

|  |  |
| --- | --- |
|  | UNIVERSITY OF NIŠ FACULTY OF ELECTRONIC ENGINEERING |

Fizičko projektovanje MySQL baze podataka i optimizacija podataka

Seminarski rad

|  |  |
| --- | --- |
| Student: | Mentor: |
| Ivan Šušter, br. ind. 1548 | Doc. dr Aleksandar Stanimirović |

Niš, Januar 2024.

SADRŽAJ

[1. UVOD 1](#_Toc162275344)

[2. RADNO OPTEREĆENJE 2](#_Toc162275345)

[3. INDEKSI 2](#_Toc162275346)

[3.1. Kreiranje indeksa 4](#_Toc162275347)

[4. FIZIČKO PROJEKTOVANJE BAZE PODATAKA 6](#_Toc162275348)

[4.1. Denormalizacija 6](#_Toc162275349)

[4.1.1. Tip denormalizacije veza jedan prema jedan 7](#_Toc162275350)

[4.1.2. Tip denormalizacije veza jedan prema više 8](#_Toc162275351)

[4.2. Particionisanje 9](#_Toc162275352)

[4.2.1. Vertikalno particionisanje 10](#_Toc162275353)

[4.2.2. Horizontalno particionisanje 13](#_Toc162275354)

[4.2.3. Horizontalno particionisanje u MySQL bazi podataka 13](#_Toc162275355)

[4.3. InnoDB buffer pool 19](#_Toc162275356)

[5. ZAKLJUČAK 21](#_Toc162275357)

[LITERATURA 22](#_Toc162275358)

# 1. UVOD

U svetu baza podataka, ključnu ulogu u osiguravanju efikasnosti i brzine pristupa podacima zauzima fizičko projektovanje baze podataka. Optimalan rad baze podataka zahteva duboko razumevanje konteksta i pravilnu implementaciju njenih fizičkih aspekata. Proces fizičkog projektovanja baze podataka započinje nakon kreiranja konceptualnih i eksternih šema. Zbog dinamičke prirode poslovnih zahteva i tehnološkog napretka, fizičko projektovanje nije statičan proces, već zahteva kontinuiranu reviziju i adaptaciju, osiguravajući tako da se efikasnost i performanse baze podataka stalno poboljšavaju u skladu sa rastućim potrebama sistema koje baza podataka podržava.

U ovom radu istaknuta je važnost predviđanja i odabira pravih indeksa, koji su ključni za ubrzanje obrade upita i predstavljaju temelj fizičkog projektovanja baze podataka. Jednako bitna komponenta tog procesa jeste analiza radnog opterećenja baze podataka, jer upravo radno opterećenje omogućava efikasno usmeravanje i optimizaciju kroz sagledavanje i razumevanje koje upite sistema baza najčešće koristi, kao i koliko često se ti upiti izvršavaju nad određenim kolonama.

Osim indeksiranja, postoje i druge metode i tehnike ključne za optimizaciju performansi baze podataka. Horizontalno i vertikalno particionisanje tabele predstavljaju strategije koje bazi omogućavaju efikasnije upravljanje velikim količinama podataka, tako što ih dele na manje, jednostavnije za upravljanje segmente. Denormalizacija, iako na prvi pogled može delovati kao korak unazad u odnosu na norme dizajna baza podataka, u određenim situacijama može značajno poboljšati performanse čitanja podataka. Te strategije i metode su neophodne za postizanje optimalnih performansi, a detaljnije su predstavljene u ovom radu.

# 2. RADNO OPTEREĆENJE

Radno opterećenje (engl. "workload") u kontekstu baza podataka odnosi se na ukupan obim upita, transakcija i operacija koje sistem baze podataka obrađuje u određenom vremenskom period [1-2]. To uključuje sve tipove pristupa i interakcija sa bazom podataka, kao što su čitanje, pisanje, ažuriranje i brisanje podataka, koji su inicirani od strane korisnika ili aplikacija. Definicija radnog opterećenja omogućava administratorima baza podataka i inženjerima za performanse da analiziraju i razumeju kako se resursi baze koriste, što je ključno za optimizaciju performansi, kapaciteta i pouzdanosti sistema. Analiza radnog opterećenja pomaže u identifikovanju najzahtevnijih operacija i potencijalnih uskih grla u performansama, što je neophodno za efikasno planiranje kapaciteta, skaliranje i primenu odgovarajućih optimizacija [3-4].

Opis i analiza radnog opterećenja obuhvata sledeće: [5]

* Listu upita i koliko često se koriste u odnosu na sve upite i ažuriranja
* Listu ažuriranja i njihove učestalosti
* Ciljane performanse za svaki tip upita ili ažuriranja.

Za svaki upit je potrebno odreiti:

* Kojim se relacijama pristupa
* Koji se atributi traže u SELECT iskazu
* Nad kojim atributima se vrši spajanje ili selekcija u WHERE iskazu i koliko su ti uslovi selektivni.

Za svako ažuriranje se mora odrediti:

* Nad kojim atributima se vrši spajanje ili selekcija u WHERE iskazu i koliko su ti uslovi selektivni
* Tip ažuriranja (INSER, DELETE, UPDATE) i relacija koja se ažurira
* Za UPDATE komandu, polje koje treba ažurirati.

Opis upita je veoma bitan kako bi mogao da se izvrši adekvatano fizičko projektovanje baze podataka, odnosno veoma je bitno odlučiti kada će se koristiti indeksi i koji upiti mogu dobiti na brzini prilikom izvršavanja, što znači da i prilikom izvršavanja nekih drugih upita može dovesti do povećanja vremena za izvršavanje što može biti slučaj kod naredbi ažuriranja.

# 3. INDEKSI

Indeksi predstavljalju strukture podataka koje omogućavaju efikasan i brz pristup podacima koji se nalaze u bazi podataka. Osnovni razlozi upotrebe indeksa su [6]:

* Bolje performanse: indeksi predstavljalju odnosno omogućavaju brz pristup podacima.
* Jedinstvenost podataka: indeksi osiguravaju da se vrednosti kolona koje su indeksirane ne ponavljalju. Tako da je se u bazama najčešće automatski kreira indeks za sve primarne ključeve.

Prilikom fizičkog projektovanja baze podataka veoma je bitan izbor indeksa, odnosno bitno je doneti odluku koje idndekse napraviti i nad kojim kolonama. Performanse zavise od indeksa koji se koriste kao i od toga nad kojim poljima i/ili relacijama se koriste. Takođe je potrebno odrediti da li indeks treba biti klasterovan, redak ili gust.

Još jedna definicija indeksa bi bila da su oni struktura podataka koja ubrzava pretraživanje tabela u bazi podataka, odnosno nije potrebno prolaziti kroz svaki red tabele svaki put pirlikom pirstupa bazi podataka.

Pored odabira indeksa treba razmotriti i denormalizaciju šeme baze podataka ili koju od normalnih formi iskoristiti za projektovanje baze podataka.

MySQL sistem za upravljanje bazama podataka, koristi različite tipove indeksa koji pružaju optimizaciju upita i podržavaju različite tipove aplikacija. Tipovi indeksa koji se koriste u MySQL sistemu za upravljanje bazom podataka su [7]:

* B Stabla (engl. *B-Tree*) indeks - se obično koriste da bi se olakšalo efikasno pronalažanje podataka i obrada upita. Koristi se kod operatora =, >, >=, <, <=, BETWEEN i LIKE. Može se koristiti u MyISAM, MEMORY i InnoDB *storage engine-u.*
* Full-Text Indeks - MySQL podržava full-text indeks koji je neophodan kako bi se omogućio efikasnost operacija pretraživanja zasnovanih na tekstu. Full-text indeks je koristan u aplikacijama koje zahtevaju napredne mogućnosti pretraživanja teksta, odnosno koje uključuju velike tekstualne skupove podataka. Koristi se kod InnoDB i MyISAM *storage engine-a.*
* Prostorni (engl. *Spatial*) Indeks - ovi indeksi su neophodni za aplikacije koje uključuju geografske ili prostorne podatke. Prostrni indeksi omogućavaju efikasnu obradu prostornih upita, kao što su pretraga blizine i prostorna spajanja, što ih čini ključnim za aplikacije koje koriste geografske objekte. InnoDB i MyISAM koriste B stablo indekse za prostorne podatke, dok ostali *storage engine*-i kriste R stablo.
* Hash indeks - MySQL takođe podržava hash indekse, koji su najefikasniji za traženje tačnog podatka odnosno prilikom korišćenja znaka jednakosti ili <=> WHERE delu upita, jer obezbeđuju brz pristup zapisima podataka. Koristi ih MEMORY *storage engine*.

Svi ovi tipovi indeksa navedeni u MySQL sistemu za upravljanje bazama podataka zadovoljavaju različite zahteve upita i karakteristike podataka, omogućavaju efikasnu obradu upita i pronalaženje podataka u bazi.

U nastavku su dati primeri kreiranja idenksa u MySQL-u. Opšti primer kreiranja indeksa u MySQL-u je dat na primeru ispod, pored ovog na sličan nač.

CREATE INDEX index\_name

ON table\_name (column\_name);

Pored prehodno navednih tipova indeksa, takođe je moguće kreirate kompozitne indekse (indeks od više kolona), koji se sastoje od više kolona (do 16). Kod ovih indeksa je veoma bitan redosled navođenja kolona prilikom kreiranja indeksa kako bi se osiguralo da će indeks biti iskorišćen tako npr. u sledećoj naredbi:

CREATE TABLE zaposleni1(

    jmbg NUMERIC PRIMARY KEY,

    ime VARCHAR(20) NOT NULL,

    prezime VARCHAR(30) NOT NULL,

    adresa VARCHAR(50),

    datum\_rodjenja DATE,

    INDEX ime\_prezime\_idx (ime, prezime)

);

Ovaj indeks će biti iskorišćen samo ako se za pretragu koristi kolona ime, i kolona prezime MySQL će iskoristi indeks ali ako se koristi samo kolona prezime onda indeks neće biti korišćen. Odnosno da bi se iskoristio indeks prilikom pretrage potrebno je da se uvek pretražuje po koloni ime ili po kolonama ime i prezime.

Tako ako je napravljen kompozitni indeks nad 3 kolone, kol1, kol2, kol3, indeks će biti iskorišćen ako se upit izvršava nad ko1, kol1 i kol2, kol1, kol2 i kol3.

Hash indeks se kreira pomoću naredbe prizane na primeru ispod, ono što je bitno ponoviti je da radi samo na tabelama koje koriste MEMORY *storage* *engine.*

CREATE INDEX idx\_hash\_test ON zaposleni1 (adresa) USING HASH;

Prostorni indeksi takođe da bi funkcionisali potrebno je imati kolonu (kolona u primeru) koja je tipa *spatial* kao što su *point, line* i *polygon,* a primer kreiranja dat je u nastavku.

CREATE SPATIAL INDEX idx\_spatial ON zaposleni1 (kolona);

Full-Text indeks se koristi za pretragu teksta. Odnosno ovaj indeks je kroistan kada je potrebno pretražiti tesks na osnovu neke specifične reči ili fraze u koloni koja sadrži duži tekst.

CREATE FULLTEXT INDEX idx\_fullt ON zaposleni1 (kolona);

Pri odabiru tipa indeksa za MySQL bazu podataka, ključno je pažljivo proceniti slučaj upotrebe i upite koje aplikacija često izvršava. Različite vrste indeksa, poput gore navednih, sa više kolona pružaju različite prednosti na osnovu upita koji se obrađuju na primer ukoliko je potrebno često izvršavati upite sa više kolona i zna se da će uvek biti uključena prva kolona koja se nalazi u kompozitnom indeksu.

Razumevanjem jedinstvenih zahteva aplikacije, može se odabati najprikladniji tip indeksa kako bi se unapredila efikasnost obrade upita. Testiranjem i praćenjem različitih vrsta indeksa i njihovog uticaja na preformanse, mogu se identifikovati indeksi koji najbolje odgovaraju bazi podataka.

## 3.1. Kreiranje indeksa

Prilikom kreiranja indeksa potrebno je obratiti pažnju na upite koje je potrebno izvršiti. Tako na primer ako se u WHERE delu vrši izdvajanje po nekom opsegu takvi upiti najviše dobijaju na performansama ako se koristi B+ stablo. Ako se u WHERE delu izdvajanje vrši pomoću znaka jednakosti tada se najveći benefit dobija korišćenjem Hash indeksa.

Pre dodavanja indeksa takođe je potrebno obratiti pažnju na uticaj istih kada se vrši ažuriranje kolone koja ima indeks tada je potrebno ažurirati i indekse. Takođe i ako se vrši INSERT ili DELETE je potrebno ažurirati indekse ako se vrednost atributa promeni.

U narednim uputstvima biće objašnjen način odabira indeksa [1; 7]:

1. Ne treba praviti indeks ukoliko on neće doprineti ubrzanju izvršavanja nekog upita, uključujući i one koji su deo ažuriranja. Kada god je moguće birajte indekse koji će ubržati više od jednog upita.
2. Atributi koji se pominju u WHERE klauzuli su kandidati za indeksiranje.

* Izdvajanje po tačnoj vrednosti sugeriše da treba razmotriti indeks nad izabbranim atributima i to hash indeks.
* Izdvajanje po opsegu vrednosti sugeriše da treba razmotriti B+ stablo nad izdvojenim atributima.

1. Ključ pretrage sa više atributa treba ga razmotriti u sledećim situacijama:

* WHERE klauzula sadrži uslov nad više od jednog atributa.
* Ako omogućavaju index-only strategije (tamo gde se pristup relacijama može izbeći) u važnim upitima. Ovo može voditi ka atributima koji bi bili deo ključa iako se ne pominju u WHERE klauzuli. Pri kreiranju indeksa nad ključevima pretrage sa više atributa, ukoliko se prihvataju i intervalni upiti treba biti pažljiv oko redosleda atributa u ključu pretrage.

1. Najviše jedan indeks nad datom relacijom može biti klasterovan, što veoma povećava performanse. Tako da je potrebno pažljivo odabrati klasterovan indeks.

* Upiti koji sadrže opseg mogu mnogo dobiti klasterovanim indeksom. Ukoliko se upotrebljava nekoliko intervalnih upita nad relacijom, koji uključuju drugačije skupove atributa, pri odabiru koji indeks treba biti klasterovan, treba raymotriti selektivnost upita i njihove relativne učestalosti u radnom opterećenju.
* Ukoliko indeks omogućava index-only strategiju za upit koji treba da ubrza tada nije potrebno da bude klasterovan. Klasterovanje ima uticaja samo kod dobavljanja podataka.

1. Uglavnom se preporučuje B+ stablo, zato što je dobro i za upite intervala i za upite sa tačnom vrednosti. Hash indeks je bolji u sledećim situacijama:

* Hash indeks bi trebalo da se koristi kod ugnježdene petlje spajanja, indeksirane releacije unutrašnjih relacija, gde ključ pretrage uključuje kolone po kojima se vrši spajanje. Razlog tome je taj što se izdvajanje po jednakosti vrši za svaku n-torku u spoljnoj petlji.
* Kada postoje veoma važni upiti jednakosti, a pri tome ne postoje upiti intervala, uključujući atribude koji su ključ pretrage.

1. Nakon sastavljanja liste željenih indeksa treba razmotriti njihov uticaj na ažuriranja u radnom opterećenju.

* Ukoliko održavanje indeksa usporava operaciju ažuriranja, razmotriti njegovo odbacivanje.
* Treba imati na umu da dodavanje indeksa može ubrzati i data ažuriranja.

# 4. FIZIČKO PROJEKTOVANJE BAZE PODATAKA

Potreba za fizičkim projektovanjem baze podataka javlja se nakon logičkog projektovanja baze podataka kako bi se optimizovao rad baze podataka. Fizičko projektovanje baze podataka uključuje donošenje odluka o tome kako će se čuvati podaci u bazi kao i izboru metoda koje će omogućiti pristup tim podacima.

Nakon incijalnog fizičkog projekotvanja baze podataka potrebno je nastaviti sa nadgledanjem performansi baze prilikom pristupa odnosno manipulacijom podataka. Nadgledanje i praćenje rada sa podacima nad bazom podataka omogućava da se stekene uvid u to koje upite je potrebno optimizovati kako bi se još više poboljšale performanse baze podataka. Ukoliko je potrebno tokom rada, moguće je izvršiti nova podešavanja kao što su dodavanje novih indeksa ili neke od tehnika particionisanja kako bi se poboljšale performanse. Zato je veoma važno pratiti i analizirati statistiku i performanse izvršavanja upita nad podacima u bazi podataka.

Prilikom fizičkog projektovanja baze podataka, nekoliko aspekata treba pažljivo razmotriti kako bi se obezbedilo efikasno projektovanje. Ova razmatranja su neophodna za optimizaciju performansi, upravljanje radnim opterećenjem i prilagođavanje promenljivim zahtevima. Tokom fizičkog projektovanja treba obratiti pažnju na sledeće ključne aspekte [8-10]:

1. Fleksibilnost i budući zahtevi - projektovanje baze podataka treba da predstavi trenutno stanje ali i da bude fleksibilno kako bi se prilagodilo budućim zahtevima.
2. Konceptualni, logički i fizički dizajn - faza fizičkog projektovanja uključuje konceptualni dizajn, logički i fizički dizajn od koji svaki igra ključnu ulogu u oblikovanju strukture baze podataka.
3. Performanse upita i radno opterećenje - Fizičko projektovanje značajno utiče na preformanse upita.
4. Kontinuirano praćenje i prilagođavanje - Kontinuirano praćenje i unapređenje fizičkog projekotvanja i korišćenje indeksa je potrebno kako bi se optimizovale performanse baze podataka.

## 4.1. Denormalizacija

U arhitekturi baze podataka, denormalizacija predstavlja dodavanje redundantnosti šemi baze podataka sa ciljem poboljšanja performansi upita i pojednostavljenja dobijanja potrebnih podataka. Ova metodologija podrazumeva odstupanje od pravila normalizacije baze podataka kako bi se dobilo na performansama prilikom pristupa podacima koji su od značaja [11].

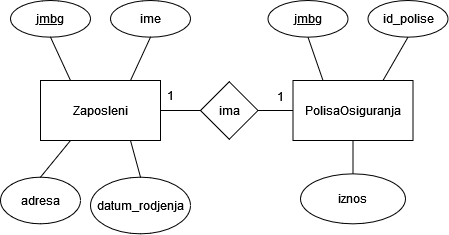
Denormalizacija spada u fižičko projektovanje baze podataka na logičkom nivou, pri čemu se prilikom denormalizacije kao početna tačka uzima normalizovani konceputalni model baze podataka. Nad potpuno normalizovanim modelom se zatim vrši denormalizacija, pri čemu se narušavaju normalne forme, uvodi redundansa podataka i narušava integritet podataka [12].

Važno je napomenuti da se uvođenjem redundanse i narušavanje integriteta podataka, neophodno je pomoću mehanizma trigera nadomestiti posledice denormalizacije odnosno očuvati integritet podataka. U nastavku su dati tipovi denormalizacije kao i primeri kako bi to izgledalo kao i kreiranje trigera za očuvanje integriteta podataka.

Dva tipa denormalizacije koje postoje su denormalizacija dva entitea u relaciji jedan prema jedan i denormalizacija dva entiteta u relaciji jedan prema više [12].

### 4.1.1. Tip denormalizacije veza jedan prema jedan

Ako uzmemo u obzir da postoje entiteti *zaposleni* i *polisaOsiguranja,* denormalizacijom možemo spojiti ta dva entiteta u jedan kako bi izbegli često spajanje tabela u upitima.



Slika 1 ER dijagram 1 prema 1 normalizovan

zaposleni(jmbg, ime, datum\_rodjenja, adresa)

polisaOsiguranja(jmbg, id\_polise, iznos)

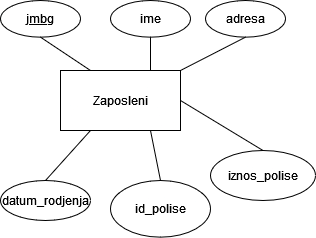
Funkcionalne zavisnosti ovih entiteta su:

zaposleni: jmbg -> ime, datum\_rodjenja

polisaOsiguranja: jmbg -> id\_polise, iznos

Denormalizacija ovih tabela predstavlja pristup prilikom kog je potrebno spojiti atribute ove dve tabele u jednu novu kombinovanu tabelu, spajajući ih na osnovu primarnog ključa. Ovaj proces kao što je već navedeno ima za cilj da optimizuje performanse upita i pojednostavi pristup podacima tako što će ukloniti potrebe korišćenja spajanja tabela u upitima. Kreiranje nove kombinovane tabele, sve informacije su lako dostupne u okviru jedne tabele, čime se povećava efikasnost baze podataka.

zaposleni(jmbg, ime, datum\_rodjenja, adresa, id\_polise, iznos\_polise)



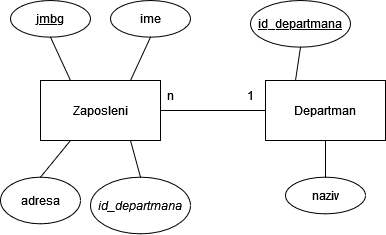
Slika 2 Denormalizovani primer veze jedan prema jedan

S obzirom da je reč o vezi jedan prema jedan, integritet podataka ostaje isti i nova tabela zaposleni osatje u 3NF.

### 4.1.2. Tip denormalizacije veza jedan prema više

U slučaju veze jedan prema više, takođe je moguće izvršiti denormalizaciju spajanjem tabela i time otkloniti potrebe za korišćenje spojeva prilikom izvršavanja upita, najveća efikasnost se postiže ukoliko denormalizaciju izvršimo nad tabelama kod kojih se često vrši spoj i upiti nad tim tabelama.

U sledećem primeru dat je tip veze jedan prema više i primer denormalizacije.



Slika 3 Primer jedan prema vise normalizovan

zaposleni(jmbg, ime, adresa, *id\_departmana*)

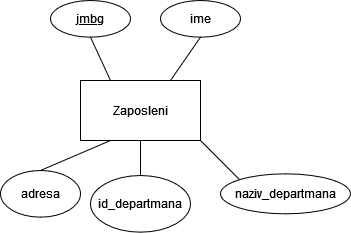
departman(id\_departmana, naziv)

Funkcionalne zavisnosti ove dve tabele su:

zaposleni: jmbg -> ime, adresa

departman: id\_departmana -> naziv

Denormalizacija će biti izvršena tako što će tabele *zaposleni* i *departman* biti spojene u jednu koja će se zvati *zaposleni* čiji je primer dat u nastavku, i sadržaće i podatke o departmanu. Na ovaj način se uspešno otklanja potreba za korišćenjem spojeva prilikom izvršavanja upita.



Slika 4 Primer denormalizacije veze jedan prema vise

zaposleni(jmbg, ime, adresa, id\_departmana, nazivDepartmana)

Ukoliko bi se ovakva tabela koristila dolazilo bi do mogućih problema prilikom brisanja nekog reda u tabeli jer bi se izgubili podaci o broju i nazivu departmana.

Denormalizaciju, posebno tipa jedan prema više koja obuhvata spajanje tabela radi poboljšanja upita, treba pristupiti sa oprezom i koristiti je za specifične slučajeve, odnosno kada je vreme izvršavanja upita nezadovoljavajuće i potrebno je skratiti vreme upita.

Implementaciju denormalizacije u svrhu optimizacije performanse upita treba dobro razmotriti, iako može da ubrza pronalaženje podataka, dovodi do potencijalnih složenosti u održavanju integriteta podataka.

U situacijama u kojima je vreme izvršavanja upita prihvatljivo, sprovođenje denormalizacije nije preporučljivo. Problemi koji su povezani sa denormalizacijom kao što su povećani zahtevi za skladištenjem i redundantnosti podataka mogu prevazići prednosti koje se dobijaju denormalizacijom.

Takođe denormalizacija može zakomplikovati upravljanje podacima odnosno, izmenu, dodavanje i brisanje podataka. Zbog toga je neophodno pažljivo proceniti prednosti i mane, kao i razmotriti neke druge dostupne strategije fizičkog projektovanja baze podataka kao što su indeksiranje. Samo u slučajevima kada su ostali pristupi neadekvatni, moguće je uzeti u obzir denormalizaciju kao potencijalno rešenje i sa opreznošću o uticaju na integritet podataka.

## 4.2. Particionisanje

Particionisanje tabele je tehnika koja se koristi za optimizaciju performansi baze podataka, deljenjem jedne tabele na dve ili više manjih tabela. Particionisanje ima za cilj da smanji količinu podataka koje sistem za upravljanje bazom podataka treba da obradi prilikom izvršavanja upita. Podelom tabela u više manjih tabela, sistem može efikasnije da upotrebi resurse, što rezultira bržim vremenom obrade upita [13].

Postoje dve metode particionisanja:

* Vertikalno particionisanje i
* Horizontalno particionisanje

Obe metode nude različite prednosti i pogodne su za rešavanje različitih problema.

I vertikalno i horizontalno particionisanje nude prednosti u pogledu preformansi i optimizacije upita. Izbor između njih zavisi od faktora kao što su priroda podataka, karakteristike upita i zahtevi za skalabilnost sistema baze podataka. Pažljivom procenom ovih faktora, može se primeniti particionisanje kako bi se poboljšale performanse sistema.

### 4.2.1. Vertikalno particionisanje

Vertikalno particionisanje tabele uključuje podelu kolona posmatrane tabele na dve ili više tabela, od kojih svaka sadrži kolone na osnovu njihove učestalosti korišćenja u upitima. Ovaj proces optimizuje pronalaženje podataka razdvajanjem kolona kojima se često pristupa u jednu tabelu i kolonama kojima se ređe pristupa u drugu tabelu. Podela može da se nastavi i dalje tj. na još tabela, sa kolonama grupisanim prema upitima koji se koriste. U svakoj tabeli koja je rezultat vertikalnog particionisanja, uključen je i primarni ključ prvobitne tabele. Ovaj primarni ključ služi kao veza koja povezuje redove u particionisanim tabelama.

Pomoću primarnog kljča u svakoj particionisanoj tabeli olakšava se manipulacija podacima u podeljenim tabelama. Ovaj pristup poboljšava performanse baze podataka smanjivanjem količine podataka koje je potrebno obraditi tokom izvršavanja upita. Upiti mogu biti takvi da pretražuju podatke samo u jednoj particiji koja sadrži relevantne kolone. Pored toga, vertikalno particionisanje može dovesti do bolje efikasnosti skladištenja podataka i korišćenja resursa, posebno u slučajevima gde se određene kolone veće ili se ređe koriste odnosno ređe im se pristupa u odnosu na druge kolone.

Primer tabele pre particionisanja:

zaposleni(jmbg, ime, adresa, datum\_rodjenja, id\_polise, iznos\_polise, polisa\_vazi\_od, polisa\_vazi\_do)

Izgled posle particionisanja:

zaposleniPolisaOsiguranja(jmbg, id\_polise, iznos, polise, polisa\_vazi\_od, polisa\_vazi\_do)

zaposleniPodaci(jmbg, ime, adresa, datum\_rodjenja)

U nastavku je dat primer kreiranja dobijenih tabela:

CREATE TABLE zaposleniPolisaOsiguranja(

    jmbg NUMERIC PRIMARY KEY,

    id\_polise NUMERIC UNIQUE NOT NULL,

    iznos\_polise NUMERIC,

    polisa\_vazi\_od DATE,

    polisa\_vazi\_do DATE

);

CREATE TABLE zaposleniPodaci(

    jmbg NUMERIC PRIMARY KEY,

    ime VARCHAR(50),

    adresa VARCHAR(50),

    datum\_rodjenja DATE

);

Nakon particionisanja tabela potrebno je definisati pogled (engl. *view*)koji podatke iz particionih tabela objedinjuje i tako kreira tabelu koja je identična početnoj tabeli. Kreiranjem pogleda skriva se vertikalno particionisanje i potreba za spajanjem dobijenih tabela kako bi se dobila početna tabela. Međutim da bi se obezbedio integritet i konzistentnost podataka, nad ovako kreiranim poglednom potrebno je definisati uskladištene procedure (ili ako je dostupno u DBMS-u, trigere) za ažuriranje, dodavanje i brisanje redova iz pogleda koji će reflekovati promene u svakoj particionisanoj tabeli.

CREATE VIEW zaposleni AS

    SELECT zo.jmbg, zo.id\_polise, zo.iznos\_polise, zo.polisa\_vazi\_od, zo.polisa\_vazi\_do,

            zp.ime, zp.adresa, zp.datum\_rodjenja

    FROM zaposleniPolisaOsiguranja zo, zaposleniPodaci zp

    WHERE zo.jmbg = zp.jmbg;

U nastavku je dat primer uskladištene procedure za dodavanje reda u pogledu, odnosno u particionisanim tabelama.

DELIMITER //

CREATE PROCEDURE insert\_zaposleni\_view (

    IN jmbg numeric,

    IN ime varchar(50),

    IN adresa varchar(50),

    IN datum\_rodjenja date,

    IN id\_polise numeric,

    IN iznos\_polise numeric,

    IN polisa\_vazi\_od date,

    IN polisa\_vazi\_do date

)

BEGIN

    INSERT INTO zaposlenipodaci VALUES (jmbg, ime, adresa, datum\_rodjenja);

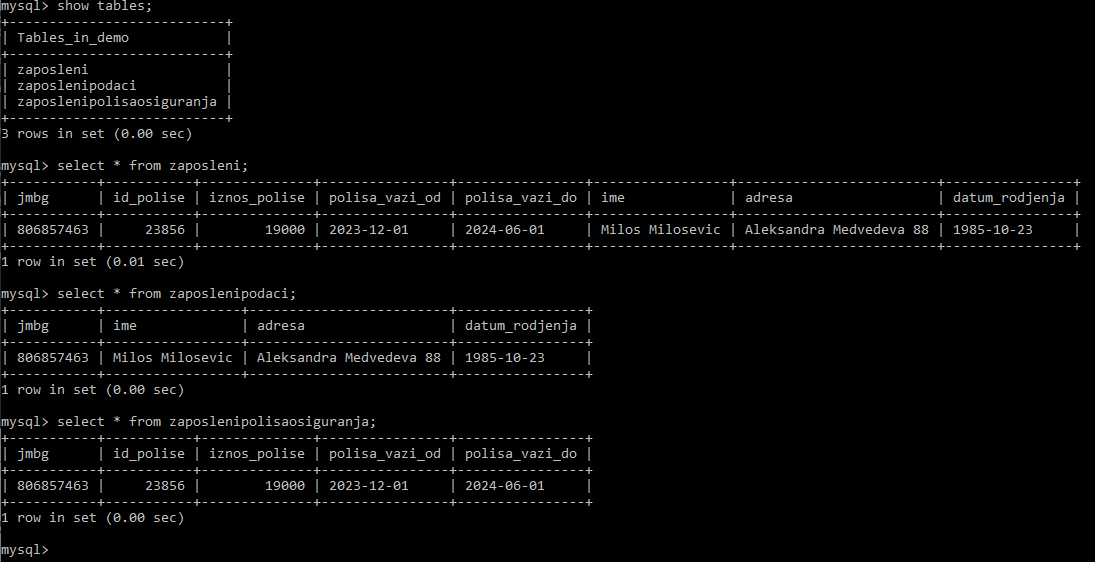
    INSERT INTO zaposlenipolisaosiguranja VALUES(jmbg, id\_polise, iznos\_polise, polisa\_vazi\_od, polisa\_vazi\_do);

END //

DELIMITER ;

Dodavanje reda tj. pozivanje uskladištene procedure se vrši pomoću ključne reči CALL i navode se redom vrednosti kako su definisane u proceduri iznad.

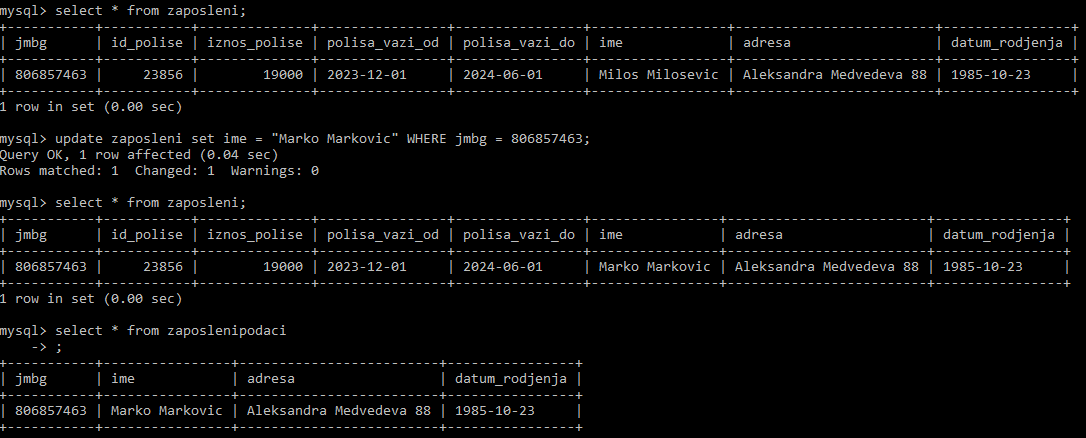
CALL insert\_zaposleni\_view(806857463, "Milos Milosevic", "Aleksandra Medvedeva 88", "1985-10-23", 23856, 19000, "2023-12-01", "2024-06-01");



Slika 5 Rezultat uskladištene procedure za dodavanje u view

U nastavku su date uskladištene procedure za ažuriranje odnosno menjanje kao i brisanje reda iz view-a.

Prilikom menjanja view-a u MySQL-u moguće je to uraditi klasičnom UPDATE klauzulom s obzirom da MySQL podržava menjanja view-a direktno i ako se on sastoji od više tabela tako da je u nastavku dat primer izmene samo imena zaposlenog nad pogledom *zaposleni,* a promena će se odraziti i na tabelu *zaposleniPodaci.*



Slika 6 Primer updatovanja view-a i prikaz rezultat u view-u i tabeli zaposleniPodaci

Uskladištena procedura za brisanje reda iz view-a na osnou primarnog ključa odnosno jmbg radnika.

DELIMITER //

CREATE PROCEDURE delete\_zaposleni\_view (

    IN v\_jmbg numeric

)

BEGIN

    DELETE FROM zaposlenipodaci

    WHERE jmbg = v\_jmbg;

    DELETE FROM zaposlenipolisaosiguranja

    WHERE jmbg = v\_jmbg;

END //

DELIMITER ;

Procedura se izvršava pomoću ključne reči CALL kao i kod dodavanja reda.

CALL delete\_zaposleni\_view(806857463);

MySQL sistem za upravljanje bazom podataka ne podržava vertikalno praticionisanje, odnosno podelu tabele prema kolonama na više manjih tabela zato je primer iznad jedini način kako je moguće trenutno izvršiti vertikalno particionisanje u MySQL-u.

### 4.2.2. Horizontalno particionisanje

Horizontalno particionisanje, takođe je poznato kao particionisanje zasnovano na redovima, je tehnika optimizacije baze podataka koja uključuje podelu tabele na manje particije na osnovu određenih kriterijuma. Ovaj metod je posebno koristan za tabele sa velikim brojem redova, jer pomaže u poboljšanju performansi upita.

Kada se primenjuje horizontalno particionisanje tabela se deli na različite particije od kojih svaka sadrži podskup redova koji zadovoljavaju neke uslove. Jedan uobičajeni kriterijum za particionisanje je vremski, gde se podaci dele na osnovu datuma, jedna particija može da obuhvata podatke za svaki mesec, praveći posebne particije za Januar, Februar, itd. Ovo vremensko particionisanje omogućava efikasno upravljanje podacima, pošto se upiti mogu odnositi na određene vremenske opsege bez pretraživanja cele tabele.

Takođe može se vršiti particionisanje podataka na osnovu opsega vrednosti unutra određene kolone. Na primer u tabeli faktura može se vršiti particionisanje na osnovu iznosa faktura. Pri čemu bi svaka partcijia sadržala podatke za određeni opseg iznosa fakture.

Koristeći horizontalno particionisanje mogu se efikasno podeliti podaci na više particija, čime se poboljšavaju performanse i skalabilnost baze podataka.

### 4.2.3. Horizontalno particionisanje u MySQL bazi podataka

U MySQL-u da bi se izvršilo horizontalno particionisanje poterebno je koristiti mašine za skladištenje (engl. *storage engine*) koje podržavaju horizontalno particionisanje. Ono što je bitno napomenuti je to da u trenutnoj verziji MySQL-a (verzija 8.0) sve particije particionisane tabele moraju koristiti isti *storage engine*.

U trenutnoj verziji MySQL-a jedini *storage engine* koji podržavaju particionisanje su *InnoDB* i *NDB (NDB cluster).* S tim što je particionisanje po ključu ili linearnom ključu (engl. *Key, Linear Key*) je moguće sa NDB storage engine-om, dok ostali tipovi nisu podržani za tabele koje koriste ovaj *engine* [14].

Prilikom particionisanja tabele, koristi se podrazumevani *storage engine.* Međutim za korišćenje nekog drugog *engine-a* potrebno je eksplicitno navesti koji se koristi u delu za kreiranje tabele, a pre bilo kojih opcija particionisanja. Ovo osigurava da se naznačeni *engine* primenjuje u celoj tabeli, ovo je binto s obzirom da je nemoguće particionisanje tabele ukolko se *storage engine* ne poklapaju.

CREATE TABLE zaposleni (

    jmbg NUMERIC,

    ime VARCHAR(50),

    datum\_rodjenja DATE

)

    ENGINE=INNODB

    PARTITION BY HASH( MONTH(datum\_rodjenja) )

    PARTITIONS 6;

Kada se upravlja podacima unutar particionsane tabele, rukovanje starim ili nepotrebnim podacima je relativno jednostavno. Uobičajeno je uklanjanje određene particije kao što može biti particija koja sadrži zastarele podatke. Ova metoda omogućava da se efikasno uklone nepotrbni podaci ostavljaljući ostatak tabale dostupnim.

Mogućnost skladjištenja podataka koji ispunjavaju određene kriterijume na jednoj ili više particija donosi značajne prednosti posebno u optimizaciji operacija pretraživanja. Prilikom pretrage sa nekim uslovom u WHERE klauzuli automatski se ograničava pretraga na relavantnu particiju i time se eliminiše potreba za skeniranjem nepotrebnih podataka.

Paritcionisanje tabela takođe omogućava prilagođavanje šeme particionisanja po potrebi. To znači da se podaci mogu reorganizovati kako bi bolje zadovoljili kriterijume često korišćenih upita, čak i ako prvobitno particionisanje nije osmišljeno da optimizuje određene upite.

Primer upita nad particijiom je sličan klasičnom upitu nad nekom tabelom, a opšti primer je dat u nastavku.

SELECT \* FROM <tabela> PARTITION <naziv\_particije> WHERE <uslov>

Ono što je bitno napomentu je da se nad particijama takođe mogu izvršavati operacije DELETE, UPDATE i INSERT.

Vrste particionisanja u MySQL su sledeće:

1. RANGE - particionisanje tabele na osnovu opsega, pri čemu svaka particije obuhvata redove čije vrednosti spadaju u definisani interval. Ovi intrvali treba da predstavljalju kontinuitet, što znači da je svaki opseg prilagođen sledećem opsegu tako da ne dolazi do preklapanja granica opsega. Za definisanje ovih granica koristi se operator VALUES LESS THAN, koji omogućava precizno definisanje gornje granice za svaku particiju.

CREATE TABLE zaposleni\_plata (

    jmbg NUMERIC,

    ime VARCHAR(50),

    datum\_rodjenja DATE,

    iznos\_plate INT,

UNIQUE KEY jmbg\_key (jmbg, iznos\_plate)

)

PARTITION BY RANGE (iznos\_plate) (

    PARTITION manja\_plata VALUES LESS THAN (10000),

    PARTITION srednja\_plata VALUES LESS THAN (20000),

    PARTITION veca\_plata VALUES LESS THAN (30000)

);

INSERT INTO zaposleni\_plata VALUES (123456789, "Marko Markovic", "1985-01-15", 9000);

INSERT INTO zaposleni\_plata VALUES (546212355, "Petar Petrovic", "1995-09-25", 10000);

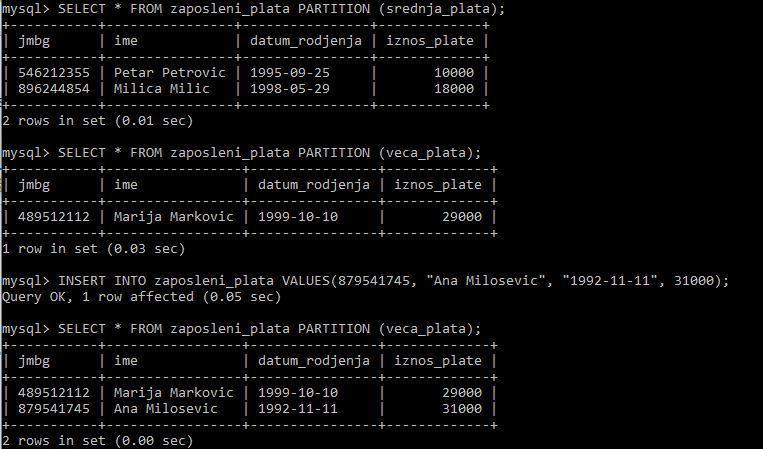
INSERT INTO zaposleni\_plata VALUES (489512112, "Marija Markovic", "1999-10-10", 29000);

INSERT INTO zaposleni\_plata VALUES (896244854, "Milica Milic", "1998-05-29", 18000);

U ovom primeru sve funkcioniše ukoliko se unose vrednosti koje su obuhvaćene particijama kao što je navedno u primeru iznad i može se lako predvideti da bi na primer radnik sa platom 9000 bio u drugoj particiji. Međutim greška može nastati ako se doda novi red gde bi plata bila 31000, tada MySQL ne bi znao šta da radi sa tom cifrom odnsno vratio bi grešku, da bi se izbegla ta greška može se koristiti izraz MAXVALUE u VALUES LESS THAN klauzuli. MAXVALUE predstavlja ceo broj koji je uvek veći od najvećeg mogućeg celog broja. Tako da se prethondi primer može izmeniti tako da umesto LESS THAN (30000) u trećoj particiji, piše:

PARTITION veca\_plata VALUES LESS THAN MAXVALUE

Sada je moguće dodavanje vrednosti u koloni plata koje su veće od 30000 i koje će biti smeštene u ovom slučaju, treću particiju, kao što je prikazano na sledećoj slici. Na slici je prikazana particija *srednja\_plata* zatim je prikazana particija *veca\_plata* nakon toga je dodat novi red kako bi bilo pokazano da radi particionisanje kada se umesto konkretnog broja koristi MAXVALUE i da je red dodat u treću particiju što se može videti iz posledenjeg upita koji prikazuje particiju *veca\_plata*.



Slika 7 Primer dodavanja reda u particionisanu tabelu sa korišćenjem ključne reči MAXVALUE

Kasnije ukoliko dolazi do čestog povećanja opsega može se iskoristi ALTER TABLE kako bi se dodale nove particije sa novim opsegom vrednosti.

Još jedan primer koji je veoma koristan u praksi je particionisanje po datumu odnosno moguće je particionisati tabelu tako da budu prikazani redovi koji se odnose na neki određeni vremenski period, što je i prikazano na sledećem primeru (koristi se ista tabela kao u prehodnim primerima).

PARTITION BY RANGE (YEAR(datum\_rodjenja)) (

    PARTITION p0 VALUES LESS THAN (1990),

    PARTITION p1 VALUES LESS THAN (2000),

    PARTITION p2 VALUES LESS THAN (2010)

);

1. LIST - slično kao RANGE particionisanje, zbog toga što je potrebno eksplicitno definisati vrednost na osnovu koje se vrši particionisanje. Međutim razlika je u tome kako se definiše opseg particionisanja, kod RANGE particije su određene susednim opsegom vrednosti, dok se kod LIST particionisanja, pripadnost particiji se određuje na osnovu pripadnosti vrednosti kolone unutar unapred definisanih listi vrednosti. Ovaj tip particionisanja omogućava veću fleksibilnost s obzirmo da nije potrebno imati kontinuirani opseg vrednosti. Za ovaj tip particionisanja koristi se klauzula PARITION BY LIST(*kol*) gde je *kol* kolona na osnovu koje se određuje pripadnost nekoj particiji i koja vraća ceo broj, takođe posle svake particije dolazi definicija liste vrednosti svake particije pomoću VALUES IN (*lista*), gde je *lista* lista celih brojeva odvojena zarezima.

CREATE TABLE zaposleni2 (

    jmbg NUMERIC,

    ime VARCHAR(50),

    id\_radnje INT

)

PARTITION BY LIST(id\_radnje) (

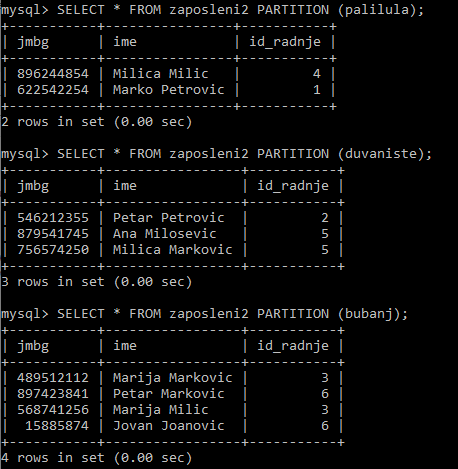
    PARTITION palilula VALUES IN(1, 4),

    PARTITION duvaniste VALUES IN(2, 5),

    PARTITION bubanj VALUES IN (3, 6)

);

Prilikom definisanja liste, potrebno je uzeti u obzi sve moguće vrednosti koje se mogu upisati u kolonu po kojoj se vrši particionisanje, u suportnom dolazi do greške da ne postoji particija za tu vrednost. U nastavku je dat prikaz particija gde se može videti da se *id\_radnje* poklapa za svaku od kreiranih particija.



Slika 8 Prikaz particija iz primera particionisanja po listi

1. COLUMNS - tip particionisanja omogućava korišćenje više kolona na osnovu kojih se vrši particionisanje, takođe postoje dve varijante ovog particionisanja, a to su RANGE COLUMNS i LIST COLUMNS particionisanje.

RANGE COLUMNS je veoma slično RANGE particionisanju ali dozvoljava definiciju particija na osnovu više kolona. Ova funkcionalnost pruža mogućnost particionisanja podataka na osnovu kompleksnijih kriterijuma koji obuhvataju nekoliko atributa. Još jedna prednost ovog načina particionisanja je ta što pruža mogućnost korišćenja i drugih tipova podataka pored celih brojeva i datuma, a to su stringovi. Ova mogućnost pruža rešenje za particionisanje koje je prilagodljivo specifičnim potrebama i karakteristikama skupa podataka.

CREATE TABLE zaposleni3 (

    jmbg NUMERIC,

    ime VARCHAR(50),

    datum\_rodjenja DATE,

    plata INT,

    id\_radnje INT

)

PARTITION BY RANGE COLUMNS(datum\_rodjenja, plata, id\_radnje) (

    PARTITION p0 VALUES LESS THAN ("1990-01-01", 10000, 2),

    PARTITION p1 VALUES LESS THAN ("2000-01-01", 20000, 3),

    PARTITION p2 VALUES LESS THAN (MAXVALUE, MAXVALUE, MAXVALUE)

);

LIST COLUMNS particionisanje se takođe nadograđuje na LIST particionisanje i kao i kod RANGE COLUMNS particionisanja omogućava korišćenje više kolona za kreiranje particija. Pored toga dozvoljava i korišćenje drugih tipova podataka kao što je *date* i *datetime,* kao i stringove.

PARTITION BY LIST COLUMNS(datum\_rodjenja) (

    PARTITION do1990 VALUES IN ("1985-01-15", "1979-05-29"),

    PARTITION do2000 VALUES IN ("1995-09-25", "1992-11-11"),

    PARTITION do2010 VALUES IN ("2002-10-10")

);

1. HASH - particionisanje služi za ravnomerno raspoređivanje odnosno distribuciju podataka preko unapred određenog skupa particija. Dok se kod RANGE ili LIST particionisanja zahteva specifikacija particije za svaku vrednost kolone ili skupa vrednosti, HASH particionisanje pojednostavljuje ovaj proces automatskim određivanje odgovarajuće partcije. Ovaj tip olakšava upotrebu time što zahteva samo specifikaciju kolone po kojoj će se vršiti particionisanje ili kolone za hash-iranje i željeni broj particija. Primer implementacije HASH particionisanja dat je u nastavku.

Particionisanje se vrši na osnovu kolone odnosno vrednosti koja je zadata po modulu koji predstavlja broj zadatih particija. Opšta formula bi bila ako *bp* predstavlja broj particija koji je zadat, *i* vrednost izraza i *n* broj partcije gde će izraz biti smešten onda je *n=* MOD(*i, bp*).

Za istu tabelu kao u prethodnom primeru kreiramo particije pomoću *hash-a.*

PARTITION BY HASH(YEAR(datum\_rodjenja))

PARTITIONS 3;

1. KEY - particionisanje po ključu u MySQL funkcioniše slično kao i *hash* particionisanje, sa razlikom u načinu na koji se particionisanje izvršava. Dok se *hash* paritcionisanje oslanja na korisnički definisan izraz za *hash-*iranje, a KEY particionisanje koristi funkciju *hash*-iranja koju obezbeđuje MySQL, takođe umesto ključne reči HASH koristi se KEY prilikom definisanja particionisanja.

Kod KEY particionisanja navodi se lista sa nula ili više imena kolona. Važno je napomenuti da kolone označene kao particioni ključ moraju ili biti deo ili sadržati ceo priamrni ključ tabele ako postoji. Ako nijedna kolona nije određena za particioni ključ, MySQL koristi primarni ključ tabele. Još jedna značajna razlika u odnosu na druge metode particionisanja je ta što kolone koje se koriste za particionisanje ključeva nisu ograničene samo na celobrojne vrednosti. Ova fleksibilnost omogućava da se koristi širi spektar tipova podataka za particionisanje.

ALTER TABLE zaposleni3 ADD PRIMARY KEY(jmbg)

PARTITION BY KEY()

PARTITIONS 3;

1. Sub - particionisanje predstavlja dalje deljenje particija. Moguće je dalje particionisanje samo po *range-u* ili *list-i* i to je moguće dalje koristiti *hash* ili *key* particionisanje. Svaka od particija mora da ima isti broj subparticija, takođe je moguće dati naziv subparticijama i to ako se daje naziv jednoj subparticiji, mora se dati i ostalim subparticijama naziv.

PARTITION BY RANGE(YEAR(datum\_rodjenja))

SUBPARTITION BY HASH(TO\_DAYS(datum\_rodjenja))

SUBPARTITIONS 2 (

    PARTITION p0 VALUES LESS THAN (1990) (

        SUBPARTITION s0,

        SUBPARTITION s1

    ),

    PARTITION p1 VALUES LESS THAN (2000) (

        SUBPARTITION s2,

        SUBPARTITION s3

    ),

    PARTITION p2 VALUES LESS THAN MAXVALUE (

        SUBPARTITION s4,

        SUBPARTITION s5

    )

);

U ovom primeru prvo se vrši particionisanje na osnovu godine rođenja, zatim se dalje svaka od tih particija koja se dobije deli na još dve particije na osnovu *hash-a* po broju dana do datuma rođenja od godine “0000-00-00”.

## 4.3. InnoDB buffer pool

*InnoDB storage engine* ima buffer pool koji predstavlja oblast u glavnoj memoriji gde InnoDB kešira podatke tabele i indeksa na osnovu pristupa. Buffer ppol omogućava da se često korišćenim podacima direktno pristupa iz memorije, što ubrzava rad i obradu podataka iz baze [15].

Za efikasnost čitanja velikog obima podataka, buffer pool je podeljen na stranice koje mogu da sadrže više redova. Ove stranice su implementirane kao lančana lista, a podaci koji se retko koriste se izbacuju iz keša pomoću varijante LRU (*Least Recently Used*) algoritma [16].

Algoritam funkcioniše tako što se 3/8 buffer pool-a dodeli “staroj” podlisti, postoji tačka između dve podliste gde se nalazi *tail* nove podliste i *head* stare podliste i kada InnoDB učita stranicu u buffer pool, dodaje je u toj tački, a čitanje se obavlja kada se koristi upit, automatski. Ukoliko se pristupi nekoj stranici koja se nalazi u staroj podlisti onda se ona premešta u mlađu podlistu, a stranice iz stare podliste kojima se ne pristupa se izbacuju iz buffera.

Podrazumevani algoritam koji koristi InnoDB za upravljanje buffer-om dinamičkom kategorizacijom stranica u stare ili nove podliste i prilagođavanjem njihove učestalosti pristupa, algoritam osigurava da podaci kojima se često pristupa ostaju lako dostupni. Ovaj proces poboljšava performanse baze podataka.

# 5. ZAKLJUČAK

Fizičko projektovanje MySQL baze podataka predstavlja ključan proces u obezbeđivanju njene optimalne performanse, skalabilnosti i pouzdanosti. Kroz pažljivo razmatranje faktora poput indeksiranja, denormalizacije, particionisanja i mehanizama keširanja, moguće je fino prilagoditi strukturu baze kako bi se zadovoljili specifični operativni zahtevi. Suština ovog procesa leži u redovnom praćenju i analizi performansi baze, kako bi se identifikovale mogućnosti za poboljšanje i implementirale potrebne promene.

Osim direktnih tehničkih prilagođavanja, važno je razmotriti i mogućnosti unapređenja hardvera, koje mogu značajno doprineti performansama baze. Uspeh fizičkog projektovanja nije samo u primeni pojedinačnih tehnika, već i u strateškom planiranju i kontinuiranom nadgledanju baze podataka, kako bi se osiguralo da promene dovode do željenih poboljšanja.

Stoga, zaključujemo da kontinuirana evaluacija performansi baze podataka, upotrebom odgovarajućih tehnika i alata za praćenje i analizu, predstavlja temelj procesa fizičkog projektovanja. Takav pristup omogućava ne samo identifikaciju područja za poboljšanje, već i osigurava da baza podataka neprestano evoluira kako bi zadovoljila kako trenutne, tako i buduće zahteve sistema koje opslužuje.

# LITERATURA

[1] Almeida, D., Lopes, M., Saraiva, L., Abbasi, M., Martins, P., Silva, J., & Váz, P. (2023, August). Performance Comparison of Redis, Memcached, MySQL, and PostgreSQL: A Study on Key-Value and Relational Databases. In 2023 Second International Conference On Smart Technologies For Smart Nation (SmartTechCon) (pp. 902-907). IEEE.

[2] Domaschka, J., Volpert, S., Maier, K., Eisenhart, G., & Seybold, D. (2023, April). Using eBPF for Database Workload Tracing: An Explorative Study. In Companion of the 2023 ACM/SPEC International Conference on Performance Engineering (pp. 311-317).

[3] <https://www.databasejournal.com/oracle/what-is-your-definition-of-database-workload/> (Pregledano: 14.01.2024.)

[4] <https://learn.microsoft.com/en-us/azure/synapse-analytics/sql-data-warehouse/sql-data-warehouse-workload-management> (Pregledano: 14.01.2024.)

[5] Ramakrishnan, R., & Gehrke, J. (2002). Database management systems. McGraw-Hill, Inc..

[6] Gao, P., Chen, Q., Xie, X., & Wang, C. (2023, May). Research on Performance Optimization of MySQL Database. In 2023 IEEE 3rd International Conference on Information Technology, Big Data and Artificial Intelligence (ICIBA) (Vol. 3, pp. 869-872). IEEE.

[7] Maesaroh, S., Gunawan, H., Lestari, A., Tsaurie, M. S. A., & Fauji, M. (2022). Query optimization in mysql database using index. International Journal of Cyber and IT Service Management, 2(2), 104-110.

[8] Teorey, T. J., Yang, D., & Fry, J. P. (1986). A logical design methodology for relational databases using the extended entity-relationship model. ACM Computing Surveys, 18(2), 197-222.

[9] Bahry, F. D. S., Amran, N., Putri, T. E., & Ramli, M. I. (2022). Database design of the malaysia public figures web archive repository: a social and cultural heritage web collections. Collection and Curation, 41(4), 133-140.

[10] Rao, J., Zhang, C., Megiddo, N., & Lohman, G. M. (2002). Automating physical database design in a parallel database. Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data.

[11] Lee, C. H., & Zheng, Y. L. (2015, October). SQL-to-NoSQL schema denormalization and migration: a study on content management systems. In 2015 IEEE International Conference on Systems, Man, and Cybernetics (pp. 2022-2026). IEEE.

[12] Buxton S., et al., „Database Design: Know it all“, Morgan Kaufmann, 2009.

[13] Kumar, A. S. (2016, August). Performance analysis of MySQL partition, hive partition-bucketing and Apache Pig. In 2016 1st India International Conference on Information Processing (IICIP) (pp. 1-6). IEEE.

[14] <https://dev.mysql.com/doc/refman/8.0/en/partitioning-overview.html> (Pregledano: 20.01.2024.)

[15] Lee, K. S., & Lee, S. W. (2023, February). Rethinking TPC-C Characteristic: At Logical Write Perspective. In 2023 IEEE International Conference on Big Data and Smart Computing (BigComp) (pp. 338-344). IEEE.

[16] <https://dev.mysql.com/doc/refman/8.3/en/innodb-buffer-pool.html> (Pregledano: 22.01.2024.)