

Sveučilište u Zagrebu
Prirodoslovno Matematički Fakultet
- Matematički odjel

Robert Manger
BAZE PODATAKA
skripta

Prvo izdanje
Zagreb, rujan 2003.

Sadržaj

1	UVOD U BAZE PODATAKA	3
1.1	Osnovni pojmovi vezani uz baze podataka	3
1.1.1	Baza podataka, DBMS, model podataka	3
1.1.2	Ciljevi koji se nastoje postići korištenjem baza podataka	4
1.1.3	Arhitektura baze podataka	4
1.1.4	Jezici za rad s bazama podataka	5
1.1.5	Poznati softverski paketi za rad s bazama podataka	6
1.2	Životni ciklus baze podataka	6
1.2.1	Analiza potreba	6
1.2.2	Modeliranje podataka	7
1.2.3	Implementacija	7
1.2.4	Testiranje	8
1.2.5	Održavanje	8
2	MODELIRANJE PODATAKA	9
2.1	Modeliranje entiteta i veza	9
2.1.1	Entiteti i atributi	9
2.1.2	Veze	9
2.1.3	Prikaz ER-sheme pomoću dijagrama	10
2.1.4	Složenije veze	11
2.2	Relacijski model	13
2.2.1	Relacija, atribut, n-torka, ključ	13
2.2.2	Pretvaranje ER-sheme u relacijsku	14
2.2.3	Usporedba relacijskog modela s mrežnim i hijerarhijskim	15
2.3	Normalizacija relacijske sheme	16
2.3.1	Funkcionalna ovisnost	16
2.3.2	Druga normalna forma	17
2.3.3	Treća normalna forma	17
2.3.4	Boyce-Codd-ova normalna forma	18
2.3.5	Višeznačna ovisnost i četvrta normalna forma	19
2.3.6	Razlozi zbog kojih se može odustati od normalizacije	20
3	JEZICI ZA RELACIJSKE BAZE PODATAKA	21
3.1	Relacijska algebra	21
3.1.1	Skupovni operatori	22
3.1.2	Selekcija	23
3.1.3	Projekcija	23
3.1.4	Kartezijski produkt	24
3.1.5	Prirodni spoj	24
3.1.6	Theta-spoj	26
3.1.7	Dijeljenje	26
3.1.8	Vanjski spoj	26
3.2	Relacijski račun	27
3.2.1	Račun orijentiran na n-torke	27
3.2.2	Račun orijentiran na domene	28

3.3	Jezik SQL	29
3.3.1	Postavljanje upita	29
3.3.2	Ažuriranje relacija	31
3.4	Optimizacija upita	31
3.4.1	Odnos između relacijske algebre i računa	32
3.4.2	Osnovna pravila za optimizaciju	32
4	FIZIČKA GRAĐA BAZE PODATAKA	35
4.1	Elementi fizičke građe	35
4.1.1	Vanjska memorija računala	35
4.1.2	Datoteke	35
4.1.3	Smještaj datoteke u vanjskoj memoriji	36
4.1.4	Pointeri	37
4.1.5	Fizička građa cijele baze	37
4.2	Pristup na osnovu primarnog ključa	37
4.2.1	Jednostavna datoteka	38
4.2.2	Hash datoteka	38
4.2.3	Datoteka s indeksom	39
4.2.4	B-stablo	41
4.3	Pristup na osnovu drugih podataka	43
4.3.1	Invertirana datoteka	43
4.3.2	Višestruke vezane liste	44
4.3.3	Podijeljena hash funkcija	45
5	IMPLEMENTACIJA RELACIJSKIH OPERACIJA	47
5.1	Implementacija prirodnog spoja	47
5.1.1	Algoritam ugniježđenih petlji	47
5.1.2	Algoritam zasnovan na sortiranju i sažimanju	48
5.1.3	Algoritam zasnovan na indeksu	48
5.1.4	Algoritam zasnovan na hash funkciji i razvrstavanju	49
5.2	Implementacija selekcije, projekcije i ostalih operacija	50
5.2.1	Implementacija selekcije	50
5.2.2	Implementacija projekcije	50
5.2.3	Implementacija ostalih operacija.	51
5.3	Optimalno izvrednjavanje algebarskog izraza	51
6	INTEGRITET I SIGURNOST PODATAKA	53
6.1	Čuvanje integriteta	53
6.1.1	Ograničenja kojima se čuva integritet domene	53
6.1.2	Ograničenja kojima se čuva integritet unutar relacije	53
6.1.3	Ograničenja kojima se čuva referencijalni integritet	54
6.2	Istovremeni pristup	54
6.2.1	Transakcije i serijalizabilnost	54
6.2.2	Lokoti i zaključavanje	56
6.2.3	Dvofazni protokol zaključavanja	56
6.2.4	Vremenski žigovi	57
6.3	Oporavak	57
6.3.1	Rezervna kopija baze	58
6.3.2	Žurnal datoteka	58
6.3.3	Neutralizacija jedne transakcije	58
6.3.4	Ponovno uspostavljanje baze	59
6.4	Zaštita od neovlaštenog pristupa	59
6.4.1	Identifikacija korisnika	59
6.4.2	Pogledi kao mehanizam zaštite	59
6.4.3	Ovlaštenja	60

1

UVOD U BAZE PODATAKA

1.1 Osnovni pojmovi vezani uz baze podataka

Baze podataka predstavljaju višu razinu rada s podacima u odnosu na klasične programske jezike. Riječ je o tehnologiji koja je nastala s namjerom da se uklone slabosti tradicionalne “automatske obrade podataka” iz 60-tih i 70-tih godina 20. stoljeća. Ta tehnologija osigurala je veću produktivnost, kvalitetu i pouzdanost u razvoju aplikacija koje se svode na pohranjivanje i pretraživanje podataka u računalu.

1.1.1 Baza podataka, DBMS, model podataka

Baza podataka je skup međusobno povezanih podataka, pohranjenih u vanjskoj memoriji računala. Podaci su istovremeno dostupni raznim korisnicima i aplikacijskim programima. Ubacivanje, promjena, brisanje i čitanje podataka obavlja se posredstvom zajedničkog softvera. Korisnici i aplikacije pritom ne moraju poznavati detalje fizičkog prikaza podataka, već se referenciraju na logičku strukturu baze.

Sustav za upravljanje bazom podataka (Data Base Management System - DBMS) je poslužitelj (server) baze podataka. On oblikuje fizički prikaz baze u skladu s traženom logičkom strukturom. Također, on obavlja u ime klijenata sve operacije s podacima. Dalje, on je u stanju podržati razne baze, od kojih svaka može imati svoju logičku strukturu, no u skladu s istim modelom. Isto tako, brine se za sigurnost podataka, te automatizira administrativne poslove s bazom.

Podaci u bazi su logički organizirani u skladu s nekim **modelom podataka**. Model podataka je skup pravila koja određuju kako može izgledati logička struktura baze. Model čini osnovu za koncipiranje, projektiranje i implementiranje baze. Dosadašnji DBMS-i obično su podržavali neki od sljedećih modela:

Relacijski model. Zasnovan na matematičkom pojmu relacije. I podaci i veze među podacima prikazuju se “pravokutnim” tabelama.

Mrežni model. Baza je predočena usmjerenim grafom. Čvorovi su tipovi zapisa, a lukovi definiraju veze među tipovima zapisa.

Hijerarhijski model. Specijalni slučaj mrežnog. Baza je predočena jednim stablom ili skupom stabala. Čvorovi su tipovi zapisa, a hijerarhijski odnos “nadređeni-podređeni” izražava veze među tipovima zapisa.

Objektni model. Inspiriran je objektno-orijentiranim programskim jezicima. Baza je skup trajno pohranjenih objekata koji se sastoje od svojih internih podataka i “metoda” (operacija) za rukovanje s tim podacima. Svaki objekt pripada nekoj klasi. Između klasa se uspostavljaju veze nasljeđivanja, agregacije, odnosno međusobnog korištenja operacija.

Hijerarhijski i mrežni model bili su u upotrebi u 60-tim i 70-tim godinama 20. stoljeća. Od 80-tih godina pa sve do današnjih dana prevladava relacijski model. Očekivani prijelaz na objektni model za sada se nije desio, tako da današnje baze podataka uglavnom još uvijek možemo poistovjetiti s relacijskim bazama.

1.1.2 Ciljevi koji se nastoje postići korištenjem baza podataka

Spomenuli smo da baze podataka predstavljaju višu razinu rada s podacima u odnosu na klasične programske jezike. Ta viša razina rada očituje se u tome što tehnologija baza podataka nastoji (i u velikoj mjeri uspijeva) ispuniti sljedeće ciljeve.

Fizička nezavisnost podataka. Razdvaja se logička definicija baze od njene stvarne fizičke građe.

Znači, ako se fizička građa promijeni (na primjer, podaci se prepisu u druge datoteke na drugim diskovima), to neće zahtijevati promjene u postojećim aplikacijama.

Logička nezavisnost podataka. Razdvaja se globalna logička definicija cijele baze podataka od lokalne logičke definicije za jednu aplikaciju. Znači, ako se logička definicija promijeni (na primjer uvede se novi zapis ili veza), to neće zahtijevati promjene u postojećim aplikacijama. Lokalna logička definicija obično se svodi na izdvajanje samo nekih elemenata iz globalne definicije, uz neke jednostavne transformacije tih elemenata.

Fleksibilnost pristupa podacima. U starijim mrežnim i hijerarhijskim bazama, staze pristupanja podacima bile su unaprijed definirane, dakle korisnik je mogao pretraživati podatke jedino onim redoslijedom koji je bio predviđen u vrijeme projektiranja i implementiranja baze. Danas se zahtijeva da korisnik može slobodno prebirati po podacima, te po svom nahođenju uspostavljati veze među podacima. Ovom zahtjevu zaista zadovoljavaju jedino relacijske baze.

Istovremeni pristup do podataka. Baza mora omogućiti da veći broj korisnika istovremeno koristi iste podatke. Pritom ti korisnici ne smiju ometati jedan drugoga, te svaki od njih treba imati dojam da sam radi s bazom.

Čuvanje integriteta. Nastoji se automatski sačuvati korektnost i konzistencija podataka, i to u situaciji kad postoje greške u aplikacijama, te konfliktne istovremene aktivnosti korisnika.

Mogućnost oporavka nakon kvara. Mora postojati pouzdana zaštita baze u slučaju kvara hardvera ili grešaka u radu sistemskog softvera.

Zaštita od neovlaštenog korištenja. Mora postojati mogućnost da se korisnicima ograniče prava korištenja baze, dakle da se svakom korisniku reguliraju ovlaštenja što on smije a što ne smije raditi s podacima.

Zadovoljavajuća brzina pristupa. Operacije s podacima moraju se odvijati dovoljno brzo, u skladu s potrebama određene aplikacije. Na brzinu pristupa može se utjecati odabirom pogodnih fizičkih struktura podataka, te izborom pogodnih algoritama za pretraživanje.

Mogućnost podešavanja i kontrole. Velika baza zahtijeva stalnu brigu: praćenje performansi, mijenjanje parametara u fizičkoj građi, rutinsko pohranjivanje rezervnih kopija podataka, reguliranje ovlaštenja korisnika. Također, svrha baze se vremenom mijenja, pa povremeno treba podesiti i logičku strukturu. Ovakvi poslovi moraju se obavljati centralizirano. Odgovorna osoba zove se **administrator** baze podataka. Administratoru trebaju stajati na raspolaganju razni alati i pomagala.

1.1.3 Arhitektura baze podataka

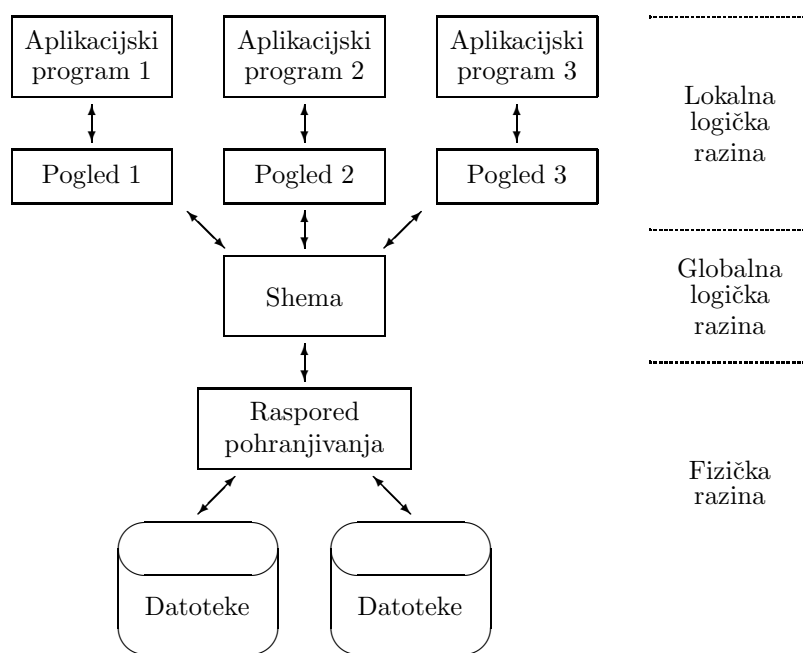
Arhitektura baze podataka sastoji se od tri "sloja" i sučelja među slojevima, kao što je prikazano na Slici 1.1. Riječ je o tri razine apstrakcije

Fizička razina odnosi se na fizički prikaz i raspored podataka na jedinicama vanjske memorije. To je aspekt kojeg vide samo sistemski programeri (oni koji su razvili DBMS). Sama fizička razina može se dalje podijeliti na više pod-razina apstrakcije, od sasvim konkretnih staza i cilindara na disku, do već donekle apstraktnih pojmova datoteke i zapisa kakve susrećemo u klasičnim programskim jezicima. **Raspored pohranjivanja** opisuje kako se elementi logičke definicije baze preslikavaju na fizičke uređaje.

Globalna logička razina odnosi se na logičku strukturu cijele baze. To je aspekt kojeg vidi projektant baze odnosno njen administrator. Zapis logičke definicije naziva se **shema** (engleski također schema). Shema je tekst ili dijagram koji definira logičku strukturu baze, i u skladu je sa zadanim

modelom. Dakle imenuju se i definiraju svi tipovi podataka i veze među tim tipovima, u skladu s pravilima korištenog modela. Također, shema uvodi i ograničenja kojim se čuva integritet podataka.

Lokalna logička razina odnosi se na logičku predodžbu o dijelu baze kojeg koristi pojedina aplikacija. To je aspekt kojeg vidi korisnik ili aplikacijski programer. Zapis jedne lokalne logičke definicije zove se **pogled** (engleski view) ili pod-shema. To je tekst ili dijagram kojim se imenuju i definiraju svi lokalni tipovi podataka i veze među tim tipovima, opet u skladu s pravilima korištenog modela. Također, pogled zadaje preslikavanje kojim se iz globalnih podataka i veza izvode lokalni.



Slika 1.1: Arhitektura baze podataka

Za stvaranje baze podataka potrebno je zadati samo shemu i poglede. DBMS tada automatski generira potrebni raspored pohranjivanja i fizičku bazu. Administrator može samo donekle utjecati na fizičku građu baze, podešavanjem njemu dostupnih parametara.

Programi i korisnici ne pristupaju izravno fizičkoj bazi, već dobivaju ili pohranjuju podatke posredstvom DBMS-a. Komunikacija programa odnosno korisnika s DBMS-om obavlja se na lokalnoj logičkoj razini.

1.1.4 Jezici za rad s bazama podataka

Komunikacija korisnika odnosno aplikacijskog programa i DBMS-a odvija se pomoću posebnih jezika. Ti jezici tradicionalno se dijele na sljedeće kategorije.

Jezik za opis podataka (Data Description Language - DDL). Služi projektantu baze ili administratoru u svrhu zapisivanja sheme ili pogleda. Dakle tim jezikom definiramo podatke i veze među podacima, i to na logičkoj razini. Koji puta postoji posebna varijanta jezika za shemu, a posebna za poglede. Naredbe DDL obično podsjećaju na naredbe za definiranje složenih tipova podataka u jezicima poput COBOL, PL/I, C, Pascal.

Jezik za manipuliranje podacima (Data Manipulation Language - DML). Služi programeru za uspostavljanje veze između aplikacijskog programa i baze. Naredbe DML omogućuju “manevriranje” po bazi, te jednostavne operacije kao što su upis, promjena, brisanje ili čitanje zapisa. U nekim softverskim paketima, DML je zapravo biblioteka potprograma: “naredba” u DML svodi se na poziv potprograma. U drugim paketima zaista se radi o posebnom jeziku: programer tada piše program u kojem su izmiješane naredbe dvaju jezika, pa takav program treba prevoditi s dva prevodioca (DML-precompiler, obični compiler).

Jezik za postavljanje upita (Query Language - QL). Služi neposrednom korisniku za interaktivno pretraživanje baze. To je jezik koji podsijeca na govorni (engleski) jezik Naredbe su neproceduralne, dakle takve da samo specificiraju rezultat kojeg želimo dobiti, a ne i postupak za dobivanje rezultata.

Ovakva podjela na tri jezika danas je već prilično zastarjela. Naime, kod relacijskih baza postoji tendencija da se sva tri jezika objedine u jedan sveobuhvatni. Primjer takvog **integriranog** jezika za relacijske baze je SQL: on služi za definiranje podataka, manipuliranje i pretraživanje. Integrirani jezik se može koristiti interaktivno (preko on-line interpretera) ili se on može pojavljivati uklopljen u aplikacijske programe.

Naglasimo da gore spomenute vrste jezika nisu programski jezici. Dakle ti jezici su nam nužni da bi se povezali s bazom, no oni nam nisu dovoljni za razvoj aplikacija koje će nešto raditi s podacima iz baze.

Tradicionalni način razvoja aplikacija koje rade s bazom je korištenje klasičnih programskih jezika (COBOL, PL/I, C, Pascal . . .) s ugniježđenim DML-naredbama. U 80-tim godinama 20. stoljeća bili su dosta popularni i tzv. **jezici 4. generacije** (4-th Generation Languages - 4GL): riječ je o jezicima koji su bili namijenjeni isključivo za rad s bazama, te su zato u tom kontekstu bili produktivniji od klasičnih programskih jezika opće namjene. Problem s jezicima 4. generacije je bio u njihovoj nestandardnosti: svaki od njih je u pravilu bio dio nekog određenog softverskog paketa za baze podataka, te se nije mogao koristiti izvan tog paketa (baze).

U današnje vrijeme, aplikacije se najčešće razvijaju u standardnim **objektno orijentiranim** programskim jezicima (Java, C++, . . .). Za interakcije s bazom koriste se unaprijed pripremljene klase objekata. Ovakva tehnika je dovoljno produktivna zbog korištenja gotovih klasa, a rezultirajući program se lako dotjeruje, uklapa u veće sustave ili prenosi s jedne baze na drugu.

1.1.5 Poznati softverski paketi za rad s bazama podataka

Baze podataka se u pravilu realiziraju korištenjem nekog od provjerenih softverskih paketa. Tabelarni prikaz 1.1 daje pregled softvera koji u ovom trenutku predstavljaju tehnološki vrh te imaju značajan udjel na svjetskom tržištu.

Gotovo svi današnji softverski paketi podržavaju relacijski model i SQL. Svaki od njih sadrži svoj DBMS, uobičajene klijente (na primjer interaktivni interpreter SQL), te biblioteke i alate za razvoj aplikacija. Svaki paket isporučuje se u verzijama za razne računalne platforme (operacijske sustave).

Konkurencija među proizvođačima softvera za baze podataka je izuzetno velika, tako da je posljednjih godina često dolazilo do njihovog nestanka, spajanja ili preuzimanja. Lista relevantnih softverskih paketa zato je svake godine sve kraća. Jedino osvježanje predstavlja nedavna pojava public-domain softvera poput MySQL.

1.2 Životni ciklus baze podataka

Uvođenje baze podataka u neko poduzeće ili ustanovu predstavlja složeni zadatak koji zahtijeva timski rad stručnjaka raznih profila. To je projekt koji se može podijeliti u pet faza: analiza potreba, modeliranje podataka, implementacija, testiranje i održavanje.

1.2.1 Analiza potreba

Proučavaju se tokovi informacija u poduzeću. Uočavaju se **podaci** koje treba pohranjivati i veze među njima. U velikom poduzećima, gdje postoje razne grupe korisnika, pojavit će se razni “pogledi” na podatke. Te poglede treba uskladiti tako da se eliminira redundancija i nekonzistentnost. Na primjer, treba u raznim pogledima prepoznati sinonime i homonime, te uskladiti terminologiju.

Analiza potreba također treba obuhvatiti analizu **transakcija** (operacija) koje će se obavljati nad bazom podataka, budući da to može isto imati utjecaja na sadržaj i konačni oblik baze. Važno je procijeniti frekvenciju i opseg pojedinih transakcija, te zahtjeve na performanse.

Rezultat analize je dokument (pisan neformalno u prirodnom jeziku) koji se zove **specifikacija potreba**.

Proizvođač	Produkt	Operacijski sustav	Jezici
IBM Corporation	DB2	Linux, UNIX (razni), MS Windows NT/2000/XP, VMS, MVS, VM, OS/400	SQL, COBOL, Java, ...
Oracle Corporation	Oracle	MS Windows (razni), Mac OS, UNIX (razni), Linux i drugi	SQL, Java i drugi
IBM Corporation (prije : Informix Software Inc.)	Informix	UNIX (razni), Linux, MS Windows NT/2000/XP	SQL, Java i drugi
Microsoft	MS SQL Server	MS Windows NT/2000/XP	SQL, C++, ...
MySQL AB	MySQL	Linux, UNIX (razni), MS Windows (razni), Mac OS	SQL, C, PHP, ...
Sybase Inc.	Sybase SQL Server	MS Windows NT/2000, OS/2, Mac, UNIX (razni), UNIXWare	SQL, COBOL, ...
Hewlett Packard Co.	Allbase/SQL	UNIX (HP-UX)	SQL, 4GL, C, ...
Cincom Systems Inc.	Supra	MS Windows NT/2000, Linux, UNIX (razni), VMS, MVS, VM	SQL, COBOL, ...
Microsoft Corporation	MS Access	MS Windows (razni)	Access Basic, SQL

Tabelarni prikaz 1.1: Poznati softverski paketi za rad s bazama podataka

1.2.2 Modeliranje podataka

Različiti pogledi na podatke, otkriveni u fazi analize, sintetiziraju se u jednu cjelinu - globalnu shemu. Precizno se utvrđuju tipovi podataka. Shema se dalje dotjeruje ("normalizira") tako da zadovolji neke zahtjeve kvalitete. Također, shema se prilagođava ograničenjima koje postavlja zadani model podataka, te se dodatno modificira da bi bolje mogla udovoljiti zahtjevima na performanse. Na kraju se iz sheme izvode pogledi (pod-sheme) za pojedine aplikacije (grupe korisnika).

1.2.3 Implementacija

Na osnovu sheme i pod-shema, te uz pomoć dostupnog DBMS-a, fizički se realizira baza podataka na računalu. U DBMS-u obično postoje parametri kojima se može utjecati na fizičku organizaciju baze. Parametri se podešavaju tako da se osigura efikasan rad najvažnijih transakcija. Razvija se skup programa koji realiziraju pojedine transakcije te pokrivaju potrebe raznih aplikacija. Baza se inicijalno puni podacima.

1.2.4 Testiranje

Korisnici pokusno rade s bazom i provjeravaju da li ona zadovoljava svim zahtjevima. Nastoje se otkriti greške koje su se mogle potkrasti u svakoj od faza razvoja: dakle u analizi potreba, modeliranju podataka, implementaciji. Greške u ranijim fazama imaju teže posljedice. Na primjer, greška u analizi potreba uzrokuje da transakcije možda korektno rade, no ne ono što korisnicima treba već nešto drugo. Dobro bi bilo kad bi takve propuste otkrili prije implementacije. Zato se u novije vrijeme, prije prave implementacije, razvijaju i približni **prototipovi** baze podataka, te se oni pokazuju korisnicima. Jeftinu izradu prototipova omogućuju jezici 4. generacije i objektno-orijentirani jezici.

1.2.5 Održavanje

Odvija se u vrijeme kad je baza već ušla u redovnu upotrebu. Sastoji se od sljedećeg: popravak grešaka koje nisu bile otkrivene u fazi testiranja; uvođenje promjena zbog novih zahtjeva korisnika; podešavanje parametara u DBMS u svrhu poboljšavanja performansi. Održavanje zahtijeva da se stalno prati rad s bazom, i to tako da to praćenje ne ometa korisnike. Administratoru baze podataka trebaju stajati na raspolaganju odgovarajući alati (utility programi).

2

MODELIRANJE PODATAKA

2.1 Modeliranje entiteta i veza

Bavimo se pitanjem: kako oblikovati shemu za bazu podataka, usklađenu s pravilima relacijskog modela. U stvarnim situacijama dosta je teško direktno pogoditi relacijsku shemu. Zato se služimo jednom pomoćnom fazom koja se zove **modeliranje entiteta i veza** (Entity-Relationship Modelling). Riječ je o oblikovanju jedne manje precizne, konceptualne sheme, koja predstavlja apstrakciju realnog svijeta. Ta tzv. ER-shema se dalje, više-manje automatski, pretvara u relacijsku. Modeliranje entiteta i veza zahtijeva da se svijet promatra preko tri kategorije:

entiteti: objekti ili događaji koji su nam od interesa;

veze: odnosi među entitetima koji su nam od interesa;

atributi: svojstva entiteta i veza koja su nam od interesa.

2.1.1 Entiteti i atributi

Entitet je nešto o čemu želimo spremati podatke, nešto što je u stanju postojati ili ne postojati, te se može identificirati. Entitet može biti objekt ili biće (na primjer kuća, student, auto), odnosno događaj ili pojava (na primjer nogometna utakmica, praznik, servisiranje auta).

Entitet je opisan **atributima** (na primjer atributi kuće su: adresa, broj katova, boja fasade, ...). Ukoliko neki atribut i sam zahtijeva svoje atribute, tada ga radije treba smatrati novim entitetom (na primjer model auta). Isto pravilo vrijedi i ako atribut može istovremeno imati više vrijednosti (na primjer kvar koji je popravljen pri servisiranju auta).

Ime entiteta, zajedno sa pripadnim atributima, zapravo određuje **tip** entiteta. Može postojati mnogo **primjeraka** (pojava) entiteta zadanog tipa (na primjer *STUDENT* je tip čiji primjerci su Petrović Petar, Marković Marko, ...).

Kandidat za ključ je atribut, ili skup atributa, čije vrijednosti jednoznačno određuju primjerak entiteta zadanog tipa. Dakle, ne mogu postojati dva različita primjerka entiteta istog tipa s istim vrijednostima kandidata za ključ. (Na primjer za tip entiteta *AUTO*, kandidat za ključ je atribut *REG_BROJ*). Ukoliko jedan tip entiteta ima više kandidata za ključ, tada biramo jednog od njih i proglašavamo ga **primarnim ključem**. (Na primjer primarni ključ za tip entiteta *STUDENT* mogao bi biti atribut *BROJ_INDEKSA*).

2.1.2 Veze

Veze se uspostavljaju između dva ili više tipova entiteta (na primjer veza *IGRA_ZA* između tipova entiteta *IGRAČ* i *TIM*). Zapravo je riječ o imenovanoj binarnoj ili k -narnoj relaciji između primjeraka entiteta zadanih tipova. Za sada ćemo se ograničiti na veze između točno dva tipa entiteta.

Funkcionalnost veze može biti:

Jedan-naprama-jedan (1 : 1). Jedan primjerak prvog tipa entiteta može biti u vezi s najviše jednim primjerkom drugog tipa entiteta, te također jedan primjerak drugog tipa može biti u vezi s najviše jednim primjerkom prvog tipa. Na primjer veza *JE_PROČELNIK* između tipova entiteta *NASTAVNIK* i *ZAVOD* (na fakultetu).

Jedan-naprma-mnogo ($1 : N$). Jedan primjerak prvog tipa entiteta može biti u vezi s 0, 1 ili više primjeraka drugog tipa entiteta, no jedan primjerak drugog tipa može biti u vezi s najviše jednim primjerkom prvog tipa. Na primjer veza *PREDAJE* između tipova entiteta *NASTAVNIK* i *KOLEGIJ*.

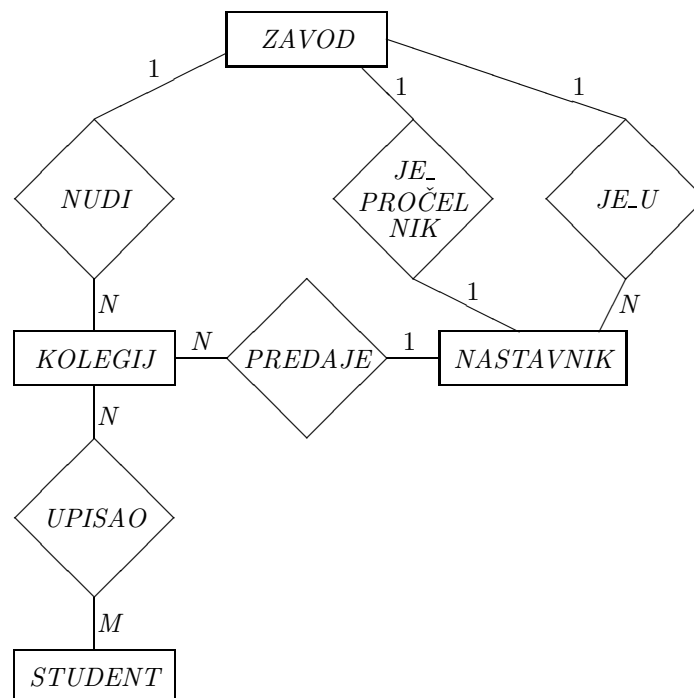
Mnogo-naprma-mnogo ($M : N$). Jedan primjerak prvog tipa entiteta može biti u vezi s 0, 1 ili više primjeraka drugog tipa entiteta, te također jedan primjerak drugog tipa može biti u vezi s 0, 1 ili više primjeraka prvog tipa. Na primjer veza *UPISAO* između tipova entiteta *STUDENT* i *KOLEGIJ*.

Veza može imati i svoje atribute koje ne možemo pripisati ni jednom od tipova entiteta (na primjer veza *UPISAO* može imati atribut *DATUM_UPISA*).

Ako svaki primjerak entiteta nekog tipa mora sudjelovati u zadanoj vezi, tada kažemo da tip entiteta ima **obavezno članstvo** u toj vezi. Inače tip entiteta ima neobavezno članstvo. (Na primjer između tipova entiteta *ISPIT* i *KOLEGIJ* zadana je veza *IZ*, koja ima funkcionalnost ($N : 1$). *ISPIT* ima obavezno članstvo u vezi *IZ*, jer svaki ispit mora biti iz nekog kolegija.) Odluka da li je članstvo obavezno ili neobavezno koji put je stvar dogovora odnosno projektantove odluke (na primjer članstvo za *KOLEGIJ* u vezi *PREDAJE*).

2.1.3 Prikaz ER-sheme pomoću dijagrama

Običaj je da se ER-shema nacrti kao dijagram u kojem pravokutnici predstavljaju tipove entiteta, a rombovi veze. Veze su povezane bridovima s odgovarajućim tipovima entiteta. Imena tipova entiteta i veza, te funkcionalnost veza, uneseni su u dijagram. Posebno se prilaže lista atributa za svaki entitet odnosno vezu. U toj listi možemo specificirati obaveznost članstva u vezama.



Slika 2.1: ER-shema baze podataka o fakultetu

Kao primjer, pogledajmo dijagram na Slici 2.1 koji prikazuje bazu podataka o fakultetu. Tipovi entiteta su:

1. *ZAVOD*, s atributima IME_ZAVODA, *ADRESA*, ...
2. *KOLEGIJ*, s atributima BR_KOLEGIJA, *NASLOV*, *SEMESTAR*, ...
3. *STUDENT*, s atributima BR_INDEKSA, *IME_STUDENTA*, *ADRESA*, *SPOL*, ...

4. *NASTAVNIK*, s atributima *IME_NASTAVNIKA* (pretpostavljamo da je jedinstveno), *BR_SOBE*, ...

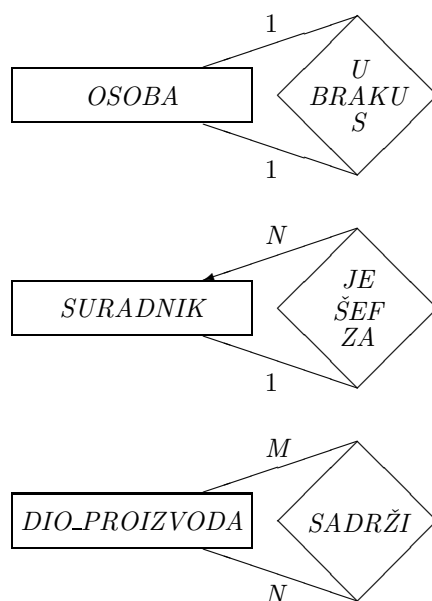
Podvučeni atributi čine primarni ključ. Veze su:

1. *JE_PROČELNIK*, bez atributa. *ZAVOD* ima obavezno članstvo.
2. *JE_U*, bez atributa. *NASTAVNIK* ima obavezno članstvo.
3. *NUDI*, bez atributa. *KOLEGIJ* ima obavezno članstvo.
4. *UPISAO*, s atributom *DATUM_UPISA*.
5. *PREDAJE*, bez atributa. *KOLEGIJ* ima na primjer obavezno članstvo.

2.1.4 Složenije veze

U stvarnim situacijama pojavljuju se i složenije veze od onih koje smo do sada promatrali. Navest ćemo neke od njih.

Involuirana veza povezuje jedan tip entiteta s tim istim tipom. Dakle riječ je o binarnoj relaciji između raznih primjeraka entiteta istog tipa. Funkcionalnost takve veze opet može biti $(1 : 1)$, $(1 : N)$, odnosno $(M : N)$.

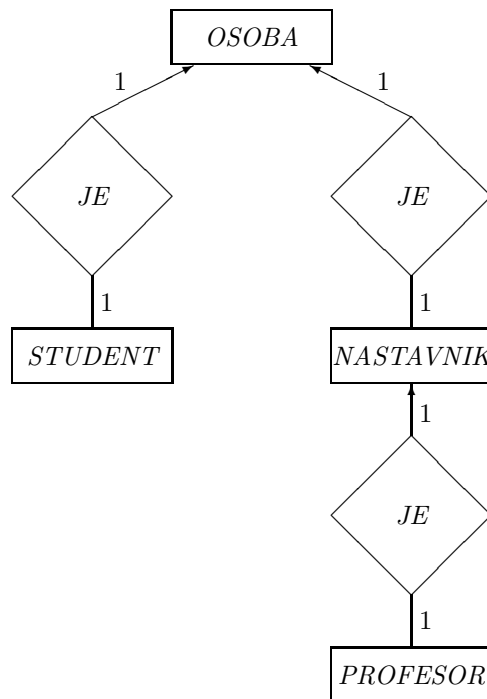


Slika 2.2: Primjeri za involuirane veze

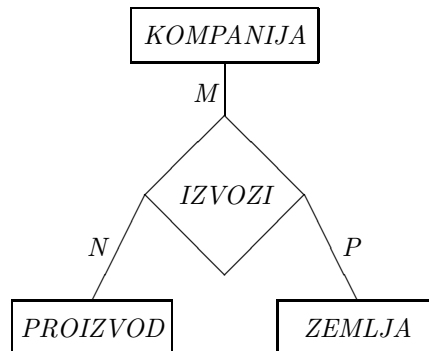
Slika 2.2 sadrži primjere za involuirane veze s različitim funkcionalnostima. Prvi dijagram na Slici 2.2 napravljen je pod pretpostavkom da su prošli brakovi osobe zaboravljeni, a poligamija zabranjena. Članstvo u u vezi *U_BRAKU_S* je neobavezno. Drugi dijagram na Slici 2.2 ima ucrtanu strelicu koja pokazuje smjer tumačenja veze *JE_ŠEF_ZA*. Možemo uzeti da je članstvo u toj vezi neobavezno, jer postoji barem jedan suradnik koji nema šefa. Treći dijagram na Slici 2.2 odnosi se na dijelove proizvoda koji se proizvode u nekoj tvornici. Pritom jedan složeniji dio sadrži više jednostavnijih. Isti jednostavniji dio pojavljuje se u više složenijih.

Pod-tipovi. Tip entiteta E_1 je podtip tipa entiteta E_2 ako je svaki primjerak od E_1 također i primjerak od E_2 . E_1 nasljeđuje sve attribute od E_2 , no E_1 može imati i dodatne attribute. Situaciju opisujemo pomoću specijalne $(1 : 1)$ veze *JE* (engleski *IS_A* koja se može pojaviti više puta unutar ER-sheme.

Slika 2.3 sadrži primjer ER-sheme s pod-tipovima i nad-tipovima. Riječ je o tipovima entiteta za osobe koje se pojavljuju na fakultetu. *NASTAVNIK* uključuje profesore, docente i asistente.



Slika 2.3: Primjer ER-scheme s pod-tipovima entiteta



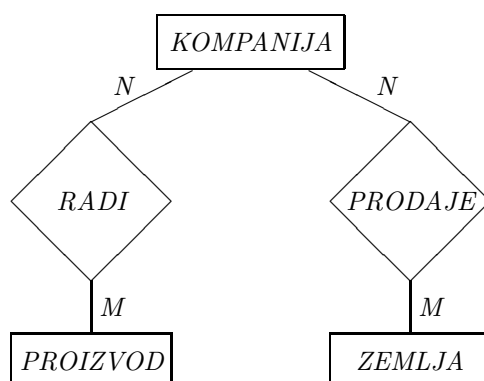
Slika 2.4: Primjer ternarne veze

Ternarne veze uspostavljaju se između tri tipa entiteta. Znači riječ je o ternarnoj relaciji između primjeraka triju tipova entiteta. Postoje brojne mogućnosti za funkcionalnost ternarne veze, na primjer $(N : M : P)$, $(1 : N : M)$, $(1 : 1 : N)$ ili čak $(1 : 1 : 1)$.

Primjer ternarne veze sa Slike 2.4 odnosi se na podatke o kompanijama, proizvodima koje one proizvode i zemljama u koje one izvoze svoje proizvode. Funkcionalnost ove veze je mnogo-naprama-mnogo-naprama-mnogo, dakle $(N : M : P)$, jer na primjer za zadani par (kompanija, proizvod) postoji mnogo zemalja u koje ta kompanija izvozi taj proizvod, itd.

Ternarnu vezu uvodimo samo onda kad se ona ne može rastaviti na dvije binarne. Uzmimo da u primjeru sa Slike 2.4 vrijedi pravilo: ako kompanija izvozi u neku zemlju, tada ona odmah izvozi sve svoje proizvode u tu zemlju. Uz ovo pravilo, razmatrana ternarna veza može se zamijeniti s dvije binarne, u skladu s dijagramom na Slici 2.5.

ER model dovoljno je jednostavan da ga ljudi različitih struka mogu razumjeti. Zato ER shema služi za komunikaciju projektanta baze podataka i korisnika, i to u najranijoj fazi razvoja baze. Postojeći DBMS ne mogu direktno implementirati ER shemu, već zahtijevaju da se ona detaljnije razradi, te modificira u skladu s pravilima relacijskog, mrežnog, odnosno hijerarhijskog modela.



Slika 2.5: Rastavljanje ternarne veze na dvije binarne

2.2 Relacijski model

Relacijski model bio je teoretski zasnovan još krajem 60-tih godina 20. stoljeća, u radovima E.F. Codd-a. Model se dugo pojavljivao samo u akademskim raspravama i knjigama. Prve realizacije na računalu bile su suviše spore i neefikasne. Zahvaljujući intenzivnom istraživanju, te napretku samih računala, efikasnost relacijskih baza postepeno se poboljšavala. Sredinom 80-tih godina 20. stoljeća relacijski model je postao prevladavajući. I danas većina DBMS koristi taj model.

2.2.1 Relacija, atribut, n-torka, ključ

Relacijski model zahtijeva da se baza podataka sastoji od skupa pravokutnih tabela - tzv. **relacija**. Svaka relacija ima svoje ime po kojem je razlikujemo od ostalih u istoj bazi. Jedan stupac relacije obično sadrži vrijednost jednog atributa (za entitet ili vezu) - zato stupac poistovjećujemo s **atributom** i obratno. Atribut ima svoje ime po kojem ga razlikujemo od ostalih u istoj relaciji. Vrijednosti jednog atributa su podaci istog tipa. Dakle, definiran je skup dozvoljenih vrijednosti za atribut, koji se zove **domena** atributa. Vrijednost atributa mora biti jednostruka i jednostavna (ne da se rastaviti na dijelove). Pod nekim uvjetima toleriramo situaciju da vrijednost atributa nedostaje (nije upisana). Jedan redak relacije obično predstavlja jedan primjerak entiteta, ili bilježi vezu između dva ili više primjeraka. Redak nazivamo **n-torka**. U jednoj relaciji ne smiju postojati dvije jednake n-torke. Broj atributa je **stupanj** relacije, a broj n-torki je **kardinalnost** relacije.

Kao primjer, navodimo u Tabelarnom prikazu 2.1 relaciju *AUTO*, s atributima *REG_BROJ*, *PROIZVOĐAČ*, *MODEL*, *GODINA*. Relacija sadrži podatke o automobilima koji se nalaze na popravku u nekoj radionici.

<i>AUTO</i>			
<i>REG_BROJ</i>	<i>PROIZVOĐAČ</i>	<i>MODEL</i>	<i>GODINA</i>
ZID654	Ford	Fiesta	1997
BXI930	Volkswagen	Golf	1996
COI453	Nissan	Primera	1997
ZXI675	Ford	Escort	1995
RST786	Fiat	Punto	1993
TXI521	Ford	Orion	1995
HCY675	Volkswagen	Jetta	1997

Tabelarni prikaz 2.1: Relacija s podacima o automobilima.

Relacija ne propisuje nikakav redoslijed svojih n-torki i atributa. Dakle, permutiranjem redaka i stupaca tabele dobivamo drukčiji zapis iste relacije.

Uvedena terminologija potječe iz matematike. Naime, neka je R relacija stupnja n i neka su domene njenih atributa redom D_1, D_2, \dots, D_n . Tada se R može interpretirati kao podskup Kartezijevog produkta $D_1 \times D_2 \times \dots \times D_n$. Znači, naš pojam relacije odgovara matematičkom pojmu n -arne

relacije. U komercijalnim DBMS, umjesto “matematičkih” termina (relacija, n-torka, atribut), češće se koriste neposredni termini (tabela, redak, stupac) ili termini iz tradicionalnih programskih jezika (datoteka, zapis, polje).

Ključ K relacije R je podskup atributa od R koji ima sljedeća “vremenski neovisna” svojstva:

1. Vrijednosti atributa iz K jednoznačno određuju n -torku u R . Dakle ne mogu u R postojati dvije n -torke s istim vrijednostima atributa iz K .
2. Ako izbacimo iz K bilo koji atribut, tada se narušava svojstvo 1.

Budući da su sve n -torke u R međusobno različite, K uvijek postoji. Naime, skup svih atributa zadovoljava svojstvo 1. Izbacivanjem suvišnih atributa doći ćemo do podskupa koji zadovoljava i svojstvo 2.

Dešava se da relacija ima više kandidata za ključ. Tada jedan on njih proglašavamo **primarnim ključem**. Atributi koji sastavljaju primarni ključ zovu se **primarni atributi**. Vrijednost primarnog atributa ne smije ni u jednoj n -torki ostati neupisana.

Građu relacije kratko opisujemo tzv. **shemom relacije**, koja se sastoji od imena relacije i popisa imena atributa u zagradama. Primarni atributi su podvučeni. Na primjer, za relaciju o automobilima (Tabelarni prikaz 2.1), shema izgleda ovako:

AUTO (REG-BROJ, PROIZVOĐAČ, MODEL, GODINA) .

2.2.2 Pretvaranje ER-sheme u relacijsku

U nastavku objašnjavamo kako se pojedini elementi ER-sheme pretvaraju u relacije. Na taj način pokazat ćemo kako se iz cijele ER-sheme dobiva relacijska shema.

Pretvorba tipova entiteta. Svaki tip entiteta prikazuje se jednom relacijom. Atributi tipa postaju atributi relacije. Jedan primjerak entiteta prikazan je jednom n -torkom. Primarni ključ entiteta postaje primarni ključ relacije. Na primer tip *STUDENT* iz naše fakultetske baze podataka sa Slike 2.1 prikazuje se relacijom:

STUDENT (BR-INDEKSA, IME-STUDENTA, ADRESA, SPOL, ...) .

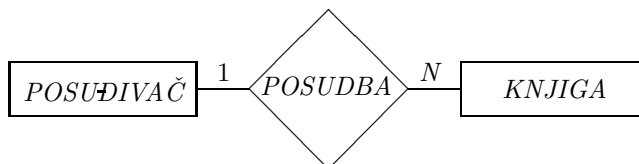
Doduše, sudjelovanje entiteta u vezama može zahtijevati da se u relaciju dodaju još neki atributi koji nisu postojali u odgovarajućem tipu entiteta.

Pretvorba binarnih veza. Ako tip entiteta E_2 ima obavezno članstvo u $(N : 1)$ vezi s tipom E_1 , tada relacija za E_2 treba uključiti primarne attribute od E_1 . Na primjer ako u ER-shemi sa Slike 2.1 svaki kolegij mora biti ponuđen od nekog zavoda, tada se veza *NUDI* svodi na to da u relaciju *KOLEGIJ* ubacimo ključ relacije *ZAVOD* :

KOLEGIJ (BR-KOLEGIJA, IME-ZAVODA, NASLOV, SEMESTAR, ...) .

Ključ jedne relacije koji je prepisan u drugu relaciju zove se **strani ključ** (u toj drugoj relaciji).

Ako tip entiteta E_2 ima neobavezno članstvo u $(N : 1)$ vezi s tipom E_1 , tada vezu možemo prikazati na prethodni način, ili uvođenjem nove relacije čiji atributi su primarni atributi od E_1 i E_2 .



Slika 2.6: ER-shema za dio baze podataka o knjižnici

Kao primjer, promotrimo vezu na Slici 2.6 koja prikazuje posuđivanje knjiga u knjižnici. Odgovarajuće relacije za prikaz te veze i pripadnih entiteta mogle bi biti:

POSUĐIVAČ (BR-ISKAZNICE, PREZIME-IME, ADRESA, ...) ,

KNJIGA (REG-BROJ, BR-ISKAZNICE, NASLOV, ...) .

Ovdje smo u relaciju *KNJIGA* dodali *BR-ISKAZNICE* osobe koja je posudila knjigu. Vrijednost atributa *BR-ISKAZNICE* bit će prazna u mnogim n -torkama relacije *KNJIGA*, tj. za sve knjige koje trenutno nisu posuđene. Drugo rješenje zahtijeva tri relacije:

POSUĐIVAČ (*BR_ISKAZNICE*, *PREZIME_IME*, *ADRESA*, ...) ,
KNJIGA (*REG_BROJ*, *NASLOV*, ...) ,
POSUDBA (*REG_BROJ*, *BR_ISKAZNICE*) .

Samo one knjige koje su trenutno posuđene predstavljene su n-torkom u relaciji *POSUDBA*. Posebna relacija za prikaz veze je pogotovo preporučljiva ako relacija ima svoje atribute. Na primjer u relaciju *POSUDBA* mogli bi uvesti atribut *DATUM_VRAĆANJA*.

(*N* : *M*) veza se uvijek prikazuje posebnom relacijom, koja se sastoji od primarnih atributa za oba tipa entiteta zajedno s eventualnim atributima veze. Na primjer veza *UPISAO* iz fakultetske baze sa Slike 2.1 prikazuje se relacijom:

UPISAO (*BR_INDEKSA*, *BR_KOLEGIJA*, *DATUM_UPISA*) .

Činjenica da je jedan student upisao jedan kolegij prikazuje se jednom n-torkom u relaciji *UPISAO*. Ključ za *UPISAO* je očito složen od atributa *BR_INDEKSA* i *BR_KOLEGIJA*, jer ni jedan od ovih atributa sam nije dovoljan da jednoznačno odredi n-torku u toj relaciji.

Pretvorba involuiranih veza. Obavlja se slično kao za binarne veze. Poslužit ćemo se primjerima sa Slike 2.2. Tip entiteta *OSOBA* i (1 : 1) vezu *U-BRAKU-S* najbolje je (zbog neobaveznosti) prikazati pomoću dvije relacije:

OSOBA (*JMBG*, *PREZIME_IME*, *ADRESA*, ...) ,
BRAK (*JMBG_MUŽA*, *JMBG_ŽENE*, *DATUM_VJENČANJA*) .

Tip entiteta *SURADNIK* i (1 : *N*) vezu *JE-ŠEF-ZA* prikazujemo jednom relacijom:

SURADNIK (*ID-BR*, *ID-BR-ŠEFA*, *PREZIME_IME*, ...) .

Ovo neće uzrokovati mnogo praznih vrijednosti atributa, budući da većina suradnika ima šefa. Tip entiteta *DIO_PROIZVODA* i (*N* : *M*) vezu *SADRŽI* moramo prikazati pomoću dvije relacije:

DIO_PROIZVODA (*BR_DIJELA*, *IME_DIJELA*, *OPIS*, ...) ,
SADRŽI (*BR_DIJELA_SLOŽENOG*, *BR_DIJELA_JEDNOSTAVNOG*, *KOLIČINA*) .

Dodali smo i atribut *KOLIČINA* koji kaže koliko jednostavnih dijelova ulazi u jedan složeni.

Pretvorba pod-tipova. Pod-tip se prikazuje posebnom relacijom koja sadrži primarne atribute nadređenog tipa i atribute specifične za taj pod-tip. Na primjer hijerarhija tipova sa Slike 2.3 prikazuje se sljedećim relacijama.

OSOBA (*JMBG*, ... atributi zajednički za sve tipove osoba ...) ,
STUDENT (*JMBG*, ... atributi specifični za studente ...) ,
NASTAVNIK (*JMBG*, ... atributi specifični za nastavnike ...) ,
PROFESOR (*JMBG*, ... atributi specifični za profesore ...) .

Veza *JE* uspostavlja se na osnovu pojavljivanja istog *JMBG* u raznim relacijama.

Pretvorba ternarnih veza. Ternarna veza prikazuje se posebnom relacijom, koja sadrži primarne atribute svih triju tipova entiteta zajedno s eventualnim atributima veze. Za primjer sa Slike 2.4 imamo:

KOMPANIJA (*IME_KOMPANIJE*, ...) ,
PROIZVOD (*IME_PROIZVODA*, ...) ,
ZEMLJA (*IME_ZEMLJE*, ...) ,
IZVOZI (*IME_KOMPANIJE*, *IME_PROIZVODA*, *IME_ZEMLJE*) .

Činjenica da se jedan proizvod jedne kompanije izvozi u jednu zemlju prikazana je jednom n-torkom u relaciji *IZVOZI*. Kod ternarnih veza čija funkcionalnost nije (*M* : *N* : *P*) broj primarnih atributa je manji.

2.2.3 Usporedba relacijskog modela s mrežnim i hijerarhijskim

Mrežni i hijerarhijski model su prilično slični. Ustvari, hijerarhijski model možemo smatrati specijalnom vrstom mrežnog. S druge strane, relacijski model se po svom pristupu bitno razlikuje od ostala dva. Osnovna razlika je u načinu prikazivanja veza među entitetima, te načinu korištenja tih veza.

- U mrežnom ili hijerarhijskom modelu moguće je izravno prikazati vezu. Doduše, postoje ograničenja na funkcionalnost veze, te na konfiguraciju svih veza u shemi. Veza se “materijalizira” pomoću pointera, tj. u jednom zapisu piše adresa drugog (vezanog) zapisa. Mrežni odnosno hijerarhijski DML omogućuje samo jednostavne operacije s jednim zapisom (upis, promjena, brisanje, čitanje), te “manevriranje” kroz shemu putem veza (dohvat prvog zapisa, koji je u zadanoj vezi sa zadanim zapisom, dohvat idućeg zapisa, ...). Ovakav pristup ima svoje prednosti i mane. Prednost je da je rad s bazom u tehničkom pogledu brz i efikasan. Mana je da korisnik može upotrijebiti samo one veze koje su predviđene shemom pa su u skladu s time i materijalizirane.
- U relacijskom modelu veze su samo implicitno naznačene time što se isti ili sličan atribut pojavljuje u više relacija. Veza nije materijalizirana, već se dinamički uspostavlja za vrijeme rada s podacima, usporedbom vrijednosti atributa u n-torkama raznih relacija. Relacijski DML, osim jednostavnih operacija s jednom relacijom, mora omogućiti slobodno kombiniranje podataka iz raznih relacija. I ovaj pristup ima svoje prednosti i mane. Mana je da se veza svaki put mora iznova uspostavljati; potrebno je pretraživanje podataka, a to troši vrijeme. Prednost je da korisnik može uspostaviti i one veze koje nisu bile predviđene u fazi modeliranja podataka. Štoviše, kao kriterij za povezivanje, osim jednakosti za vrijednost atributa, mogu poslužiti i razni drugi (složeni) kriteriji. To još više povećava fleksibilnost korištenja baze.

Iz upravo rečenog vidi se da je u relacijskom modelu težište bačeno na dinamički aspekt (manje pohranjivanja a više manipuliranja). Zato upotrebljivost relacijskog DBMS bitno ovisi o izražajnim mogućnostima njegovog jezika za rad s podacima. Poželjno je također da taj jezik bude u što većoj mjeri neproceduralan i razumljiv neposrednim korisnicima. U Poglavlju 3 upoznat ćemo neke relacijske jezike. Njih treba smatrati sastavnim dijelom relacijskog modela.

2.3 Normalizacija relacijske sheme

Relacijska shema, dobivena iz ER-sheme na osnovu prethodnih uputa, može sadržavati nedorečenosti koje treba otkloniti prije implementacije. Proces daljnjeg dotjerivanja sheme zove se **normalizacija**.

Teorija normalizacije zasnovana je na pojmu normalnih formi. Relacije dobivene u skladu s potpoglavljem 2.2 morale bi u najmanju ruku biti u **prvoj normalnoj formi** (1NF). Naime, relacija je u 1NF ako je vrijednost svakog atributa jednostruka i nedjeljiva - to svojstvo već je bilo uključeno u našu definiciju relacije. U svojim radovima (1970-1974. godina) E.F. Codd je najprije definirao **drugu i treću normalnu formu** (2NF, 3NF), a zatim i poboljšanu varijantu 3NF koja se zove **Boyce-Codd-ova normalna forma** (BCNF). R. Fagin je 1977. i 1979. uveo **četvrtu i petu normalnu formu** (4NF, 5NF). U praksi je lako naići na relacije koje odstupaju od 2NF, 3NF, BCNF, no vrlo rijetko se susreću relacije u BCNF koje nisu u 4NF i 5NF. Zato su “više” normalne forme prvenstveno od teorijskog značaja.

Teorija normalizacije nije ništa drugo nego formalizacija nekih intuitivno prihvatljivih principa o “zdravom” oblikovanju sheme. Ukoliko već na početku dobro uočimo sve potrebne entitete, attribute i veze, tada nam nikakva daljnja normalizacija neće biti potrebna. No ako je polazna relacijska shema bila loše oblikovana, tada će postupak normalizacije ispraviti te greške.

2.3.1 Funkcionalna ovisnost

Za zadanu relaciju R , atribut B od R je **funkcionalno ovisan** o atributu A od R (oznaka: $A \rightarrow B$) ako vrijednost od A jednoznačno određuje vrijednost od B . Dakle ako u isto vrijeme postoje u R dvije n -torke s jednakom vrijednošću A , tada te n -torke moraju imati jednaku vrijednost B . Analogna definicija primjenjuje se i za slučaj kad su A i B složeni atributi (dakle skupovi atributa).

Kao primjer, promotrimo sljedeću (loše oblikovanu) relaciju:

IZVJEŠTAJ (BR_INDEKSA, BR_KOLEGIJA, NASLOV_KOLEGIJA, IME_NASTAVNIKA,
BR_SOBE_NASTAVNIKA, OCJENA) .

Pretpostavimo da svaki kolegij ima jednog nastavnika, a svaki nastavnik jednu sobu. Navodimo neke od funkcionalnih ovisnosti:

$(BR_INDEKSA, BR_KOLEGIJA) \rightarrow OCJENA$,
 $BR_KOLEGIJA \rightarrow NASLOV_KOLEGIJA$,

$BR_KOLEGIJA \rightarrow IME_NASTAVNIKA$,
 $BR_KOLEGIJA \rightarrow BR_SOBE_NASTAVNIKA$,
 $IME_NASTAVNIKA \rightarrow BR_SOBE_NASTAVNIKA$.

Za zadanu relaciju R , atribut B od R je **potpuno funkcionalno ovisan** o (složenom) atributu A od R ako vrijedi: B je funkcionalno ovisan o A , no B nije funkcionalno ovisan ni o jednom pravom podskupu od A .

Svaki atribut relacije je funkcionalno ovisan o ključu, no ta ovisnost ne mora biti potpuna. Na primjer $OCJENA$ je potpuno funkcionalno ovisna o primarnom ključu ($BR_INDEKSA$, $BR_KOLEGIJA$). S druge strane, $NASLOV_KOLEGIJA$, $IME_NASTAVNIKA$ i $BR_SOBE_NASTAVNIKA$ su **parcijalno** ovisni o ključu, budući da su ovisni samo o $BR_KOLEGIJA$, a ne i o $BR_INDEKSA$.

Za $BR_SOBE_NASTAVNIKA$ se kaže da je **tranzitivno** ovisan o $BR_KOLEGIJA$, budući da je ovisan o $IME_NASTAVNIKA$, koji je opet ovisan o $BR_KOLEGIJA$.

Parcijalne i tranzitivne ovisnosti mogu uzrokovati probleme kod manipuliranja s podacima, pa ih je poželjno ukloniti.

2.3.2 Druga normalna forma

Relacija je u **drugoj normalnoj formi** (2NF) ako je u 1NF i ako je svaki ne-primarni atribut potpuno funkcionalno ovisan o primarnom ključu.

Malo prije definirana relacija $IZVJEŠTAJ$ nije u 2NF, i to dovodi do anomalija:

- Ako u bazu želimo unijeti podatke o novom kolegiju, to ne možemo učiniti sve dok bar jedan student ne upiše taj kolegij (naime ne smijemo imati praznu vrijednost primarnog atributa $BR_INDEKSA$). Slično, ako želimo unijeti podatke o novom nastavniku i njegovoj sobi, to ne možemo učiniti dok nastavnika ne zadužimo s bar jednim kolegijem i dok bar jedan student ne upiše taj kolegij.
- Ako želimo promijeniti naslov kolegija 361 iz “Linearna algebra” u “Vektorski prostori”, tada moramo naći sve n -torke koje sadrže tu vrijednost za $BR_KOLEGIJA$, te promijeniti vrijednost za $NASLOV_KOLEGIJA$ u svim takvim n -torkama. Bit će onoliko promjena koliko ima studenata koji su upisali kolegij 361. Ako zaboravimo izvršiti neku od promjena, imat ćemo kontradiktorne podatke.
- Pretpostavimo da svi studenti koji su upisali kolegij 361 odustanu od tog kolegija. Ako shodno tome pobrišemo odgovarajuće n -torke, iz baze će nestati svi podaci o kolegiju 361.

Ove anomalije rješavamo svođenjem relacije u 2NF. Polaznu relaciju razbijamo u dvije, tako da iz stare relacije prebacimo u novu sve one attribute koji su parcijalno ovisni o ključu:

$IZVJEŠTAJ (BR_INDEKSA, BR_KOLEGIJA, OCJENA)$,
 $KOLEGIJ (BR_KOLEGIJA, NASLOV_KOLEGIJA, IME_NASTAVNIKA, BR_SOBE_NASTAVNIKA)$.

Objе relacije su sada u 2NF. No relacija $KOLEGIJ$ zahtijeva daljnju normalizaciju; naime u njoj još uvijek postoji tzv. tranzitivna ovisnost, gdje srednji atribut nije kandidat za ključ:

$BR_KOLEGIJA \rightarrow IME_NASTAVNIKA \rightarrow BR_SOBE_NASTAVNIKA$.

2.3.3 Treća normalna forma

Relacija je u **trećoj normalnoj formi** (3NF) ako je u 2NF i ako ne sadrži tranzitivne ovisnosti. Preciznije, relacija R je u 3NF ako za svaku funkcionalnu ovisnost $X \rightarrow A$ u R , takvu da A nije u X , vrijedi: X sadrži ključ za R ili je A primarni atribut.

Prije navedena relacija $KOLEGIJ$ nije u 3NF jer imamo ovisnost

$IME_NASTAVNIKA \rightarrow BR_SOBE_NASTAVNIKA$,

i pritom $IME_NASTAVNIKA$ nije ključ, a $BR_SOBE_NASTAVNIKA$ nije primarni atribut. Ova tranzitivna ovisnost može dovesti do anomalija:

- Ne možemo unijeti podatke o novom nastavniku i njegovoj sobi, sve dok ga nismo zadužili s kolegijem.
- Da bi promijenili broj sobe nastavnika, moramo izvršiti promjenu u svakoj n-torki koja odgovara nekom kolegiju kojeg predaje taj nastavnik.
- Ako nastavnik (privremeno) ne predaje ni jedan kolegij, tada iz baze nestaju svi podaci o tom nastavniku i njegovoj sobi.

Da bi relaciju *KOLEGIJ* prebacili u 3NF, razbijamo je u dvije, i time prekidamo tranzitivnu ovisnost. Konačna relacijska shema koja zamjenjuje polaznu relaciju *IZVJEŠTAJ* izgleda ovako:

IZVJEŠTAJ (*BR_INDEKSA*, *BR_KOLEGIJA*, *OCJENA*) ,
KOLEGIJ (*BR_KOLEGIJA*, *NASLOV_KOLEGIJA*, *IME_NASTAVNIKA*) ,
NASTAVNIK (*IME_NASTAVNIKA*, *BR_SOBE_NASTAVNIKA*) .

Relacije su sada do kraja normalizirane. Primijetimo da bi konačnu shemu odmah dobili da smo u fazi modeliranja entiteta postupili ispravno, te studente, kolegije i nastavnike odmah prikazali posebnim tipovima entiteta.

2.3.4 Boyce-Codd-ova normalna forma

Determinanta je atribut (ili kombinacija atributa) o kojem je neki drugi atribut potpuno funkcionalno ovisan.

Relacija je u **Boyce-Codd-ovoj normalnoj formi** (BCNF) ako je svaka njezina determinanta ujedno i kandidat za ključ.

Očito je relacija koja je u BCNF također i u 2NF i 3NF. No postoje relacije koje su u 3NF, no nisu u BCNF. Primjer za to možemo konstruirati tako da gledamo relaciju u kojoj postoje dva kandidata za ključ, oba ključa su složena, i preklapaju se u bar jednom atributu.

Na primjer neka na fakultetu jedan kolegij predaje više nastavnika, ali svaki nastavnik predaje samo jedan kolegij. Svaki student upisuje više kolegija, no ima samo jednog nastavnika za zadani kolegij. Situacija se može opisati relacijom:

UPISAO (*BR_INDEKSA*, *IME_NASTAVNIKA*, *BR_KOLEGIJA*) .

Relacija nije ni u 2NF, jer postoji parcijalna ovisnost:

$IME_NASTAVNIKA \rightarrow BR_KOLEGIJA$.

No mi možemo drukčije izabrati primarni ključ:

UPISAO (*BR_INDEKSA*, *BR_KOLEGIJA*, *IME_NASTAVNIKA*) .

Sad je relacija u 3NF, no ne i u BCNF jer postoji ovisnost:

$IME_NASTAVNIKA \rightarrow BR_KOLEGIJA$

i pritom *IME_NASTAVNIKA* nije kandidat za ključ.

Zbog odstupanja od BCNF nastaju slične anomalije kao kod odstupanja od 3NF. Ne možemo evidentirati činjenicu da zadani nastavnik predaje zadani kolegij, sve dok bar jedan student ne upiše taj kolegij kod tog nastavnika. Također, veza nastavnika i kolegija je zapisana s velikom redundancijom, onoliko puta koliko ima studenata, što otežava ažuriranje. Ako svi studenti odustanu, briše se evidencija da zadani nastavnik predaje zadani kolegij.

Rješenje problema je, kao i prije, razbijanje relacije na dvije. Prekida se nepoželjna funkcionalna ovisnost.

KLASA (*BR_INDEKSA*, *IME_NASTAVNIKA*) ,
PREDAJE (*IME_NASTAVNIKA*, *BR_KOLEGIJA*) .

Sad su obje relacije u BCNF.

Problemi s relacijom *UPISAO* izbjegli bi se da smo već u fazi modeliranja entiteta i veza uočili da zapravo imamo posla s dvije nezavisne binarne veze: (*N* : *M*) veza između *STUDENT* i *NASTAVNIK*, te (1 : *N*) veza između *KOLEGIJ* i *NASTAVNIK*. Ternarna veza je tada suvišna.

2.3.5 Višeznačna ovisnost i četvrta normalna forma

Četvrtu normalnu formu najlakše je opisati pomoću primjera. Promatrajmo relaciju koja prikazuje vezu između kompanija, proizvoda i zemalja:

IZVOZI (*KOMPANIJA*, *PROIZVOD*, *ZEMLJA*) .

Jedna n-torka označava da zadana kompanija zadani proizvod izvozi u zadanu zemlju. Relacija može u jednom trenutku izgledati kao na Tabelarnom prikazu 2.2. Lagano je provjeriti da je relacija u BCNF.

<i>IZVOZI</i>		
<i>KOMPANIJA</i>	<i>PROIZVOD</i>	<i>ZEMLJA</i>
IBM	Desktop	Francuska
IBM	Desktop	Italija
IBM	Desktop	Velika Britanija
IBM	Mainframe	Francuska
IBM	Mainframe	Italija
IBM	Mainframe	Velika Britanija
HP	Desktop	Francuska
HP	Desktop	Španjolska
HP	Desktop	Irska
HP	Server	Francuska
HP	Server	Španjolska
HP	Server	Irska
Fujitsu	Mainframe	Italija
Fujitsu	Mainframe	Francuska

Tabelarni prikaz 2.2: Relacija koja nije u četvrtoj normalnoj formi.

U nastavku uzimamo da vrijedi pravilo: čim kompanija izvozi u neku zemlju, ona odmah izvozi sve svoje proizvode u tu zemlju. Tada relacija očito sadrži veliku dozu redundancije. Na primjer, da bi dodali novi proizvod IBM-a, moramo upisati n-torku za svaku zemlju u koju IBM izvozi. Slično, ako HP počne izvoziti u Kinu, morat će se ubaciti posebna n-torka za svaki njegov proizvod.

Redundancija će se eliminirati ako zamijenimo polaznu relaciju *IZVOZI* s dvije manje relacije *RADI* i *PRODAJE*:

RADI (*KOMPANIJA*, *PROIZVOD*) ,
PRODAJE (*KOMPANIJA*, *ZEMLJA*) .

Podacima iz prethodnog Tabelarnog prikaza 2.2 tada odgovaraju podaci iz Tabelarnog prikaza 2.3.

<i>RADI</i>		<i>PRODAJE</i>	
<i>KOMPANIJA</i>	<i>PROIZVOD</i>	<i>KOMPANIJA</i>	<i>ZEMLJA</i>
IBM	Desktop	IBM	Francuska
IBM	Mainframe	IBM	Italija
HP	Desktop	IBM	Velika Britanija
HP	Server	HP	Francuska
Fujitsu	Mainframe	HP	Španjolska
		HP	Irska
		Fujitsu	Italija
		Fujitsu	Francuska

Tabelarni prikaz 2.3: Svođenje na četvrtu normalnu formu.

Dosadašnja pravila normalizacije nam ne pomažu da eliminiramo redundanciju u relaciji *IZVOZI*. To je zato što redundancija nije bila uzrokovana funkcionalnim ovisnostima, već tzv. višeznačnim ovisnostima:

$KOMPANIJA \rightarrow\rightarrow PROIZVOD$,
 $KOMPANIJA \rightarrow\rightarrow ZEMLJA$.

Zadana je relacija $R(A, B, C)$. **Višeznačna ovisnost** $A \rightarrow\rightarrow B$ vrijedi ako: skup B -vrijednosti koje se u R pojavljuju uz zadani par (A -vrijednost, C -vrijednost) ovisi samo o A -vrijednosti, a ne i o C -vrijednosti. Ista definicija primjenjuje se i kad su A , B i C složeni atributi (dakle skupovi atributa).

U gornjem primjeru, skup zemalja u koje zadana kompanija izvozi zadani proizvod ovisi samo o kompaniji a ne i o proizvodu. Slično, skup proizvoda koje zadana kompanija izvozi u zadanu zemlju ovisi samo o kompaniji a ne o zemlji.

Relacija R je u **četvrtjoj normalnoj formi** (4NF) ako vrijedi: kad god postoji višeznačna ovisnost u R , na primjer $A \rightarrow\rightarrow B$, tada su svi atributi od R funkcionalno ovisni o A .

Ekvivalentno, R je u 4NF ako je u BCNF i sve višeznačne ovisnosti u R su zapravo funkcionalne ovisnosti.

U našoj relaciji *IZVOZI*, ni jedna od uočenih višeznačnih ovisnosti nije funkcionalna ovisnost. Znači, *IZVOZI* nije u 4NF i zato je treba rastaviti na *RADI* i *PRODAJE* (koje jesu u 4NF).

Odstupanje od 4NF opet možemo tumačiti kao grešku u modeliranju entiteta i veza. Promatrana relacija *IZVOZI* nastala je zbog pokušaja da se odnos triju tipova entiteta *KOMPANIJA*, *PROIZVOD*, *ZEMLJA* tretira kao ternarna veza. Zapravo se ovdje radi o dvije nezavisne binarne veze.

Postoje, naravno, i prave ternarne veze. Na primjer ako skup proizvoda koje zadana kompanija izvozi varira od zemlje do zemlje, tada prije uočene višeznačne ovisnosti ne vrijede, relacija *IZVOZI* je u 4NF i ne možemo je rastaviti na dvije manje relacije.

2.3.6 Razlozi zbog kojih se može odustati od normalizacije

Za većinu praktičnih primjera dovoljno je relacije normalizirati do 3NF. Koji put je potrebno neku relaciju i dalje normalizirati do BCNF ili 4NF. Peta normalna forma, koja se također navodi u literaturi, nije od praktičnog značaja.

Postoje razlozi zbog kojih iznimno možemo odustati od pune normalizacije. Navesti ćemo dva takva razloga.

Složeni atribut . Dešava se da nekoliko atributa u relaciji čine cjelinu koja se u aplikacijama nikad ne rastavlja na sastavne dijelove. Na primjer, promatrajmo relaciju

KUPAC (*PREZIME_IME*, *POŠTANSKI_BROJ*, *GRAD*, *ULICA*) .

Strogo govoreći, *GRAD* je funkcionalno ovisan o *POŠTANSKI_BROJ*, pa relacija nije u 3NF. No mi znamo da *POŠTANSKI_BROJ*, *GRAD* i *ULICA* čine cjelinu koja se zove adresa. Budući da se podaci iz adrese koriste i ažuriraju “u paketu”, ne može doći do prije spominjanih anomalija. Nije preporučljivo razbijati ovu relaciju na dvije.

Efikasno čitanje podataka . Normalizacijom se velike relacije razbijaju na mnogo manjih. Kod čitanja je često potrebno podatke iz malih relacija ponovo sastaviti u veće n-torke. Uspostavljanje veza među podacima u manjim relacijama traje znatno dulje nego čitanje podataka koji su već povezani i upisani u jednu veliku relaciju.

Projektant baze podataka treba procijeniti kada treba provesti normalizaciju do kraja a kada ne. Za tu procjenu je važno razumijevanje značenja podataka i načina kako će se oni koristiti.

3

JEZICI ZA RELACIJSKE BAZE PODATAKA

3.1 Relacijska algebra

Relacijsku algebru uveo je E.F. Codd u svojim radovima iz 70-tih godina 20. stoljeća. Riječ je o teorijskoj (matematičkoj) notaciji, a ne o praktičnom jeziku kojeg bi ljudi zaista neposredno koristili. Zato niti ne postoji standardna sintaksa.

Relacijska algebra se svodi na izvrednjavanje algebarskih izraza, građenih od relacija te unarnih i binarnih operatora (čiji operandi su relacija a rezultat je opet relacija). Svaki algebarski izraz predstavlja jedan upit (pretraživanje).

U ovom i idućim poglavljima, kao primjer će služiti pojednostavnjena (engleska) verzija naše baze o fakultetu:

STUDENT (*SNO*, *SNAME*, *LEVEL*) ,
COURSE (*CNO*, *TITLE*, *LNAME*) ,
REPORT (*SNO*, *CNO*, *MARK*) ,
LECTURER (*LNAME*, *ROOMNO*) .

Uzimamo da relacije sadrže n-torke koje se vide u Tabelarnom prikazu 3.1.

<i>STUDENT</i>			<i>COURSE</i>		
<i>SNO</i>	<i>SNAME</i>	<i>LEVEL</i>	<i>CNO</i>	<i>TITLE</i>	<i>LNAME</i>
876543	Jones	2	216	Database Systems	Black
864532	Burns	1	312	Software Engineering	Welsh
856434	Cairns	3	251	Numerical Analysis	Quinn
876421	Hughes	2	121	Compilers	Holt

<i>REPORT</i>			<i>LECTURER</i>	
<i>SNO</i>	<i>CNO</i>	<i>MARK</i>	<i>LNAME</i>	<i>ROOMNO</i>
876543	216	82	Black	1017
864532	216	75	Welsh	1024
864532	312	71	Holt	2014
856434	121	49	Quinn	1010
876421	312	39		
876543	251	70		
864532	251	69		
864532	121	78		

Tabelarni prikaz 3.1: Demo-baza o fakultetu.

Relacija *STUDENT* popisuje studente, *COURSE* nabraja kolegije, a *LECTURER* sadrži podatke o nastavnicima. Student je jednoznačno određen svojim brojem iz indeksa *SNO*, kolegij je jednoznačno

određen svojom šifrom *CNO*, a nastavnici se razlikuju po svojim imenima *LNAME*. Za svakog studenta pamtimo njegovo ime *SNAME* i godinu studija *LEVEL*. Za kolegij se pamti njegov naslov *TITLE* i ime (jednog) nastavnika koji ga predaje *LNAME*. Za svakog nastavnika poznat nam je broj njegove sobe *ROOMNO*. Relacija *REPORT* bilježi koji je student upisao koji kolegij i koju ocjenu *MARK* je dobio.

3.1.1 Skupovni operatori

Relacije su ustvari skupovi *n*-torki. Zato na njih možemo primjenjivati uobičajene skupovne operatore. Neka *R* i *S* označavaju relacije. Tada je:

R union S ... skup *n*-torki koje su u *R* ili u *S* (ili u obje relacije).

R intersect S ... skup *n*-torki koje su u *R* i također u *S*.

R minus S ... skup *n*-torki koje su u *R* no nisu u *S*.

Da bi se operatori mogli primijeniti, relacije *R* i *S* moraju biti “kompatibilne”, to jest moraju imati isti stupanj i iste attribute (ista imena i tipove). Primijetimo da uvijek vrijedi:

R intersect S = *R minus (R minus S)* .

Znači, **intersect** nije nužan.

Za primjer, promatramo relaciju *NEW_STUDENT*, građenu kao *STUDENT*, u kojoj se nalaze *n*-torke popisane u Tabelarnom prikazu 3.2. Tada primjenom skupovnih operacija dobivamo rezultate iz Tabelarnog prikaza 3.3.

NEW_STUDENT

<i>SNO</i>	<i>SNAME</i>	<i>LEVEL</i>
876342	Smith	3
865698	Turner	2
875923	Murphy	2
856434	Cairns	3
871290	Noble	1

Tabelarni prikaz 3.2: Relacija kompatibilna sa *STUDENT*.

STUDENT union NEW_STUDENT

<i>SNO</i>	<i>SNAME</i>	<i>LEVEL</i>
876543	Jones	2
864532	Burns	1
856434	Cairns	3
876421	Hughes	2
876342	Smith	3
865698	Turner	2
875923	Murphy	2
871290	Noble	1

STUDENT minus NEW_STUDENT

<i>SNO</i>	<i>SNAME</i>	<i>LEVEL</i>
876543	Jones	2
864532	Burns	1
876421	Hughes	2

STUDENT intersect NEW_STUDENT

<i>SNO</i>	<i>SNAME</i>	<i>LEVEL</i>
856434	Cairns	3

Tabelarni prikaz 3.3: Rezultati skupovnih operacija.

3.1.2 Selekcija

Selekcija je unarni operator koji izvlači iz relacije one n-torke koje zadovoljavaju zadani Booleovski uvjet. Selekciju na relaciji R u skladu s Booleovskim uvjetom \mathcal{B} označavamo s R **where** \mathcal{B} . Uvjet \mathcal{B} je formula koja se sastoji od:

- operanada koji su ili konstante ili atributi,
- operatora za uspoređivanje $=, <, >, \leq, \geq, \neq$,
- logičkih operatora **and**, **or**, **not**.

Navest ćemo nekoliko primjera od kojih svaki sadrži jedan upit sa selekcijom i odgovarajući izraz u relacijskoj algebri. Izračunate vrijednosti izraza (odgovori na upite) vide se u Tabelarnom prikazu 3.4.

Upit 1: Pronađi sve studente na stupnju (godini) 1.

$RESULT1 := STUDENT$ **where** $LEVEL = 1$.

Upit 2: Pronađi sve kolegije s naslovom ‘Database Systems’.

$RESULT2 := COURSE$ **where** $TITLE = \text{‘Database Systems’}$.

Upit 3: Pronađi sve studente koji su iz kolegija 216 dobili ocjenu veću od 70.

$RESULT3 := REPORT$ **where** $((CNO=216) \text{ and } (MARK>70))$.

$RESULT1$			$RESULT2$		
SNO	$SNAME$	$LEVEL$	CNO	$TITLE$	$LNAME$
864532	Burns	1	216	Database Systems	Black

$RESULT3$		
SNO	CNO	$MARK$
876543	216	82
864532	216	75

Tabelarni prikaz 3.4: Rezultati selekcije.

3.1.3 Projekcija

Projekcija je unarni operator koji iz relacije izvlači zadane attribute, s time da se u rezultirajućoj relaciji eliminiraju n-torke duplikati. Projekciju relacije R na njene attribute A_1, A_2, \dots, A_m označavat ćemo s $R[A_1, A_2, \dots, A_m]$. Smatrat ćemo da projekcija ima viši prioritet od ostalih operacija.

Navest ćemo dva primjera koji se sastoje od upita s projekcijom i odgovarajućeg izraza u relacijskoj algebri. Izračunate vrijednosti izraza (odgovori na upite) vide se u Tabelarnom prikazu 3.5.

Upit 1: Pronađi brojeve soba od svih nastavnika.

$RESULT1 := LECTURER[ROOMNO]$.

Upit 2: Pronađi ime nastavnika koji predaje kolegij 312.

$RESULT2 := (COURSE \text{ where } CNO = 312)[LNAME]$.

<i>RESULT1</i>		<i>RESULT2</i>	
<i>ROOMNO</i>		<i>LNAME</i>	
1017			
1024			
2014			
1010			
		Welsh	

Tabelarni prikaz 3.5: Rezultati projekcije.

3.1.4 Kartezijev produkt

Neka su R i S relacije stupnja n_1 odnosno n_2 . Tada algebarski izraz R **times** S daje Kartezijev produkt od R i S , dakle skup svih $(n_1 + n_2)$ -torki čijih prvih n_1 komponenti čine n_1 -torku u R , a zadnjih n_2 komponenti čine n_2 -torku u S .

Atribut u R **times** S ima isto ime kao odgovarajući atribut u R odnosno S , s time da se po potrebi to ime proširuje imenom polazne relacije (slično kao za komponentu zapisa u C-u).

Kao primjer korištenja Kartezijevog produkta, ispisat ćemo za svakog studenta sve kolegije koje on nije upisao:

$ALL_COMB := STUDENT[SNO]$ **times** $COURSE[CNO]$,
 $DO_NOT_TAKE := ALL_COMB$ **minus** $REPORT[SNO, CNO]$.

Drugi primjer pronalazi sve parove studenata koji su na istom stupnju (godini). Za to nam je potrebno napraviti Kartezijev produkt relacije $STUDENT$ sa samom sobom. Da ne bi došlo do pometnje s imenima atributa. specijalnim operatorom **aliases** uvodimo “pseudonim” (drugo ime, nadimak) za relaciju $STUDENT$:

$TEMP$ **aliases** $STUDENT$,
 $PAIRS := ((TEMP$ **times** $STUDENT)$
 where $((TEMP.LEVEL = STUDENT.LEVEL)$
 and $(TEMP.SNO < STUDENT.SNO)))$
 $[TEMP.SNAME, STUDENT.SNAME]$.

Rezultati za oba upita vide se u Tabelarnom prikazu 3.6.

<i>DO_NOT_TAKE</i>		<i>PAIRS</i>	
<i>SNO</i>	<i>CNO</i>	<i>TEMP.SNAME</i>	<i>STUDENT.SNAME</i>
876543	312		
876543	121		
856434	216		
856434	312		
856434	251		
876421	216		
876421	251		
876421	121		
		Hughes	Jones

Tabelarni prikaz 3.6: Rezultati primjene Kartezijevog produkta.

3.1.5 Prirodni spoj

Prirodni spoj (natural join) je binarni operator primjenjiv na dvije relacije R i S koje imaju bar jedan zajednički atribut. R **join** S sastoji se od svih n -torki dobivenih spajanjem jedne n -torke iz R s jednom n -torkom iz S koja ima iste vrijednosti zajedničkih atributa. U rezultirajućoj relaciji zajednički atribut se pojavljuje samo jednom.

Primjer za prirodni spoj vidi se u Tabelarnom prikazu 3.7. Pretpostavlja se da su zajednički atributi dviju relacija točno oni koji imaju ista imena.

Prirodni spoj omogućuje nam da u našoj fakultetskoj bazi podataka odgovorimo na složenije upite koji zahtijevaju uspostavljanje veza između podataka u raznim tablicama.

<i>R</i>			<i>S</i>		
<i>A</i>	<i>B</i>	<i>C</i>	<i>B</i>	<i>C</i>	<i>D</i>
<i>a</i> ₁	<i>b</i> ₁	<i>c</i> ₁	<i>b</i> ₁	<i>c</i> ₁	<i>d</i> ₁
<i>a</i> ₂	<i>b</i> ₁	<i>c</i> ₁	<i>b</i> ₁	<i>c</i> ₁	<i>d</i> ₂
<i>a</i> ₃	<i>b</i> ₂	<i>c</i> ₂	<i>b</i> ₂	<i>c</i> ₃	<i>d</i> ₃
<i>a</i> ₄	<i>b</i> ₂	<i>c</i> ₃			

<i>R join S</i>			
<i>A</i>	<i>B</i>	<i>C</i>	<i>D</i>
<i>a</i> ₁	<i>b</i> ₁	<i>c</i> ₁	<i>d</i> ₁
<i>a</i> ₁	<i>b</i> ₁	<i>c</i> ₁	<i>d</i> ₂
<i>a</i> ₂	<i>b</i> ₁	<i>c</i> ₁	<i>d</i> ₁
<i>a</i> ₂	<i>b</i> ₁	<i>c</i> ₁	<i>d</i> ₂
<i>a</i> ₄	<i>b</i> ₂	<i>c</i> ₃	<i>d</i> ₃

Tabelarni prikaz 3.7: Primjer prirodnog spoja.

Upit 1: Pronađi imena svih studenata koji su upisali kolegij 251.

$RESULT1 := ((REPORT \textbf{ where } CNO = 251) \textbf{ join } STUDENT) [SNAME] .$

Upit 2: Pronađi broj sobe nastavnika koji predaje kolegij 351.

$RESULT2 := ((COURSE \textbf{ where } CNO = 351) \textbf{ join } LECTURER) [ROOMNO] .$

Upit 3: Pronađi imena nastavnika koji predaju kolegije koje je upisao bar jedan student na stupnju (godini) 2.

$RESULT3 := (((STUDENT \textbf{ where } LEVEL=2) \textbf{ join } REPORT) \textbf{ join } COURSE) [LNAME] .$

U Tabelarnom prikazu 3.8 nalazi se detaljnije izvrednjavanje Upita 3.

<i>STUDENT where LEVEL = 2</i>			<i>(STUDENT where LEVEL=2) join REPORT</i>				
<i>SNO</i>	<i>SNAME</i>	<i>LEVEL</i>	<i>SNO</i>	<i>SNAME</i>	<i>LEVEL</i>	<i>CNO</i>	<i>MARK</i>
876543	Jones	2	876543	Jones	2	216	82
876421	Hughes	2	876543	Jones	2	251	70
			876421	Hughes	2	312	39

<i>((STUDENT where LEVEL=2) join REPORT) join COURSE</i>							<i>RESULT3</i>
<i>SNO</i>	<i>SNAME</i>	<i>LEVEL</i>	<i>CNO</i>	<i>MARK</i>	<i>TITLE</i>	<i>LNAME</i>	<i>LNAME</i>
876543	Jones	2	216	82	Database Systems	Black	Black
876543	Jones	2	251	70	Numerical Analysis	Quinn	Quinn
876421	Hughes	2	312	39	Software Engineering	Welsh	Welsh

Tabelarni prikaz 3.8: Izvrednjavanje Upita 3 s prirodnim spojem.

Prirodni spoj uvijek se može izraziti preko ostalih operatora. Na primjer, za relacije $R(A, B, C)$ i $S(C, D)$ vrijedi:

$R \textbf{ join } S = ((R \textbf{ times } S) \textbf{ where } R.C = S.C) [A, B, C, D].$

3.1.6 Theta-spoj

Theta-spoj relacija R i S , pisano $R \text{ join}(A \theta B) S$, je skup onih n -torki Kartezijevog produkta R sa S za koje je predikat $R.A \theta S.B$ istina. Simbol θ predstavlja jedan od operatora za usporedbu ($=, >, <, \neq, \leq, \geq$). Theta-spoj se uvijek može izraziti pomoću Kartezijevog produkta i selekcije. Prirodni spoj se može smatrati specijalnim slučajem theta-spoja.

3.1.7 Dijeljenje

Neka je R relacija stupnja n , a S relacija stupnja m , i neka se svi atributi od S pojavljuju i u R . Rezultat dijeljenja R sa S , oznakom $R \text{ divideby } S$, je skup svih $(n - m)$ -torki $\langle x \rangle$ takvih da se n -torke $\langle x, y \rangle$ pojavljuju u R za sve m -torke $\langle y \rangle$ u S . Ovdje x i y predstavljaju grupu od jedne ili više vrijednosti atributa.

U Tabelarnom prikazu 3.9 nalazi se najprije jedan apstraktni primjer dijeljenja. Dalje slijede primjeri s našom fakultetskom bazom podataka.

$R1$									
SNO	CNO			$R2$	$R3$	$R1 \text{ divideby } R2$		$R1 \text{ divideby } R3$	
s_1	c_1			CNO	CNO	SNO		SNO	
s_1	c_2			c_1	c_1	s_1		s_1	
s_1	c_3				c_2	s_2			
s_2	c_1				c_3	s_3			
s_2	c_2								
s_3	c_1								
s_4	c_4								

Tabelarni prikaz 3.9: Apstraktni primjer djeljenja.

Upit 1: Pronađi imena studenata koji su upisali sve kolegije.

$RESULT1 := ((REPORT[SNO,CNO] \text{ divideby } COURSE[CNO]) \text{ join } STUDENT) [SNAME]$.

Za naše konkretne podatke, jedini student koji zadovoljava upit je Burns.

Upit 2: Pronađi brojeve onih studenata koji su upisali barem one kolegije koje je upisao student s brojem 856434 (i možda još neke kolegije).

$RESULT2 := REPORT [SNO,CNO] \text{ divideby } (REPORT \text{ where } SNO=856434)[CNO]$.

Kao odgovor dobivamo brojeve studenata 856434, 864532.

Operator dijeljenja predstavlja način implementiranja univerzalne kvantifikacije \forall u relacijskoj algebri. Dijeljenje se također može izraziti pomoću prethodno opisanih operatora. Na primjer, ako imamo relacije $R(A, B, C, D)$ i $S(C, D)$, tada je:

$R \text{ divideby } S = R[A,B] \text{ minus } ((R[A,B] \text{ times } S) \text{ minus } R) [A,B]$.

3.1.8 Vanjski spoj

Vanjski spoj (outer join) je binarni operator vrlo sličan prirodnom spoju. Primjenjiv je pod istim uvjetima i daje kao rezultat relaciju s istom građom (shemom). No sadržaj od $R \text{ outerjoin } S$ je nešto bogatiji nego sadržaj $R \text{ join } S$. Naime, pored svih n -torki iz $R \text{ join } S$, relacija $R \text{ outerjoin } S$ sadrži i sve “nesparene” (nespojene) n -torke iz R odnosno S (s time da su te nesparene n -torke na odgovarajući način proširene null vrijednostima).

Dakle za isti primjer R i S kao u Tabelarnom prikazu 3.7, R **outerjoin** S izgleda kao na Tabelarnom prikazu 3.10.

R			S		
A	B	C	B	C	D
a_1	b_1	c_1	b_1	c_1	d_1
a_2	b_1	c_1	b_1	c_1	d_2
a_3	b_2	c_2	b_2	c_3	d_3
a_4	b_2	c_3			

R outerjoin S			
A	B	C	D
a_1	b_1	c_1	d_1
a_1	b_1	c_1	d_2
a_2	b_1	c_1	d_1
a_2	b_1	c_1	d_2
a_4	b_2	c_3	d_3
a_3	b_2	c_2	—

Tabelarni prikaz 3.10: Primjer vanjskog spoja.

Vanjski spoj obično se koristi za traženje podataka koji ne zadovoljavaju neki uvjet. Na primjer:

Upit 1: Pronađi imena studenata koji nisu upisali ni jedan kolegij.

$RESULT1 := ((STUDENT \text{ outerjoin } REPORT) \text{ where } (CNO \text{ is null})) [SNAME]$.

3.2 Relacijski račun

Relacijski račun također je bio predložen od E.F. Codd-a, kao alternativa relacijskoj algebri. Riječ je o matematičkoj notaciji zasnovanoj na predikatnom računu. Upit se izražava tako da zadamo predikat kojeg n -torke moraju zadovoljavati.

Postoje dvije vrste relacijskog računa: račun orijentiran na n -torke (gdje su osnovni objekti n -torke) i račun orijentiran na domene (gdje su osnovni objekti vrijednosti iz domena za atribut).

3.2.1 Račun orijentiran na n -torke

U izrazima se pojavljuju sljedeći elementi:

Varijable- n -torke poprimaju vrijednosti iz imenovane relacije. Ako je t varijabla koja “prolazi” relacijom R , tada $t.A$ označava A -komponentu od t , gdje je A atribut od R .

Uvjeti oblika $x \theta y$, gdje je θ operator za uspoređivanje ($=, <, >, \neq, \leq, \geq$). Bar jedno od x i y mora biti oblika $t.A$, a drugo može biti konstanta. Uvjeti također mogu biti oblika $R(t)$, što znači da je t n -toraka u relaciji R .

Dobro oblikovane formule (WFF) građene od logičkih veznika (**and**, **or**, **not**) i kvantifikatora \exists (postoji) i \forall (za svako), u skladu sa sljedećim pravilima:

- Svaki uvjet je WFF.
- Ako su f_1 i f_2 WFF, tada su to i f_1 **and** f_2 , f_1 **or** f_2 , **not** f_1 , i **not** f_2 .
- Ako je f jedna WFF u kojoj se t pojavljuje kao **slobodna** varijabla, tada su $\exists t(f)$ i $\forall t(f)$ također WFF. (Pojava varijable u WFF je **vezana** ukoliko je ta varijabla uvedena kvantifikatorom. Inače je slobodna).
- Ništa drugo nije WFF.

Izraz računa orijentiranog na n-torke je oblika: $\{t.A, u.B, v.C, \dots \mid f\}$, gdje su t, u, v, \dots varijable n-torke, A, B, C, \dots su atributi odgovarajućih relacija, a f je WFF koja sadrži t, u, v, \dots kao slobodne varijable. Dalje slijede primjeri vezani uz našu fakultetsku bazu podataka (potpoglavlje 3.1).

Upit 1: Pronađi sve brojeve kolegija.

$$\{c.CNO \mid COURSE(c)\} .$$

Upit 2: Pronađi brojeve svih studenata na stupnju 1.

$$\{s.SNO \mid STUDENT(s) \textbf{ and } s.LEVEL = 1\} .$$

Upit 3: Pronađi brojeve i imena studenata koji su upisali kolegij 121.

$$\{s.SNO, s.SNAME \mid STUDENT(s) \textbf{ and } \exists r (REPORT(r) \textbf{ and } r.SNO = s.SNO \textbf{ and } r.CNO = 121)\} .$$

Upit 4: Pronađi brojeve onih kolegija koje je upisao bar jedan student na stupnju 1.

$$\{c.CNO \mid COURSE(c) \textbf{ and } \exists r (REPORT(r) \textbf{ and } r.CNO = c.CNO \textbf{ and } \exists s (STUDENT(s) \textbf{ and } s.SNO = r.SNO \textbf{ and } s.LEVEL = 1))\} .$$

Upit 5: Pronađi imena onih studenata koji su upisali sve kolegije.

$$\{s.SNAME \mid STUDENT(s) \textbf{ and } \forall c \exists r (COURSE(c) \textbf{ and } REPORT(r) \textbf{ and } c.CNO = r.CNO \textbf{ and } r.SNO = s.SNO)\} .$$

3.2.2 Račun orijentiran na domene

Varijable “prolaze” domenama a ne relacijama. Također, imamo takozvane **uvjete članstva** oblika: $R(A : v_1, B : v_2, C : v_3, \dots)$, gdje su A, B, C, \dots atributi od relacije R , a $v_1, v_2, v_3 \dots$ su ili varijable ili konstante. Na primjer, uvjet $STUDENT(SNO : 872501, LEVEL : 1)$ je istina ako i samo ako postoji n-torka u relaciji $STUDENT$ takva da je $SNO = 872501, LEVEL = 1$. Pravila za WFF su ista kao u računu orijentiranom na n-torke. Slijede primjeri upita za našu fakultetsku bazu podataka.

Upit 1: Pronađi sve brojeve kolegija.

$$\{C \mid COURSE(CNO : C)\} .$$

Upit 2: Pronađi brojeve svih studenata na stupnju 1.

$$\{S \mid STUDENT(SNO : S, LEVEL : 1)\} .$$

Upit 3: Pronađi brojeve i imena studenata koji su upisali kolegij 121.

$$\{S, N \mid STUDENT(SNO : S, SNAME : N) \textbf{ and } REPORT(SNO : S, CNO : 121)\} .$$

Upit 4: Pronađi brojeve i naslova kolegija koje je upisao bar jedan student na stupnju 1.

$$\{C, T \mid COURSE(CNO : C, TITLE : T) \textbf{ and } \exists S (REPORT(CNO : C, SNO : S) \textbf{ and } STUDENT(SNO : S, LEVEL : 1))\} .$$

Upit 5: Pronađi imena onih studenata koji su upisali sve kolegije.

$$\{N \mid \exists S (STUDENT(SNO : S, SNAME : N) \textbf{ and } \forall C (\textbf{if } COURSE(CNO : C) \textbf{ then } REPORT(SNO : S, CNO : C))))\}$$

(ovdje se implikacija **if ... then** može zapisati pomoću **and**, **or** i **not**).

Relacijski račun je u većoj mjeri “neproceduralan” nego relacijska algebra. Stoga praktični relacijski jezici (namijenjeni neposrednim korisnicima) više liče na račun nego na algebru. Najrašireniji današnji jezik SQL zasnovan je na računu orijentiranom na n-torke. Drugi poznati jezik QBE (Query By Example) koristi račun orijentiran na domene.

3.3 Jezik SQL

Structured Query Language (SQL) razvio je IBM u sklopu projekta “System R” (Chamberlin i drugi, kasne 70-te godine 20. stoljeća). Jezik se postepeno usavršavao, a njegova dotjerana varijanta pojavljuje se u današnjem IBM-ovom relacijskom DBMS-u zvanom DB2. Druge softverske kuće (na primjer Oracle Corporation) ugradile su SQL u svoje DBMS-e, te ga time učinile vrlo popularnim i dostupnim na svim važnijim računalnim platformama. Preostale kuće (INGRES Corporation, DEC, ...) koje su razvijale svoje jezike, bile su prisiljene da se prilagode SQL-u. Zbog pojave raznih “dijalekata” donesen je ISO/ANSI standard za SQL (zadnja verzija 1998. godine).

SQL je uglavnom zasnovan na relacijskom računu, s time da je matematička notacija zamijenjena ključnim riječima nalik na govorni engleski jezik. No lagano se realiziraju i sve operacije iz relacijske algebre. Osim postavljanja upita, jezik također omogućuje: definiranje relacija, ažuriranje relacija (upis, promjena, brisanje n-torki), sortiranje i formatiranje ispisa, neke aritmetičke operacije s podacima, definiranje “pogleda” (virtuelnih relacija izvedenih iz postojećih), utjecaj na fizičku građu baze (na primjer stvaranje tzv. indeksa), te kontrolu sigurnosti. SQL ćemo detaljnije proraditi na vježbama. Sada ćemo navesti samo nekoliko primjera.

3.3.1 Postavljanje upita

Upit se postavlja fleksibilnom naredbom **SELECT** (nije isto što i operacija selekcije u relacijskoj algebri). Rezultat upita se shvaća kao nova privremena relacija, izvedena iz stalnih (po tome je SQL sličan relacijskoj algebri).

Upit 1: Pronađi brojeve i imena svih studenata na stupnju 1.

```
SELECT SNO, SNAME
FROM STUDENT
WHERE LEVEL = 1;
```

ili (ukoliko želimo uzlazno sortirani ispis)

```
SELECT SNO, SNAME
FROM STUDENT
WHERE LEVEL = 1
ORDER BY SNAME;
```

Upit 2: Pronađi brojeve i imena studenata koji su upisali kolegij 121.

```
SELECT STUDENT.SNO, STUDENT.SNAME
FROM STUDENT, REPORT
WHERE STUDENT.SNO = REPORT.SNO
AND REPORT.CNO = 121;
```

Ovdje smo morali proširiti imena atributa da bi izbjegli dvoznačnost. Vidimo kako SQL **SELECT** naredba zamjenjuje prirodni spoj iz relacijske algebre. Štoviše, to je mogao biti i theta-spoj. Drugo rješenje za isti upit glasi:

```
SELECT SNO, SNAME
FROM STUDENT
WHERE SNO IN
      (SELECT SNO
FROM REPORT
WHERE CNO = 121);
```

Upit 3: Pronađi brojeve i imena studenata koji su upisali bar jedan kolegij kojeg predaje Quinn.

```
SELECT SNO, SNAME
FROM STUDENT
WHERE SNO IN
      (SELECT SNO
FROM REPORT
WHERE CNO IN
      (SELECT CNO
FROM COURSE
WHERE LNAME = 'Quinn'));
```

Drugo rješenje za isti upit glasi:

```
SELECT STUDENT.SNO, STUDENT.SNAME
FROM STUDENT, REPORT, COURSE
WHERE STUDENT.SNO = REPORT.SNO
AND REPORT.CNO = COURSE.CNO
AND COURSE.LNAME = 'Quinn';
```

Upit 4: Pronađi sve parove brojeva studenata takve da su odgovarajući studenti na istom stupnju.

```
SELECT TEMP1.SNO, TEMP2.SNO
FROM STUDENT TEMP1, STUDENT TEMP2
WHERE TEMP1.SNO < TEMP2.SNO
AND TEMP1.LEVEL = TEMP2.LEVEL;
```

Ovdje smo uveli drugo ime (alias) za relaciju *STUDENT*.

Upit 5: Pronađi sve podatke o studentima koji nisu upisali kolegij 121.

```
SELECT *
FROM STUDENT
WHERE SNO NOT IN
      (SELECT SNO
FROM REPORT
WHERE CNO = 121);
```

Znak * ovdje označava sve attribute relacije.

Upit 6: Pronađi brojeve onih studenata koji su upisali sve kolegije. Budući da SQL nema univerzalni kvantifikator ali ima egzistencijalni, upit moramo ovako preformulirati: "Pronađi brojeve onih studenata za koje ne postoji kolegij kojeg oni nisu upisali".

```

SELECT SNO
FROM STUDENT
WHERE NOT EXISTS
    (SELECT *
     FROM COURSE
     WHERE NOT EXISTS
        (SELECT *
         FROM REPORT
         WHERE REPORT.SNO = STUDENT.SNO
         AND REPORT.CNO = COURSE.CNO));

```

Ovdje je **EXISTS (SELECT ...)** istina ukoliko je rezultat uključene **SELECT** naredbe neprazan.

3.3.2 Ažuriranje relacija

Upis, promjena, odnosno brisanje n-torke u relaciji postiže se naredbom **INSERT**, **UPDATE**, odnosno **DELETE**. Sve tri naredbe građene su po analogiji sa **SELECT**.

Primjer 1: Upiši u relaciju *STUDENT* novu n-torku sa sljedećim vrijednostima atributa: *SNO* = 867520, *SNAME* = 'Smith', *LEVEL* = 2.

```

INSERT
INTO STUDENT
VALUES (867520, 'Smith', 2);

```

Primjer 2: Promijeni nastavnika kolegija 251 iz Quinn u Black.

```

UPDATE COURSE
SET LNAME = 'Black'
WHERE CNO = 251;

```

Ova naredba mijenja **svaku** n-torku u kojoj je *CNO* = 251.

Primjer 3: Briši sve studente na stupnju 3.

```

DELETE
FROM STUDENT
WHERE LEVEL = 3;

```

3.4 Optimizacija upita

Jezici za relacijske baze podataka daju korisniku veliku slobodu u postavljanju upita. Teret efikasnog odgovaranja na te raznolike upite prebačen je na DBMS. Naime, odgovor na jedan te isti upit obično se može dobiti na razne načine, a zadatak DBMS-a je da odabere najefikasniji način. Odabir dobrog postupka za odgovaranje zove se optimizacija upita. Moderni DBMS provodi optimizaciju na dvije razine:

viša (logička) razina : preformuliranje algebarskog izraza u oblik koji je ekvivalentan polaznom, ali je pogodniji sa stanovišta izvrednjavanja;

niža (fizička) razina : odabir dobrog algoritma za izvrednjavanje svake od osnovnih operacija u algebarskom izrazu. Pritom se nastoji iskoristiti eventualno prisustvo pomoćnih struktura podataka (indeksi i slično).

U ovom poglavlju bavimo se višom razinom optimizacije, dok će niža razina biti obrađena u poglavlju 5. Prešutno smo pretpostavili da je upit zadan pomoću relacijske algebre. Slijedi opravdanje za tu pretpostavku.

3.4.1 Odnos između relacijske algebre i računa

Svaki upit zapisan u relacijskoj algebri može se zamijeniti ekvivalentnim upitom zapisanim u relacijskom računu. Strogi dokaz ove tvrdnje može se naći u literaturi. Mi kao ilustraciju navodimo ekvivalente za “bitne” algebarske operacije:

$$\begin{array}{lll} R_1 \textbf{ union } R_2 & \dots & \{t \mid R_1(t) \textbf{ or } R_2(t)\} , \\ R_1 \textbf{ minus } R_2 & \dots & \{t \mid R_1(t) \textbf{ and not } R_2(t)\} , \\ R_1 \textbf{ times } R_2 & \dots & \{< t, r > \mid R_1(t) \textbf{ and } R_2(r)\} , \\ R_1 \textbf{ where } f(X) & \dots & \{t \mid R_1(t) \textbf{ and } f(t.X)\} , \\ R_1[X] & \dots & \{t.X \mid R_1(t)\} . \end{array}$$

Ovdje $< t, r >$ znači kombinaciju n -torki t i r . E.F. Codd je u svom članku iz 1972. godine dokazao i obratnu tvrdnju, tj. da se svaki upit zapisan pomoću relacijskog računa može formulirati i pomoću relacijske algebre. Štoviše, Codd je izložio **redukcijski algoritam** kojim se izraz u računu pretvara u izraz u algebri.

Praktični jezici poput SQL kombiniraju elemente računa i algebre. Stoga je očigledno da se upiti zapisani u tim jezicima također mogu preformulirati u relacijsku algebru.

Zahvaljujući ovim rezultatima, prilikom razmatranja optimizacije možemo bez gubitka općenitosti smatrati da je upit već zapisan u relacijskoj algebri. Naime, ako to nije tako, tada bi prvi korak optimizacije bila pretvorba upita u algebarski izraz.

3.4.2 Osnovna pravila za optimizaciju

Značajna ušteda vremena postiže se mijenjanjem redoslijeda operacija, u cilju da se što prije smanji veličina relacija s kojima radimo.

Kombiniranje selekcija . Očito vrijedi ekvivalencija:

$$(R \textbf{ where } \mathcal{B}_1) \textbf{ where } \mathcal{B}_2 = R \textbf{ where } (\mathcal{B}_1 \textbf{ and } \mathcal{B}_2).$$

Ovime smanjujemo potrebno vrijeme ukoliko se obje selekcije odvijaju podjednako “sporo” (tj. pregledom cijele relacije). Ako se jedna relacija odvija brzo (na primjer zahvaljujući prisustvu pomoćnih fizičkih struktura podataka) a druga sporo, tada se kombiniranje ne isplati, jer će rezultirajuća selekcija također biti spora. Znači, odluka što je bolje ovisi o fizičkoj građi baze podataka.

Izvlačenje selekcije ispred prirodnog spoja odnosno Kartezijevog produkta . Ako uvjet B sadrži samo atribute od R , a ne one od S , tada vrijedi:

$$\begin{array}{l} (R \textbf{ join } S) \textbf{ where } \mathcal{B} = (R \textbf{ where } \mathcal{B}) \textbf{ join } S, \\ (R \textbf{ times } S) \textbf{ where } \mathcal{B} = (R \textbf{ where } \mathcal{B}) \textbf{ times } S. \end{array}$$

Ova transformacija može znatno smanjiti broj n -torki koje ulaze u prirodni spoj odnosno Kartezijev produkt. Općenitije, ako \mathcal{B} rastavimo na $\mathcal{B} = \mathcal{B}_R \textbf{ and } \mathcal{B}_S \textbf{ and } \mathcal{B}_C \textbf{ and } \mathcal{B}'$, gdje \mathcal{B}_R sadrži samo atribute od R , \mathcal{B}_S sadrži samo atribute od S , \mathcal{B}_C sadrži zajedničke atribute od R i S , \mathcal{B}' predstavlja ostatak od \mathcal{B} , tada vrijedi:

$$\begin{array}{l} (R \textbf{ join } S) \textbf{ where } \mathcal{B} = \\ ((R \textbf{ where } (\mathcal{B}_R \textbf{ and } \mathcal{B}_C)) \textbf{ join } (S \textbf{ where } (\mathcal{B}_S \textbf{ and } \mathcal{B}_C))) \textbf{ where } \mathcal{B}'. \end{array}$$

Slična ekvivalencija može se napisati i za operator **times**.

Na primjer, želimo naći brojeve svih studenata na stupnju 1 koji su upisali neki kolegij kod nastavnika Quinna i dobili ocjenu iznad 70 iz tog kolegija. Tada se to može izraziti kao:

$RESULT := ((STUDENT \textbf{join} REPORT \textbf{join} COURSE) \textbf{where} \\ (LEVEL=1 \textbf{and} MARK>70 \textbf{and} LNAME='Quinn')) [SNO].$

Primjenom transformacije dobivamo:

$RESULT := (((STUDENT \textbf{where} (LEVEL=1) \textbf{join} \\ (REPORT \textbf{where} MARK>70)) \textbf{join} \\ (COURSE \textbf{where} LNAME='Quinn')) [SNO] .$

Izvlačenje selekcije ispred projekcije . Ako uvjet \mathcal{B} sadrži samo projicirane attribute X , tada je:

$$R[X] \textbf{ where } \mathcal{B} = (R \textbf{ where } \mathcal{B}) [X] .$$

Projiciranje može dugo trajati zbog eliminacije “duplikata”. Zato je dobro najprije smanjiti broj n-torki selektiranjem. To je posebno preporučljivo onda kad postoje fizička sredstva za brzu selekciju.

Kombiniranje projekcija . Ako su X, Y i Z atributi od relacije R , tada vrijedi:

$$((R[X, Y, Z])[X, Y])[X] = R[X] .$$

Umjesto tri projekcije dovoljna je samo jedna. U ovom jednostavnom slučaju, ta jednakost je očigledna. No u dugačkim i kompliciranim izrazima nije lako uočiti redundantno projiciranje.

Izvlačenje projekcije ispred prirodnog spoja . Moramo paziti da projiciranjem ne izbacimo zajednički atribut iz relacija prije nego što je bio obavljen prirodni spoj. Ako X označava zajedničke attribute od R i S , tada je:

$$(R \textbf{ join } S)[X] = R[X] \textbf{ join } S[X] .$$

No pravilo ne vrijedi za proizvoljni skup atributa X . Neka su A_R atributi od R , A_S atributi od S , $A_C = A_R \cap A_S$ zajednički atributi od R i S . Tada vrijedi:

$$(R \textbf{ join } S)[X] = (R[(X \cap A_R) \cup A_C] \textbf{ join } S[(X \cap A_S) \cup A_C])[X] .$$

Opet se smanjuje broj n-torki koje ulaze u prirodni spoj. Zahvat ne mora uvijek biti koristan, jer projekcija može spriječiti efikasnu implementaciju prirodnog spoja. Na primjer, projekcija može stvoriti privremenu relaciju na koju nisu primjenjive postojeće pomoćne fizičke strukture. Znači, odluka što je bolje opet ovisi o fizičkoj građi.

Optimizacija skupovnih operatora . Koji put su od koristi sljedeća pravila:

$$\begin{aligned} (R \textbf{ union } S) \textbf{ where } \mathcal{B} &= (R \textbf{ where } \mathcal{B}) \textbf{ union } (S \textbf{ where } \mathcal{B}) , \\ (R \textbf{ minus } S) \textbf{ where } \mathcal{B} &= (R \textbf{ where } \mathcal{B}) \textbf{ minus } (S \textbf{ where } \mathcal{B}) , \\ (R \textbf{ union } S)[X] &= R[X] \textbf{ union } S[X] , \\ (R \textbf{ minus } S)[X] &= R[X] \textbf{ minus } S[X] , \\ (R \textbf{ where } \mathcal{B}_1)[X] \textbf{ union } (R \textbf{ where } \mathcal{B}_2)[X] &= (R \textbf{ where } (\mathcal{B}_1 \textbf{ or } \mathcal{B}_2))[X] . \end{aligned}$$

Predzadnja jednakost vrijedi pod pretpostavkom da X sadrži ključne attribute od R (a time i od S). Transformacijama se nastoji smanjiti broj n-torki koje sudjeluju u operacijama **union** i **minus**. Operator **intersect** je specijalni slučaj od **join**, pa za njega vrijede ista pravila kao za **join**.

4

FIZIČKA GRAĐA BAZE PODATAKA

4.1 Elementi fizičke građe

Baza podataka se u fizičkom smislu svodi na gomilu bitova pohranjenih na magnetskim diskovima. Takav doslovni način gledanja je bez sumnje istinit, no on ne omogućuje da stvarno razumijemo fizičku građu baze. Umjesto toga, bolje je promatrati malo apstraktnije objekte, kao što su zapisi, datoteke i pointeri. Riječ je o prvom, sasvim niskom nivou apstrakcije koji je vrlo blizak fizičkoj stvarnosti.

4.1.1 Vanjska memorija računala

Operacijski sustav računala dijeli vanjsku memoriju u jednako velike **blokove**. Veličina bloka je konstanta operacijskog sustava, i ona može iznositi na primjer 512 byte ili 4096 byte. Svaki blok jednoznačno je zadan svojom **adresom**. Osnovna operacija s vanjskom memorijom je prijenos bloka sa zadanom adresom iz vanjske memorije u glavnu, ili obratno. Dio glavne memorije koji sudjeluje u prijenosu (i ima jednaku veličinu kao i sam blok) zove se **buffer**. Blok je najmanja količina podataka koja se može prenijeti; na primjer ako želimo pročitati samo jedan byte iz vanjske memorije, tada moramo prenijeti cijeli odgovarajući blok, pretražiti buffer u glavnoj memoriji i izdvojiti traženi byte. Vrijeme potrebno za prijenos bloka nije konstantno već ovisi o trenutnom položaju glave diska. Ipak, to vrijeme (mjereno u milisekundama) neusporedivo je veće od vremena potrebnog za bilo koju manipulaciju u glavnoj memoriji (mjereno u mikro- i nanosekundama). Zato je brzina nekog algoritma za rad s vanjskom memorijom određena brojem blokova koje algoritam mora prenijeti, a vrijeme potrebno za računanje i manipuliranje u glavnoj memoriji je zanemarivo.

4.1.2 Datoteke

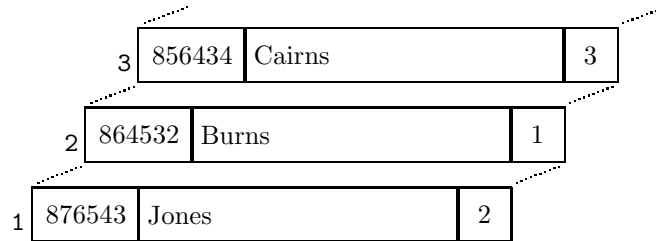
Osnovni problem fizičkog prikazivanja baze podataka je organizacija datoteke. **Datoteka** je konačni niz zapisa istog tipa pohranjenih u vanjskoj memoriji. **Tip zapisa** zadaje se kao uređena n-torka **osnovnih podataka** (komponenti), gdje je svaki osnovni podatak opisan svojim imenom i tipom (int, float, character string, ...). Primijetimo da je tip zapisa definiran nešto uže nego na primjer u C-u, naime nije dozvoljena hijerarhijska ni varijabilna građa zapisa. Sam **zapis** sastoji se od konkretnih vrijednosti osnovnih podataka. Smatramo da su zapisi **fiksne duljine**, dakle jedan zapis ima točno jednu vrijednost svakog od osnovnih podataka i ta vrijednost je prikazana fiksnim brojem byte-ova.

Tipične operacije koje se obavljaju nad datotekom su:

- ubaciti novi zapis,
- promijeniti zapis,
- izbaciti zapis,
- naći zapis ili zapise gdje zadani podaci imaju zadane vrijednosti.

Složenost građe datoteke ovisi o tome koliko efikasno želimo obavljati pojedine od ovih operacija.

(Kandidat za) **ključ** je osnovni podatak, ili kombinacija osnovnih podataka, čija vrijednost jednoznačno određuje zapis. Ukoliko ima više kandidata za ključ, tada odabiremo jednog od njih da bude **primarni** ključ. Ne mora svaka datoteka imati ključ, jer mogu postojati zapisi-duplikati.



Slika 4.1: Datoteka sa zapisima o studentima

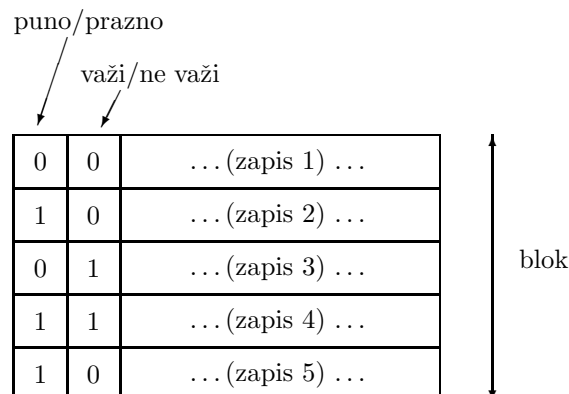
Kao primjer, promatrajmo datoteku o studentima fakulteta. Tip zapisa definiran u jeziku C je:

```
typedef struct {
    int SNO;
    char SNAME[15];
    char LEVEL;
} STUDENT;
```

Prvih nekoliko zapisa izgleda kao na Slici 4.1. Svaki zapis ima svoj redni broj. Duljina zapisa je 20 byte. Vrijednost prvog osnovnog podatka zauzima 1. do 4. byte, vrijednost drugog osnovnog podatka 5. do 19. byte, a treći podatak je u 20. byte-u.

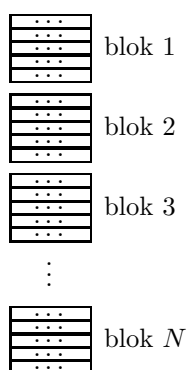
4.1.3 Smještaj datoteke u vanjskoj memoriji

Zapis je obično znatno manji od bloka. Stoga se više zapisa sprema u jedan blok. Uzimamo da je u bloku smješten cijeli broj zapisa, što znači da je dio bloka možda neiskorišten. **Adresa** zapisa gradi se kao uređeni par (adresa bloka, pomak unutar bloka). Kod nekih organizacija datoteki mogu neka mjesta za zapise u bloku ostati prazna. Kako da razlikujemo puna i prazna mjesta? Jedno rješenje je: proširiti zapis s jednim bitom koji označava da li je mjesto “puno” ili “prazno”. Koji put je potrebno “poništit” zapis (učiniti ga nevažećim), ali tako da njegovo mjesto i dalje bude zauzeto. Kako da razlikujemo važeće i nevažeće zapise? Opet proširimo zapis jednim bitom koji označava da li zapis “važi” ili “ne važi”. Sve se ovo vidi na Slici 4.2.



Slika 4.2: Prikaz zapisa unutar bloka

Cijela datoteka obično zauzima više blokova, kao što je ilustrirano Slikom 4.3. Način kako se određuju adrese tih blokova ovisi o organizaciji datoteke (vidi potpoglavlje 4.2). U svakom slučaju, ne mora se raditi o nizu uzastopnih adresa. Naime, zbog pisanja i brisanja podataka, vanjska memorija se prije ili kasnije isparcelira, pa smo prisiljeni koristiti njene nepovezane dijelove.



Slika 4.3: Prikaz datoteke kao niza blokova

4.1.4 Pointeri

Pointer je vrijednost unutar jednog zapisa koja pokazuje na neki drugi zapis (u istoj ili drugoj datoteci). Pointer može biti:

- adresa zapisa (“fizički” pointer),
- vrijednost primarnog ključa (“logički” pointer).

Postoje i prijelazni oblici između fizičkog i logičkog pointera, na primjer redni broj zapisa; no smisao takvih vrijednosti ovisi o konkretnoj organizaciji datoteke i mi ih nećemo razmatrati.

Pointeri omogućuju uspostavljanje veze između zapisa, dakle oni omogućuju da iz jednog zapisa pristupimo drugom. Pointer-adresa omogućuje brzi pristup. Pointer-ključ je “spor” - on samo implicitno određuje zapis kojeg tek treba pronaći nekim mehanizmom za traženje na osnovu ključa.

Prisustvo pointera-adrese može uzrokovati probleme kod ažuriranja ili reorganiziranja datoteke. Ako na zapis pokazuje pointer-adresa, kažemo da je zapis **prikovan** (pinned) - zapis se naime ne smije fizički pomicati sa svog mjesta jer bi nakon pomaka pointer krivo pokazivao. Jedini način da pomaknemo prikovani zapis je da također ažuriramo i pointer, no problem je da za zadani zapis obično ne znamo gdje se sve mogu nalaziti pointeri koji na njega pokazuju.

Prisustvo pointera-ključa ne uzrokuje nikakve probleme kod ažuriranja ili reorganizacije datoteke. To je i razlog zašto koristimo takve pointere, unatoč njihovoj “sporosti”.

4.1.5 Fizička građa cijele baze

Cijela baza je građena kao skup datoteki. Zapisi u datotekama mogu biti međusobno povezani pointerima. Sve operacije s bazom svode se na osnovne operacije s datotekama. Zbog toga se u potpoglavljima 4.2 i 4.3 bavimo isključivo datotekama.

Ukoliko imamo posla s relacijskim bazama, tada se svaka relacija prikazuje jednom datotekom. Atributi relacije odgovaraju osnovnim podacima u zapisu. Jedna n-torka relacije prikazana je jednim zapisom. Primarni ključ relacije određuje primarni ključ datoteke. Primijetimo da se ovakvim fizičkim prikazom uvodi umjetni redoslijed među atributima relacije, te umjetni redoslijed n-torki. No, relacijski DBMS u stanju je promijeniti taj redoslijed prilikom rada s relacijom.

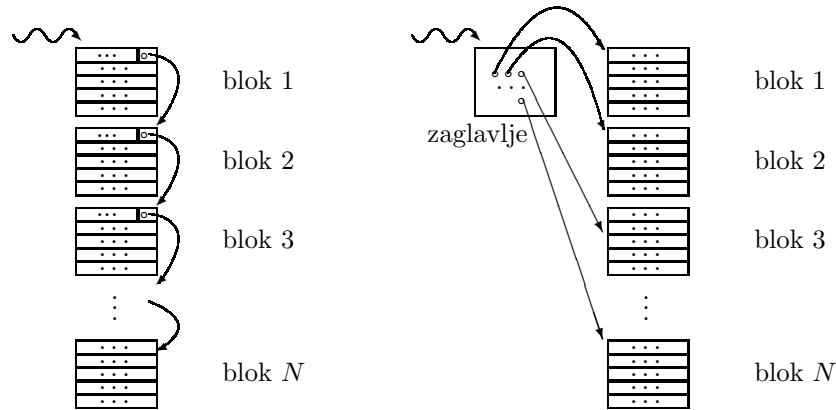
Osim osnovnih datoteki koje odgovaraju pojedinim relacijama, fizička građa relacijske baze može sadržavati i dodatne pomoćne datoteke koje ubrzavaju pretraživanje i uspostavljanje veza među podacima. Tipični primjeri takvih datoteki su indeksi.

4.2 Pristup na osnovu primarnog ključa

Važna operacija u radu s datotekama je pristup na osnovu primarnog ključa. Dakle, za zadanu vrijednost primarnog ključa treba odrediti adresu (najviše jednog) zapisa koji sadrži tu vrijednost ključa. Proučit ćemo razne organizacije datoteki i načine kako da se za njih realizira pristup na osnovu primarnog ključa.

4.2.1 Jednostavna datoteka

Zapravo se radi o odsustvu bilo kakve organizacije. Zapise datoteke poredamo u onoliko blokova koliko je potrebno. Blokovi koji čine datoteku mogu biti povezani u vezanu listu (svaki blok sadrži adresu idućeg bloka) ili može postojati tablica adresa svih blokova (koja zauzima prvi ili prvih nekoliko blokova). Te dvije varijante jednostavne datoteke vide se na Slici 4.4.



Slika 4.4: Jednostavna datoteka (dvije varijante)

Traženje zapisa sa zadanom vrijednošću ključa zahtijeva sekvencijalno čitanje cijele datoteke (ili bar pola datoteke u prosjeku), sve dok ne naiđemo na traženi zapis. Ako je datoteka velika, morat ćemo učitati mnogo blokova, pa pristup traje dugo.

Da bi ubacili novi zapis, stavljamo ga na prvo slobodno mjesto u prvom nepopunjenom bloku, ili priključujemo novi blok ukoliko su svi postojeći blokovi popunjeni.

Ako su zapisi prikovani, tada ne smijemo zaista izbacivati zapise, već ih samo možemo proglasiti nevažećima. No ako zapisi nisu prikovani, tada ih smijemo izbacivati, pa će se time ispražnjena mjesta koristiti kod budućih ubacivanja. Promjena zapisa obavlja se bez ograničenja.

4.2.2 Hash datoteka

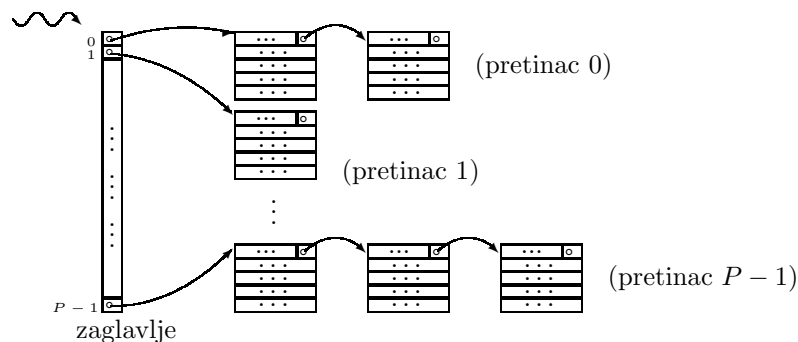
Zapise datoteke smještamo u p **pretinaca**, označenih rednim brojevima $0, 1, 2, \dots, p - 1$. Svaki pretinac sastoji se od jednog ili više blokova. Zadana je tzv. **hash funkcija** h , koja daje redni broj $h(k)$ pretinca u kojem se treba nalaziti zapis s vrijednošću ključa k .

Skup mogućih vrijednosti ključa obično je znatno veći od broja pretinaca. Važno je da h uniformno distribuiraju vrijednosti ključa na pretince. Tada se neće dešavati da se pretinci neravnomjerno pune. Dajemo primjer dobre hash funkcije:

- vrijednosti ključa promatraju se kao nizovi bitova fiksne duljine;
- zadani niz bitova se podijeli na skupine fiksne duljine, s time da se zadnja skupina nadopuni nulama ako je potrebno;
- skupine bitova se zbroje kao cijeli brojevi;
- zbroj se podijeli s brojem pretinaca;
- ostatak kod dijeljenja je traženi redni broj pretinca.

Hash datoteka obično se realizira tako da pretinci budu vezane liste blokova. Adrese početaka tih listi čine zaglavlje koje se smješta u prvi blok ili prvih nekoliko blokova datoteke. Cijela organizacija hash datoteke vidljiva je na Slici 4.5. Zaglavlje je obično dovoljno malo, pa se za vrijeme rada s datotekom može držati u glavnoj memoriji.

Zapis sa zadanom vrijednošću ključa k tražimo tako da izračunamo $h(k)$, te sekvencijalno pretražimo $h(k)$ -ti pretinac. Ako je hash funkcija zaista uniformna, te ako je broj pretinaca dobro odabran, tada ni jedan od pretinaca nije suviše velik. Pristup na osnovu ključa zahtijeva svega nekoliko čitanja bloka (obično oko 2 čitanja).



Slika 4.5: Hash datoteka

Novi zapis s vrijednošću ključa k ubacuje se u $h(k)$ -ti pretinac, u prvi blok gdje ima mjesta. Ako su svi blokovi $h(k)$ -te vezane liste puni, tada na kraj te vezane liste priključujemo novi blok.

Ako su svi zapisi prikovani, tada ih ne smijemo izbacivati, već ih samo možemo proglasiti “nevažecima”. Ako zapisi nisu prikovani, tada ih smijemo izbacivati, čime oslobađamo mjesta za buduća ubacivanja.

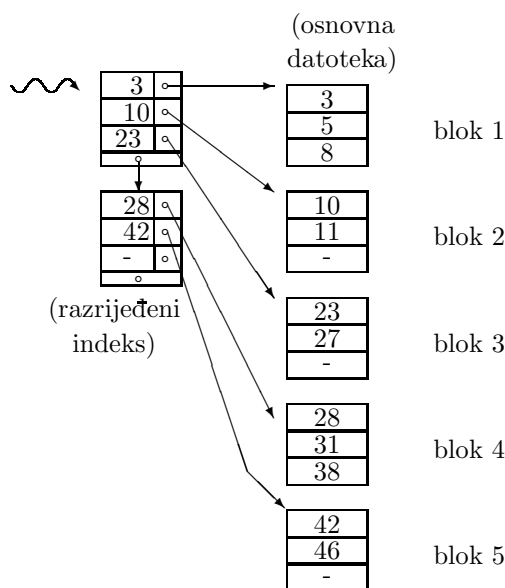
Kod promjene zapisa ne smije se mijenjati vrijednost ključa, jer bi se time mogao promijeniti broj pretinca kojem zapis mora pripadati (takvu promjenu možemo ostvariti izbacivanjem stare verzije zapisa te ubacivanjem nove verzije).

Hash datoteka zahtijeva povremene reorganizacije (povećanje broja pretinaca). Također, ona nije pogodna onda kad obrađujemo zapise u sortiranom redoslijedu po ključu.

4.2.3 Datoteka s indeksom

Indeks je mala pomoćna datoteka koja olakšava traženje u velikoj (osnovnoj) datoteci. Izložiti ćemo dvije varijante datoteke s indeksom.

Indeks sekvencijalna organizacija zahtijeva da zapisi u osnovnoj datoteci budu sortirani po vrijednostima ključa (na primjer uzlazno). Blokovi ne moraju biti sasvim popunjeni. Dodajemo tzv. **razrijeđeni indeks**. Svaki zapis u indeksu odgovara jednom bloku osnovne datoteke i oblika je (k, a) , gdje je k najmanja vrijednost ključa u dotičnom bloku, a je adresa bloka. Sam indeks je također sortiran po ključu i za sada zamišljamo da je jednostavno organiziran. Cijela organizacija vidljiva je na Slici 4.6.



Slika 4.6: Indeks-sekvencijalna organizacija datoteke

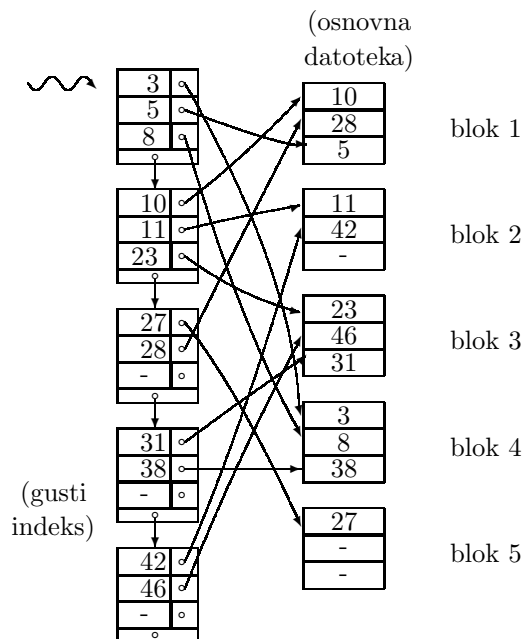
Da bi u osnovnoj datoteci našli zapis sa zadanom vrijednošću ključa k_0 , čitamo indeks te tražimo najveći k_1 takav da je $k_1 \leq k_0$ i pritom par (k_1, a_1) postoji u indeksu. Zatim učitamo i pretražimo blok s adresom a_1 . Pristup po primarnom ključu zahtijeva (u najgorem slučaju) onoliko čitanja bloka koliko ima blokova u indeksu $+1$.

Pretpostavimo da zapisi iz osnovne datoteke nisu prikovani, tj. da na njih ne pokazuju drugi pointeri-adrese osim onih iz indeksa. Tada možemo obavljati ubacivanja i izbacivanja zapisa.

Opisujemo ubacivanje novog zapisa u osnovnu datoteku. Najprije uz pomoć indeksa odredimo blok koji bi morao sadržavati taj novi zapis. Pokušamo umetnuti zapis u blok, i to na pravo mjesto u smislu sortiranog redoslijeda. Ukoliko u tome ne uspijemo (blok bi se prepunio), tada pokušamo zadnji zapis iz tog prepunjenog bloka prebaciti u idući blok. Ako ni u tome ne uspijemo (ne postoji idući blok ili je i on pun) tada u osnovnu datoteku iza prepunjenog bloka uključimo novi blok, te u njega prebacimo zadnji zapis iz prepunjenog bloka. Ubacivanje zapisa u osnovnu datoteku ponekad zahtijeva promjene u indeksu. Na primjer, ako se u starom bloku promijenila najmanja vrijednost ključa, tada ažuriramo odgovarajući par (k, a) u indeksu. Također, uključivanje novog bloka zahtijeva ubacivanje novog para (k, a) u indeks; to se obavlja po algoritmu koji je sličan upravo opisanom.

Ako se izbacivanjem zapisa neki od blokova osnovne datoteke isprazni, isključujemo ga iz datoteke. Izbacivanje zapisa u osnovnoj datoteci također može zahtijevati promjene u indeksu. Kod promjene zapisa u osnovnoj datoteci ne smije se mijenjati vrijednost ključa, jer bi se time promijenio položaj zapisa u sortiranom redoslijedu. Indeks-sekvencijalna organizacija omogućuje (sekvencijalno) čitanje osnovne datoteke u sortiranom redoslijedu po ključu.

Indeks-direktna organizacija dopušta da zapisi u osnovnoj datoteci budu upisani u proizvoljnom redoslijedu. Dodajemo tzv. **gusti indeks**. Svaki zapis u indeksu odgovara jednom zapisu osnovne datoteke i oblika je (k, a) , gdje je k vrijednost ključa u dotičnom zapisu osnovne datoteke, a je pointer-adresa zapisa iz osnovne datoteke. Za razliku od nesortirane osnovne datoteke, indeks je sortiran po ključu. Zamišljamo da je indeks jednostavno organiziran. Cijela organizacija prikazana je na Slici 4.7.



Slika 4.7: Indeks-direktna organizacija datoteke

Da bi u osnovnoj datoteci našli zapis sa zadanom vrijednošću ključa k_0 , čitamo indeks te tražimo par (k_0, a_0) . Direktno čitamo blok u kojem je zapis s adresom a_0 . Zapise iz osnovne datoteke možemo ubacivati i izbacivati pod pretpostavkom da nisu prikovani. Postupak je sličan kao kod jednostavne organizacije. Svaki put se mora ažurirati indeks, na način sličan onom za indeks-sekvencijalnu organizaciju. Ako kod promjene zapisa iz osnovne datoteke promijenimo i vrijednost ključa, potrebno je jedno ubacivanje i jedno izbacivanje u indeksu.

Indeks-direktna organizacija omogućuje čitanje cijele osnovne datoteke u sortiranom redoslijedu po ključu (preko indeksa). No očekujemo da to traje znatno dulje nego kod indeks-sekvencijalne organizacije, jer se dešava da zbog raznih zapisa više puta učitavamo isti blok.

Prednost indeks-direktne organizacije u odnosu na indeks-sekvencijalnu je jednostavnije ubacivanje i izbacivanje. Također, manje je rasipanje memorije u osnovnoj datoteci. Daljnja prednost je mogućnost odgovora na pitanje da li postoji zadana vrijednost ključa, bez da uopće čitamo osnovnu datoteku. Mana indeks-direktne organizacije je: znatno veći indeks.

4.2.4 B-stablo

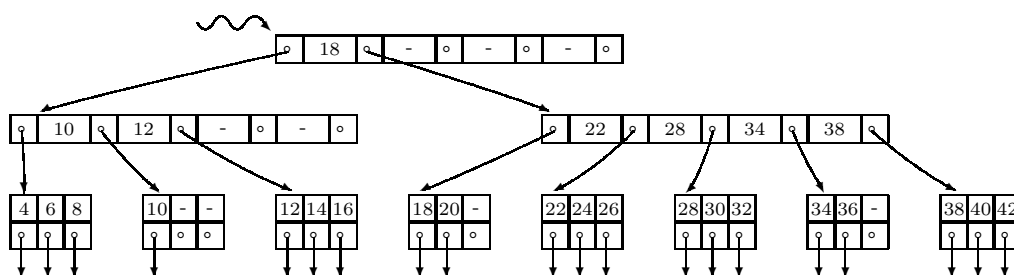
U prošlom odjeljku zamišljali smo da je indeks jednostavno organiziran. To je prihvatljivo rješenje samo onda kad je indeks jako mali, tako da stane u svega nekoliko blokova. No kod velikih osnovnih datoteki, indeks (pogotovo gusti) je također velik, pa bi njegovo sekvencijalno pretraživanje trajalo predugo. Potrebna je bolja organizacija indeksa. Današnji DBMS-i u tu svrhu redovito koriste tzv. B-stabla.

B-stablo reda m je m -narno stablo sa sljedećim svojstvima:

- korijen je ili list ili ima bar dvoje djece;
- svaki čvor, izuzev korijena i listova, ima između $\lceil m/2 \rceil$ i m djece;
- svi putovi od korijena do lista imaju istu duljinu.

U nastavku promatramo prikaz gustog indeksa pomoću B-stabla (prikaz razrijeđenog indeksa ide analogno). Indeks je tada jedno B-stablo sagrađeno od blokova vanjske memorije, i to tako da jedan čvor bude jedan blok. Veza roditelj-dijete realizira se tako da adresa bloka-djeteta piše u bloku-roditelju. Također vrijedi:

- Unutrašnji čvor ima sadržaj $(a_0, k_1, a_1, k_2, a_2, \dots, k_r, a_r)$, gdje je a_i adresa i -tog djeteta dotičnog čvora ($0 \leq i \leq r$), k_i je vrijednost ključa ($1 \leq i \leq r$). Vrijednosti ključa unutar čvora su sortirane, dakle $k_1 \leq k_2 \leq \dots \leq k_r$. Sve vrijednosti ključa u pod-stablu koje pokazuje a_0 su manje od k_1 . Za $1 \leq i < r$, sve vrijednosti ključa u pod-stablu kojeg pokazuje a_i su u poluotvorenom intervalu $[k_i, k_{i+1})$. Sve vrijednosti ključa u pod-stablu kojeg pokazuje a_r su veće ili jednake k_r .
- List sadrži parove oblika (k, a) , gdje je k vrijednost ključa, a je pointer-adresa pripadnog zapisa u osnovnoj datoteci. Parovi unutar lista su uzlazno sortirani po k . List ne mora biti sasvim popunjen. Jednom zapisu osnovne datoteke odgovara točno jedan par (k, a) u listovima B-stabla.



Slika 4.8: Gusti indeks prikazan kao B-stablo reda 5

Na slici 4.8 vidimo gusti indeks neke osnovne datoteke, prikazan kao B-stablo reda 5. Primijetimo da se indeks prikazan pomoću B-stabla može shvatiti kao hijerarhija manjih jednostavnih indeksa.

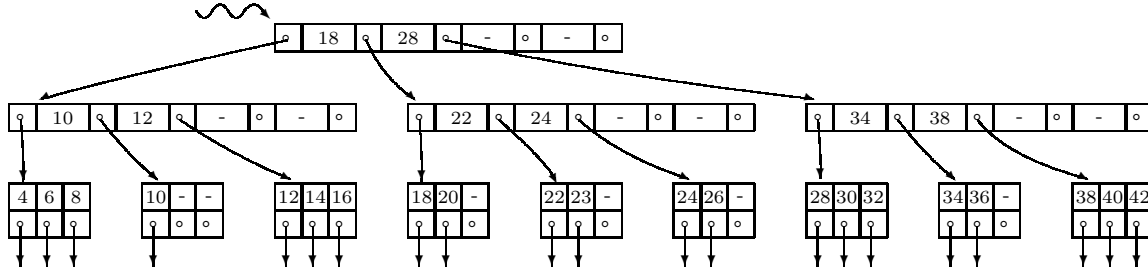
Traženje. Želimo u B-stablu pronaći par (k, a) , gdje je k zadana vrijednost ključa, a je pointer-adresa pripadnog zapisa u osnovnoj datoteci. U tu svrhu slijedimo put od korijena do lista koji bi morao sadržavati k . To se radi tako da redom čitamo unutrašnje čvorove oblika $(a_0, k_1, a_1, k_2, a_2, \dots, k_r, a_r)$, te usporedimo k s k_1, k_2, \dots, k_r . Ako je $k_i \leq k < k_{i+1}$, dalje čitamo čvor kojeg pokazuje a_i . Ako je $k < k_1$, dalje čitamo čvor s adresom a_0 . Ako je $k \geq k_r$, koristimo adresu a_r . Kad nas taj postupak konačno dovede u list, tražimo u njemu zadani k .

Ubacivanje. Da bi ubacili novi par (k, a) u B-stablo, prvo pronađemo list L kojem bi k morao pripadati. Pokušamo umetnuti par (k, a) u L , i to na pravo mjesto u sortiranom redoslijedu. Ukoliko u tome uspijemo, tada smo gotovi.

Ukoliko umetanje ne uspije (L bi se prepunio), uzimamo novi blok L' , te premjestimo u L' zadnju polovicu sortiranog sadržaja prepunjenog L . Neka je P roditelj od L , k' najmanja vrijednost ključa u

L' , a' adresa od L' . Potrebno je L' uključiti u B-stablo kao list. U tu svrhu u P umetnemo k' i a' . Postupak umetanja u P je analogan već opisanom postupku umetanja u L .

Ako P već ima m adresa, umetanje k' i a' uzrokovat će da se P rascijepi na dva bloka, pa ćemo morati umetnuti nove vrijednosti u roditelja od P . Umetanje se rekurzivno ponavlja i može doći najdalje do korijena stabla. Ako se i korijen rascijepi, tada stvaramo novi korijen čija dva djeteta su dvije polovice starog korijena; visina B-stabla se time povećala za 1. Na Slici 4.9 vidi se B-stablo dobiveno iz stabla sa Slike 4.8 umetanjem nove vrijednosti ključa 23.



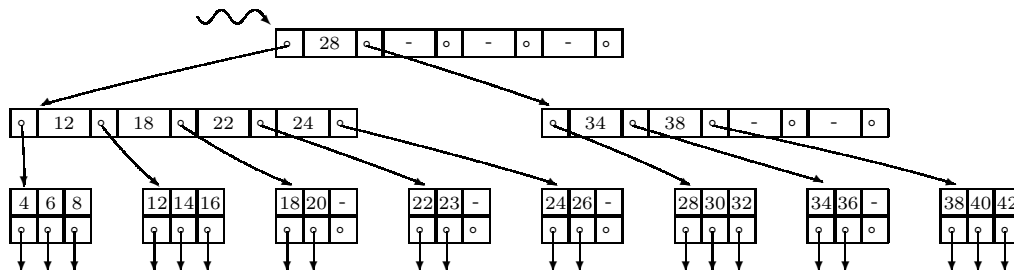
Slika 4.9: Ubacivanje u B-stablo

Izbacivanje. Da bi izbacili par (k, a) iz B-stabla, prvo pronađemo list L koji ga sadrži. Izbacimo (k, a) iz L . Ako je to bio prvi zapis u L , tada idemo u čvor P koji je roditelj od L , te korigiramo vrijednost ključa u P koja se odnosi na L . Pritom, ako je L prvo dijete od P , najmanja vrijednost ključa u L se ne pojavljuje u P već u nekom pretku od P , pa tada moramo prosljediti korekciju duž puta od L do korijena.

Ako je L nakon izbacivanja postao prazan, isključujemo ga iz stabla. Također, moramo korigirati sadržaj od P u skladu s nestankom L . Ako je broj djece od P sad pao ispod $m/2$, gledamo čvor P' koji je neposredno lijevi (ili neposredno desni) brat od P . Ako P' ima bar $\lceil m/2 \rceil + 1$ djece, jednako raspodijelimo vrijednosti ključa i adrese u P i P' tako da oba čvora imaju bar $\lceil m/2 \rceil$ djece. Dalje korigiramo vrijednosti ključa za P i P' u roditelju od P , s time da po potrebi prosljedimo korekciju na više pretke od P .

Ako P' ima točno $\lceil m/2 \rceil$ djece, ujedinitimo P i P' u jedan čvor s $2\lceil m/2 \rceil - 1$ djece (to je $\leq m$). Tada također moramo izbaciti vrijednost ključa i adresu za P' iz roditelja od P . To izbacivanje se obavlja rekurzivnom primjenom upravo opisane procedure.

Ako se posljedice izbacivanja prošire cijelim putem natrag do korijena, može se desiti da trebamo ujediniti jedina dva djeteta korijena. U tom slučaju rezultirajući kombinirani čvor postaje novi korijen, a stari korijen se isključuje iz stabla. Visina B-stabla se time smanjila za 1. Na Slici 4.10 vidi se B-stablo dobiveno iz stabla sa Slike 4.9 nakon izbacivanja vrijednosti ključa 10.



Slika 4.10: Izbacivanje iz B-stabla

Neka B-stablo sadrži u svojim listovima n parova oblika (k, a) . Neka je u jednom listu smješteno u prosjeku b parova (k, a) . Broj čitanja bloka potrebnih za pristup na osnovu ključa proporcionalan je visini stabla, a to je u najgorem slučaju $\approx \log_{m/2} \lceil n/b \rceil$. U praksi m i b mogu biti veliki, pa B-stablo obično ima svega nekoliko (3-4) nivoa. Znači, pristup preko indeksa B-stabla je skoro jednako brz kao pristup u hash datoteku, mada najčešće ipak malo sporiji. Prednost B-stabla pred hash datotekom je čuvanje sortiranog redoslijeda zapisa po ključu.

4.3 Pristup na osnovu drugih podataka

Osim pristupa na osnovu primarnog ključa, u radu s datotekama javlja se i potreba za pristupom na osnovu drugih podataka. Dakle, traže se zapisi u kojima zadani podatak (koji nije ključ) ima zadanu vrijednost. Takvih zapisa može biti i više. Općenitije, mogu se tražiti zapisi u kojima istovremeno podatak A_1 ima zadanu vrijednost v_1 , podatak A_2 ima vrijednost v_2 , ..., podatak A_r ima vrijednost v_r . Problem se uvijek može riješiti sekvencijalnim pregledom cijele datoteke. No ukoliko želimo da se pristup obavlja brzo, potrebne su posebne organizacije datoteke.

4.3.1 Invertirana datoteka

Svi indeksi koje smo razmatrali u Potpoglavlju 4.2 mogli bi se zvati **primarni indeksi**, jer olakšavaju pristup na osnovu primarnog ključa. U ovom poglavlju uvodimo **sekundarni indeks** - to je mala (pomoćna) datoteka koja olakšava pristup zapisima velike (osnovne) datoteke na osnovu podatka koji nije ključ. Sekundarni indeks za podatak A sastoji se od zapisa oblika $(v, \{p_1, p_2, p_3, \dots\})$, gdje je v vrijednost za A , a $\{p_1, p_2, p_3, \dots\}$ je skup pointera na zapise u osnovnoj datoteci u kojima A ima vrijednost v . Ukoliko postoji ovakav indeks, kaže se da je osnovna datoteka **invertirana** po podatku A . Datoteka koja je invertirana po svakom od svojih podataka zove se **potpuno invertirana**.

Kao primjer, promatramo Tabelarni prikaz 4.1. Riječ je o datoteci nastavnika na fakultetu. Ako invertiramo tu datoteku po svakom podatku osim *IME*, dobivamo gusti primarni indeks za ključ *MAT_BR* i tri sekundarna indeksa za *ODJEL*, *DOB* i *STUPANJ*. Indeks za *DOB* sadrži intervale umjesto pojedinačnih vrijednosti.

Osnovna datoteka o nastavnicima					
	<i>MAT_BR</i>	<i>IME</i>	<i>ODJEL</i>	<i>DOB</i>	<i>STUPANJ</i>
a_1	12453	Matić, R.	Matematika	35	Dr.sc.
a_2	16752	Pavić, D.	Matematika	26	Dipl.inž.
a_3	27453	Horvat, I.	Računarstvo	37	Dr.sc.
a_4	34276	Dimić, J.	Fizika	55	Dr.sc.
a_5	37564	Katić, K.	Računarstvo	45	Dipl.inž.
a_6	43257	Radić, M.	Matematika	32	Mr.sc.
a_7	45643	Janić, Z.	Fizika	24	Dipl.inž.
a_8	56321	Popović, G.	Računarstvo	34	Dr.sc.
a_9	57432	Simić, J.	Matematika	52	Dr.sc.

Gusti primarni indeks		Sekundarni indeks za <i>DOB</i>	
<i>MAT_BR</i>	Pointer na zapis	<i>DOB</i>	Pointer na zapis
12453	a_1	21-30	a_2, a_7
16752	a_2	31-40	a_2, a_3, a_6, a_8
27453	a_3	41-50	a_5
34276	a_4	51-65	a_4, a_9
37564	a_5		
43257	a_6		
45643	a_7		
56321	a_8		
57432	a_9		

Sekundarni indeks za <i>ODJEL</i>		Sekundarni indeks za <i>STUPANJ</i>	
<i>ODJEL</i>	Pointer na zapis	<i>STUPANJ</i>	Pointer na zapis
Fizika	a_4, a_7	Dipl.inž.	a_2, a_5, a_7
Matematika	a_1, a_2, a_6, a_9	Mr.sc.	a_6
Računarstvo	a_3, a_5, a_8	Dr.sc.	a_1, a_3, a_4, a_8, a_9

Tabelarni prikaz 4.1: Invertirana organizacija datoteke.

Invertirani podaci mogli bi se u principu ispustiti iz osnovne datoteke. No to se obično ne radi, zato da se ne bi trošilo vrijeme na ponovno sastavljanje zapisa.

Pristup na osnovu bilo kojeg podatka ostvaruje se tako da u odgovarajućem indeksu pronađemo zadanu vrijednost podatka, pročitamo popis pointera, te za svaki od pointera pročitamo zapis u osnovnoj datoteci. Invertirana organizacija omogućuje i brz odgovor na pitanja o postojanju ili broju zapisa sa zadanim svojstvima. Na primjer, promatramo upit: “Koliko nastavnika u odjelu Računarstvo ima doktorat?” Odgovor nalazimo tako da u sekundarnim indeksima nađemo skup pointera na nastavnike iz Računarstva, odnosno skup pointera na doktore. U presjeku tih dvaju skupova nalaze se samo dva pointera. Dakle odgovor smo dobili bez čitanja osnovne datoteke.

Zapisi osnovne datoteke u prošlom primjeru bili su prikovani jer su sekundarni indeksi sadržavali pointer-e-adrese. Prikovanost se može izbjeći ako pointer-e-adrese u sekundarnim indeksima zamijenimo pointerima-vrijednostima ključa. To će olakšati ažuriranje i reorganizaciju osnovne datoteke, no usporit će pristup na osnovu ne-ključnih podataka - naime stalno ćemo morati pretraživati primarni indeks da bi vrijednosti ključa prevodili u stvarne adrese. Na primjer, sekundarni indeks s pointerima-vrijednostima ključa za podatak *ODJEL* izgledao bi kao u Tabelarnom prikazu 4.2

Sekundarni indeks za *ODJEL*

<i>ODJEL</i>	Pointer na zapis
Fizika	34276, 45643
Matematika	12453, 16725, 43257, 57432
Računarstvo	27423, 37564, 56321

Tabelarni prikaz 4.2: Sekundarni indeks s pointerima-vrijednostima ključa.

Mana invertirane organizacije je utrošak memorije za indekse. No još veća mana je višestruko povećanje posla prilikom svakog ažuriranja osnovne datoteke. Naime, kod svakog ubacivanja, izbacivanja ili promjene zapisa u osnovnoj datoteci, mora se napraviti i korekcija u svakom od indeksa.

Zapisi u sekundarnom indeksu oblika $(v, \{p_1, p_2, p_3, \dots\})$ su varijabilne duljine. Oni se fizički prikazuju kao više zapisa fiksne duljine: (v, p_1) , (v, p_2) , (v, p_3) , Cijeli sekundarni indeks može biti prilično glomazan, te se redovito fizički prikazuje kao B-stablo. Prikaz je sličan onome za primarni gusti indeks (vidi Potpoglavlje 4.2), s time da listovi stabla mogu sadržavati više parova oblika (v, p) s istom vrijednošću v . To zahtijeva da se definicija unutarnjih čvorova B-stabla malo poopći.

4.3.2 Višestruke vezane liste

Pristup na osnovu zadanog ne-ključnog podatka A možemo ubrzati tako da sve zapise osnovne datoteke s istim vrijednostima za A povežemo u vezanu listu. Veza se ostvaruje pomoću pointera kojeg smo uklopili u zapis kao dodatni podatak. Potreban nam je još i takozvani **indeks listi** za podatak A - to je mala (pomoćna) datoteka sastavljena od zapisa oblika (v, p) , gdje je v vrijednost za A , a p je pointer na početak odgovarajuće vezane liste zapisa u osnovnoj datoteci.

Ukoliko sve ovo učinimo za više ne-ključnih podataka, imat ćemo više indeksa listi, a jedan zapis osnovne datoteke imat će više uklopljenih pointera te će istovremeno biti uključen u više vezanih listi.

U Tabelarnom prikazu 4.3 ponovo se vidi primjer datoteke nastavnika. No sada su uvedene vezane liste za podatke *ODJEL* i *STUPANJ*. U skladu s time, pojavljuju se odgovarajući indeksi listi.

Organizacija pomoću vezanih listi pogodna je onda kad su upiti unaprijed fiksirani i svode se na zadavanje vrijednosti samo jednog podatka. Na primjer: “Ispiši imena svih nastavnika iz odjela Računarstvo”, ili “Ispiši imena svih doktora”. Prednost u odnosu na invertiranu organizaciju je manji i jednostavniji indeks, u kojem nema višestrukih vrijednosti podataka.

Vezane liste su manje pogodne onda kad su u upitu zadane vrijednosti za više podataka. Na primjer, odgovor na upit “Ispiši imena doktora iz odjela Računarstvo” može se dobiti na dva načina:

- čitamo vezanu listu doktora i uvažimo samo one koji su iz odjela Računarstvo,
- čitamo vezanu listu nastavnika iz odjela Računarstvo i uvažimo samo one koji su doktori.

U oba slučaja pročitat ćemo više zapisa nego što ih ima u odgovoru. Od dvije vezane liste bolje je odabrati kraću - zato je zgodno ako u indeksu listi piše i duljina svake liste.

Organizacija pomoću vezanih listi zahtijeva izuzetno mnogo posla prilikom ažuriranja osnovne datoteke. Da bi ubacili, izbacili ili promijenili zadani zapis u osnovnoj datoteci, potrebno je prekrojiti nekoliko vezanih listi, a to se može ostvariti jedino tako da prođemo svim tim listama te promijenimo i razne druge zapise osim zadanog. Koji put se također mora obaviti korekcija u indeksu listi.

Osnovna datoteka o nastavnicima

	<i>MAT_BR</i>	<i>IME</i>	<i>ODJEL</i>	<i>DOB</i>	<i>STUPANJ</i>	Pointer za <i>ODJEL</i>	Pointer za <i>STUPANJ</i>
a_1	12453	Matić, R.	Matematika	35	Dr.sc.	a_2	a_3
a_2	16752	Pavić, D.	Matematika	26	Dipl.inž.	a_6	a_5
a_3	27453	Horvat, I.	Računarstvo	37	Dr.sc.	a_5	a_4
a_4	34276	Dimić, J.	Fizika	55	Dr.sc.	a_7	a_8
a_5	37564	Katić, K.	Računarstvo	45	Dipl.inž.	a_8	a_7
a_6	43257	Radić, M.	Matematika	32	Mr.sc.	a_9	-
a_7	45643	Janić, Z.	Fizika	24	Dipl.inž.	-	-
a_8	56321	Popović, G.	Računarstvo	34	Dr.sc.	-	a_9
a_9	57432	Simić, J.	Matematika	52	Dr.sc.	-	-

Indeks listi za *ODJEL*

<i>ODJEL</i>	Pointer na početak vezane liste
Fizika	a_4
Matematika	a_1
Računarstvo	a_3

Indeks listi za *STUPANJ*

<i>STUPANJ</i>	Pointer na početak vezane liste
Dipl.inž.	a_2
Mr.sc.	a_6
Dr.sc.	a_1

Tabelarni prikaz 4.3: Organizacija datoteke s višestrukim vezanim listama.

4.3.3 Podijeljena hash funkcija

Riječ je o poopćenju hash organizacije iz Potpoglavlja 4.2, s time da hash funkcija osim ključa uzima kao argument i ne-ključne podatke. Pretpostavimo da se redni broj pretinca može izraziti kao niz od B bitova. Postupamo na sljedeći način.

- Podijelimo B bitova u skupine: b_1 bitova za podatak A_1 , b_2 bitova za podatak A_2 , ..., b_r bitova za podatak A_r .
- Zdamo pogodne "male" hash funkcije $h_i(v_i)$, $i = 1, 2, \dots, r$, gdje h_i preslikava vrijednost v_i podatka A_i u niz od b_i bitova.
- Redni broj pretinca za zapis u kojem je $A_1 = v_1$, $A_2 = v_2$, ..., $A_r = v_r$ zadaje se spajanjem malih nizova bitova, Dakle: $h(v_1, v_2, \dots, v_r) = h_1(v_1)|h_2(v_2)|\dots|h_r(v_r)$. Ovdje je $|$ oznaka za "lijepljenje" nizova bitova.

Doprinos jednog podatka u razdijeljenoj hash funkciji treba biti proporcionalan s veličinom domene tog podatka i s frekvencijom njegovog pojavljivanja u upitima. Dakle, veća domena ili češće pojavljivanje u upitima zahtijeva veći broj bitova.

Kao primjer, promatramo zapise o studentima fakulteta i želimo ih spremati u hash datoteku s $1024 (= 2^{10})$ pretinaca. Definicija tipa zapisa u jeziku C izgleda ovako:

```
typedef struct {
    int SNO;
    char SNAME[20];
    enum {1,2,3,4} LEVEL;
    enum {M,F} GENDER;
} STUDENT;
```

Podijelimo 10 bitova u rednom broju pretinca na sljedeći način: 4 bita za **SNO**, 3 bita za **SNAME**, 2 bita za **LEVEL**, 1 bit za **GENDER**. Zadaјemo sljedeće hash funkcije, gdje su v_1 , v_2 , v_3 , v_4 vrijednosti za **SNO**, **SNAME**, **LEVEL**, **GENDER**:

$$\begin{aligned}
 h_1(v_1) &= v_1 \% 16 \text{ (ostatak kod dijeljenja s 16),} \\
 h_2(v_2) &= (\text{broj znakova u } v_2 \text{ koji su različiti od bjeline}) \% 8,
 \end{aligned}$$

$$\begin{aligned} h_3(v_3) &= v_3 - 1, \\ h_4(v_4) &= (0 \text{ za } v_4 \text{ jednak } M, 1 \text{ inače}). \end{aligned}$$

Tada je redni broj pretinca za zapis (58651, Smith, 3, M) zadan s $(1101|101|10|0)_2 = (748)_{10}$.

Da bi pronašli zapis (ili više njih) u kojem je $A_1 = v_1, A_2 = v_2, \dots, A_r = v_r$, računamo redni broj pretinca $h(v_1, v_2, \dots, v_r)$ i sekvencijalno pretražimo taj jedan pretinac. Ako u upitu nije fiksirana vrijednost podatka A_i , tada b_i bitova u rednom broju pretinca ostaje nepoznato, a broj pretinaca koje moramo pretražiti povećava se 2^{b_i} puta. Što manje vrijednosti za A_1, A_2, \dots, A_r je poznato, to će biti veći broj pretinaca koje moramo pretražiti.

Na primjer, ako tražimo sve muške studente na drugoj godini, tada su nam u rednom broju pretinca poznata zadnja 3 bita: 010, no prvih 7 bitova je nepoznato. Treba pretražiti $2^7 = 128$ pretinaca, dakle $1/8$ datoteke.

Prednost podijeljene hash funkcije u odnosu na invertiranu datoteku je da se ne troši dodatni prostor za indekse. Također, operacije ubacivanja, izbacivanja i promjene zapisa su znatno jednostavnije budući da nema indeksa koje treba održavati. No traženje zapisa u kojem su specificirane vrijednosti samo nekih od podataka traje dulje nego kod invertirane organizacije. Smatra se da je organizacija pomoću podijeljene hash funkcije dobra za datoteke koje nisu prevelike i čiji sadržaj se često mijenja.

5

IMPLEMENTACIJA RELACIJSKIH OPERACIJA

5.1 Implementacija prirodnog spoja

Gradivo izloženo u Poglavlju 4 uglavnom predstavlja “statički” aspekt fizičke organizacije baze podataka. No, kod relacijskih baza težište je bačeno na “dinamički” aspekt, koji se svodi na izvrednjavanje izraza u relacijskoj algebri. Stoga, da bi bolje razumjeli što se sve dešava unutar jednog relacijskog DBMS-a, potrebno je proučiti kako se fizički odvija izvrednjavanje algebarskog izraza. Osnovni korak je izvrednjavanje pojedine algebarske operacije. Raspravljat ćemo o implementaciji triju najvažnijih operacija: prirodnog spoja (ovo potpoglavlje), te selekcije i projekcije (iduće potpoglavlje). U zadnjem potpoglavlju reći ćemo nešto o optimalnom izvrednjavanju cijelog izraza.

Promatramo relacije $R_1(A, B)$ i $R_2(B, C)$ sa zajedničkim atributom B . Označimo sa $S(A, B, C)$ prirodni spoj od R_1 i R_2 . Svaka od ovih triju relacija fizički se prikazuje jednom (istoimenom) datotekom; n-torke se pretvaraju u zapise, a atributi u istoimene osnovne podatke. Razmotrit ćemo nekoliko načina kako da se pomoću datoteki R_1 i R_2 generira datoteka S .

5.1.1 Algoritam ugniježđenih petlji

To je očigledan, makar ne nužno i najefikasniji način. Osnovna ideja je sljedeća.

```
inicijaliziraj praznu S;  
učitaj prvi zapis iz  $R_1$ ;  
dok ( nismo prešli kraj od  $R_1$  ) {  
    učitaj prvi zapis iz  $R_2$ ;  
    dok ( nismo prešli kraj od  $R_2$  ) {  
        ako ( tekući zapisi iz  $R_1$  i  $R_2$  sadrže istu vrijednost za  $B$  )  
            stvori kombinirani zapis i ispiši ga u  $S$ ;  
        pokušaj učitati idući zapis iz  $R_2$ ;  
    }  
    pokušaj učitati idući zapis iz  $R_1$ ;  
}
```

Za svaki zapis iz R_1 moramo iznova sekvencijalno čitati cijelu datoteku R_2 . Algoritam se može poboljšati tako da u glavnu memoriju učitamo segment od što više blokova iz R_1 . Preinačimo petlje tako da uspoređujemo svaki zapis učitani iz R_2 sa svakim zapisom iz R_1 koji je trenutno u glavnoj memoriji. Nakon što smo pročitali cijelu R_2 i obavili sva potrebna uspoređivanja, učitamo idući segment od R_1 u glavnu memoriju te ponavljamo postupak. R_1 se očigledno čita samo jednom. R_2 se čita onoliko puta koliko ima segmenata u R_1 . Poboljšana verzija algoritma osobito je dobra onda kad je jedna od datoteki dovoljno mala da cijela stane u glavnu memoriju. Tada se i R_1 i R_2 čitaju samo jednom.

5.1.2 Algoritam zasnovan na sortiranju i sažimanju

Pretpostavimo da su datoteke R_1 i R_2 uzlazno sortirane po zajedničkom podatku B . Tada u R_1 i u R_2 možemo uočiti skupine uzastopnih zapisa s istom vrijednošću za B . Datoteka S koja sadrži prirodni spoj od R_1 i R_2 lagano se može generirati sljedećim algoritmom koji podsijeca na klasično sažimanje.

```
inicijaliziraj praznu S;
učitaj prvu skupinu zapisa iz  $R_1$ ;
učitaj prvu skupinu zapisa iz  $R_2$ ;
dok ( nismo prešli kraj ni od  $R_1$  ni od  $R_2$  ) {
    ako ( tekuća skupina zapisa iz  $R_1$  sadrži manju
    vrijednost za  $B$  nego tekuća skupina zapisa iz  $R_2$  )
        pokušaj učitati iduću skupinu zapisa iz  $R_1$ ;
    inače ako ( tekuća skupina zapisa iz  $R_2$  sadrži
    manju vrijednost za  $B$  nego tekuća skupina zapisa iz  $R_1$  )
        pokušaj učitati iduću skupinu zapisa iz  $R_2$ ;
    inače {
        svaki zapis iz tekuće skupine iz  $R_1$  kombiniraj sa svakim zapisom
        iz tekuće skupine iz  $R_2$  te sve generirane zapise ispiši u  $S$ ;
        pokušaj učitati iduću skupinu zapisa iz  $R_1$ ;
        pokušaj učitati iduću skupinu zapisa iz  $R_2$ ;
    }
}
```

Pretpostavili smo da su skupine zapisa iz R_1 odnosno R_2 s istom vrijednošću za B dovoljno male tako da stanu u glavnu memoriju. Pod ovakvim pretpostavkama se i R_1 i R_2 čitaju samo jednom. Ukoliko skupine ne stanu u glavnu memoriju, algoritam treba preraditi tako da učitava segment po segment od svake skupine. Segmenti jedne od datoteki će se tada morati više puta učitavati.

Ako R_1 i R_2 nisu sortirane kao što se tražilo u opisanom algoritmu, tada ih najprije treba sortirati, pa tek onda računati prirodni spoj. U literaturi postoje mnogi dobri algoritmi za sortiranje, no oni obično predviđaju da cijela datoteka stane u glavnu memoriju. Da bi sortirali veću datoteku, dijelimo je na segmente koji stanu u glavnu memoriju, posebno sortiramo svaki segment i ispisujemo ga natrag u vanjsku memoriju. Dalje se sortirani segmenti postepeno sažimaju u sve veće i veće, sve dok na kraju ne dobijemo cijelu sortiranu datoteku. Riječ je o prilično dugotrajnom postupku koji zahtijeva višestruko prepisivanje cijele datoteke. Dakle sortiranje polaznih R_1 i R_2 trajat će znatno dulje nego generiranje S od već sortiranih R_1 i R_2 . Ipak, ako su R_1 i R_2 jako velike, cijeli postupak se isplati u odnosu na algoritam ugniježenih petlji.

5.1.3 Algoritam zasnovan na indeksu

Pretpostavimo da jedna od datoteki R_1 i R_2 , na primjer R_2 , ima sekundarni indeks za zajednički podatak B . Tada se datoteka S , koja sadrži prirodni spoj od R_1 i R_2 , može generirati na sljedeći način.

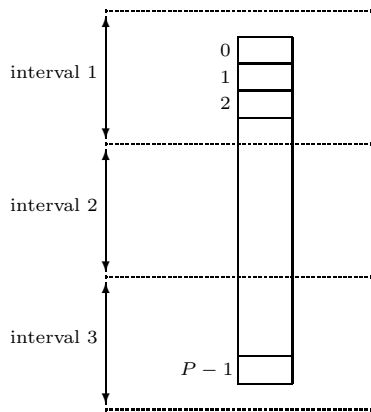
```
inicijaliziraj praznu S;
učitaj prvi zapis iz  $R_1$ ;
dok ( nismo prešli kraj od  $R_1$  ) {
    pomoću indeksa pronađi i učitaj sve zapise iz  $R_2$ 
    koji imaju istu vrijednost za  $B$  kao tekući zapis iz  $R_1$ ;
    tekući zapis iz  $R_1$  kombiniraj sa svakim od učitanih
    zapisa iz  $R_2$  te generirane zapise ispiši u  $S$ ;
    pokušaj učitati idući zapis iz  $R_1$ ;
}
```

Algoritam jednom pročita cijelu R_1 . No iz R_2 se neposredno čitaju samo oni zapisi koji sudjeluju u prirodnom spoju. To može dovesti do značajne uštede u obimu posla. Ako i R_1 i R_2 imaju indeks za B , tada treba sekvencijalno čitati manju datoteku, a koristiti indeks veće datoteke.

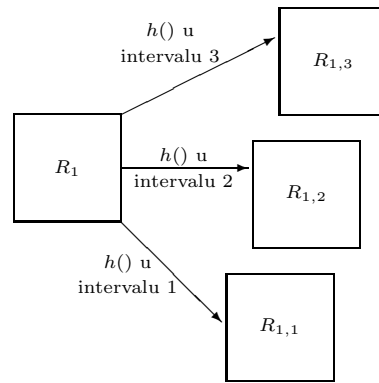
5.1.4 Algoritam zasnovan na hash funkciji i razvrstavanju

Zadajemo hash funkciju h koja ovisi o zajedničkom podatku B . Kombinacija zadanog zapisa iz datoteke R_1 sa zadanim zapisom iz datoteke R_2 pojavljuje se u prirodnom spoju S ako i samo ako oba zadana zapisa imaju jednaku vrijednost za B . Zato hash funkcija za oba takva zapisa daje istu vrijednost. Razvrstavanjem zapisa iz R_1 i R_2 u skupine onih s bliskom vrijednošću h lakše ćemo odrediti koji parovi zapisa se mogu kombinirati.

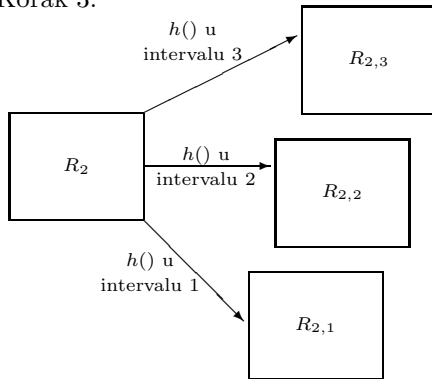
Korak 1:



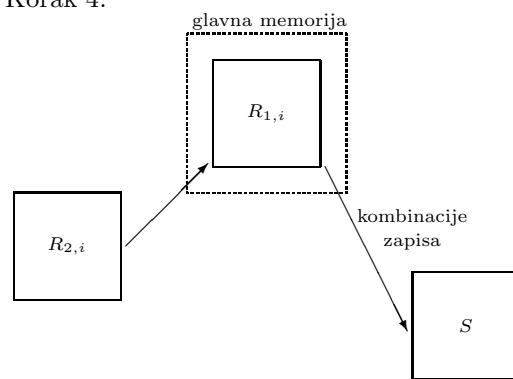
Korak 2:



Korak 3:



Korak 4:



Slika 5.1: Izvrednjavanje prirodnog spoja pomoću hash funkcije i razvrstavanja

Neka je R_1 manja od R_2 . Algoritam se tada sastoji od sljedećih pet koraka. Pritom su koraci 1, 2, 3 i 4 ilustrirani na Slici 5.1.

1. Inicijaliziraj praznu datoteku S . Odaberi hash funkciju h . Podijeli ukupni raspon hash vrijednosti na k podjednakih intervala. Pritom je k odabran tako da $1/k$ od datoteke R_1 stane u glavnu memoriju.
2. Čitaj sekvencijalno R_1 i razvrstaj njene zapise u k skupina (pomoćnih datoteki) tako da jedna skupina sadrži sve zapise iz R_1 koje h preslikava u jedan od intervala. Ako je h zaista uniformna hash funkcija, tada su sve skupine podjednako velike - znači da jedna skupina stane u glavnu memoriju.
3. Čitaj sekvencijalno R_2 i razvrstaj njene zapise u k skupina (pomoćnih datoteki), slično kao što smo to napravili s datotekom R_1 .
4. Odaberi jedan od intervala za vrijednost h . Učitaj u glavnu memoriju odgovarajuću skupinu zapisa iz R_1 . Sekvencijalno čitaj odgovarajuću skupinu zapisa iz R_2 . Kombiniraj tekući zapis iz R_2 sa svim zapisima iz R_1 (u glavnoj memoriji) koji imaju jednaku vrijednost za B . Dobivene kombinacije ispiši u S .

5. Ponovi korak 4, s time da odabereš novi interval za vrijednost od h . Ako smo u koraku 4 već obradili svaki od k intervala, tada je algoritam završen.

Analizom algoritma lagano se vidi da se svaka od datoteki R_1 i R_2 čita točno dvaput.

Na kraju ovog potpoglavlja usporedit ćemo četiri izložena načina za implementiranje prirodnog spoja.

- Ako su datoteke R_1 i R_2 već sortirane po zajedničkom podatku B , tada je najefikasniji algoritam zasnovan na sortiranju i sažimanju.
- Ukoliko je jedna datoteka dovoljno mala da stane u glavnu memoriju, treba odabrati algoritam ugniježđenih petlji.
- Ako je jedna datoteka znatno veća od druge i ima odgovarajući indeks, tada je najbolji algoritam zasnovan na indeksu.
- Za velike datoteke R_1 i R_2 bez indeksa, najbolji je algoritam zasnovan na hash funkciji i razvrstavanju.

Ovime nisu iscrpljeni svi slučajevi. Općenito ne možemo baš lako odrediti koji algoritam je najbolji, već to zahtijeva detaljnije procjene koje ovise o veličini R_1 i R_2 , o veličini glavne memorije, distribuciji vrijednosti za B itd.

5.2 Implementacija selekcije, projekcije i ostalih operacija

Osim prirodnog spoja, najvažnije relacijske operacije su selekcija i projekcija. Izložiti ćemo osnovne ideje za implementaciju tih dviju operacija, a zatim ćemo kratko napomenuti kako se implementiraju ostale operacije.

5.2.1 Implementacija selekcije

Zadana je relacija R i Booleovski uvjet \mathcal{B} . R je fizički prikazana istoimenom datotekom, na standardni način. Implementacija selekcije R **where** \mathcal{B} ovisi o obliku uvjeta \mathcal{B} , no obično se svodi na traženje zapisa u datoteci R sa zadanom vrijednošću nekih podataka. Dakle, obično se radi o pristupu na osnovu primarnog ključa ili o pristupu na osnovu ostalih podataka. U poglavlju 4 već smo opisali algoritme za pristup. Sad ćemo samo ukratko rezimirati ideje.

Naivni algoritam bio bi: sekvencijalno čitanje cijele R i provjera svakog zapisa da li on zadovoljava \mathcal{B} . Ako je R velika, to traje predugo, pa treba izgraditi pomoćne strukture podataka koje omogućuju direktniji pristup do traženih zapisa. Invertiranje datoteki pomoću indeksa je vjerojatno najfleksibilnija metoda. Većina današnjih relacijskih DBMS-a oslanja se na jednostavno organizirane datoteke nadopunjene sekundarnim indeksima koji su prikazani B-stablama (primarni gusti indeks je samo specijalni slučaj sekundarnog). Rjeđe se koriste hash organizacije i višestruke vezane liste. Neki oblici uvjeta \mathcal{B} zahtijevaju da u R tražimo zapise gdje vrijednost zadanog podatka nije fiksirana, već se kreće u zadanom intervalu. Primijetimo da indeks-B-stablo podržava ovakvo intervalno pretraživanje, budući da je u njemu sačuvan sortirani redoslijed zapisa po dotičnom podatku. Hash organizacija ne podržava intervalno pretraživanje.

5.2.2 Implementacija projekcije

Zadane je relacija R i njen atribut A . R je fizički prikazana istoimenom datotekom na standardni način. Da bi generirali datoteku koja odgovara projekciji $S = R[A]$, očito treba pročitati cijelu datoteku R i izdvojiti sve vrijednosti podatka A koje se pojavljuju. No ista vrijednost za A može se pojaviti više puta. Osnovni problem implementiranja projekcije je: kako u S eliminirati zapise-duplikate?

Najjednostavniji algoritam za implementaciju projekcije zasnovan je na ugniježđenim petljama. Vanjska petlja čita datoteku R , a unutrašnja petlja prolazi trenutno stvorenim dijelom datoteke S :

```

inicijaliziraj praznu S;
učitaj prvi zapis iz R;
dok ( nismo prešli kraj od R ) {
    duplikat = 0;
    učitaj prvi zapis iz S;
    dok ( nismo prešli kraj od S i duplikat==0 ) {
        ako ( tekući zapisi iz R i S sadrže istu vrijednost za A ) duplikat = 1;
        pokušaj učitati idući zapis iz S;
    }
    ako ( duplikat == 0 )
        prepisi vrijednost za A iz tekućeg zapisa iz R na kraj od S kao novi zapis;
    pokušaj učitati idući zapis iz R;
}

```

Ako je R velika, algoritam s ugniježđenim petljama zahtijeva previše vremena. Tada je bolje postupiti na sljedeći način: izdvojiti sve vrijednosti za A koje se pojavljuju u R te zatim sortirati niz izdvojenih vrijednosti. U sortiranom nizu duplikati će se pojavljivati jedan iznad drugoga, pa ih je lako eliminirati jednim sekvencijalnim čitanjem. Druga ideja je: razvrstati izdvojene vrijednosti za A u skupine pomoću hash funkcije. Duplikati će se tada naći u istoj skupini, pa ih je opet lako eliminirati.

5.2.3 Implementacija ostalih operacija.

Kartezijski produkt dviju relacija R_1 i R_2 implementira se ugniježđenom petljom. Bolji algoritam nije moguć jer se ionako svaki zapis iz datoteke R_1 mora kombinirati sa svakim zapisom iz datoteke R_2 .

Unija dviju relacija R_1 i R_2 implementira se na očigledni način, s time da, slično kao kod projekcije, treba eliminirati zapise-duplikate. Opet pomaže sortiranje datoteke R_1 i R_2 , ili korištenje hash funkcije.

Presjek dviju relacija može se shvatiti kao specijalni slučaj prirodnog spoja gdje su svi atributi zajednički. Zato algoritmi za računanje presjeka liče na one za računanje prirodnog spoja. Implementiranje skupovne razlike je također slično.

Daljnji relacijski operatori mogu se izraziti pomoću već razmatranih, pa se oni obično ne implementiraju zasebno.

5.3 Optimalno izvrednjavanje algebarskog izraza

Relacijski DBMS interno reprezentira korisnikov upit kao izraz u relacijskoj algebri. U potpoglavlju 3.4 govorili smo o višoj (logičkoj) razini optimizacije upita, koji se svodi na preformuliranje izraza u oblik ekvivalentan polaznom ali pogodniji sa stanovišta izvrednjavanja. U ovom potpoglavlju govorimo o nižoj (fizičkoj) razini optimizacije, koja se svodi na izbor dobrih algoritama za izvrednjavanje izraza dobivenog nakon “logičke” faze.

Izvrednjavanje izraza očito se svodi na izvrednjavanje svake od osnovnih operacija. Obično je riječ o operacijama prirodnog spoja, selekcije, projekcije, te možda o još ponekoj. Za svaku od osnovnih operacija DBMS raspolaže s nekoliko algoritama. Također, DBMS-u su poznati parametri kao što su: kardinalnost relacija (datoteki), veličina glavne memorije, postojanje ili nepostojanje odgovarajućih indeksa, i slično. Služeći se tim parametrima, DBMS za zadanu operaciju i svaki od raspoloživih algoritama procjenjuje vrijeme potrebno da se ta operacija izvrši tim algoritmom. Procjene se dobivaju na osnovu ugrađenih heurističkih pravila. DBMS zatim za zadanu operaciju odabire algoritam s najmanjim procijenjenim vremenom.

Osim ovakve izolirane optimizacije pojedinih operacija, suvremeni DBMS-i nastoje također promatrati i izraz kao cjelinu, te otkriti daljnje mogućnosti za uštedu posla. Na primjer, zamislimo da treba izvrijedniti izraz:

$$((R_1 \text{ join } R_2) \text{ join } (R_3 \text{ where } A = 10)) [A, B] .$$

Tada nije potrebno do kraja izračunati prvi prirodni spoj prije nego što počnemo računati drugi prirodni spoj te zatim projekciju. Umjesto toga, svaki puta kad proizvedemo novu n-torku iz $(R_1 \text{ join } R_2)$, proslijedimo je odmah da bude spojena sa selektiranim n-torkama iz R_3 , a rezultirajuće n-torke se

odmah projiciraju na atribute A i B . Ovakva “pipeline” strategija može rezultirati velikim uštedama u vremenu, budući da se privremene relacije ne moraju spremati u vanjsku memoriju ni ponovo čitati.

I na logičkom i na fizičkom nivou optimizacije upita relacijski DBMS donosi odluke služeći se ugrađenim “znanjem”. U tom smislu, relacijski DBMS predstavlja primjer ekspertnog sustava, dakle radi se o softveru s osobinama umjetne inteligencije. Pravila za donošenje odluka nisu ni izdaleka savršena, te su predmet daljnjeg intenzivnog istraživanja.

6

INTEGRITET I SIGURNOST PODATAKA

6.1 Čuvanje integriteta

Poglavlje 6 posvećeno je raznim aspektima integriteta i sigurnosti podataka. Riječ je o problemima čije rješavanje je nužno da bi velika višekorisnička baza podataka mogla uspješno funkcionirati. U ovom potpoglavlju bavimo se čuvanjem **integriteta** baze. Pod time se misli na čuvanje **korektnosti** i **konzistentnosti** podataka. Integritet se lako može narušiti na primjer pogrešnim upisom neopeznih korisnika ili pogrešnim radom aplikacijskog programa.

Voljeli bismo kad bi se baza podataka mogla sama braniti od narušavanja integriteta. U tu svrhu, suvremeni DBMS-i dozvoljavaju projektantu baze da definira tzv. **ograničenja** (constraints). Riječ je o uvjetima (pravilima) koje konzistentni podaci moraju zadovoljavati. Kod svake promjene podataka DBMS će automatski provjeravati da li su sva ograničenja zadovoljena. Ako neko ograničenje nije zadovoljeno, tada DBMS neće izvršiti traženu promjenu, već će poslati poruku o greški.

6.1.1 Ograničenja kojima se čuva integritet domene

Izražavaju činjenicu da vrijednost atributa mora biti iz zadane domene. Zahtjev da vrijednost primarnog atributa ne smije biti prazna također spada u ovu kategoriju.

Uzmimo na primjer da u relaciji *STUDENT* postoji atribut *DOB*. Tada bi ograničenje moglo biti: "Vrijednost za *DOB* je cijeli broj između 10 i 60". U većini DBMS-a orijentiranih na SQL, ograničenje na integritet domene prvenstveno se izražava time što se u naredbi **CREATE** atributu pridruži tip (uz eventualnu klauzulu **NOT NULL**). Popis podržanih tipova obično nije velik (standardno to su char-stringovi fiksirane duljine, brojevi s fiksnom točkom, cijeli brojevi, brojevi s plivajućom točkom, datumi, ...). Zato na taj način obično nećemo biti u mogućnosti da precizno izrazimo naše ograničenje. Na primjer, morat ćemo zadati da je *DOB* tipa **SMALLINT**. DBMS će tada automatski spriječiti upis vrijednosti 12.5 za *DOB*, ali će propustiti -5. Integritet domene tada se u potpunosti može čuvati jedino tako da ugradimo kontrole u naš aplikacijski program. Neki DBMS-i omogućuju da se unutar naredbe **CREATE** ugradi i precizniji uvjet kojeg vrijednosti zadanog tipa moraju zadovoljavati da bi mogle biti vrijednosti dotičnog atributa. Kod takvih DBMS-a mogli bi u potpunosti zadati naše ograničenje za *DOB*: deklarirali bi da je *DOB* tipa **SMALLINT**, uz dodatni uvjet: $(DOB \geq 10) \text{ AND } (DOB \leq 60)$.

6.1.2 Ograničenja kojima se čuva integritet unutar relacije

Čuva se korektnost veza između atributa unutar relacije (na primjer funkcionalne ovisnosti). Najvažniji primjer takvog ograničenja je ono koje traži da dvije n-torke unutar iste relacije ne smiju imati jednaku vrijednost ključa.

Stariji DBMS-i orijentirani na SQL nisu pružali mogućnost da se direktno izrazi svojstvo ključa. Postojao je doduše jedan zaobilazni način - **UNIQUE** indeks. Noviji standardi za SQL predviđaju klauzulu **PRIMARY KEY** odnosno **UNIQUE** unutar naredbe **CREATE**, što znači da se traženo ograničenje za ključ može eksplicitno zadati. Na primjer:

```
CREATE TABLE STUDENT
(SNO INTEGER NOT NULL,
 SNAME CHAR(20),
 LEVEL SMALLINT,
 PRIMARY KEY(SNO));
```

```
CREATE TABLE COURSE
(CNO INTEGER NOT NULL,
 TITLE CHAR(40),
 LNAME CHAR(20),
 PRIMARY KEY(CNO));
```

Nakon ovih definicija, DBMS neće dozvoliti da se u relaciju *STUDENT* upišu dvije n-torke s istim *SNO*, niti da se u relaciju *COURSE* upišu dvije n-torke s istim *CNO*.

6.1.3 Ograničenja kojima se čuva referencijalni integritet

Čuva se korektnost i konzistentnost veza između relacija. Uglavnom je riječ o ograničenjima koja se odnose na strani ključ, dakle na atribut u jednoj relaciji koji je ujedno primarni ključ u drugoj relaciji. Svaka vrijednost takvog atributa u prvoj relaciji mora biti prisutna i u drugoj relaciji.

Stariji DBMS-i orijentirani na SQL nisu pružali mogućnost da se izrazi svojstvo stranog ključa. Odgovarajuću provjeru morao je obavljati aplikacijski program. Noviji standardi za SQL predviđaju klauzulu **FOREIGN KEY** u naredbi **CREATE**, kojom se задаje odgovarajuće ograničenje. Na primjer, u kontekstu prethodnih definicija, možemo dalje pisati:

```
CREATE TABLE REPORT
(SNO INTEGER NOT NULL,
 CNO INTEGER NOT NULL,
 MARK SMALLINT,
 PRIMARY KEY(SNO,CNO),
 FOREIGN KEY(SNO) REFERENCES STUDENT,
 FOREIGN KEY(CNO) REFERENCES COURSE);
```

Nakon ovih definicija DBMS neće dozvoliti da se naruši referencijalni integritet za tri tabele. Na primjer, u *REPORT* se neće moći upisati n-torka s vrijednošću *SNO* koja se ne pojavljuje u *STUDENT*. Ili na primjer, iz *COURSE* se neće moći brisati n-torka s vrijednošću *CNO* koja se pojavljuje u *REPORT*.

Izbor ograničenja u današnjim DBMS-ima ipak nije dovoljan da izrazi sva moguća pravila integriteta. Također, mogućnosti pojedinih DBMS-a se razlikuju. Ipak, današnji DBMS-i u tom pogledu su znatno napredniji od onih prije 15-tak godina. Treba biti svjestan da svako ograničenje predstavlja teret prilikom ažuriranja podataka (troši se vrijeme na provjeru). Zato prilikom zadavanja ograničenja ne treba pretjerivati.

6.2 Istovremeni pristup

Većina baza podataka po svojoj prirodi je **višekorisnička**. Znači jedan te isti podatak, pohranjen na jednom mjestu, potreban je raznim osobama i raznim aplikacijama, možda čak u isto vrijeme. Na primjer, broj prodanih avionskih karata za isti let treba simultano biti dostupan raznim poslovnica avionske kompanije. Od DBMS-a se zato traži da korisnicima omogući **istovremeni pristup** do podataka. Obično je riječ o prividnoj istovremenosti (dijeljenje vremena istog računala). Ipak, DBMS i u tom slučaju mora pažljivo koordinirati konfliktne radnje. Svaki korisnik treba imati dojam da sam radi s bazom.

6.2.1 Transakcije i serijalizabilnost

Rad korisnika s bazom podataka svodi se na pokretanje unaprijed definiranih procedura, tzv. **transakcija**. Makar jedna transakcija sa korisničkog stanovišta predstavlja jednu nedjeljivu cjelinu, ona se

obično realizira kao niz od nekoliko elementarnih zahvata u samoj bazi. Na primjer, za avionsku kompaniju prodaja jedne avionske karte može predstavljati jednu transakciju. U sklopu jedne karte obično je uključeno više letova (prelasci, povratak), pa to povlači nekoliko operacija upisa u bazu.

Osnovno svojstvo transakcije je da ona prevodi bazu iz jednog konzistentnog stanja u drugo. No među-stanja, koja nastaju nakon pojedinih operacija unutar transakcije, mogu biti nekonzistentna. Da bi se čuvao integritet baze, transakcija mora u cijelosti biti izvršena, ili uopće ne smije biti izvršena. Transakcija koja iz bilo kojeg razloga nije do kraja bila obavljena, mora biti neutralizirana - dakle svi podaci koje je ona do trenutka prekida promijenila moraju natrag dobiti svoje polazne vrijednosti. U SQL-u se početak transakcije određuje implicitno ili posebnom naredbom, dok završetak (i način završetka) mora eksplicitno biti zadan. Sljedeći nizovi SQL naredbi predstavljaju uspješno izvršenu transakciju, odnosno neuspješnu i neutraliziranu transakciju:

BEGIN WORK*naredba 1 ;**naredba 2 ;*

.....;

*naredba k ;***COMMIT WORK ;**

odnosno

BEGIN WORK*naredba 1 ;**naredba 2 ;*

.....;

*(greška) ;***ROLLBACK WORK ;**

U višekorisničkoj bazi dešavat će se da se nekoliko transakcija izvodi paralelno. Osnovne operacije koje pripadaju raznim transakcijama tada će se vremenski ispreplesti. Tražimo da učinak tih transakcija bude isti kao da su se one izvršavale sekvencijalno, dakle jedna iza druge u nekom (bilo kojem) redoslijedu. Traženo svojstvo, da učinak istovremenog izvršavanja transakcija mora biti ekvivalentan nekom sekvencijalnom izvršavanju, naziva se *serijalizabilnost*. Dakle, ukoliko to svojstvo zaista vrijedi, kažemo da je dotično paralelno izvršavanje skupa transakcija bilo **serijalizabilno**. Sljedeći primjer s avionskom kompanijom pokazuje da se a priori ne može garantirati serijalizabilnost, te da nekontrolirano paralelno izvršavanje transakcija može dovesti do neželjenih efekata.

Dva putnika istovremeno stižu u dvije različite poslovnice avionske kompanije i traže kartu za isti let istog dana. U tom trenutku postoji samo jedno slobodno sjedalo. Službenici sa svojih terminala pokreću dvije transakcije, T_1 i T_2 , koje DBMS "istovremeno" obavlja. Dolazi do slijedećeg vremenskog redoslijeda obavljanja elementarnih operacija.

1. T_1 učitava iz baze broj slobodnih sjedala.
2. T_1 smanjuje pročitane vrijednost s 1 na 0. Promjena je za sada učinjena samo u radnoj memoriji računala.
3. T_2 učitava iz baze broj slobodnih sjedala. Budući da je stanje baze još uvijek nepromijenjeno, učitani broj je neažuran, dakle 1.
4. T_2 smanjuje pročitane vrijednost s 1 na 0. Promjena je opet učinjena samo u radnoj memoriji.
5. T_1 unosi u bazu promijenjenu vrijednost za broj slobodnih sjedala. Dakle u bazi sada piše da nema slobodnih sjedala.
6. Slično i T_2 unosi u bazu svoju promijenjenu vrijednost za broj slobodnih sjedala. Dakle u bazu se ponovo upisuje da nema slobodnih sjedala.
7. Budući da je na početku svog rada naišla na broj slobodnih sjedala veći od 0, T_1 izdaje putniku kartu.
8. Iz istih razloga i T_2 izdaje drugom putniku kartu.

Vidimo da je avionska kompanija prodala jednu kartu previše. Ili, drugim riječima, dva putnika sjede na istom sjedalu.

Ozbiljni DBMS ne smije dopustiti slijed događaja iz prethodnog primjera, dakle DBMS u svakom trenutku mora garantirati serijalizabilnost. Znači, jedan (bilo koji) putnik trebao je dobiti kartu, dok je drugi trebao dobiti obavijest da više nema slobodnih mjesta. Zato je potrebna neka vrsta kontrole (koordinacije) istovremenog izvršavanja transakcija. Razni DBMS-i to rade na razne načine. Uobičajena tehnika zasniva se na lokotima.

6.2.2 Lokoti i zaključavanje

Lokoti su pomoćni podaci koji služe za koordinaciju konfliktnih radnji. Baza je podijeljena na više dijelova, tako da jednom dijelu odgovara točno jedan lokot. Transakcija koja želi pristupiti nekom podatku najprije mora “uzeti” odgovarajući lokot i time **zaključati** dotični dio baze. Čim je obavila svoju operaciju, transakcija treba “vratiti” lokot i time **otključati** podatke. Kad transakcija naiđe na podatke koji su već zaključani, ona mora čekati dok ih prethodna transakcija ne otključa. Time se zapravo izbjegava (sasvim) istovremeni pristup istom podatku.

Ovaj mehanizam dovoljan je da otkloni probleme iz prošlog primjera. Zamislimo da sada obje transakcije T_1 i T_2 zaključavaju podatak u bazi kojem misle pristupiti. Tada će T_1 prva zaključati broj slobodnih sjedala, a otključat će ga tek nakon što ga ažurira. T_2 će (nakon kratkog čekanja) učitati već ažuriranu vrijednost (0 slobodnih sjedala), pa drugi putnik neće dobiti kartu. Znači “istovremeno” izvršavanje T_1 i T_2 sada je serijalizabilno.

Veličina dijela baze kojem je pridružen jedan lokot određuje **zrnatost** zaključavanja (granularity). Što je zrno krupnije, to je kontrola zaključavanja jednostavnija za DBMS, no stupanj paralelnosti rada je manji. Zrnatost suvremenih DBMS-a je obično reda veličine n -torke ili bar fizičkog bloka.

Upotreba lokota krije u sebi i određene opasnosti. Najveća od njih je mogućnost međusobne **blokade** dviju ili više transakcija (tzv. “deadlock”). Da bi točnije objasnili o čemu se radi, opet ćemo se poslužiti jednim zamišljenim “scenarijem” vezanim uz našu avionsku kompaniju.

Uzmimo da za zadani let i za svako sjedalo u avionu baza podataka pohranjuje ime putnika koji sjedi na tom sjedalu. Pretpostavimo da postoji transakcija kojom dva putnika mogu zamijeniti sjedala (na primjer pušač i nepušač). Zamislimo sada da su istovremeno pokrenute dvije transakcije ovakve vrste, nazovimo ih T_1 i T_2 . Neka T_1 mijenja imena putnika na sjedalima 1 i 2, a T_2 obavlja istu zamjenu ali u suprotnom redoslijedu. Tada je moguć slijedeći način obavljanja elementarnih operacija.

1. T_1 zaključa podatak o sjedalu 1 jer mu misli pristupiti.
2. Iz istih razloga T_2 zaključa podatak o sjedalu 2.
3. T_1 traži lokot za sjedalo 2, ali mora čekati jer je T_2 već zaključala dotični podatak.
4. Slično T_2 traži lokot za sjedalo 1, ali mora čekati jer je T_1 već zaključala taj podatak.

Očigledno ni T_1 ni T_2 više ne mogu nastaviti rad - one će vječno čekati jedna drugu.

Software koji koristi lokote mora računati s upravo opisanom mogućnošću blokade transakcija, te mora osigurati da se ta blokada spriječi ili prekine. Rješenje koje se danas najčešće koristi je sljedeće.

Privremeno se dopušta blokada, no povremeno se kontrolira da li ima blokiranih transakcija (traži se ciklus u usmjerenom grafu koji prikazuje koja transakcija čeka koju). Ukoliko takve blokirane transakcije postoje, tada se jedna od njih prekida, neutralizira se njen dotadašnji učinak, te se ona se ponovo starta u nekom kasnijem trenutku.

6.2.3 Dvofazni protokol zaključavanja

Na osnovu do sada pokazanih primjera, moglo bi se povjerovati da korištenje lokota (uz izbjegavanje blokade) daje garanciju za serijalizabilnost. To na žalost nije točno, a pogrešni utisak stekao se zato što su primjeri bili suviše jednostavni. Naime, postoji mogućnost za nekorektno izvršavanje istovremenih transakcija i onda kad se podaci zaključavaju. Da bi to pokazali, još jednom ćemo se poslužiti zamišljenim “scenarijem” vezanim uz našu avionsku kompaniju.

Opet promatramo transakciju kojom na zadanom letu dva putnika mijenjaju sjedala. Uzmimo da su istovremeno pokrenute dvije identične transakcije, T_1 i T_2 , koje obje mijenjaju imena putnika na istim sjedalima 1 i 2. Neka na početku na tim sjedalima sjede Ivan i Marko. Tada je moguć slijedeći redoslijed obavljanja elementarnih operacija.

1. T_1 zaključa sjedalo 1, čita ime Ivan i pamti ga kao prvo ime, te otključa sjedalo 1.
2. T_1 zaključa sjedalo 2, čita ime Marko i pamti ga kao drugo ime, te otključa sjedalo 2.
3. T_1 ponovo zaključa sjedalo 1, upisuje mu zapamćeno drugo ime (dakle Marko), te otključa sjedalo 1.

4. T_2 zaključa sjedalo 1, čita ime Marko i pamti ga kao svoje prvo ime, te otključa sjedalo 1.
5. T_2 zaključa sjedalo 2, čita ime Marko i pamti ga kao svoje drugo ime, te otključa sjedalo 2.
6. T_1 ponovo zaključa sjedalo 2, upisuje mu zapamćeno prvo ime (dakle Ivan), te otključa sjedalo 2.
7. T_2 ponovo zaključa sjedalo 1, upisuje mu svoje zapamćeno drugo ime (dakle Marko), te otključa sjedalo 1.
8. T_2 ponovo zaključa sjedalo 2, upisuje mu svoje zapamćeno prvo ime (dakle opet Marko), te otključa sjedalo 2.

Vidimo da sada na oba sjedala sjedi Marko, a Ivana nema nigdje. Znači opisani način izvršavanja transakcija T_1 i T_2 nije serijalizabilan. Naime, bilo koji sekvencijalni redoslijed izvršavanja proizveo bi dvostruku zamjenu, tj. Ivan bi opet bio na sjedalu 1, a Marko na sjedalu 2.

Upravo navedeni primjer uvjerio nas je da zaključavanje podataka samo po sebi nije garancija za serijalizabilnost. No srećom, stvar se lagano može spasiti. Dovoljno je od transakcija zahtijevati da se pokoravaju nekom strožem “pravilu ponašanja”. Preciznije, može se dokazati da vrijedi slijedeća tvrdnja.

Ako u svakoj od transakcija sva zaključavanja slijede prije prvog otključavanja, tada proizvoljno istovremeno izvršavanje tih transakcija mora biti serijalizabilno.

Navedeno pravilo zove se **dvofazni protokol zaključavanja**. Naime, zaključavanja i otključavanja se zbivaju u dvije razdvojene faze tokom izvršavanja transakcije.

Dvofazni protokol zaključavanja bit će bolje razumljiv ukoliko se vratimo prethodnom primjeru s dvostrukom zamjenom sjedala. Razlog zašto transakcije T_1 i T_2 nisu korektno radile je baš taj što se one nisu pokoravale protokolu. Zaista, obje su otključavale podatke, pa ih zatim opet zaključavale. Da bi bila u skladu sa protokolom, transakcija mora samo jednom zaključati podatak o sjedalu (prije čitanja), te ga samo jednom otključati (nakon ažuriranja). U razdoblju između čitanja i ažuriranja podatak mora ostati zaključan. Lako se uvjeriti da, uz ovu promjenu “ponašanja”, redoslijed događaja iz prošlog primjera više nije moguć. U najmanju ruku, koraci 5 i 6 morati će zamijeniti redoslijed, jer T_2 neće moći pristupiti sjedalu 2 dok ga T_1 nije ažurirala i otključala. Obje transakcije tada će korektno obaviti svoj posao.

6.2.4 Vremenski žigovi

Dvofazni protokol zaključavanja (uz izbjegavanje blokade) predstavlja primjer tehnike za koordinaciju paralelnog izvršavanja transakcija zasnovane na lokotima. No postoje i metode koje ne koriste lokote. Kao primjer, kratko spominjemo tehniku zasnovanu na **vremenskim žigovima** (time stamps).

Svakoj transakciji pridružuje se identifikacijski broj, tzv. vremenski žig. Čitanja i promjene istog podatka dozvoljavaju se samo ako se one odvijaju u redoslijedu vremenskih žigova pripadnih transakcija. U slučaju narušavanja tog redoslijeda, jedna od transakcija mora se prekinuti, neutralizirati i ponovo startati s većim vremenskim žigom. Na primjer, ako T_1 ima žig t_1 , T_2 ima žig $t_2 > t_1$, T_1 želi pročitati podatak x , a T_2 je već mijenjala taj isti x , tada se T_1 mora neutralizirati.

Opisani način izvršavanja transakcija garantira serijalizabilnost. Naime, ukupni učinak svih transakcija je isti kao da se svaka od njih izvršavala trenutačno, u posebnom trenutku.

6.3 Oporavak

U toku svog rada, baza podataka može se naći u “neispravnom” stanju. Razlozi koji mogu dovesti do “oštećenja” baze su:

- prekid transakcije (naredba **ROLLBACK WORK**, posljedica kontrole istovremenog rada, nestanak struje, ...);
- pogrešan rad same transakcije;

- greška u DBMS-u ili operacijskom sustavu;
- hardverska greška (na primjer kvar diska) ili čak fizičko uništenje cijelog računala.

Od suvremenog DBMS-a očekuje se da u svim ovim slučajevima omogući “oporavak” baze, dakle njen povratak u stanje koje je što ažurnije i pritom još uvijek konzistentno. Taj povratak bi se trebao obavljati po mogućnosti automatski, ili bar na relativno jednostavan način. Da bi oporavak bio moguć, DBMS osim same baze mora održavati još i neka pomoćna sredstva; tipično to su: **rezervna kopija baze** (backup copy) i **žurnal datoteka** (journal file, log file). Rezervna kopija i žurnal datoteka omogućuju razne vrste oporavka. Dva najvažnija oblika su: **neutralizacija** prekinute ili pogrešne transakcije, te **ponovno uspostavljanje** baze nakon njenog znatnijeg oštećenja.

6.3.1 Rezervna kopija baze

Dobiva se snimanjem cijele baze na drugi medij (magnetska traka ili drugi disk), i to u trenutku kad smatramo da je baza u konzistentnom stanju. Za vrijeme kopiranja ne smije se obavljati nikakva transakcija koja mijenja podatke. Stvaranje rezervne kopije je dugotrajna operacija koja ometa redovni rad korisnika. Zato se kopiranje ne obavlja suviše često, već periodički u unaprijed predviđenim terminima (na primjer jednom tjedno).

6.3.2 Žurnal datoteka

Riječ je o datoteci gdje je ubilježena “povijest” svake transakcije koja je mijenjala bazu nakon stvaranja zadnje rezervne kopije. Za jednu transakciju žurnal evidentira:

- identifikator transakcije,
- adresu svakog podatka kojeg je transakcija promijenila, zajedno s prethodnom vrijednošću tog podatka (pre-image) i novom vrijednošću (post-image),
- kontrolne točke (checkpoints) u napredovanju transakcije: točka početka, točka isporuke (commit) odnosno odustajanja (rollback).

6.3.3 Neutralizacija jedne transakcije

Riječ je o rutinskoj i vrlo čestoj operaciji koju DBMS obično obavlja automatski. Primjenjuje se za neutralizaciju transakcije koja je počela pisati u bazu no nije došla do kraja. Isti postupak mogao bi se primijeniti za neutralizaciju već dovršene transakcije za koju se ustanovilo da je pogrešna. Svodi se na to da se podacima koje je transakcija mijenjala vrate prethodne vrijednosti. Uobičajeni postupak naziva se **odmotavanje unatrag** (roll-back):

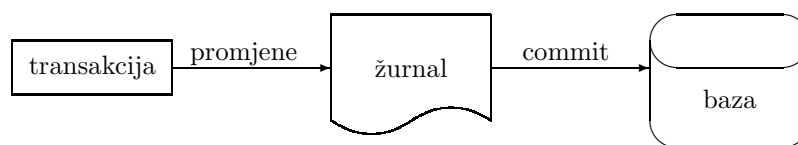
- čita se žurnal i pronalaze se stare vrijednosti (pre-images) podataka koje je transakcija mijenjala;
- te stare vrijednosti se ponovo upisuju na odgovarajuća mjesta u bazu.

Osim toga, ako je neka druga transakcija pročitala u bazi vrijednost unesenu od upravo neutralizirane transakcije, tada treba i tu drugu transakciju neutralizirati (i ponovo startati). Detalji cijelog postupka mogu biti dosta komplicirani i oni ovise o tome kako se obavlja koordinacija paralelnog rada više transakcija.

Stvari se bitno pojednostavnjuju ukoliko zamislimo da DBMS odjednom obavlja samo jednu transakciju. Tada se neutralizacija prekinute transakcije elegantno rješava tzv. **tehnikom odgođenog pisanja** (deferred write). Ta tehnika zahtijeva da se promjene podataka (nastale kao rezultat rada transakcije) ne upisuju u bazu odmah, nego tek nakon što transakcija unese u žurnal točku isporuke. Znači, rad transakcije se odvija u dvije faze, kao što se vidi na Slici 6.1:

- transakcija upisuje u žurnal sve promjene koje bi trebala napraviti u bazi, te nakon toga bilježi točku isporuke;
- čim je ubilježena točka isporuke, DBMS prenosi odjednom sve promjene iz žurnala u bazu.

Uz primjenu ove tehnike, problem neutralizacije prekinute transakcije postaje trivijalan. Naime prekinuta transakcija ne uspijeva u žurnal ubilježiti točku isporuke, pa stoga DBMS ne prenosi njene promjene u bazu.



Slika 6.1: Izvršavanje transakcije primjenom tehnike odgođenog pisanja

6.3.4 Ponovno uspostavljanje baze

Riječ je o izvanrednoj i opsežnoj operaciji koja se pokreće na zahtjev administratora. Primjenjuje se nakon znatnijeg oštećenja baze ili u slučaju njenog potpunog uništenja. Svodi se na ponovni upis svih podataka.

Uobičajeni postupak naziva se **odmotavanje prema naprijed** (roll-forward). U žurnalu moraju biti upisane posebne kontrolne točke koje označavaju trenutke kad je baza još bila konzistentna (obično su to trenutci kad nije bilo aktivnih transakcija). Postupak je tada sljedeći:

- uspostavlja se stanje baze zabilježeno zadnjom rezervnom kopijom (snimanje s magnetskih traka na disk);
- odredi se zadnja (posebna) kontrolna točka u žurnalu;
- pročita se dio žurnala od početka do zadnje kontrolne točke;
- ponovo se unose u bazu promjene podataka (post-images), i to redom za svaku isporučenu transakciju iz promatranog dijela žurnala.

Nakon ovoga, uspostaviti će se stanje baze koje odgovara zadnjoj (posebnoj) kontrolnoj točki. To je najbolje što možemo učiniti.

6.4 Zaštita od neovlaštenog pristupa

Baza podataka mora biti zaštićena od nedopuštenih ili čak zlonamjernih radnji. Osnovni vid zaštite je fizičko ograničenje pristupa do samog računala. Dalje, ograničava se udaljeni pristup do računala kroz mrežu. No nas ovdje prvenstveno zanima finiji softverski vid zaštite, koji je ugrađen u DBMS. Njime se ljudima koji imaju mogućnost rada na dotičnom računalu ograničavaju mogućnosti rada s bazom.

6.4.1 Identifikacija korisnika

Uobičajeno je da svaki korisnik baze ima svoje **korisničko ime** (username) i **lozinku** (password). Da bi smio raditi s bazom, korisnik se mora predstaviti DBMS-u navođenjem imena, te mora dokazati svoj identitet navođenjem lozinke. DBMS raspolaže popisom korisničkih imena i pripadnih lozinki. Ukoliko korisnik navede ime i lozinku koji ne odgovaraju popisu, DBMS mu ne dopušta rad. Zaštita se zasniva na tajnosti lozinke. DBMS se može osloniti na imena i lozinke koji već postoje u operacijskom sustavu računala, ili se može služiti vlastitim popisom.

6.4.2 Pogledi kao mehanizam zaštite

U Poglavlju 1 spomenuli smo **poglede** (pod-sheme) kao sredstvo za postizavanje logičke neovisnosti podataka. No pogledi ujedno služe i za zaštitu podataka. Naime, DBMS može određenom korisniku pridružiti njegov pogled na bazu. Korisnik tada “vidi” samo dio baze, pa su time bitno ograničene njegove mogućnosti rada.

U relacijskom modelu, i globalna shema i pogled (pod-shema) zadaju se kao skup relacija. Pritom se relacije koje čine pogled izvode iz relacija koje čine globalnu shemu. U SQL-u se relacija-pogled zadaje naredbom **CREATE VIEW**, a izvođenje iz globalnih relacija opisuje se naredbom **SELECT** koja je ugniježdena u **CREATE VIEW**.

U nastavku navodimo primjer globalne sheme i pripadnih pogleda kojima se povjerljivi podaci skrivaju od neovlaštenih korisnika. Cijelu bazu čine dvije relacije, koje predstavljaju zaposlene u poduzeću i njihove odjele.

EMPLOYEE (EMPNO, ENAME, ADDRESS, SALARY, DEPTNO)
DEPARTMENT (DEPTNO, DNAME, MANAGERNO)

Pogled ćemo opisati pomoću SQL naredbi **CREATE VIEW**.

Prvi pogled namijenjen je korisniku koji smije pristupiti podacima o zaposlenima, no ne i njihovim plaćama:

```
CREATE VIEW EMPVIEW1
AS SELECT EMPNO, ENAME, ADDRESS, DEPTNO
FROM EMPLOYEE ;
```

Korisnik vidi samo “vertikalni” segment relacije.

Drugi pogled namijenjen je korisniku koji smije pristupiti cijeloj n-torki o zaposlenima, no samo za zaposlene u odjelu broj 3:

```
CREATE VIEW EMPVIEW2
AS SELECT *
FROM EMPLOYEE
WHERE DEPTNO = 3 ;
```

Korisnik vidi samo “horizontalni” segment relacije.

Treći pogled služi za korisnika-menadžera. On smije pristupiti samo podacima o osobama koje su zaposlene u njegovim odjelima:

```
CREATE VIEW EMPVIEW3
AS SELECT EMPLOYEE.EMPNO, EMPLOYEE.ENAME, EMPLOYEE.SALARY,
DEPARTMENT.DEPTNO, DEPARTMENT.DNAME
FROM EMPLOYEE, DEPARTMENT
WHERE EMPLOYEE.DEPTNO = DEPARTMENT.DEPTNO
AND DEPARTMENT.MANAGERNO = ... (menadžerov identifikacijski broj) ... ;
```

Korisnik vidi relaciju koja zapravo ne postoji u bazi, već se dobiva kombiniranjem podataka iz postojećih relacija.

6.4.3 Ovlaštenja

Pogled određuje kako korisnik “vidi podatke”, no on ne određuje što korisnik može raditi s tim podacima. Osim pogleda, DBMS uz korisnika veže njegova **ovlaštenja** (authorities). Uobičajena ovlaštenja u SQL-u su:

SELECT - ovlaštenje čitanja podataka iz zadane relacije (pogleda);

INSERT - ovlaštenje ubacivanja novih n-torki u zadanu relaciju (pogled);

DELETE - ovlaštenje brisanja n-torki u zadanoj relaciji (pogledu);

UPDATE - ovlaštenje mijenjanja podataka u zadanoj relaciji (pogledu);

ALTER - ovlaštenje mijenjanja građe zadane relacije (na primjer, dodavanje novih atributa);

CONNECT - ovlaštenje korisniku računala da se smije prijaviti za rad s bazom;

DBA - ovlaštenje koje korisniku daje status administratora baze (i povlači sva ostala ovlaštenja).

Neka korisnikova ovlaštenja zadaju se posebno za svaku relaciju (pogled), a neka se zadaju univerzalno. Ovlaštenje **UPDATE** se kod nekih DBMS-a može zadati posebno za svaki atribut. Na primjer, menadžer koji koristi pogled *EMPVIEW3* može imati ovlaštenje **SELECT** za pogled *EMPVIEW3*, te ovlaštenje **UPDATE** za atribut *SALARY* u pogledu *EMPVIEW3*. Također, možemo uskratiti **INSERT** i **DELETE** za isti pogled. Tada bi manager mogao mijenjati plaće svojim suradnicima, no ne bi ih mogao izbacivati iz poduzeća, niti bi mogao dovoditi nove suradnike.

Ukoliko korisnik pokuša obaviti radnju za koju nije ovlašten, DBMS neće izvršiti dotičnu operaciju te će umjesto toga ispisati poruku o greški (povreda ovlaštenja).

U velikom bazama podataka, brigu oko zaštite od neovlaštenog pristupa preuzima administrator baze. On upisuje popis korisnika, zadaje poglede i podešava ovlaštenja. Da bi sve to mogao raditi, sam administrator mora imati najveća moguća ovlaštenja. Neka od posebnih administratorovih ovlaštenja su:

- pravo stvaranja ili brisanja cijelih relacija ili indeksa;
- pravo definiranja pogleda ili brisanja tih pogleda;
- pravo upravljanja fizičkim resursima od kojih se gradi baza (prostor na disku, datoteke);
- pravo davanja i oduzimanja ovlaštenja drugim korisnicima baze.

U nekim DBMS-ima administrator može neke od svojih specifičnih ovlaštenja prenijeti drugima. Tako na primjer administrator može nekom korisniku dati ovlaštenje **UPDATE** za zadanu relaciju, uz dozvolu da taj korisnik to isto ovlaštenje dalje daje drugim korisnicima.

Literatura

- [1] Date C.J.: *An Introduction to Database Systems*, 7th Edition. Addison-Wesley, Reading MA, 1999.
- [2] Korth H.F., Silberschatz A.: *Database System Concepts*, 4th Edition. McGraw-Hill, New York, 2001.
- [3] Ramakrishnan R.: *Database Management Systems*. McGraw-Hill, New York, 1998.
- [4] Abiteboul S., Hull R., Vianu V.: *Foundations of Databases*. Addison-Wesley, Reading MA, 1995.
- [5] Van der Lans R.F.: *Introduction to SQL*. Addison-Wesley, Reading MA, 1999.
- [6] Tharp A.L.: *File Organization and Processing*. John Wiley and Sons, New York, 1988.
- [7] Varga M.: *Baze podataka - konceptualno, logičko i fizičko modeliranje podataka*. DRIP, Zagreb, 1994.
- [8] Widenius M., Axmark D.: *MySQL Reference Manual*. O'Reilly, Sebastopol CA, 2002.