Baze podataka II

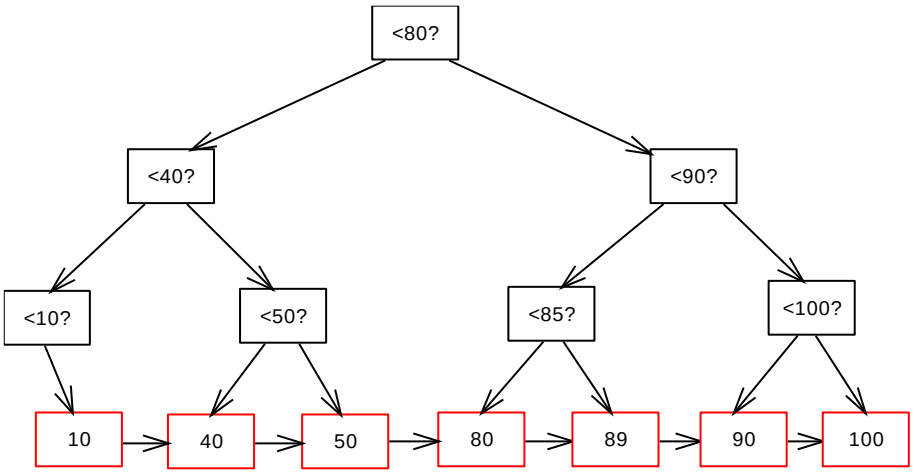
Autori:
Milan Segedinac
Goran Savić

1. Indeksi

Kada je bilo reči o algoritmima, videli smo da se pretraga elementa u neuređenoj listi realizuje u linearnom vremenu ($\Theta(n)$). To znači da će broj koraka izvršavanja algoritma linearno rasti sa brojem elemenata u listu. Obzirom da je broj slogova u bazi u realnoj aplikaciji najčešće veoma velik (nekoliko miliona i više), često nije prihvatljivo da se slogovi pronalaze linearnom pretragom.

Kada smo obrađivali temu naprednih struktura podataka, videli smo da odabir strukture podataka može značajno da utiče na efikasnost algoritma. Ukoliko bismo podatke smestili u *binarno stablo pretrage*, umesto u linearnom vremenu elemente bismo mogli da pronalazimo u logaritmskom vremenu. Primera radi, ako bismo linearnom pretragom probali da pronađemo ime u telefonskom imeniku grada koji ima 12 000 000 stanovnika (toliko stanovnika ima Nju Jork) linearnom pretragom bi nam trebalo u proseku 6 000 000 pokušaja. Ako bismo podatke imali u binarnom stablu pretrage trebalo bi $\log_2 12000000 = 24$ pokušaja.

Ova ideja je iskorišćena u sistemima za upravljanje bazama podataka tako da se za ubrzanje performansi pretrage koriste *indeksi*. Indeks je kopija podataka iz polja organizovana u sortirano binarno stablo tako da bude bolje pretraživa. Najčešće se koriste *B+* stabla u kojima samo *listovi* sadrže vrednosti, a ostali čvorovi sadrže informacije potrebne za određivanje rute u toku pretrage. Indeks nad celobrojnim poljem prikazan je slikom ispod.



Slika – Struktura indeksa

Postavljanje indeksa na polje u MySQL SUBP je jednostavno. Potrebno je samo navesti koja kolona će biti indeksirana. Osnovne operacije za rad sa indeksima prikazane su u listingu ispod.

```
drop table Person;

CREATE TABLE Person(
     id INTEGER  NOT NULL AUTO_INCREMENT,
     first_name VARCHAR(255),
     last_name VARCHAR(255),
     PRIMARY KEY (id),
     -- postavljanje indeksa prilikom kreiranja tabele
     INDEX first_name_FI_1 (first_name)
);

-- postavljanje indeksa nakon kreiranja tabele
ALTER TABLE Person ADD INDEX last_name_FI_1 (last_name);

-- uklanjanje indeksa
ALTER TABLE Person DROP INDEX first_name_FI_1;

-- prikaz svih indeksa za tabelu
SHOW INDEX FROM Person;
```

Listing – Rad sa indeksima u MySQL SUBP

Identifikator **first_name_FI_1** je proizvoljni naziv indeksa i koristi se da bi se indeks kasnije referencirao, na primer prilikom brisanja.

Indeksi nam omogućuju da pretragu učinimo efikasnijom. Ali u programiranju nema besplatnog ručka: efikasnost pretrage ostvarena je pravljenjem i održavanjem indeksne struktutre što se odražava na efikasnost dodavanja, izmene i brisanja jer sada, pri ovim operacijama moraju da se menjaju i indeksi. Takođe, indeksi čuvaju kopije vrednosti iz baze podataka, tako da imamo veće zauzeće memorije. Stoga je veoma važan pažljiv odabir kolona na koje će se postavljati indeksi.

2. Transakcije

Kako bismo mogli da implementiramo program koji omogućuje elektronski transfer novca? Trebali bismo da obezbedimo da se vrednost na jednom računu umanji za iznos, a na drugom uveća za taj iznos, kao što je prikazano u listingu ispod. Pretpostavljamo da je inicijalno na oba računa bilo po 500 novčanih jedinica i da se vrši prenos 100 novčanih jedinica sa računa sa id=1 na račun sa id=2.

```
UPDATE account SET amount=400 WHERE id=1;
UPDATE account SET amount=600 WHERE id=2;
```

Listing – naivna implementacija prenosa sredstava

Šta bi se desilo kada bi nestalo struje nakon što je izvršena prva linija koda iz listinga, a dok još nije počela da se izvršava druga linija koda? Prvi račun bi imao 100 novčanih jedinica manje, a drugi bi i dalje ima 500 novčanih jedinica. Znači, u sistemu je negde „nestalo“ 100 novčanih jedinica. Ovakve greške mogu da budu katastrofalne u informacionim sistemima.

Da bi se ovakve greške izbegle, potrebno je da možemo da skup naredbi okupimo u celinu koja će se ili čitava izvršiti ili se ni jedna njena komanda neće izvršiti. Takva celina se u bazama podataka zove *transakcija*. Tačnije, transakcija mora da zadovolji sledeće kriterijume (*ACID*):

- **A** – *Atomičnost* (*Atomicity*) – izvršiće se sve ili nijedna od operacija iz skupa
- **C** – *Konzistentnost* (*Consistency*) – transakcija će prevesti bazu iz jednog validnog stanja u drugo (ostaviće bazu u konzistentnom stanju)
- **I** – *Izolovanost* (*Isolation*) – Druge transakcije koje pristupaju istim podacima neće uticati na rezultat transakcije.
- **D** – *Trajnost* (*Durability*) – Jednom potvrđena (commited) transakcija će trajno uskladištiti podatke

Zadavanje transakcije prikazano je listingom ispod.

```
START TRANSACTION;
      UPDATE account SET amount=400 WHERE id=1;
      UPDATE account SET amount=600 WHERE id=2;
COMMIT;
```

Listing – prenos sredstava u trnasakciji

Transakcija počinje komandom `START TRANSACTION` i završava se komandom `COMMIT`. Rezultat transakcije je moguće poništiti komandom `ROLLBACK`.

Od navedenih svojstava transakcija, *Izolovanost* može da bude najrelaksiranija. Sistemi za upravljanje bazama podataka dozvoljavaju postavaljanje nivoa izolovanosti transakacija. Tako MySQL dozvoljava:

- `SERIALIZABLE` – čitanje i pisanje su blokirani nad selektovanim slogom dok se ne završi transakcija. Ako se upit izvršava nad opsegom podataka (na primer ako u upitu postoji where iskaz) i svi podaci u opsegu su blokirani dok se ne završi transakcija. Ovo je najrestritkivniji režim izolovanosti transakcija.
- `REPEATABLE READ` – čitanje i pisanje su blokirani nad selektovanim slogom dok se ne završi transakcija. Ne blokira se opseg podataka.
- `READ COMMITTED` – podaci koji se zapisuju su zaključani do kraja transakcije. Podaci koji se čitaju u transakciji su zaključani samo u svom selekt upitu.
- `READ UNCOMMITTED` – najmanje restriktivni režim izolovanosti traksankcije. U ovom režimu moguće je „prljavo čitanje“ (dirty read). Moguće je videti izmenjene podatke transakcija koja još uvek nije komitovana.

Primer postavljanja nivoa izolovanosti transakcije dat je listingom ispod.

```
SET GLOBAL TRANSACTION ISOLATION LEVEL READ UNCOMMITTED
```

Listing – postavljanje niova izolovanosti transakcija

3. Uskladištene procedure

U primeru sa prenosom sredstava postavljali smo vrednost prvog računa na 400, a drugog na 600. Bilo bi bolje da smo prvi račun umanjili za prosleđeni iznos, a drugi uvećali za taj isti prosleđeni iznos, jer bismo onda taj programski kod mogli da iskoristimo i u drugim situacijama osim kada je stanje

računa baš 500 novčanih jedinica ili kada se prenosi iznos različit od 100 novčanih jedinica. Za to koristimo *uskladištene procedure*. Pisanje uskladištenih procedura je programiranje koje zahteva znanje programskog jezika u kom se one pišu, pa ćemo, obzirom na obim i namenu ovog materijala, samo ilustrativno prikazati jednu uskladištenu proceduru i opisati je. Uskladištena procedura za prenos novca data je listingom ispod.

```
DELIMITER //
CREATE PROCEDURE transfer(IN srcAccountId INT, IN dstAccountId INT, IN amount INT)
 BEGIN
  START TRANSACTION;
        SELECT @srcAmount := account.amount FROM account WHERE id=srcAccountId;
        SELECT @dstAmount := account.amount FROM account WHERE id=dstAccountId;
        UPDATE account SET amount=@srcAmount-amount WHERE id=srcAccountId;
        UPDATE account SET amount=@dstAmount+amount WHERE id=dstAccountId;
  COMMIT;
 END //
DELIMITER ;

CALL transfer(1,2,100);
```

Listing – uskladištena procedura

Na početku smo postavili da je SQL delimiter '//', zato što je default delimiter u pisanju procedura takođe ';'. Nakon definisanja procedure vratili smo SQL delimiter na ';'.

Procedura se zove transfer i ima tri ulazna parametra: id izvorišnog računa, id odredišnog računa i iznos koji se prenosi. Svi parametri su tipa INT. Iznos na izvorišnom računu preuzimamo u SELECT upitu u varijablu `@srcAmount`, a iznos na odredišnom u varijablu `@dstAmount`. Zatim u UPDATE upitima postavljamo izmenjene vrednosti. Proceduru pozivamo ključnom rečju `CALL` i navodimo konkretne argumente nad kojima će se procedura izvršiti.

4. SQL upiti

U prošloj lekciji videli smo kako možemo da pravimo SQL upite kojima preuzimamo podatke iz jedne tabele iz baze podataka. Međutim, priroda relacionih baza podataka je takva da nam često trebaju objedinjeni podaci iz nekoliko tabela. Ovakve upite formiramo *spajanjem tabela*, odnosno koristeći SQL naredbu JOIN. Takođe, često nam ne trebaju sami podaci, već sumarni prikazi podataka. U tom slučaju koristimo *agregatne funkcije*. O ovim konceptima će biti reči u ovoj sekciji.

JOIN

Posmatrajmo bazu podataka o mestima datu listingom ispod.

```
create table country (
      id integer,
      name varchar(255),
      primary key (id)
);

create table place (
      place_id integer,
      name varchar(255),
      population integer,
      country_id integer,
      primary key(place_id),
      foreign key(country_id)
            references country(id)
);
```

Listing – tabele country i place

Ova baza podataka ima dve tabele: `country` i `place`. Slog u tabeli `place` moći će da ima kao spoljašnji ključ identifikator sloga u tabeli `country`. To znači da ćemo za mesto mogći da znamo kojoj državi pripada. Bazu podataka ćamo popuniti podacima kao što je prikazano listingom ispod.

```
insert into country values (1, 'Srbija');
insert into country values (2, 'Rumunija');

insert into place values (1, 'Novi Sad', 252459, 1);
insert into place values (2, 'Beograd', 1351000, 1);
insert into place values (3, 'Temisvar', 306462, 2);
insert into place values (4, 'Bukurest', 1913000, 2);
insert into place (place_id, name, population) values
  (5, 'Budimpesta', 1300000);
```

Listing – podaci u bazu

U bazi su dve države: Srbija i Rumunija. Dodato je po dva grada za svaku od ove dve države. Takođe, dodat je i jedan grad (Budimpešta) koji nije ni u jednoj od ove dve države. Potrebno nam je da preuzmemo grad *zajedno sa njegovom državom*.

Za povezivanje tabela u select upitu koristi se klauzula JOIN. Ako bismo u upitu spojili ove dve tabele pomoću JOIN klauzule (SELECT * FROM place JOIN country) dobili bismo rezultat prikazan listingom ispod.

place_id	name	population	country_id	id	name
1	Novi Sad	252459	1	1	Srbija
1	Novi Sad	252459	1	2	Rumunija
2	Beograd	1351000	1	1	Srbija
2	Beograd	1351000	1	2	Rumunija
3	Temisvar	306462	2	1	Srbija
3	Temisvar	306462	2	2	Rumunija
4	Bukurest	1913000	2	1	Srbija
4	Bukurest	1913000	2	2	Rumunija
5	Budimpesta	1300000	Null	1	Srbija
5	Budimpesta	1300000	Null	2	Rumunija

Tabela – rezultat select upita

Kako je formirana ova tabela? Prvo je uzet slog Novi Sad i na njega su dodata sva polja sloga Srbija. Zatim je uzet ponovo slog Novi Sad i na njega su dodata sva polja sloga Rumunija. Pošto više nije bilo slogova u tabeli country, uzet je sledeći slog iz tabele place (Beograd), pa su na njega dodata sva polja iz prvog sloga tabele coutnry (Srbija), pa je to ponovljeno za sledeći slog (Rumunija). Ovo je ponavljano dok nisu „potrošeni“ svi slogovi iz obe tabele.

U stvari, rezultat upita je formiran kao *Dekartov proizvod skupova* place i country. Nama najčešće ne treba ceo Dekartov proizvod skupova, nego samo njegov podskup. Na primer, trebaće nam samo oni elementi koji predstavljaju grad zajedno sa **njegovom** državom. Takav rezultat dobijamo tako što Dekartov proizvod filtriramo pomoću klauzule **ON**.

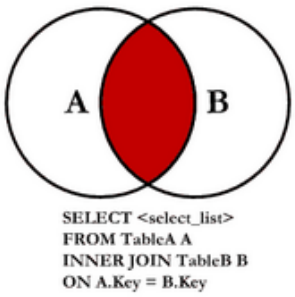
Upit za preuzimanje gradova iz svoje države zajedno sa rezultatima dat je listingom ispod.

```
SELECT * FROM place JOIN country ON country_id=id;

place_id   name       population country_id id    name
1          Novi Sad   252459     1          1     Srbija
2          Beograd    1351000    1          1     Srbija
3          Temisvar   306462     2          2     Rumunija
4          Bukurest   1913000    2          2     Rumunija
```

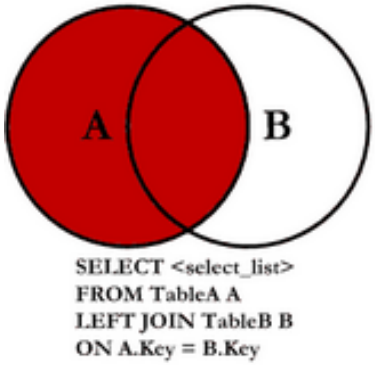
Listing – JOIN i ON

Sada je svaki grad prikazan sa njegovom državom, ali nisu prikazani **svi** gradovi. Izostala je Budimpešta, jer za ovaj grad nije bila uneta država. Klauzula JOIN je sinonim za INNER JOIN, kojom se selektuju svi redovi iz obe tabele za koje je zadovoljena on klauzula u obe tabele. Ovakva selekcija grafički je prijazana na slici ispod.



Slika – inner join

Ako bismo hteli da budu selektovani i gradovi koji nemaju postavljenu državu, koristili bismo LEFT JOIN. Tako se selektuju svi redovi iz leve tabele. Ukoliko ne postoji odgovarajući, postavlja se null. Grafički prikaz LEFT JOIN dat je slikom ispod, a rezultat listingom ispod.



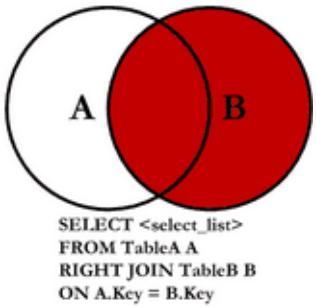
Slika – left join

```
SELECT * FROM place LEFT JOIN country ON country_id=id;

place_id   name       population counry_id  id    name
1          Novi Sad   252459     1          1     Srbija
2          Beograd    1351000    1          1     Srbija
3          Temisvar   306462     2          2     Rumunija
4          Bukurest   1913000    2          2     Rumunija
5          Budimpesta 1300000    null       null  null
```

Listing – LEFT JOIN

Operacija RIGHT JOIN radi obrnuto – biće preuzeti svi slogovi iz desne tabele, a ako ne postoje vrednosti u levoj postavlja se null. Grafički prikaz i ilustracija ove operacije dati su slikom i listingom ispod. Radi ilustracije dodali smo još jednu državu u bazu (Grčku) za koju ne postoji ni jedan dodat grad.



Slika – RIGHT JOIN

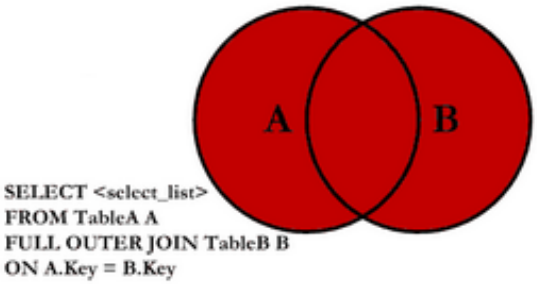
```
INSERT INTO country VALUES (3, 'Grcka');
```

```
SELECT * FROM place RIGHT JOIN country ON country_id=id;

place_id   name       population counry_id  id    name
1          Novi Sad   252459     1          1     Srbija
2          Beograd    1351000    1          1     Srbija
3          Temisvar   306462     2          2     Rumunija
4          Bukurest   1913000    2          2     Rumunija
null       null       null       3          3     Grcka
```

Listing – RIGHT JOIN

Operacijom FULL OUTER JOIN preuzumaju se svi slogovi i leve i desne tabele. Grafički prikaz ove operacije dat je slikom ispod. Ova operacija nije podržana u MySQL ali može da se simulira.



Slika – FULL OUTER JOIN

Rezultat dobijen izvršavanjem SELECT upita sa JOIN klauzulom možemo posmatrati kao tabelu, tako da možemo da imamo složene upite. Ako bismo, pored grada i države trebali da predstavimo i adresu, imali bismo tabele Country, Place (koja bi kao strani ključ imala id iz tabele Country) i Address (koja bi kao strani ključ imala id iz tabele Place). Nad ovakvom shemom mogli bismo da postavimo upit koji selektuje sve adrese zajedno sa njihovim gradovima i sa državam tih gradova. Ovakav upit, u kome imamo dve JOIN klauzule, dat je listingom ispod.

```
SELECT *
FROM Address LEFT JOIN Place
ON address.Place_id=place.id
LEFT JOIN Country
ON Place.Country_id = Country.id;
```

Listing – složeni upit

Agregatne funkcije

Agregatne funkcije nam omogućavaju da postavljamo upite u kojima će se izvršiti izračunavanje vrednosti u koloni. Najčešće korišćene agregatne funkcije su:

- *COUNT* – broj vrednosti u koloni
- *SUM* – suma vrednosti u koloni
- *MIN* – najmanja vrednost u koloni
- *MAX* – maksimalna vrednost u koloni
- *AVG* – prosečna vrednost u koloni

Ukoliko ne navedemo drugačije, agregatna funkcija će se izvršiti nad celom kolonom. Na primer, ako bismo hteli da dobijemo sumu broja stanovnika u svim gradovima u bazi, mogli bismo da koristimo upit `SELECT SUM(population) FROM place;`

Agregatna funkcija iz prethodnog primera vratila nam je rezultat koji se odnosi na celu kolonu. Često nam treba da dobijemo sumarne podatke za grupe slogova u tabeli. U tom slučaju koristimo GROUP BY klauzulu. Primer korišćenja ove klauzule zajedno sa rezultatima dat je listingom ispod.

```
SELECT SUM(population), country.name FROM place LEFT JOIN
  country ON country_id=id GROUP BY country.name;

SUM(population)  name
1300000          null
2219462          Rumunija
1603459          Srbija
```

Listing – Agregatna funkcija i grupisanje

Klauzulom BY napravljeno je tri grupe gradova prema tome kakav im je country.name. Za svaku grupu je pirmenjen SUM kojim je sračunata suma populacije za tu grupu. Pošto smo koristili LEFT JOIN, dobili smo da su populacije gradova kojima nisu dodeljene države okupljene u zasebnu grupu.