za uspostavljanje veze tipa “jedan prema jedan”:

- korišćenjem deljenog primarnog ključa,
- korišćenjem stranog ključa.

U objektnom modelu ova razlika nije očigledna, ali u relacionom modelu jeste. Ilustrovaćemo ovu vezu na primeru narednih klasa Automobil i Motor:

```

public class Automobil {
    private int id;
    private String proizvođač = null;
    private String boja = null;
    private Motor motor = null;
    ...
}

public class Motor {
    private int id = 0;
    private String proizvođač = null;
    private String model = null;
    private Automobil automobil = null;
}

```

Svaka od klasa sadrži referencu na drugu klasu kroz polja motor i automobil. Ovim je zadata dvosmerna “jedan prema jedan” veza u kojoj je moguće kretati se od jednog entiteta ka drugom i obratno.

Razmotrimo kako zadati sheme tabela za uspostavljanje veze “jedan prema jedan” korišćenjem *deljenog primarnog ključa*.

```

CREATE TABLE Automobil (
    ID_automobila integer NOT NULL,
    proizvođač varchar(20) DEFAULT NULL,
    boja varchar(20) DEFAULT NULL,
    PRIMARY KEY (ID_automobila))

```

```

CREATE TABLE Motor (
    ID_automobila integer NOT NULL,

```

```

    proizvođač varchar(20) DEFAULT NULL,
    model varchar(20) DEFAULT NULL,
    PRIMARY KEY (ID_automobila),
    FOREIGN KEY (ID_automobila) REFERENCES Automobil (ID_automobila))

```

U ovom pristupu treba obratiti pažnju da je primarni ključ tabele Motor kolona ID_automobila. Takođe, postoji ograničenje stranog ključa na tabelu Automobil. Stoga motor mora uvek da bude kreiran sa istim identifikatorom kao i automobil, te kažemo da ove dve tabele dele primarni ključ.

Razmotrimo sada kako zadati preslikavanje za vezu tipa "jedan prema jedan":

```

<hibernate-mapping>
  <class name = "Automobil" table = "Automobil">
    <id name = "id" column = "ID_automobila">
      <generator class = "assigned"/>
    </id>
    <property name = "proizvođač" column = "proizvođač"/>
    <property name = "boja" column = "boja"/>
    <one-to-one name = "motor" class = "Motor" cascade = "all"/>
  </class>
  <class name = "Motor" table = "Motor">
    <id name = "id" column = "ID_automobila" >
      <generator class = "foreign">
        <param name = "property">automobil</param>
      </generator>
    </id>
    <one-to-one name = "automobil" class = "Automobil"
      constrained = "true"/>
    ...
  </class>
</hibernate-mapping>

```

Ono što je novo jeste etiketa <one-to-one> kojom se preslikava polje motor. Element <one-to-one> se koristi na obe strane veze da bi označio njihovu uzajamnu vezu tipa "jedan prema jedan". Ranije su elementi ovog tipa bili ugnježdjeni u element kolekcijskog tipa (<set>, <map> itd), pa nismo zadavali sve ove attribute ovom elementu. U ovom slučaju je potrebno navesti attribute name i cascade uz element <one-to-one>, koji su u ranijim primerima kod kolekcija bili navedeni uz odgovarajući element kolekcije. Postoji još jedan novi atribut koji se koristi u elementu <one-to-one>, a to je constrained = "true". Na ovaj način se za objekat klase Motor postavlja uslov da automobil mora da postoji, tj. motor ne može da postoji bez automobila.

Znamo da je primarni ključ (id) motora isti kao id automobila. Ovu činjenicu treba na neki način pomenuti u samom preslikavanju. U ove svrhe postoji specijalna klasa generatora foreign koju do sada nismo spominjali. Ovim generatorom se nameće uslov da se primarni ključ tabele Motor generiše na osnovu primarnog ključa tabele Automobil.

Aplikaciju možemo testirati korišćenjem naredne funkcije:

```

private void sacuvaj() {
    ...
    // Kreiramo instancu klase Automobil, postavljamo id i ostala polja
    Automobil automobil = new Automobil();

    // Setimo se, koristimo generator aplikacije za id
    automobil.setId(1);
    automobil.setProizvodjac("Kadilak Sedan");
    automobil.setBoja("Bela");

    // Kreiramo instancu klase Motor i postavljamo vrednosti polja
    // Paznja: ne postavljamo id
    Motor motor = new Motor();
    motor.setProizvodjac("Sedan");
    motor.setModel("DTS");

    // Sada ih povezujemo zajedno
    automobil.setMotor(motor);
    motor.setAutomobil(automobil);

    // Na kraju, trajno pamtimo objekte
    sesija.save(automobil);
    ...
}

```

Jedinstveni ključ motora se ne postavlja prilikom pravljenja: to je zato što pozajmljujemo i delimo id automobila. Objekat klase Motor čuva se automatski tako što se sačuva samo objekat klase Automobil – ovo je omogućeno postavljanjem vrednosti atributa `cascade = "all"`.

Modelovanju odnosa “jedan prema jedan” korišćenjem *ograničenja stranog ključa* odgovaraju nešto izmenjene definicije tabela i preslikavanja:

```

CREATE TABLE Automobil_V2 (
    ID_automobila integer NOT NULL,
    ID_motora integer NOT NULL,
    boja varchar(20) DEFAULT NULL,
    proizvodjac varchar(20) DEFAULT NULL,
    PRIMARY KEY (ID_automobila),
    CONSTRAINT fk_motor_id FOREIGN KEY (ID_motora)
        REFERENCES Motor_V2 (ID_motora)
)

CREATE TABLE Motor_V2 (
    ID_motora integer NOT NULL,
    proizvodjac varchar(20) DEFAULT NULL,
    model varchar(20) DEFAULT NULL,
    PRIMARY KEY (ID_motora)
)

```

Definicija tabele Motor_V2 je jednostavna: to je standardna tabela sa primarnim ključem ID_motora. Primetna razlika je u tabeli Automobil_V2: pored

svog primarnog ključa (ID_automobila), postoji i kolona ID_motora koja predstavlja strani ključ na tabelu Motor_V2.

Preslikavanje klase Motor nije ni u čemu drugačije od dosadašnjeg, a preslikavanje klase Automobil se razlikuje:

```
<hibernate-mapping >
  <class name = "Motor" table = "Motor_V2">
    <id name = "id" column = "ID_motora" >
      <generator class = "assigned"/>
    </id>
    ...
  </class>
  <class name = "Automobil" table = "Automobil_V2">
    <id name = "id" column = "ID_automobila" >
      <generator class = "assigned"/>
    </id>
    <property name = "proizvodjac" column = "proizvodjac" />
    <property name = "boja" column = "boja"/>

    <many-to-one name = "motor" class = "Motor"
      column = "id_motora"
      unique = "true"
      cascade = "all" />
  </class>
</hibernate-mapping>
```

Za modelovanje odnosa “jedan prema jedan” putem ograničenja stranog ključa potrebno je iskoristiti element `<many-to-one>` (koji omogućava zadavanje stranih ključeva), ali se stavkom `unique = 'true'` postavlja da je tačan odnos između njih u stvari “jedan prema jedan”. Pored atributa koje smo već ranije koristili uz ovakve elemente, navodi se i atribut `column` koji u ovom slučaju označava naziv kolone koja predstavlja strani ključ.

Izmenimo klijenta za testiranje:

```
private void sacuvaj_V2() {

    // pravimo motor
    Motor m = new Motor();
    m.setId(1);
    m.setProizvodjac("Sedan");
    m.setModel("DTS");

    // pravimo automobil
    Automobil automobil = new Automobil();
    automobil.setId(1);
    automobil.setProizvodjac("Kadilak Sedan");
    automobil.setBoja("Bela");

    // povezujemo ih zajedno
    automobil.setMotor(m);
}
```

```
// trajno cuvamo automobil koriscenjem metoda save
sesija.save(automobil);
...
}
```

Primetimo da se motor ne pamti eksplicitno. Ovo je zato što kada se auto sačuva, njemu pridruženi objekti (motor u ovom slučaju) se takođe pamte zbog opcije `cascade = "all"` navedene u datoteci preslikavanja.

Modelovanje veze tipa “jedan prema više” Odnos između automobila i salona automobila jeste odnos tipa “jedan prema više”: svaki automobil pripada tačno jednom salonu automobila, dok jedan salon automobila sadrži više automobila. Pretpostavimo za početak da klasa `Automobil` nema nikakvu referencu na objekat klase `Salon` (tj. da je ovaj odnos jednosmeran). Podsetimo se definicija odgovarajućih klasa:

```
public class Automobil {
    private int id;
    private String proizvođač = null;
    private String boja = null;
    ...
}

public class Salon {
    private int id = 0;
    private String menadžer = null;
    private String lokacija = null;
    private Set<Automobil> automobili = null;
    ...
}
```

Pre nego što razmotrimo kako bi izgledale XML datoteke preslikavanja, pogledajmo skriptove za pravljenje odgovarajućih tabela. Setimo se, tabela `Automobil` će sadržati strani ključ koji odgovara primarnom ključu tabele `Salon`:

```
CREATE TABLE Salon_1premaviše (
    ID_salona integer NOT NULL,
    lokacija varchar(255) DEFAULT NULL,
    menadžer varchar(255) DEFAULT NULL,
    PRIMARY KEY (ID_salona)
)

CREATE TABLE Automobil_1premaviše (
    ID_automobila integer NOT NULL,
    boja varchar(255) DEFAULT NULL,
    proizvođač varchar(255) DEFAULT NULL,
    ID_salona integer DEFAULT NULL,
    PRIMARY KEY (ID_automobila),
    FOREIGN KEY (ID_salona) REFERENCES Salon_1premaviše (ID_salona)
)
```

Sledi odgovarajuća datoteka preslikavanja (definicija preslikavanja za klasu Automobil je pravolinijska, dok se u definiciji preslikavanja klase Salon vodi računa o vezi između ove dve klase):

```
<hibernate-mapping>
  <class name = "Automobil" table = "Automobil_1premavise">
    <id name = "id" column = "ID_automobila">
      <generator class = "assigned"/>
    </id>
    <property name = "boja" column = "boja" />
    ...
  </class>
  <class name = "Salon" table = "Salon_1premavise">
    <id name = "id" column = "ID_salona" >
      <generator class = "assigned"/>
    </id>
    <property name = "menadzer" column = "menadzer" />
    ...
    <set name = "automobili" table = "Automobil_1premavise"
      cascade = "all">
      <key column = "ID_automobila" not-null = "true"/>
      <one-to-many class = "Automobil"/>
    </set>
  </class>
</hibernate-mapping>
```

Definicijama preslikavanja je zadato da za svaki salon automobila možemo izdvojiti skup automobila iz tabele Automobil_1premavise koji je povezan sa klasom Automobil, na osnovu ključa ID_automobila. Ovo odgovara primeru koji smo razmatrali kada smo govorili o preslikavanju skupova. S obzirom na to da za neusmerene "jedan prema više" veze važi da vrednost stranog ključa može biti NULL, a da je kolona deklarirana kao NOT NULL, potrebno je eksplicitno zadati opciju not-null = "true".

U ovom primeru, s obzirom na to da relacija nije dvosmerna, nismo u mogućnosti da na osnovu podataka o automobilu izdvojimo salon kome on pripada. Implementiranje dvosmerne relacije zahteva neke male izmene u klasama i preslikavanjima. Najpre je potrebno dodati referencu na objekat Salon u klasi Automobil:

```
public class Automobil {
  private int id;
  private String proizvođač = null;
  private String boja = null;
  ...
  private Salon salon = null;
  public Salon getSalon(){
    return salon;
  }
  public void setSalon(Salon salon){
    this.salon = salon;
  }
}
```



```

    }
    // ostale set i get metode
}

```

Druga izmena bi se odnosila na preslikavanje klase `Automobil`; sada bi i ovo preslikavanje sadržalo element preslikavanja tipa “jedan prema više”:

```

<hibernate-mapping>
  <class name = "Automobil" table = "Automobil_1premavise">
    <id name = "id" column = "ID_automobil">
      <generator class = "assigned"/>
    </id>
    <property name = "boja" column = "boja" />
    <many-to-one name = "salon" column = "ID_salona" class = "Salon">
    </class>
</hibernate-mapping>

```

Element `<many-to-one>` ukazuje na stranu “više” veze između automobila i salona – odnosno ukazuje na to da više automobila može imati istu vrednost polja `salon`. Sada, ne samo da možemo da dobijemo sve automobile jednog salona automobila, već je moguće dobiti i salon automobila u kome se automobil izlaže, pozivom metoda: `automobil.getSalon()`. Ovim se uspostavlja dvosmerna veza između automobila i salona.

Modelovanje veze tipa “više prema više” Veza tipa “više prema više” uspostavlja odnos između dve klase na taj način da svaka od klasa sadrži potencijalno više instanci druge klase. U slučaju klasa `Student` i `Kurs`, odnos je tipa “više prema više” jer student može da upiše više kurseva, a takođe jedan isti kurs može da upiše više studenata. U programskom jeziku Java se ovaj odnos realizuje tako što obe strane imaju atribut koji je tipa kolekcija objekata druge klase. U bazama podataka se ovaj odnos najčešće realizuje korišćenjem vezne tabele.

Pretpostavimo da klase `Student` i `Kurs` imaju sledeći format:

```

public class Student {
    private int id = 0;
    private String ime = null;
    private Set<Kurs> kursevi = null;
    ...
}

public class Kurs {
    private int id = 0;
    private String proizvođač = null;
    private Set<Student> studenti = null;
    ...
}

```

Odgovarajuća preslikavanja bila bi oblika:

```
<hibernate-mapping>
  <class name = "Student" table = "Student">
    ...
    <set name = "kursevi" table = "Student_kurs" cascade = "all">
      <key column = "ID_studenta" />
      <many-to-many column = "ID_kursa" class = "Kurs"/>
    </set>
  </class>
  <class name = "Kurs" table = "Kurs">
    ...
    <set name = "studenti" table = "Student_kurs" inverse = "true">
      <key column = "ID_kursa" />
      <many-to-many column = "ID_studenta" class = "Student"/>
    </set>
  </class>
</hibernate-mapping>
```

Kao što smo već više puta pomenuli, u okruženju Hibernate je samo jedna strana odgovorna za održavanje dvosmerne veze (izmenu vrednosti kolona kojima se uspostavlja veza). Dakle, u okviru dvosmernih veza potrebno je zadati ključnu reč *inverse* kojom se zadaje koja strana veze je odgovorna za nju: opcija *inverse = "false"* znači da je ova strana odgovorna za vezu, odnosno da će dodavanje, brisanje ili ažuriranje kurseva za nekog studenta trajno sačuvati informacije o vezama studenta i kurseva u veznoj tabeli. Opcija *inverse = "true"* znači da ova strana nije odgovorna za vezu, tj. da je ovo inverzna, pridružena strana veze. Drugim rečima ovo znači da nećemo dodavati, brisati ili ažurirati kolekcije studenata za konkretan kurs, a i da to uradimo Hibernate neće trajno sačuvati ove informacije. Podrazumevana vrednost atributa *inverse* je *false*, te iz tog razloga nismo naveli ovu opciju u preslikavanju klase *Student*. Ovo odgovara opciji *mappedBy* kod anotacija.

Dodatno, potrebno je napraviti veznu tabelu:

```
CREATE TABLE Student_kurs (
  ID_kursa integer NOT NULL,
  ID_studenta integer NOT NULL,
  PRIMARY KEY (ID_kursa, ID_studenta),
  FOREIGN KEY (ID_kursa) REFERENCES Kurs (ID_kursa),
  FOREIGN KEY (ID_studenta) REFERENCES Student (ID_studenta)
)
```

Preslikavanje putem anotacija

Modelovanje veze tipa “jedan prema jedan” Vezu tipa “jedan prema jedan” između klasa *Automobil* i *Motor* možemo preslikati korišćenjem anotacija na sledeći način:

```
@Entity
@Table(name = "Automobil_1prema1_an")
public class Automobil {
  @Id
  @Column(name = "ID_automobila")
```

```

    @GeneratedValue(strategy = GenerationType.AUTO)
    private int id;
    private String proizvođjac = null;

    @OneToOne(cascade = CascadeType.ALL)
    @JoinColumn(name = "ID_motora")
    private Motor motor = null;
    ...
}

```

Preslikavanje tipa “jedan prema jedan” se deklarira korišćenjem anotacije @OneToOne. Anotacija @JoinColumn u kombinaciji sa anotacijom @OneToOne ukazuje na to da data kolona u entitetu koji je vlasnik veze referira na primarni ključ u pridruženoj strani veze.

Klasu Motor je potrebno anotirati na sledeći način:

```

@Entity
@Table(name = "Motor_1prema1_an")
public class Motor {
    @Id
    @Column(name = "ID_motora")
    @GeneratedValue(strategy = GenerationType.AUTO)
    private int id = 0;

    @OneToOne(mappedBy = "motor")
    private Automobil automobil = null;
    ...
}

```

S obzirom na to da je veza dvosmerna i da se može obići u oba smera, potrebno je postaviti koja strana veze će biti odgovorna za tu vezu. Atribut mappedBy anotacije @OneToOne (ili analogno @OneToMany ili @ManyToMany) se koristi za definisanje polja entiteta koje će biti odgovorno za održavanje veze. Ovaj atribut je neophodno navesti za dvosmerne veze i to u okviru pridruženog entiteta. On predstavlja analogon atributu inverse u XML preslikavanjima. mappedBy ima sledeće značenje: izmene na ovoj strani veze su već mapirane na drugoj strani veze, te ih nema potrebe pratiti u vidu zasebne tabele. U objektnom modelu ovo odgovara tome da postoje dve reference na svakoj od strana veze i one predstavljaju istu vezu, međutim, potrebno je znati koju od ove dve reference treba trajno pamtit i koju ne.

Modelovanje veze tipa “više prema više” Ako koristimo anotacije, potrebno je vezu tipa “više prema više” anotirati tako što se u jednoj od klasa (npr. Student) doda anotacija oblika:

```

@ManyToMany
@JoinTable
(name = "Student_kurs",
 joinColumns = @JoinColumn(name = "ID_studenta")
)
@Cascade(CascadeType.ALL)
private Set<Kurs> kursevi = null;

```

a u klasi Kurs svojstvo:

```
@ManyToMany(mappedBy = "kursevi")  
private Set<Student> studenti = null;
```

Anotacija @JoinTable ukazuje na to da će biti kreirana spojna tabela i najčešće se koristi za preslikavanje asocijacija tipa “više prema više” i jednosmerne “jedan prema više” veze, mada se može koristiti i za ostale tipove asocijacija. Ova anotacija se primenjuje na entitet koji zadužen za održavanje ove asocijacije. Atributom joinColumns se postavlja kolona stranog ključa koja referiše na primarni ključ tabele entiteta koji je odgovoran za održavanje asocijacije. Moguće je atributom inverseJoinColumns zadati kolone stranog ključa entiteta koje referišu na primarni ključ entiteta koji je pridružen asocijaciji (ali nije odgovoran za nju): ovde bi to bila klasa Automobil.

Modelovanje veze tipa “jedan prema više” Dvosmernu vezu tipa “jedan prema više” između salona automobila i automobila možemo preslikati korišćenjem anotacija analogno preslikavanju veze tipa “više prema više” samo bismo koristili anotacije @OneToMany i @ManyToOne.

Hibernate Query Language (HQL)

Hibernate Query Language (skraćeno HQL) je upitni jezik specijalno dizajniran za zadavanje upita nad Java objektima, nastao sa idejom da poveže koncepte Java klase i tabela baza podataka. To je jednostavan upitni jezik, nalik SQL-u, koji predstavlja deo Hibernate biblioteke. On podržava sve operacije relacionog modela (čak omogućava i pisanje čistih SQL upita), ali za razliku od SQL-a HQL podržava koncepte asocijacija, nasleđivanja i polimorfizma i rezultat upita vraća u vidu objekata.

HQL je inspirisao nastanak upitnog jezika Java Persistence Query Language (skraćeno JPQL) iste namene koji je deo JPA standarda. JPA ne predstavlja alat niti okruženje, već definiše skup koncepata koji mogu biti implementirani u proizvoljnom alatu ili okruženju. Jezik JPQL predstavlja podskup HQL-a, dakle svaki JPQL upit predstavlja ispravan HQL upit, dok obratno ne važi. Mi ćemo u nastavku izložiti osnovne koncepte jezika HQL.

SQL se koristi za formulisanje upita nad relacionim tabelama. Ipak, namera alata Hibernate bila je da se kreira jednostavan upitni jezik za ugodan rad sa objektnim modelom. Pritom bi on trebalo da bude što sličniji SQL-u. Na primer, sve redove tabele Knjige bismo u SQL-u izdvojili upitom “SELECT * FROM Knjige”, dok bi isti upit u HQL-u glasio “FROM Knjiga”. Primećimo da se u ovim upitima razlikuje naziv objekta iz kog izdvajamo redove. Naime, s obzirom na to da u HQL-u radimo sa objektima, potrebno je umesto naziva tabele navesti ime klasnog entiteta kojim je predstavljena tabela (u ovom primeru Knjiga je naziv trajnog Java entiteta koji se preslikava u tabelu Knjige). Ime klasnog entiteta je podrazumevano jednako imenu klase, a ako želimo da drugačije imenujemo entitet koji će se koristiti u HQL upitima, to možemo postići postavljanjem vrednosti atributa name na željeno ime u anotaciji @Entity. Mi ćemo u daljem tekstu smatrati da je ime entiteta jednak imenu klase i koristićemo naziv klase u HQL upitima. Ukoliko izdvajamo

sve kolone tabele, u HQL-u nije neophodno navesti ključnu reč SELECT. Ipak, moramo je navesti ako želimo da izdvojimo jednu ili više pojedinačnih kolona.

U Hibernate okruženju postoji Query API za rad sa objektno-relacionim upitima. Klasa Query, koja ima jednostavan interfejs, predstavlja centralni deo API-a. Očigledno, prva stvar koja nam je potrebna jeste instanca klase Query. Nju dobijamo pozivom metoda `createQuery` na tekućem objektu sesije i nad njom možemo pozivati različite metode kako bismo dobili željeni rezultujući skup iz baze podataka. Naredni fragment kôda ilustruje na koji način možemo dobiti instancu Query:

```
// otvaramo novu sesiju
Session sesija = fabrikaSesija.openSession();
// instanca upita se pravi iz ove sesije
Query upit = sesija.createQuery("FROM Knjiga");
```

Metod `createQuery` prihvata stringovnu reprezentaciju upita koji želimo da izvršimo nad bazom podataka i pravi instancu klase Query. Hibernate nakon toga transformiše prosleđeni HQL upit u odgovarajući SQL upit (s obzirom na to da relacione baze podataka razumeju samo SQL upite) i izvršava ga da bi izdvojio sve redove tabele Knjiga. Ovo je slično korišćenju odgovarajućeg SQL upita "SELECT * FROM Knjige", ali postoji nekoliko suptilnih razlika:

- za razliku od SQL-a, u HQL-u se ključna reč SELECT ne piše (tj. opcionalno) ako izdvajamo celu tabelu;
- dok se u SQL naredbama javljaju relacione tabele, HQL koristi nazive klase.

Sa izuzetkom imena Java klase i njihovih polja, u HQL upitima se ne pravi razlika između malih i velikih slova (nisu "case-sensitive").

Nije neophodno ali je poželjno metodi `createQuery` proslediti klasu koja predstavlja entitet rezultata, a upitu proslediti tip rezultata navođenjem klase koja predstavlja entitet rezultata u uglastim zagradama. Na primer prethodni primer bismo zapisali kao:

```
Query<Knjiga> upit = sesija.createQuery("FROM Knjiga", Knjiga.class);
```

Ispisivanje svih redova tabele može se uraditi na sledeći način:

```
private void izdvojiSveKnjige() {
    Query<Knjiga> upit = sesija.createQuery("FROM Knjiga", Knjiga.class);
    List<Knjiga> knjige = upit.list();
    for (Knjiga knjiga : knjige) {
        System.out.println("Knjiga: " + knjiga);
    }
    ...
}
```

Dakle, kada kreiramo instancu upita sa ugrađenim odgovarajućim HQL upitom, pozivamo metod `list` kojim se upit izvršava i njime se izdvajaju svi redovi u rezultatu upita. Hibernate iza scene transformiše sve redove tabele `Knjige` u instance klase `Knjiga` i izdvaja ih korišćenjem kolekcije `java.util.List`. S obzirom na to da je ovo standardna Java kolekcija, na uobičajen način korišćenjem `for` petlje iteriramo kroz nju da bismo izdvojili pojedinačne knjige.

Uporedimo ovu operaciju sa odgovarajućim JDBC kôdom:

```
private void izdvojiKnjige() {
    List<Knjiga> knjige = new ArrayList<Knjiga>();
    Knjiga k = null;
    try {
        Statement st = getConnection().createStatement();
        ResultSet rs = st.executeQuery("SELECT * FROM Knjige");
        while (rs.next()) {
            k = new Knjiga();
            k.setId(rs.getInt("id"));
            k.setNaziv(rs.getString("naziv"));
            knjige.add(k);
        }
    } catch (SQLException ex) {
        System.err.println(ex.getMessage());
    }
}
```

Najpre pravimo naredbu i izvršavamo upit nad njom, a kao rezultat dobijamo instancu klase `ResultSet`. Ona se sastoji od redova, ali oni nisu u objektnom formatu, već u sirovoj formi, te je potrebno proći kroz svaku instancu da bismo izdvojili svaki pojedinačni red u rezultatu upita; zatim je iz svakog pojedinačnog reda potrebno izdvojiti kolone, bilo prema njihovom nazivu ili lokaciji. Iz ovako ekstrahovanih kolona pravimo objekat klase `Knjiga` i dodajemo ga kolekciji `knjiga`. U Hibernate-u smo sve ove korake apstrahovali u par linija kôda. Šta više, može biti i samo jedna linija kôda ukoliko vršimo nadovezivanje metoda:

```
List<Knjiga> knjige =
    sesija.createQuery("FROM Knjiga").list();
```

Ukoliko želimo da izdvojimo samo određeni broj redova u rezultatu upita, to je moguće postići pozivom metoda `setMaxResults` koji postavlja maksimalan broj redova koji treba izdvojiti, na primer:

```
Query<Knjiga> upit = sesija.createQuery("FROM Knjiga", Knjiga.class);
upit.setMaxResults(100);
upit.setFirstResult(10);
List<Knjiga> knjige = upit.list();
```

Metod `setMaxResults` stavlja do znanja Hibernate-u da smo zainteresovani da vidimo samo 100 redova. Metodom `setFirstResult` postavlja se početna pozicija rezultujućeg skupa. Na primer, prethodni kod izdvaja 100 redova iz tabele, počev od 10. reda.

Metod `list` primenjen na upit vraća listu, koja ima pridruženi iterator. Iteratori predstavljaju integralni deo Java kolekcija koji obezbeđuje mogućnost iteriranja kroz elemente kolekcije. Da bismo dobili iterator, potrebno je pozvati metod `iterator` nad listom, kao u narednom fragmentu koda:

```
Query<Knjiga> upit = sesija.createQuery("FROM Knjiga", Knjiga.class);
Iterator iteratorUpita = upit.list().iterator();
while(iteratorUpita.hasNext()){
    Knjiga k = (Knjiga)iteratorUpita.next();
    System.out.println("Knjiga:" + k);
}
```

Metodom `hasNext` proverava se da li u kolekciji postoji još elemenata, dok metod `next` vraća naredni element kolekcije.

Kao i u SQL-u stavkom `WHERE` možemo precizirati listu instanci koja se izdvaja. Na primer, možemo kreirati HQL upit:

```
FROM Knjiga WHERE naziv = 'Na Drini cuprija'
```

HQL podržava i predikat `IS NULL` te na sledeći način možemo izdvojiti sve knjige kojima je poznata godina izdavanja:

```
FROM Knjiga WHERE god_izdavanja IS NOT NULL
```

Zanimljiva je i naredna mogućnost: ukoliko bi klasa `Knjiga` sadržala referencu na instancu klase `Izdavac` (koja između ostalog sadrži i polje `adresa`), ispravan HQL upit bi mogao da glasi:

```
FROM Knjiga WHERE izdavac.adresa IS NOT NULL
```

Ovo bi odgovaralo SQL upitu koji koristi spajanje tabela `Knjiga` i `Izdavac`.

Kada znamo da postoji jedan i samo jedan red u rezultatu upita, možemo pozvati metod `uniqueResult`. Pretpostavimo da želimo da izdvojimo knjigu "Prokleta avlija" pod pretpostavkom da postoji samo jedna knjiga sa ovim nazivom. Ovo je moguće uraditi narednim upitom:

```
private void izdvojiKnjigaJedinstvena() {
    ...
    Query<Knjiga> upit = sesija.createQuery
        ("FROM Knjiga WHERE naziv = 'Prokleta Avlija'", Knjiga.class);
    Knjiga knjiga = (Knjiga) upit.uniqueResult();
}
```

Metod `uniqueResult` će ukoliko ne postoji odgovarajući red vratiti vrednost `null`, a ako postoji više redova u rezultatu biće izbačen izuzetak.

Imenovani parametri

U prethodnom HQL upitu, direktno smo u kod upisali naziv 'Prokleta avlija'. Direktno kodiranje ulaznih kriterijuma nije najbolji izbor jer je poželjno podržati mogućnost da se ovaj kriterijum može izmeniti. Stoga je zgodno parametrizovati ovakve stvari.

HQL podržava zadavanje imenovanih parametara. Ulazne parametre objekta Query moguće je postaviti na sledeći način:

```
private void izdvojiKnjigeParametarski(String knjiga) {
    ...
    Query<Knjiga> upit = sesija.createQuery(
        "FROM Knjiga WHERE naziv = :nazivId", Knjiga.class);
    upit.setParameter("nazivId", knjiga);
    List<Knjiga> knjige = upit.list();
}
```

U prethodnom primeru, nazivId je promenljiva zadužena za čuvanje vrednosti ulaznog parametra HQL upita – naziva knjige. Imenima parametara u HQL upitima prethodi dvotačka. Vrednosti vezujemo za parametre putem njihovog naziva, korišćenjem metoda setParameter koji kao prvi argument prima naziv imenovanog parametra, a kao drugi njegovu vrednost.

Moguće je u jednom upitu koristiti više imenovanih parametara:

```
Query<Knjiga> upit = sesija.createQuery(
    "FROM Knjiga WHERE naziv = :nazivId AND k_sifra = :IdKnjige", Knjiga.class);
upit.setParameter("nazivId", "Prokleta Avlija");
upit.setParameter("IdKnjige", 102);
```

Podržana je i varijanta metoda setParameter sa tri argumenta, čiji je treći argument tip parametra.

Nekada je potrebno izdvojiti podatke koji zadovoljavaju kriterijum da neka vrednost pripada datoj listi vrednosti. Na primer, želimo da izdvojimo knjige čiji je izdavač "Prosveta" ili "Sluzbeni glasnik". U toj situaciji možemo iskoristiti HQL-ovu klauzu IN, koja je ekvivalentna korišćenju SQL-ovog operatora IN. Listu vrednosti pravimo korišćenjem standardne Java kolekcije ArrayList. Nakon toga možemo pozvati metod setParameterList nad upitom, koji prihvata listu vrednosti kojom treba popuniti datu parametarsku listu:

```
private void izdvojiKnjigeParametarskaLista() {
    ...
    // definisemo listu i popunjavamo je nasim kriterijumima
    List izdavaci = new ArrayList();
    izdavaci.add("Prosveta");
    izdavaci.add("Sluzbeni glasnik");

    // pravimo upit
    Query<Knjiga> upit = sesija.createQuery("FROM Knjiga
        WHERE izdovac IN (:listaIzdavaca)", Knjiga.class);
```



```

    // vezujemo konkretnu listu za parametarsku listu
    upit.setParameterList("listaIzdavaca", izdavaci);
    List<Knjiga> knjige = upit.list();
    ...
}

```

Dakle, za parametre možemo vezati i Java kolekcije, a ne samo promenljive osnovnih tipova.

Hibernate transformiše ovaj upit u ekvivalentan SQL upit:

```

SELECT * FROM Knjige WHERE izdavac IN
('Prosveta','Sluzbeni glasnik')

```

Ključna reč SELECT

Ukoliko upit u HQL-u želimo da izdvojimo odabrane kolone iz tabele baze podataka (umesto kompletnih redova) moramo koristiti ključnu reč SELECT.

Na primer, ako želimo da izdvojimo samo nazive knjiga (a ne sve kolone) za sve knjige iz tabele, mogli bismo to uraditi na sledeći način:

```

Query<Knjiga> upit = sesija.createQuery
    ("SELECT k.naziv FROM Knjiga AS k", Knjiga.class);

List<String> nazivi = upit.list();
System.out.println("Nazivi knjiga:");

for (String naziv : nazivi) {
    System.out.println("\t" + naziv);
}

```

S obzirom na to da izdvajamo samo jednu kolonu, poznat je očekivani tip te kolone. Šta se dešava ako upit izdvaja više kolona – na koji način se onda vraća rezultujući skup? U situaciji kada se izdvaja n kolona, rezultujući skup vraća se u vidu n -torki, kao u narednom primeru:

```

private void izdvojiKnjigeSaTorkama() {
    ...
    String upitViseKolona =
        "SELECT k.naziv, k.izdavac FROM Knjiga as k";
    Query<Object[]> upit = sesija.createQuery(upitViseKolona,
        Object[].class);

    // izdvajamo knjige
    Iterator it = upit.list().iterator();
    while(it.hasNext()){
        Object[] o = (Object[])it.next();
        System.out.print("Naziv: " + o[0] + "\t");
        System.out.println("Izdavac: " + o[1]);
    }
}

```

Lista izdvojenih elemenata se prosleđuje kao niz objekata. Naziv i izdavač se stoga izdvajaju kao prvi i drugi element niza.

Rad sa *n-torkama* nije najelegantniji. Efikasnija alternativa je korišćenje još jedne mogućnosti Hibernate-a, a to je prebacivanje rezultata u domenski objekat. Pretpostavimo da klasa *Knjiga* ima definisan konstruktor za prosleđene vrednosti polja naziv i izdavač. Moguće je kreirati instancu svakog reda podataka koji izdvajamo na sledeći način:

```
String upit = "SELECT new Knjiga(k.naziv, k.izdavac)
              FROM Knjiga as k";

// izdvajamo knjige
List<Knjiga> knjige = sesija.createQuery(upit).list();
for (Knjiga knjiga : knjige) {
    System.out.println("Knjiga: " + knjiga);
}
```

Obratimo pažnju na instanciranje objekta *Knjiga* u samom upitu.

Spajanje tabela

U HQL-u je dozvoljeno uvoditi nova imena za tabele - tzv. alijase. Oni su posebno korisni prilikom spajanja tabela ili korišćenja podupita. Na primer:

```
FROM Knjiga AS k WHERE k.naziv = :naziv AND k.k_sifra = :id_knjige"
```

Ključna reč *AS* je opcionalna.

HQL podržava operatore spajanja i to:

- INNER JOIN,
- LEFT OUTER JOIN i
- RIGHT OUTER JOIN.

Pritom se ne navode uslovi spajanja, ali je neophodno da između entiteta bude implementirana veza (asocijacija). Na primer, ako u klasi *Knjiga* postoji polje *bib* koje odgovara biblioteci u kojoj se nalazi data knjiga, onda se vrši spajanje preko ovog polja. Neke dodatne uslove moguće je navesti ključnom rečju *WITH*. Npr. HQL upit bi mogao da glasi:

```
FROM Knjiga AS k
INNER JOIN k.bib AS b
WITH k.izdavac = 'Prosveta'
```

Ukoliko želimo da izdvojimo samo neka od polja u rezultatu spajanja potrebno je navesti ključnu reč *SELECT* za kojom slede nazivi polja odgovarajućih klasa. U suprotnom, rezultat će biti tipa *List<Object[]>*, gde je svaki element liste jedan od objekata klasa koje učestvuju u spajanju.

Dekartov proizvod dve tabele se može dobiti navođenjem obe odgovarajuće klase u *FROM* stavci:

```
FROM Knjiga AS k, Biblioteka AS b
```

Pored eksplicitnog spajanja, HQL podržava i implicitno spajanje, korišćenjem operatora . (tačka) koji smo ranije videli:

```
FROM Knjiga AS k WHERE k.autor.adresa = "Beograd"
```

Agregatne funkcije

HQL podržava agregatne funkcije poput AVG, MIN, MAX, COUNT(*), COUNT(...), COUNT(DISTINCT ...) koje su ekvivalentne odgovarajućim funkcijama u SQL-u. Na primer, ako bi u tabeli Knjige bila prisutna informacija o preporučenoj ceni knjige, maksimalnu cenu knjige bismo mogli da izdvojimo na sledeći način:

```
List lista = sesija.createQuery("SELECT max(cena) FROM Knjiga")
    .list();
int maxCena = ((Integer)lista.get(0)).intValue();
```

Nakon izdvajanja rezultata u vidu liste, potrebno je prvi (i jedini) element liste transformisati u celobrojnu vrednost.

Polimorfni upiti

Pomenimo i to da upit

```
FROM Knjiga as k
```

vraća instance ne samo klase Knjiga, već i svih potklasa ove klase. Tako bi recimo bilo moguće izdvojiti sve trajne objekte upitom

```
FROM java.lang.Object o
```

Ažuriranje, brisanje i umetanje

Podatke je moguće ažurirati i obrisati pozivom metoda executeUpdate. Ovaj metod kao argument očekuje string koji sadrži odgovarajući upit, kao u narednom primeru:

```
String update_upit = "UPDATE Knjiga SET naziv = :naziv
    WHERE k_sifra = 102";
Query upit = sesija.createQuery(update_upit);
upit.setParameter("naziv", "Don Kihot");
int uspeh = upit.executeUpdate();
```

Prethodni upit koristi imenovani parametar za postavljanje naziva sloga čija je šifra 102. Metod executeUpdate ažurira tabelu i vraća ceo broj kao indikator uspešnosti operacije ažuriranja. Povratna vrednost sadrži broj redova koji su pogođeni ovom operacijom. Isti metod se može iskoristiti i za izvršavanje naredbe brisanja:

```
String delete_upit = "DELETE FROM Knjiga WHERE id = 106";
Query upit = sesija.createQuery(delete_upit);
int uspeh = upit.executeUpdate();
```

Što se operacije umetanja podataka tiče, nije moguće umetnuti entitet koji ima proizvoljne vrednosti (kao što je to bilo moguće klauzom `VALUES` u SQL-u), već je jedino moguće umetnuti entitete formirane na osnovu informacija dobijenih naredbom `SELECT`. Dakle, mogućnost primene ove operacije je dosta manja nego što je to bio slučaj u SQL-u. Sintaksa HQL naredbe `INSERT` je sledeća: nakon klauze `INSERT INTO` navodi se naziv entiteta za kojim u zagradi slede nazivi polja tog entiteta, a nakon toga ide odgovarajuća HQL naredba `SELECT`. Na primer:

```
String insert_upit =
    "INSERT INTO Knjiga_backup(id_knjige, naziv, izdavac, god_izdavanja)
    SELECT id_knjige, naziv, izdavac, god_izdavanja
    FROM Knjiga WHERE god_izdavanja = 1990";
Query upit = sesija.createQuery(insert_upit);
int uspeh = upit.executeUpdate();
```

Imenovani upiti

U svim dosadašnjim primerima upite smo formulisali u samom kodu. Direktno kodiranje upita u kodu nije najbolja praksa. Bolje rešenje predstavlja korišćenje *imenovanih upita* kojim se upiti grupišu na jednom mestu, a koriste kasnije u kodu gde god je potrebno. Na ovaj način dobija se čistiji kod, upiti se mogu koristiti veći broj puta sa različitih mesta u kodu i njihova ispravnost se proverava prilikom pravljenja fabrike sesija. Postoje dva načina na koja možemo kodirati imenovane upite: možemo koristiti anotaciju `@NamedQuery` da vežemo upite za entitete na nivou klasa ili ih možemo deklarirati u datotekama preslikavanja.

U pristupu kada koristimo anotacije, entitetu `Knjiga` potrebno je dodati anotaciju `@NamedQuery`. Ova anotacija prihvata naziv upita i sam upit kao u narednom primeru:

```
@Entity
@NamedQuery(name = "Izdvoji_knjige",
            query = "FROM Knjiga")
public class Knjiga{
    ...
}
```

Jednom kada smo definisali imenovani upit preko anotacije, njemu se može pristupiti iz sesije u vreme izvršavanja pozivom metoda `getNamedQuery` koji kao argument prima naziv imenovanog upita.

```
private void koriscenjeImenovanihUpita() {
    ...
    // pristupamo predefinisanoj imenovanoj upitu
    Query upit = sesija.getNamedQuery("Izdvoji_knjige");
    List knjige = upit.list();
}
```

Za jedan isti entitet moguće je vezati više različitih upita, tako što definišemo roditeljsku anotaciju `@NamedQueries` koja kao vrednost atributa `value` sadrži niz definicija oblika `@NamedQuery`.

```

@Entity
@NamedQueries(
    value = {
        @NamedQuery(name = "Izdvoji_knjige",
            query = "FROM Knjiga"),
        @NamedQuery(name = "Izdvoji_knjige_za_naziv",
            query = "FROM Knjiga WHERE naziv = :naziv")
    }
)
public class Knjiga{ ... }

```

Ako koristimo preslikavanja zadata u vidu XML datoteka, potrebno je definisati upite u datotekama preslikavanja. Na primer, definišemo upite koji su vezani za knjige u datoteci `Knjiga.hbm.xml` na sledeći način:

```

<hibernate-mapping>
    <class name = "Knjiga" table = "Knjige">
        ...
    </class>

    <!-- Ovde definisemo upite vezane za entitet Knjiga -->
    <query name = "Izdvoji_knjige">
        <![CDATA[ FROM Knjiga ]]>
    </query>
    <query name = "Izdvoji_knjige_za_naziv">
        <![CDATA[ FROM Knjiga WHERE naziv = :naziv ]]>
    </query>
</hibernate-mapping>

```

Dobra praksa je “omotati” tekst upita u CDATA, da XML parser ne bi izbacio grešku zbog korišćenja rezervisanih XML karaktera u tekstu upita poput karaktera ‘>’ i ‘<’.

Ukoliko se upit odnosi na veći broj entiteta onda je takav upit potrebno zadati u zasebnoj datoteci u paketu koji je deo aplikacije, prilikom pravljenja fabrike sesija potrebno je, pored anotiranih klasa, dodati i naziv paketa i onda je u aplikaciji moguće na isti način koristiti imenovane upite.

Korišćenje “čistog” SQL-a

Hibernate takođe podržava mogućnost izvršavanja “čistih” SQL upita. Metod `createNativeQuery` vraća objekat klase `NativeQuery`, slično kao što `createQuery` vraća `Query` objekat. Ova klasa nasleđuje klasu `Query`. Iako je HQL pogodan za korišćenje nad Java objektima, postoje specijalni upiti koji mogu da zavise od funkcija specifičnih za proizvod baza podataka i tada je zgodno koristiti SQL. Primer korišćenja SQL-a dat je u nastavku:

```

NativeQuery upit = sesija.createNativeQuery("SELECT * FROM Knjige");
List knjige = upit.list();

```

Možemo primetiti da je tekst upita ispravna SQL naredba (na primer, postoji ključna reč `SELECT` na početku upita koja se preskače u slučaju HQL-a).

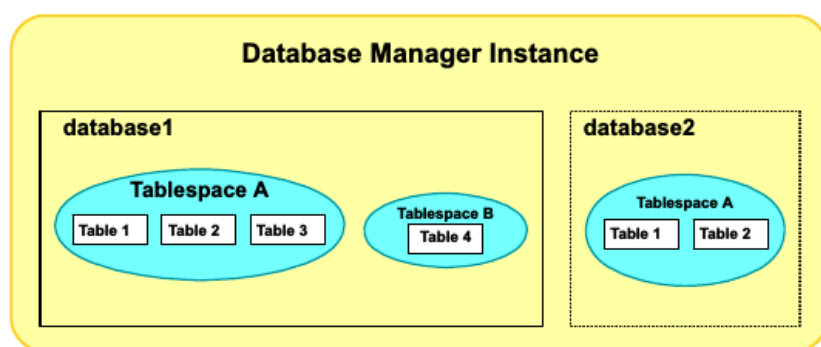
Kao što smo već pomenuli, upite u Hibernate okruženju je moguće deklarirati i van koda. Na primer, imenovane SQL upite možemo deklarirati dodavanjem anotacija `@NamedNativeQuery` i `NamedNativeQueries` na samom entitetu, ili deklarisanjem elementa `<sql-query>` u datoteci preslikavanja.

Glava 5

Administracija baza podataka

5.1 Kreiranje baze podataka

Prilikom implementacije baze podataka, postoje brojni faktori koje treba razmotriti, a koji se tiču fizičkog okruženja u kome će baza biti kreirana. Među njima su odabir instance u kojoj će baza podataka biti kreirana, veličina prostora potrebna za smeštanje podataka, informacije o tome gde će podaci biti fizički smešteni, kao i to koja vrsta prostora tabela će biti korišćena za smeštanje podataka.



Slika 5.1: Hijerarhijska organizacija instance menadžera baze podataka

Baze podataka se kreiraju unutar *instance menadžera baze podataka* (eng. Database Manager Instance). *Menadžer baze podataka* je aplikacija koja je deo SUBP-a i koja obezbeđuje osnovne funkcionalnosti rada sa bazama podataka, kao što su kreiranje, preimenovanje, brisanje i održavanje baze podataka, mogućnost pravljenja rezervnih kopija i povratak podataka.

Instanca je logičko okruženje menadžera baza podataka koje omogućava upravljanje bazama podataka. U okviru instance moguće je bazu podataka uneti u katalog (katalogizirati) i postaviti joj konfiguracione parametre. Moguće je kreirati više od jedne instance na istom fizičkom serveru (jednu instancu, na primer, možemo koristiti kao razvojno okruženje, a drugu kao okruženje za produkciju). Naredbom `db2ilist` listaju se sve instance koje su raspoložive

na sistemu. Naredbom `get instance` prikazuju se detalji o instanci koja je trenutno pokrenuta. Naredbom `set DB2INSTANCE = <naziv_instance>` može se izmeniti tekuća instanca, a ona se pokreće naredbom `db2start`. Tekuća instanca se može zaustaviti naredbom `db2stop`.

Prostori tabela su skladišne strukture koje sadrže tabele, indekse i velike objekte. Oni predstavljaju logički sloj između baze podataka i objekata koje se čuvaju u okviru baza podataka. Dakle, prostori tabela se prave unutar baze podataka, dok se tabele prave unutar prostora tabela. Prostori tabela se čuvaju u *grupama particionisanja* baze podataka koje predstavljaju skupove od jedne ili više particija koje pripadaju bazi podataka. U okviru svake instance moguće je definisati jednu ili više baza podataka, a svaka baza podataka može da sadrži veći broj prostora tabela. Svaka baza podataka pravi se sa tri podrazumevana *sistemska prostora tabela*:

- SYSCATSPACE: tabele sistemskog kataloga DB2;
- TEMPSPACE1: privremene sistemske tabele (koje recimo omogućavaju prostor za rad za sortiranje ili spajanje, s obzirom na to da ove aktivnosti zahtevaju dodatan prostor za obradu redova u rezultatu);
- USERSPACE1: inicijalni prostor tabela za definisanje korisničkih tabela i indeksa.

Naravno, moguće je praviti i dodatne prostore tabela za skladištenje tabela, indeksa i sl (slika 5.1).

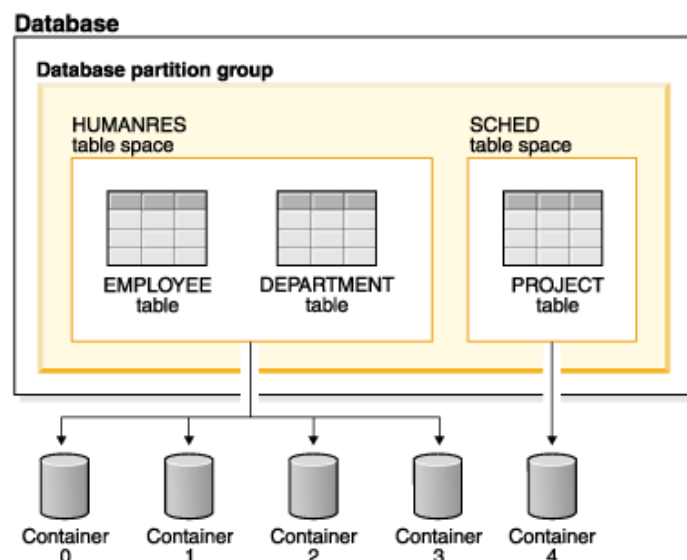
Prostori tabela se sastoje iz jednog ili većeg broja *kontejnera* (eng. container) koji mogu biti naziv direktorijuma, naziv uređaja ili naziv datoteke (slika 5.2). Kreiranjem prostora tabela unutar baze podataka se prostoru tabela dodeljuju kontejneri i pamte se njihove definicije i atributi u sistemskom katalogu baze podataka.

Za svaku bazu podataka potrebno je obezbediti skup kontrolnih datoteka baze podataka, kao što su datoteka konfiguracije baze podataka (koja sadrži informacije o kodnoj strani, uparivanju i slično), datoteka istorije oporavka (koja čuva informacije o svim izmenama nad bazom podataka i ažurira se nakon svakog kopiranja i oporavka baze podataka, nakon kreiranja / brisanja / menjanja prostora tabela i slično), skup kontrolnih log datoteka, i druge. Takođe, za svaku bazu podataka potreban je skup log datoteka baze podataka.

Podrazumevana vrednost putanje baze podataka definisana je parametrom `dftdbpath` u `DATABASE MANAGER CONFIGURATION`. Podrazumevano, to je home direktorijum vlasnika instance na Linux i UNIX sistemima, odnosno drjav na kome je instaliran DB2 na Windows sistemima. Analogno već viđenim naredbama `GET DATABASE CONFIGURATION` i `UPDATE DATABASE CONFIGURATION`, informacije o konfiguraciji menadžera baza podataka možemo dobiti i izmeniti naredbama:

- `GET DATABASE MANAGER CONFIGURATION` ili skraćeno `GET DBM CFG` i
- `UPDATE DATABASE MANAGER CONFIGURATION` ili skraćeno `UPDATE DBM CFG`.

Imenovani skup putanja za skladištenje tabela i drugih objekata baze podataka naziva se *grupom skladišta*. Prostori tabela se mogu dodeliti grupi



Slika 5.2: Hijerarhijska struktura baze podataka

skladišta. Prilikom kreiranja baze podataka, svi prostori tabela se prave u podrazumevanoj grupi koja je IBMSTOGROUP. Sve grupe skladišta moguće je izlistati naredbom:

```
SELECT * FROM SYSCAT.STOGROUPS
```

Nova grupa skladišta st1 na putanji /data1 kreira se naredbom:

```
CREATE STOGROUP stg1 ON '/data1'
```

Novi prostor tabela ts1 se može kreirati korišćenjem postojeće grupe skladišta stg1 naredbom:

```
CREATE TABLESPACE ts1 USING STOGROUP stg1
```

Sve prostore tabela možemo izlistati naredbom:

```
SELECT * FROM SYSCAT.TABLESPACES
```

Prostori tabela se mogu konfigurisati na različite načine, u zavisnosti od toga kako želimo da ih koristimo. Može se zadati da operativni sistem upravlja alokacijom prostora tabela, može se ostaviti menadžeru baze podataka da alokira prostor ili se može odabrati automatska alokacija prostora tabela za podatke.

DB2 podržava naredna tri tipa upravljanja prostorima:

- SMS (eng. System Managed Space) – menadžer fajl sistema operativnog sistema alokira i upravlja prostorom u kom je sačuvana tabela,

- DMS (eng. Database Managed Space) – menadžer baza podataka upravlja prostorom za skladištenje,
- automatsko skladištenje.

U SMS prostorima tabela, menadžer fajl sistema operativnog sistema alokira i upravlja prostorom u kome se čuvaju tabele. Model skladišta se obično sastoji od velikog broja datoteka koje predstavljaju objekte baze podataka i čuvaju se u prostoru fajl sistema. Prostor se alokira na zahtev. Korisnik odlučuje o lokaciji datoteka, DB2 vrši kontrolu njihovih imena, a fajl sistem je odgovoran za njihovo upravljanje. Prilikom brisanja tabele, pridružene datoteke se brišu te se prostor na disku oslobađa. Počev od verzije DB2 10.1 SMS prostori tabela smatraju se zastarelim za korisnički definisane prostore tabela, ali se i dalje koriste za prostore tabela sistemskog kataloga i za privremene prostore tabela.

U DMS prostorima tabela, menadžer baza podataka upravlja prostorom za skladištenje. Model skladištenja se sastoji od ograničenog broja uređaja ili datoteka. Administrator baze podataka odlučuje koji se uređaji i datoteke koriste i DB2 upravlja prostorom na ovim uređajima i datotekama. Kada se tabela obriše iz DMS prostora tabela, dodeljene stranice su na raspolaganju drugim objektima u tom prostoru tabela, ali se prostor na disku ne oslobađa. Prostor za skladištenje se alokira unapred prilikom pravljenja prostora tabela, dok se kod SMS prostora tabela, prostor alokira na zahtev. Od verzije 10.1 DMS prostori tabela smatraju se zastarelim za korisnički definisane prostore tabela, ali nije za sistemske i privremene prostore tabela.

Kod prostora tabela sa automatskim skladištenjem, prostorom se upravlja automatski. Menadžer baze podataka kreira i proširuje kontejnere po potrebi. Za razliku od druga dva tipa prostora tabela, u ovom slučaju nije neophodno navoditi definicije kontejnera: menadžer baze podataka se stara o kreiranju i proširivanju kontejnera da iskoristi prostor alocirani od strane baze podataka. Svakim prostorom tabela koji se kreira se upravlja automatski osim ako se ne navede drugačije ili ako je baza podataka kreirana sa opcijom `AUTOMATIC STORAGE NO`. U jednoj bazi podataka moguće je koristiti sva tri tipa prostora tabela.

Tip upravljanja skladištem se postavlja prilikom pravljenja prostora tabela naredbom

```
CREATE TABLESPACE <prostor_tabela>
```

U bazama podataka za koje nije omogućeno automatsko skladištenje, prilikom pravljenja prostora tabela neophodno je navesti stavku `MANAGED BY DATABASE` ili `MANAGED BY SYSTEM`. Korišćenje ovih stavki rezultuje u pravljenju DMS prostora tabela, odnosno SMS prostora tabela. Moguće je i eksplicitno navesti stavku `MANAGED BY AUTOMATIC STORAGE` čime se obezbeđuje da se vrši automatsko upravljanje prostorom tabela. Ako se stavka `MANAGED BY ...` u potpunosti izostavi onda se podrazumeva automatsko skladištenje, tako da je navođenje stavke `MANAGED BY AUTOMATIC STORAGE` opciono.

Prilikom kreiranja prostora tabela moguće je navesti i inicijalnu veličinu, vrednost za koliko će se uvećati veličina prostora tabela kada se prostor tabela popuni, maksimalna veličina do koje prostor tabele može da raste i tip upravljanja. Na primer:

```
CREATE TABLESPACE prostor_automatski INITIALSIZE 8K
  INCREASESIZE 20 PERCENT
  MAXSIZE 1G
  MANAGED BY AUTOMATIC STORAGE
```

ili:

```
CREATE TABLESPACE prostor_sms
  MANAGED BY SYSTEM USING ('d:\acc_tbsp')
```

ili

```
CREATE TABLESPACE prostor_dms
  MANAGED BY DATABASE USING (FILE 'd:\db2data\acc_tbsp' size)
```

Moguće je navesti i grupu skladišta, npr:

```
CREATE TABLESPACE prostor_auto
  MANAGED BY AUTOMATIC STORAGE
  USING STOGROUP STOGROUP1
```

Za svaku bazu podataka pravi se i održava skup tabela sistemskog kataloga. Ove tabele sadrže informacije o definicijama objekata baze podataka (kao što su tabele, pogledi, indeksi itd.), kao i informacije o vrstama pristupa koje korisnici imaju na ovim objektima. One se čuvaju u prostoru tabela SYSCATSPACE. Ove tabele se ažuriraju prilikom izvođenja operacija nad bazom podataka; na primer, kada se kreira tabela. Svi pogledi sistemskog kataloga se prave prilikom pravljenja baze podataka i oni se ne mogu eksplicitno praviti ili brisati, ali se mogu postavljati upiti nad njima i listati njihov sadržaj. Menadžer baze podataka je zadužen za pravljenje i održavanje skupa pogleda kataloga. Razlikujemo dva skupa pogleda. Prvi skup pogleda se nalazi u okviru sheme SYSCAT, ovi pogledi se mogu samo čitati i pravo SELECT nad ovim pogledima se podrazumevano daje svima (PUBLIC). Drugi skup pogleda se nalazi u okviru sheme SYSSTAT i on je formiran na osnovu podskupa pogleda koji se nalaze u okviru sheme SYSCAT. Ovaj pogled sadrži statističke informacije koje koristi SQL optimizator. Pogledi u okviru sheme SYSSTAT sadrže neke kolone čije je vrednosti moguće menjati.

Tabela 5.1: Primeri pogleda i odgovarajućih objekata

SYSCAT.TABLES	TABSCHEMA, TABNAME
SYSCAT.INDEXES	INDSCHEMA, INDNAME
SYSCAT.VIEWS	VIEWSCHEMA, VIEWNAME
SYSCAT.COLUMNS	COLNAME
SYSCAT.TRIGGERS	TRIGSCHEMA, TRIGNAME

Na primer možemo izvršiti upit:

```
SELECT TABNAME, TABSCHEMA, TBSPACE
FROM SYSCAT.TABLES
```

Prilikom kreiranja nove baze podataka, ona se automatski katalogizira u sistemskoj datoteci. Moguće je i eksplicitno katalogizirati bazu podataka pod drugim nazivom - aliasom naredbom `CATALOG DATABASE` ili ako je prethodno bila obrisana iz sistema naredbom `UNCATALOG DATABASE`.

Okruženje baza podataka može sadržati jednu ili veći broj particija. Za svaku instancu kreira se jedna konfiguraciona datoteka pod nazivom `db2nodes.cfg`. Ona sadrži po jedan red za svaku particiju baze podataka. Kod baza podataka koje koriste veći broj particija tabele se mogu čuvati na jednoj ili na većem broju particija (npr. neki redovi jedne tabele na jednoj, a drugi redovi na drugoj particiji). Pre kreiranja više-particione baze podataka, potrebno je odabrati koja particija će biti glavna i na njoj će biti čuvane sve tabele sistemskog kataloga, tako što ćemo kreirati bazu podataka direktno sa te particije baze podataka. Prilikom kreiranja baze podataka, ona se automatski kreira korišćenjem svih particija baze podataka definisanih u datoteci `db2nodes.cfg`.

Naredba CREATE DATABASE

Naredba `CREATE DATABASE` inicijalizuje novu bazu podataka, pravi tri inicijalna prostora tabela, kreira sistemske tabele i alokira log datoteku oporavka. Prilikom inicijalizacije nove baze podataka, podrazumevano se poziva naredba `AUTOCONFIGURE`, kojom se vrši automatsko podešavanje konfiguracionih parametara na osnovu raspoloživih resursa.

Na primer, naredbom:

```
CREATE DATABASE test1 ON '/data1','/data2' DBPATH ON '/dpath1'
```

pravimo bazu podataka sa imenom `test1` na putanji `/dpath1` korišćenjem putanja skladištenja `/data1` i `/data2` (definisanim na podrazumevanoj grupi skladišta `IBMSTOGROUP`).

Da bismo na Windows operativnom sistemu napravili bazu `test2` na drajvu `D:`, sa skladištem na `E:\data`, potrebno je pokrenuti naredbu:

```
CREATE DATABASE test2 ON 'E:\data' DBPATH ON 'D:'
```

U ovom primeru `E:\data` se koristi kao putanja skladišta definisana za podrazumevanu grupu skladišta `IBMSTOGROUP`.

Ako imamo naredbu:

```
CREATE DATABASE knjige1 ON /bpknjige1
```

putanja baze podataka za ovu bazu bila bi `/bpknjige1`. Baza podataka bi imala omogućeno automatsko skladištenje sa jednom automatskom putanjom skladišta `/bpknjige1`.

Ukoliko se ne navede stavka `ON`, baza podataka se kreira na podrazumevanoj putanji koja je zadata u konfiguracionoj datoteci menadžera baze podataka (parametar `dftdbpath`).

U primeru naredbe:

```
CREATE DATABASE knjige2
AUTOMATIC STORAGE NO
ON /bpknjige2
```

putanja baze podataka bila bi /bpknjige2. Za ovu bazu podataka bilo bi onemogućeno automatsko skladištenje i bilo bi korišćeno SMS upravljanje za tri sistemska prostora tabela

Ako bismo imali naredbu:

```
CREATE DATABASE knjige3
  AUTOMATIC STORAGE YES
  ON /bpauto31,/bpauto32,/bpauto33
  DBPATH ON /bpknjige3
```

putanja baze podataka bila bi /bpknjige3, za bazu bi bilo omogućeno automatsko skladištenje sa tri putanje automatskog skladišta: /bpauto31, /bpauto32, /bpauto33. Prostori tabela sa automatskim skladištenjem smanjuju zadatke održavanja koje korisnik treba da obavi.

U okviru naredbe CREATE DATABASE moguće je zadati kodiranje, zemlju i uparivanje koji će se koristiti za podatke unete u datu bazu podataka. U te svrhe potrebno je navesti stavku USING CODESET <kodni-skup> TERRITORY <zemlja> COLLATE USING <uparivanje>. Na primer, za Srbiju moguće je navesti:

```
CREATE DATABASE knjige4 AUTOMATIC STORAGE yes
  USING CODESET utf-8 TERRITORY rs
```

Počev od verzije DB2 9.5 ukoliko se ne navede kodna strana prilikom pravljenja baze podataka, podrazumevana kodna strana je UNICODE. Ako se eksplicitno zada kodiranje, onda je potrebno zadati i identifikator zemlje. Podrazumevana vrednost teritorije zavisi od lokacije klijenta koji pokreće naredbu CREATE DATABASE. Kodiranje i identifikator zemlje moraju biti kompatibilni. Nakon kreiranja baze podataka nije moguće izmeniti kodni skup niti teritoriju.

Uparivanje (eng. collation) se odnosi na skup pravila kojima se zadaje način sortiranja i poređenja podataka. Podrazumevana vrednost je SYSTEM i ona se postavlja u zavisnosti od identifikatora zemlje date baze podataka.

Prilikom kreiranja baze podataka moguće je definisati i alias te baze u sistemskom direktorijumu baza podataka, navođenjem stavke ALIAS <alias_bp>. Ukoliko se ne navede alias baze podataka, koristi se naziv baze podataka.

Napomenimo da postoji još veliki broj drugih opcija koje je moguće navesti prilikom kreiranja baze podataka.

U okviru baze podataka moguće je praviti pojedinačne objekte baze podataka: sheme, tabele, poglede, alijase, indekse, trigere, itd. Nećemo se zadržavati na ovome, jer su ove naredbe već obrađivane u okviru prethodnog kursa o bazama podataka.

Dakle, da rezimirimo, prilikom pravljenja baze podataka izvršavaju se naredne radnje:

- pravi se baza podataka na navedenoj putanji (ili podrazumevanoj putanji);
- ako je omogućeno automatsko skladištenje, pravi se podrazumevana grupa skladišta pod imenom IBMSTOGROUP;
- za sve particije navedene u datoteci db2nodes.cfg prave se particije baze podataka;

- prave se prostori tabela SYSCATSPACE za tabele sistemskog kataloga, TEMPSPACE1 za privremene tabele koje se kreiraju tokom obrada nad bazom podataka, USERSPACE1 za korisnički definisane tabele i indekse;
- prave se tabele sistemskog kataloga i logovi oporavka;
- baza podataka se katalogizuje u direktorijumu lokalnih baza podataka i direktorijumu sistemskih baza podataka;
- čuvaju se navedene vrednosti za kodni skup, teritoriju i uparivanje;
- prave se sheme SYSCAT, SYSFUN, SYSIBM, SYSSTAT;
- ugrađuju se prethodno definisane vezne datoteke menadžera baze podataka u bazu podataka;
- daju se prava: npr. svim korisnicima se daje pravo SELECT na pogledima sistemskog kataloga, kao i prava CREATETAB i CONNECT; informacije o tome ko ima koja prava nad kojim objektom nalaze se u pogledu sistemskog kataloga SYSCAT.DBAUTH.

5.2 Premeštanje podataka

U sistemu DB2 postoji više mogućih načina za premeštanje podataka.

Naredbom INSERT moguće je dodati jedan ili više redova podataka u tabelu ili pogled (što se takođe realizuje kao umetanje podataka u tabelu na kojoj se pogled zasniva). Međutim, naredba INSERT ne predstavlja najbolji i najbrži način za učitavanje velike količine podataka u bazu podataka.

Naredbe IMPORT i EXPORT

Naredbom IMPORT moguće je izvršiti umetanje podataka iz ulazne datoteke u tabelu, pri čemu ulazna datoteka sadrži podatke iz druge baze podataka ili programa. Naredbu EXPORT moguće je koristiti za kopiranje podataka iz tabele u izlaznu datoteku za potrebe korišćenja podataka od strane druge baze podataka ili programa za tabelarnu obradu podataka. Ova datoteka se može kasnije koristiti za popunjavanje tabele, čime se dobija zgodan metod za migriranje podataka sa jedne baze podataka na drugu.

Mogući su različiti formati ovih datoteka:

- DEL – ASCII format u kome su podaci međusobno razdvojeni specijalnim karakterima za razdvajanje redova i kolona (eng. Delimited ASCII);
- ASC – ASCII format bez razdvajanja kolona (eng. Non-delimited ASCII); podržane su dve varijante ovog formata: sa fiksnom dužinom podataka i sa fleksibilnom dužinom podataka – u tom slučaju podaci su međusobno razdvojeni karakterom za razdvajanje redova. Ovaj format se ne može koristiti sa naredbom EXPORT;
- IXF – verzija integrisanog formata za razmenu podataka (eng. Integrated Exchange Format). To je binarni format, koji se može koristiti za prebacivanje podataka između operativnih sistema. Ovaj format se najčešće

koristi za izvoz podataka iz tabele, pri čemu je kasnije moguće te iste podatke uneti u istu ili neku drugu tabelu. Kod ovog formata podataka, tabela ne mora da postoji pre početka naredbe `IMPORT`, dok je kod `DEL` i `ASC` formata potrebno da tabela bude definisana (sa listom kolona i njihovim tipovima) pre nego što se izvrši naredba `IMPORT`.

"Smith, Bob",4973,15.46	Smith, Bob;4973;15.46
"Jones, Bill",12345,16.34	Jones, Bill;12345;16.34
"Williams, Sam",452,193.78	Williams, Sam;452;193.78

Slika 5.3: `DEL` format datoteke gde je kao karakter za razdvajanje redova iskorišćen karakter za prelazak u novi red, a za razdvajanje kolona iskorišćena (a) zapeta (b) tačka-zapeta, pri čemu niska karaktera nije terminisana

Smith, Bob	4973	15.46
Jones, Suzanne	12345	16.34
Williams, Sam	452123	193.78

Slika 5.4: `ASC` format datoteke sa fiksnom dužinom podataka, gde je kao karakter za razdvajanje redova iskorišćen karakter za prelazak u novi red

Osnovni format naredbe `EXPORT` je:

```
EXPORT TO <ime_datoteke> OF <tip_datoteke>
[MODIFIED BY <modifikator>]
[MESSAGES <datoteka_za_poruke>]
<select_naredba>
```

Dakle, ovom naredbom se podaci izvoze iz baze podataka u jedan od nekoliko mogućih formata datoteka. Naredbom `SELECT` korisnik zadaje koje podatke treba izvesti. Podržani formati datoteka su `DEL` i `IXF`. Opcijom `MODIFIED BY` je za razdvojene podatke moguće zadati nisku karaktera koja se koristi za razdvajanje – postoji veći broj predefinisanih modifikatora i mi nećemo ovde ulaziti u punu sintaksu. Na primer, za `DEL` format gde su kolone međusobno razdvojene karakterom `,` to je moguće zadati modifikatorom `coldel` (eng. column delimiter) za kojim sledi karakter koji se koristi za razdvajanje odnosno zapeta:

```
EXPORT TO temp OF del
MODIFIED BY coldel,
SELECT * FROM knjiga;
```

Stavka `MESSAGES <datoteka_za_poruke>` zadaje odredište na kome treba pisati poruke o greškama i upozorenjima koja se eventualno jave tokom operacije izvoza.

Na primer, možemo imati sledeći niz naredbi:

```
CONNECT TO bp_knjiga;
EXPORT TO knjiga_dat.ixf OF ixf
  MESSAGES knjiga_poruke.txt
  SELECT * FROM knjiga;
```

Pre poziva naredbe EXPORT potrebno je povezati se na bazu podataka iz koje želimo da izvezemo podatke.

Sintaksa naredbe IMPORT je sledeća:

```
IMPORT FROM <ime_datoteke> OF <tip_datoteke>
  [MODIFIED BY <modifikator>]
  [ALLOW NO ACCESS|ALLOW WRITE ACCESS]
  [MESSAGES <datoteka_za_poruke>]
  [COMMITCOUNT (n|AUTOMATIC)] [RESTARTCOUNT n]
  (INSERT|INSERT_UPDATE|REPLACE|REPLACE_CREATE)
  INTO <naziv_tabele>
```

Podrazumevano se prilikom uvoza podataka postavlja X katanac na tabeli iz koje se izvoze podaci, da bi se sprečilo da konkurentne aplikacije pristupe podacima iz tabele. Ova opcija se može i eksplicitno zadati navođenjem opcije ALLOW NO ACCESS. Ako je navedena opcija ALLOW WRITE ACCESS onda se postavlja IX katanac na tabelu, čime se omogućava konkurentnim aplikacijama da pristupe tabeli.

Opcija COMMITCOUNT n izvodi naredbu COMMIT nakon svakih n uvezenih redova. Ako je umesto vrednosti n zadata opcija AUTOMATIC, onda sama operacija interno utvrđuje kada treba da izvrši naredbu COMMIT. Opcijom RESTARTCOUNT n zadaje se da se operacija uvoza podataka treba početi od $(n+1)$ -og reda, dok se prvih n redova preskaču.

Ako je na kraju naredbe IMPORT navedena naredba INSERT onda se podaci dodaju u tabelu, bez izmene postojećih podataka u tabeli. Ako je navedena naredba INSERT_UPDATE podaci se dodaju u tabelu ili se menjaju postojeći redovi sa odgovarajućom vrednošću primarnog ključa. Ako je navedena naredba REPLACE onda se brišu svi postojeći podaci iz tabele i umeću novi redovi, dok ako je navedena opcija REPLACE_CREATE onda se, ako tabela postoji, briše njen sadržaj i dodaju novi redovi, a ako tabela ne postoji ona se pravi.

Na primer, možemo imati narednu naredbu:

```
IMPORT FROM knjiga_dat.ifx OF ixf
  MESSAGES knjiga_poruke.txt
  INSERT INTO knjiga
```

Naredba LOAD

Pored naredbe IMPORT, podatke je moguće dodati u tabelu i korišćenjem naredbe LOAD. Ove dve naredbe imaju sličnu namenu, ali se u mnogo čemu razlikuju. Naredbom IMPORT se u stvari izvodi operacija INSERT, te su stoga njene mogućnosti analogne pozivanju naredbe INSERT. Za razliku od nje, naredbom LOAD se formatirane stranice direktno upisuju u bazu podataka, te je na ovaj način moguće efikasno prebaciti veliku količinu podataka u novokreirane tabele ili u tabele koje već sadrže podatke.

Naredba LOAD se sastoji iz četiri faze:

1. faze punjenja
2. faze izgradnje
3. faze brisanja
4. faze kopiranja indeksa

Tokom faze punjenja, tabele se pune podacima i prikupljaju se ključevi indeksa i statistike tabela. Tačke pamćenja se uspostavljaju nakon intervala koji se zadaje kroz opciju SAVECOUNT naredbe LOAD. Generišu se poruke kojima se ukazuje na to koliko puta su ulazni redovi uspešno dodati u trenutku tačke pamćenja.

U drugoj fazi, fazi izgradnje, formiraju se indeksi na osnovu ključeva indeksa sakupljenih tokom faze punjenja. Ključevi indeksa se sortiraju tokom faze punjenja.

Tokom faze brisanja, iz tabele se brišu redovi koji narušavaju ograničenja jedinstvenog ključa ili primarnog ključa. Ovi obrisani redovi se čuvaju u tabeli izuzetaka ako je zadata (ona mora biti kreirana pre poziva naredbe LOAD), a poruke o odbačenim redovima se pamte u datoteci poruka. Nakon završetka naredbe LOAD poželjno je pregledati ovu datoteku, razrešiti svaki problem i uneti izmenjene redove u tabelu.

Tokom faze kopiranja indeksa, indeksirani podaci se kopiraju iz sistemskog privremenog prostora tabela u originalni prostor tabela. Ovo će se desiti samo ako je prilikom poziva naredbe LOAD zadat sistemski privremeni prostor tabela za pravljenje indeksa opcijom ALLOW READ ACCESS.

Sintaksa naredbe LOAD je:

```
LOAD FROM <datoteka> OF [ASC|DEL|IXF] [MODIFIED BY <mod>]
[SAVECOUNT n] [ROWCOUNT n] [WARNINGCOUNT n]
[MESSAGES <datoteka_zaporuke>]
(INSERT|REPLACE|RESTART|TERMINATE) INTO <naziv_tabele>
[FOR EXCEPTION <tabela_izuzetaka>]
[ALLOW NO ACCESS| ALLOW READ ACCESS [USE naziv_prostora_tabela]]
```

Navođenjem opcije SAVECOUNT n obezbeđuje se da se naredbom LOAD tačke pamćenja postavljaju na svakih n redova. Ova vrednost se konvertuje u broj stranica i zaokružuje. Opcijom ROWCOUNT n zadaje se da se iz datoteke učitava samo prvih n redova. Ako se zada opcija WARNINGCOUNT n , naredba LOAD se zaustavlja nakon n upozorenja.

Postoje četiti moda pod kojima se naredba LOAD može izvršavati: INSERT, REPLACE, RESTART, TERMINATE. Ako se navede naredba INSERT onda se podaci dodaju u tabelu bez izmene postojećih podataka u tabeli, ako se navede opcija REPLACE brišu se postojeći podaci iz tabele i učitavaju novi podaci, dok se definicije tabele i indeksa ne menjaju. Ukoliko se navede opcija RESTART ponovo započinje prethodno prekinuta naredba LOAD i ona se automatski nastavlja od poslednje tačke pamćenja. Ako se navede opcija TERMINATE terminše se prethodno prekinuta naredba LOAD i vrši povratak na tačku u kojoj je ona započela (čak i ako se nakon nje stiglo do nekih tačaka pamćenja); pritom se poništavaju akcije od tog momenta nadalje, bez obzira na to što se možda uspešno stiglo do nekih tačaka pamćenja. Dakle, kada se pokrene

naredba `LOAD` ciljna tabela se prebacuje u stanje “učitavanje u toku”. Ukoliko se ova naredba iz nekog razloga ne završi uspešno, tabela prelazi u stanje “učitavanje zaustavljeno”. Da bi tabela izašla iz ovog stanja, može se zahtevati naredba `LOAD TERMINATE` da se naredba poništi, može se zahtevati naredba `LOAD REPLACE` da se ponovo učitava kompletna tabela ili `LOAD RESTART`. Obično je ova poslednja varijanta najbolja jer se štedi vreme time što se učitavanje nastavlja od poslednje uspešne tačke pamćenja.

5.3 Pravljenje rezervnih kopija i oporavak

Naredba `BACKUP DATABASE`

Naredbom `BACKUP DATABASE` pravi se kopija podataka iz baze podataka i čuva je na nekom drugom medijumu. Ovi podaci se mogu koristiti u slučaju pada medijuma i štete na originalnim podacima. Moguće je napraviti rezervnu kopiju cele baze podataka, particije ili samo određenih prostora tabela.

Prilikom pravljenja rezervne kopije baze podataka, nije neophodno biti povezan na bazu podataka; operacija pravljenja rezervne kopije automatski uspostavlja vezu sa bazom podataka i ova veza se prekida nakon uspešnog završetka naredbe `BACKUP`. Ime baze podataka može biti lokalno ili da odgovara udaljenoj bazi podataka.

Osnovna sintaksa naredbe `BACKUP` je sledeća:

```
BACKUP DATABASE <alias_bp> [USER <username> [USING <password>]]
TO <dir/dev>
```

gde je `<alias_bp>` alias baze podataka čiju rezervnu kopiju pravimo, `<username>` identifikuje korisnika u čije se ime pravi rezervna kopija, a `<password>` njegovu šifru. Ukoliko se šifra eksplicitno ne zada, korisnik će biti upitan da je naknadno unese.

Na svim operativnim sistemima, nazivi datoteka koje se koriste kao slike rezervnih kopija sastoje se od nekoliko nadovezanih komponenti, razdvojenih tačkom, formata:

```
alias_bp.tip.inst.DBPARTnnn.timestamp.redni_br
```

Na primer: `Knjiga.0.DB201.DBPART000.20150922120112.001`. Ovi elementi redom označavaju:

- `alias_bp` – alias baze podataka: ovo je naziv dužine od 1 do 8 karaktera koji se zadaje prilikom pozivanja naredbe `BACKUP`,
- `tip` – tip operacije pravljenja rezervne kopije, pri čemu 0 označava pravljenje kompletne kopije na nivou baze podataka, 3 predstavlja pravljenje rezervne kopije na nivou prostora tabela, a 4 pravljenje slike rezervne kopije (dobijene naredbom `LOAD COPY TO`),
- `inst` – naziv instance: ovo je naziv tekuće instance, dužine između 1 i 8 karaktera, koji se uzima iz promenljive okruženja `DB2INSTANCE`,

- `DBPARTnnn` – broj particije baze podataka - u okruženjima baza podataka sa jednom particijom, ovo je uvek `DBPART000`. U okruženjima u kojima postoje particije, ovaj parametar ima vrednost `DBPARTnnn`, gde je `nnn` broj pridružen particiji baze podataka u datoteci `db2nodes.cfg`,
- `timestamp` – reprezentacija datuma i vremena u vidu niske od 14 karaktera koja predstavlja u kom trenutku se izvela operacija pravljenja rezervne kopije, njen format je `yyyymmddhhnnss` pri čemu:
 - `yyyy` predstavlja godinu (1995 do 9999)
 - `mm` predstavlja mesec (01 do 12)
 - `dd` predstavlja dan u mesecu (01 do 31)
 - `hh` predstavlja sate (00 do 23)
 - `nn` predstavlja minute (00 do 59)
 - `ss` predstavlja sekunde (00 do 59)
- `redni_br` – trocifreni broj koji se koristi kao ekstenzija datoteke.

Na primer, naredbom:

```
BACKUP DATABASE bp_knjiga TO /backup
```

vrši se pravljenje rezervne kopije u datom direktorijumu.

Inkrementalno pravljenje rezervnih kopija podrazumeva da uzastopne kopije baze podataka sadrže samo onaj deo podataka koji je menjan od poslednje napravljene rezervne kopije. Ono je često poželjnije jer smanjuje prostor koji se koristi za skladištenje i brže se izvodi. Navođenjem opcije `INCREMENTAL` pri likom poziva naredbe `BACKUP DATABASE` pravi se kopija samo onih podataka koji su izmenjeni u odnosu na najsvežiju kompletnu kopiju, dok navođenjem opcije `DELTA` nakon toga pravi se kopija samo onih podataka koji su izmenjeni u odnosu na poslednju rezervnu kopiju (koja može biti i inkrementalna). Na primer, naredbom:

```
BACKUP DATABASE bp_knjiga INCREMENTAL DELTA
```

pravi se samo kopija podataka izmenjenih od poslednje postojeće rezervne kopije.

Naredba RESTORE

Naredbom `RESTORE` moguće je povratiti bazu podataka čija je rezervna kopija bila napravljena korišćenjem naredbe `BACKUP`. Najjednostavniji oblik naredbe `RESTORE` zahteva samo navođenje alijasa baze podataka koju želimo da povratimo. Na primer,

```
RESTORE DATABASE bp_knjiga
```

U slučaju da baza podataka sa ovim nazivom postoji, ona će ovom operacijom biti zamenjena, te se korisniku prikazuje naredna poruka:

SQL2539W Warning! Restoring to an existing database that is the same as the backup image database. The database files will be deleted. Do you want to continue ? (y/n) If you specify y, the restore operation should complete successfully.

Moguće je zadati i putanju do direktorijuma ili uređaja na kome se nalazi rezervna kopija. Ukoliko se ona ne navede podrazumeva vrednost je tekući radni direktorijum. Ukoliko želimo da eksplicitno zadamo putanju, navodimo stavku FROM <direktorijum/uredjaj>:

```
RESTORE DATABASE bp_knjiga FROM /dev/backup
```

Takođe, moguće je zadati i direktorijum u kome se kreira baza podataka stavkom TO <ciljni-direktorijum>. Ovaj parametar se ignoriše ako se vrši povratak na postojeću bazu podataka.

Ukoliko u datom direktorijumu postoji više od jedne rezervne kopije potrebno je zadati vremenski trenutak kada je napravljena rezervna kopija baze podataka stavkom TAKEN AT, npr:

```
RESTORE DATABASE bp_knjiga  
TAKEN AT 20200202010101
```

Da bismo povratili bazu podataka na osnovu inkrementalnih rezervnih kopija potrebno je navesti opciju INCREMENTAL AUTOMATIC:

```
RESTORE DATABASE bp_knjiga  
INCREMENTAL AUTOMATIC  
TAKEN AT 20200202010101
```

Proces restauracije mora da nađe poslednju kompletnu rezervnu kopiju, a da zatim prođe kroz sve inkrementalne rezervne kopije do tačke restauracije.

Naredba RESTORE DATABASE zahteva ekskluzivnu konekciju, tj. prilikom startovanja operacije nijedna druga aplikacija ne sme da radi nad tom bazom podataka i ne dozvoljava se da bilo koja aplikacija pristupi bazi dok se operacija povratka podataka ne završi uspešno. Moguće je povratiti bazu podataka iz slike rezervne kopije koja je napravljena na 32-bitnom sistemu na 64-bitni sistem, ali ne i obratno.

Naredba RECOVER DATABASE

Naredbom RECOVER DATABASE vrši se povratak baze podataka na neku rezervnu kopiju a zatim ponovo izvršavaju transakcije koje su uspešno završene do određenog vremenskog trenutka ili do kraja loga. Ona koristi informacije iz datoteke istorije da odredi sliku rezervne kopije koju treba iskoristiti u određenom trenutku, te sam korisnik ne mora da zada određenu sliku rezervne kopije. Ako je zahtevana slika rezervne kopije inkrementalna rezervna kopija, naredba RECOVER će pozvati inkrementalnu automatsku logiku da izvede povratak podataka. Ako se zahteva oporavak u nekom vremenskom trenutku, ali najranija slika rezervne kopije iz datoteke istorije je kasnija od tog vremenskog trenutka, naredba RECOVER DATABASE vratiće grešku. Inače se za povratak baze podataka koristi slika rezervne kopije iz datoteke istorije sa

najkasnijim vremenom pravljenja rezervne kopije koje prethodi zahtevanom vremenskom trenutku.

Stavkom TO zadaje se vremenski trenutak (timestamp) na koji se treba vratiti i u kom treba povratiti sve potvrđene transakcije do tog momenta.

Ukoliko se želi povratak na najsvežiju rezervnu kopiju i ponovno izvršavanje svih transakcija iz log datoteke koristi se naredba:

```
RECOVER DATABASE bp_knjiga
```

Narednom naredbom bio bi izvršen povratak na dati vremenski trenutak:

```
RECOVER DATABASE bp_knjiga TO 2020-02-02-01.01.01
```

5.4 Aktivnosti na održavanju baze podataka

Naredba RUNSTATS

Tačne statistike o tabelama i indeksima su veoma značajne za DB2 optimizator koji na taj način bira efikasne planove pristupa za obradu zahteva aplikacije. Naredba RUNSTATS se može iskoristiti za sakupljanje novih statistika tabela i indeksa i ažuriranje statistika u sistemskom katalogu. Ako sadržaj tabele raste ili se dodaju novi indeksi, važno je ažurirati vrednosti ovih statistika (kao što su broj redova, broj stranica i prosečnu dužinu reda) tako da oslikavaju ove promene. DB2 baza podataka se može konfigurisati da automatski odredi tabele i indekse za koje je potrebno izračunati nove statistike.

Na ovaj način moguće je dobiti informacije o:

- broju stranica koje sadrže redove,
- broju stranica koje su u upotrebi,
- broju redova u tabeli,
- vrednostima statistika koje se tiču raspodele podataka,
- detaljne statistike za indekse kojima se određuje koliko je efikasno pristupiti podacima preko indeksa, ...

Na primer, osnovne statistike o tabeli knjiga možemo dobiti pozivom naredbe:

```
RUNSTATS ON TABLE knjiga
```

U okviru ove naredbe moguće je navesti mnogo dodatnih opcija. Na primer, za velike tabele možemo da koristimo uzorkovanje umesto da obrađujemo čitav skup podataka. Na primer, osnovne statistike o tabeli knjiga i detaljne statistike o svim indeksima, korišćenjem uzorkovanja, možemo sakupiti korišćenjem naredbe:

```
RUNSTATS ON TABLE knjiga  
AND SAMPLED DETAILED INDEXES ALL
```

Naredbe REORG i REORGCHK

Nakon velikog broja izmena nad podacima u tabeli, podaci koji su logički povezani mogu da budu na stranicama koje nisu fizički susedne; takođe, ako se iz tabele briše mnogo redova veliki broj stranica tabele može da sadrži malo ili ništa od podataka. Naredba REORG se može iskoristiti za reorganizaciju tabele i indeksa da bi se popravila efikasnost skladištenja i smanjili troškovi pristupa. Reorganizacija tabele vrši defragmentaciju podataka i uklanja prazne prostore; time se koristi manje stranica, čime se štedi prostor na disku i sadržaj tabele se može pročitati sa manjim brojem ulazno/izlaznih operacija. Može se takođe vršiti i preuređenje redova u tabeli, npr. podaci se mogu preurediti u skladu sa nekim indeksom, da bi se mogli izdvojiti sa što manjim brojem operacija čitanja. Veliki broj izmena na podacima u tabeli može da degradira i performanse indeksa.

Nećemo razmatrati punu sintaksu i sve opcije koje pruža ova naredba, već ćemo videti njene najčešće primere upotrebe. Na primer, naredbom:

```
REORG TABLE knjiga USE TEMPSPACE1
```

se tabela knjiga reorganizuje korišćenjem privremenog prostora tabela.

Ako bismo hteli da reorganizujemo indeks knjigaInd nad tabelom knjiga, to bi bilo moguće uraditi naredbom:

```
REORG INDEX knjigaInd ON TABLE knjiga
```

dok bismo reorganizovali sve indekse nad datom tabelom naredbom:

```
REORG INDEXES ALL FOR TABLE knjiga
```

Naredba REORGCHK se može koristiti za dobijanje preporuka koje tabele i koji indeksi bi imali koristi od reorganizacije. Nakon toga se može iskoristiti naredba REORG za implementiranje kompresije postojećih tabela i indeksa.

Naredba REORGCHK ima narednu sintaksu:

```
REORGCHK [UPDATE STATISTICS | CURRENT STATISTICS]
ON [SCHEMA <naziv_sheme> |
TABLE [USER | SYSTEM | ALL | <naziv_tabele>]]
```

Opcijom UPDATE STATISTICS poziva se naredba RUNSTATS da bi se ažurirale statistike o tabelama i indeksima i onda se ažurirane vrednosti statistika koriste za utvrđivanje da li je potrebna reorganizacija tabele ili indeksa, dok ako je navedena opcija CURRENT STATISTICS koriste se trenutno raspoložive vrednosti statistika o tabelama i indeksima. Proveru je moguće izvršiti za konkretnu tabelu, za sve korisničke tabele, za sve sistemske tabele ili za sve tabele neke sheme.

Na primer, naredbom:

```
REORGCHK ON SCHEMA AUTO
```

se za svaku tabelu ove sheme računa, primenom nekoliko formula, da li bi je trebalo reorganizovati. U koloni REORG rezultata se za svaku od formula ispisuje karakter * ako izračunata vrednost te statistike premaši preporučenu vrednost, a - ako ne premašuje. Slično važi i za indekse.