IT350-Teorija

**Potrebe za trajnim čuvanjem podataka:**

* Podaci koji se unesu u informacioni sistem, se mogu upotrebiti u budućnosti; transakcije koje se u sistemu trenutno izvršavaju, se mogu odnositi na podatke koji su bili zapamćeni u prošlosti;
* Podaci koji se trenutno nalaze u operativnoj memoriji nekog računara, nisu dostupni većem broju korisnika. Ukoliko njima treba da pristupi veći broj korisnika, oni moraju biti sačuvani na sekundarnim memorijama računara.

**Kako se podaci trajno čuvaju?**

U datotekama ili bazama podataka

1. **Čuvanje podataka u file-ovima**

Najjednostavniji način za trajno čuvanje podataka

File je podeljen na slogove od kojih svaki predstavlja grupu izvesnog broja polja koji sadrže podatke koje treba zapamtiti

Slogovi u datoteci mogu imati različite forme:

* + **Slogovi** **fiksne dužine:** svaki slog se satoji od više polja, od kojih svako polje ima fiksnu dužinu u bajtovima
  + **Slogovi promenljive dužine**: svaki slog ima maksimalnu i minimalnu dužinu. Dužina svakog sloga može biti zapamćena na početki sloga. Polja mogu biti raždvojena delimiterima.
  + **Zaglavlja i detalji:** svaki slog transakcije se sastoji od sloga zaglavlja koji je praćen izvesnim brojem slogova detalja.
  + **Tagovani podaci:** korist se za podatke u file-ovima koji koriste HTML i XML.

**Organizacija file-ova**

Postoje tri načina na koji file-ovi mogu biti organizovani:

* + ***Serijska organizacija:*** Svaki sledeći slog u file-u se upisuje na kraj predhodnog sloga.
  + ***Sekvencijalna organizacija:*** svaki slog se upisuje po nekom predefinisanom redosledu, dobijenog na osnovu jednog polja u slogu (ključa). Novi slog se upisuje, tako što se file kopira do tačke gde slog treba insertovati, novi slog se insertuje a tada se kopira ostatak file-a.
  + ***Random organizacija:*** Slogovima se pristupa direktno, bez prolaska kroz ostatak file-a, korišćenjem algoritma koji konvertuje polje ključa svakog sloga u adresu file-a kojoj se može direktno pristupiti.

**Načini za pristup podacima u file-ovima**

Zavisno od izabrane organizacije file-ova, postoje tri načina za pristup podacima u file-ovima:

* 1. **Serijski pristup:**

Da bi se pronašao određeni slog, čita se slog po slog file-a, sve dok se ne locira određeni slog. Serijski se može pristupiti serijskim i sekvencijalnim file-ovima.

* 1. **Indeks-sekvencijalni pristup**

Pronalaženje sloga se vrši sekvencijalnim pretraživanjem područja indeksa, dok se ne nađe traženi ključ, a zatim se na osnovu adrese koja odgovara nađenom ključu brzo pronalazi traženi slog. Koristi se kada je neophodno da se file čita sekvencijalno,slog po slog i korišćenjem ključa ode pravo do određenog sloga.

* 1. **Direktan pristup**

Zasniva se na korišćenju algoritama za konverziju vrednosti polja ključa u adresu sloga u file-u. Najjednostavnija od njih je relativna adresa. Ovaj metod pristupa zahteva korišćenje slogova fiksne dužine i pozitivne sukcesivne vrednosti ključeva. Ako je svaki slog dugačak 200 bajtova, tada će prvi slog početi na prvom bajtu, drugi na bajtu 201, treći na bajtu 301 itd.

* 1. **Hashing adresiranje**

Da bi se odredo bloku kojem se čuva slog, vrši se heširanje ključa (algoritam koji ASCII string konvertuje u intedžer).

Primer: uzeti karaktere u stringu i konvertovati ih u njihove ASCII vrednosti (na primer A je 65). Te ASCII vrednosti se sumiraju i suma podeli sa brojem blokova u file-u. Ostatak deljenja daje broj bloka u file-u u kojem treba staviti slog, počevši od bloka 0. Ako je taj blok pun, koristi se dodadtni blok kao overflow a njegova adresa se čuva u bloku u kojem je slog trebalo da bude zapamćen.

* **Nedostaci čuvanja podataka file-ovima:**

1. Redudantnost podataka: različiti file-ova mogu da sadrže iste podatake za različite aplikacije u različitim formama
2. Ažuriranje podataka u različitim aplikacijama neće biti sinhronizovano.
3. Ukoliko se podaci promene ili se proemi način na koji su oni zapamćeni, aplikacije koje pristupaju tim podacima moraju biti prilagođene izmenama.
4. Pravljenje izveštaja u kojem se kombinuju podaci iz različitih aplikacija se ne može implementirati bez značajnih napora programera.

**Čuvanje podataka u bazama podataka**

Baze podataka su nastale u pokušaju da se reše problemi sa čuvanjem podataka u fajl sistemima. Problem se može rešiti kroz više koraka:

* + Prvi korak: napraviti korporativnu bazu podataka koja će sadržati podatake iz različitih aplikacija. Svaka aplikacija tada za zadovoljenje svojih zahteva koristi podskup podataka iz zajedničke baze.
  + Drugi korak je iskoristiti sistem za upravljanje bazom podataka tako da se mehanizmi za pamćenje podataka razdvoje od aplikativnih programa

**! Mehanizam kojim se pamćenje podataka razdvaja od aplikativnih programa**

Danas je najčešće u upotrebi Tri-šeme arhitektura:

1. Eksterna šema predstavlja način na koji se podaci koriste u aplikativnim programima.
2. Konceptualna šema je logički model podataka i nezavisna je i od eksterne šeme i od detalja o tome kako se podaci pamte.
3. Fizička organizacija podataka se nalazi u internoj šemi, kojom se definišu file-ovi za pamćenje podataka.

Cilj je izolovati aplikativne programe od načina na koji se podaci pamte!!!

**Interna šema baze podataka-**Strukture podataka će u internoj šemi biti predstavljene korišćenjem file-ova. Značajna karakteristika DBMS-a je da vrste file-ova koje se koriste, metodi pristupa i indeksi koje koriste file-ovi su u DBMS sakriveni od programera i mogu se menjati bez uticaja na programe koji ih koriste.

**+ DBMS (Data base management system)**

* DBMS pored načina za pamćenje podataka obezbeđuje alate koji se mogu koristiti za upravljanje podacima.
* Data definition laguage (DDL): koristi se za specifikaciju podataka koji se drže u DBMS
* Data manipulation language (DML): se koristi da bi se ažurirali i pretraživali podaci u DBMS.
* Ograničenja integriteta: se specificiraju da bi se održao integritet podataka
* Upravljanje transakcijama: u okviru transakcije ažuriranja, se moraju izvršiti sve njene komponente
* Konkurentnost: Više korisnika može da simultano koristi bazu podataka i ažurira njen sadržaj.
* Zaštita: pristup podacima u BP i dozvole koje su dodeljene različitim korisnicima za različite nivoe pristupa
* **Nedostaci korišćenja DBMS-a**
* Za investiranje u velike DBMS sisteme je potrebno mnogo novca
* Zapošljavanje osoblja koje će upravljati DBMS-om je takođe povezano s mnogo troškova
* Postupci konverzije podataka iz baze podataka u formate koje zahtevaju aplikativni programi mogu biti složeni

**Tipovi baza podataka:**

* + 1. Relacione
    2. Objektne
    3. Relaciono-objektne

**Relaciona baza podataka**

Skup međusobno povezanih podataka uskladištenih u spoljnoj memoriji računara uz minimalnu redudansu, koja dozvoljava njihovo korišćenje od strane jedne ili više aplikacija.

Podaci su nezavisni od programa koji ih koriste u smislu dodavanja novih ili modifikacije postojećih podataka kao i radi pretraživanja podataka.

Podaci su istovremeno dostupni raznim korisnicima i aplikativnim programima.

Korisnici i aplikacije ne moraju poznavati detalje fizičkog prikaza podataka, već se referenciraju na logičku strukturu baze.

**Struktura relacione baze podataka**

Relacionu bazu podataka sačinjavaju podaci sa kojim radimo, smešteni u jednu ili više tabela.

Svaka tabela se sastoji od redova a svaki red sadrži vrednosti atributa koje su organizovane u kolone. Jedna veoma važna karakteristika baza podataka je to da se u njima ne pamte samo podaci već i relacije među podacima.

**Metapodaci u relacion bazama podataka**

Pored tabela korisničkih podataka, relaciona BP sadrže i tabele podataka koje opisuju te korisnike podatke (metapodatke). Format i forma metapodataka se razlikuje zavisno od DBMS-a.

Relacioni DBMS su trenutno najšire korišćeni DBMS-ovi!!!

Najpoznatiji je svakako Access a zatim Oracle, SQL-Server, DB2, Informix, Ingres, Sybase, MySQL itd...

**Komponenete jednostavnih RDBM sistema (Access)**

Access ima tri komponenete:

* + Aplikacije:programi sa kojima korisnici imaju direktnu interakciju
  + SQL: standardni jezik kojeg razumeju svi komercijalni RDBMS
  + DBMS:veliki i komplikovan program koji je licenciran od strane proizvođača kao što su Microsoft, Oracle i IBM.

Access sadrži DBMS (administrira BP) i generatore formi, izveštaja i upita!

Izdvajaju se dve različite vrste mogućih komandi DBMS-a:

* Obični korisnici i aplikativni programi (eng. User/Application): koji samo pristupaju ili menjaju podatke u bazi podataka
* Administratori baze podataka (eng. Database administrator): osobe koje su odgovorne za strukturu i šemu baze podataka.

DDL komande predstavljaju komande koje su jednostavnije za obradu. *Primer: administrator baze podataka DDL komandama za bazu podataka univerziteta kreira tabele za studente, smerove i kurseve itd.*

**Obrada upita i ažuriranja**

1. Korisnik ili aplikativni program pokreće neku akciju, koristeći jezik za upravljanje podacima (engl. data-manipulation language - DML). Ovakve komande ne menjaju strukturu baze podataka već samo menjaju sadržaj podataka unutar same baze
2. DML komande se izvršavaju na sledeći način:
3. Upit se prvo parsira i optimizuje u kompajleru upita.
4. Rezultat ovoga se predaje execution engine-u.
5. Execution engine izdaje niz zahteva menadžeru skladišta koji zahtev za dostavljanje podataka predaje menadžeru bafera.
6. Zadatak menadžera bafera je da dostavi podatke iz sekundarnog skladišta gde su oni trajno sačuvani

**Obrada transakcija**

Upiti i ostale DML akcije su grupisane u transakcije, koje moraju da budu izvršene atomično i izolovano jedna od druge.

Svaka transakcija mora da bude trajna što znači da odmah nakon izvršenja transakcije posledice izvršenja transakcije moraju da budu vidljive.

U bazama podataka podaci se uglavnom skladište u sekundarno skladište (hard disk) a da bi se izvršila neka izmena podataka, podaci moraju biti smešteni u glavnu memoriju.

Posao menadžera skladišta je da upravlja podacima između glavne memorije i sekundarnog skladišta!!!

U praksi je uobičajeno da se nekoliko operacija nad bazom podataka grupišu u transakciju, koja predstavlja jedinicu posla koja mora biti izvršena atomično i potpuno izolovana od drugih transakcija.

Menadžer transakcija dobija transakcione komande od aplikativnih programa.

Uloga procesora transakcija je da vrši sledeće operacije:

* Logovanje (svaka promena koja je nastala nad bazom podataka se beleži u poseban fajl)
* Kontrola paralelnog rada: u mnogim sistemima više transakcija izvršavati istovremeno.
* Blokada transakcija: kako se transakcije međusobno nadmeću za resurse kako bi bile izvršene, može se doći u situaciju da svakoj treba neki resurs koji druga transakcija trenutno drži zaključanim

**Objektne baze podataka**

Nasuprot relacijama ili tabelama, objekti u OOBP nisu ravne, dvo dimenzionalne strukture.

Svaki objekat može sadržati druge objekte!!!

Nastale su da bi upravljali objektima složene strukture!

Tu spadaju:

* + multimedijalne aplikacije u kojima objekte kao što su zvuk, slike i video klipovi nije lako predstaviti tabelama
  + aplikacije kao što su CAD sistemi u kojima dizajneri mogu imati potrebu da rade s različitim nivoima apstrakcije.

Za razliku od relacionih modela, u svetu objektno-orijentisanih baza podataka ne postoji univerzalno prihvaćeni dogovor o modelima podataka!!!

**-Nedostaci OO DBMS-ova**

* Nepostojanje univerzalnog modela podataka.
* Nedostatak iskustva – korišćenje OO DBMS-ova je još uvek u povoju
* Nedostatak standarda – ne postoje standardi za OO upitne jezike.
* Složenost – uvođenjem dodatnih funkcionalnosti, povećala se kompleksnost pa su aplikacije skuplje i teže za korišćenje
* Nedostatak podrške za bezbednost – nema ugrađenih mehanizama za poglede i bezbednost pristupa podacima.

**Objektno-relacione baze podataka(ORDBMS)**

Objektno-relacioni model i objektno-relacione baze podataka reprezentuju takozvani

evolucijski pristup koji se ogleda u integraciji objektne paradigme u relacioni model podataka

i njegovim proširenjem objektno-orjentisanim karakteristikama. Objektno-relacione baze se

mogu nazvati i hibridne baze.

Objektno-relacioni sistem za upravljanje bazom podataka(ORDBMS) je sličan sistemu za upravljanje relacionim bazama podataka ali se bazira na objektno-orjentisanom modelu baze podataka: objekti, klase i nasleđivanje su direktno podržane u šemama baza podataka i upitnim jezicima.

Osnovna postavka objektno-relacionih baza podataka je upravljanje objektima i pravilima

uz očuvanje kompatibilnosti sa relacionim modelom i relacionim bazama podataka.

SQL standardom iz 2003.ostvareno je potpuno slaganje nad objektno-relacionim modelom i svi glavni proizvođači sistema za upravljanje bazama nastoje u što je moguće većem obimu

da implementiraju taj standard.

+Tradicionalni relacioni model se proširuje osnovnim objektnim konceptima kao što su:

apstraktni tipovi podataka, enkapsulacija, polimorfizam, nasleđivanje i sl.

Osnovni koncepti su:

1. **Korisnički definisani tipovi podataka** - omogućavaju kreiranje aplikacijskih

specifičnih, kompleksnih tipova podataka. Prekoračuje se jedno od osnovnih

ograničenja relacionog modela koji podrazumeva i zahteva da domeni atributa

relacije budu prosti. Korisnički definisani tip utemeljen je na konceptu apstraktnih

tipova podataka.

2. **Relacije** – su i dalje jedan od osnovnih koncepata objektno-relacionog modela.

Koncept relacije je proširen dopuštanjem da atributi kao domene ne moraju imati

samo proste, nego i korisničke tipove podataka. Ukoliko su elementi relacije primeri

korisničkog tipa podataka ona se naziva objektna relacija.

3. **Korisnički tipovi podataka** mogu biti, kao i standardni jednostavni tipovi, domeni

atributa standardnih relacija. Ne postoji formalna definicija objektnog modela (ODMG

specifikacija). Objektno-relacioni model bazira se na konceptima objektne paradigme.

Objektni aspekti su:

1. **Apstraktni tipovi podataka** - može se definisati klasa podataka koja se sastoji od

atributa i operacija uz skrivanje detalja njihove implementacije

2. **Nasleđivanje** - mogućnost da više tipova podataka dele atribute i operacije

3. **Identitet objekata** - svaki primerak abstraktnog tipa podataka poseduje jedinstveni

ID, nevidljiv korisnicima

4. **Polimorfizam** - egzistencija više operacija sa istim imenom, a sa različitom

semantikom tj. mogućnost rada nad različitim tipovima podataka.

**RELACIONI MODEL**

Velika većina informacionih sistema danas koristi ***relacione modele baza podataka***;

Relacija u relacionom modelu je isto što i relacija u matematici s tom razlikom da su relacije u relacionom modelu podataka vremenski promenljive.

**Pojam funkcije**

***f*: E → F**

* Funkcija ***f*** **elementu x** ∈ **E pridružuje element *f(x)* iz F;**

***Primer:*** Svaki učenik ima svoje prezime. Sa **E** možemo označiti skup svih učenika nekog razreda a sa **F** prezimena tih učenika. Između elemenata skupa **E** i skupa **F** postoji potpuno određen odnos, jer svakom elementu **x ∈ E**, tj. svakom učeniku pripada potpuno određen element **y** iz skupa **F** (tj. prezime učenika **x**).

Definicija funkcije uključuje tri objekta:

* + skup **E** (domen)
  + skup **F** (kodomen) i
  + postupak (propis, radnju, transformaciju) prema kojem je svakom elementu **x** iz **E** pridružen jedinstven element **f(x)** iz **F**.

**Primer:** Ako stanemo ispred nekog izloga, onda svi predmeti u tom izlogu obrazuju skup **E**, a njihove cene skup **F**;

Svakom predmetu **x ∈ E** pridružen je po jedan element iz **F**, tj. njegova cena. Tu se radi o funkciji **f** koja artiklu **x ∈ E** pridružuje njegovu cenu **f(x)**;

**Dekartov proizvod skupova**

Neka su **A** i **B** dva skupa i **a ∈ A** i **b ∈ B** zadati elementi;

***(a, b)*** je uredjen par, ako je element ***a*** proglašen prvim, a ***b*** drugim u tome paru;

Dva uređena para ***(a,b)*** i ***(a′, b′)*** su jednaka ako i samo ako je:

***a=a***′ i ***b=b′*** što pišemo kao ***(a, b) = (a′, b′ )***

element ***a*** uredjenog para ***(a,b)*** se zove prva projekcija a ***b*** druga projekcija;

Za proizvoljne neprazne skupove ***S′*** i ***S′′*** , skup svih uređenih parova ***(s′, s′′)***, označavamo sa:

**S = S′ x S′′ = {(s′, s′′) : s′ ∈ S′, s′′ ∈ S′′}**

**Skup *S* se zove Dekartov proizvod skupova *S′* i *S′′*.**

**Pojam relacije**

***Definicija:*** Neka su **E** i **F** skupovi. Svaki podskup **ρ** Dekartovog proizvoda **E x F** zove se **relacija**;

Za element **x ∈ E** kažemo da je u relaciji **ρ** sa **y ∈F**, ako i samo ako je **(x, y) ∈ ρ**;

* Činjenicu da **(x, y) ∈ρ** često pišemo u obliku

**x ρ y** ili  **y= ρ (x)**

i kažemo da **x** ima *osobinu* da je u relaciji **ρ** sa **y**;

Prva projekcija skupa **ρ** zove se područje definicije ili domena relacije **ρ**, a druga projekcija skupa **ρ** zove se slika relacije **ρ**.

Pojam i definicija relacije iz matematike iskorišćeni su za definiciju i razvoj relacionog modela (baze) podataka;

Polazna osnova za definisanje relacije u relacionom modelu podataka je da je svaka relacija opisana skupom atributa Ai.

Svaki atribut Ai opisuje neku odabranu osobinu klase entiteta koji relacija opisuje.

Svakom atributu je pridružen domen, koji predstavlja specifikaciju skupa mogućih vrednosti iz kojeg atribut uzima vrednosti. Uobičajeni način specifikacije domena je preko tipa podataka.

Polazeći od ovih pretpostavki, relacija u relacionom modelu se može definisati kao skup parova atributa relacije i imena domena.

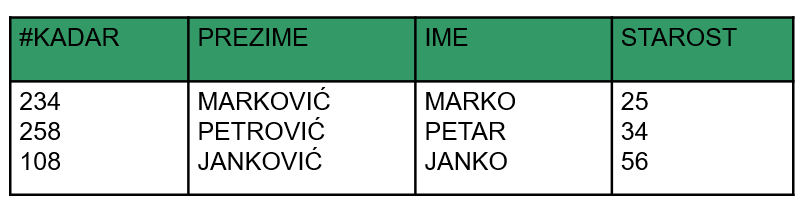
Relacija R je skup preslikavanja imena atributa u odgovarajuće domene primenom odgovarajuće funkcije.

Drugim rečima, relacija R je skup n-torki (A1:d1, A2:d2, . . . , An:dn) tako da je d1 ∈ D1, d2 ∈ D2, . . . , dn ∈ Dn

Svaki element n-torke se sastoji od atributa i vrednosti za taj atribut.

**Relacija kao imenovana tabela**

Sa aspekta korisnika, relacije je imenovana tabela;

Relacija **KADAR (#kadar, prezime, ime, starost)** zapisana u obliku tabele je:

Imenovane kolone tabele su atributi!

Redovi tabele (n-torke relacije) su pojedinačna pojavljivanja tipa entiteta!

**Šema relacije**

Šema relacije se sastoji od:

* + naziva šeme relacije,
  + skupa atributa relacije i
  + skupa ograničenja integriteta relacije

Šema relacije opisuje semantiku relacije na prirodnom jeziku.

Uobičajeno je da se šema relacije obeležava tako što se navede njen naziv, a nakon toga skup atributa.

**Ograničenja relacionog modela**

**1. Svi elementi skupa su različiti**

Iz ovog proizilazi definicija **ključa relacije:** to jepodskup atributa čije vrednosti jedinstveno identifikuju **n**-torke relacije;

Moguće je da relacija ima više različitih ključeva. Svi ključevi čine skup kandidata za ključ;

Ako neki skup atributa u relaciji nije ključ ali je ključ u nekoj drugoj relaciji, onda se on naziva **strani ključ.**

Da bi neki skup atributa relacije bio kandidat za ključ on mora zadovoljiti dva uslova:

* 1. **Uslov jedinstvenosti:** Vrednost ključa svake **n**-torke relacije jedinstveno određuje **n**-torku, tj. ne postoje dva reda u tabeli takva da imaju sve iste vrednosti svih atributa koje čine ključ.
  2. **Uslov ne redudantnosti**: Ne postoji ni jedan atribut kao deo ključa koji se može izostaviti iz ključa a da se pri tome uslov jedinstvenosti ne gubi, tj. ključ je unija minimalnog broja atributa.

Od niza kandidata za ključ bira se jedan **primarni ključ**!

Za **primarni ključ** su pored navedenih definisana i dodatna ograničenja:

**Integritet entiteta**. Neka je atribut **A** deo primarnog ključa relacije **R**. Tada atribut **A** ne sme poprimiti nul vrednost. To znači da ne postoji **n**-torka u relaciji **R** takva da je vrednost atributa **A** te **n**-torke jednaka nul vrednosti

**Referencijalni integritet**

Neka postoji *relacija* ***S*** *sa prostim primarnim ključem* ***A*** *definisanim nad domenom* ***D***.

Neka postoji *relacija* ***R*** *sa atributom* ***A*** *definisanim nad* ***D***.

Tada vrednost atributa **A** bilo koje **n**-torke u relaciji **R** može biti ili nul vrednost ili vrednost "**k**", pri čemu vrednost "**k**" postoji u domenu **D** i postoji **n**-torka u relaciji **S** takva da je vrednost ključnog atributa **A** te relacije jednaka vrednosti "**k**".

Relacije **R** i **S** ne moraju biti različite.

Atribut **A** relacije **R**, koji postoji kao primarni ključ u relaciji **S**, naziva se spoljni (strani) ključ.

**Osnove relacionog modela-primer**

Relacioni model pruža jedinstven način za predstavljanje podataka kao dvo dimenzionih tabela koje se zovu **relacije**. Kao primer relacija Movies.

Kolone relacije se nazivaju **atributi**.

Ime relacije i skup atributa relacije se naziva **šema relacije** (npr. Šema za relaciju ***Movies (title , year, length , genre)***

**Relaciona algebra**

Operacije tradicionalne relacione algebre se mogu razvrstati u četiri klase:

* 1. podrazumevani set operacija - **unija i razlika**
  2. operacije za izveštavanje koje izveštavanje baziraju na uklanjaju delova relacija kod kreiranja rezultata: **selekcija** eliminiše neke redove koji ne ispunjavaju tražene uslove kod izveštavanja, i **projekcija** eliminiše neke kolone!
  3. operacije koje kombinuju zapise, uključujući **Dekartov proizvod**, i razne vrste **join** operacija, koje selektivno uparuju zapise iz dve relacije.
  4. operacije preimenovanja koje ne utiču na zapise iz relacija, ali menjaju šemu relacije tako što menjaju imena atributa ili ime samog odnosa

**UNIJA** - operacija nad dve relacije koja omogućava dodavanje i upis različitih n-torki jedne relacije drugoj relaciji;

**RAZLIKA** - operacija nad dve relacije koja omogućava izbacivanje istih n-torki u obe relacije.

* + Ova operacija nije komutativna, te je poredak operanada važan.
  + Rezultat su n-torke iz prve navedene relacije.

**PROJEKCIJA** - operacija nad jednom relacijom koja omogućava izdvajanje skupa atributa jedne relacije u novu relaciju pri čemu se eliminšu duple n-torke.

**Primer:** Ako je data relacija R1(A,B,C,D,E) tada je relacija R2(A,D) rezultat projekcije relacije R1 po atributima A i D ako je:

R2 (A,D) ═ PROJEKCIJA (A,D) od R1 (A,B,C,D,E)

**SELEKCIJA** - operacija nad jednom relacijom kojom se izdvaja skup n-torki jedne relacije u novu. Izdvajaju se samo one n-torke koje zadovoljavaju uslov uz operator selekcije.

**Primer:** Neka je data relacija R1 (A,B,C) i uslov B>15.

Relacija R2 (A,B,C) je rezultat relacije R1 po uslovu B>15 ako je:

R2 (A,B,C) = R1[ B>15] (A,B,C)

**DEKARTOV PROIZVOD:** operator kojim se dobija *nova relacija sa svim atributima obe relacije, tako što se sve n-torke jedne relacije spajaju sa svim n-torkama druge relacije*. Oznaka za operator je "**X**„

**PRESEK:** operator koji za rezultat daje *relaciju koja sadrži samo one n-torke koje se istovremeno nalaze u obije relacije*. Oznaka za operator je "**∩**„

**THETA JOIN**: operator spajanja dve relacije kojim se dobija *nova relacija a n-torke nastaju konkateniranjem n-torki relacija*.

***Na primer:*** *Relacije R1 (A,B,C) i R2 (D,E,F) se spajaju u relaciju*

*R3 (A,B,C,D,E,F)*

U zavisnosti od odabranog aritmetičkog operatora poređenja theta

{ =, ≠, <, ≤, ≥, > } join se naziva:

EQUI JOIN, LESS-THEN JOIN, itd.

Kod **EQUI JOIN** se zahteva da su vrednosti atributa iz dve relacije jednake, pa je sadržaj dve kolone identičan

**NATURAL JOIN**isključuje jednu od redudatnih kolona

**Koncepualno modeliranje**

S ciljem da se definiše struktura podataka relacione baze podataka koja se naziva konceptualnim modelom podataka

To je pojednostavljeno predstavljanje realnog sistema preko skupa entiteta, veza između entiteta i atributa entiteta.

Konceptualni model podataka se predstavlja korišćenjem grafičkog jezik za predstavljanje strukture podataka.

Razvijen je veliki broj alata i tehnika za izradu model podataka. Od njih se najduže u upotrebi zadržala tehnika Entity-Relationship (E/R) dijagrami.

**E/R dijagrami**

E/R dijagram je obično rezultat sistematske analize koja se vrši s ciljem da se definiše i opiše ono što je važno za procese u oblasti poslovanja.

Obično se crta korišćenjem pravougaonika koji predstavljaju tipove entiteta a koji su povezane linijama (relacijama) koje izražavaju asocijacije i zavisnosti između tipova entiteta.

Tipovi entiteta se mogu karakterisati ne samo relacijama, već i dodatnim osobinama (atributima), koji uključuju i identifikatore entiteta.

Kao softverska podrška za izradu konceptualnih modela korišćenjem E/R dijagrama koriste se CASE (Computer AidedSoftware Engineering) alati.

E/R model podataka je nezavistan od bilo kojeg sistema za upravljanje bazama podataka (DBMS) kao i hardverske platforme na kojoj će se BP implementirati

Osnovni koncepti E/R modela su:

* + - **Tip entiteta**
    - **Atributi**
    - **Relacije**

**Tip entiteta** predstavljaskup entiteta iz stvarnog, realnog sveta koji imaju ista svojstva.

Svaki entitet u realnom svetu poseduje nezavisnu egzistenciju i može biti:

* + objekat sa fizičkim (stvarnim) postojanjem (osoba, automobil, kuća ili zaposleni) ili
  + objekat sa konceptualnim (apstraktnim) postojanjem (posao, predmet koji se sluša na fakultetu, utakmica itd.)

Jedan **entitet** je jedan pojavni oblik datog tipa entiteta. Svaki entitet mora imati identitet različit od svih ostalih entiteta.

**Atributi**

Predstavljaju svojstvo tipa entiteta (ili tipa relacije).

Sadrže vrednosti koje opisuju svaki entitet (najveći deo podataka smeštenih u BP su vrednosti atributa)

Svaki atribut može imati jednu od skupa mogućih vrednosti koji se naziva domen.

**Prost atribut** se sastoji od jedne komponente koja ima nezavisnu egzistenciju. Ne mogu dalje deliti. ***Primer-****Pol*

**Složen atribut** se sastoji od više komponenti od kojih svaka ima nezavisnu egzistenciju. Mogu se dalje deliti na podkomponente. ***Primer-****Adresa*

Raščlanjivanje složenog atributa na proste komponente se vrši u zavisnosti od toga kako korisnik vidi atribut, kao jedinstvenu celinu ili kao pojedinačni delovi.

**Atribut sa jednom vrednošću:** sadrži jednu vrednost za jedan entitet (većina atributa ima jednu vrednost)

**Atribut sa više vrednosti** sadrži više vrednosti za jedan entitet. ***Primer-telefon(***mobilni ili fiksni)

**Izvedeni atribut** predstavlja vrednost koja je izvedena na osnovu vrednosti nekog povezanog atributa ili skupa atributa, pri čemu oni ne moraju pripadati istom entitetu.***Primer-***broj godina na osnovu datuma rođenja

**Relacija (veza)**

Predstavlja asocijacije koje se mogu uspostaviti izmedju tipova entiteta;

Svakoj relaciji se zadaje naziv koji opisuje njenu funkciju.

Predstavljaju ograničenja koja mogu da postoje u asocijacijama izmedju tipa entiteta (u skladu sa ograničenjima koja postoje u realnom svetu)

**Maksimalna kardinalnost** predstavlja maksimalni broj instanci tipova entiteta koji učestvuju u relaciji

**Minimalna kardinalnost** predstavlja minimalni broj instanci tipova entiteta koji učestvuju u relaciji

Tri najčešće ***maksimalne kardinalnosti*** su tipa:

* + **jedan-prema-jedan**
  + **jedan-prema-više**
  + **više-prema-više**

**Jedan-prema-jedan (1:1)**

Jedna entitet prvog tipa entiteta se odnosi na maksimalno jedan entitet drugog tipa entiteta

Jedna instanca entiteta drugog tipa se odnosi na maksimalno jednu instancu entiteta prvog tipa. **Primer-**zaposleni može da upravlja samo jednim odeljenjem i odeljenje može imati samo jednog menadžera koji njime upravlja

**Jedan-prema-više (1:M)**

Jedna instanca entiteta prvog tipa se odnosi na više instanci entiteta drugog tipa i

Jedna instanca entiteta drugog tipa se odnosi na maksimalno jednu instancu entiteta prvog tipa. **Primer-** Jedan smer na fakultetu ima više studenata a jedan studen je na jednom smeru

**Više-prema-više (M:N)**

Jedna instanca entiteta prvog tipa se odnosi na više instanci entiteta drugog tipa i

Jedna instanca entiteta drugog tipa se odnosi na više instanci entiteta prvog tipa

**Primer-** Oglas za nekretnine može da bude postavljen u više novina, a i više novina može da reklamira nekretnine

**Minimalna kardinalnost**

Odredjuje broj instanci entiteta koji mora učestvovati u relaciji;

Generalno, minimum je odredjen sa nula ili jedan:

* + ako je nula, učešće u relaciji je neobavezna (opciono)
  + ako je jedan, učešće u relaciji je obavezno!!!

**Stepen relacije**

Označava broj učesnika (tipova entiteta) u relaciji

**Binarna relacija** – broj učesnika u relaciji je dva.

**Ternarna relacija -** broj učesnika u relaciji je tri.

**Uloga (Role)**

Svaki tip entiteta koji učestvuje u relaciji ima posebnu ulogu u toj relaciji.

Ime uloge (engl. role) označava ulogu koju entitet iz datog tipa entiteta igra u relaciji i pomaže da se objasni šta znači relacija.

**Rekurzivna (unarna) relacija**

Relacija u kojoj jedan isti tip entiteta učestvuje više puta sa različitim ulogama;

Tip entiteta *Zaposlen* dva puta učestvuje u vezi:

* + prvi put kao rukovodilac,
  + drugi put kao zaposleni kojim se rukovodi.

**Pravila pri modelovanju**

Četiri osnovna poslovna pravila se odnose na:

* + **Integritet entiteta:** Svaka instanca jednog tipa entiteta mora da ima jedinistveni identifikator koji nije null!!!
  + **Ograničenja referencijalnog integriteta:** Pravila se odnose na veze izmedju tipova entiteta
  + **Domene:** Ograničenja na validne vrednosti atributa
  + **Trigere:** Poslovna pravila koja štite validnost vrednosti atribut

**Domen**je skup svih tipova podataka i opsega vrednosti koje mogu da imaju atributi

Definicije domena obično specificiraju neke (ili sve) karakteristike atributa: tip podataka, dužina, format, opseg, dozvoljene vrednosti, značenje, jedinstvenost i null vrednosti.

**Trigeri**

Trigeri su pravila pod kojim se izvršavaju operacije za manipulaciju podacima kao što su: insert, update i delete.

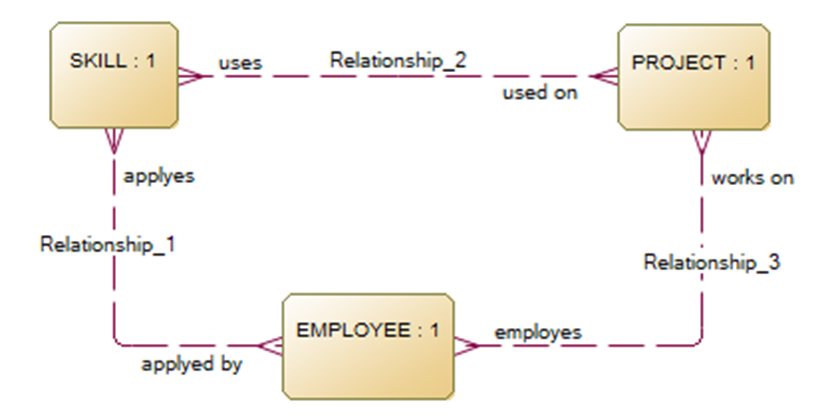
Operacije za izvršenje trigera obično imaju sledeće komponente:

* + Poslovno pravilo: koncizna rečenica kojom se opisuje poslovno pravilo koje treba da bude izvršeno operacijom
  + Događaj: operacija koja se inicira (insert, delete ili update)
  + Ime tipa entiteta kojem se pristupa ili se modifikuje
  + Uslov: pod kojim se operacija izvršava

Akcija: koja se izvršava kada se operacija trigeruje

+++ Napredni E/R dijagrami podrazumevaju korišćenje složenih relacija kako što su ternarna, rekurzivna, asocijativna relacija, specijalizacija.

**Ternarna relacija** je relacija čiji je stepen relacije tri.



Na slici je dat primer na koji se može primeniti ternarna relacija: potrebno je da znamo koje veštine (SKILL) zaposleni (EMPLOYEE) koriste na kojim projektima (PROJECT).

1. Uloga primenjuje (*applies*) ukazuje na to koji zaposleni ima koje veštine.
2. Uloga korišćena na (*used on*) ukazuje koja se veština koristi na kom projektu.
3. Uloga radi na (*works on*) ukazuje koji  
   zaposleni radi na kom projektu.

!!! Ternarni odnos ne isključuje potrebu za binarnim.

Binarni odnosi su suvišni samo kada odražavaju informaciju koja je podskup informacije koju sadrži ternarni odnos.

Ako binarni odnos sadrži informaciju koja se razlikuje od one sadržane u ternarnom, binarni odnosi se zadržavaju.

Ternarna relacija-određivanje kardinalnosti

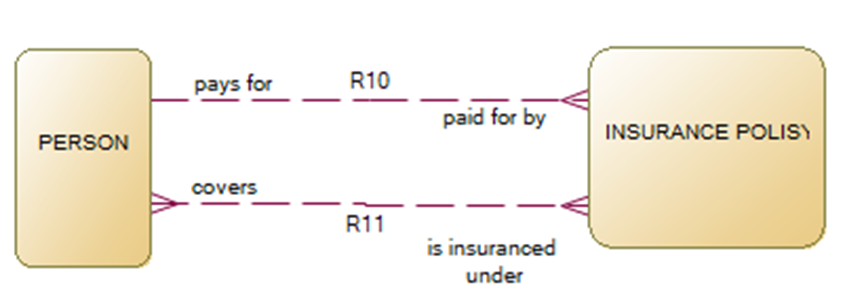
Za preostali tip entiteta, zapitati se: da li može da postoji najviše jedan entitet tog tipa entiteta, za svaku pojedinačnu kombinaciju preostalih entiteta, ili ih može biti više?(U primeru, zapitati se: da li neki zaposleni na nekom projektu može imati najviše jednu veštinu koju primenjuje na projektu, ili može primenjivati veći broj veština?)

* + Ako je odgovor “više”, staviti simbol rakljice na liniju koja ulazi u ternarnu relaciju;
  + ako je odgovor “1”, staviti crticu preko te linije.

Ponoviti proces dok se ne ispita svaki tip entiteta u relaciji.

**Višestruke relacije**

Višestruka relacije se javljaju u slučaju kada dva entiteta grade više od jednog tipa relacije.

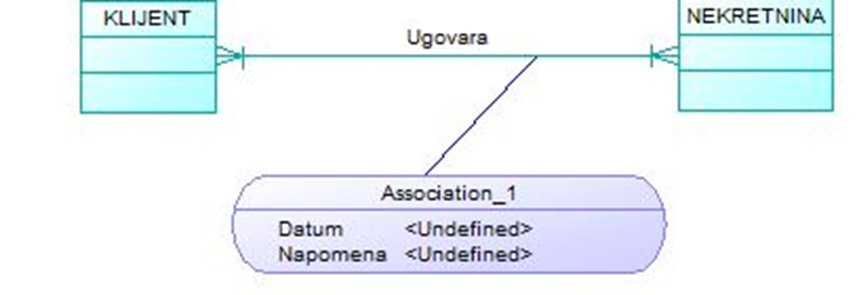


Posmatrajmo tipove entiteta PERSOM (osoba) i INSURANCE POLISY (polisa osiguranja) i relacije između njih:

1. ***pays for (uplaćuje****) -* Jedna osoba uplaćuje nula ili više polisa osiguranja. Jednu polisu  
   osiguranja uplaćuje tačno jedna osoba.
2. ***is insured under (osiguran je po )-***Jedna osoba je osigurana po osnovu nula ili više polisa osiguranja. Jedna polisa osiguranja pokriva jednu ili više osoba.

**Asocijativni tip entiteta**

Asocijativni entitet je specijalan slučaj relacije sa maksimalnom kardinalnošću “više prema više” koji zahteva da neki novo kreirani tip entiteta pamti atribute te relacije.



Primer: na slici je prikazana relacija između tipova entiteta KLIJENT i NEKRETNINA. Atribut Datum će se koristiti da bi se zabeležio datum kada je klijent razmatrao nekretninu, a atribut Komentar će se koristiti da bi se zapamtio svaki komentar koji je klijent imao u vezi sa nekretninom.

Prvo treba nacrtati jednostavniji dijagrami (sa relacijama sa maksimalnom kardinalnošću više-prema-više ) a dijagram unaprediti dodavanjem asocijativnih  
tipova entiteta koji ih zamenjuju!!!

**Specijalizacija**

Specijalizacija je proces maksimiziranja razlika između entiteta nekog tipa entiteta identifikovanjem karakteristika koje ih razlikuju.

Ovaj proces predstavlja prilaz odozgo-nadole, definisanjem super klase i njegovih pod klasa

Pod klase se definišu na osnovu njihovih specifičnih karakteristika (atributa) u odnosu na nad klase. Primer: Super klasa ZAPOSLENI poseduje pod tipove entiteta MANAGER, SEKRETAR i PRODAVAC.

Relacija super klasa i pod klasa donekle podseća na relaciju 1: 1! Glavna razlika je u tome što su u odnosu 1: 1 povezana dva različita entiteta, dok se ovde radi o istim entitetima.

**Generalizacija**

Proces minimizacije razlika između entiteta identifikovanjem zajedničkih karakteristika.

*Primer:tipovi entiteta CAR i TRUCK*

Pošto imaju nekoliko zajedničkih atributa, oni se mogu generalizovati u tip entiteta VEHICLE.

Nadklase i podklasese se mogu koristiti da bi se izbeglo opisivanje sličnih tipova entiteta korišćenjem različitih atributa u jednom istom tipu entitetu (tada se može javiti veliki broj null vrednosti).

+ Pod klasa je takođe entitet i on sam može imati svoje pod klase.

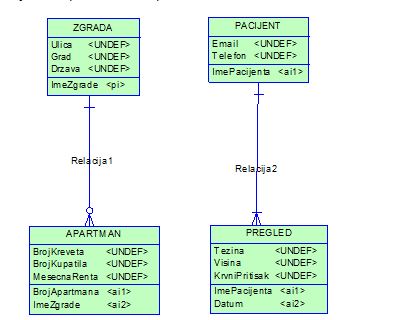
Ovako definisane nad klase i pod klase formiraju hijerarhiju specijalizacije

U hijerarhiji specijalizacije svaka pod klasa učestvuje u samo jednoj klasi / pod klasi relacije; tj. svaka pod klasa ima samo jednog roditelja!

+++ Za rešetku specijalizacije, podklasa može biti podklasa u više od jedne klase / pod klase relacije!!!

Kada jedna relacija super klase / pod klase ima više od jedne supe klase, gde super klase predstavljaju različite tipove entiteta, takvu podklasu nazivamo **unija ili kategorija.**

**Primer:** pretpostavimo da imamo tri tipa entiteta: PERSON, BANK i COMPANY. U bazi podataka za registraciju motornih vozila, vlasnik vozila može biti osoba, banka (sa založnim pravom na vozilu) ili kompanija. Potrebno je da kreiramo klasu (kolekciju entiteta) koja uključuje entitete sve tri vrste da bi igrali ulogu vlasnik vozila. Za ovu svrhu može se kreirati kategorija (unija) OWENER koja je unija pod klasa od tri skupa entiteta COMPANY, BANC i PERSON.

**ID zavisni entiteti**

Entiteti čiji identifikator uključuje identifikator drugog entiteta.

***Primer:***Identifikator entiteta apartman je složen (BrojApartmana, ImeZgrade), gde je ImeZgrade identifikator entiteta Zgrada - BrojApartmana sam po sebi nije dovoljan da nekome kažete gde živite.

ID zavistan entitet ne može da postoji ukoliko ne postoji entitet od koga zavisi (roditelj) - minimalna kardinalnost na strani roditelja je 1.

Roditelj može a i ne mora da ima ID zavistan entitet!

**Slabi entiteti** - entiteti čije postojanje zavisi od prisustva drugih entiteta;

Jak entitet je entitet koji nije slab!

Svi ID zavisni entiteti su slabi entiteti.

**Primer:** Entitet MODEL - dizajn jedne linije automobila; Entitet AUTOMOBIL - pojedinačni automobil; AUTOMOBIL ne može postojati bez modela - te je on slab entitet.

Identifikator entiteta AUTOMOBIL je VIN (broj motora) i on ne uključuje identifikator modela - AUTOMOBIL nije ID zavistan entitet!!!

Konceptualno i logičko modeliranje

Principi dobrog modelovanja konceptualnog modela

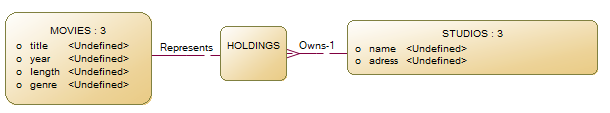
**Verodostojnost:** Znači da tipovi entiteta, njihovi atributi i relacije između tipova entiteta treba da odslikavaju realnost!

***Primer****:* Ako u bazi podataka o filmovima definišemo **relaciju Stars-in** između tipova entiteta STARS and MOVIES, to bi trebalo da bude relacija više na prema više. Razlog je očigledan jer u realnom svetu jedna zvezda filma se može pojaviti u više od jednog filma a da u filmu može igrati više od jedne zvezde. Ako više na više relaciju predstavimo kao tip entiteta, korektno je da relacija Stars-in bude jedan na prema više u oba smera!

**Sprečavanje redudantnosti**

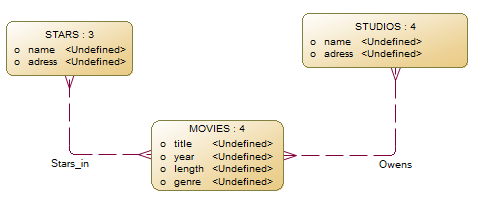
Pretpostavimo da umesto relacije između tipova entiteta MOVIES i STUDIOS treba da postavimo relaciju “movie-holdings,” koja opisuje studio koji je zakupac filma.

Tada možemo kreirati još jedan tip entiteta HOLDINGS (zakupac) i između tipa entiteta HOLDINS i tipa entiteta MOVIES uspostaviti relaciju Represents tipa jedan na prema jedan a takođe i relaciju Owens tipa više na prema jedan između tipova entiteta HOLDINS i STUDIO čime je slika kompletirana.



Struktura na slici u potpunosti predstavlja realni svet, međutim, može se reći da tip entiteta HOLDINGS nema neku korisnu svrhu i da se kao takav može izostaviti.

**Izbor prave relacije**



Da li nam je u ovom slučaju zaista potrebna relacija između Movies i Stars (**Stars in**)?

Odgovor je:

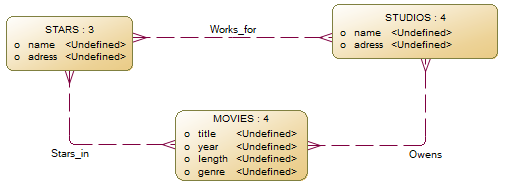
* 1. Ukoliko neka zvezda može da se pojavi u filmu samo ako postoji ugovor, film i studio koji je vlasnik filma, tada nema potrebe za relacijom Stars-in.
  2. Međutim, ako mi u našoj bazi ne znamo da ugovor postoji – imamo potrebu da zadržimo Stars-in relaciju.

Da li nam je u ovom slučaju zaista potrebna relacija između Movies i Studios (**Owens**)?

Odgovor je:

* 1. Ako za svaki film postoji bar jedan ugovor koji obuhvata film, njegov studio i neke zvezde filma, tada možemo izostaviti Owns.
  2. Međutim, ako je studio vlasnik filma a još uvek nema zvezdi pod ugovorom za taj film, tada moramo ostaviti Owns.

!!! Alternativno, mi možemo koristiti relaciju Works for kako bi označila nešto potpuno drugo: Na primer, činjenicu da je zvezda pod ugovorom sa studijom na način koji nije povezan sa filmom (npr. zvezda može da bude pod ugovorom sa jednim studio ali da takođe radi na filmu koji je u vlasništvu drugog studija. U tom slučaju, Owns relacija sigurno ne bi bila redundantna.



**Pravila “analize teksta”**

Da bi identifikovali kandidate za entitete, atribute i relacije mogu se koristiti pravila “analize teksta” koja glase:

1. Zajedničke imenice ili imenice koje nisu vlastite su kandidati za tipove entiteta
2. Vlastite imenice podrazumevaju instance entiteta
3. Zbirne imenice podrazumevaju tipove entiteta sačinjen od grupe drugih entiteta
4. Pridev podrazumeva atribut tipa entiteta
5. Glagol “biti“ podrazumeva relaciju generalizacije između instanci entiteta i tipa entiteta
6. Glagol „imati“ podrazumeva vezu agregacije ili asocijacije

**Klasni dijgrami**

U UML dijagramima klasa, klasa (slična tipu entiteta u ER) je prikazana kao pravougaonik.

Tipovi veza se nazivaju asocijacijama u UML terminologiji, a instance veze se nazivaju veze.

Binarna asocijacija (binarni tip odnosa) je predstavljena kao linija koja povezuje klase koje učestvuju (tipovi entiteta), i može opciono imati ime.

Oznaka (min, max) se koristi za određivanje ograničenja relacije, koja se nazivaju višestrukost (eng. multiplicities)u UML terminologiji. Višestrukosti su navedene u obliku min..maks, a zvezdica (\*) označava da ne postoji maksimalno ograničenje za učešće.

**Agregacija** treba da predstavi odnos između celog objekta i njegovih sastavnih delova, i ima različitu šematsku notaciju.

+++ UML takođe razlikuje jednosmerne i dvosmerne asocijacije (ili agregacije):

* U jednosmernom slučaju, linija koja povezuje klase se prikazuje sa strelicom da bi ukazala da je potreban samo jedan smer za pristupanje povezanim objektima.
* Ako nije prikazana nijedna strelica, pretpostavlja se dvosmerni slučaj, što je podrazumevajuća vrednost.

Imena asocijacija (relacija) su opciona u UML-u, a atributi relacija se prikazuju uokvireno u pravougaoniku koji je priključen isprekidanom linijom na liniju koja predstavlja asocijaciju / agregaciju!

**Logički model podataka**

Logički model podataka: predstavlja model podataka određenog domena problema koji se izražava nezavisno od određenog sistema za upravljanje bazom podataka ili tehnologije skladištenja (fizički model podataka), ali se vezuje za određenu struktura podataka kao što su relacione baze podataka, objektno orijentisane baze, ili tekstualne baze!

+ Kada je validiran i odobren, model logičkog podatka može postati osnova fizičkog modela podataka i formirati bazu podataka.

Logički model se sastoji od potpuno normalizovanih objekata čiji su svi atributi definisani.

Definisan je tip podataka i domen svih atributa, kao i atributi koji su kandidati za ključeve.

Može se reći da je logičko projektovanje relacionih baza podataka postupak kojim se ranije definisani konceptualni model transformiše u šemu, odnosno opis baze podataka u strukturu relacione baze podataka.

Ako je konceptualni model već preveden u relacioni model, na primer normalizacijom tabela, nikakva transformacija nije potrebna. Međutim, ako je konceptualni model izrađen kao model objekti – veze, potrebno je izvršiti njegovu transformaciju u relacioni model.

**Fizički model podataka**

Fizički model predstavlja način na koji treba struktuirati i povezati podatke u određenom DBMS-u, tako da je važno razmotriti konvenciju i ograničenje DBMS-a koji se koristi prilikomprojektovanja fizičkog E/R dijagrama.

Većina CASE alata koja prati RDBMS vrše automatsku transformaciju modela entiteti-veze u relacioni model i generišu bazu podataka u izabranom RDBMS.Mada se transformacija logičkog modela u fizički model može izvršiti automatski korišćenjem CASE alata, potrebno je znati pravila transformacije!

**!Transformacija logičkog u fizički model baze podataka!**

Prvi korak u transformisanju logičkog modela u fizički model je transformisanje koncepata logičkog modela u fizičke objekte. Ovo podrazumeva sledeće aktivnosti:

1. **Transformacija entiteta u tabele**
2. **Transformacija atributa u kolone**
3. **Transformacija domena u tipove podataka i ograničenja**
4. **Specifikacija primarnih ključeva**
5. **Specifikacija redosleda kolona**
6. **Definisanje referencijalnih ograničenja za sve veze**
7. **Definisanje prostora tabela (tablespace)**

**Transformacija entiteta**

Svaki tip entiteta iz E-R dijagramu se transformiše u jednu relaciju odnosno tabelu;

Identifikator tipa entiteta postaje primarni ključ relacije a drugi atributi tipa entiteta postaju atributi neprimarnih ključeva relacije;

Primarni ključ treba da ima sledeće svojstva:

* + vrednost primarnog ključa jedinstveno identifikuje svaki red u relaciji.
  + mora da bude neredudantan;
  + treba da bude kratak, numerički i fikasan. U slučajevima kada se to ne može postići, treba razmotriti mogućnost korišćenja nekog kandidat ključa kao primarnog; Ako nema kandidat ključeva, treba razmotriti mogućnost korišćenja **surogat ključa(**Primer: ID**)**

**Surogat ključ**

Ključ kojeg generiše DBMS

Ima jedinstvenu vrednost u okviru tabele i nikada se ne menja!

Dodeljuje se u trenutku kreiranja reda u tabeli a uništava se kada se red obriše

Problem sa korišćenjem surogat ključeva se javlja kada su podaci podeljeni u više baza podataka!

**Primer:** Predpostavimo da kompanija održava tri BP i da svaka od njih ima tabelu PORUDŽBINE\_KUPACA sa surogat ključem, koji je jedinstvene na nivou jedne BP → dva reda tabele u dve različite BP mogu imati istu vrednost surogat ključa, što postaje problem kada podatake iz različitih baza treba umerdžovati.

**Kandidat (alternativni) ključevi**

To su alternativni identifikatori jedinstvenih redova u tabeli

Na primer: U relaciji ***ZAPOSLENI (BrojZaposlenog, ImeZaposlenog, Telefon, Email, DatumZaposlenja, DatumProvere, KodZaposlenog)***

***Primarni ključ:*** BrojZaposlenog

***Kandidat (alternativni) ključ***: Email

**Specificiranje svojstava kolona**

**Null status:**

* dozvoljena null vrednost se definiše frazom NULL;
* nedozvoljena null vrednost se definiše sa NOT NULL.

**Tip podataka:** U generičke tipove spadaju:

* CHAR(n) – karakter string fiksne dužine
* VARCHAR(n) - karakter string varjabilne dužine
* DATE
* TIME
* MONEY
* INTEGER
* DECIMAL

(svaki DBMS ima i svoje specifične tipove)

**Default vrednost:** generiše je DBMS prilikom kreiranja redova tabele (može biti: konstanta, rezultat neke funkcije; sračunata korišćenjem složenije logike specificirane trigeriom)

**Ograničenja podataka**

**Domen** – vrednost kolona ograničava na odredjeni skup vrednosti;

Npr:vrednost KodZaposlenog može biti ograničen na (’novo zaposleni’, ’stalno zaposlen’, ’honorarno zaposlen’ itd.)

**Opseg vrednosti** – vrednosti se ograničavaju na odredjeni interval;

Npr:datum zaposlenja može biti u opsegu od 1. januara 1990 do 31. decembra 2007. godine.

**Intrarelaciono ograničenje** – vrednost kolone ograničava u odnosu na druge kolone iste tabele; Npr: datum provere može biti najmanje 3 meseca posle datuma zaposlenja.

**Interrelaciono ograničenje** - ograničava vrednost kolone u odnosu na druge kolone drugih tabela; Npr: ograničenje referencijalnog integriteta

**1:1 veza izmedju jakih entiteta**

Može biti predstavljena na jedan od dva načina:

* + Dodavanjem primarnog ključa relacije A kao stranog ključa u B.
  + Dodavanjem primarnog ključa relacije B kao stranog ključa u A.

**1:M veza izmedju jakih entiteta**

Predstavlja se spuštanjem atributa primarnog ključa relacije koja je na strani veze jedan, kao stranog ključa u relaciju koja je na strani veze više ili „Spuštanje ključa tabele roditelja u tabelu deteta.”

**M:N veza izmedju jakih entiteta**

Za vezu više-prema-više izmedju dva tipa entiteta A i B se kreira posebna relacija C!

Primarni ključ te realacije: ***složeni ključ koji se sastoji od primarnih ključeva svake od dve relacije koje se dobijaju transformacijom entiteta koji učestvuju u vezi;***

Bilo koji atribut koji nije ključ a koji je pridružen vezi M:N je uključen u relaciju C.

**Asocijativne veze**

Slične N:M relacijama, jedina razlika je u tome što one imaju jedan ili više atributa koji su dodeljeni toj relaciji a ne entitetima koji u njoj učestvuju!

**Unarna (rekurzivna) jedan-prema-više veza**

Kao i u slučaju drugih tipova 1:M relacija:

* + tip entiteta se modelira kao relacija;
  + primarni ključ relacije je isti kao i za tip entiteta;
  + relaciji se dodaje strani ključ koji se referencira na vrednost primarnog ključa (*rekurzivni strani ključ )*;

Rekurzivni strani ključje strani ključ u relaciji koji se referencira na vrednost primarnog ključa iste relacije.

**Unarna (rekurzivna) M:N veza-** Predstavlja se primerom koji se naziva struktura sastavnice

Kao i slučaju drugih tipova M:N veza:

* + tip entiteta se modelira kao relacija;
  + M:N veza se predstavlja kao nova relacija;
  + primarni ključ nove relacije je složeni ključ koji se sastoji od dva atributa (koji ne moraju da imaju isto ime) a koji uzimaju vrednost iz istog primarnog ključa.
  + ako je relaciji pridružen jedan ili više atributa (npr. atribut kolicina), oni su u novu relaciju uključeni kao atributi koji nisu primarni ključevi.

**Dizajniranje minimalne kardinalnosti**

**Problematično u relacijama 1:M (roditelj – deca),** jer ne može da se podrži kroz samu strukturu BP; moraju se dizajnirati specijalne procedure koje izvršava DBMS ili aplikacija koja radi nad bazom!

Mogu se analizirati četiri slučaja minimalne kadinalnosti:

* 1. **opcioni i roditelji i deca (O-O),**
  2. **obavezni roditelji a opciona deca (M-O),**
  3. **opcioni roditelji a obavezna decu (O-M),**
  4. **obavezni roditelji i obavezna deca (M-M)**

Analize je obavljena u dva pravca:

* + za slučaj kada su obavezni roditelji (relacije M-O i M-M) i
  + za slučaj kada su obavezna deca (O-M i M-M).

U slučaju relacije O-O ne treba preduzimati nikakve akcije!

**Akcije u slučaju obaveznog roditelja**

Potrebno je obezbediti da svaki red tabele deteta ima validan, ***not-null*** strani ključ!

Potrebno je ograničiti akcije ažuriranja i brisanja primarnog ključa roditelja i akcije kreiranja i modifikovanja stranog ključa kod dece!

**Akcije nad redovima roditelja u slučaju obaveznog roditelja**

U slučaju:

* + ***kreiranja novog roditelja*** ne treba raditi ništa, jer ni jedan red deteta još uvek ne zavisi od tog novog reda;
  + ***promene vrednosti primarnog ključa roditelja***, morale bi se promeniti sve postojeće vrednosti stranih ključeva dece (ako ih ima) ili se modifikacija primarnih ključava roditelja mora zabraniti **(kaskadno ažuriranje)**
  + ***brisanja reda roditelja***, moraju se brisati i svi postojeći odgovarajući redovi tabele dece (ako ih ima) ili se brisanje mora zabraniti **(kaskadno brisanje)**

**Akcije nad redovima dece u slučaju obaveznih roditelja**

U slučaju:

* + ***kreiranja novog reda dece***, red mora imati validan strani ključ, ako se on zahteva. Ako toga nema, kreiranje reda nije dozvoljeno
  + ***promene stranog ključa deteta***, nova vrednost mora da se podudara sa vrednošću primarnog ključa roditelja. U suprotnom, menjanje je zabranjeno
  + ***brisanja redova dece***, nema nikavih restrikcija. Redovi dece se mogu izbrisati bez ikakavih konsekveci na redove roditelja

**Akcije u slučaju obavezne dece**

Treba obezbediti bar jedan red u tabeli dece za svaki red tabele roditelja!

Za proveru obaveznosti dece mora se uvek stračunavati broj dece koji je pridružen svakom roditelju za šta treba napisati odgovarajući kod!

**Akcije nad redovima roditelja kada su obavezna deca**

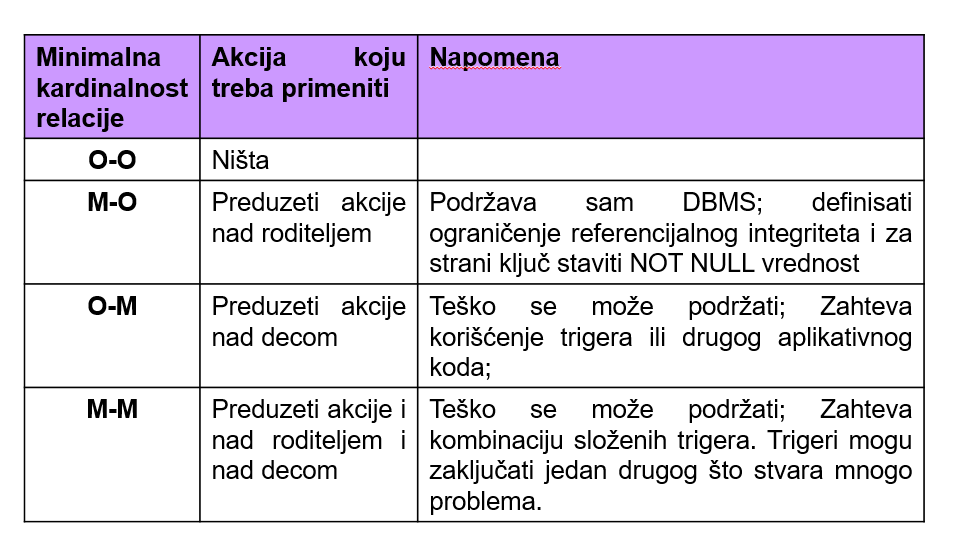
U slučaju:

* + ***kreiranja novog reda roditelja*** mora se kreierati i red deteta (ili naći neki postojeći red deteta i promeniti njegov strani ključ tako da se podudara sa primarnim ključem roditelja ili kreirati novi red dece). U suprotnom, kreiranje je zabranjeno!
  + ***ažuiranja primarnog ključa roditelja***, mora se ažurirati i strani ključ bar jednog reda deteta ili se ažuriranje ne sme dozvoliti
  + ***brisanja reda roditelja***, ne treba preduzimati nikakve akcije

**Akcije nad redovima dece kada su obavezna deca**

U slučaju:

* + ***ubacivanja novog reda dece*** ne treba preduzimati nikakve akcije
  + ***ažuriranja stranog ključa dece***, ne postoje nikakve restrikcije osim u slučaju kada je dete zadnje za datog roditelja (tada se ažuriranje ne sme izvršiti); da bi se odredio broj dece za datog roditelja, potrebno je napisati proceduru
  + ***brisanja redova dece*** postoje slične restrikcije kao i prilikom ažuriranja



**Arhitektura organizacije podataka**

**Funkcionalna zavisnost**

Pretpostavimo da smo kupili nekoliko kutija keksa i da svaka od njih košta po 5o dinara. Imajući ovu činjenicu u vidu cenu nekoliko kutija keksa možemo izračunati kao:  
CenaKeksa=BrojKutija \* 50 dinara  
Uopšteniji način da se izrazi veza između CeneKeksa i BrojKutija je reći da **CeneKeksa funkcionalno zavisna od BrojKutija** što se može napisati kao:

***BrojKutija → CeneKeksa***

Ovaj izraz se može čitati kao: BrojKutija određuje CenuKeksa!

Korišćenjem druge formule, CenaKeksa se može sračunati množenjem Količine i JediničneCene.  
*Cena = Količina \* JediničnaCena*

***( Količina, JediničnaCena ) → Cena***  
U ovom slučaju je determinanta složena ( Količina, JediničnaCena ).

! Funkcionalna zavisnost postoji kada vrednost jednog ili više atributa odredjuje vrednosti drugih atributa.

Ako posmatramo relaciju:

***Movies (title, year, length, genre, studioName, starName)***

Postoje sledeće funkcionalne zavisnosti:

**Title, year —> length, genre, studioName**

! Ukoliko dve torke imaju istu vrednost za title atribut, tada te dve torke moraju takođe da imaju istu vrednost za atribute length, genre i studioName.

Sa druge strane, ukoliko posmatramo:

**Title, year —> starName**

Vidimo da je greška i da ovo ne predstavlja funkcionalnu zavisnost. Sa sigurnošću možemo da znamo da u bazi postoji više zvezda za određeni film!!!

**Normalizacija**

Normalizacijom želimo da projektujemo skup relacija-tabela za našu bazu podataka koje:

* + sadrže sve podatke neophodne za primenu date baze podataka
  + imaju što manje ponavljanja
  + smeštaju višestruke vrednosti za tipove podataka koje to traže
  + dozvoljavaju efikasno ažuriranje podataka u bazi podataka
  + izbegavaju opasnosti gubitka podataka bez znanja

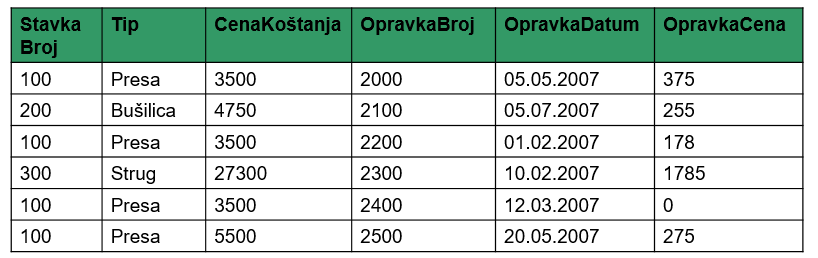
Primarni razlog za normalizaciju BP je eliminisanje anomalija:

* 1. "prilikom ubacivanja“
  2. "prilikom brisanja“
  3. "prilikom ažuriranja“

U pravilno normalizovanoj BP podaci o entitetu se smeštaju samo na jednom mestu, pa je ubacivanje, brisanje ili ažuriranje podataka moguće uraditi bez greške!!!

U nedovoljno normalizovanoj BP, podaci o entitetima se mogu čuvati na više mesta, pa je moguće lakše napraviti grešku!!!

**Primer anomalija**



**Anomalija brisanja:**

Ako se izbriše slog čiji je OpravkaBroj 2100 – gube se informacije o dva različita entiteta: mašini i njenoj opravci.

**Anomalija unosa:**

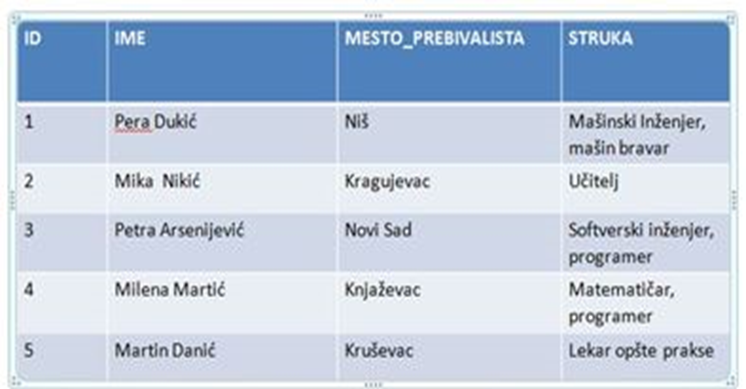
Struktura tabele nas tera da unosimo podatke o dva entiteta – opravka (broj opravke, datum i cenu opravke) i mašina (tip i cenu koštanja) što može biti problem jer oni koji rade na opravci neznaju podatke o mašini.

**Anomalija ažuriranja:**

Ako se vrednosti zadnjeg reda prethodne tabele promeni na (100, ‘Presa’, 5500, 2500, 20.05.2007, 275)

**1NF**

Relacija je u 1NF ako svi njeni atributi imaju samo atomske (nedeljive) vrednost!!!

*Primer:* Ukoliko želimo da saznamo ko su programeri moramo proći kroz sve korisnike i proveriti sadržaj polja STRUKA i pregledati poprilično nepregledno polje STRUKA i odkriti da li dotični korisnik zna programiranje ili ne.

**2NF**

Relacija je u 2NF ako je svaki atribut koji nije primarni ključ funkcionalno zavistan od celog primarnog ključa.

Druga normalna forma je zadovoljena ako se može primeniti svaki od sledećih uslova:

* 1. **Primarni ključ se sastoji od samo jednog atributa**
  2. **U relaciji ne postoje atributi koji nisu primarni ključevi**
  3. **Svaki atribut koji nije primarni ključ je funkcionalno zavistan od potpunog skupa atributa primarnih ključeva**

**Primer:**

ZAPOSLENI je jedan primer relacije koja nije u drugoj normalnoj formi. Skraćena notacija za za tu relaciju je:

* ZAPOSLENI (Zaposleni\_ID, Kurs Ime, Odeljenje, Plata, Datum\_Zavrsetka)

Funkcionalne zavisnosti u ovoj relaciji su:

* Zaposleni\_ID→ Ime, Odeljenje, Plata
* Zaposleni\_ID, Kurs→ Datum\_Zavrsetka

Primarni ključ za ovu relaciju je sastavljen od Zaposleni\_ID i Kurs. Međutim, atributi koji nisu primarni ključevi, Ime, Odeljenje i plata su funkcionalno zavisni samo od Zaposleni\_ID a ne i od Kurs.

+ Da bi se relacija prebacila u drugu normalnu formu, relaciju treba dekomponovati u nove relacije korišćenjem atributa determinanti; determinante su primarni ključevi tih relacija.

ZAPOSLENI je dekomponovan u sledeće dve relacije:

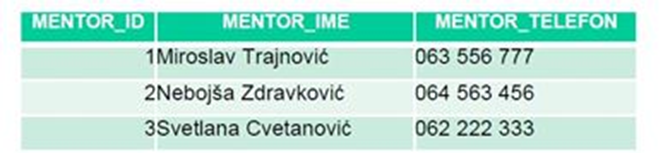
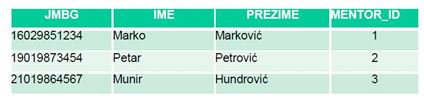
* **ZAPOSLENI (Zaposleni\_ID, Ime, Odeljenje, Plata):** ova relacija zadovoljava prvi uslov druge normalne forme
* **ZAPOSLENI\_KURS (Zaposleni\_ID, Kurs, Datum\_Zavrsetka):** ova relacija zadovoljava treći uslov druge normalne forme

**3NF**

Relacija u 3NF ako je u 2NF i ako su svi atributi koji nisu deo primarnog ključa međusobno nezavisni!!!

[](file:///C:\Users\Cecili\Desktop\PPT4DITA-Template-BMU\IT350\IS350Lekcija%203\resources\IT-BP-BPO-0.1-TrecaNormalnaForma-Slika1.JPG)Navedena relacija nije u 3NF jer MENTOR\_TELEFON zavisi od MENTOR, a MENTOR zavisi od studenta. Dakle, postoji tranzitivna zavisnost!

Kako bismo ovu relaciju stavili u 3NF moramo je razbiti i kreirati novu relaciju MENTORI koja će voditi računa o mentorima dok relacija STUDENTI dobija novi oblik.

[](file:///C:\Users\Cecili\Desktop\PPT4DITA-Template-BMU\IT350\IS350Lekcija%203\resources\IT-BP-BPO-0.1-TrecaNormalnaForma-Slika3.JPG)

**Boyce-Codd normalna forma (BCNF)**

Relacija je BCNF kada je svaka determinanta kandidat za ključ.

Dovođenjem relacije na BCNF se eliminiše funkcionalna zavisnost.

Često je dovođenje relacije na 3NF dovoljno da se eliminišu sve funkcionalne zavisnosti ali postoje slučajevi gde problemi ostaju.

Ti problemi se mogu eliminisati ukoliko se relacije dizajniraju ili redizajniraju svođenjem na Boyce-Codd-ovu normalnu formu (BCNF).

Dovođenje relacije na BCNF se vrši njenom dekompozicija na nekoliko drugih relacija. Generalna strategija je:

1. Identifikovati svaku funkcionalnu zavisnost

2. Identifikovati svaki kandidat ključ

3. Ako postoji funkcionalna zavisnost koja ima determinantu koja nije kandidat ključ:

◦ kolone te funkcionalne zavisnosti treba prebaciti u novu relaciju

◦ determinantu te funkcionalne zavisnosti treba proglasiti primarnim ključem nove

relacije

◦ kopiju determinante treba ostaviti kao sekundarni ključ originalne relacije

◦ kreirati ograničenje referencijalnog integriteta između nove i originalne relacije

4. Ponavljati korak 3. sve dok je svaka determinanta svake relacije kandidat ključ!

**4NF**

Tabela je u 4NF ako je u BCNF i ako nema zavisnosti od više vrednosti. Zavisnost od više vrednosti se javlja kada jedna determinanta određuje skup vrednosti a ne samo jednu vrednost.

Zavisnost od više vrednosti ne predstavlja problem ako je izdvojena u posebnu tabelu!



Normalizacija se u dizajniranju baza podataka primenjuje kada se iz nekih izvora dobiju

podaci koje treba smestiti u bazu podataka. Postavlja se pitanje: da li te podatke, treba na

neki način transformisati ili zapamtiti takve kakvi jesu. U donošenju takve odluke, veliku ulogu ima normalizacija.

Pri donošenju konačne odluke treba takođe pažljivo proučiti principe normalizacije tj. znati šta su njene prednosti i mane što se može sumirati na sledeći način:

+ Prednosti normalizacije

◦ Eliminisanje anomalija

◦ Smanjenje redundantnih (duplih) podataka čime se eliminiše problem integriteta

podataka i čuva prostor na diskovima

* Nedostaci normalizacije

◦ Komplikovaniji SQL upiti sa join-om više tabela i više pod upita

◦ Sporiji rad aplikacija

Denormalizacija baza podataka

Razlozi za denormalizaciju su:

1. svi ili skoro svi najčešći upiti zahtevaju pristup celom skupu spojenih podataka

2. većina aplikacija skenira tabele prilikom spajanja tabela

3. računarska kompleksnost dobijenih kolona zahteva privremene tabele ili jako kompleksne upite

+ Prednosti denormalizacije su što ona može poboljšati performanse:

1. minimiziranjem potreba za spajanjem

2. smanjenjem broja stranih ključeva u tabeli

3. smanjenjem broja indeksa, ušteda u prostoru i umanjenje vremena ažuriranja podataka

4. određivanjem vrednosti agregata ranije, to jest, njihovo računanje prilikom ažuriranja podataka umesto prilikom izbora

5. umanjenjem broja tabela - u nekim slučajevima

- Nedostaci denormalizacije su:

1. obično ubrzava dobijanje ali može usporiti ažuriranje podataka

2. uvek je specifična za svaku aplikaciju i treba je ponovo proceniti ako se aplikacija izmeni

3. može uvećati tabele

4. u nekim slučajevima pojednostavljuje kodiranje u drugim ga usložnjava

**Denormalizacija spajanjem tabela**

Primenjuje se ako se u SQL upitima dve ili više tabela redovno spajaju, a cena njihovog spajanja je velika;

Rezultujuća tabela bi trebalo da:

* Ne sadrže redundantne podatke;
* Sadrži samo kolone koje su potrebne da bi se zadovoljili upiti;
* Bude periodično kreirana;

+Prednost spajanja tabela:

* + spajanje samo jednom opterećuje sistem – u trenutku kreiranja;
  + nad unapred spojenom tabelom se upiti izvršavaju vrlo efikasno;

- Negativni aspekti spojenih tabela:

* + teško održavanje ažurnosti podataka (mogu brzo da izgube sinhronizaciju sa tabelama od kojih su kreirane, pa su korisne samo kod relativno statičkih podataka)

**Tabele za izveštavanje**

Koristi se u slučajevima kad izveštaj za korisnika nije moguće razviti uz pomoć jednostavnog SQL upita

Podaci za izveštaj se kreiraju najčešće aplikativnim programom ili SQL-om u *batch* okruženju a onda se upisuju u tabelu kao sekvencijalni podaci

Tabele su idealne za smeštanje rezultata višestrukih spajanja tabela, međuzavisnih upita ili ostalih složenih SQL iskaza

Za učitavanje rezultata iz tabele se umesto složenih i sporih upita, koristi jednostavna SELECT naredba

**Denormalizacija dupliciranjem/deljenjem tabela**

**Dupliciranje tabela:** Vrši se kada je grupi korisnika uvek potreban samo podskup podataka iz tabele; Taj podskup podataka se tada za datu grupu korisnika duplicira stavljanjem u posebnu tabelu;

**Deljenje tabela:** Vrši se kada različite grupe korisnika pristupa samo pojedinim delovima tabele; Tabela se tada deli na više denormalizovanih tabela – po jedna za svaku grupu;

Tabele mogu biti podeljene:

* + Horizontalno (slogovi su klasifikovani u grupe u zavisnosti od načina grupisanja ključa tabele)
  + Vertikalno (odvaja samo neke kolone u nove tabele)

**Denormalizacija dodavanjem kolona**

Ponekad se jednoj ili više kolona jedne tabele pristupa svaki put kad se traže podaci iz drugih tabela. U takvim slučajevima potrebno je razmotriti mogućnost dodavanja redundantnih kolona tako da podaci iz tabela kojima se pristupa budu dostupni u drugim tabelama čime se eliminiše potreba za spajanje tabela št o poboljšava performanse izvršenja upita.

**SQL-**Standard Query Language

**Osnovne karakteristike su:**

***Jednostavnost i jednoobraznost pri korišćenju-*** Tabela (relacija) se kreira jednom izvršnom naredbom. Odmah po kreiranju tabela je raspoloživa za korišćenje. Svi podaci memorisani su u tabelama i rezultat bilo koje operacije se logički prikazuje u obliku tabele

***Mogućnost interaktivnog i klasičnog (aplikativnog) programiranja***- Koristeći SQL dobijaju se odgovori na trenutne, unapred ne predviđene zahteve ili se SQL blokovi "ugrađuju" u klasični viši programski jezik (Java. PHP) omogućujući klasičnu obradu

***Neproceduralnost (tj. proceduralnost u minimalnom stepenu)-*** Ni za jedan jezik se ne može reći da je potpuno neproceduralan, već da je neproceduralan u većem ili manjem stepenu. SQL je u velikoj meri neproceduralan jer definiše ŠTA, a ne KAKO: koji podaci se žele, koje tabele se referenciraju i koji uslovi treba da budu ispunjeni, bez specifikacije procedure za dobijanje željenih podataka.

Funkcija SQL-a je da omogući definisanje, korišćenje i kontrolu podataka relacione baze kroz:

***Naredbe za definisanje podataka (data definition statements-DDL):*** Omogućuju definisanje resursa relacione baze podataka.

***Naredbe za manipulisanje (rukovanje) podacima (data manipulation statements-DML):***Omogućuju ažuriranje u širem smislu značenja te reči i izveštavanje iz relacione baze podataka.

***Naredbe za kontrolne (upravljačke) funkcije (data control functions-DCF):***Omogućavaju oporavak, konkurentnost, sigurnost i integritet relacione baze podataka.

**Naredbe za definisanje podataka – DDL**

* Tu spadaju naredbe:
  + **CREATE TABLE (kreiranje tabele baze podataka);**
  + **CREATE INDEX (kreiranje indeksa)**
  + **CREATE VIEW (kreiranje virtuelne tabele – "pogleda");**
  + **ALTER TABLE (izmena definicije tabele);**
  + **DROP TABLE (izbacivanje tabele iz baze podataka);**

**Naredbe za manipulisanje podacima – DML**

* Omogućuju ažuriranje sadržaja podataka u relacionoj bazi podataka:
  + **UPDATE (izmena vrednosti kolona tabele);**
  + **DELETE (izbacivanje redovatabele);**
  + **INSERT (dodavanje redova postojećoj tabeli);**

**Naredbe za kontrolne (upravljačke) funkcije - DCL**

* Omogućuju oporavak, konkurentnost, sigurnost i integritet relacione baze podataka:
  + **GRANT (dodela prava korišćenja sopstvene tabele drugim korisnicima);**
  + **REVOKE (oduzimanje prava korišćenja sopstvene tabele od drugih korisnika);**
  + **COMMIT (prenos dejstava transakcije na bazu podataka);**
  + **ROLLBACK (poništavanje dejstava transakcije);**

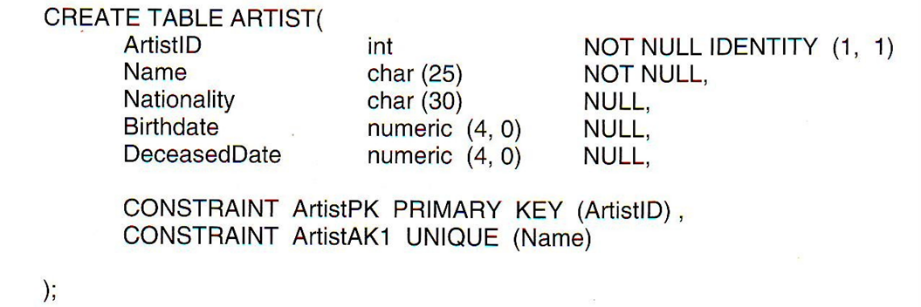
Da bi kreirali novu SQL bazu podataka, koristimo naredbu**CREATE DATABASE imeBaze;**

Ipak pri kreiranju baze podataka moramo voditi računa da ime baze mora biti jedinstveno (unique) u okviru RDBMS. U suprotnom, javiće se greška.  
Ukoliko baza podataka sa datim imenom već postoji, javlja se greška, pa morate dati neko drugo ime bazi.

Da bi olakšali ovaj proces, u MySQL-u postoji klauzula kojom možete navesti da se kreira baza podataka sa datim imenom ali samo ukoliko ne postoji*.***CREATE DATABASE IF NOT EXISTS imeBaze;**

KREIRANJE TABELA

Korišćenje naredbe **CREATE TABLE** spada u grupu naredbi za definisanje podataka (DDL).Naredba *CREATE TABLE* se koristi za:

1. kreiranje novih tabela,
2. definisanje kolona,
3. **ograničenja koja se odnose na kolone i kreiranje relacija.

Ova naredba se na isti način može koristiti u svim RDBMS (SQL Server, DB2, Oracle, MySQL), razlike postoje samo u tipovima podataka koji se koriste za opis kolona tabele.

**CREATE TABLE** naredba ima pet tipova ograničenja:

1. PRIMARY KEY,
2. UNIQUE,
3. NULL/NOT NULL,
4. FOREIGN KEY
5. CHECK.

**U prvom delu naredbe**: svaka kolona je definisana svojim imenom, tipom podataka i null statusom;

**U drugom delu naredbe**: ograničenja - primarni ključ i alternativni ključ (njegova namena je da obezbedi jedinstvenost vrednosti neke kolone pa se definiše ograničenjem UNIQUE).

Ima pet tipova ograničenja: PRIMARY KEY, UNIQUE, NULL/NOT NULL, FOREIGN KEY i CHECK;

FOREIGN KEY se koristi za definisanje ograničenja referencijalog integriteta dok se CHECK koristi za definisanje ograničenja nad podacima;

Format za pisanje ograničenja je:

**reč CONSTRAINT, ime graničenja koje daje sam programer, neko od ograničenja (u datom primeru PRIMARY KEY i UNIQUE) i jedna ili više kolona koje se stavljaju u zagradi.**

Ograničenje **FOREIGN KEY** može da sadrži klauzule kojima se specificira da li je ažuriranje ili brisanje kaskadno i to:

1. Izraz *DELETE NO ACTION* - označava da je zabranjeno brisanje redova tabele čiji se primarni ključevi koriste kao strani ključevi (foreign key) u drugim tabelama.
2. Izraz *DELETE CASCADE* -označava da je brisanje kaskadno što znači da se prilikom brisanja redova jedne tabele brišu i redovi drugih tabela u kojima su primarni ključevi date tabele koriste kao strani ključevi (foreign key). Predefinisana vrednost je DELETE NO ACTION!
3. Izraz *UPDATE NO ACTION* - označava da je zabranjeno ažuriranje primarnog ključa redova tabela ukoliko se on koristi kao strani ključ u drugim tabelama.
4. Izraz *UPDATE CASCADE* označava da je ažuriranje kaskadno. Predefinisana vrednost je UPDATE NO ACTION.

**CREATE INDEX**

Naredbom CREATE INDEX se kreiraju kao bi se:

1. naglasila jedinstvenost kolona,
2. olakšalo sortiranje i
3. omogućilo brže pretraživanje po vrednostima nekih kolona.

Kolone koje se često koriste u WHERE klauzuli predstavljaju dobre kandidate za kreiranje indeksa. One se mogu koristiti kao jednostavni uslovi u WHERE klauzuli ili se mogu pojaviti u join-u.

Da bi se kreirao jedinstveni indeks, pre ključne reči INDEX treba dodati ključnu reč UNIQUE!!!

Na primer: da bi se obezbedilo da se jedan posao ne doda dva puta u tabelu WORK, treba kreirati jedinstveni indeks nad poljima tabele (Title, Copy, ArtistID) na sledeći način:

**CREATE UNIQUE INDEX WorkUniqueIndex ON WORK (Title, Copy, ArtistID);**

**ALTER TABLE**

Naredba **ALTER TABLE** se koristi ukoliko želimo da promenimo strukturu postojeće tabele. Može se koristiti za:

1. dodavanje, izbacivanje ili menjanje kolona kada se koriste klauzule ADD, DROP, ALTER respektivno iza kojih stoji ime kolone
2. dodavanje i izbacivanje ograničenja nad tabelama kada se koriste klauzule ADD CONSTRAINT i DROP CONSTRAINT respektivno iza kojih stoji ime ograničenja

Primer 1: Ukoliko želimo da postojećoj tabeli KUPAC dodamo novu kolonu, naredbe je:ALTER TABLE KUPAC ADD NovaKolona Char(5) NULL;

Primer 2: Ukoliko hoćemo da izmenimo definiciju postojeće kolone tabela (dužinu, tip, not null), naredbe je:  
ALTER TABLE KUPAC ALTER COLUMN StaraKolona date NOT NULL;

Primer 3: Ukoliko hoćemo da izbrišemo postojeću kolonu tabele, naredbe je:  
ALTER TABLE KUPAC DROP COLUMN StaraKolona;

Primer 4: Ukoliko želimo da dodamo ograničenje:  
ALTER TABLE KUPAC ADD CONSTRAINT NovoOgraničenje CHECK ([Ime] NOT IN (‘Nikola’));

Primer 5: Ukoliko želimo da izbrišemo ograničenje:  
ALTER TABLE KUPAC DROP CONSTRAINT StaroOgraničenje;

**DROP TABLE**

Ukoliko hoćemo da izbacimo definiciju tabele, zajedno sa podacima koje sadrži, iz baze podataka koristi se naredba **DROP TABLE** naredba. Opšti oblik naredbe je:

**DROP TABLE imeTabele;**

Primer: Izbaciti definiciju tabele PREMIJA iz baze podataka.  
DROP TABLE PREMIJA;

!!! Obratiti pažnju na različito dejstvo naredbi DROP TABLE i DELETE. Naredbom DELETE se može brisati kompletan sadržaj tabele, ali sama tabela ostaje u bazi podataka i sa njom se može dalje raditi. Naredbom DROP TABLE tabela sa sadržajem se izbacuje iz baze podataka i više nije dostupna!

DBMS neće dozvoliti brisanje tabele koja ima ograničenje FOREIGN KEY. Brisanje se može izvršiti samo ako njen primarni ključ nije spušten ni u jednu drugu tabelu ili ako je naznačeno kaskadno brisanje (DELETE CASCADE). Takođe, prilikom brisanja takve tabele se može najpre izbrisati ograničenje FOREIGN KEY ili tabela u kojoj je spušten ključ.

+++ Kada hoćemo da obrišemo sve podatke jedne tabele (sav sadržaj) iz baze podataka, ali ne i da obrišemo samu definiciju tabele (kako bi je kasnije ponovo koristili), koristi se naredba TRUNCATE TABLE;

Opšti oblik naredbe je:  
**TRUNCATE TABLE imeTabele;**

**INSERT INTO TABLE**

Naredba **INSERT INTO TABLE** se koristi za dodavanje redova tabele.

Standardna verzija naredbe INSERT je;

**INSERT INTO ime\_tabele (imena kolona u koje se unose podaci) VALUES (lista podataka)**

Na primer:

INSERT INTO TABELA\_UMETNIK (Ime, Nacionalnost, Datum\_rodjenja, Datum\_smrti) VALUES (‘Tomy’, ‘Meksikanac’, ‘1927’, ‘1998’);

Treba napomenuti da se i lista naziva kolona i lista podataka nalaze unutar zagrada!

1.Ubacivanje vrednosti NEKIH kolona ili svih kolona ali redosled podataka i kolona u tabeli nije isti.

U ovom slučaju nazivi tih kolona se moraju eksplicitno navesti pa naredba insert ima oblik:

INSERT INTO naziv\_tabele (kolona1, kolona2,...) VALUES (vrednost\_1, vrednost\_2,...);

2.Prepisivanje podataka iz jedne tabele u drugu (ukoliko obe tabele imaju isti broj atributa i ukoliko su atributi identično definisani):

INSERT INTO tabela1 SELECT \* FROM tabela2;

! Međutim, ukoliko želimo da prepišemo podatke iz jedne tabele u drugu a obe tabele nemaju isti broj atributa i atributi nisu identično definisani treba koristiti sledeći oblik naredbe:

INSERT INTO tabela1 (kolona1, kolona2,...) SELECT atribut, izraz FROM tabela2 WHERE kriterijum selekcije;

**UPDATE TABLE**

Naredba UPDATE TABLE služi za menjanje postojećeg sadržaja tabele. Opšti oblik naredbe UPDATE je:

UPDATE tabela SET kolona1=izraz1 [kolona2=izraz2] [WHERE kriterijum selekcije];

ili:

UPDATE tabela SET(kolona1, kolona2, ...) = (pod upit) [WHERE kriterijum selekcije]

Pod upit mora vratiti najviše po jednu vrednost za svaku od kolona u listi kolona iza SET klauzule!!!

Primer 1: Primljeni pravnik Bojan imaće predsednika za neposrednog rukovodioca. Realizovati tu odluku gde rukovodilac ima svoju šifru.UPDATE RADNIK SET RUKOV=4468 WHERE IME='BOJAN';

**DELETE FROM TABLE**

Naredba DELETE TABLE služi za uklanjanje zapisa iz tabela navedenih u odrebdi FROM koji zadovoljavaju odredbu WHERE. Opšti oblik naredbe je:

DELETE FROM tabela WHERE kriterijumi

*Primer: Izbaciti podatke o Bojanu, jer je on napustio radnu organizaciju.*

**DELETE FROM RADNIK**  **WHERE IME = 'BOJAN';**

DELETE briše samo podatke. Struktura, atributi, indeksi ostaće netaknuti!

**UPITI**

Nad BP se mogu postavljati upiti različite složenosti:

* Upiti nad jednom tabelom kojima se prikazuje prost, neizmenjen sadržaj te tabele;
* Upiti nad jednom tabelom uz prikaz modifikovanog sadržaja tabele korišćenjem funkcija i izraza;
* Upiti kojima se realizuje dinamička zamena rezultata jednog upita u WHERE klauzuli drugog, čime se vrši povezivanje više tabela;
* Upiti kojima se realizuje spajanje tabela (JOIN) što je uobičajeni način povezivanja relacija.
* Upiti nad tabelama koje u sebi logički imaju strukturu tipa stabla (hijerarhijsku strukturu).

!!!Svi upiti se realizuju korišćenjem naredbe SELECT;

**Naredba SELECT**

SELECT naredba u SQL-u se može sastojati od do šest klauzula, ali samo prve dve - SELECT i FROM - su obavezne. Upit se kao i svaka druga SQL naredba završava se tačkom i zarezom. Klauzule su navedene u sledećem redosledu, a klauzule između uglastih zagrada su opcione:

1. **SELECT** klauzula navodi atribute ili funkcije koje treba naći.
2. **FROM** klauzula specificira sve relacije (tabele) koje su potrebne u upitu, uključujući spojene (join-ovane) relacije, ali ne one u ugnježdenim upitima.
3. **WHERE** klauzula specificira uslove za izbor torki iz ovih relacija, uključujući i uslove spajanja ako je potrebno.
4. **GROUP BY** specificira atribute grupisanja, dok **HAVING** specificira uslov za izbor grupa, a ne za pojedinačne torke. Ugrađene agregatne funkcije COUNT, SUM, MIN, MAX i AVG se koriste zajedno sa grupisanjem, ali one se takođe mogu primeniti na sve izabrane torke u upitu bez GROUP BY klauzule
5. **ORDER BY** specificira redosled prikazivanje rezultata upita.

Upiti u SQL-u mogu biti veoma složeni. Osnovni oblik SELECT izraza, koji se ponekad naziva mapiranje ili blok za selekciju se formira od tri klauzule SELECT, FROM i WHERE i ima sledeći oblik:

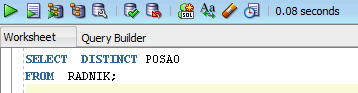
**SELECT <lista atributa>**

**FROM <lista tabela>**

**WHERE <uslov>;**

Kada se traže svi atributi neke relacije, umesto navođenja svih atributa pojedinačno, moguće je koristiti znak "**\***"sa istim dejstvom!

Koristeći klauzulu **DISTINCT** obezbeđujemo prikazivanje samo različitih vrednosti atributa!



**Klauzula WHERE**

**SQL klauzula WHERE** se koristi za specificiranje uslova prilikom preuzimanja torki (redova) iz relacije (tabele): prikazaće se samo oni redovi tabele koji zadovoljavaju zadati uslov u klauzuli WHERE. Može se reći da WHERE klauzulu treba koristiti za filtriranje i prikazivanje samo neophodnih podataka.

Napomenimo i da WHERE klauzulu ne koristimo isključivo u sprezi sa SELECT klauzulom, već se koristi i sa UPDATE, DELETE, itd. klauzulama!!!

Za specificiranje kvalifikacionih izraza u WHERE klauzuli se mogu koristiti različite vrste operatora zavisno od toga da li se trebaju primeniti nad numeričkim ili ne numeričkim vrednostima atributa.

Nad numeričkim vrednostima atributa mogu se primeniti:

1. operatori poređenja
2. logički operatori

Nad ne numeričkim (karakter vrednostima) atributa mogu se primeniti specifični operatori kao što su IN, NOT IN, LIKE …

**Korišćena u SELECT bloku ona omogućuje:**

1. Selekciju specifičnih n-torki relacije (redova tabele)

2. Selekciju n-torki koje zadovoljavaju višestruke uslove

3. Selekciju n-torki koje zadovoljavaju bar jedan od više uslova

4. Selekciju n-torki koje ne zadovoljavaju određene uslove

5. Selekciju n-torki istovremenim korišćenjem AND i OR logičkih operatora

6. Selekciju n-torki unutar izvesnog raspona

7. Selekciju n-torki koje zadovoljavaju vrednost u listi vrednosti

8. Selekciju n-torki koje sadrže određenu kombinaciju karaktera

**U WHERE klauzuli možemo koristiti operatore poređenja i to:**

**=** (jednako)

**!=** (nije jednako) -

**<>** (nije jednako) - drugi način predstavljanja !=

**>** (veće od)

**<** (manje od)

**>=** (veće od ili jednako)

**<=** (manje ili jednako)

**!>** (nije veće od)

**!<** (nije manje od)

**U WHERE klauzuli se mogu koristiti logički operatori:**

**AND** - omogućava višestruke uslove u SQL upitu, ali svi uslovi moraju biti ispunjeni da

**OR** - koristi se za kombinovanje više uslova u upitu, tako da mora biti ispunjen samo jedan uslov

**NOT** - negacija, okreće značenje logičkih operatora sa koji se koristi

Izrazi nad nizom karaktera koji se najčešće koriste su: **IN, NOT IN, LIKE**

Operator **IN** služi zaselekciju n-torki koje zadovoljavaju vrednost iz liste vrednosti.

SELECT IMEPREZIME, POSAO, S\_RJ

FROM RADNIK

WHERE POSAO **IN** ('ANALITICAR',

'SAVETNIK','TRG PUTNIK');

Klauzula **LIKE** omogućuje pretraživanje na osnovu "UZORKA" odnosno dobijanje informacija i kada ne znamo potpun naziv (tj. vrednost) određenog atributa tipa character. Ona koristi dva specijalna karaktera ("%","\_") sa sledećim značenjem:

"**%**" predstavlja string od 0 ili više karaktera

"**\_**" predstavlja poziciju jednog karaktera

*Primeri:*

*.. ime se završava sa N.*

**WHERE IME LIKE '%N'**

*.. treći karakter imena je R.*

**WHERE IME LIKE ‘\_ \_R%'**

.. ime je dugačko 5 karaktera.  
**WHERE IME LIKE ‘\_ \_ \_ \_ \_’**

.. ime nije dugačko 5 karaktera.

**WHERE IME NOT LIKE ‘\_ \_ \_ \_ \_’**

.. u imenu je slovo G posle R.

**WHERE IME LIKE '%R%G%’**

**Aritmetičke funkcije** u naredbi SELECT se koriste za izvršenje bilo koje aritmetičke operacije u upitu. Sastavljene su od imena kolona i konstantnih vrednosti povezanih aritmetičkim operatorima ("+". "\*", "-", "/"). U aritmetičkim izrazima se mogu koristiti i grupne funkcije.

Pored aritmetičkih operatora "+". "\*", "-", "/" SQL podržava sledeće i aritmetičke funkcije:

1. POWER (broj, e) – diže broj na e-ti stepen
2. ROUND (broj [,d]) – zaokruzuje broj na d decimala
3. TRUNC (broj [,d]) – odbacuje ostatak od d-tog decimalnog mesta
4. ABS (broj) – nalazi apsolutnu vrednost broja
5. SIGN (broj) – daje +1 ako je broj >0, 0 ako je broj = 0, -1 ako je broj <0.
6. MOD (broj1, broj2) – izračunava broj1 po modulu broj2
7. SQRT (broj) – nalazi pozitivan kvadratni koren broja

Funkciju za dobijanje sumarnih informacija u naredbi SELECT su:

**AVG** (atribut) – izračunava srednju vrednost

**SUM** (atribut) – izračunava ukupnu vrednost

**MIN** (atribut) – nalazi minimalnu vrednost

**MAX** (atribut) – nalazi maksimalnu vrednost

**COUNT** - prouzrokuje sažimanje sadržaja kolona nad kojima se primenjuje

Klauzula **GROUP BY** omogućava dobijanje sumarne informacije za svaku različitu vrednost kolone ili grupe kolona po kojoj se vrši grupisanje. Klauzula GROUP BY se gotovo uvek koristi uz neku funkciju za dobijanje sumarnih informacija (MIN, MAX, AVG, COUNT, SUM).

Klauzule GROUP BY se koristi u slučajevima kada je potrebno izvršiti grupisanje redova tabela i izdvajanje pojedinih grupa.

U ovim slučajevima potrebno je podeliti relaciju u podskupove (ili grupe) koji se ne preklapaju. Svaka grupa (particija) će sastoje se od redova kojiimaju istu vrednost nekih atributa, koji se nazivaju atribut (i) grupisanja. Primenom klauzule GROUP BY na svaku grupu možemo da primenimo sumarnu funkciju i za svaku od njih dobijemo sumarne informacije. Atributi grupisanja treba da se pojave u SELECT klauzuli, tako da se vrednost koja proističe iz primene svake sumarne funkcije na grupu torki pojavljuje zajedno sa vrednošću atributa grupisanja.

Klauzula **HAVING** određuje kriterijume za selekciju grupa pošto su grupe već formirane sa GROUP BY klauzulom.

Ponekad želimo da dođemo do vrednosti sumarnih funkcija samo za grupe koje zadovoljavaju određene uslove. Na primer, pretpostavimo da želimo prikazati koje poslove obavlja više od jednog radnika u svakom odeljenju. SQL obezbeđuje HAVING klauzulu, koja se može pojaviti u vezi sa GROUP BY klauzulom, u tu svrhu. HAVING-om se postavlja uslov nad sumarnim informacijama koji se odnosi na grupe torki formirane za svaku vrednost atributa grupisanja. Samo grupe koje zadovoljavaju uslove se dobijaju u rezultatu upita.

SELECT ODELJENJE, POSAO, COUNT(\*)

FROM RADNIK

GROUP BY ODELJENJE, POSAO

HAVING COUNT(\*) >= 2;

Primenom klauzule **ORDER BY** je rezultujuću tabelu moguće sortirati po jednom ili više atributa u rastućem ili opadajućem redosledu.

Za specifikaciju rastućeg redosleda koristi se klauzula **ASC**, a za specifikaciju opadajućeg redosleda klauzula **DESC**.

Rastući redosled se podrazumeva, pa klauzulu ASC nije neophodno navoditi, za razliku od klauzule DESC koju uvek treba navesti kada se sortira u opadajućem redosledu. ORDER BY je uvek poslednja klauzula u SELECT bloku.

**NULL** vrednost je nedefinisana vrednost i zbog toga se ne koristi pri izračunavanju izraza i funkcija.

Da bi se izračunavanje ipak omogućilo, koristi se **NVL** funkcija (u Oracle ili ekvivalenti ove funkcije) koja privremeno menja NULL vrednost sa vrednošću za koju se sami odlučimo, tj. vrednošću koja je neutralna u odnosu na željenu operaciju!

SELECT IMEPREZIME, POSAO, LD,

LD + **NVL** (PREMIJA, 0)

FROM RADNIK

WHERE ODELJENJE = 30;

**Formiranje upita nad dve ili više tabela**

SQL raspolaže sa dve različite tehnike za pravljenje upita nad više tabela:

1. korišćenje podupita tj. umetanje upita nad jednom relacijom u upit nad drugom i
2. **JOIN** operacije za kombinovanje više tabela

Jedan od načina povezivanja tabela relacione baze podataka je korišćenje podupita za spajanje tabela, preciznije rečeno, dinamička zamena rezultata jednog upita u WHERE klauzuli drugog.

Primer koji ukazuje na situaciju kada se mogu spojiti dva upita je sledeći: Prikazati ime i posao svakog radnika koji ima isti posao kao Dejan.

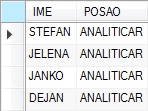
Najpre bi trebalo utvrditi koji posao obavlja Dejan. To se realizuje sledećom SQL naredbom:

http://mdita.metropolitan.ac.rs/qdita-temp/IT350/L10/IT350-IT2008-IM-DQL-SQDM-Slika103.jpgSELECT POSAO

FROM RADNIK

WHERE IME = 'DEJAN';

Iz ove naredbe saznajemo da DEJAN obavlja posao analitičara

Sada se polazni zahtev svodi na prikazivanje imena i posla svakog radnika koji obavlja posao analitičara, što se realizuje sledećom naredbom.

SELECT IME, POSAO

FROM RADNIK

WHERE POSAO = 'ANALITICAR';

!!!Kad god se uslov u WHERE klauzuli ugnježdenog upita referencira na neki atribut relacije deklarisan u spoljnom upitu, za dva upita se može reći da su povezana. Podupit za spajanje tabela možemo bolje da razumemo uzimajući u obzir da se ugnežđeni upit (onaj koji se nalazi u WHERE klauzuli) proverava po jednom za svaku torku (ili kombinaciju torki) u spoljnom upitu.

Povezivanje tabela dinamičkom zamenom rezultata jednog upita u WHERE klauzuli drugog se sastoji upravo u tome da se umesto rezultata prvog upita (u konkretnom slučaju 'ANALITIČAR') u WHERE klauzuli drugog, piše prvi upit tj. upit koji vraća taj rezultat (u konkretnom slučaju ), tako da SELECT naredba ima sledeći izgled:

SELECT IME, POSAO

FROM RADNIK

WHERE POSAO = (

SELECT POSAO

FROM RADNIK

WHERE IME = 'DEJAN'

);

Prvi upit, odnosno upit u zagradama se naziva unutrašnji upit i on se uvek izvršava prvi. Drugi upit se naziva spoljašnji upit. Pre nego što počne njegovo izvršavanje unutrašnji upit je već završen i u zagradi se nalazi konkretna vrednost rezultata izvršavanja unutrašnjeg upita.

Funkcija **EXISTS** se u SQL-u koristi se za proveru da li je rezultat podupit prazan (ne sadrži torke) ili ne. Rezultat funkcije EXISTS ima Boolean vrednost TRUE ako rezultat pod upita sadrži barem jednu torku ili FALSE ako rezultat pod upita ne sadrži nijednu torku.

Nasuprot funkciji EXISTS, **NOT EXISTS** (Q) vraća TRUE ako ne postoje torke u rezultatu podupita, inače vraća FALSE.

Podupiti su veoma moćno sredstvo za spajanje dveju tabela, a li oni imaju i niz ograničenja:

1. mogu se selektovati podaci samo iz tabele koja se koristi na najvišem nivou;
2. podupit se ne može koristiti za dobijanje podataka iz više od jedne tabele.

Zbog toga je u mnogim situacijama za selektovanje podataka iz više tabela korisnija upotreba **JOIN** operacija.

**Operacija JOIN**

Operacija **JOIN**, se koristi za kombinovanje povezanih torki iz dve relacije u pojedinačne "duže" torke. Ova operacija je veoma važna za bilo koju relacionu bazu podataka sa više od jedne relacije, jer nam omogućava da obradimo uspostavljene relacije (relationships) koje postoje među relacijama (relation).

Najčešća upotreba JOIN-a uključuje samo uslove spajanja koji podrazumevaju poređenje na jednakost. Takav JOIN, gde je jedini operater poređenja koji se koristi =, naziva se **EQUIJOIN**. Primetno je da u rezultatu EQUIJOIN uvek imamo jedan ili više para atributa koji imaju identične vrednosti u svakoj torci.

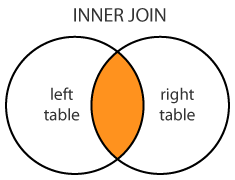
Pošto je jedan od para atributa sa istim vrednostima suvišan, kreirana je nova operacija nazvana **NATURAL JOIN**- označena sa \* - kako bi se oslobodili drugog (suvišnog) atributa u EQUIJOIN uslovu. Standardna definicija NATURAL JOIN zahteva da dva atributa po kojima se vrši spajanje (ili svaki par atributa po kojima se vrši spajanje) imaju isto ime u obe relacije. Ako to nije slučaj, prvo se preimenjuje operacija preimenovanja.

!!!Karakteristike JOIN-a za spajanje dve ili više tabela:

1. U SELECT listi atributa se daje potpuna specifikacija naziva za atribute (sastoji se od naziva tabele, tačke i naziva kolone);
2. U WHERE klauzuli se javlja uslov spajanja (do sada su prikazivani samo uslovi selekcije);
3. Uslov spajanja sadrži potpunu specifikaciju naziva atributa po kojima se vrši spajanje i operator spajanja (u ovom slučaju jednakost =);
4. Spajanje tabela moguće je vršiti u odnosu na bilo koji operator poređenja.

JOIN se može pojaviti kao:

* (INNER) JOIN
* CROSS JOIN (CARTESIAN JOIN)
* LEFT (OUTER) JOIN (LEFTOUTERJOIN)
* RIGHT (OUTER) JOIN (RIGHT OUTER JOIN)
* FULL (OUTER) JOIN
* SELF JOIN

**JOIN (INNER JOIN)** povezuje n-torke različitih tabela korišćenjem zajedničkih atributa, odnosno atributa definisanih nad istim domenima. To je uobičajeni način povezivanja tabela relacione baze podataka.

Karakteristika ovog JOIN-a je da se u WHERE klauzuli javlja uslov spajanja. Pored potpune specifikacije naziva atributa po kojima se vrši spajanje tabela navodi se i operator spajanja, u ovom slučaju jednakost (=).

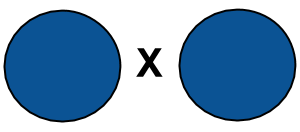
SELECT Orders.OrderID, Customers.CustomerName, Orders.OrderDate

FROM Orders **INNER JOIN** Customers

ON Orders.CustomerID=Customers.CustomerID;

U INNER JOIN torka je uključena u rezultat samo ako se podudara sa torkom u drugoj relaciji prema uslovu spajanja.

Ovaj način spajanja, kod koga je torka je uključena u rezultat samo ako se podudara sa torkom u drugoj relaciji naziva se INNER JOIN, (unutrašnja veza) i može sesmatrati podrazumevanim (default) tipom spajanja.



**CROSS JOIN** se izvršava ukoliko se u WHERE klauzuli izostavi uslov spajanja. Ako se primenjuje na dve tabele tada se svaka n-torka prve spaja sa svakom n-torkom druge tabele. Ako se primenjuje na više od dve tabele tada se rezultujuća tabela sa stoji od svih kombinacija n-torki svih tabela.

Korišćenjem CROSS JOIN nad dve tabele se dobija DEKARTOV PROIZVOD tabela pri čemu se spaja svaka torka jedne tabele sa svakom torkom druge tabele.

SELECT ime, grad

FROM RADNIK CROSS JOIN ODELJENJE;

Primer prikazuje da primena Dekartovog proizvoda u ovom slučaju nema smisla. U nekim slučajevima, međutim, izbor tog načina realizacije zahteva je veoma elegantno rešenje. Ukoliko svakoj n-torki tabele sa velikim brojem n-torki treba pridružiti jednu istu vrednost, moguće je formirati pomoćnu tabelu sa jednom kolonom i tom vrednošću u koloni i izvršiti Dekartov proizvod te dve tabele.

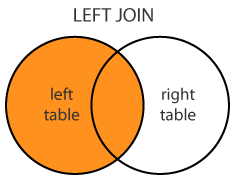
Kao što je rečeno CROSS JOIN daje Dekartov proizvod redova tabela, i pošto nema kolona po kojima se vrši spajanje, dobijaju se sve moguće kombinacije redova dve tabele.

U **OUT JOIN**-u su uključene sve torke relacije a ne samo one koje se podudaraju sa torkama u drugoj relaciji prema uslovu spajanja kao što je to slučaj kod INNER JOIN

Postoji nekoliko vrsti OUT JOIN-a:

* + **LEFT OUTER JOIN**-u rezultatu JOIN-a mora da se pojavi svaka torka iz leve relacije, koja može biti ispunjena NULL vrednostima za atribute desne relacije
  + **RIGHT OUTER JOIN**-u rezultatu mora da se pojavi svaka torka iz desne relacije, ako nema odgovarajuće torke, ispunjena je NULL vrednostima za atribute leve relacije
  + **FULL OUTER JOIN**-u rezultatu se pojavljuju sve torke iz leve relacije koje mogu biti ispunjene NULL vrednostima za atribute desne relacije kao i sve torke iz desne relacije koje mogu biti ispunjene NULL vrednostima za atribute leve relacije

**LEFT (OUTER) JOIN** daje kompletni skup zapisa iz tabele A (leve tabele sa odgovarajućim podacima iz tabele B (desne tabele) koji se podudaraju sa levom tabelom A.



SELECT column\_name(s)

FROM table1

**LEFT (OUTER) JOIN** table2

ON table1.column\_name=table2.column\_name;

Napomena: gleda se levo u odnosu na poziciju JOIN-a!!!

Ako korisnik zahteva da budu uključeni sve torke relacije u JOIN-u a ne samo one koje se podudaraju sa torkama u drugoj relaciji prema uslovu spajanja kao što je to slučaj kod INNER JOIN, eksplicitno mora biti korišćen OUTER JOIN:

LEFT OUTER JOIN: u rezultatu mora da se pojavi svaka torka iz leve relacije, koja može biti ispunjena NULL vrednostima za atribute desne relacije.

Primer:

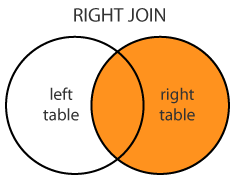
SELECT Customers.CustomerName, Orders.OrderID

FROM Customers LEFT JOIN Orders

ON Customers.CustomerID = Orders.CustomerID

ORDER BY Customers.CustomerName;

-Izlaz iz SELECT-a će vratiti sve redove iz tabele CUSTOMER bez obzira da li za kupce postoje narudžbine u desnoj tabeli ORDER.

**RIGHT (OUTER) JOIN** daje kompletni skup zapisa iz desne tabele (right table) sa odgovarajućim podacima iz leve tabele (left table).

SELECT column\_name(s)

FROM left table

**RIGHT (OUTER)** JOIN right table

ON table1.column\_name=table2.column\_name;

Napomena: gleda se desno u odnosu na poziciju JOIN-a!!!

Primer:

SELECT Orders.OrderID, Employees.LastName, Employees.FirstName

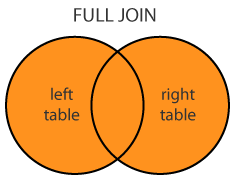
FROM Orders

RIGHT JOIN Employees ON Orders.EmployeeID = Employees.EmployeeID

ORDER BY Orders.OrderID;

-Vraća sve redove desne tabele RADNIK bez obzira da li se oni podudaraju sa levom tabelom ORDER (imena svih radnika bez obzira da su plasirali porudžbenicu ili ne)

**FULL (OUTER)** JOIN selektuje sve redove koji se nalaze i u levoj i u desnoj tabeli!

FULL (OUTER) JOIN kombinuje redove i iz LEFT i iz RIGHT joina.

SELECT column\_name(s)

FROM table1

**FULL OUTER JOIN** table2

ON table1.column\_name=table2.column\_name;

**SELF JOIN** omogućava spajanje tabele sa samom sobom po kolonama koje sadrže isti tip informacija. SELF JOIN spaja redove tabele sa ostalim redovima te iste tabele.

Primer:

ime i posao svakog radnika koji ima pretpostavljenog. Takođe prikaži ime i posao pretpostavljenog.

SELECT PODR.IME, PODR.POSAO, PODR.RUKOV, NADR.RADNIK SEF, NADR.IME SEF\_IME, NADR.POSAO SEF\_POSAO

FROM RADNIK *PODR*, RADNIK *NADR*

WHERE PODR.RUKOV = NADR.RADNIK;

Da bi se realizovao ovaj upit, tabeli RADNIK se daju dva sinonima  
– PODR i NADR. Kada su potrebni podaci o radnicima, tabela RA DNIK se referencira sinonimom PODR, a kada su potrebni podaci o neposrednim rukovodiocima – sinonimom NADR. Na taj način, pomoću navedenih sinonima, praktično su formirane dve virtuelne tabele sa identičnim sadržajem.

**Korišćenje pogleda – VIEW**

**View (pogled):** virtuelna tabela koja se kreira iz drugih tabela ili view-ova.

View nema svoje podatke i ne zauzima nikakav memorijski prostor!

Sa view-ovima se radi gotovo isto kao i sa baznim tabelama.

Rad sa pogledima je moguć korišćenjem naredbi:

1. CREATE VIEW
2. UPDATE VIEW
3. DROP VIEW

Opšti oblik naredbe za kreiranje pogleda je:

CREATE VIEW naziv-pogleda

[(nazivi atributa pogleda)]

AS SELECT ... FROM ...}

+++VIEW se može koristiti da se neke kolone sakriju kako bi se uprostila tabela il i sprečilo prikazivanje osetljivih podataka.

Na taj način se može: uprostiti tabela ili sprečiti prikazivanje osetljivih podataka!!!

Primer: Pretpostavimo da želimo da uprostimo listu kupaca galerije slika pr iskazivanjem samo njihovog imena i broja telefona:

CREATE VIEW BasicCustomerData AS

SELECT Name, AreaCode, PhoneNumber

FROM CUSTOMER;

+++Svrha VIEW-ova je da se prikažu rezultati izračunatih kolona bez obaveze korisnika da unese izraz za izračunavanje.

Primer : Kreirati view u kojem je kombinovan AreaCode(pozivni broj) i PhoneNumber.

CREATE VIEW CustomerPhone AS

SELECT Name, ‘(‘ || AreaCode || ’)’ PhoneNumber Phone

FROM CUSTOMER;

VIEW za slojevito ugrađivanje neke funkcije

Kao što je rečeno, u WHERE klauzuli SQL naredbe se ne mogu koristiti funkcije ili izrazi kojima se nešto sračunava. Umesto toga, treba kreirati VIEW u kojem je to sračunavanje obavljeno a zatim SELECT naredbu koja u WHERE klauzuli koristi taj VIEW.

PRIMER: Naći imena onih umetnika kod kojih je razlika između prodajne i nabavne cene radova (NetPrice) radova veća od 10 000;

CREATE VIEW WorkNet AS

SELECT Name, Title, Copy,

SUM(NetPrice) TotalNet

FROM ArtistWorkNet

GROUP BY Name, Title, Copy;

Sračunate vrednosti iz drugog view-a (TotalNet) se koriste u WHERE klauzuli SELECT naredbe:

SELECT Name, NetPrice

FROM ArtistWorkNet

WHERE NetPrice > 10000;

+++VIEW-ovi se mogu koristiti za izolaciju izvornih tabela od aplikativnog koda!!!

Na primer: za tabelu CUSTOMER se može kreirati:

* + VIEW **CustomerRead** koji ima samo dozvolu čitanja
  + VIEW **CustomerUpdate** sa dozvolom čitanja i ažuriranja itd

Aplikacija koja ne ažurira podatake koristi **CustomerRead** dok one koje imaju potrabu da ih ažuriraju view **CustomerUpdate**.

**UPDATE VIEW** omogućava ažuriranje pogleda. Neki pogledi se mogu ažurirati a drugi ne!

Da bi mogli vršiti ažuriranje pogleda, pogled mora biti definisan nad jednom tabelom, u definiciju pogleda moraju biti uključene sve NOT NULL kolone tabele i kolone pogleda moraju sadržati samo prost, neizmenjen sadržaj kolona tabele nad kojom je pogled definisan!

UPDATE VIEW CustomerTable1

SET Email = ‘NewEMailAddress’

WHERE CustomerID = 1000;

Ažuriranje preko pogleda se ne može izvršiti ako je:

* pogled definisan nad više tabela. T akvo ažuriranje bi značilo da se jednom naredbom vrši ažuriranje više tabela ba ze podataka, što je suprotno definiciji INSERT, DELETE i UPDATE naredbi.
* definicija pogleda napravljena tako da se iza SELECT klauzule nalaze funkcije i aritmetički izrazi.
* definicija pogleda napravljena tako da u nju nisu uključene sve NOT NULL kolone tabele nad kojom je pogled definisan.

**DROP VIEW**- izbacivanje definicije pogleda

Naredbom se izbacuje definicija pogleda iz rečnika a podaci ostaju u bazi podataka (pogled nema svoje podatke)!!!

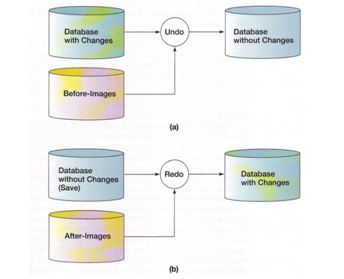
**Administracija baza podataka**

**Administracija baze podataka** se prvenstveno odnosi na komponente sistema za upravljanje bazama podataka koje omogućuju istovremeno, korektno i bezbedno izvršavanje zahteva većeg broja korisnika.

Osnovni zadaci administracije baze podatka su:

* Obezbediti oporavak (recovery) baze podataka
* Obezbediti kontrolu konkurentnosti
* Razviti zaštitu podataka
* Upravljati pravima obrade i odgovornostima
* Upravljati DBMS-om

Oporavak (RECOVERY) baze podataka

Jedan od veoma važnih elemenata, vezanih za server baze podataka, jeste mogućnost oporavka baze podataka (engl. data base recovery), što predstavlja povratak baze podataka u stanje pre softverskog ili hardverskog otkaza sistema. Razlozi za otkaz sistema mogu biti:

1. greške u operativnom sistemu,
2. greške u programiranju,
3. greške u samom SUBP-u ili
4. padanje glave diska, nestanak struje i dr.

Tehnika redundantnog pamćenja podataka i tehnika oporavka baze su veoma kompleksne. Poznato je da su do sada korišćene jednostavne procedure koje su se bazirale na periodičnom kopiranju baze podataka u neku arhivsku memoriju i svih transakcija koje su se u međuvremenu dogodile. Iako jednostavne, ove metode imaju čitav niz nedostataka.

Proces oporavka izvodi se globalno izvodi u tri koraka:

1. ***periodično kopiranje*** *(save)* baze podataka na eksternu memoriju
2. ***zapisivanje transakcija****(promena)*nad bazom podataka u žurnal, tzv.**log file**
   * Transakcije se u log file mogu upisivati pre nego što se njima ažurira baza podataka, tako da ako sistem padne u trenutku između upisa transakcije u log file i ažuriranja baze podataka postoji trag o ne izvršenoj transakciji.
   * Transakcije se u log file mogu upisivati pošto se njima ažurira baza podataka, kada postoji mogućnost da se baza ažurira ali da nema traga o tom ažuriranju u log-u pa se može desiti da korisnik ponovo unese transakciju koja je već završena.
3. oporavak baze podataka koji se može vršiti na više načina:
   * **oporavak kroz ponovnu obradu transakcija**
   * **oporavak kroz rollback ili rollforward**

**Oporavak baza podataka kroz ponovnu obradu**

-Jedan od načina za opravak baze podataka je oporavak kroz ponovnu obradu transakcija.

Predstavlja najprostiji način oporavka koji podrazumeva periodično pravljenje kopije baze podataka (koja se zove save baze podataka) u koju su zabeležene sve transakcije koje su se obavile do trenutka pravljenja kopije. U ovom slučaju nema zapisivanja transakcija u log file.

Kada dođe do greške, vrši se restore baze iz urađenog save-a i ponovo obrađuju sve transakcije koju su se desile od trenutka pravljenja save-a do trenutka pada sistema. Ovakva strategija nije uvek primenljiva jer ponovna obrada transakcija zahteva isto vreme kao i njihova prvobitna obrada a ona se ne može primeniti u slučaju sistema sa konkurentnom obradom.

Oporavak baze podataka kroz rollback ili rollforward

Oporavak kroz rollback ili rollforward podrazumeva pravljenje kopije baze podataka (database save) i čuvanja svih promena u log file-u, od trenutka pravljenja save-a. U slučaju pojave greške može se primeniti jedna od dve metode:

**Rollforward** – izvrši se restore baze podataka na osnovu napravljenog save-a i vrate sadržaji promenjenih slogova baze podataka, koji su od trenutka pravljenja save-a do trenutka otkaza sačuvani u log file-u.

**Rollback** – baza podataka se ostavlja u zatečenom stanju i vrate se nepromenjeni sadržaji slogova, koji su od trenutka pravljenja save-a do trenutka otkaza sačuvani u log file-u.

Na osnovu ovoga se može reći da se u slučaju greške, log može koristiti i za poništavanje (undo) i za vraćanje (redo) transakcija koje su se u bazi podataka dešavale od trenutka pravljenja save do trenutka nastanka greške!

Da bi se transakcije poništile (**undo**), log file mora da sadrži kopiju svakog sloga baze podataka pre nego što je on promenjen. Takvi slogovi se zovu before image. Transakcije se poništavaju primenom before image-a svih promena nad bazom podataka.

Da bi se transakcije vratile (**redo**), log file mora da sadrži kopije svih slogova baze podataka pošto su nad njima izvršene promene. Ti slogovi se zovu after image. Transakcije se vraćaju primenom after image-a svih promena nad bazom podataka.

**Checkpoint**

Restore baze podataka na neko prethodno stanje i ponovna obrada transakcija može zahtevati značajnu obradu, čiji se obim može smanjiti korišćenjem checkpoint-a.

**Checkpoint** je tačka sihronizacije između baze podataka i transakcija log file-a. Prilikom izvršenja checkpoint-a, DBMS odbacije novi zahtev, završava obradu zaostalih zahteva i njihove bafere upisuje na disk. DBMS tada čeka sve dok mu operativni sistem ne javi da su svi zahtevi za upisom u bazu podataka i u log file uspešno završeni.

!!! U trenutku checkpont-a, log file i baza podataka su sinhronizovani i checkpoint slog se upisuje u log file.

+ Kasnije se može izvršiti restore baza podataka od checkpoint-a i obraditi after image transakcija koje su se izvršile samo posle checkpoint-a.

**Oporavak baze podataka od pada sistema**

Oporavak se preduzima u slučaju da se, u toku izvršenja neke obrade, otkrije neki razlog koji onemogućava njeno uspešno kompletiranje. Taj razlog može biti:

* u samoj transakciji, kao što je prekoračenje neke od dozvoljenih vrednosti (pad transakcije)
* u sistemu, npr. prestanak električnog napajanja (pad sistema), ili
* u disku na kome je baza podataka, npr. oštećenje glava diska (pad medija).

U slučaju pada sistema, sadržaj unutrašnje memorije je izgubljen. Zato se, po ponovnom startovanju sistema, za oporavak koriste podaci iz log datoteke da bi se poništili efekti transakcija koje su se obrađivale u trenutku pada sistema. Ove transakcije se mogu identifikovati čitanjem sistemskog loga unazad, kao transakcije za koje postoji BEGIN TRANSACTION slog ali ne postoji COMMIT slog.

+++ Fizički upis u chek-point tački znači da će se sadržaj bafera prepisati na spoljni medijum bez obzira da li su baferi puni ili ne. To dalje znači da se fizički upis svih ažuriranih podataka uspešno kompletirane transakcije može garantovati tek u tački pamćenja koja sledi za uspešnim kompletiranjem transakcije.

Tako dolazimo do zaključka da su kompletiranje transakcije (upis COMMIT sloga u log datoteku) i definitivni upis svih ažuriranja te transakcije u bazu – dve odvojene radnje, za koje se ne sme dogoditi da se jedna izvrši a druga ne.

Mehanizam koji to obezbeđuje je protokol upisivanja u log koje se vrši unapred (**WAL – Write Ahead Log**). Prema ovom protokolu, pri izvršenju operacije COMMIT prvo se odgovarajući slog fizički upisuje u log datoteku, pa se zatim podaci upisuju iz bafera podataka u bazu. Ako dođe do pada sistema posle upisa COMMIT sloga u log datoteku a pre nego što je sadržaj bafera podataka prepisan u bazu, taj se sadržaj može restaurisati iz sistemskog loga REDO logikom.

Pri ponovnom startovanju sistema, posle pada sistema, moguće je prema sadržaju log datoteke identifikovati neuspele transakcije – kandidate za poništavanje, i uspele transakcije – kandidate za ponovno izvršavanje.

**Oporavak BP od pada transakcija**

Jedan aplikativni program može se sastojati od većeg broja transakcija. Izvršenje jedne transakcije može da se završi planirano ili neplanirano.

Do planiranog završetka dolazi izvršenjem COMMIT operacije kojom se uspešno kompletira transakcija, ili eksplicitne ROLLBACK operacije, koja se izvršava kada dođe do greške za koju postoji programska provera (tada se radnje transakcije poništavaju a program nastavlja sa radom, izvršenjem sledeće transakcije).

Neplanirani završetak izvršenja transakcije događa se kada dođe do greške za koju ne postoji programska provera; tada se izvršava implicitna (sistemska) ROLLBACK operacija, radnje transakcije se poništavaju a program prekida sa radom.

Izvršavanjem operacije COMMIT, efekti svih ažuriranja transakcije postaju trajni, tj. više se ne mogu poništiti procedurom oporavka. U log datoteku se upisuje odgovarajući slog o kompletiranju transakcije (COMMIT slog), a svi katanci koje je transakcija držala nad objektima – oslobađaju se.

Izvršenje COMMIT operacije ne podrazumeva fizički upis svih ažuriranih podataka u bazu (neki od njih mogu biti u baferu podataka, pri čemu se, efikasnosti radi, sa fizičkim upisom u bazu čeka dok se baferi ne napune). Činjenica da efekti ažuriranja postaju trajni znači da se može garantovati da će se upis u bazu dogoditi u nekom narednom trenutku; u slučaju pada sistema, na primer, posle COMMIT operacije a pre fizičkog upisa podataka iz bafera u bazu, upis se garantuje REDO-logikom.

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

Pojedinačne radnje ažuriranja u okviru jedne transakcije, kao i operacija početka transakcije (eksplicitna ili implicitna BEGIN TRANSACTION operacija), izvršavaju se na način koji omogućuje oporavak baze u slučaju pada transakcije. Oporavak podataka u bazi i vraćanje baze u konzistentno stanje omogućuje se upisom svih informacija o ovim radnjama i operacijama u sistemski log.

Pri izvršavanju operacija ažuriranja (u užem smislu), osim što se ažurira baza, u log se beleže vrednosti svakog ažuriranog objekta pre i posle ažuriranja (uz ostale informacije o tom ažuriranju).

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

Operacija ROLLBACK, bilo eksplicitna ili implicitna, sastoji se od poništavanja učinjenih promena nad bazom. Izvršava se čitanjem unazad svih slogova iz log datoteke koji pripadaju toj transakciji, do BEGIN TRANSACTION sloga. Za svaki slog, promena se poništava primenom odgovarajuće UNDO-operacije.

**Kontrola konkurentnosti**

Mere kontrole konkurentnosti se preduzimaju da bi se sprečilo da rad jednog korisnika loše utiče na rad drugog. U mnogim slučajevima, mere konkurentnosti obezbeđuju da korisnik u obradi u kojoj učestvuje više korisnika dobije iste rezultate koje bi dobio kada bi radio sam. Kontrola konkurentnosti se vezuje za pojam transakcije koja se definiše kao skup akcija koje predstavljaju logičku jedinicu posla nad bazom podataka.

*Primer: u sistemu baza podataka jedne banke, prenošenje novca sa jednog računa na drugi bi predstavljalo jednu transakciju. Ona se sastoji od bar dve akcije: isplate sa jednog i uplate na drugi račun, ali su te radnje logički nedeljive i samo se zajedno smatraju jednim poslom. Na efekte transakcije ne sme da utiče eventualno istovremeno izvršenje drugih transakcija, ili nestanak struje*.

Transakcija se može definisati i kao logička jedinica posla pri radu sa podacima. Ona predstavlja niz, tj. sekvencijalnu kompoziciju radnji koja ne narušava uslove integriteta. Kontrola konkurentnostiznači da po izvršenju kompletne transakcije stanje baze treba da bude konzistentno, tj. da nisu narušeni uslovi integriteta baza podataka

Dakle, posmatrana kao jedinica posla, transakcija prevodi jedno konzistentno stanje baze u drugo takvo stanje baze, dok u među koracima transakcije konzistentnost podataka može biti i narušena. Transakcija tako predstavlja i bezbedno sredstvo za interakciju korisnika sa bazom.

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

1. **atomičnost (Atomicity):** transakciju treba izvršiti u celosti ili je uopšte ne treba izvršiti (ni jednu njenu radnju);
2. **konzistentnost (Consistency):** transakcija prevodi jedno konzistentno stanje baze u drugo konzistentno stanje baze;
3. **izolacija (Isolation):** efekti izvršenja jedne transakcije su nepoznati drugim transakcijama sve dok se ona uspešno ne kompletira;
4. **trajnost (Durability):** svi efekti uspešno kompletirane transakcije su trajni, tj. mogu se poništiti samo drugom transakcijom.

Ove osobine znače da kada se transakcija završi, njeni podaci su konzistentni i stabilni na disku. Ovo je dobro za one koji rade na razvoju aplikacija ali zahteva sofisticirano zaključavanje koje može prouzrokovati nedostupnost podataka i obično predstavlja težak uzorak (eng. pattern) za većinu slučajeva korišćenja.

Kao osnovne komponente transakcije posmatraćemo objekte (npr. slogove) i dve radnje:

1. čitanje
2. upis (ili ažuriranje)

Dve radnje u ovom modelu su konfliktne ako se obavljaju nad istim objektom a jedna od njih je radnja upisa. To znači da parovi konfliktnih radnji mogu biti samo parovi:

**(čitanje, upis)** **(upis, upis)**

Problem konkuretnosti:

Korisnik transakcije A čita stavku 100, menja je i ponovo zapisuje u bazi podataka. Korisnik transakcije B izvršava iste akcije, ali nad stavkom 200. CPU obrađuje transakciju korisnika A sve dok se ne pojavi neki prekid zbog I/O operacije ili nekog drugog kašnjenja kada operativni sistem prenosi kontrolu na korisnika B.

CPU tada obrađuje transakcije korisnika B do pojave drugog prekida kada operativni sistem ponovo prenosi kontrolu na korisnika A. Korisnicima izgleda da je obrada transakcija istovremena ali ona je konkurentna, što je prikazano na slici.

Prilikom konkurentnog izvršenja transakcija nad bazom podataka može doći do sledećih problema:

1. Nekonzistentne analize
2. Izgubljenog ažuriranja
3. Poništenog ažuriranja



**Zaključavanje**

Problemi konkurentnosti opisani u prethodnom odeljku mogu se rešiti mehanizmom dinamičkog zaključavanja i otključavanja objekata, koji obezbeđuje linearizovanost izvršenja. Linearizovana izvršenja, s obzirom da su ekvivalentna serijskim, ne poznaju ni jedan od pomenutih problema.

Moguće je realizovati različitu granularnost, tj. veličinu objekata zaključavanja; na primer, objekti zaključavanja mogu biti cele tabele, pojedinačne n-torke relacije, delovi n-torki, grupe atributa, skupovi n-torki koje zadovoljavaju zadati uslov, itd.

Zaključavanje objekta (postavljanje katanca na objekat) je postupak koji obezbeđuje transakciji pristup objektu, i kojim transakcija istovremeno sprečava druge transakcije da pristupe tom objektu.

Svaka transakcija na kraju svog izvršavanja otključava sve objekte koje je sama zaključala (a koje nije već otključala). Pretpostavimo da u jednostavnom modelu postoji samo jedna vrsta katanca, i da pri izvršenju skupa transakcija nijedna transakcija ne može zaključati već zaključani objekat. U realnim modelima transakcija postoji više vrsta katanaca, od kojih su neki deljivi a neki ekskluzivni; pri tom više transakcija može da postavi deljivi katanac na jedan objekat, ali najviše jedna može da postavi ekskluzivni katanac.

**Eksluzivno zaključavanje** zaključava objekat za bilo kakav drugi pristup, ni jedna druga transakcija ne može da čita ili da menja taj objekat.

**Deljivo zaključavanje** zaključava objekat za menjanje ali ne i za čitanje, što znači da druge transakcije mogu da ga čitaju ali ne i da ga menjaju.

Kod zaključavanja, postoje dve faze:

1. faze zaključavanja objekata
2. faze otključavanja objekata

U fazi zaključavanja nema nijedne radnje otključavanja objekta, odnosno u fazi otključavanja više nema radnji zaključavanja (naravno, obe ove faze uključuju i druge operacije nad objektima kao što su čitanje, obrada, upis objekata). Ova strategija omogućava da se zaključavanje izvrši kada je to potrebno, ali kada se otključa prvi slog, zaključavanja više nema. Faza otključavanja objekata počinje prvom radnjom otključavanja.

+++Specijalan slučaj dvofaznog zaključavanja je korišćenje komandi COMMIT i ROLLBACK što je prisutno u mnogim RDBMS sistemima. U tom slučaju se zaključavanje vrši samom transakcijom, ali se zaključavanje ne ukida sve dok se ne izvrši komanda COMMIT ili ROLLBACK.

Osim što obezbeđuje linearizovanost, dvofaznost transakcija može proizvesti i neželjeni efekat poznat kao **uzajamno blokiranje transakcija (deadlock)!**

+Skoro svi DBMS sistemi imaju algoritme za rešavanja situacija u kojima su transakcije blokirane. Prvo, DBMS mora da detektuje da je došlo do blokiranja a zatim da prekine jednu od transakcija i iz baze podataka ukloni sve promene koje je ona uzrokovala.

Generalno, postoje dva osnovna stila zaključavanja:

1. **optimistično**
2. **pesimistično**

Kod optimističnog zaključavanja, pretpostavka je da se konflikt neće dogoditi. Podaci se čitaju, transakcije se obrađuju, vrše se ažuriranja i tada se proverava da li je došlo do konflikta. Ako nije, transakcija se završava. Ako se konflikt pojavio, transakcija se ponavlja sve dok se ona ne obradi bez konflikta.

Kod pesimističnog zaključavanja, pretpostavka je da će se konflikt pojaviti. Najpre se vrši zaključavanje, zatim obrađuje transakcija, a zatim se vrši otključavanje.

Ne može se definisati generalna strategija zaključavanja!

Nivo i tip zaključavanja se mogu menjati koričćenjem komandi:

1. COMMIT: označava uspešan kraj transakcije i trajan upis efekata svih radnji transakcije u bazu;
2. ROLLBACK: označava neuspešan kraj transakcije i poništenje svih efekata koje su proizvele radnje te transakcije;

Transakcija postaje deo aplikacije ako se izvršava:

od početka programa do prve radnje COMMIT ili ROLLBACK;

između dve susedne takve radnje u programu;

odnosno od poslednje takve radnje do kraja programa;

Ako se COMMIT i ROLLBACK ne uključe u program, čitav program predstavlja jednu transakciju. Na kraju programa se generiše:

komanda COMMIT ako je kraj programa uspešan;

komanda ROLLBACK ako je kraj programa neuspešan;

**Zaštita baza podataka**

Bezbednost treba razvijati u dva smera. Jedan je na osnovu uvažavanja i svesti o postojanju pretnji, a drugi na osnovu tehnika koje postoje za njihovo izlečenje. Pretnje uključuju:

1. **Neovlašćenu modifikaciju:** Promene vrednosti podataka zbog sabotaže, kriminala ili neznanja koje mogu biti omogućene neodgovarajućim bezbednosnim mehanizmima, ili deljenjem password-a, na primer.

Neovlašćeno otkrivanje: Kada su informacije koje nije trebalo da budu obelodanjene otkrivene. Opšte pitanje od ključnog značaja je da li je to slučajno ili namerno.

2. **Gubitak dostupnosti:** Ponekad se naziva uskraćivanje usluge. Kada baza podataka nije dostupna, nastaje gubitak. Dakle, treba izbegavati svaku pretnju koja izaziva prestanak rada sistema, čak i da bi se proverilo da li se nešto dogodilo.

3. **Finansijski gubitak:** Većina finansijskih gubitaka nastalih zbog prevare potiče od zaposlenih. Kontrola pristupa obezbeđuje i zaštitu od krivičnih dela i dokaze o pokušajima (uspešnim ili drugim) da se izvrše radnje koje su štetne za organizaciju, bilo da se radi o prevari, izvlačenju osetljivih podataka ili gubitku dostupnosti.

**4. Ličnu privatnost i zaštitu podataka:** Na međunarodnom planu, lični podaci su obično predmet zakonodavne kontrole. Lični podaci su podaci o pojedincu koji se može identifikovati. Pojedinac mora biti živ, ali metod identifikacije nije u potpunosti propisan. U nekim slučajevima, pojedinca može identifikovati poštanski broj mesta stanovanja, ako samo jedna osoba živi na adresi sa datim poštanskim brojem. Takvi podaci zahtevaju pažljivo rukovanje i kontrolu. Lične podatke treba identifikovati kao takve. Mora postojati kontrola upotrebe tih podataka (što može ograničiti ad-hoc upitima). Kao dokazi se moraju čuvati tragovi revizije svih pristupa i objavljivanja informacija.

5. **Zloupotrebu računara:** pri zloupotrebi računara se primenjuje opšte zakonodavstvo. Zloupotreba uključuje kršenje kontrole pristupa i pokušaje da se prouzrokuje šteta promenom stanja baze podataka ili uvođenjem crva i virusa koji ometaju njeno ispravno funkcioniranje.

6. **Zahteve za revizijom:** Ovo su operativna ograničenja koja su nastala zbog potrebe da se zna ko je šta uradio, ko je pokušao da nešto uradi i gde i kada se sve to dogodilo. Zahtevi za revizijom omogućavaju otkrivanje događaja izvršenih nad bazom podataka (uključujući CONNECT i GRANT transakcije), pružanje dokaza za osiguranje kao i odbranu ili krivično gonjenje.

**Bezbedonosni modeli** objašnjavaju raspoložive osobine DBMS-a koje treba koristiti za razvoj i upravljanje sigurnosnim sistemima. Oni otelovljuju koncepte, implementiraju politiku i obezbeđuju servere za takve funkcije. Bilo koja greške u bezbedonosnom modelu može prouzrokovati ili nezaštićen rad sistema ili nesiguran sistem. U tu svrhu treba vršiti:

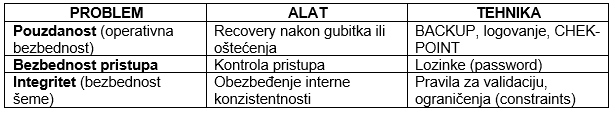
1. Kontrolu pristupa
2. Autentifikaciju i autorizaciju

**Autentifikacija i autorizacija:** Svi smo mi kao korisnici upoznati sa zahtevima za prijavljivanje (log-in) na većini sistema. Pristup IT resursima obično zahteva proces prijavljivanja koji je siguran. Ovde se autentifikacija i autorizacija prvenstveno odnosi na pristup sistemima za upravljanje bazama podataka i predstavlja sagledavanje procesa iz perspektive DBA. Drugi modeli sistema se razlikuju u većoj ili manjoj meri, iako osnovni principi ostaju isti.

**Kontrola pristupa:** Svrha kontrole pristupa mora uvek biti jasna. Kontrola pristupa je skupa u smislu analize, projektovanja i operativnih troškova. Primenjuje se na poznate situacije, na poznate standarde, radi postizanja poznatih ciljeva. Kontrolu pristupa ne treba primenjivati bez svih gore navedenih znanja. Kontrola uvek mora da odgovara situaciji.

Diskreciona kontrola pristupa postoji kada se specifičnoj imovini odrede određene privilegije, tako da je ovlašćeni korisnici mogu koristiti na određeni način.

Da bi se DBMS zaštitio, mora se konstruisati matrica pristupa, koja uključuje objekte kao što su tabele, redovi tabela, pogledi i operacije za svakog korisnika - svaki unos razdvaja privilegije kreiranja, čitanja, unošenja i ažuriranja. Ova matrica može postati veoma složena pošto ovlašćenja mogu varirati od objekta do objekta.



Za dodeljivanje privilegije korišćenja drugim korisnicima koristi se naredba **GRANT**:

Opšti oblik GRANT naredbe je:

**GRANT** <Privilegija>

ON <tabele ili pogled>

TO <korisnik ili grupa korisnika>

[WITH GRANT OPTION];

Privilegije mogu biti:

* SELECT
* UPDATE
* INSERT
* DELETE
* INDEX
* EXPAND (dodavanje atributa relacije)
* ALL (važi za sve navedene privilegije)
* RESOURCE (omogućuje korisniku kreiranje objekata baze podataka, kao što su: tabele, indeksi, klasteri)
* DBA (obavlja administrativne zadatke, kao što su: CREATE TABLESPACE i CREATE ROLLBACK SEGMENT)

Kada treba poništiti privilegije koristi se naredba **REVOKE**.

Primer: ako se želi oduzeti privilegija korisniku ALEKSANDAR za ubacivanje podataka u tabelu RADNIK, to se piše:

REVOKE INSERT ON RADNIK FROM ALEKSANDAR;

Privilegije se mogu dati određenom čoveku (korisniku) ali se mnogo češće daju grupi ljudi. Ta grupa ljudi se često naziva rola ili grupa korisnika. Davanje dozvola rolama je uobičajeno ali nije obavezno. Međutim, kada se koriste role, neophodno je imati način da se korisnici dodele odgovarajućim rolama.

Dodeljivanje privilegija odnosno odgovornosti koje imaju pojedini korisnici ne može omogućiti sam DBMS ili aplikacija koja radi nad bazom podataka već se ono mora odraditi ručnim procedurama i objasniti korisnicima u toku obuke za korišćenje sistema. Zadatak DBMS je da upravlja obradom prava i odgovornosti.

+++ Kada se korisnik prijavi na bazu podataka, RDBMS ograničava akcije korisnika na dozvole koje su definisane za korisnika ili dozvole koje su definisane na nivou rola koje su korisniku dodeljene. Određivanje da li je neko zaista onaj za kog se deklariše, je generalno veoma težak zadatak. Svi komercijalni RDBMS koriste verifikaciju korisnika uz pomoć username-a i password-a i takva zaštita je sasvim dovoljna ukoliko korisnik svoj password čuva samo za sebe.

Zaštita na nivou RDBMS se može proširiti i zaštitom na nivou aplikacije. Generalno, uvek prvo treba obezbediti zaštitu na nivou RDBMS pa ako se proceni da ona nije adekvatna, treba joj dodati i zaštitu na nivou aplikacije.

Zaštitu na nivou aplikacije se može objasniti na primeru Internet aplikacija kod kojih se zaštita najčešće obezbeđuje preko Web servera. Izvršenje aplikacija na Web serverima predstavlja način da se osetljivi podaci ne prenose kroz mrežu i na taj način zaštite. Pretpostavimo da je Web aplikacija pisana tako da se na klik korisnika na stranici pregledača prema Web serveru a zatim i prema RDBMS šalje sledeći upit:

SELECT \* FROM EMPLOYEE;

Ako se politikom zaštite na nivou aplikacije korisniku dozvoljava da pristupi samo svojim sopstvenim podacima, Web server bi mogao da doda sledeću WHERE klauzulu:

SELECT \* FROM EMPLOYEE

WHERE EMPLOYEE.Name = ’<% = SESSION ((„EmployeeName“))%>’;

U ovakvom upitu će Web server u WHERE klauzuli popuniti ime zaposlenog.

**Paralelna obrada baza podataka**

Dok se mnoge baze podataka nalaze samo na jednoj mašini, postoje i ***baze podataka koje su distribuirane na više mašina***. Takođe postoje i baze podataka koje su smeštene samo na jednom računaru sa više paralelnih procesora. Kada se obrada podataka vrši bilo paralelno bilo distribuirano, javljaju se mnogi implementacioni problemi koje je potrebno rešiti.

Operacije nad bazama podataka često zahtevaju dosta vremena i obuhvataju veliku količinu podataka pa se mogu unaprediti ukoliko se primeni paralelna obrada podataka. Suština paralelne obrade je korišćenje više procesora. Obično je broj procesora p veliki po nekoliko stotina ili hiljada. Pretpostavka je da svaki procesor ima svoju lokalnu keš memoriju. U mnogim organizacijama, svaki procesor ima i svoju spoljašnju memoriju (po svakom procesoru postoji jedan ili više diskova a u nekim arhitekturama i velika kolekcija diskova kojima procesor može direktno da pristupa).

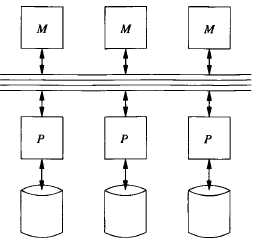
Osim toga, u paralelnim obradama postoji mogućnost prenosa informacija između procesora (npr. svi procesori u jednom reku su obično povezani)

Paralelna arhitektura se može podeliti na tri grupe:

1. Kod najčvršće povezane paralelne arhitekture se deli glavna memorija (**Shared -Memory Machines**).
2. Kod slabo povezane arhitekture se deli spoljašnja memorija (disk) a ne glavna memorija (**Shared-Disk Machines**)
3. Arhitektura koja se često koristi kod baza podataka se naziva arhitektura kod koje se ništa ne deli (**Shared-Nothing Machines**), kod koje se ne deli disk a procesori su povezani i dele se samo prenosom poruka.

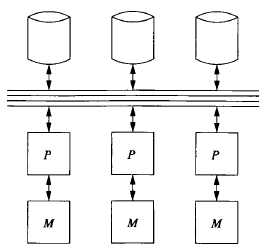
**Mašina sa deljivom memorijom (eng. Shared -Memory Machines):**

U ovoj arhitekturi, ilustrovanoj na slici, svaki procesor ima pristup do svih memorija svih procesora. Na taj način, postoji samo jedan fizički adresni prostor za čitavu mašinu, umesto jednog adresnog prostora za svaki procesor. Na slici je prikazano da procesor nema svoju lokalnu keš memoriju (označenu sa M), ali ima direktan pristup do memorije drugih procesora kada je to potrebno. S druge strane, svaki procesor ima svoju lokalnu spoljašnju memoriju (disk) koju koristi kad god je to moguće.



Velike mašine ove klase su tipa NUMA (nonuniform memory access) i njima je potrebno više vremena da bi procesor pristupio podacima u memoriji koja pripada drugim procesorima nego što mu je potrebno da pristupi svojoj sopstvenoj memoriji ili memoriji procesora u njegovom lokalnom klasteru. Međutim, u ovoj arhitekturi, razlika u vremenu pristupa memoriji nije velika. Svi pristupi sekundarnoj memoriji, bez obzira gde se nalaze podaci troše mnogo više vremena nego što traje pristup kešu, tako da je kritičan problem da li su ili ne podaci koji su potrebni procesoru na svom sopstvenom disku.

**Mašina sa deljivim diskom (engl. Shared-Disk Machines)**

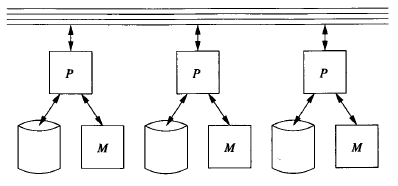
U ovoj arhitekturi, prikazanoj na slici svaki procesor ima svoju sopstvenu keš memoriju kojoj se ne može direktno pristupiti iz drugih procesora. Međutim, ovde se može pristupiti disku iz bilo kog procesora kroz komunikacionu mrežu. Kontroler diska razrešava potencijalno konkurentske zahteve koji dolaze iz različitih procesora.

Potrebno je da broj diskova i procesora bude identičan, kako se to vidi na slici. Ova se arhitektura danas pojavljuje u dva oblika, zavisno od jedinice za transfer između diska i procesora:

1. kao farma diskova koja se naziva network attached storage (NAS) i
2. kao storage area networks (SAN) koji prenosi blokove sa diska u i iz procesora.

**Mašina kod koje se ništa ne deli (engl. Shared-Nothing Machines)**

Ovde, svi procesori imaju svoju sopstvenu keš memoriju i svoj sopstveni disk ili diskove kao što je prikazano na slici. Čitava komunikacija se obavlja preko mreže, od procesora do procesora. Na primer, ako jedan procesor P želi da pročita torke sa diska drugog procesora Q, tada procesor P šalje poruku procesoru Q tražeći od njega podatke. Q dobija torke sa svog diska i isporučuje ih preko mreže drugom porukom koju prima P. Kao što je napomenuto, shared-nothing arhitektura se najčešće koristi u bazama podataka. Ona je relativno jeftina.



*Primer: Paralelni algoritam u slučaju shared-nothing machine na primeru naredbe SELECT kojom se vrši selekcija (izdaju redovi tabele prema definisanom uslovu) nad jednom relacijom:*

Najpre je potrebno razmotriti na koji način je najbolje sačuvati podatke. Korisno je podatke distribuirati na što veći broj diskova. Pretpostavka je da imamo jedan disk po procesoru. Tako, ako imamo p procesora torke svake relacije R će biti podeljene na diskove p procesora.

Da bismo izvršili selekciju nad relacijom R za uslov C, možemo koristiti svaki od procesora kako bi ispitao torke relacije R na svom sopstvenom disku. Svaki procesor treba da pronađe torke koje zadovoljavaju uslov C i da ih kopira na izlaz. Kako bi se sprečila komunikacija između procesora, nađene torke t čuvamo u istom procesoru koji torke t ima na svom disku. Na taj način je rezultat selekcije podeljen između procesora kao što je podeljena i relacija R.

Kako selekcija torki R koje zadovoljavaju uslov C može biti ulazna relacija u drugu operaciju i kako želimo da skratimo proteklo vreme i što manje držimo sve procesore zauzete sve vreme, potrebno je da selekcija relacije R bude podeljena jednako među procesorima.

**Distribuirane baze podataka**

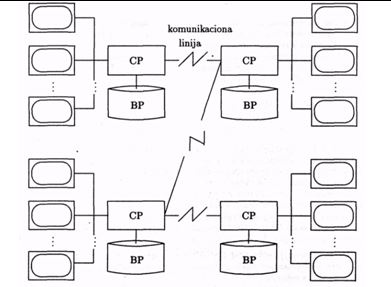
Sistem za upravljanje distribuiranim bazama podataka je i sam distribuiran, pa se označava sa DDBMS – distribuirani DBMS. Distribuirane baze podataka i DDBMS zajedno obrazuju sistem distribuiranih baza podataka. Za obe vrste sistema koristi se i kraći termin distribuirani sistem, kada je iz konteksta jasno na koju vrstu sistema se termin odnosi.

Osnovna karakteristika distribuiranih sistema je velika količina poruka i podataka koji se prenose preko komunikacione mreže. Zato je glavni cilj u postizanju efikasnosti distribuiranih sistema – smanjenje mrežne komunikacije. Ovaj cilj se projektuje na sve ključne funkcije DBMS; one moraju da se razmatraju iz novog ugla, a problemi u njihovoj realizaciji zahtevaju nova rešenja.

Neformalno govoreći, distribuirana baza podataka je baza podataka koja se ne nalazi u celosti na jednoj fizičkoj lokaciji (na jednom računaru), već je razdeljena na više lokacija koje su povezane komunikacionom mrežom. Svaka lokacija, koja se zove i čvor komunikacione mreže, poseduje svoj sopstveni, autonomni sistem za upravljanje bazama podataka, sa sopstvenom kontrolom, upravljačem transakcija i oporavka od pada, i ostalim značajnim funkcijama, a ima i svoj centralni procesor i ulazno/izlazne uređaje.

Osnovna pretpostavka za uspešnost sistema za upravljanje distribuiranim bazama podataka je nevidljivost lokacije. Naime, ovaj sistem treba da obezbedi još jedan nivo fizičke nezavisnosti podataka: korisnik i njegov program ne treba da znaju na kojoj se lokaciji u distribuiranom sistemu nalaze podaci koji su im potrebni. Za korisnika, distribuirani sistem treba da izgleda identično sa nedistribuiranim (centralizovanim), tj. pristup podacima treba da je isti kao da su svi podaci smešteni u lokalnom čvoru korisnika. Sistem odlučuje (ne uključujući korisnika) o tome da li će potrebne podatke, ukoliko su na udaljenoj lokaciji, doneti na lokalni čvor za obradu, ili će obradu preneti na udaljenu lokaciju.

Sistem distribuiranih baza podataka se može predstaviti grafom čiji su čvorovi – lokacije, a grane – komunikacione linije. Svaka lokacija sadrži centralni procesor (CP), lokalnu bazu podataka (BP) i pripadni DBMS, kao i skup klijentskih računara. Dve lokacije su povezane ako među njima postoji direktna veza koja funkcioniše, ili ako postoji treća lokacija koja je sa njima u direktnoj vezi.

Na različitim lokacijama, lokalni sistemi za upravljanje bazama podataka mogu biti različiti. Tada se DDBMS naziva heterogenim. Ako su DBMS na svim lokacijama isti, DDBMS se naziva homogenim.

**+ Prednosti distribuiranih baza podataka**

1. ***Lokalna autonomija podataka, upravljanja i kontrole:*** Okruženje u kome se distribuirane baze primenjuju, obično je i samo logički i fizički distribuirano (npr. univerzitet, fakulteti, odseci, odeljenja; centralna biblioteka, matične biblioteke, ogranci; ministarstvo, regionalni centri, itd.). Distribuiranje baza podataka kao i sistema za upravljanje njima, omogućuje pojedinim grupama da lokalno kontrolišu sopstvene podatke, uz mogućnost pristupa podacima na drugim lokacijama kada je to potrebno;
2. ***Veći kapacitet i postupni rast:*** Čest razlog za instaliranje distribuiranog sistema je nemogućnost jednog računara da primi i obrađuje sve potrebne podatke. U slučaju da potrebe nadmaše postojeće kapacitete, dodavanje čvora distribuiranom sistemu je znatno jednostavnije nego zamena centralizovanog sistema većim;
3. ***Pouzdanost i raspoloživost:*** Distribuirani sistemi mogu da nastave svoje funkcionisanje i kada neki od čvorova privremeno izgube funkcionalnost
4. ***Efikasnost i fleksibilnost:*** Podaci su fizički blizu onome ko ih stvara i koristi, pa je znatno smanjena potreba za udaljenom komunikacijom.

+++ Jedan jednostavan specijalni slučaj distribuiranog sistema je klijent/server sistem. To je distribuirani sistem u kome su neki čvorovi klijenti a neki serveri, pri čemu su na serverima smešteni podaci (i sistemi za upravljanje podacima) a, na klijentima se izvršavaju aplikacije. Korisnik, odnosno aplikacija vodi računa o tome na kom serveru su smešteni relevantni podaci, što znači da u ovim sistemima nije ostvarena nevidljivost lokacije.

**Postizanje efikasnosti u distribuiranim BP**

Da bi se postigla efikasnost DDBMS-a, sve njegove komponente treba realizovati na način koji smanjuje mrežnu komunikaciju.

Najznačajniji problemi koje tom prilikom treba rešiti su:

1. fragmentacija podataka
2. distribuirana obrada upita
3. distribuirano (preneto) ažuriranje
4. upravljanje katalogom
5. distribuirano izvršenje skupa transakcija, što uključuje konkurentnost, integritet, oporavak i protokole kompletiranja transakcija

FRAGMENTACIJA PODATAKA: Podaci u distribuiranom sistemu mogu biti podeljeni (partitioned) ili ponovljeni (replicated) u fizičkoj memoriji:

**U slučaju ponavljanja podataka (replicated)**, jedan logički objekat može imati više fizičkih reprezentacija (veći broj kopija) na većem broju lokacija. Ponovljenost podataka povećava raspoloživost podataka i efikasnost pristupa podacima, ali u značajnoj meri usložnjava ažuriranje podataka, koji moraju biti konzistentni u svim svojim kopijama. Složenost koju nosi sobom strategija ponavljanja podataka mora biti sakrivena od korisnika, tj. mora biti obezbeđena nevidljivost ponavljanja podataka.

**U slučaju deljenja podataka (partitioned**), logički skup podataka (skup podataka sa logičkog nivoa) treba na neki način podeliti, a zatim delove – fragmente (eventualno sa ponovljenim kopijama) razdeliti po raznim lokacijama. Logički skup podatka u relacionom sistemu je relacija, a prirodni fragment relacije (koji je opet relacija) jeste neki njen podskup definisan uslovom projekcije i restrikcije. Fragmentacija mora biti izvedena tako da se spajanjem fragmenata može dobiti polazna relacija.

DISTRIBUIRANA OBRADA UPITA

Distribuirana obrada upita podrazumeva distribuiranu optimizaciju kao i distribuirano izvršavanje upita. Strategije optimizacije upita nad distribuiranom bazom podataka imaju za cilj da minimizuju cenu obrade i vreme za koje će korisnik dobiti odgovor. U troškovima obrade najveću stavku čine troškovi mrežne komunikacije, tj. prenosa podataka kroz mrežu, dok su troškovi komunikacije sa ulazno/izlaznim uređajima i korišćenja procesora niži za nekoliko redova veličine. Zbog toga je veoma značajno, u zavisnosti od propusnosti mreže (količine podataka koje može da primi u sekundi) i vremena kašnjenja, pravilno odabrati relacije i njihove fragmente koji će biti prenošeni sa jedne lokacije na drugu u cilju obrade upita (globalna optimizacija).

Razlog za prenošenje podataka može biti to što su podaci na lokaciji različitoj od one na kojoj se postavlja upit, ili što u upitu učestvuje veći broj relacija sa različitih lokacija. Izbor strategije za izvršenje operacija na jednoj lokaciji poznat je kao lokalna optimizacija.

Ako se n relacija R1, R2,..., M koje učestvuju u upitu nalaze na k različitih lokacija l1,l2,...,lk, pri čemu je svaka relacija Ri – u celosti na lokaciji lj, onda se osnovna strategija distribuirane obrade upita sastoji od sledeća dva koraka:

1. Maksimalna redukcija svake relacije na njenoj lokaciji (lokalna restrikcija i projekcija na atribute spajanja i izlazne atribute). Prenošenje dobijenih relacija na jednu lokaciju, ili na više lokacija, redom, na kojima je moguće izvršiti pojedinačna spajanja i projekciju na izlazne atribute.
2. Za drugi korak osnovne strategije vezana je odluka o tome koje se relacije prenose i na koje lokacije. Ta odluka se donosi na osnovu procene količine podataka koji se prenose između lokacija u svakom pojedinačnom slučaju; izbor relacija i lokacija vrši se tako da minimizuje protok podataka kroz mrežu.

Distribuirana optimizacija se posmatra na dva nivoa:

1. Globalna optimizacija
2. Lokalna optimizacija

**Globalna optimizacija** ima za cilj da minimiza:

* cenu obrade (gde najveću stavku čine troškovi mrežne komunikacije)
* vreme za koje će korisnik dobiti odgovor;
* veoma je značajno, u zavisnosti od propusnosti mreže i vremena kašnjenja, pravilno odabrati relacije i njihove fragmente koji će biti prenošeni sa jedne lokacije na drugu;

**Lokalna optimizacija:** smanjiti količinu podataka koji se prenose kroz mrežu jer se:

* podaci nalaze na lokaciji različitoj od one na kojoj se postavlja upit;
* u upitu koristi veći broj relacija sa različitih lokacija;

ALGORITMI ZA IZVOĐENJE OPERACIJA SPAJANJA

Znatno poboljšanje efikasnosti može se postići primenom sofisticiranih algoritama za izvođenje nekih operacija. Jedan od takvih algoritama, primenjen u sistemu SDD-1, odnosi se na izvršenje operacije spajanja i uključuje operaciju polu-spajanja (semijoin). Rezultat operacije polu-spajanja (engl. semijoin) relacija R i S jednakje rezultatu spajanja tih relacija, projektovanom na atribute relacije R.

Tako, ako je potrebno izvršiti spajanje relacije R (sa lokacije l1) i relacije S (sa lokacije l2), umesto da se cela relacija R prenese na lokaciju l2 i tamo spoji sa relacijom S, moguće je izvršiti sledeći niz radnji:

* izračunati projekciju relacije S po atributu spajanja, na lokaciji l2 (rezultat je relacija TEMPI)
* preneti relaciju TEMPl na lokaciju l1
* izvršiti polu-spajanje relacije R i TEMP1 po atributu spajanja, na lokaciji l1 (rezultat je relacija TEMP2)
* preneti relaciju TEMP2 na lokaciju l2,
* izvršiti spajanje relacija TEMP2 i S po atributu spajanja, na lokaciji l2.

PRENETO AŽURIRANJE

Kao što je već rečeno, ponavljanje podataka podrazumeva da jedan logički objekat (npr. relacija ili jedan njen fragment) mogu imati više fizičkih reprezentacija (kopija) na većem broju lokacija. Posledica ove ideje je da se, s obzirom na potrebu za konzistentnošću podataka u svim kopijama, ažuriranje jednog logičkog objekta mora preneti i na sve fizičke kopije tog objekta. Međutim, momentalno prenošenje ažuriranja na sve kopije može da onemogući (ili da nedopustivo dugo odloži) uspešno izvršenje ažuriranja u slučaju da je bilo koja od lokacija u padu; time ponavljanje podataka smanjuje umesto da povećava raspoloživost podataka.

Jedan široko prihvaćeni pristup prenošenju ažuriranja oslanja se na koncept primarne kopije, i sastoji se u sledećem postupku:

1. jedna kopija svakog ponovljenog objekta proglašava se za primarnu kopiju tog objekta, pri čemu primarne kopije različitih objekata mogu biti na različitim lokacijama
2. operacija ažuriranja objekta smatra se logički izvršenom čim se izvrši ažuriranje primarne kopije tog objekta; ažuriranje ostalih kopija je sada u nadležnosti lokacije na kojoj je primarna kopija, ali se mora izvršiti pre kompletiranja transakcije

Ovaj postupak zahteva primenu protokola dvofaznog kompletiranja transakcije, koji se pak ne može uspešno sprovesti ako je bar jedna relevantna lokacija u padu, što je nezanemarljivo čest slučaj. Ukoliko se dopusti ažuriranje kopija i posle kompletiranja transakcije, ne može se garantovati konzistentnost podataka u svim njihovim kopijama. Ipak, neki komercijalni distribuirani sistemi pribegavaju tom rešenju jer puno poštovanje ažuriranja svih kopija pre kompletiranja transakcije može bitno da poveća vreme obrade.

**Katalog** = sistemska BP koja sadrži podatke o baznim relacijama, pogledima, indeksima, korisnicima, itd, a u slučaju distribuiranog sistema, i o načinu i lokacijama na koje su podaci razdeljeni i (eventualno) ponovljeni;

Katalog u distribuiranom sistemu može biti:

* centralizovan (samo na jednoj lokaciji)
* potpuno ponovljen (na svim lokacijama po jedna kopija kataloga)
* podeljen (na svakoj lokacijije deo katalogakoji se odnosi na objekte sa te lokacije)
* kombinovan (katalog je podeljen, ali na jednoj lokaciji postoji i jedna centralna kopija kompletnog kataloga)

Vrste napada na baze podataka

Spoljni napadi na baze podataka izvode hakeri koji dolaze sa eksterne mreže, oni deluju na zaštitni zid poslovnog okruženja s ciljem da prođu kroz njega i dođu do bitnih informacija ili da naruše operativnost sistema. Zaštita unutar poslovnog okruženja postaje jedna od najvažnijih stvari prilikom zaštite baza podataka. Taj vid zaštite je izuzetno važan zbog korisnika koji imaju pristup mrežnim resursima iznutra.

Kada dođe do takvog vida pretnje, tada pretnje ne dolaze samo spolja već i iznutra od strane zaposljenih koji imaju pristup bitnim podacima pri čemu je teško definisati preventivno polje zaštite. Tada zaštitni zid nije od neke koristi. Prilikom zaštite podataka iznutra, mora se znati ko može imati pristup kom delu baza podataka.

1. **Zloupotreba preterane privilegije (excessive privilege abuse)** se javlja u slučaju kada se korisnicima ili aplikacijama odobre privilegije za pristup bazi podataka, pa se takve privilegije zloupotrebe. Kao primer možemo uzeti naš univerzitet. Administrator na univerzitetu ima sposobnost da izmeni informacije o raznim informacijama kako o zaposlenima tako i o studentima. Ukoliko bi došlo do zloupotrebe privilegija, na primer u bazi bi mogla da se izmeni ocena studenta. Kako bi se sprečila zloupotreba zbog preteranih privilegija, može se uvesti kontrola pristupa na nivou upita. Ona se odnosi na neki mehanizam koji ograničava privilegije za bazu podataka na minimalni broj SQL operacija i podataka.
2. **Zloupotreba legitimne privilegije** je zloupotreba koja se javlja od strane legitimnog korisnika. Na primer na fakultetu postoje podaci o ispitnim pitanjima koja se nalaze na sistemu. Ispit se formira tako što se pitanja sa testova, sa svih petnaest predavanja sakupe i random listaju. Samo unošenje testova i realizacija istih se razvija preko Interneta. Ukoliko zlonamerni student ima dovoljno iskustva može zaobići ograničenja i ući u bazu podataka. Snimajući podatke, student može koristiti to na ispitu. Sprečavanje zloupotrebe kada se radi o legitimnom pristupu je kontrola pristupa bazi podataka koja se primenjuje ne samo na specifične upite već i na kontekst koji se odnosi na pristup bazi podataka.
3. Do **uvećavanja privilegije** dolazi tako što sami napadači iskoriste ranjivost softverske platforme baze podataka i tako povećaju privilegije pristupa uobičajnog korisnika. Običan korisnik tada dobija privilegije administratora i samim tim mu se sva vrata otvaraju. Ranjive tačke se mogu pronaći u skladištenim procedurama, ugrađenim funkcijama, implementacijama protokola, SQL upita. Kako bi se sprečio ovakav način upada uvode se kombinacije tradicionalnih sistema za detekciju upada i kontrola pristupa na nivou upada. Sami ISP ispituje saobraćaj i javlja o aktivnostima koje se dešavaju. Za razliku od zaštitnog zida on nema mogućnost da zaustavi saobraćaj već samo da ga detektuje i prijavi. Tada se preuzimaju odgovarajući koraci.
4. Sami operativni sistemi sadrže **ranjive tačke**, i dodatni servisi koji se instaliraju u server baze podataka te mogu dovesti do pojave neovlašćenog pristupa, oštećenja podataka ili do odbijanja izvršenja usluge. Kako bi se sprečio napad na platformu potrebno je stalno ažuriranje softvera i sistema za sprečavanje upada. To znači da je potrebno redovno krpiti sistem kako bi sistem bio što sigurniji.

**5. SQL injekcija** je napadačka tehnika koja se koristi kako bi se ugrozila sigurnost web aplikacije koja konstruiše SQL upite iz korisnički unesenih podataka. SQL injekcija je tehnika koja se sastoji od upisivanja odgovarajućeg teksta u polja obrasca koja aplikacija koristi u svojim SQL upitima, tako da upiti iz baze vrate podatke koje inače ne bi. Ova metoda, kao i većina hakerskih postupaka, počiva na metodi pokušaja i grešaka i temelji se na pogađanju strukture SQL upita i imena atributa i tablica. Teško je reći da postoji siguran način da se spreče SQL injekcije. Kako bi se smanjile verovatnoće njihovih događaja mogu se koristiti neke od poznatih tehnika:

* Proveravati ulaz
* Zabraniti navodnike i escape znakove u ulazu
* Koristiti ograničene parametre
* Zaključavati sistem
* Koristiti uskladištene procedure za pristup bazi podataka
* Izolovati web server
* Podesiti prijavljivanje grešaka
* Enkripcija podataka

**6. Slaba autentikacija** napadačima omogućava da pribave identitet legitimnih korisnika baze podataka kada putem krađe ili na neki drugi način dobiju akreditive neophodne za prijavljivanje.

**7. Skladištenje arhiviranih baza podataka često je nezaštićeno od napada**. Kao rezultat toga, nekoliko ozbiljnih previda u pogledu bezbednosti su doveli do krađa traka sa arhiviranim podacima iz baze podataka. Kako bi se sprečilo otkrivanje podataka iz arhive svi podaci iz baze bi trebali biti šifrovani.

**8. Odbijanje pružanja usluge (Denial of Service - DOS)** je kategorija generalnog napada kod kojeg se budućim korisnicima ne dozvoljava pristup aplikacijama na mreži ili podacima. Odbijanje pružanja usluge može se izvesti na osnovu mnogih tehnika. DOS se može realizovati tako što bi se iskoristila ranjivost platforme u bazi podataka sa ciljem da se prekine rad servera. Uobičajene tehnike DOS-a obuhvataju oštećene podataka, preplavljivanja mreže, i preopterećenje resursa servera. Preopterećenje servera naročito je uobičajno u okruženjima baze podataka. Zaštita od aktiviranja DOS zahteva preuzimanje zaštite na višestrukim nivoima. Neophodne su zaštite na svim nivoima mreže aplikacije i baze podataka.

**XML baze podataka**

U interakciji između XML-a i baza podataka, XML se nože koristiti na dva načina:

1. u bazi podataka se može pamtiti sam XML dokument (native baze podataka)
2. XML se smešta u relacionu bazu podataka kao skup tabela eksplicitno dizajniranih za pamćenje XML dokumenata (XML enabled baze podataka)

U XML enable bazama podataka treba posmatrati dva procesa:

1. ***izdvajanje podataka iz baze i konstruisanje XML dokumenta*** – što je poznato kao publikovanje ili kompozicija.
2. ***izdvajanje podataka iz XML dokumenata i pamćenje tih podataka u bazi podataka*** – što je poznato kao isecanje ili dekompozicija.

Bez obzira da li se XML dokumenti seku ili publikuju, treba naglasiti dve važne stvari:

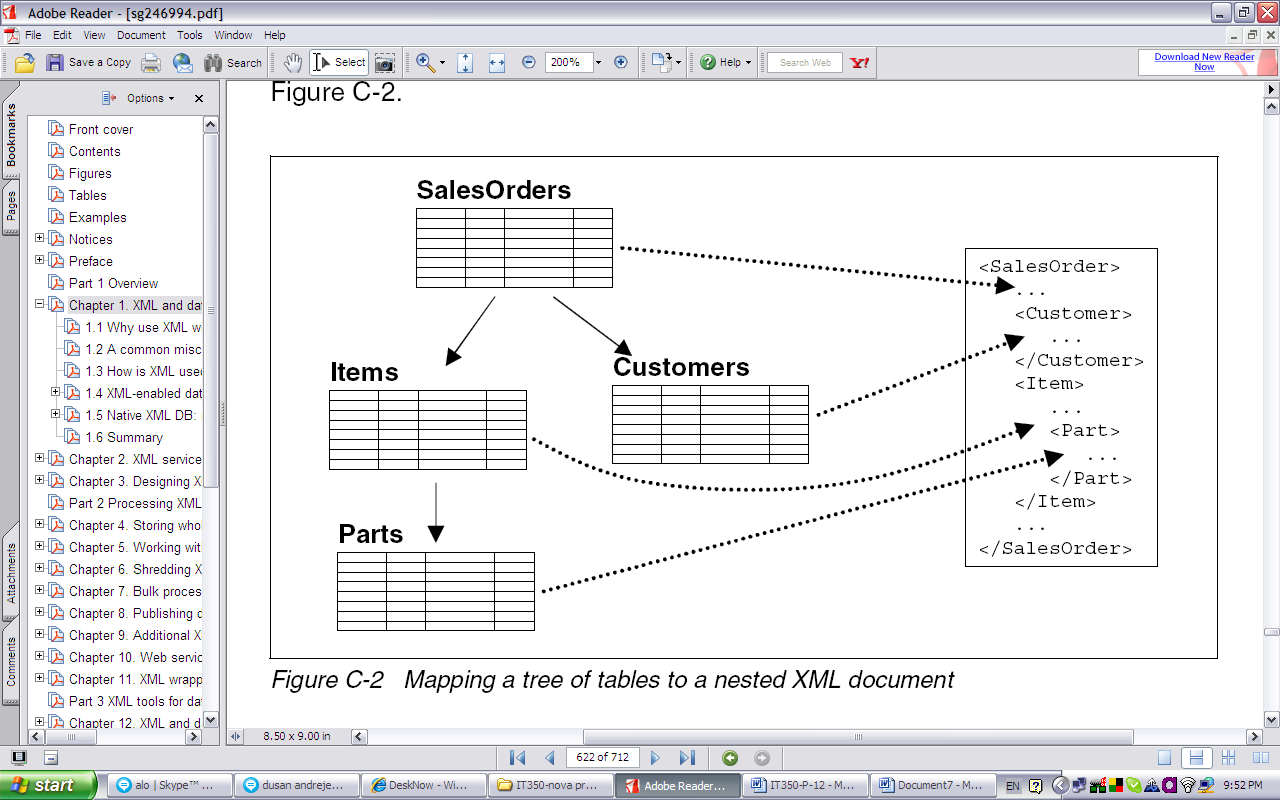
1. U samoj bazi podataka XML se “ne vidi” – tj. XML dokument je potpuno izvan baze podatke. On se konstruiše na osnovu podataka koji se nalaze u bazi podataka ili se koristi kao izvor podataka koji se pamte u bazi.
2. Drugo, šema baze podataka se slaže sa XML šemom, što znači da je za svaku šemu baze podataka potrebna različita XML šema. Baza podataka koja na taj način koristi XML je poznata kao XML-enabled baza podataka.

Kada se koristi XML-enabled baza podataka, neophodno je šemu baze podataka mapirati u XML šemu (ili obrnuto)!!!

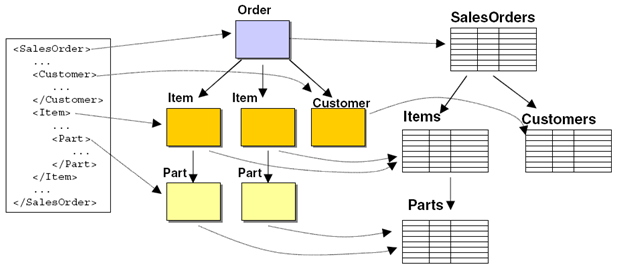
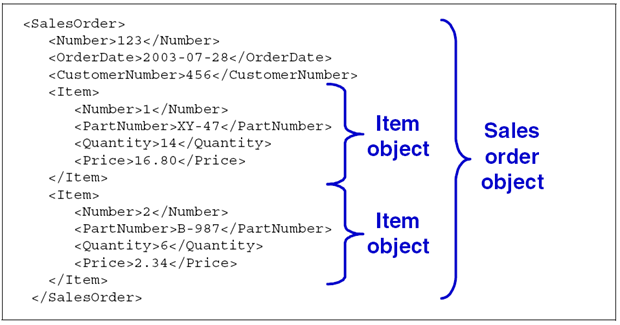
Mapiranje šeme baze podataka u XML šemu je moguće na tri načina:

1. **bazirano na tabelama** (table-based)
2. **objektno-relacionim mapiranjem** (object-relational)
3. **upitnim jezicima**

**Mapiranje bazirano na tabelama**: individualni redovi tabela se umeću u XML dokument, s tim što relacije primarni ključ/ strani ključ koje postoje u bazi podataka određuju kako će ti redovi biti umetnuti. Tako stabla tabela u bazi podataka postaju stablo elemenata u XML dokumentu.



**U objektno-relacionom mapiranju** XML dokument se posmatra kao serijsko stablo povezanih objekata kojim se na jedan očigledan način ukazuje na relacije koje postoje između njih. Objekti iz XML dokumenta se u bazu podataka mapiraju tehnikom objektno-relacionog mapiranja, što podrazumeva da se objekti mapiraju u tabele, svojstva objekata u kolone a međusobne veze između objekata u relacije primarni ključ / strani ključ.



Native XML baze podataka

Native XML baza podataka je baza podataka koja:

1. Definiše XML model podataka. Minimalni model uključuje elemente, atribute, tekst i redosled dokumenata
2. XML dokument koristi kao osnovnu logičku jedinicu za pamćenje podataka
3. Može koristiti bilo koju fizičku strategiju za pamćenje

XML modeli podataka se koristekao osnova za XML upitne jezike i definiše minimalnu količinu informacija koju native XML baze podataka moraju da pamte.

Na primer: svi XML modeli podataka uključuju elemente, atribute, tekst i redosled dokumenata, dok neki modeli uključuju stvari kao što su reference na entitete i sekcije CDATA a drugi ne.

**+ Native XML baze podataka** su slobodne da definišu svoje sopstvene XML modele jer u vreme kada su se one prvi put pojavile, nisu postajali standardizovani XML modeli. Tako među njima postoje native XML baze koje se zasnivaju na info setu, DOM objektima, XPath modelu podataka i XQuery modelu podataka. Mnoge native XML baze podataka danas podržavaju XPath 1.0 i njihov model podataka s tendencijom da mnoge od njih u budućnosti koriste XQuery model podataka.

+++ Najbolji način da se razume potreba za native XML bazama podataka je analizirati primere iz realnog okruženja. Na primer:

**1. Upravljanje dokumentima** kao što je korisnička dokumentacija, marketinške brošure, Web stranice itd. Native XML baze podataka posebno odgovaraju ovim slučajevima jer se XML modeli podataka dobro uklapaju u ovu vrstu dokumenata.

**2. Polu-struktuirani podaci koji se takođe dobro uklapaju u XML model** podataka iz dva razloga. Prvo, XML model podataka može smestiti razgranatu šemu bez gubljenja prostora. Drugo, XML model podataka je proširiv pa se ne zahteva fiksna šema.

**3. Transakcije čije izvršenje traje dugo.** Mnoge transakcije elektronske trgovine danas koriste XML dokumente kao način za razmenu informacija između različitih delova aplikacije. Pošto ove transakcije često mogu da traju nekoliko nedelja (na primer, svaki deo mora da bude potvrđen od strane nekog ovlašćenog lica), native XML baze podataka su dobar način za pamćenje jednog stanja transakcije dok je ona u fazi izvršenja, čak i kada se krajnje stanje te transakcije pamti u relacionoj bazi podataka. Native XML baza podataka dozvoljava da se nad tekućim stanjem transakcije postavi upit korišćenjem XML upitnih jezika a omogućava i korišćenje drugih XML alata kao što je XSLT transformacija.

**4. Arhiviranje dokumenata.** Mnoge kompanije kao što su one u farmaceutskoj i finansijskoj industriji moraju da arhiviraju dokumente jer ih na to obavezuju pravni propisi. Ako su to XML dokumenti, tada je native XML baza podataka prirodni izbor za arhiviranje jer njena podrška za XML upitni jezik dozvoljava da se dokumenti koriste kao izvor istorijskih podataka koji se često koriste pri analizi trendova.

***Minimalni model podataka uključuje elemente, atribute, tekst i redosled dokumenata!***

+++ Može koristiti bilo koju fizičku strategiju za pamćenje.

XML modeli podataka se koriste kao osnova za XML upitne jezike i definiše minimalnu količinu informacija koju native XML baze podataka moraju da pamte.

Osnovna jedinica za pamćenje podataka je najmanja grupa logički povezanih podataka koja se pamti. U relacionim bazama podataka to je red a u native XML bazama osnovna jedinica za pamćenje podataka je dokument. Obzirom da je bilo koji fragment dokumenta koji je označen kao jedan element potencijalno XML dokument, izbor šta će činiti jedan XML dokument je čisto stvar odluke dizajnera.

Na primer:

1. Knjiga pisana u XML-u. Mada će se mnogi ljudi složiti da knjiga treba da bude podeljena na više dokumenata a da je takođe smešno da za svaki paragraf postoji poseban dokument, nije jasno šta je idealna veličina dokumenta. Na primer, treba li svaka knjiga da sadrži poglavlje? Sekciju? Podsekciju? To u potpunosti zavisi od knjige i uslova pod kojim će se ona koristi.

2. Medicinski podaci zapamćeni kao XML dokumenti. Poseban XML dokument može postajati za svakog pacijenta, za svakog pružaoca medicinskih usluga (doktor, bolnica, klinika itd.) i za svaku osiguravajuću kuću. Na taj način je šema baze podataka do izvesnog stepena normalizovana, mada ta normalizacija nije kompletna u meri u kojoj se to može postići u relacionoj bazi podataka.

3. Search engine XML dokumenata. U ovom slučaju, nisu pravljeni pokušaji da se nađe idealna veličina dokumenta već se svaki dokument jednostavno pamti u bazi podataka bez obzira da li je njegova veličina idealna.

Native XML baze podataka se mogu implementirati na jedan od sledećih načina Pamćenjem:

1. Celih XML dokumenata u CLOB strukturama relacionih baza podataka i indeksiranjem individualnih vrednosti elemenata i atributa
2. Parsiranih XML dokumenata u fiksnim skupovima tabela (elementi, atributi, tekst itd.) u relacionoj bazi podataka
3. Parsiranih XML dokumenata kao DOM stabla u objektno-orijentisanoj bazi podataka
4. Parsiranih XML dokumenata u indeksiranom skupu heš tabela

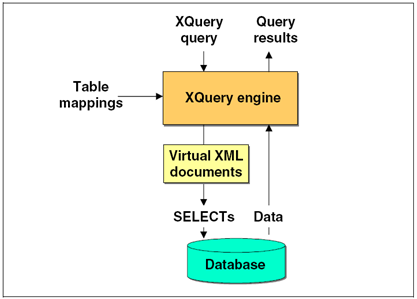
***Najznačajniji faktor koji utiče na performanse native XML baza podataka je da li dokument zapamćen u celini ili u parsiranoj formi.***

Da bi se sagledalo kako to utiče na performanse, mogu se razmotriti sledeće klase upita:

1. ***Upiti kojima se pretražuje čitav dokument a koji se baziraju na indeksiranim vrednostima.*** Na takve upite bolje performanse pruža baza podataka koja pamti ceo dokument, jer ne postoji potreba da se dokument ponovo kreira tj. da se izvrši njegova ponovna serijalizacija što je slučaj kada je dokument zapamćen u parsiranoj formi. Upiti kojima se pretražuju indeksirane vrednosti. Za datu ekvivalentnu šemu indeksiranja, ovakve upite izvršavaju jednako sve baze podataka, jer se oni mogu rešiti jednostavno pretraživanjem indeksa.
2. ***Upiti kojima se pretražuju neindeksirane vrednosti ili fragmenti dokumenata.*** Ovaj upit bolje izvršavaju baze podataka koje pamte prethodno parsirane dokumente jer baze koje pamte cele dokumente moraju da svaki dokument parsiraju kako bi ovakav upit rešile.
3. ***Upiti koji ažuriraju dokumenta.*** Ove upite najbolje izvršavaju baze podataka koje pamte prethodno parsirana dokumenta, jer one mogu da ažuriraju, insertuju ili brišu pojedine čvorove. Baze podataka koje pamte cela dokumenta ne moraju samo da parsiraju i modifikuju dokument; one moraju takođe da pretraže čitav dokument da bi ga zapamtile posle ažuriranja.

**XQuery**

**XQuery** je XML upitni jezik koji je definisan od strane W3C ikoji još uvek nije završen. Međutim, već postoji više od dvadeset implementacija Xquery-a.

On je dizajniran za upite nad XML dokumentima a ne relacionim bazama podataka. Da bi bio implementiran nad relacionom bazom, neophodno je prvo mapirati relacionu bazu na jedan ili više virtualnih dokumenata. Najlakši način da se to uradi je mapirati svaku tabelu u poseban dokument korišćenjem table-based mapiranja. Tako se XQuery naredba za upit nad virtualnim dokumentom može mapirati u SELECT naredbu za upit nad tabelom!

Primer jedne XQuery naredbe glasi:

FOR $c IN document("customers")/row/Customer

RETURN

<Customer>

<Name>{$c/name}</Name>

<ID>{$c/id}</ID>

</Customer>

XQuery koristi tzv. path izraze za navigaciju (prolazak) kroz elemente u XML dokumentu.

+++XQuery koristi predikate za ograničavanje izdvojenih elemenata iz XML dokumenata.

Sledeći predikat je iskorišćen za selektovanje svih elemenata book ispod elementa bookstore koji imaju element price sa vrednošću koja je manja od 30.

doc("books.xml")/bookstore/book[price<30]

***Najbolji način da se objasni XQuery jezik je reći da je XQuery za XML isto što je i SQL za baze podataka.*** XQuery je dizajniran da pravi upite nad XML-om ne samo nad XML fajlovima već svim podacima koji mogu biti predstavljeni XML-om pa čak i baze podataka, i poznat je još kao XML query. Dizajniran je tako da omogući lako manipulisanje metodama i upitima da bi se pristupilo podacima iz XML-a. Za učenje XQuery-ja potrebno je prethodno znanje HTML / XHTML, XML / XML namespace -a i XPath-a.

***XQuery je jezik za pronalaženje, ekstrakciju i manipulaciju elementima i atributima iz XML dokumenta.*** On svaki XML dokument definiše kao stablo čiji su čvorovi elementi u odgovarajućem XML-u. Dizajniran je tako da lako može da manipuliše kolekcijom dokumenata ili fragmenata dokumenata. XQuery je funkcionalan jezik gde je svaki upit predstavljen izrazom.

**NoSQL baze podataka**

Dok su relacione baze podataka prvobitno namenjene za smeštanje tabelarnih struktura, one se nisu najbolje pokazale pri pokušaju modeliranja ad-hoc relacija koje se pojavljuju u realnom svetu.

* Relacioni model može postati neefikasan u slučaju spajanja velikog broja tabela
* Povećanjem broja veza u relacionim bazama podataka znači i povećanje broja spajanja (join) između tabela što smanjuje performanse

Ulažući napor da se izađe na kraj sa velikim skupovima podataka , u svetu NoSQL-a je usvojeno nekoliko alternativa relacionim bazama podataka.

Četiri osnovne funkcionalnosti u kojima se ogledaju razlike između NoSQL baza podataka i standardnih SQL sistema su:

1. **Nepotrebnost šeme:** Šema baze podataka je opis svih mogućih podataka i njihovih tipova koji se u bazi mogu naći. NoSQL nije baziran na šemi, tako da šema nije neophodna. Korisnici ovim dobijaju fleksibilnost u načinu na koji strukturiraju podatke.

2. **Relacionalnost:** U relacionim bazama pravimo puno relacija između tabela u kojima čuvamo podatke. Na primer lista transakcija može da bude povezaan sa korisnicima. Sa NoSQL bazama ova informacija je sačuvana kao agregacija, jedinstveni slog koji sadrži sve u vezi transakcija, uključujući i podatke o korisniku.

3. **Jeftiniji hardver:** Neke baze su dizajnirane da rade samo na najboljim serverima, NoSQL baza je drugačija i omogućava korisnicima da korišćenjem jeftinijih servera i spajanjem više njih postignu izuzetne performanse.

4. **Visoka distribuiranost:** Distribuirane baze mogu da skladište podatke na više od jednog uređaja. Sa NoSQL-om, klaster servera može da sadrži jednu veliku bazu podataka. NoSQL proizvođači olakšavaju rad svojim korisnicima (programerima koji koriste NoSQL). Tehnologija može brzo biti prihvaćena samo ako je razvojni timovi vide kao jeftinu alternativu. U procesu razvoja se mogu brže i efikasnije prevazići postojeći problemi. Mogućnost prilagođavanja hardveru i brzo izbacivanje novih servisa su osnovne karakteristike NoSQL implementacija. Sa NoSQL je moguće mnogo brže primeniti iterativni razvoj i dodavanje novina u odnosu na DBMS. Primenom NoSQL principa se ne smanjuje samo cena i olakša život programerima već se rešava i niz menadžerskih problema.

Konzistentnost NoSQL baze podataka

Konzistentnost je atributbaze koji označava kakav pogledna podatke ima baza podataka tj. Da li svako čitanje posle upisivanja odmah prikazuje upisanu vrednosti podataka ili ne. +Visoka konzistentnost je potrebna kod proizvoda koji koriste transakcije.

Konzistentnost se često deli u dva nivoa:

1. **ACID konzistentnost**: predstavlja atomske transakcije i obično znači da će svaki upit nakon upisa videti izmenjene podatke.
2. **BASE konzistentnost**: znači da će upisani podaci pre ili kasnije početi da se prikazuju u upitima.

-Ne postoji jedinstveno mišljenje o tome koje je rešenje je bolje, istina je da nijedno nije uvek idealno i da je fleksibilnost u izboru rešenja jako bitna za razvoj visoko kvalitetnog softvera.

U svetu relacionih baza podataka smo familijarni sa **ACID transakcijama** kojima se garantuje sigurno okruženje u kojem se radi sa podacima. ACID je generalni skup principa za transakcione sisteme i nije vezan jedino za baze već se javlja i u puno drugih oblasti. **ACID** znači:

1. Atomičnost (Atomic): Sve operacije u transakciji moraju biti uspešne ili se transakcija mora poništiti (rolled back).
2. Konzistentnost (Consistent): Po završetku transakcije, baza podataka je u konzistentnom (ispravnom) stanju.
3. Izolovanost (Isolated): Akcije transakcija se sekvencijalno izvršavaju jedna nakon druge. Nije moguće nesekvencijalno, naizmenično izvršenje akcija transakcija.
4. Trajnost (Durable): Rezultat primene transakcija je trajan, osim ako ne postoji greška.

Ove osobine znače da kada se transakcija završi, njeni podaci su konzistentni i stabilni na disku. Ovo je dobro za one koji rade na razvoju aplikacija ali zahteva sofisticirano zaključavanje koje može prouzrokovati nedostupnost podataka i obično predstavlja težak uzorak (pattern) za većinu slučajeva korišćenja.

Umesto korišćenja ACID, pojavio se termin **BASE** kao popularan način za opisivanje svojstva optimističnije strategije čuvanja podataka. **BASE konzistentnost** značida baza nema ACID garancije. Ovo je dosta interesantno kod sistema koji su implementirani kao veliki klasteri, neki delovi sistema mogu da sadrže samo podatke za čitanja koji bivaju povremeno ažurirani. BASE ima sledeće značenje:

1. **Osnovna raspoloživost** (Basic availability): Ima se utisak da se sve vreme vrši pamćenje podataka
2. **Soft-state:** Pamćenje ne mora da bude konzistentno sa upisom niti različita ažuriranja moraju biti međusobno konzistentna sve vreme.
3. **Konačna konzistentnost** (Eventual consistency): Konzistentnost se javlja u nekom kasnijem trenutku

**BASE** svojstva su značajno labavija u odnosu na ACID garancije i između njih se ne može uspostaviti direktno preslikavanje. U BASE pamćenju podataka, podaci postaju raspoloživi ali se ne garantuje konzistentnost u trenutku upisa. BASE pamćenje ne obezbeđuje striktnu sigurnost, podaci postaju konzistentni kasnije, možda u vreme čitanja, ili će biti konzistentni ali samo za neke procese. Prednost BASE pristupa je što se transakcije sporije komituju, što znači i brže čitanje.

Mane ovoga su što klijenti koji se konektuju da čitaju replike podataka mogu da vide informacije koje više nisu validne. Ovakav sistem je primenjiv na socijalnim mrežama, ukoliko napravite post na facebook-u, a vaši prijatelji to vide tek kroz nekoliko minuta, gubitak nije veliki, a poboljšanja u performansama će biti velika.

Kao što možemo očekivati NoSQL proizvođači mogu različito da se odnose prema svojim bazama i da zauzmu karakterističan stav u određenim oblastima.

Mnoge NoSQL baze imaju kao cilj da koriste ACID, iako su trenutno bazirane na BASE sistemu, što pokazuje koliko je ACID garancija bitna za enterprise sisteme.

Mnoge kompanije koriste BASE konzistentne proizvode kada testiraju svoje ideje jer su besplatne a ACID konzistentnost koriste za produkcione sisteme za koje plaćaju.

Najlakši način za određivanje da li ćemo koristiti ACID je da pogledamo kako naši upisi i čitanja funkcionišu. Ukoliko čitanja sa kašnjenjem ne zadovoljavaju naše poslovne potrebe, mora se koristiti ACID konzistentnost.

U NoSQL bazama podataka korite se sledeći model podataka:

1. **čuvanje dokumenata po ID-u,**
2. **čuvanje ključeva-vrednosti (eng. Key-Value Stores),**
3. **čuvanje familija kolona (eng. Column Family)**
4. **graf baze podataka**

**Čuvanje dokumenata po ID-u:** Na najjednostavnijem nivou, dokumenti se mogu čuvati i pretraživati po ID. U opštem slučaju, pamćenje podataka se bazira na indeksima koji olakšavaju pristup dokumentima na osnovu njihovih atributa. Na primer, indeksi se mogu koristiti za predstavljanje različitih tipova proizvoda kako bi se nudili potencijalnim prodavcima.

Generalno, kod čuvanja dokumenata po ID-u u NoSQL bazama podataka indeksi se koriste da bi se pretražio skup povezanih dokumenata za potrebe neke aplikacije. Slično indeksima u relacionim bazama podataka, indeksi pri pamćenju dokumenata omogućavaju brže čitanje ali sporiji upis dokumenata.

**Čuvanje ključeva-vrednosti (Key-Value)** je sličan pamćenju familije dokumenata, i potiče sa Amazon’s Dynamo baze podataka. On deluju kao velika distribuirana hashmap struktura podataka koja čuva i pretražuje nerazumljive vrednosti po ključevima. Prostor sa ključevima hashmap-a se širi preko velikog broja cilindara (eng. buckets) na mreži. Zbog tolerancije na greške, svaki cilindar se replicira na nekoliko mašina.

Sa tačke gledišta korisnika, Key-Value način pamćenja je lak za korišćenje. Klijent čuva element podataka heširanjem nekog identifikatora koji je specifičan za domen problema (ključ). Heš funkcija je izrađena tako da omogućava podjednaku distribuciju kroz raspoložive cilindre omogućavajući tako da ni jedna mašine ne postane žarište. Pošto dobije ključ za heširanje, klijent može da koristi tu adresu da bi zapamtio vrednost u odgovarajućem cilindru. Sličan proces se koristi i pri pretraživanju zapamćenih podataka.

U ovakvom modelu, aplikacije koje žele da pamte podatke ili ih pretražuju, potrebno je da znaju (ili sračunaju) odgovarajući ključ. Kako postoji veoma veliki broj mogućih ključeva u skupu ključeva, u praksi postoji tendencija da to budu prirodi ključevi iz domena problema: Usernames i email adresa, socijalni broj itd.

**Čuvanje familija kolona (Column Family)**-Model podataka se bazira na retko popunjenim tabelama čiji redovi mogu da sadrže proizvoljne kolone, ključeve koji omogućavaju prirodno indeksiranje.

Četiri zajednička bloka za izgradnju (eng. building blocks) koja se koriste u Column Family bazama podataka:

1. ***Kolona*** (eng. Column)
2. ***Super kolona*** (Super Column) koja se dobija kombinacijom bilo kog broja kolona
3. ***Familija kolona*** (Column Family) koja se dobija kada se više kolone zapamti kao red tabele koji ima svoj ključ
4. ***Super familija kolona*** (Super column family) koja se dobija kada se više super kolona zapamti kao red tabele koji ima svoj ključ

Individualni redovi na taj način postaju važni u ovoj strukturi jer oni omogućavaju umreženu hashmap strukturu u okviru koje denormalizujemo naše podatke.

U bazama podataka zasnovanim na familiji kolona svaki red u tabeli predstavlja pojedinačan sveobuhvatni entitet (npr. sve o umetniku). Te familije kolona su kontejneri za povezane delove podataka, kao što je ime umetnika i dela.

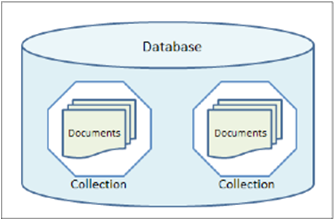
Nasuprot pamćenju podataka o kojem je do sada bilo reči, u **graf bazama podataka** (Graph Databases), povezani podaci se pamte na diskovima kao povezani.

Na primer:

U socijalnoj mreži, kao i u mnogim slučajevima iz realnog okruženja veze između entiteta ne prikazuju uniformnost na nivou domena već je domen **promenljivo struktuiran** (variably-structured). Socijalna mreža je popularan primer gusto povezane, promenljivo struktuirane mreže koja je rezidentna i odgovara svim mogućim šemama. Naša jednostavna šema prijatelja može da raste u veličini i postaje sve bogatija.

OSNOVNI ELEMENTI NoSQL BAZA PODATAKA

Osnovni elementi u NoSQL bazama podataka su:

1. **Database** - bazu čini fizički kontejner strukture koja se zove kolekcija. Svaka baza ima svoj sopstveni skup file-ova na file sistemu. Jedan DB server obično ima više baza na sebi.
2. **Kolekcija** - slična RDBMS tabelama. Imena kolekcija su unikatna. Obično su kolekcije koje se sadrže u istoj bazi povezane ali nije neophodno naznačiti kako su povezane, što predstavlja ključnu razliku u odnosu na RDBMS.
3. **Dokumenti** – Ovo je najjednostavnija jedinica podataka u NoSQL bazama podataka, obično se sastoji od skupa parova ključ-vrednosti. Za razliku od unosa u RDBMS, NoSQL dokumenti imaju dinamičku šemu. Dokumenti u jednoj kolekciji ne moraju da imaju uvek istu strukturu.

**JSON**

**JSON (JavaScript Object Notation)** je format za predstavljanje podataka, lak je za čitanje i pisanje, a s druge strane je lak i mašinama za generisanje i parsiranje. Baziran je na podskupu definicije JavaScrip standarada iz Decembra 1999. JSON je tekstualni format koji je kompletno nezavisan od programskog jezika, a koristi konvencije slične C-porodici jezika, uključujući C, C++, C#, Java, JavaScript, Perl, Python, i mnoge druge. Sve ove osobine čine JSON dobrim formatom za predstavljanje podataka. **JSON koristi dve strukture:**

1. Kolekciju parova ime/vrednost koje se u različitim jezicima mogu zvati objekti, rekordi, strukture, rečnici…
2. Raspoređenu listu vrednosti koja se često naziva niz, vector, list, ili sekvenca.

Ove univerzalne strukture podataka se mogu razlikovati od jezika do jezika. JSON je nastao kao zamena **za XML, ali podaci se mogu predstaviti na kraći način i lakše ih je za čitati.**

Primer XML-a: Primer JSON-a:

<employee>

<firstName>Bogdan</firstName> {"firstName":"Bogdan","lastName":"Jankovic"}

<lastName>Jankovic</lastName>

</employee>

Elementi NoSQL baze orijentisane na kolonama

Gledajući model podataka odozgo na dole, on se sastoji od sledećih elemenata:

1. Klastera
2. Keyspace-ova
3. Familije kolona
4. Kolona
5. Super kolona

**Familija kolona** je kontejner za uređenu kolekciju redova, od kojih svaki predstavlja uređenu kolekcija kolona. U relacionom svetu, kada se fizički kreira baza podataka iz modela, odredi se ime baze podataka (keyspaces), imena tabela (slično familiji kolona mada familije kolona ne treba izjednačava sa tabelama - one to nisu), a zatim definišu imena kolona koje će se nalaziti u svakoj tabeli.

Postoji nekoliko dobrih razloga da se familija kolona ne izjednačava sa tabelom u relacionoj bazi. Prvo, iako su definisane familije kolona, NoSQL baza ne poseduje šemu. Može se slobodno dodati bilo koja kolona u bilo koju familiju kolona u bilo koje vreme, u zavisnosti od potreba. Drugo, familija kolona ima dva atributa: ime i komparator (vrednost za upoređivanje). Vrednost komparatora definiše kako će se kolone sortirati u rezultatu upita - prema dužini, bajtu, UTF8 ili drugom redosledu.

U relacionoj bazi podataka korisniku je često transparentno kako se tabele čuvaju na disku, ali se modeliranje podataka ne vrši na osnovu toga kako RDBMS čuva podatke na disku. To je još jedan razlog koji treba imati na umu kada se kaže da familija kolona nije tabela. Pošto se svaka familija kolona čuva u zasebnoj datoteci na disku, važno je da kolone koje su u vezi budu definisane zajedno u istoj familiji.

Drugi način na koji se familije kolona razlikuju od relacionih tabela je taj što su kod relacionih tabela definisane samo kolone, a korisnik definiše vrednosti, kojima se popunjavaju redovi. U NoSQL bazi tabela može sadržati kolone ili se može definisati kao familija super kolona. Prednost upotrebe familije super kolona je mogućnost ugnježđenja.

Kada upisujete podatke u familiju kolona u NoSQL bazu podataka zadaje se vrednosti za jednu ili više kolona. Ta kolekcija vrednosti zajedno s jedinstvenim identifikatorom naziva se red. Red ima jedinstveni ključ, koji se zove ključ reda, koji deluje kao jedinstveni identifikator (primarni ključ) tog reda. Dakle, iako nije pogrešno nazvati ga orijentisanim na kolone, model se može lakše razumeti ako razmišljate o redovima kao kontejnerima za kolone.

**Klaster:** NoSQL model podataka je veoma skalabilan i i dizajniran tako da se može distribuirati na nekoliko mašina koje rade zajedno a koje se za krajnjeg korisnika pojavljuju kao jedinstvena instanca. U tom smislu, najudaljenija struktura je klaster, koji se ponekad naziva i prsten, jer se podaci dodeljuju čvorovima u klasteru tako što se raspoređuju u prstenove.

**Keyspace:** Klaster je kontejner za keyspace - obično za jedan keyspace. Keyspace je krajnji kontejner za podatke, a u potpunosti odgovara relacionoj bazi podataka. Kao i u relacionoj bazi podataka, keyspace ima naziv i skup atributa koji definišu ponašanje na nivou čitavog keyspace. Iako se često smatra da je za jednu aplikaciju potrebno kreirati jedinstveni keyspace, čini se da to nema mnogo praktičnih osnova. To je svakako prihvatljiva praksa, ali savršeno je u redu kreirati onoliko keyspace-ova koliko vašoj aplikaciji treba.

**Kolona** je najosnovnija jedinica strukture podataka u NoSQL modelupodataka orijentisanom na kolonama. Kolona je trojka sastavljena od imena, vrednosti i vremenskog žiga. Iako smo upoznati sa pojmom „kolone“ iz relacionog sveta, u ovom modelu ne treba razmišljati o kolonama na isti način.

Pre svega, prilikom dizajniranja relacione baze podataka struktura tabela se određuje unapred dodeljivanjem svih kolona; kasnije, kada se upisuju podaci, jednostavno se dodeljuju vrednosti za unapred definisanu strukturu. U NoSQL bazi, kolone se ne definišu unapred; definišu se samo familije kolona koje želite u keyspace-u i tada možete upisivati podatke bez definisanja kolona. To je zato što u NoSQL modelu klijenti daju imena svim kolonama. Ovim se postiže značajna fleksibilnost u načinu na koji aplikacija radi sa podacima i može joj se dozvoliti da se tokom vremena razvija.

Kada se tabela dizajnira u tradicionalnoj relacionoj bazi podataka, ona obično odgovara „tipovima entiteta“ ili skupom atributa koji opisuju određenu imenicu (hotel, korisnik, proizvod itd.). Ne razmišlja se mnogo o veličini samih redova, jer se o veličini reda ne može pregovarati nakon što odlučite koju imenicu predstavlja tabela. Međutim, kada radite sa NoSQL bazom, zapravo morate doneti odluku o veličini redova: oni mogu biti debeli ili mršavi, zavisno od broja kolona koji red sadrži!!!

**Širokired (Wide Rows)** označava red koji sadrži mnogo, mnogo (možda desetine hiljada ili čak miliona) kolona. Obično postoji mali broj redova sa toliko kolona. Suprotno tome, mogli biste imati redove bliže relacionom modelu, gde definišete manji broj kolona i koristiti mnogo različitih redova - to je mršavi model.

Debeli redovi obično sadrže automatski generisana imena (poput UUID-a ili vremenskih oznaka) i koriste se za čuvanje liste nekih stvari. Razmotrite na primer aplikaciju za monitoring: možete imati red koji predstavlja vremenski period, koristeći modifikovanu vremensku oznaku kao ključ reda, a zatim kolone koje predstavljaju IP adrese koje su pristupale vašoj aplikaciji u tom vremenskom periodu

Zatim možete da kreirate ključeve **tankih redova (Skinny Rows)** koji će ličiti na tradicionalne RDBMS redove, jer će svaki red sadržavati slične skupove imena kolona. Međutim, oni se razlikuju od redova RDBMS-a po tome što su sve kolone u osnovi neobavezne.

!!!Druga razlika između debelih i mršavih redova je ta što je samo kod debelih redova važan redosled u kojem treba sortirati imena kolona.

**Super kolona** je posebna vrsta kolone. Obe vrste kolona (regularne i super kolone) su parovi ime / vrednost, ali regularna kolona čuva vrednost u obliku niza byte-ova, dok je vrednost super kolone mapa pod kolona (koje čuvaju vrednosti niza bytova). Imajte na umu da one čuvaju samo mapu kolona; ne možete definisati super kolonu koja čuva mapu drugih super kolona. Dakle, ideja super kolona podrazumeva samo jedan nivo dubine, ali može imati neograničen broj kolona.

Osnovna struktura super kolone se sastoji od imena super kolone, koje je niz bayt-ova (isto kao i kod regularne kolone) i kolone u koju se vrši pamćenje. Njene kolone su mapa čiji su ključevi imena kolona i čije su vrednosti kolone.

Da bi se koristila super kolona, familiju kolona treba definisati kao tip Super. I dalje treba imati ključeve redova kao u regularnoj familiji kolona, ali i super kolonu koja upućuje na listu ili mapu regularnih kolona (koje se nekad nazivaju i pod kolone)!!!

***NoSQL baze kao rešenje problema upravljanja velikim bazama podataka***

Relacione baze podataka mogu stvoriti probleme kada je u pitanju skaliranje (obrada velike količine podataka).

Jedan od načina za skaliranje relacionih baza podataka je primena deljivosti (sharding) (pristup je korišćen na eBay, koja podržava milijarde SQL upita dnevno i u drugim Veb 2.0 aplikacijama).

Ideja je da se podaci podele tako da umesto da se hostuju na jednom serveru ili kopiraju na svim serverima u klasteru, dele horizontalno i hostuju se odvojeno.

**Primer:** tabela kupaca - kada tabela dostigne desetine miliona redova, verovatno ćete morati da razmišljate o tome da dodate više mašina. Da li samo treba kopirati podatke tako da ih sve mašine imaju ili podelite jednu tabelu klijenta tako da svaka baza podataka ima samo neke zapise?

Najjednostavnije za sve programere je vertikalno skaliranje dodavanjem CPU-a, memorije i bržih hard diskova, ali ako firma i dalje uspešno posluje i njen broj kupaca raste, u nekom trenutku (možda kada tabela dostigne desetine miliona redova), ćete verovatno morati da počnete da razmišljate o tome da dodate više mašina. Ili, umesto toga, podelite jednu tabelu klijenta tako da svaka baza podataka ima samo neke zapise, sa sačuvanim redosledom? Kada klijenti izvršavaju upite, opterećuju samo mašinu na kojoj se nalazi zapis koji traže, ne opterećujući ostale mašine. Čini se da je jasno da prilikom deljenja zapisa treba pronaći dobar ključ pomoću kojeg možete urediti svoje zapise.

Postoje tri osnovne strategije za definisanje strukture prilikom deljenja:

**1. Deljenje zasnovano na karakteristikama ili funkcionalna segmentacija**

Pomoću ove strategije podaci se dele ne deljenjem zapisa jedne tabele (kao u prethodnom primeru kupaca), već razdvajanjem u odvojene baze podataka koje se međusobno ne preklapaju. Na primer, na eBay su korisnici u jednom delu, a predmeti za prodaju u drugom. Na Flixster -u su ocene filmova u jednom delu, a komentari u drugom. Dobra segmentacija podataka može da se izvrši samo na osnovu jasnog razumevanja domena.

**2. Deljenje na osnovu ključa**

U ovom pristupu u svojim podacima nalazite ključ na osnovu kojeg će se podaci ravnomerno rasporediti po delovima. Dakle, umesto da jednostavno čuvate jedno slovo abecede na svakom serveru kao u prethodnom primeru (naivno i nepravilno), koristite jednosmerni hash nad ključem podataka i distribuirate podatke na mašinama u skladu sa hash-om. U ovoj je strategiji uobičajeno koristiti vremenski bazirane ili numeričke ključeve za hešing.

**3. Tabela pretraživanja (Lookup table)**

U ovom pristupu, jedan od čvorova u klasteru deluje kao direktorijum "žutih stranica" i traži koji čvor ima podatke kojima pokušavate da pristupite. On ima dva očita nedostatka. Prvi je da ćete pogodak ostvariti samo ako prođete kroz tabelu pretraživanja kao dodatni korak. Drugi je da tabela pretraživanja ne pretstavlja samo usko grlo, već i jednu tačku neuspeha.

+++Deljenjem se u zavisnosti od izabrane strategije mogu smanjiti problemi i omogućiti ne samo horizontalno već i preciznije skaliranje, jer se snaga može dodati samo određenim delovima kojima je potrebna.

**!!!Arhitektura kod koje se ništa ne deli:** ne postoji centralizovano stanje, a svaki čvor u distribuiranom sistemu je nezavisan, tako da ne postoji potreba za deljenjem resursa.

Ovu arhitekturu je nedavno popularizovao Google, koji je kreirao sisteme poput Bigtable baze podataka i implementirao je kao MapReduce koji je u stanju da vrši beskrajno skaliranje. Open source rešenje **Cassandra** za upravljanje NoSQL bazama podataka podržava arhitekturu kod koje se ne deli ništa; ona nema centralni kontroler i ne podržava pojma master / slave; svi njeni čvorovi su isti.