Međimursko veleučilište u Čakovcu

RELACIJSKE BAZE PODATAKA I SQL

Nastavni materijal za kolegij Baze podataka

Dr.sc. Sanja Brekalo

2025.

AUTOR:
dr.sc. Sanja Brekalo, prof.struč.stud.

RECENZENTI:
dr.sc. Bruno Trstenjak, prof.struč.stud.
doc.dr.sc. Davor Cafuta, prof.struč.stud.

LEKTOR:
dr.sc. Ivan Bujan

NAKLADNIK:
Međimursko veleučilište u Čakovcu

ZA NAKLADNIKA:
nasl.izv.prof.dr.sc. Igor Klopotan, v.pred.

Objavljivanje skripte odobrilo je Vijeće odjela za tehničke studije (Odlukom: klasa: 007-09/25-01/19, urbroj:144-13-01-25-3)

Sadržaj

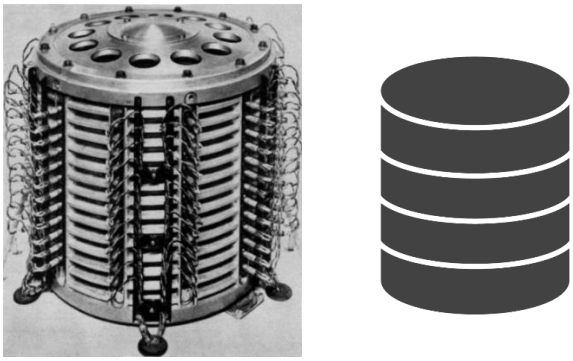
1. UVOD U BAZE PODATAKA

1.1. Uvod u baze podataka

Baze podataka predstavljaju strukturirane zbirke podataka koje omogućuju organizirano pohranjivanje, dohvaćanje i manipulaciju informacijama. One se sastoje od dva ključna aspekta: hardverskog i softverskog dijela. Hardverski dio obuhvaća računalo ili poslužitelj na kojem se baza podataka izvršava, dok softverski dio uključuje sustav za upravljanje bazama podataka (engl. d*atabase management system – DBMS*), koji omogućuje učinkovito upravljanje pohranjenim podacima. Kroz DBMS, korisnicima se omogućuje unos, pretraga, izmjena i brisanje podataka, čime baza podataka postaje ključan element u obradi i analizi informacija.

Kompanije koje imaju mnogo podataka danas su najvrijednije kompanije na svijetu (Amazon, Uber, Alibabba i sl.). Podaci se svakodnevno stvaraju preko mobilnih i *desktop* aplikacija, društvenih mreža, senzora, IoT uređaja, e-trgovine, *streaming* usluga, kamera, dronova, vožnjom u automobilima i sl. Prema procjenama, globalno se svakodnevno generira oko 2.5 kvintilijuna (10^{18}) bajtova podataka (2.5 eksabajta). Ovaj broj stalno raste zbog digitalizacije i povećane upotrebe tehnologije. Tvrdi se da je više podataka nastalo u posljednje dvije godine nego u cijeloj povijesti prikupljanja podataka[1].

Kako bi se baze podataka prikazale u literaturi i shemama, često se koriste slike u obliku naslaganih cilindara. Razlog tome je ranije korištena bubanj-memorija. Ovaj izraz opisuje vrstu magnetske memorije koja koristi rotirajući cilindrični bubanj obložen magnetskim materijalom za pohranu podataka. Bubanj-memorija bila je preteča modernih tvrdih diskova i koristila se u ranim računalima sredinom 20. stoljeća.



Slika 1. Izgled bubanj-memorije (engl. drum memory) i reprezentacija baze podataka. Izvor: https://upload.wikimedia.org/wikipedia/commons/d/d2/Pamiec_bebnowa_1.jpg

[1] https://spacelift.io/blog/how-much-data-is-generated-every-day

1.2. Povijest baza podataka i SQL-a

Baze podataka korištene su za organizaciju i pohranu podataka mnogo prije računalnog doba. Računala su omogućila automatizaciju pohrane i korištenja podataka. Isprva su se koristile jednostavne datoteke (engl. *flat files*) u obliku tablica u koje su zapisivani podaci. To je zahtijevalo pretraživanje od početka liste što je spor način za upravljanje velikim količinama podataka.

Sredinom 1960-ih, IBM je razvio hijerarhijski model za sustav upravljanja podacima—IMS (engl. *information management system*), koji je organizirao podatke u hijerarhijske strukture. Fleksibilniji mrežni model (engl. *network model*) razvio je Charles Bachman za *General Electric*, no takav sustav, koji se oslanjao na složene mreže međusobno povezanih čvorova putem pokazivača (engl. *pointers*), postajao je vrlo kompleksan za održavanje i upravljanje pri velikim količinama podataka.

U 1970-ima, Edgar F. Codd, informatičar iz IBM-a, objavio je članak pod nazivom „*A Relational Model of Data for Large Shared Data Banks*“, u kojem je opisao model pohrane podataka organiziran u tablice s povezanim podacima. Ovaj rad postavio je temelje za relacijske baze podataka i SQL jezik (engl. *structured query language*).

Michael Stonebraker je na Sveučilištu Berkeley 1970-ih, slijedeći Coddov rad, osmislio relacijsku bazu podataka INGRES (engl. *Interactive Graphics and Retrieval System*), koju su mnoge kompanije iskoristile za uspješne komercijalne proizvode. INGRES je kasnije evoluirao u druge sustave, poput Postgresa (također djelo Michaela Stonebrakera), koji je postao osnova za PostgreSQL, jedan od najpopularnijih sustava za upravljanje bazama podataka danas. INGRES nije saživio kao vodeća komercijalna baza podataka jer nije uspio kapitalizirati svoj rani početak zbog nedostatka fokusirane komercijalizacije, marketinške strategije i tehnološke prilagodbe. Ipak, njegov doprinos razvoju relacijskih baza podataka bio je ključan za napredak cijelog polja.

IBM je 1974. godine započeo razvoj eksperimentalne relacijske baze podataka nazvane System R. Dvojica IBM-ovih programera, Donald D. Chamberlin i Raymond F. Boyce, izradili su prvu verziju SQL-a, uključujući jezik i softver za kreiranje i upravljanje bazama podataka. Izvorni naziv za SQL bio je SEQUEL (engl. *structured english query language*), ali je preimenovan u SQL zbog problema s autorskim pravima.

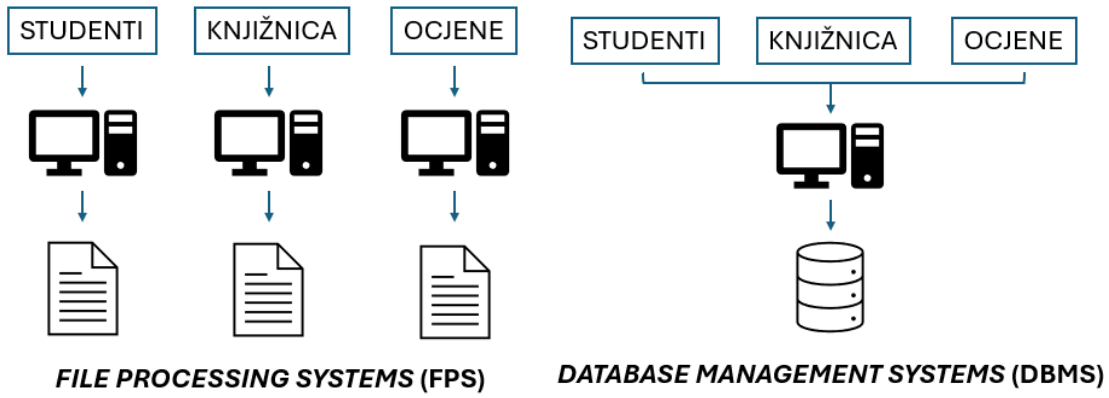
Poduzetnik Larry Ellison osnovao je 1977. godine, zajedno s Edom Oatesom i Bobom Minerom, softversku tvrtku koja je 1979. godine razvila i prodavala prvu komercijalnu relacijsku bazu podataka Oracle. Ta baza bila je kompatibilna s IBM-ovim System R, čime je olakšana migracija korisnika s IBM-ovih sustava na Oracle.

IBM je 1983. godine isporučio svoju prvu komercijalnu bazu podataka DB2, no Oracle je već dominirao tržištem, utječući na način na koji je većina podataka danas organizirana.

1.3. Usporedba sustava za obradu datoteka i sustava za upravljanje bazama podataka

Sustavi za obradu datoteka (engl. *File Processing Systems – FPS*) primitivni su sustavi za pohranu podataka prikladni za jednostavne aplikacije s malim skupovima podataka. Podaci se pohranjuju u zasebnim datotekama (engl. *flat files*). Nema centralizirane kontrole, već svaki program upravlja vlastitim podacima. Podaci su fragmentirani, često se dupliciraju i pohranjuju u različitim datotekama te nisu međusobno povezani. Pristup podacima zahtijeva specifične programe za svaku datoteku te ne postoji standardizirani jezik za manipulaciju podacima. Rad s podacima u velikim datotekama je spor jer često treba pregledavati cijele datoteke. Najveći problemi u ovakvom načinu rada su redundantni, izolirani podaci i datotečne strukture u koje se spremaju informacije.

Sustavi za upravljanje bazama podataka (engl. *Database Management Systems – DBMS*) pružaju moćne alate za centralizirano upravljanje, integraciju, sigurnost i analizu velikih količina podataka. Umjesto fragmentiranog pristupa kao u FPS sustavima, gdje su se sustavi za upravljanje informacijama gradili zajedno s aplikacijskim softverom, DBMS omogućuje odvojenost logike pohrane i upravljanja podacima od aplikacija koje ih koriste. Ovaj pristup pojednostavljuje razvoj, održavanje i skaliranje informacijskih sustava. Podaci su integrirani i povezani kroz relacije, omogućujući složene upite i analize. Izbjegava se ponavljanje informacija prilikom spremanja zahvaljujući normalizaciji podataka i centraliziranoj pohrani. Usporedba sustava vidljiva je na slici 2.



Slika 2. Datotečni pristup podacima te isti sustav postavljen kao DBMS

Slika 3. *Database management systems*

Na slici 3. prikazani su dijelovi sustava za upravljanje bazama podataka, gdje je ilustrirano kako se putem upita u jeziku SQL pristupa softveru za upravljanje bazama podataka i samim pohranjenim podacima.

1.4. Modeli i vrste baza podataka

Danas je dostupno mnogo popularnih baza podataka[2], a njihov odabir ovisi o potrebama korisnika, vrsti podataka i specifičnoj primjeni. Kako bi se bolje razumjela organizacija i način rada baza podataka, potrebno je razlikovati konceptualne modele baza podataka i konkretne vrste baza prema njihovoj implementaciji.

1.4.1. Modeli baza podataka

Model baze podataka određuje način na koji se podaci strukturiraju i kako se njima upravlja. Kroz povijest razvoja baza podataka, definirano je nekoliko ključnih modela:

1. **Hijerarhijski model** (engl. *hierarchical*) – podaci su organizirani u hijerarhijsku strukturu, poput stabla. Ovaj model se danas rijetko koristi.
2. **Mrežni model** (engl. *network*) – slično hijerarhijskom modelu, ali omogućuje složenije veze između podataka. Koristi se u specijaliziranim sustavima.
3. **Relacijski model** (engl. *relational*) – organizacija podataka u tablice povezane primarnim i stranim ključevima. Najčešće korišten model u tradicionalnim bazama podataka.
4. **Objektno-relacijski model** (engl. *object-relational*) – kombinira značajke relacijskih i objektno-orijentiranih baza.
5. **Objektni model** (engl. *object-oriented*) – temeljen na objektno-orijentiranom programiranju, koristi se u specifičnim primjenama.
6. **Grafički model** (engl. *Graph*) – model koji koristi čvorove i veze za predstavljanje podataka, pogodan za složene odnose.
7. **Dokumentni model** (engl. *Document*) – struktura podataka temelji se na dokumentima (npr. JSON, XML).
8. **Ključ-vrijednost model** (engl. *Key-Value*) – podaci su organizirani u jednostavne parove ključ-vrijednost.
9. **Kolonski model** (engl. *Column-Family*) – podaci su pohranjeni po stupcima umjesto redaka, što poboljšava analitičke performanse.
10. **Multimodel** (engl. *Multi-Model*) – baze podataka koje podržavaju više modela unutar iste implementacije.
11. **Temporalni model** (engl. *Temporal*) – model koji uključuje dimenziju vremena za praćenje promjena podataka.
12. **NoSQL** – Opći pojam koji obuhvaća više modela (dokumentni, grafički, ključ-vrijednost, kolonski) i koristi se za rad s velikim, nestrukturiranim podacima.

[2] Popis baza podataka: https://en.wikipedia.org/wiki/Lists_of_database_management_systems

1.4.2. Vrste baza podataka prema primjeni

Na temelju gore navedenih modela, razvijene su različite vrste baza podataka koje se koriste u praksi. U nastavku su navedene najčešće korištene baze podataka, podijeljene prema njihovoj strukturi i primjeni.

1. **Relacijske baze podataka (engl. *Relational Databases*)**
   Relacijske baze organiziraju podatke u tablice i koriste SQL za dohvat i manipulaciju podacima. Pogodne su za sustave koji zahtijevaju strogu shemu i dosljednost, poput poslovnih i financijskih aplikacija.
   Primjeri:
   - MariaDB – baza otvorenog koda popularna za web aplikacije.
   - MySQL – jedna od najkorištenijih baza za web aplikacije.
   - PostgreSQL – robusna i proširiva baza otvorenog koda.
   - Microsoft SQL Server – komercijalna baza podataka za Windows i Azure.
   - Oracle Database – snažna komercijalna baza.
   - SQLite – lagana baza za mobilne i ugrađene sustave.
2. **Dokumentne baze podataka (engl. d*ocument databases*)**
   Dokumentne baze pohranjuju podatke u obliku dokumenata (npr. JSON, BSON) i prikladne su za nestrukturirane i polustrukturirane podatke.
   Primjeri:
   - MongoDB – fleksibilna dokumentno orijentirana baza.
   - Firebase Realtime Database – baza za mobilne i web aplikacije s fokusom na sinkronizaciju.
   - CouchDB – baza optimizirana za spremanje i repliciranje dokumenata.
3. **Baze podataka ključ-vrijednost (engl. *Key-Value Databases*)**
   Podaci se pohranjuju u obliku parova ključ-vrijednost, što omogućuje brz i jednostavan pristup.
   Primjeri:
   - Redis – brza baza u radnoj memoriji, idealna za keširanje.
   - DynamoDB – Amazonova visoko skalabilna NoSQL baza podataka.
   - Riak – distribuirana baza ključ-vrijednost.
4. **Graf baze podataka (engl. *Graph Databases*)**
   Graf baze koriste čvorove i veze za modeliranje odnosa među podacima, što ih čini pogodnima za društvene mreže, analizu povezanosti i složene odnose.
   Primjeri:
   - Neo4j – najpoznatija graf baza podataka.
   - Amazon Neptune – graf baza optimizirana za AWS aplikacije.
5. **Stupčaste baze podataka (engl. *Wide Column Databases*)**
   Podaci se pohranjuju po stupcima umjesto redaka, što omogućuje brzo izvođenje analitičkih upita na velikim setovima podataka.

Primjeri:

- Apache HBase – distribuirana kolumnarna baza često korištena za velike podatkovne sustave.
- Google Bigtable – baza podataka koja pokreće Googleove servise poput Gmaila.
- Cassandra – visoko dostupna baza podataka za velike sustave.

1.5. Sustavi za upravljanje bazama podataka (DBMS)

Sustav za upravljanje bazom podataka (engl. *Database Management System - DBMS*) je program kojima se upravlja bazom podataka. Njime radimo promjene u bazama koje uključuju CRUD[3], menadžment podataka i transakcija te se osigurava da su podaci spremljeni sukladno odabranom modelu pohrane podataka. Također, DBMS omogućuje sigurnost podataka postavljanjem pravila sigurnosti i dozvola za pristupanje podacima. Softver za upravljanje bazama podataka uglavnom služi kako bi osigurao da podaci uvijek budu sigurni i dostupni.

Postoje različiti DBMS sustavi koji su građeni na različite načine, različito procesiraju zahtjeve, posjeduju različite mehanizme za upravljanje podacima i podržavaju različite funkcionalnosti. Neki od najznačajnijih proizvođača DBMS sustava su Oracle (Oracle DB, MySQL), Microsoft (SQL Server), IBM (DB2), PostgreSQL, SAP (HANA), te open-source sustavi poput MongoDB i MariaDB. Također, rješenja u oblaku, poput Amazon Aurora i Google BigQuery, sve su popularnija. Svi navedeni alati imaju svoje prednosti i nedostatke. Kod DBMS sustava prednost može biti komuniciranje putem SQL jezika, kojeg sve SQL baze podržavaju. Valja napomenuti da su danas na tržištu dostupni i DBMS sustavi koji ne podržavaju SQL, već se temelje na drugim paradigmama, pa se shodno tome nazivaju NoSQL DBMS sustavi (npr. MongoDB). Amazon Aurora i Google BigQuery nisu potpuno NoSQL sustavi, već nude podršku za različite paradigme, uključujući i SQL i NoSQL funkcionalnosti.

1.5.1. Relacijske baze podataka (*RDBMS*)

Relacijske baze podataka (engl. *Relational DataBase Management* System – RDBMS) su sustavi za pohranu, organizaciju i upravljanje podacima koji koriste tablični model. Podaci su predstavljeni u obliku tablica s redovima i stupcima. Svaka tablica (relacija) sadrži jedinstveni skup podataka s definiranim atributima, a odnosi između podataka u različitim tablicama uspostavljaju se pomoću ključeva. Relacijske baze koriste SQL (engl. *Structured Query Language*) za upravljanje i dohvat podataka te su idealne za aplikacije koje zahtijevaju strogu konzistentnost i strukturirane podatke.

U niže prikazanim tablicama postoje ključevi preko kojih se stvaraju poveznice između relacija *narudžba*, *proizvod* i *kupac*. Ove relacije su ključne za povezivanje podataka i omogućuju analizu i upravljanje sustavom. Tablice su povezane u tablici *narudžba* sa *kupac_id* i *proizvod_id*. Preko navedenih vrijednosti možemo identificirati kupca koji je naručio određeni proizvod. Povezivanjem narudžbi i proizvoda prema *proizvod_id* možemo vidjeti koliko puta je svaki proizvod naručen. Relacije omogućuju organizaciju i analizu

[3] CRUD je akronim za četiri osnovne operacije u radu s bazama podataka: CREATE (stvaranje ili pohranjivanje novih podataka), READ (čitanje ili dohvaćanje podataka iz baze), UPDATE (ažuriranje postojećih podataka) i DELETE (brisanje podataka).

podataka u sustavu, čineći ga jednostavnim za praćenje kupaca, proizvoda i njihovih narudžbi. Relacijska baza podataka omogućuje izvođenje upita i generiranje izvještaja temeljenih na ovim povezanim podacima.

Tablica 1. Prikaz tablice *narudžba*.

narudzba_id	kupac_id	datum_narudzbe	proizvod_id	kolicina
1	1	2023-10-01	2	1
2	4	2023-10-02	1	100
3	3	2023-10-03	4	32
4	2	2023-10-04	3	1

Tablica 2. Prikaz tablice *kupac*

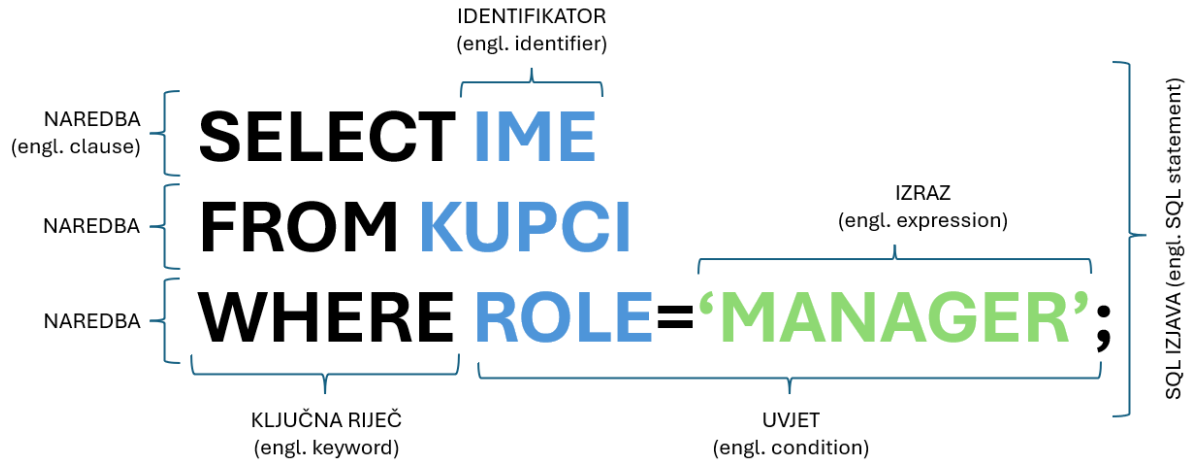
kupac_id	ime	adresa
1	Ana	Ulica Kralja Tomislava 1, Zagreb
2	Marko	Trg Bana Jelačića 5, Split
3	Ivana	Riva 12, Rijeka
4	Petar	Obala Kneza Domagoja 7, Dubrovnik

Tablica 3. Prikaz tablice *proizvod*

proizvod_id	naziv	cijena
1	Laptop	900
2	Miš	30
3	Tipkovnica	75
4	Monitor	250

1.6. SQL jezik

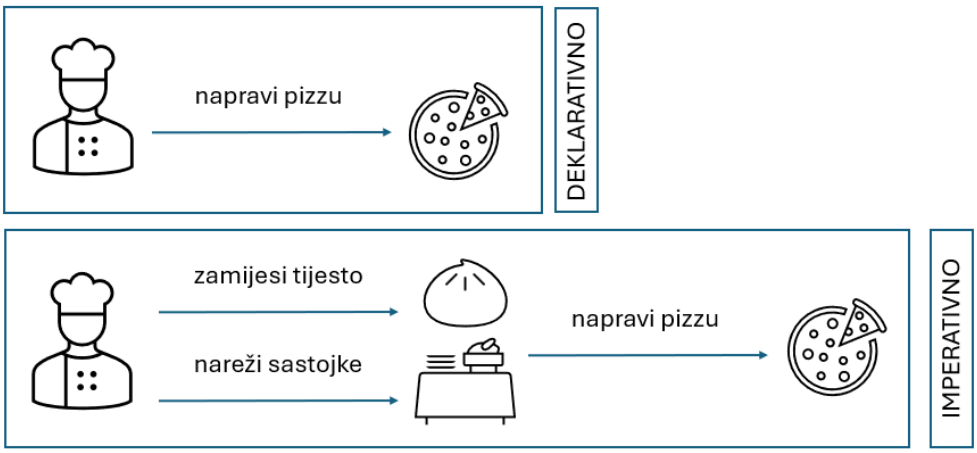
SQL (engl. *Structured Query Language*) je standardizirani programski jezik koji se koristi za upravljanje, dohvat i manipulaciju podacima pohranjenima u relacijskim bazama podataka. Pomoću SQL-a korisnici mogu kreirati i modificirati strukturu baza podataka, unositi, ažurirati i brisati podatke te izvoditi složene upite radi analize podataka. SQL upit (engl. *Query*) je zahtjev tj. instrukcija kojim korisnik komunicira s bazom podataka kako bi dobio informacije ili izvršio operaciju.



Slika 4. Dijelovi SQL upita

Slika 4. prikazuje dijelove SQL upita. Naredbe (engl. *clause*) su dio upita koje su sačinjene od ključne riječi (engl. *keyword*), identifikatora (engl. *identifier*) i uvjeta (engl. *condition*). Zadnja linija sa WHERE ključnom riječi predstavlja filtriranje podataka.

SQL je deklarativan jezik u kojem izjavljujemo što će se dogoditi za razliku od imperativnog jezika u kojem izjavljujemo kako će se dogoditi. Usporedba deklarativnog i imperativnog programiranja prikazana je na slici 5. Kod deklarativnih jezika izjavljujemo što će se dogoditi bez da znamo kako će se dogoditi.



Slika 5. Usporedba deklarativnog i imperativnog programiranja

1.6.1. SQL standardi

Prvi SQL standard definiran je 1986. godine od strane organizacije ANSI (engl. *American National Standards Institute*), a ubrzo ga je preuzela i ISO (engl. *International Organization for Standardization*), čime je postao globalni standard.

Baze podataka podržavaju standardiziranu verziju SQL zbog kompatibilnosti i pristupačnosti. Standardizirane verzije omogućuju korisnicima jednostavnu migraciju između različitih sustava bez potrebe za potpunom prilagodbom njihovih aplikacija. Standardizacija također omogućuje učenje jednog jezika koji se može koristiti široko, umjesto prilagodbe svakom sustavu pojedinačno. Međutim, različite baze podataka dodaju svoja proširenja SQL standardu kako bi se diferencirale i ponudile dodatne funkcionalnosti. Primjerice Oracle ima PL/SQL, što je proceduralni dodatak SQL-u. Microsoft SQL Server koristi T-SQL (Transact-SQL) koji uključuje specifične funkcije poput upravljanja transakcijama i izračuna. PostgreSQL dodaje naprednu podršku za prilagođene tipove podataka i proceduralni jezik PL/pgSQL. Ta proširenja omogućuju specifične optimizacije i dodatne značajke, ali često dolaze s cijenom—manjom prenosivošću koda između različitih baza podataka.

Nove inačice SQL standarda donose se kroz koordinirani proces u kojem sudjeluju stručnjaci iz raznih industrija, uključujući predstavnike velikih tehnoloških tvrtki i akademske zajednice. Proces izdavanja nove inačice započinje prijedlozima za dodatke ili promjene u standardu, nakon čega slijedi evaluacija i testiranje.

Svaka nova inačica SQL-a nosi svoj naziv, često vezan uz godinu izdavanja. Do sada su izdane sljedeće važnije verzije SQL-a:

- SQL-86 (prva standardna verzija);
- SQL-89 (manje revizije);
- SQL-92 (veće proširenje funkcionalnosti);
- SQL:1999 (uvođenje objektnog-relacijskog modela i rekurzivnih upita);
- SQL:2003 (XML podrška);
- SQL:2008 (dodaci za OLAP funkcije);
- SQL:2011 (dodaci za vremenske podatke);
- SQL:2016 (dodaci za JSON podršku);
- SQL:2019 (proširenja za *big data* analitiku i polustrukturirane podatke);
- SQL:2023 (podrška za grafove i dodatna proširenja za JSON).

Uvođenje nove inačice ne znači da baze podataka odmah implementiraju sve nove funkcionalnosti. Proizvođači baza podataka obično selektivno implementiraju dijelove standarda koji su relevantni za njihove korisnike, što često rezultira asinkronim usvajanjem novih značajki u industriji.

2. RELACIJSKI MODEL BAZA PODATAKA

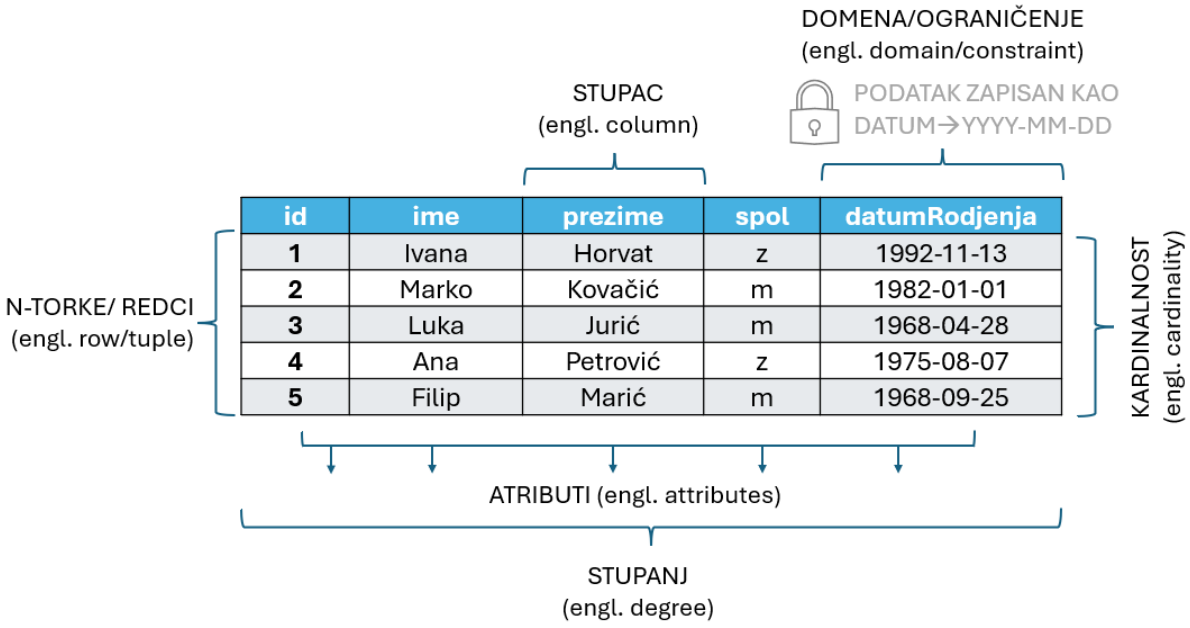
2.1. Struktura tablica u relacijskom modelu

Tablice u relacijskim bazama podataka (RDBMS) su objekti, tj. osnovne strukture za pohranu podataka organiziranih u redove i stupce. Tablice su temeljna jedinica organizacije podataka u RDBMS sustavima i omogućuju učinkovito pohranjivanje, dohvaćanje, ažuriranje i brisanje podataka. Svaka tablica predstavlja skup povezanih podataka i mora imati naziv koji predstavlja taj objekt tj. entitet. Najznačajniji dijelovi relacijskih tablica su:

1. **Stupci (atributi)**
   Svaki stupac ima naziv i unaprijed definirani tip podataka koji će se spremati u ćelijama (npr. tekst, broj, datum). **Stupanj** (engl. *degree*) označava broj atributa (stupaca) u tablici. Primjerice, ukoliko tablica ima 5 stupaca, utoliko je njezin stupanj 5.
2. **Redovi (zapisi ili n-torke)**
   Predstavljaju pojedinačne instance ili unose podataka. Svaki redak u tablici je jedinstven (obično zahvaljujući primarnom ključu) kako bi se razlikovao od ostalih redaka. N-torka označava jedan redak u tablici baze podataka, koji se sastoji od n vrijednosti (podataka), gdje svaka vrijednost pripada određenom atributu (stupcu) tablice. **Kardinalnost** predstavlja broj redaka u tablici.
3. **Ključevi**
   Primarni ključ (engl. *primary key*) jedinstveni je identifikator svakog retka u tablici. **Vanjski (ili strani) ključ** (engl. *foreign key*) koristi se za povezivanje tablica u relacije.
4. **Relacije**
   Tablice su međusobno povezane putem ključeva, omogućujući modeliranje odnosa između podataka (npr. "jedan prema jedan", "jedan prema više", "više prema više").

Slika 6. prikazuje tablicu *korisnik* koja je zapisana u RDBMS-u. Svaki stupac tablice ima naziv koji definira vrstu i tip podatka koji će se pohraniti u ćeliji. Tablica s 5 stupaca ima stupanj 5. Stupanj relacije predstavlja broj atributa, odnosno kolekciju stupaca. Kod izrade tablica svakom je stupcu potrebno zadati tip podataka koji će se u njemu spremati, pa je, primjerice, za id određen automatski inkrement za svaki novi redak. Za datumRodjenja postavljena je **domena**, odnosno **ograničenje**, da se u navedeno polje može pohranjivati samo datum. Stupci se u tablici nazivaju i atributima, a atributi imaju svoja ograničenja.

Tablica ima ukupno 5 redaka unesenih podataka. Jedan redak ili n-torka predstavlja pojedinačni redak podataka, dok n-torke označavaju više redaka podataka. Svaki redak, kada se zapisuje, mora zadovoljiti ograničenja definirana za atribute. Kardinalnost tablice odnosi se na broj redaka u tablici.



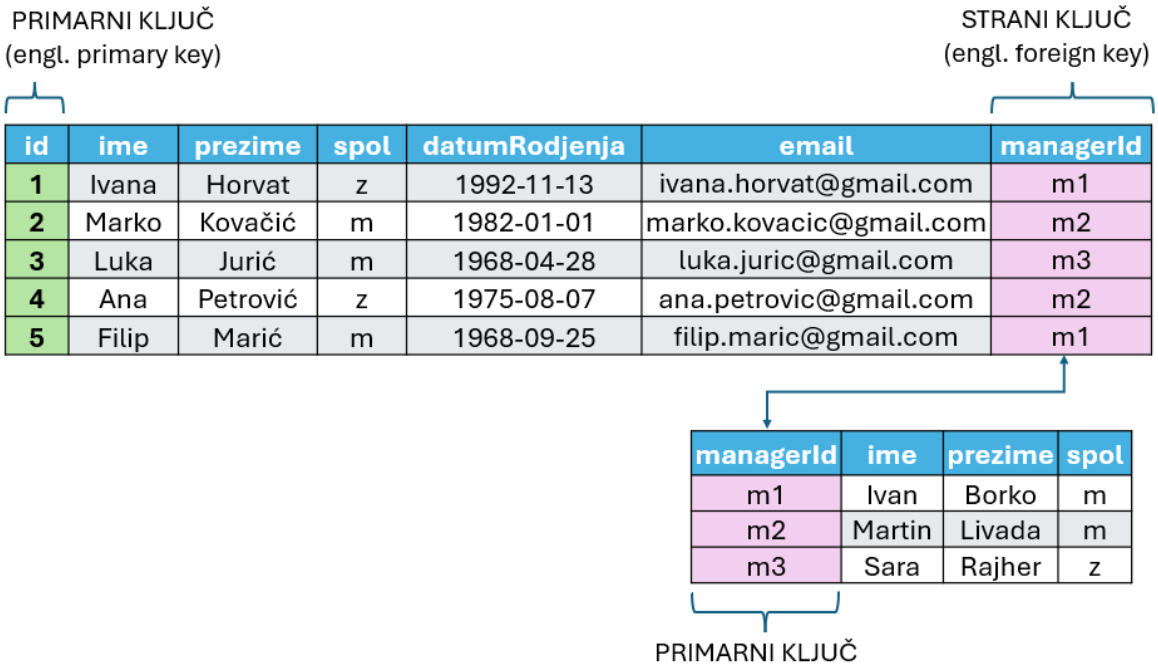
Slika 6. Temeljni pojmovi i struktura tablice *korisnik* u relacijskim bazama podataka

Slika 7. prikazuje poveznicu između relacija preko ključeva te prikazuje primarne i strane ključeve u relacijskim bazama podataka. **Primarni ključ** je atribut (stupac) ili skup atributa u tablici baze podataka koji služi kao jedinstveni identifikator svakog retka (n-torke) u tablici. Vrijednost primarnog ključa mora biti jedinstvena za svaki redak i ne smije biti *null* (prazna).

Ključne značajke primarnog ključa:

- Jedinstvenost - svaki redak mora imati jedinstvenu vrijednost primarnog ključa.
- Nepostojanje *null* vrijednosti - primarni ključ mora uvijek imati definiranu vrijednost.
- Identifikacija - omogućuje jednoznačno razlikovanje svakog retka u tablici.

Strani ključ je ključ koji se referencira na primarni ključ, jedinstveni identifikator druge tablice, čime se omogućuje uspostavljanje relacije između tablica.



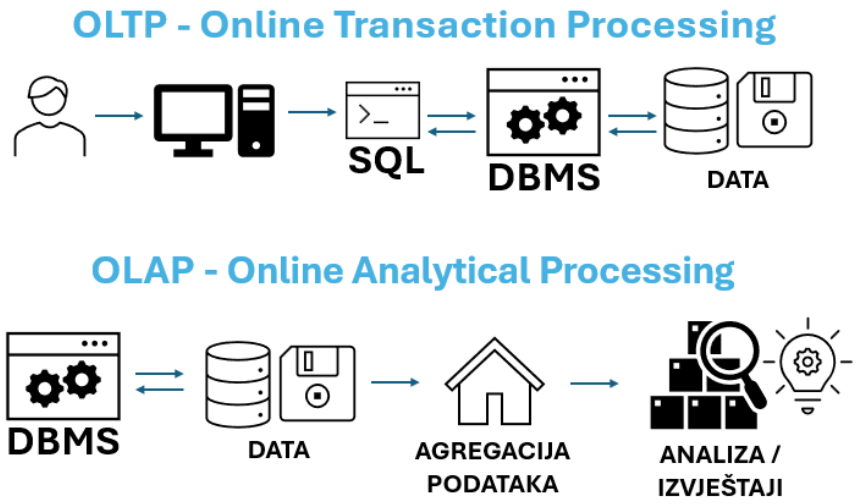
Slika 7. Primarni i strani ključevi u relacijskim bazama podataka

2.2. Korištenje relacijskih modela baza podataka

Relacijski sustavi za upravljanje bazama podataka (RDBMS) koriste se za dvije glavne vrste operacija: OLTP i OLAP.

OLTP (engl. *Online Transaction Processing*) sustavi usmjereni su na obradu transakcija u stvarnom vremenu. Njihova glavna svrha je omogućiti brzu i pouzdanu obradu velikog broja transakcija, poput unosa podataka, ažuriranja i brisanja. Tipični primjeri OLTP sustava uključuju bankarske sustave, sustave za upravljanje narudžbama ili sustave prodajnih mjesta (POS). OLTP baze su optimizirane za brzinu i dosljednost podataka, koristeći normalizirane strukture kako bi minimizirale redundanciju.

S druge strane, **OLAP (engl. *Online Analytical Processing*)** sustavi koriste se za analitičke svrhe, poput izvještavanja, analiza podataka i donošenja poslovnih odluka. OLAP baze su optimizirane za složene upite i agregacije podataka. Primjeri uključuju sustave za analizu prodaje, financijske projekcije ili praćenje trendova na tržištu. OLAP baze često koriste denormalizirane strukture kako bi se povećala učinkovitost analitičkih upita.



Slika 8. Razlike između OLTP i OLAP sustava

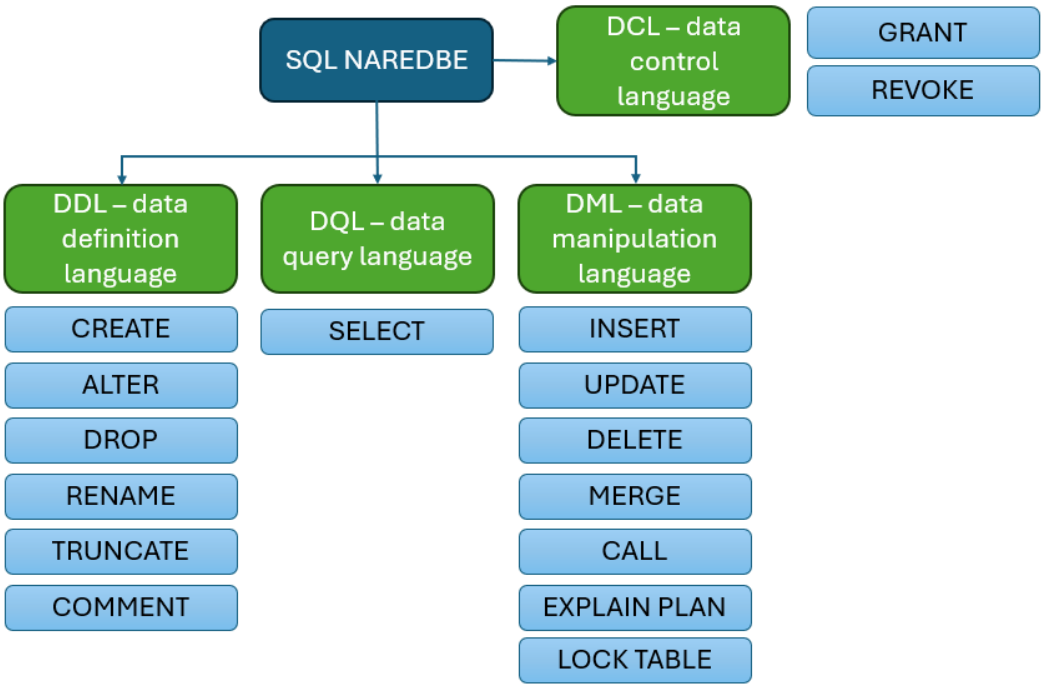
Slika 8. prikazuje razliku između pohrane i obrade podataka u OLTP i OLAP sustavima. Kod OLTP-a, podaci se pohranjuju kao zapisi koji se koriste za svakodnevne transakcije. Ti podaci su organizirani tako da budu učinkoviti za brzu obradu i izmjenu. OLTP pokreće svakodnevne poslovne operacije i usredotočuje se na upravljanje podacima potrebnim za trenutne transakcije. OLAP sustavi obrađuju i sažimaju podatke u agregate kako bi se olakšale analize. Agregacija znači da se podaci grupiraju, zbrajaju ili prosječno računaju kako bi se dobili korisni uvidi. OLAP sustav koristi podatke koje prenosimo u skladište podataka (engl. *data warehouse*), gdje se provode različite vrste analitičke obrade kako bismo otkrili ključne informacije o tim podacima. Cilj je razumjeti podatke, kako ih

iskoristiti te koje uvide možemo dobiti. Podaci se koriste za analizu koja pomaže tvrtkama u donošenju odluka. Odluke potom vode do razvoja novih proizvoda, ulaska na nova tržišta i boljih interakcija s korisnicima. To je upravo ono čemu OLAP služi, a to je pogled u budućnost i podrška donošenju strateških odluka. OLAP i OLTP, iako različiti, zajedno omogućuju tvrtkama da istovremeno upravljaju trenutnim poslovanjem i planiraju budućnost. OLTP se fokusira na operativni rad, dok OLAP omogućuje analizu i razumijevanje povijesnih podataka.

Iako OLTP i OLAP imaju različite ciljeve, oba sustava oslanjaju se na RDBMS sustave kao osnovnu tehnologiju. Moderni sustavi sve češće implementiraju oba pristupa koristeći hibridne arhitekture kako bi zadovoljili potrebe za transakcijama u stvarnom vremenu i analizom velikih podataka.

3. OSNOVE SQL-a

3.1. Podjela SQL naredbi



Slika 9. Kategorizacija SQL naredbi

Slika 9. prikazuje kategorizaciju SQL naredbi prema njihovoj ulozi pri upravljanju bazom. SQL naredbe možemo podijeliti prema njihovoj namjeni u sljedeće kategorije:

- **DCL (engl. *Data Control Language*) – jezik za upravljanje podacima**. Koristi se za kontrolu pristupa podacima u bazi, poput dodjeljivanja ili povlačenja prava (primjeri: GRANT, REVOKE).
- **DDL (engl. *Data Definition Language*) – jezik za definiranje podataka**. Služi za definiranje strukture baze podataka, poput stvaranja ili brisanja tablica i baza (primjeri: CREATE, ALTER, DROP).
- **DQL (engl. *Data Query Language*) – jezik za upite podataka**. Koristi se za dohvaćanje podataka iz baze, često putem SELECT naredbe.
- **DML (engl. *Data Manipulation Language*) – jezik za manipulaciju podacima**. Služi za manipulaciju podacima unutar baza, poput umetanja, ažuriranja ili brisanja podataka (primjeri: INSERT, UPDATE, DELETE).

3.2. Osnovna sintaksa

3.2.1. Komentari

Pri pisanju SQL naredbi mogu se koristiti komentari kako bi se dodale napomene ili objašnjenja koja olakšavaju razumijevanje koda.

- Jednolinijski komentari započinju s --.
- Višelinijski komentari započinju s /*, završavaju s */, a svaka linija unutar komentara može započeti s opcionalnom zvjezdicom za bolju čitljivost.

  Primjer:

```
-- Ovo je jednolinijski komentar
/*
  Ovo je višelinijski komentar.
* Ovdje se stavlja
* dulje objašnjenje
*/
SELECT * FROM zaposlenik;
```

3.2.2. Jednostruki i dvostruki navodnici u SQL izjavama

U SQL izjavama jednostruki (') i dvostruki (") navodnici imaju različitu namjenu i upotrebu, ovisno o sustavu za upravljanje bazama podataka koji se koristi.

- **Jednostruki navodnici (')**: Najčešće se koriste za označavanje tekstualnih vrijednosti (stringova) unutar SQL naredbi.

  Primjer:

```
SELECT *
FROM zaposlenik
WHERE odjel = 'IT';
```

- **Dvostruki navodnici (")**: U nekim DBMS-ovima, poput PostgreSQL-a ili Oracle-a, dvostruki navodnici se koriste za označavanje naziva objekata baze podataka (npr. tablica, stupaca) koji sadrže posebne znakove, razmake ili su rezervirane riječi. Identifikatori koji su u SQL izjavi napisani unutar dvostrukih navodnika postaju osjetljivi na velika i mala slova. U suprotnom, SQL je u pravilu neosjetljiv na velika i mala slova. Ovo znači da, ako se koriste dvostruki navodnici, naziv tablice, stupca ili drugog objekta mora se navesti točno onako kako je definiran (uključujući kombinaciju velikih i malih slova). Bez dvostrukih navodnika, SQL će identifikatore

automatski pretvoriti u velika slova ili ih tretirati na način specifičan za određeni DBMS.

Primjer:

```
SELECT "Ime zaposlenika", "Datum rođenja"
FROM "Zaposlenik"
WHERE odjel = 'IT';
```

Razlika između navodnika može varirati među sustavima, pa je važno poznavati pravila za korištenje navodnika u DBMS-u s kojim se radi.

U MySQL-u se mogu koristiti i jednostruki (') i dvostruki (") navodnici za definiranje tekstualnih vrijednosti. Preporučuje se korištenje jednostrukih (') navodnika jer su oni standardni u SQL-u. Dvostruki navodnici (") koriste se za nazive stupaca i tablica, ali samo ako je SQL način rada (SQL_MODE) postavljen tako da podržava ANSI_QUOTES. U MySQL-u i MariaDB-u, obično se koriste backtick (`) za identifikatore umjesto dvostrukih navodnika.

3.2.3. Funkcije u SQL naredbama

Pri pisanju SQL naredbi mogu se koristiti funkcije. Naziv funkcije jasno upućuje na zadatak koji će funkcija izvršiti. Funkcije u SQL-u primaju parametre i vraćaju vrijednosti, odnosno rezultat svoje obrade. One su setovi koraka koji generiraju jednu konačnu vrijednost. U SQL-u se funkcije dijele na **agregatne** i **skalarne** (engl. *aggregate* i *scalar*). Agregatne funkcije obrađuju skupove podataka tako da uzimaju sve vrijednosti iz zadanog skupa i pretvaraju ih u jednu jedinstvenu vrijednost. S druge strane, skalarne funkcije rade na individualnim redovima podataka, neovisno o ostalim redovima, i za svaki redak vraćaju obrađeni rezultat.

Agregatne funkcije koriste se za rad s više redaka podataka, agregirajući ih u jednu vrijednost. Primjeri takvih funkcija uključuju:

- SUM() – zbroj;
- AVG() – prosjek seta vrijednosti;
- COUNT() – broj redaka;
- MAX() – maksimalna vrijednost;
- MIN() – minimalna vrijednost.

Skalarne funkcije primjenjuju se na pojedinačne retke podataka i za svaki redak vraćaju rezultat obrade. Primjeri uključuju:

- UPPER() – pretvara tekst u velika slova;
- LOWER() – pretvara tekst u mala slova;

- LEN() – vraća dužinu teksta;
- ROUND() – zaokružuje brojeve;
- GETDATE() – vraća trenutni datum i vrijeme.

Ove funkcije omogućuju fleksibilnu obradu i analizu podataka u bazi, prilagođavajući se specifičnim potrebama korisnika.

3.3. SELECT naredba

SELECT naredba spada pod DQL naredbu. Koristi se za dohvaćanje podataka iz baze podataka gdje se podaci mogu dohvaćati pod različitim uvjetima. Tablica 4. prikazuje načine korištenja SELECT izjave.

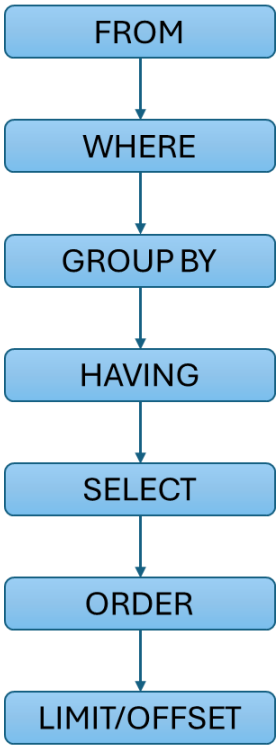
Tablica 4. SQL SELECT naredba

SQL SELECT naredba	OBJAŠNJENJE	
SELECT * FROM nazivTablice	DOHVATI SVE	Dohvaća sve stupce i retke iz tablice *nazivTablice*.
SELECT nazivStupac1, nazivStupac2 **FROM** nazivTablice	DOHVATI POBROJANE STUPCE	Dohvaća stupce nazivStupac1 i nazivStupac2 te sve retke iz tablice nazivTablice.
SELECT nazivStupac **AS** '<novo ime>' **FROM** nazivTablice	PREIMENUJ	Dohvaća vrijednosti *nazivStupac* za sve retke iz tablice te preimenuje stupac u *<novo ime>* iz tablice *nazivTablice*.
SELECT nazivStupac1 **AS** '<novo ime1>', nazivStupac2 **AS** '<novo ime2>' **FROM** nazivTablice	PREIMENUJ	Dohvaća sve vrijednosti za *nazivStupac1* te ga preimenuje u *<novo ime1> i nazivStupac2* te ga preimenuje u *<novo ime2>* iz tablice *nazivTablice*.
SELECT CONCAT (ime, ' ', prezime) **AS** 'zaposlenik' **FROM** nazivTablice	SPAJANJE STUPACA	Spajaju se vrijednosti iz 2 stupca i prikazuju u jednom stupcu naziva *zaposlenik*. Podaci se razdvajaju drugim parametrom CONCAT funkcije (ako je razmaknica u ispisu se ispisuje *ime prezime*).

3.3.1. Redoslijed izvršavanja operacija

U SQL-u redoslijed izvršavanja operacija (engl. *order of operations*) određuje način na koji se različiti dijelovi upita izvršavaju i kako se podaci obrađuju. Razumijevanje ovog redoslijeda ključno je za pisanje točnih i učinkovitih upita. SQL operacije izvršavaju se sljedećim redoslijedom:

1. **FROM:** Prvo se određuje izvor podataka (tablice, pogledi) i provode se svi potrebni spojevi (JOIN).
2. **WHERE:** Primjenjuju se uvjeti za filtriranje podataka na temelju specificiranih kriterija.
3. **GROUP BY:** Podaci se grupiraju prema navedenim stupcima, ako je to specificirano.
4. **HAVING:** Filtriraju se grupe podataka koje su već grupirane s GROUP BY.
5. **SELECT:** Odabiru se i izračunavaju traženi stupci ili izrazi.
6. **ORDER BY:** Sortiraju se rezultati prema zadanim stupcima i redoslijedu.
7. **LIMIT/OFFSET:** Ograničava se broj rezultata koji će biti vraćeni i eventualno preskaču određeni redovi.



Slika 10. Redoslijed izvršavanja operacija u SQL naredbama

3.3.2. Ključna riječ WHERE

U SQL-u ključna riječ WHERE služi za definiranje uvjeta koji moraju biti zadovoljeni kako bi se podaci dohvatili ili obradili. Dakle, WHERE se koristi za postavljanje uvjeta (engl. *condition*) unutar upita.

Primjer:

```
SELECT *
FROM zaposlenik
WHERE odjel = 'IT';
```

- *Ukoliko se iz tablice zaposlenik žele dohvatiti zaposlenik koji rade u odjelu "IT", koristit će se WHERE za definiranje uvjeta. Uvjet je: odjel = 'IT', što znači da će se dohvatiti samo oni redci gdje je vrijednost u stupcu odjel jednaka "IT".*

3.4. Operatori kod izvršavanja SQL naredbi

3.4.1. Logički operatori i prioritet izvršavanja

Ukoliko je potrebno zadovoljiti više uvjeta istovremeno ili barem jedan od uvjeta, utoliko se mogu koristiti logički operatori. Logički operatori omogućuju definiranje složenijih uvjeta unutar SQL upita, čime se preciznije filtriraju podaci prema zadanim kriterijima.

Najčešće korišteni logički operatori u SQL-u:

- **AND** – Koristi se kada je potrebno zadovoljiti više uvjeta istovremeno. Svi navedeni uvjeti moraju biti istiniti kako bi se redak uključio u rezultat.

  Primjer:

  ```
  SELECT * FROM zaposlenik
  WHERE odjel = 'IT' AND placa > 50000; ;
  ```

  - *Oba uvjeta (odjel = 'IT' i placa > 50000) moraju biti istinita za uključivanje reda u rezultat.*

- **OR** – Koristi se kada je dovoljno da barem jedan od uvjeta bude zadovoljen. Svaki uvjet povezan operatorom OR pokreće novi set uvjeta, novi filter koji se neovisno procjenjuje.

  Primjer:

  ```
  SELECT * FROM zaposlenik
  WHERE odjel = 'IT' OR placa > 50000;
  ```

  - *Barem jedan uvjet mora biti istinit za uključivanje reda u rezultat.*

- **NOT** – Koristi se za negaciju uvjeta, tj. za isključivanje redaka koji zadovoljavaju određeni uvjet.

  Primjer:

  ```
  SELECT * FROM zaposlenik
  WHERE NOT odjel = 'HR';
  ```

  - *Ovdje se isključuju svi redci gdje je odjel = 'HR'.*

Osim ovih osnovnih operatora, logički izrazi mogu se kombinirati sa zagradama **()** za **definiranje prioriteta izvršavanja**:

Primjer:

```
SELECT * FROM zaposlenik
WHERE (odjel = 'IT' AND placa > 50000) OR odjel = 'HR';
```

- *Zagrade se koriste za osiguravanje ispravnog prioriteta uvjeta. U ovom primjeru, prvo se evaluira uvjet odjel = 'IT' AND placa > 50000, a zatim se rezultat kombinira s uvjetom odjel = 'HR'.*

Ukoliko ne koristimo zagrade izraz se drugačije tumači. SQL se procjenjuje prema pravilima prioriteta, gdje AND ima viši prioritet od OR, osim ako se redoslijed nije jasno definirao zagradama.

Primjer:

```
SELECT * FROM zaposlenik
WHERE odjel = 'IT' OR odjel = 'HR' AND placa > 50000;
```

- *Ovdje će se uvjet odjel = 'HR' AND placa > 50000 evaluirati zajedno jer AND ima viši prioritet. To znači da će se uključiti svi zaposlenik iz odjela "IT" a zaposlenik iz odjela "HR" samo ako imaju plaću veću od 50.000.*

3.4.2. Operatori usporedbe

Operatori usporedbe igraju ključnu ulogu u filtriranju podataka jer omogućuju definiranje uvjeta unutar WHERE klauzule, HAVING klauzule ili drugih dijelova SQL upita. Uvjeti postavljeni pomoću ovih operatora određuju koji će se redovi dohvatiti, ažurirati ili obrisati.

Tablica 5. Operatori usporedbe

Operator	Opis	Primjer
=	Provjerava jesu li dvije vrijednosti jednake.	placa = 50000
<> ili !=	Provjerava jesu li dvije vrijednosti različite.	odjel <> 'IT'
>	Provjerava je li vrijednost veća od druge.	godine > 30
<	Provjerava je li vrijednost manja od druge.	placa < 70000
>=	Provjerava je li vrijednost veća ili jednaka drugoj.	iskustvo >= 5
<=	Provjerava je li vrijednost manja ili jednaka drugoj.	budžet <= 100000

3.4.3. Filtriranje podataka s BETWEEN ... AND

Operator **BETWEEN ... AND** koristi se za filtriranje podataka unutar određenog intervala, uključujući i granične vrijednosti. Naredba **NOT BETWEEN ... AND** koristi se za isključivanje podataka unutar intervala. Može se koristiti s brojevima, datumima, pa čak i tekstualnim vrijednostima. Ekvivalentna logika za BETWEEN a AND b bi bila WHERE stupac >= a AND stupac <= b.

Sintaksa:

```
SELECT stupac1, stupac2 FROM tablica
WHERE stupac BETWEEN vrijednost1 AND vrijednost2;
```

Primjeri:

```
SELECT * FROM proizvod WHERE cijena BETWEEN 100 AND 500;
```

- *Filtriranje brojeva*

```
SELECT * FROM narudzba WHERE
datum_narudzbe BETWEEN '2024-01-01' AND '2024-12-31';
```

- *Filtriranje datuma- svi datumi unutar 2024. godine*

```
SELECT * FROM korisnik WHERE prezime BETWEEN 'A' AND 'M';
```

- *Filtriranje teksta - sve korisnike čija prezimena počinju slovima od A do M*

3.4.4. Redoslijed operatora

Redoslijed operatora u SQL-u odnosi se na način na koji SQL procesor evaluira izraze unutar uvjeta, poput onih u WHERE, HAVING, ili ON klauzuli. SQL koristi standardni prioritet operatora, slično matematičkim pravilima, kako bi odredio redoslijed evaluacije.

Redoslijed operatora u SQL-u:

1. **Zagrade ()** – Izrazi unutar zagrada evaluiraju se prvi i imaju najviši prioritet.
2. **Aritmetički operatori:**
   - *, /, % (množenje, dijeljenje, modul) imaju viši prioritet od +, - (zbrajanje, oduzimanje).
3. **Operatori usporedbe:**
   - =, <>, !=, >, <, >=, <=, BETWEEN, LIKE, IN.
4. **Logički operatori:**
   - NOT – negacija ima viši prioritet od AND. Svi uvjeti povezani operatorom AND moraju biti zadovoljeni prije OR. OR ima najniži prioritet među logičkim operatorima.

Ukoliko operatori imaju isti prioritet evaluacije, njihov redoslijed izvršavanja ovisi o zadanoj asocijativnosti. Oni se evaluiraju s lijeva prema desno ili s desna prema lijevo, ovisno o pravilima za pojedine operatore. Ovo pravilo, poznato kao asocijativnost operatora[4], osigurava redoslijed obrade kada su svi operatori unutar izraza istog prioriteta.

[4] https://www.postgresql.org/docs/current/sql-syntax-lexical.html#SQL-PRECEDENCE

3.5. Trovrijedna logika

Trovrijedna logika (engl. *Three-Valued Logic, 3VL*) odnosi se na logički sustav koji koristi tri vrijednosti:

1. **TRUE (točno);**
2. **FALSE (netočno);**
3. **UNKNOWN (nepoznato).**

Koristi se za rad s NULL vrijednostima koje predstavljaju nepoznate ili nepostojeće podatke. NULL je posebna vrsta podatka u bazi podataka koja označava nedostatak informacije ili podatke koji još nisu uneseni. NULL nije isto što i "nula" (0) niti prazni *string* (''). Ključna karakteristika NULL vrijednosti je da predstavlja "nepoznato", što znači da se ne može uspoređivati standardnim operatorima na način na koji uspoređujemo druge vrijednosti.

SQL koristi 3VL u logičkim izrazima kod WHERE uvjeta, CASE izraza i AND, OR, NOT operatora. Uvjeti u WHERE klauzuli filtriraju samo TRUE rezultate i odbacuju FALSE i UNKNOWN. UNKNOWN je rezultat logičkog izraza kada NULL sudjeluje u usporedbama ili logičkim operacijama. Pravila evaluacije trovrijedne logike su:

Tablica 6. AND (konjunkcija) s UNKNOWN

Vrijednost 1	Vrijednost 2	Rezultat
TRUE	UNKNOWN	UNKNOWN
FALSE	UNKNOWN	FALSE
UNKNOWN	TRUE	UNKNOWN
UNKNOWN	FALSE	FALSE
UNKNOWN	UNKNOWN	UNKNOWN

Tablica 7. OR (disjunkcija) s UNKNOWN

Vrijednost 1	Vrijednost 2	Rezultat
TRUE	UNKNOWN	TRUE
FALSE	UNKNOWN	UNKNOWN
UNKNOWN	TRUE	TRUE
UNKNOWN	FALSE	UNKNOWN
UNKNOWN	UNKNOWN	UNKNOWN

Tablica 8. NOT (negacija) s UNKNOWN

Vrijednost	Rezultat
UNKNOWN	UNKNOWN

3.6. IS operator

Sve operacije usporedbe ili aritmetičke operacije nad NULL podacima vraćaju NULL kao rezultat, jer rezultat interakcije s nepoznatim podatkom ostaje nepoznat.

Primjer:

```
NULL + 5 → rezultat je NULL.
NULL = NULL → rezultat je također NULL, a ne TRUE, kako bi
se moglo očekivati.
```

Zbog ovoga, standardni operatori poput = ili != ne rade ispravno s NULL vrijednostima. Umjesto toga, treba koristiti IS operator za provjeru uvjeta koji uključuju NULL.

Provjera je li vrijednost NULL:

```
SELECT * FROM tablica WHERE naziv_stupca IS NULL;
```

Provjera je li vrijednost različita od NULL:

```
SELECT * FROM tablica WHERE naziv_stupca IS NOT NULL;
```

Ova sintaksa osigurava pravilno rukovanje NULL vrijednostima i omogućuje točno filtriranje podataka u upitima.

COALESCE je funkcija u SQL-u koja se koristi za rukovanje NULL vrijednostima. Njena osnovna svrha je vraćanje prve ne-NULL vrijednosti iz liste izraza koju prima kao argumente.

Primjer:

```
SELECT ime, COALESCE(email, 'nema e-mail adrese') AS
email_adresa FROM korisnik;
```

- *ako stupac email sadrži NULL, funkcija vraća 'nema e-mail adrese' kao zadanu vrijednost*

3.7. Filtriranje podataka s IN/NOT IN

Operator **IN** koristi se za provjeru podudarnosti vrijednosti u skupu specificiranih vrijednosti. Pomaže filtrirati podatke koji odgovaraju jednom ili više elemenata iz liste vrijednosti. Lista može sadržavati brojeve, tekst, datume ili rezultat podupita.

Primjer:

```
SELECT stupac1, stupac2 FROM tablica WHERE stupac IN
(vrijednost1, vrijednost2, vrijednost3, ...);
```

- *(vrijednost1, vrijednost2, vrijednost3, ...) predstavlja listu vrijednosti s kojima se uspoređuje stupac.*

Izvršava se poput višestrukih OR uvjeta, primjerice: WHERE stupac = vrijednost1 OR stupac = vrijednost2 OR stupac = vrijednost3. Korištenje IN čini upit kraćim i preglednijim. Može se koristiti i WHERE stupac **NOT IN** (vrijednost1, vrijednost2, vrijednost3); pri čemu se isključuju navedene vrijednosti.

Primjeri:

```
SELECT * FROM proizvod WHERE cijena IN (100, 200, 300);
```

- *Filtriranje prema brojevima. Vraća sve proizvode čije su cijene 100, 200 ili 300.*

```
SELECT * FROM korisnik WHERE grad IN ('Zagreb', 'Split',
'Rijeka');
```

- *Filtriranje prema tekstualnim vrijednostima. Vraća sve korisnike iz gradova Zagreb, Split ili Rijeka.*

```
SELECT * FROM narudzba WHERE datum_narudzbe IN
('2024-01-01', '2024-06-01', '2024-12-31');
```

- *Filtriranje prema datumima. Vraća narudžbe na navedene datume.*

```
SELECT * FROM zaposlenik WHERE odjel_id IN (10, 20, NULL);
```

- *Rezultat će uključivati samo retke gdje je odjel_id = 10 ili odjel_id = 20, ali neće uključivati redak gdje je odjel_id = NULL, jer NULL nije jednak NULL*

```
SELECT * FROM zaposlenik WHERE odjel_id IN (10, 20) OR
odjel_id IS NULL;
```

- *Ako želimo uključiti i NULL, koristimo OR IS NULL*

```
SELECT * FROM zaposlenik WHERE odjel_id NOT IN (10, 20,
NULL);
```

- *Upit neće vratiti ništa, jer MySQL vidi NULL u skupu i ne može pouzdano usporediti vrijednosti (UNKNOWN rezultat).*

```
SELECT * FROM zaposlenik WHERE odjel_id IS NOT NULL AND
odjel_id NOT IN (10, 20);
```

- *Osiguravamo da NULL ne uzrokuje neočekivano ponašanje.*

```
SELECT ime FROM zaposlenik
WHERE odjel IN (
    SELECT odjel FROM odjeli WHERE lokacija = 'Zagreb'
);
```

- *IN s podupitom. U primjeru se vraćaju svi zaposlenik čiji su odjeli smješteni u Zagrebu.*

3.8. Pretraživanje podudaranja s LIKE

Operator LIKE koristi se za pretraživanje podataka u bazama podataka prema uzorku podudaranja. Najčešće se koristi za rad s tekstualnim podacima, gdje se koriste zamjenski znakovi za definiranje uzorka.

Zamjenski znakovi (engl. *wildcards*):

- % - nula ili više znakova. Zamjenjuje bilo koji broj znakova.
- _ - jedan znak. Zamjenjuje točno jedan znak.

Primjeri:

```
SELECT ime FROM zaposlenik WHERE ime LIKE 'Ana%';
```

- *Sva imena koja počinju s 'Ana' (npr. Ana, Anamarija)*

```
SELECT ime FROM zaposlenik WHERE ime LIKE '%ko%';
```

- *Sva imena koja sadrže 'ko' (npr. Marko, Nikolina)*

```
SELECT ime FROM zaposlenik WHERE ime LIKE 'A_a';
```

- *Sva imena od tri slova gdje srednje slovo može biti bilo što (npr. Ana, Ava)*

```
SELECT ime FROM zaposlenik WHERE ime LIKE '_ara';
```

- *Sva imena s četiri slova koja završavaju na 'ara' (npr. Sara, Mara)*

```
SELECT ime FROM zaposlenik WHERE ime LIKE '_n%';
```

- *Kombinacija % i _. Imena gdje drugo slovo mora biti 'n' (npr. Ante, Anita)*

Osjetljivost na mala i velika slova (engl. *Case sensitivity*) ovisi o SQL implementaciji i bazi podataka. MySQL baza nije osjetljiva na velika/mala slova prema zadanim postavkama (može se mijenjati). PostgreSQL baza je osjetljiva na velika/mala slova (koristi **ILIKE** za neosjetljivo pretraživanje). Kod SQL Servera ponašanje ovisi o postavkama kolacije (engl. *collation*).

PostgreSQL baza koristi LIKE samo na tekstualnim podacima pa ukoliko želimo podudaranja s primjerice brojevima, trebali bi ih najprije pretvoriti (engl. *cast*) u tekst. Pretvorba se radi s **CAST** ključnom riječi, primjerice: CAST(broj AS text) ili izjavom broj::text.

NOT LIKE se koristi za traženje vrijednosti koje ne odgovaraju uzorku.

Ako uzorak sadrži % ili _ kao obične znakove koristi se **ESCAPE** znak kao u primjeru:

```
WHERE popust LIKE '50\% popusta' ESCAPE '\';
```

3.9. DISTINCT ključna riječ

DISTINCT je ključna riječ u SQL-u koja se koristi za uklanjanje duplikata iz rezultata upita. Omogućuje da se vrate samo jedinstvene vrijednosti iz jednog ili više stupaca.

Sintaksa je:

```
SELECT DISTINCT stupac1, stupac2, ... FROM tablica;
```

- *Ako je upit SELECT DISTINCT stupac1, tada se vraćaju jedinstvene vrijednosti iz tog stupca. Ukoliko je upit SELECT DISTINCT stupac1, stupac2 – traži se jedinstvenost kombinacija vrijednosti.*

3.10. Uvjetne izjave (engl. *conditional statements*)

3.10.1. CASE

CASE izjave u SQL-u koriste se za uvjetnu logiku unutar SQL upita. Omogućuju definiranje različitih izlaznih vrijednosti ovisno o uvjetima koji se postavljaju. Vrlo su korisne za prilagodbu rezultata upita, izračune i grupiranja.

Postoje dvije glavne varijante CASE izraza:

1. **Jednostavna** (engl. *Simple*) CASE izjava uspoređuje **vrijednost** sa specifičnim vrijednostima.

   Primjer:

   ```
   SELECT zaposlenik_id,
          CASE odjel_id
              WHEN 1 THEN 'Ljudski resursi'
              WHEN 2 THEN 'Financije'
              WHEN 3 THEN 'IT'
              ELSE 'Ostalo'
          END AS naziv_odjela
   FROM zaposlenik;
   ```

   - *Ovdje CASE provjerava odjel_id i vraća odgovarajući naziv odjela.*

2. **Prema uvjetu** (Searched) CASE izjava provjerava **uvjete** pomoću logičkih izraza.

   Primjer:

   ```
   SELECT zaposlenik_id, placa,
          CASE
              WHEN placa < 3000 THEN 'Niska'
              WHEN placa BETWEEN 3000 AND 7000 THEN 'Srednja'
              ELSE 'Visoka'
          END AS kategorija_place
   FROM zaposlenik;
   ```

   - *Ovdje CASE analizira vrijednosti placa i dodjeljuje kategoriju.*

CASE izrazi se koriste unutar SELECT, WHERE, ORDER BY ili HAVING klauzula. Uvijek završavaju s END. ELSE je opcionalan, ali se preporučuje da pokrije slučajeve kad nijedan uvjet nije ispunjen. Ako nijedan uvjet nije zadovoljen i nema ELSE, CASE vraća NULL.

3.10.2. Uvjetne izjave u vlastitim funkcijama (engl. *custom functions*)

Ukoliko su potrebni složeniji uvjeti, utoliko se mogu definirati vlastite funkcije koje uključuju uvjetnu logiku.

Primjer:

```
CREATE FUNCTION KategorijaPlace(placa DECIMAL)
RETURNS VARCHAR(10)
BEGIN
    RETURN CASE
        WHEN placa < 4000 THEN 'Niska'
        WHEN placa BETWEEN 4000 AND 8000 THEN 'Srednja'
        ELSE 'Visoka'
    END;
END;
```

Primjena funkcije:

```
SELECT zaposlenik_id, placa, KategorijaPlace(placa) AS
kategorija_place FROM zaposlenik;
```

3.10.3. NULLIF

NULLIF uspoređuje dva izraza i vraća NULL ako su jednaki, inače vraća prvu vrijednost. Koristi se za izbjegavanje podjele s nulom ili za posebne slučajeve logike.

Primjer:

```
SELECT iznos, broj_transakcija,
    iznos / NULLIF(broj_transakcija, 0) AS prosjek
FROM transakcije;
```

- *NULLIF(broj_transakcija, 0) vraća NULL ako je broj_transakcija = 0, inače vraća vrijednost broj_transakcija.*
- *Ako je rezultat NULL, dijeljenje (iznos / NULL) vraća NULL, što SQL interpretira kao valjani rezultat umjesto greške. Greška bi nastala ukoliko bi dijelili s 0.*

3.10.4. IF-ELSE (u procedurama i funkcijama)

Za uvjetnu logiku unutar običnih SQL upita koristi se samo CASE izraz. CASE je deklarativan i služi za vraćanje različitih vrijednosti ovisno o uvjetima, ali ne može upravljati kontrolom toka poput IF-ELSE. Standardni SQL je deklarativan, što znači da opisuje što se želi postići (npr. dohvatiti podatke, ažurirati tablicu), ali ne specificira kako to treba biti izvedeno. S druge strane, proceduralni ili imperativni jezici opisuju kako obaviti zadatak korak po korak, koristeći kontrolu toka (npr. petlje, grananja, uvjete). IF-ELSE se može koristiti samo unutar

proceduralnih jezika baze podataka (kao što su PL/pgSQL u PostgreSQL-u, T-SQL u SQL Serveru ili proceduralni SQL u MySQL-u). IF-ELSE je striktno proceduralna kontrola toka, što znači da je rezervirana za okruženja koja podržavaju proceduralne jezike unutar baze, poput procedura, funkcija ili blokova koda (BEGIN...END). Proceduralni jezici unutar SQL-a podržavaju naredbe poput IF, CASE, LOOP, FOR, WHILE i druge kontrole toka.

Primjer:

```
DELIMITER //
CREATE PROCEDURE ProvjeraPopusta(IZNOS NARUDŽBE
DECIMAL(10,2))
BEGIN
    IF iznos_narudzbe > 100 THEN
        SELECT 'Ispunjeni uvjeti za popust';
    ELSE
        SELECT 'Nema popusta';
    END IF;
END //
DELIMITER;
```

3.11. Odabir podataka iz više tablica - JOIN

U bazama podataka podaci su često raspodijeljeni u više tablica koje su međusobno povezane putem ključeva, poput primarnih i stranih ključeva. Kako bi se analizirali podaci iz više tablica istovremeno, koriste se JOIN operacije. Ove operacije omogućuju kombiniranje redaka rezultata iz dvije ili više tablica na temelju zajedničkih stupaca. Stupac jedne tablice mapira se sa stupcem druge tablice kako bi se uspostavila veza između podataka. SQL naredba JOIN omogućuje dohvaćanje podataka iz povezanih tablica, čime se pruža cjelovita slika informacija koje bi inače bile razdvojene.

Vrste JOIN-ova:

1. **INNER JOIN:** Vraća samo redove s podudaranjem u obje tablice.
2. **LEFT JOIN (LEFT OUTER JOIN):** Vraća sve redove iz lijeve tablice i podudaranja iz desne, ako ih ima. Ako nema podudaranja, vrijednosti iz desne tablice bit će NULL.
3. **RIGHT JOIN (RIGHT OUTER JOIN):** Vraća sve redove iz desne tablice i podudaranja iz lijeve. Ako nema podudaranja, vrijednosti iz lijeve tablice bit će NULL.
4. **FULL JOIN (FULL OUTER JOIN):** Kombinira rezultate iz LEFT JOIN i RIGHT JOIN, vraćajući sve redove iz obje tablice s podudaranjima ili bez njih.
5. **CROSS JOIN:** Vraća kartezijanski produkt tablica (svaka kombinacija redaka iz prve i druge tablice).
6. **SELF JOIN:** Kada se tablica povezuje sama sa sobom, obično radi analize odnosa unutar iste tablice.

U primjerima pojedinih vrsta JOIN-ova naredbe i njihovi rezultati objašnjeni su na sljedeće dvije tablice:

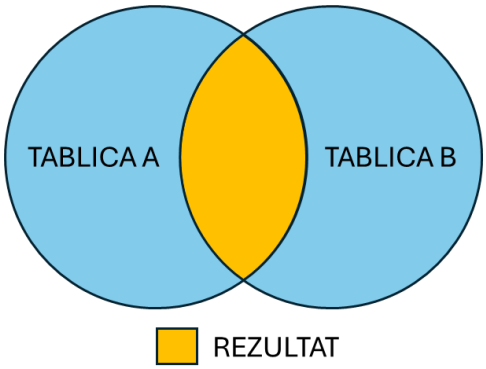
Tablica 9. Tablica u bazi naziva *kupac*

kupac_ID	ime	grad
1	Ana	Zagreb
2	Marko	Split
3	Ivana	Rijeka

Tablica 10. Tablica u bazi naziva *narudzba*

narudzba_ID	kupac_ID	iznos
101	1	250
102	2	400
103	4	150

3.11.1. INNER JOIN



Slika 11. Rezultati INNER JOIN naredbe

INNER JOIN je SQL operacija koja vraća intersekciju podataka iz dvije ili više tablica. To znači da dohvaća samo one retke koji imaju podudaranje u svim tablicama na temelju uvjeta spajanja. Vraća samo zajedničke podatke iz tablica gdje je uvjet spajanja (ON) zadovoljen. Redovi bez podudaranja neće biti uključeni u rezultate.

JOIN je skraćeni zapis za INNER JOIN i ukoliko u SQL izjavi imamo zapisan samo JOIN podrazumijevano je to INNER JOIN.

Primjer:

```
SELECT kupac.ime, kupac.grad, narudzba.narudzba_ID,
narudzba.iznos
FROM kupac
INNER JOIN narudzba ON kupac.kupac_ID = narudzba.kupac_ID;
```

- *Tablice kupac i narudzba spajaju se na temelju zajedničkog atributa kupac_ID.*
- *ON kupac.kupac_ID = narudzba.kupac_ID određuje pronalaženje svih redova gdje je kupac_ID u tablici kupac jednak kupac_ID u tablici narudzba.*

Tablica 11. Rezultat INNER JOIN naredbe

ime	grad	narudzba_ID	iznos
Ana	Zagreb	101	250
Marko	Split	102	400

```
SELECT ime, grad, narudzba_ID, iznos
FROM kupac INNER JOIN narudzba
USING (kupac_ID);
```

- *Isto rješenje se dobiva kao u prethodnom primjeru.*
- *Ključna riječ USING automatski prepoznaje zajednički stupac.*
- *Pojednostavljuje upit kada tablice imaju stupce istog imena.*
- *Smanjuje redundanciju u rezultatima jer zajednički stupac prikazuje samo jednom.*

Lijeva tablica (**FROM** tablica A) postavlja osnovu za usporedbu. Redovi iz lijeve tablice uspoređuju se s desnom tablicom (JOIN tablica B) na temelju uvjeta spajanja. Redovi iz lijeve tablice koji imaju podudaranje u desnoj bit će uključeni, a rezultati će zadržati strukturu definiranu ORDER BY klauzulom, ali osnovni redoslijed često ovisi o lijevoj tablici. Logički, lijeva tablica obično predstavlja primarne podatke, dok desna dodaje dodatne informacije. Kod INNER JOIN-a, lijeva tablica daje osnovu za usporedbu redaka s desnom tablicom.

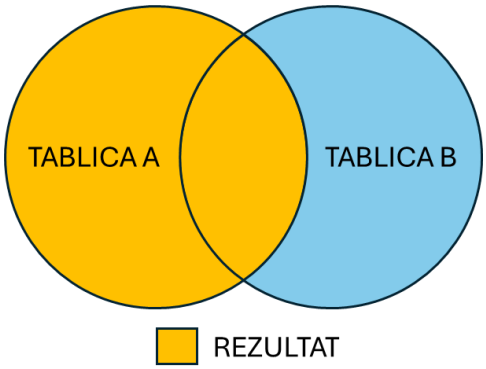
Kako bi se skratio upit često se primjenjuju aliasi (nadimci). Aliasi omogućuju kraći i pregledniji zapis te pojednostavljuju pisanje uvjeta. U slučaju JOIN operacija, za aliase nije potrebno koristiti ključnu riječ AS. Sljedeći primjer prikazuje korištenje aliasa:

Primjer:

```
SELECT k.ime, k.grad, n.narudzba_ID, n.iznos
FROM kupac k INNER JOIN narudzba n
ON k.kupac_ID = n.kupac_ID;
```

- *k je alias za tablicu kupac. n je alias za tablicu narudzba.*
- *Kod postaje pregledniji i lakši za pisanje. Posebno je koristan kod tablica koje imaju duga imena.*

3.11.2. LEFT OUTER JOIN – LEFT JOIN



Slika 12. Rezultati LEFT OUTER JOIN naredbe

Vraća **sve redove iz lijeve tablice** i odgovarajuće redove iz desne tablice. Ako nema podudaranja, vrijednosti iz desne tablice bit će `NULL`.

Primjer:

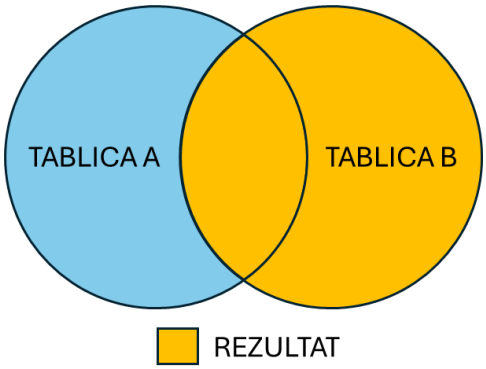
```
SELECT kupac.ime, kupac.grad, narudzba.narudzba_ID,
narudzba.iznos
FROM kupac LEFT JOIN narudzba
ON kupac.kupac_ID = narudzba.kupac_ID;
```

Tablica 12. Rezultat LEFT JOIN naredbe

ime	grad	narudzba_ID	iznos
Ana	Zagreb	101	250
Marko	Split	102	400
Ivana	Rijeka	NULL	NULL

-

3.11.3. RIGHT OUTER JOIN – RIGHT JOIN



Slika 13. Rezultati RIGHT OUTER JOIN naredbe

Vraća **sve redove iz desne tablice** i odgovarajuće redove iz lijeve tablice, Ako nema podudaranja, vrijednosti iz lijeve tablice bit će `NULL`.

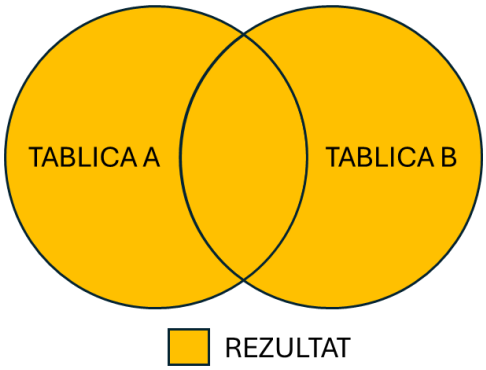
Primjer:

```
SELECT kupac.ime, kupac.grad, narudzba.narudzba_ID,
narudzba.iznos
FROM kupac RIGHT JOIN narudzba
ON kupac.kupac_ID = narudzba.kupac_ID;
```

Tablica 13. Rezultat RIGHT JOIN naredbe

ime	grad	narudzba_ID	iznos
Ana	Zagreb	101	250
Marko	Split	102	400
NULL	NULL	103	150

3.11.4. FULL OUTER JOIN – OUTER JOIN



Slika 14. Rezultati FULL OUTER JOIN naredbe

Vraća sve redove iz obje tablice. Tamo gdje nema podudaranja, vrijednosti iz nepovezane tablice bit će `NULL`.

Primjer:

```
SELECT kupac.ime, kupac.grad, narudzba.narudzba_ID,
narudzba.iznos
FROM kupac FULL OUTER JOIN narudzba
ON kupac.kupac_ID = narudzba.kupac_ID;
```

- *Vraćaju se svi kupci iz tablice kupac, čak i ako nemaju narudžbe te sve narudžbe iz tablice narudzba, čak i ako nisu vezane za nekog kupca.*

Tablica 14. Rezultat FULL OUTER JOIN naredbe

Ime	Grad	narudzba_ID	Iznos
Ana	Zagreb	101	250
Marko	Split	102	400
Ivana	Rijeka	NULL	NULL
NULL	NULL	103	150

3.11.5. CROSS JOIN

Vraća **kartezijanski produkt** (kombinaciju svih redova iz obje tablice). Svaki red iz jedne tablice kombinira se sa svakim redom iz druge.

Primjer:

```
SELECT kupac.ime, kupac.grad, narudzba.narudzba_ID,
narudzba.Iznos
FROM kupac CROSS JOIN narudzba;
```

Tablica 15. Rezultat CROSS JOIN naredbe

Ime	Grad	narudzba_ID	Iznos
Ana	Zagreb	101	250
Ana	Zagreb	102	400
Ana	Zagreb	103	150
Marko	Split	101	250
Marko	Split	102	400
Marko	Split	103	150
Ivana	Rijeka	101	250
Ivana	Rijeka	102	400
Ivana	Rijeka	103	150

3.11.6. SELF JOIN

SELF JOIN je tehnika spajanja tablice sa samom sobom. Koristi se kada podaci unutar iste tablice trebaju biti uspoređeni ili povezani. Može se raditi kada tablica ima strani ključ koji referencira njen primarni ključ ili kada se rade usporedbe unutar iste tablice. Kod SELF JOIN-a tablica se u upitu koristi dva puta s različitim alias-ima, čime se omogućuje razlikovanje dviju instanci iste tablice. Koristi se kod hijerarhijskih struktura kada se analiziraju odnosi između redaka ili kod usporedbe redaka kada se uspoređuju vrijednosti unutar iste tablice.

Primjer 1:

Imamo tablicu zaposlenik gdje je cilj dohvatiti ime svakog zaposlenika i ime njegovog menadžera.

Tablica 16. Tablica u bazi naziva Zaposlenik

zaposlenik_ID	ime	menadzer_ID
1	Ana	NULL
2	Marko	1
3	Ivana	1
4	Petar	2

```
SELECT z1.ime AS zaposlenik_ime, z2.ime AS menadzer_ime
FROM Zaposlenik z1 LEFT JOIN Zaposlenik z2
ON z1.menadzer_ID = z2.zaposlenik_ID;
```

- *ON z1.menadzer_ID = z2.zaposlenik_ID povezuje zaposlenike (z1) s menadžerima (z2).*
- *Alias-i z1 i z2 omogućuju razlikovanje dviju instanci iste tablice.*
- *Koristi se LEFT JOIN kako bi dobili sve zaposlenike u rezultatima a ne samo one koji imaju definiranog menadžera.*

Tablica 17. Rezultat SELF JOIN naredbe

zaposlenik_ime	menadzer_ime
Ana	NULL
Marko	Ana
Ivana	Ana
Petar	Marko

Primjer 2:

Tablica 18. Tablica u bazi naziva Proizvodi

ProizvodID	Proizvod	Cijena
1	Laptop	900
2	Miš	30
3	Tipkovnica	75
4	Monitor	250
5	Web kamera	75
6	USB kabel	30

```
SELECT p1.Proizvod AS Proizvod1, p2.Proizvod AS Proizvod2
FROM Proizvodi p1 JOIN Proizvodi p2
ON p1.Cijena = p2.Cijena AND p1.ProizvodID <>
p2.ProizvodID;
```

- *p1.Cijena = p2.Cijena: provjerava podudaranje cijena dok p1.ProizvodID <> p2.ProizvodID osigurava da se uspoređuju različiti proizvodi*
- *Pronaći će sve parove proizvoda koji imaju istu cijenu, ali različite ID-e.*

Tablica 19. Rezultat SELF JOIN naredbe

Proizvod1	Proizvod2
Miš	USB kabel
USB kabel	Miš
Tipkovnica	Web kamera
Web kamera	Tipkovnica

- *Ukoliko se želi dobiti samo jedinstvene rezultate kao uvijete bi postavili*

```
ON p1.Cijena = p2.Cijena AND p1.ProizvodID <
p2.ProizvodID;
```

3.12. UNION i UNION ALL

UNION je SQL operator koji kombinira rezultate dvaju ili više SELECT upita u jedan rezultat. Koristi se za objedinjavanje podataka iz više izvora (tablica, pogleda ili upita), dok uklanja dupliciranе redove iz konačnog rezultata (osim ako se koristi UNION ALL).

Kombinira rezultate dvaju ili više SELECT upita u jedan skup podataka. Broj stupaca mora biti isti u svim upitima. Redoslijed i tipovi podataka stupaca u svim upitima moraju biti kompatibilni.

Primjer:

Tablica 20. kupac

ime	grad
Ana	Zagreb
Marko	Split

Tablica 21. posjetitelj

ime	grad
Ivana	Rijeka
Ana	Zagreb

```
SELECT ime, grad FROM kupac
UNION SELECT ime, grad FROM posjetitelj;
```

- *UNION – bez duplikata:*
- *Duplicirani redak ("Ana, Zagreb") pojavljuje se samo jednom.*

Tablica 22. Rezultat s UNION

ime	grad
Ana	Zagreb
Marko	Split
Ivana	Rijeka

```
SELECT ime, grad FROM kupac
UNION ALL
SELECT ime, grad FROM Posjetitelji;
```

- *UNION ALL – zadržava duplikate*

Tablica 23. Rezultat s UNION ALL

ime	grad
Ana	Zagreb
Marko	Split
Ivana	Rijeka
Ana	Zagreb

Dohvaćanje podataka iz iste tablice na temelju različitih uvjeta pomoću UNION omogućuje kombiniranje rezultata iz različitih SELECT upita u jedan zajednički rezultat.

Primjer:

Tablica 24. Zaposlenik

ZaposlenikID	Ime	Odjel	Plaća
1	Ana	Marketing	900
2	Marko	IT	1500
3	Ivana	Prodaja	1200
4	Petar	Marketing	950
5	Martina	IT	2000

```
SELECT Ime, Odjel, Plaća
FROM Zaposlenik
WHERE Odjel = 'Marketing'
UNION
SELECT Ime, Odjel, Plaća
FROM Zaposlenik
WHERE Plaća > 1500;
```

- *Zadatak je dohvatiti zaposlenike iz odjela Marketing i zaposlenike čija je plaća veća od 1.500.*

Tablica 25. Rezultat UNION na istoj tablici

Ime	Odjel	Plaća
Ana	Marketing	900
Petar	Marketing	950
Martina	IT	2000

4. NAPREDNI SQL

4.1. Naredba GROUP BY

GROUP BY je SQL naredba koja se koristi za grupiranje redova u tablici na temelju jednog ili više stupaca. Omogućuje izvođenje agregatnih funkcija na grupama podataka umjesto na cijeloj tablici. Želi se reducirati podatke za jednu grupu na jednu vrijednost. Redovi s istim vrijednostima u specificiranim stupcima grupiraju se zajedno. Svaka grupa ima jednu vrijednost za stupac(e) koji se koristi za grupiranje.

Najčešće se koristi zajedno s agregatnim funkcijama poput:

- `SUM()` – zbroj vrijednosti;
- `COUNT()` – broj redova u grupi;
- `AVG()` – prosjek vrijednosti;
- `MIN()` i `MAX()` – minimalna i maksimalna vrijednost u grupi.

Primjer:

Tablica 26. Prikaz tablice Prodaja

Prodavac	Proizvod	Iznos
Ana	Laptop	900
Marko	Miš	30
Ana	Tipkovnica	75
Marko	Monitor	250
Ana	Miš	30

```
SELECT Prodavac, SUM(Iznos) AS UkupnoProdano
FROM Prodaja GROUP BY Prodavac;
```

- *Redovi iz tablice grupiraju se prema vrijednostima u stupcu Prodavac.*
- *Za svaku grupu (Ana i Marko) izračunava se zbroj stupca Iznos.*

Tablica 27. Prikaz rezultata upita s GROUP BY

Prodavac	UkupnoProdano
Ana	1005
Marko	280

Model **podijeli-primjeni-kombiniraj** (engl. *Split – Apply – Combine*) opisuje način na koji SQL koristi GROUP BY za obradu podataka. Model se izvršava kako slijedi:

1. **Podijeli (engl. *split*):**
   Podaci u tablici se dijele u grupe na temelju vrijednosti u stupcu (ili stupcima) specificiranim u naredbi GROUP BY. Svaka grupa sadrži redove koji imaju iste vrijednosti u određenom stupcu.
   Ako tablicu 26. grupiramo prema stupcu Prodavac, tablica se "dijeli" u dvije grupe:
   - Grupa za Ana: (Laptop - 900, Tipkovnica - 75, Miš - 30);
   - Grupa za Marko: (Miš - 30, Monitor - 250).
2. **Primijeni (engl. *apply*):**
   Na svaku grupu primjenjuju se agregatne funkcije (npr. SUM, AVG, COUNT). Ove funkcije obrađuju podatke unutar svake grupe. U primjeru koristimo SUM(Iznos) te SQL izračunava zbroj vrijednosti unutar svake grupe:
   - Za Ana: 900 + 75 + 30 = 1005;
   - Za Marko: 30 + 250 = 280.
3. **Kombiniraj (engl. *combine*):**
   Rezultati agregatnih funkcija iz svake grupe kombiniraju se u konačni rezultat. Konačni rezultat sadrži jednu vrijednost za svaku grupu kako je prikazano u Tablica 27. Prikaz rezultata upita s GROUP BY.

4.1.1. GROUP BY GROUPING SET

GROUPING SETS omogućuje definiranje višestrukih razina grupiranja u jednom GROUP BY upitu, bez potrebe za korištenjem više UNION operacija. To je korisno kad trebamo agregate za različite kombinacije stupaca unutar jednog upita.

Primjer:

Tablica 28. Prodaja

Prodavac	Proizvod	Regija	Iznos
Ana	Laptop	Sjever	900
Marko	Miš	Jug	30
Ana	Tipkovnica	Sjever	75
Marko	Monitor	Jug	250
Ivana	Miš	Istok	100

```
SELECT Prodavac, Regija, SUM(Iznos) AS UkupnaProdaja
FROM Prodaja
GROUP BY GROUPING SETS (
    (Prodavac),       -- Grupiranje po prodavaču
    (Regija),         -- Grupiranje po regiji
    ()                -- Sveukupno (bez grupiranja)
);
```

- *Izračunava se ukupna prodaja po prodavaču, regiji i sveukupno.*

Tablica 29. Rezultat GROUPING SETS

Prodavac	Regija	UkupnaProdaja
Ana	NULL	975
Marko	NULL	280
Ivana	NULL	100
NULL	Sjever	975
NULL	Jug	280
NULL	Istok	100
NULL	NULL	1355

Isti rezultat moguće je dobiti i s UNION kako slijedi:

```
SELECT Prodavac, NULL AS Regija, SUM(Iznos) AS
UkupnaProdaja FROM Prodaja GROUP BY Prodavac
UNION
SELECT NULL AS Prodavac, Regija, SUM(Iznos) AS
UkupnaProdaja FROM Prodaja GROUP BY Regija
UNION
SELECT NULL AS Prodavac, NULL AS Regija, SUM(Iznos) AS
UkupnaProdaja FROM Prodaja;
```

GROUPING SETS i UNION omogućuju postizanje sličnih rezultata pri agregaciji podataka, ali se razlikuju po načinu implementacije i efikasnosti. GROUPING SETS dio je GROUP BY klauzule i omogućuje definiranje višestrukih razina grupiranja unutar jednog SQL upita. To smanjuje potrebu za ponovljenim prolazima kroz podatke jer se sve kombinacije grupiranja obrađuju u jednom prolazu. S druge strane, UNION kombinira rezultate više odvojenih SELECT upita, što često zahtijeva višestruke prolaze kroz podatke. Dok je GROUPING SETS kompaktniji i čitljiviji za scenarije s kompleksnim grupiranjem, UNION pruža fleksibilnost u pisanju različitih upita, ali uz cijenu dodatne složenosti i potencijalno niže performanse. Za složenije analize s višestrukim razinama grupiranja, GROUPING SETS je obično efikasniji izbor.

4.1.2. ROLLUP i CUBE

ROLLUP je dodatak GROUP BY klauzuli koji automatski generira hijerarhijske agregate podataka. Koristi se za stvaranje međuzbrojeva i ukupnih vrijednosti (engl. *subtotals* i *grand totals*) za različite razine grupiranja unutar jednog upita.

Svaki stupac u **ROLLUP**-u stvara novu razinu grupiranja. Dodaje redove koji prikazuju međuzbrojeve za svaku razinu grupiranja. Na kraju, uključuje sveukupni zbroj za sve redove.

Primjer:

Tablica 30. Prodaja

Prodavac	Proizvod	Iznos
Ana	Laptop	900
Ana	Monitor	250
Ana	Tipkovnica	75
Marko	Miš	30
Marko	Monitor	300
Marko	Monitor	250

```
SELECT Prodavac, Proizvod, SUM(Iznos) AS Ukupno
FROM Prodaja
GROUP BY ROLLUP (Prodavac, Proizvod)
ORDER BY Prodavac, Proizvod;
```

Tablica 31. Rezultat i objašnjenje ROLLUP

Prodavac	Proizvod	Ukupno	Objašnjenje
Ana	Laptop	900	Zbroj svih Laptopa koje je Ana prodala.
Ana	Monitor	250	Zbroj svih Monitora koje je Ana prodala.
Ana	Tipkovnica	75	Zbroj svih Tipkovnica koje je Ana prodala.
Ana	NULL	1225	Ukupno sve što je Ana prodala.
Marko	Miš	30	Zbroj svih Miševa koje je Marko prodao.
Marko	Monitor	550	Zbroj svih Monitora koje je Marko prodao.
Marko	NULL	580	Ukupno sve što je Marko prodao.
NULL	NULL	1805	Sveukupni zbroj za sve prodavače i proizvode.

ROLLUP je koristan za analizu podataka, izvještavanje i stvaranje hijerarhijskih prikaza podataka (npr. prodaja po regiji, odjelima ili prodavačima).

CUBE generira sve moguće kombinacije grupiranja, uključujući grupiranje po svim stupcima, pojedinačnim stupcima i ukupne vrijednosti (engl. *grand total*).

Primjer:

```
SELECT Prodavac, Proizvod, SUM(Iznos) AS Ukupno
FROM Prodaja
GROUP BY CUBE (Prodavac, Proizvod)
ORDER BY Prodavac, Proizvod;
```

Tablica 32. Rezultat i objašnjenje CUBE

Prodavac	Proizvod	Ukupno	Objašnjenje
Ana	Laptop	900	Zbroj svih Laptopa koje je prodala Ana.
Ana	Monitor	250	Zbroj svih Monitora koje je prodala Ana.
Ana	Tipkovnica	75	Zbroj svih Tipkovnica koje je prodala Ana.
Ana	NULL	1225	Ukupno sve što je Ana prodala.
Marko	Miš	30	Zbroj svih Miševa koje je prodao Marko.
Marko	Monitor	550	Zbroj svih Monitora koje je prodao Marko.
Marko	NULL	580	Ukupno sve što je Marko prodao.
NULL	Laptop	900	Ukupno prodano Laptopa.
NULL	Miš	30	Ukupno prodano Miševa.
NULL	Monitor	800	Ukupno prodano Monitora.
NULL	Tipkovnica	75	Ukupno prodano Tipkovnica.
NULL	NULL	1805	Sveukupni zbroj za sve prodavače i proizvode.

ROLLUP i CUBE su dodatci SQL-u za generiranje višestrukih razina grupiranja, ali se razlikuju po opsegu kombinacija koje stvaraju. ROLLUP generira hijerarhijsko grupiranje, počevši od svih specificiranih stupaca prema manje detaljnim razinama, završavajući sveukupnim zbrojem. Na primjer, omogućuje grupiranje po prodavaču, zatim međuzbroj po svakom prodavaču, i na kraju ukupni zbroj. CUBE, s druge strane, proširuje funkcionalnost generiranjem svih mogućih kombinacija grupiranja, uključujući ukupne zbrojeve po pojedinom stupcu (npr. ukupan zbroj za svaki proizvod, bez obzira na prodavača) i ukupne zbrojeve za sve kombinacije. Ukratko, ROLLUP je fokusiran na hijerarhiju, dok CUBE pruža potpunu fleksibilnost i pokriva sve kombinacije.

4.1.3. Naredba HAVING

HAVING je SQL klauzula koja se koristi za filtriranje grupa podataka nakon što su grupirane naredbom GROUP BY. Dok se WHERE koristi za filtriranje redova prije grupiranja, HAVING djeluje na grupe podataka i često uključuje uvjete s agregatnim funkcijama poput SUM, AVG, COUNT, itd. Filtrira grupe na temelju uvjeta (npr. SUM > 1000).

Razlika između WHERE i HAVING:

- WHERE: Filtrira sve pojedinačne redove tabele prije grupiranja;
- HAVING: Filtrira grupe podataka nakon grupiranja.

Sljedeći primjer primjenjuje SQL na Tablicu 26.

Primjer:

```
SELECT Prodavac, SUM(Iznos) AS UkupnoProdano
FROM Prodaja GROUP BY Prodavac
HAVING SUM(Iznos) > 500;
```

- *Filtrira grupe i uključuje samo one gdje je ukupna prodaja veća od 500.*
- *Grupa za Marko se isključuje jer je njegov ukupni iznos 280, što je manje od 500.*

Tablica 33. Prikaz rezultata upita s GROUP BY i HAVING

Prodavac	UkupnoProdano
Ana	1005

4.2. ORDER BY

ORDER BY je SQL naredba koja se koristi za sortiranje rezultata upita prema jednom ili više stupaca. Sortiranje može biti uzlazno (ASC) i silazno (DESC).

Sljedeći primjer primjenjuje SQL na Tablicu 26.

Primjer:

```
SELECT Prodavac, Proizvod, Iznos
FROM Prodaja ORDER BY Iznos ASC;
```

Tablica 34. Prikaz rezultata upita s GROUP BY i HAVING

Prodavac	Proizvod	Iznos
Marko	Miš	30
Ana	Miš	30
Ana	Tipkovnica	75
Marko	Monitor	250
Ana	Laptop	900

Naredba ORDER BY koristi se za sortiranje podataka iz tablica prema jednom ili više stupaca u uzlaznom (**ASC**) ili silaznom (**DESC**) redoslijedu. Omogućuje prilagodbu redoslijeda prikaza podataka. Ukoliko nije navedeno drugačije, ORDER BY koristi uzlazni redoslijed (ASC) kao zadanu opciju. ORDER BY se može koristiti i u kombinaciji s funkcijama.

Primjeri:

```
SELECT naziv, cijena FROM proizvod ORDER BY cijena ASC;
```

- *Prikazuje proizvode sortirane prema cijeni od najjeftinijeg do najskupljeg.*

```
SELECT ime, prezime, grad FROM kupac ORDER BY ime ASC,
prezime ASC;
SELECT ime, prezime, grad FROM kupac ORDER BY ime,
prezime;
```

- *Prikazuje kupce abecednim redom prema imenu, a zatim prema prezimenu*

```
SELECT ime, prezime, grad FROM kupac ORDER BY LENGTH(ime)
ASC;
```

- *Sortira rezultate prema duljini imena (ime) u uzlaznom redoslijedu (ASC).*
- *Najkraća imena prikazat će se prva, a dulja dolaze kasnije.*

```
SELECT ime, prezime, grad FROM kupac ORDER BY LENGTH(ime)
ASC, prezime ASC;
```

- *Sortira prema duljini imena u uzlaznom redoslijedu.*
- *Ako više redaka ima istu duljinu imena, dodatno ih sortira prema prezimenu.*

4.3. *WINDOW* FUNKCIJE

Window funkcije u SQL-u vrlo su moćan alat za rad s podacima jer omogućuju izvođenje izračuna preko "prozora" (engl. *window*) redaka, a da ne grupiraju podatke kao GROUP BY. To znači da se mogu koristiti agregati, rangiranja i razlike u redovima unutar skupa podataka, dok se zadržava pristup svim originalnim redovima. GROUP BY grupira podatke i vraća jednu vrijednost po grupi, dok *window* funkcije omogućuju agregaciju unutar prozora, zadržavajući sve redove u rezultatu. *Window* funkcije omogućuju napredne analize i rangiranja bez gubitka detalja u redovima.

Kako bi se koristili prozori primjenjuje se naredba **OVER** koja definira "prozor" (skup redaka) nad kojima se funkcija primjenjuje. Originalni redovi se čuvaju, ne grupiraju se podatci, pa svaki red ostaje u rezultatu. *Window* funkcije stvaraju novi stupac u rezultatima na temelju funkcija izvedenih na podskupu ili 'prozoru' podataka.

PARTITION BY je dio SQL *window* funkcija koji omogućuje dijeljenje podataka u manje grupe (particije) na temelju vrijednosti u određenim stupcima. Funkcija se zatim primjenjuje unutar svake particije zasebno, dok se zadržavaju svi redovi u tablici. Ako PARTITION BY nije specificiran, funkcija se primjenjuje na cijeli skup podataka, cijelu tablicu ili na dio tablice koji je filtriran sa WHERE naredbom.

Prozori se primjenjuju kod sljedećih funkcija:

1. Agregacija (npr. SUM, AVG, COUNT) bez grupiranja;
2. Rangiranja (npr. RANK, ROW_NUMBER, DENSE_RANK);
3. Pomaci između redaka (npr. LAG, LEAD).

Tablica 35. Popis funkcija koje se mogu koristiti s *window* funkcijama

Funkcija	Svrha
SUM/MIN/MAX/ AVG	Izračunava zbroj, minimum, maksimum ili prosjek svih vrijednosti u particiji.
FIRST_VALUE	Vraća vrijednost iz prvog reda unutar particije.
LAST_VALUE	Vraća vrijednost iz zadnjeg reda unutar particije.
NTH_VALUE	Vraća vrijednost iz n-tog reda u particiji (prema ORDER BY).
PERCENT_RANK	Izračunava relativni rang trenutnog reda: (rang - 1) / (ukupan broj redova - 1).
RANK	Rangira trenutni redak unutar particije, dopuštajući praznine u rangu.
DENSE_RANK	Rangira trenutni redak bez praznina u rangu.
ROW_NUMBER	Dodjeljuje broj reda, počevši od 1 za svaki red u particiji.
LAG/LEAD	Pristupa vrijednostima iz prethodnog ili sljedećeg reda.
COUNT	Broji broj redova u particiji ili okviru.

CUME_DIST	Vraća kumulativnu distribuciju trenutnog reda unutar particije: udio redova manjih ili jednakih trenutnom.
NTILE	Dijeli particiju na n jednake grupe i vraća broj grupe za svaki red.

Primjeri:

Tablica 36. Zaposlenik

ZaposlenikID	Ime	Odjel	Placa
1	Ana	IT	1500
2	Marko	IT	2000
3	Ivana	IT	1800
4	Petar	Prodaja	1700
5	Martina	Prodaja	2100
6	Luka	Marketing	1900
7	Ema	Marketing	2200

```
SELECT *,
MAX(Placa) OVER () AS NajvecaPlaca
FROM Zaposlenik;
```

- *Izračunava najveću plaću u cijeloj tablici Zaposlenik.*
- *OVER () funkcija bez argumenata znači da se izračunava bez particioniranja ili dodatnog sortiranja – dakle, uzima cijeli skup podataka.*

Tablica 37. Rezultat MAX(Placa) OVER ()

ZaposlenikID	Ime	Odjel	Placa	NajvecaPlaca
1	Ana	IT	1500	2200
2	Marko	IT	2000	2200
3	Ivana	IT	1800	2200
4	Petar	Prodaja	1700	2200
5	Martina	Prodaja	2100	2200
6	Luka	Marketing	1900	2200
7	Ema	Marketing	2200	2200

```
SELECT *,
MAX(Placa) OVER (PARTITION BY Odjel) AS NajvecaPlaca,
MAX(Placa) OVER (PARTITION BY Odjel) - Placa AS Razlika
FROM Zaposlenik;
```

- *Računa se maksimalna plaća na odjelu i razlika plaće pojedinca od maksimalne na odjel*

Tablica 38. Rezultat MAX(Placa) OVER (PARTITION BY Odjel)

ZaposlenikID	Ime	Odjel	Plaća	NajvecaPlaca	Razlika
1	Ana	IT	1500	2000	500

2	Marko	IT	2000	2000	0
3	Ivana	IT	1800	2000	200
4	Petar	Prodaja	1700	2100	400
5	Martina	Prodaja	2100	2100	0
6	Luka	Marketing	1900	2200	300
7	Ema	Marketing	2200	2200	0

```
SELECT Ime, Odjel, Placa,
SUM(Placa) OVER (PARTITION BY Odjel) AS
UkupnaPlacaPoOdjelu
FROM Zaposlenik;
```

- *računa se ukupna plaća po odjelu, prozor je Odjel – PARTITION BY Odjel*
- *za svakog zaposlenika prikazuje se plaća i ukupnu plaću u njegovom odjelu*

Tablica 39. Rezultat SUM(Placa) OVER (PARTITION BY Odjel)

Ime	Odjel	Placa	UkupnaPlacaPoOdjelu
Ana	IT	1500	5300
Marko	IT	2000	5300
Ivana	IT	1800	5300
Petar	Prodaja	1700	3800
Martina	Prodaja	2100	3800
Luka	Marketing	1900	4100
Ema	Marketing	2200	4100

```
SELECT Ime, Odjel, Placa,
RANK() OVER (PARTITION BY Odjel ORDER BY Placa DESC) AS
Rang FROM Zaposlenik;
```

- *za svakog zaposlenika, prikazuje njegov rang prema plaći unutar odjela.*

Tablica 40. Rezultat RANK() OVER (PARTITION BY Odjel ORDER BY Placa DESC)

Ime	Odjel	Placa	Rang
Marko	IT	2000	1
Ivana	IT	1800	2
Ana	IT	1500	3
Martina	Prodaja	2100	1
Petar	Prodaja	1700	2
Ema	Marketing	2200	1
Luka	Marketing	1900	2

```
SELECT Ime, Odjel, Placa,
LAG(Placa) OVER (PARTITION BY Odjel ORDER BY Placa) AS
PrethodnaPlaca,
Placa - LAG(Placa) OVER (PARTITION BY Odjel ORDER BY
Placa) AS Razlika FROM Zaposlenik;
```

- *Funkcija LAG(Placa) uzima plaću prethodnog zaposlenika unutar istog odjela (prema rastućem redoslijedu plaća).*
- *Izračun Placa - LAG(Placa) daje razliku u plaći između trenutnog i prethodnog zaposlenika*

Tablica 41. Rezultat LAG(Placa) izjave

Ime	Odjel	Placa	PrethodnaPlaca	Razlika
Ana	IT	1500	NULL	NULL
Ivana	IT	1800	1500	300
Marko	IT	2000	1800	200
Petar	Prodaja	1700	NULL	NULL
Martina	Prodaja	2100	1700	400
Luka	Marketing	1900	NULL	NULL
Ema	Marketing	2200	1900	300

```
SELECT Ime, Odjel, Placa,
ROW_NUMBER() OVER (PARTITION BY Odjel ORDER BY Placa DESC)
AS RedniBroj FROM Zaposlenik;
```

- *ROW_NUMBER() daje jedinstveni redni broj svakom zaposleniku unutar njegovog odjela – grupirano prema odjelu, sortirano prema plaći u silaznom redoslijedu*

Tablica 42. Rezultat ROW_NUMBER() izjave

Ime	Odjel	Placa	RedniBroj
Marko	IT	2000	1
Ivana	IT	1800	2
Ana	IT	1500	3
Martina	Prodaja	2100	1
Petar	Prodaja	1700	2
Ema	Marketing	2200	1
Luka	Marketing	1900	2

4.3.1. Definiranje okvira (engl. *framing*)

Framing je način preciznog ograničavanja opsega redaka koji ulaze u izračun za *window* funkcije. Koristi se u kombinaciji s klauzulom OVER i definira točan okvir ili raspon redaka unutar particije (ili cijele tablice) za izračune poput SUM, AVG, ROW_NUMBER, i drugih agregatnih funkcija.

Ako se ne koristi klauzula okvira (FRAME), tada se ORDER BY ponaša tako da predefinirano uključuje sve prethodne redove (od početka particije) do trenutnog reda u izračun. Prilikom korištenja klauzule okvira (engl. *FRAME clause*) unutar *window* funkcije, možemo definirati pod-raspon ili okvir redaka koji će biti uključeni u izračun. ORDER BY koristi se za određivanje redoslijeda redova prije nego što se primijeni klauzula okvira, a klauzula okvira

omogućuje precizno specificiranje redaka koji ulaze u izračun, relativno prema trenutnom redu.

Osnovne vrste specifikacija okvira:

1. **ROWS** – Definira okvir prema redoslijedu redaka.
2. **RANGE** – Definira okvir na temelju vrijednosti u stupcu, odnosi se na vrijednost sadržaja reda u stupcu definiranom u ORDER BY.
3. **GROUPS** – Koristi se za grupiranje redaka koji dijele iste vrijednosti i izračuni se rade unutar grupa.

Tablica 43. Ključne riječi kod kreiranja okvira

Ključna riječ	Značenje
ROWS ili RANGE	Definira da li se koristi raspon vrijednosti (RANGE) ili broj redaka (ROWS) kao okvir.
GROUPS	Definira redove unutar grupa s istim vrijednostima iz ORDER BY kao okvir unutar kojeg se radi izračun.
PRECEDING	Redovi prije trenutnog reda.
FOLLOWING	Redovi nakon trenutnog reda.
UNBOUNDED PRECEDING	Uključuje sve redove prije trenutnog reda u particiji.
UNBOUNDED FOLLOWING	Uključuje sve redove nakon trenutnog reda u particiji.
CURRENT ROW	Uključuje samo trenutni red.

*Primjeri **ROWS**:*

Tablica 44. Prodaja

ID_Prodaje	Proizvod	Datum	Kolicina	Cijena
1	Jabuka	2025-01-01	10	5
2	Banana	2025-01-02	20	3
3	Jabuka	2025-01-03	15	5
4	Grožđe	2025-01-04	10	4
5	Banana	2025-01-05	25	3
6	Jabuka	2025-01-06	20	5

```
SELECT  Proizvod, Datum,
SUM(Kolicina) OVER (PARTITION BY Proizvod ORDER BY Datum)
AS Kumulativna_Kolicina
FROM prodaja;
```

- *Predefinirano ponašanje*
- *Podijele se redovi u particije prema vrijednostima stupca Proizvod i unutar svake particije, redovi se poredaju prema vrijednosti u stupcu Datum (kronološki redoslijed).*

- *SUM(Kolicina): Računa kumulativni zbroj stupca Količina od prvog do trenutnog reda u particiji*

Tablica 45. Rezultat naredbe sa zadanim ponašanjem s predefiniranim okvirom

Proizvod	Datum	Kumulativna_Kolicina
Jabuka	2025-01-01	10
Jabuka	2025-01-03	25
Jabuka	2025-01-06	45
Banana	2025-01-02	20
Banana	2025-01-05	45
Grožđe	2025-01-04	10

```
SELECT  Proizvod, Datum,
SUM(Kolicina) OVER (PARTITION BY Proizvod ORDER BY Datum
ROWS BETWEEN 1 PRECEDING AND CURRENT ROW) AS
Kumulativna_Kolicina
FROM prodaja;
```

- *Okviri s ROWS: precizno kontroliranje redaka*
- *1 PRECEDING: Uključuje jedan redak neposredno prije trenutnog reda*
- *CURRENT ROW: Uključuje trenutni red.*

Tablica 46. Rezultat naredbe sa okvirom
ROWS BETWEEN 1 PRECEDING AND CURRENT ROW

Proizvod	Datum	Kumulativna_Količina
Jabuka	2025-01-01	10
Jabuka	2025-01-03	25
Jabuka	2025-01-06	35
Banana	2025-01-02	20
Banana	2025-01-05	45
Grožđe	2025-01-04	10

```
SELECT Proizvod, Datum,
SUM(Kolicina) OVER (PARTITION BY Proizvod ORDER BY Datum
ROWS BETWEEN CURRENT ROW AND 1 FOLLOWING) AS
Kumulativna_Kolicina
FROM prodaja;
```

- *CURRENT ROW: Uključuje trenutni red i 1 FOLLOWING: Uključuje sljedeći red.*
- *Računa kumulativni zbroj Kolicina koristeći samo trenutni red i sljedeći red unutar particije (ako postoji). Ako je trenutni red posljednji, zbroj uključuje samo njega.*

Tablica 47. Rezultat naredbe sa okvirom
ROWS BETWEEN CURRENT ROW AND 1 FOLLOWING

Proizvod	Datum	Kumulativna_Količina
Jabuka	2025-01-01	25
Jabuka	2025-01-03	35
Jabuka	2025-01-06	20
Banana	2025-01-02	45
Banana	2025-01-05	25
Grožđe	2025-01-04	10

*Primjeri **RANGE**:*

Tablica 48. Placanja

ID_Placanja	Klijent	Datum	Iznos
1	Ana	2025-01-01	100
2	Ana	2025-01-03	200
3	Ana	2025-01-05	300
4	Ivan	2025-01-01	150
5	Ivan	2025-01-04	250
6	Ivan	2025-01-06	350

```
SELECT Klijent, Datum, Iznos,
SUM(Iznos) OVER (PARTITION BY Klijent ORDER BY Datum
RANGE BETWEEN INTERVAL 2 DAY PRECEDING AND CURRENT ROW) AS
Kumulativni_Iznos
FROM placanja;
```

- *PARTITION BY Klijent: Dijeli redove po klijentima (Ana i Ivan)*
- *ORDER BY Datum: poslaguje transakcije unutar svakog klijenta prema datumu (kronološki).*
- *RANGE BETWEEN INTERVAL 2 DAY PRECEDING AND CURRENT ROW: Računa kumulativni iznos za sve redove unutar intervala od 2 dana unazad do trenutnog reda.*
- *Fokusira se na vrijednosti datuma (ne na broj redova kao ROWS).*
- *Za postgreSQL bazu sintaksa je RANGE BETWEEN '2 days'::INTERVAL PRECEDING AND CURRENT ROW*

Tablica 49. Rezultat naredbe sa okvirom
RANGE BETWEEN INTERVAL 2 DAY PRECEDING AND CURRENT ROW

ID_Placanja	Klijent	Datum	Iznos	Kumulativni_Iznos
1	Ana	2025-01-01	100	100
2	Ana	2025-01-03	200	300
3	Ana	2025-01-05	300	500
4	Ivan	2025-01-01	150	150
5	Ivan	2025-01-04	250	250
6	Ivan	2025-01-06	350	350

*Primjeri **GROUPS**:*

Tablica 50. Transakcije

Klijent	Kategorija	Iznos
Ana	Hrana	100
Ana	Hrana	50
Ana	Odjeća	200
Ana	Odjeća	300
Ana	Tehnika	400

```
SELECT Klijent, Kategorija, Iznos,
SUM(Iznos) OVER (PARTITION BY Klijent
ORDER BY Kategorija
GROUPS BETWEEN 1 PRECEDING AND CURRENT ROW) AS
Kumulativni_Iznos
FROM transakcije;
```

- *Naprave se prozori prema klijentu (PARTITION BY Klijent).*
- *Unutar svakog prozora, podaci se podijele prema vrijednostima u stupcu za sortiranje (npr. ORDER BY Kategorija), gdje svaka vrijednost postaje jedna grupa.*
- *Za svaki redak koji se obrađuje, uključuju se cijela prethodna grupa (1 PRECEDING) i trenutna grupa kojoj redak pripada (CURRENT ROW).*
- *Zatim se izračunava vrijednost (npr. SUM) koristeći sve redove iz tih dviju grupa.*

Tablica 51. Rezultat s okvirom GROUPS BETWEEN 1 PRECEDING AND CURRENT ROW

Klijent	Kategorija	Iznos	Kumulativni_Iznos
Ana	Hrana	100	150
Ana	Hrana	50	150
Ana	Odjeća	200	650
Ana	Odjeća	300	650
Ana	Tehnika	400	1100

4.4. *VIEW* SINTAKSA

Views (pogledi) u SQL-u su virtualne tablice koje se temelje na rezultatima SQL upita. One prikazuju podatke koji se dohvaćaju iz jedne ili više spremljenih tablica i rezultat su ranije provedenog SQL upita. Omogućuju da se rade SQL upiti na već ranije napravljenim upitima.

Ključne značajke:

- *View* ne pohranjuje podatke fizički u bazi već koristi podatke iz izvorišnih tablica.
- Kada se promijene podaci u izvorišnim tablicama i *view* se automatski ažurira.
- Omogućuju ograničavanje pristupa podacima jer korisnici mogu vidjeti samo određene stupce ili redove definirane u *view*-u. Prikazujete samo podatke koji su potrebni korisnicima.
- Kompleksni upiti mogu se sažeti u *view*, olakšavajući ponovne upite.

Primjer:

Tablica 52. Zaposlenik

id	ime	odjel	placa
1	Ana	Prodaja	5000
2	Marko	IT	7000
3	Ivana	Prodaja	5500

```
CREATE VIEW prodaja_view AS
SELECT ime, placa
FROM zaposlenik
WHERE odjel = 'Prodaja';
```

- *view koji prikazuje samo zaposlenike iz odjela Prodaja*

```
SELECT * FROM prodaja_view;
```

- *korištenje view*

Tablica 53. Rezultat čitanja iz kreiranog *view*

ime	plaća
Ana	5000
Ivana	5500

Performanse mogu biti lošije kod kompleksnih upita u view-ima, posebno ako se koriste često i na velikim skupovima podataka.

U SQL-u postoje dvije vrste pogleda: **materijalizirani pogled (engl. *materialized view*)** i **nematerijalizirani pogled *(engl. non-materialized view*).** Obje vrste koriste se za prikaz podataka, ali se razlikuju u načinu pohrane i izvedbi.

Nematerijalizirani pogled virtualna je tablica koja se temelji na SQL upitu. Ona ne pohranjuje podatke fizički, već svaki put kada se pozove, izvodi se originalni upit nad

izvornim tablicama. Nema pohrane i podaci nisu fizički spremljeni, već se generiraju u stvarnom vremenu kada korisnik izvrši upit. Sprema se samo definicija pogleda a ne podaci koje pogled vraća. Uvijek prikazuje najnovije podatke iz izvorišnih tablica. Jednostavan je za održavanje i pogodan za jednostavne upite. Nepogodan je za složene ili dugotrajne upite. Sporiji je u izvedbi jer se originalni upit mora izvoditi svaki put. Raniji primjer prikazuje nematerijalizirani pogled.

Materijalizirani pogled fizički je spremljena kopija podataka koji proizlaze iz SQL upita. Podaci u materijaliziranom pogledu pohranjuju se u bazi podataka i mogu se povremeno ažurirati. Spremanje podataka omogućuje brže izvođenje upita jer se izbjegava ponovno izvršavanje originalnog upita. Podaci u materijaliziranom pogledu mogu postati zastarjeli, pa je potrebno osvježavanje (manualno ili automatsko) kako bi bili sinkronizirani s izvornim podacima. Koristi se u situacijama gdje je potrebno optimizirati performanse, osobito za kompleksne ili dugotrajne upite.

Primjer:

```
CREATE MATERIALIZED VIEW prodaja_view AS
SELECT ime, placa FROM zaposlenik
WHERE odjel = 'Prodaja';
```

Tablica 54. Neke od uobičajenih naredbi koje se izvršavaju nad pogledima

Naredba	Sintaksa	Objašnjenje
CREATE VIEW	CREATE VIEW *naziv* AS *select_upit*;	Kreira novi pogled temeljen na rezultatu SQL upita.
REPLACE VIEW	CREATE OR REPLACE VIEW *naziv* AS *select_upit*;	Ako pogled već postoji, zamjenjuje ga novim definiranim upitom. Ako ne postoji, kreira ga.
ALTER VIEW	ALTER VIEW *naziv* AS *select_upit*; ALTER VIEW *naziv* RENAME TO *noviNaziv*	Mijenja definiciju postojećeg pogleda (ovisi o bazi podataka, ponekad se koristi CREATE OR REPLACE VIEW umjesto).
DROP VIEW	DROP VIEW [IF EXISTS] *naziv*;	Briše definirani pogled iz baze podataka.
UPDATE View	UPDATE *naziv* SET stupac = vrijednost WHERE uvjet;	Ažurira podatke u izvorišnim tablicama putem jednostavnog pogleda koji podržava ažuriranje.
REFRESH View	REFRESH MATERIALIZED VIEW *naziv*;	Ažurira podatke u materijaliziranom pogledu kako bi bili usklađeni s izvorišnim tablicama (koristi se za materijalizirane poglede).
GRANT View	GRANT SELECT ON *naziv* TO korisnik;	Dodjeljuje prava pristupa (npr. SELECT) korisnicima za određeni pogled.
REVOKE View	REVOKE SELECT ON *naziv* FROM korisnik;	Uklanja prava pristupa korisnicima za određeni pogled.

4.5. Indeksi

Indeksi su posebne strukture u bazama podataka koje ubrzavaju pretraživanje i dohvaćanje podataka iz tablica. To su pokazivači na podatke u tabeli. Oni funkcioniraju kao sadržaj knjige – omogućuju brži pronalazak željene informacije bez potrebe za pretraživanjem svakog retka tablice. Indeksi značajno smanjuju vrijeme potrebno za izvršavanje upita, posebno kod velikih tablica.

Indeksi se obično postavljaju na stupce koji se često koriste u:

- Uvjetima WHERE;
- Sortiranju (ORDER BY);
- JOIN operacijama;
- Grupiranju (GROUP BY).

Vrste indeksa:

1. **Implicitni indeksi (engl. *Implicit indexes*):** Ovo su indeksi koji se automatski stvaraju u bazi podataka. Primjeri uključuju indekse za primarne ključeve, jedinstvena ograničenja (UNIQUE) i vanjske ključeve (FOREIGN KEY).
2. **Primarni indeks (engl. *Primary index*):** Automatski se stvara kada se definira primarni ključ. Osigurava jedinstvenost i koristi se za pretraživanje.
3. **Jedinstveni indeks (engl. *Unique index*):** Osigurava da vrijednosti u indeksiranom stupcu budu jedinstvene.
4. **Složeni indeks (engl. *Composite indeks, multi-column*):** Kreira se kao kombinacija dvaju ili više stupaca.
5. **Djelomični indeks (engl. *Partial*):** Specifična podvrsta indeksa gdje indeks uključuje samo one retke koji zadovoljavaju određeni uvjet (obično koristi WHERE klauzulu prilikom stvaranja).
6. **Klasterirani indeks (engl. *Clustered index*):** Fizički raspoređuje podatke u tablici prema vrijednostima indeksa.
7. **Neklasterirani indeks (engl. *Non-clustered index*):** Sadrži pokazivače (pointere) na stvarne podatke u tablici.

Nedostaci indeksa uključuju zauzimanje dodatnog prostora na disku i sporije spremanje podataka u bazu (dodavanje, ažuriranje i brisanje podataka) jer treba osvježiti i indekse.

Sintaksa:

```
CREATE INDEX ime_indeksa ON naziv_tablice (stupac);
```

- *Kreiranje jednostavnog indeksa*

```
CREATE UNIQUE INDEX ime_indeksa ON naziv_tablice (stupac);
```

- *Kreiranje jedinstvenog indeksa*

```
CREATE INDEX ime_indeksa ON naziv_tablice (stupac1,
stupac2);
```

- *Složeni indeks (više stupaca)*

```
DROP INDEX ime_indeksa;
```

- *Brisanje indeksa*

```
CREATE INDEX djelomicni_indeks ON tablica (stupac) WHERE uvjet;
```

- *Djelomični indeks koji sadrži WHERE filtriranje*

Indeksi su ključni alat za optimizaciju baza podataka, ali ih treba koristiti pažljivo kako bi se izbjeglo nepotrebno opterećenje sustava. Sljedećih bi se pravila trebalo pridržavati prilikom izrade indeksa:

- Dodavanje indeksa treba imati svrhu, jer nepotrebni indeksi povećavaju zauzeće prostora i usporavaju operacije pisanja.
- Indeksi ne bi trebali biti korišteni na malim tablicama. Kod malih tablica, pretraživanje cijele tablice može se obaviti brzo i bez indeksa.
- Indeksi ne bi trebali biti korišteni na tablicama koje se često ažuriraju jer se kod ažuriranja tablica trebaju ažurirati i indeksi
- Indeksi ne bi trebali biti korišteni na stupcima koji mogu sadržavati NULL vrijednosti.
- Indeksi ne bi trebali biti korišteni na stupcima koji sadrže velike vrijednosti.
- Indeksiranje velikih podataka, kao što su TEXT ili BLOB, može zauzimati previše prostora i biti neučinkovito.
- Indeksi ne bi trebali biti korišteni na stupcima s malo jedinstvenih vrijednosti. Na primjer, stupci s vrijednostima da/ne ili muško/žensko neće značajno smanjiti broj redaka koji se pretražuju.
- Indeksi ne bi trebali biti korišteni na tablicama koje se rijetko pretražuju.
- Indeksi ne bi trebali biti korišteni na privremenim tablicama.
- Kod privremenih tablica, dodavanje indeksa može nepotrebno povećati vrijeme potrebno za njihovo kreiranje i brisanje.

4.5.1. Algoritmi indeksiranja

Algoritmi indeksiranja koriste se u bazama podataka kako bi omogućili učinkovito dohvaćanje podataka. Oni definiraju način na koji se podaci organiziraju, pohranjuju i pretražuju u indeksima. Pravilnim odabirom algoritma mogu se značajno poboljšati performanse baze podataka, posebno za velike tablice i složene upite.

Različite baze podataka koriste različite algoritme indeksiranja zbog razlika u načinu na koji baza pohranjuje i upravlja podacima. Neke baze poput PostgreSQL-a fokusirane su na fleksibilnost i podršku za složene strukture, dok je MySQL optimiziran za brzinu i jednostavnost.

Neki od uobičajenih algoritama indeksiranja su:

1. **B-Tree indeksiranje (engl. *Balanced Tree*)**
   Kod B-Tree indeksiranja podaci se organiziraju u uravnoteženo stablo gdje se svaki čvor povezuje na određeni raspon vrijednosti. Ovo omogućuje brzo pretraživanje, umetanje i brisanje. Podržavaju ga PostgreSQL, MySQL (InnoDB), SQLite i Oracle. Dobar je za pretraživanja u kojima se koriste operatori usporedbe (=, <, >, BETWEEN, IN, IS NULL, IS NOT NULL).

2. **Hash indeksiranje**
   Vrijednosti se pretvaraju u "hash" pomoću funkcije raspršivanja. Podaci se pohranjuju na temelju dobivenih *hash* vrijednosti. Koristi se kod PostgreSQL, MySQL (*Memory* tablice). Ne podržava rasponske upite (<, >), već samo točne pretrage (=).

3. **GIN indeks (engl. *Generalized inverted index*)**
   Organizacija podataka u obliku inverznog indeksa. Koristi se za pretraživanje više vrijednosti unutar jednog polja (npr. nizova, JSON objekata). Podržava ga PostgreSQL. Koristi se za kompleksne tipove podataka, poput punog tekstualnog pretraživanja.

4. **Full-Text indeksiranje**
   Organizacija podataka omogućuje brzo pretraživanje po tekstualnim poljima. Koriste ga MySQL, PostgreSQL. Omogućuje brzo pretraživanje ključnih riječi unutar tekstova, npr. za tražilice.

5. **Spatial indeksiranje**
   Organizacija podataka prilagođena za geometrijske tipove podataka (točke, linije, poligoni). Koriste ga MySQL (Spatial Index), PostgreSQL (PostGIS). Koristi se za prostorne podatke i geografske analize.

Algoritmi indeksiranja prilagođeni su specifičnim potrebama baze i tipovima podataka. Najjednostavniji poput B-Tree i Hash indeksa koriste se za standardne upite i male skupove podataka, dok složeniji poput GIN i prostornog indeksiranja podržavaju specijalizirane slučajeve. Odabir pravog algoritma ključan je za optimizaciju performansi baze podataka.

Algoritmi indeksiranja ne primjenjuju se uobičajeno izravno sintaksom u SQL-u, već se implicitno koriste pri kreiranju različitih tipova indeksa. Baza podataka odabire algoritam koji odgovara tipu indeksa i strukturi podataka. Međutim, korisnik može odabrati željeni tip indeksa (npr. B-Tree, Hash) u skladu s mogućnostima koje baza podataka nudi.

Sintaksa:

```
CREATE INDEX indeks_gin ON tablica USING GIN (stupac);
```

- *u PostgreSQL-u za pretraživanje složenih podataka*

```
CREATE SPATIAL INDEX indeks_prostor ON tablica
(stupac_geometrija);
```

- *postavljanje u MySQL*

Algoritmi indeksiranja zahtijevaju da SQL upiti budu prilagođeni njihovim mogućnostima. Ukoliko upiti nisu usklađeni s algoritmom, utoliko baza može zaobići indeks, što značajno usporava pretraživanje.

Primjerice B-Tree je dobar za rasponske (<, >, BETWEEN) i točne pretrage (=) ali ukoliko koristimo funkcije tjera se baza da pretražuje cijelu tablicu (engl. *full table scan*), a ne indekse, što usporava dohvaćanje.:

Pogrešno:

```
SELECT * FROM narudzbe WHERE YEAR(datum) = 2023;
```

Ispravno:

```
SELECT * FROM narudzbe WHERE datum >= '2023-01-01' AND
datum < '2024-01-01';
```

Hash algoritam je brz za točne pretrage (=) i brže će se izvršavati na takvim upitima nego *B-Tree*, ali ne podržava rasponske (<, >).

Pogrešno:

```
SELECT * FROM korisnici WHERE id > 100;
```

Ispravno:

```
SELECT * FROM korisnici WHERE id = 12345;
```

4.6. Podupiti

Podupiti (engl. *Subqueries*) su SQL upiti unutar drugog SQL upita. Često se nazivaju i unutarnji upiti (engl. *Inner Query, Inner SELECT*). Koriste se za dohvaćanje podataka koji će se zatim koristiti u glavnom upitu. Često služe za rješavanje složenih zadataka poput filtriranja, agregacija ili usporedbe. Postoje različite vrste podupita, ovisno o načinu korištenja. Podupiti moraju biti unutar zagrada. Uvijek se stavljaju s desne strane operatora usporedbe. Njihov rezultat se ne može manipulirati, primjerice, ORDER BY se ignorira. Podupiti se mogu i ugnježđivati.

Pretpostavimo da imamo tablicu zaposlenik sa sljedećim stupcima i podacima:

Tablica 55. Zaposlenik

id	ime	prezime	placa	odjel
1	Ivan	Horvat	8000	IT
2	Ana	Kovačić	9500	IT
3	Marko	Perić	7000	Financije
4	Petra	Novak	6500	Financije
5	Luka	Jurić	8500	Marketing

1. **Podupit u SELECT dijelu**
   Koristi se za izračun dodatnih vrijednosti koje nisu direktno dostupne u glavnoj tablici. Kada se koriste u SELECT dijelu moraju vratiti uvijek jednu vrijednost koja se odabire ili se koriste agregatne funkcije.

   Primjer:

```
SELECT id, ime, prezime, placa,
       (SELECT MAX(placa) FROM zaposlenik) AS
najveca_placa
FROM zaposlenik;
```

   - *Ovaj upit pronalazi sve zaposlenike i dodaje kolonu koja prikazuje najveću plaću u tablici.*

Tablica 56. Rezultat podupita u SELECT dijelu

id	ime	prezime	placa	najveca_placa
1	Ivan	Horvat	8000	9500
2	Ana	Kovačić	9500	9500
3	Marko	Perić	7000	9500
4	Petra	Novak	6500	9500
5	Luka	Jurić	8500	9500

2. **Podupit u WHERE dijelu**

   Podupiti se koriste najčešće u WHERE dijelu. Koriste se za filtriranje podataka na temelju rezultata podupita.

   Primjer:

```
SELECT id, ime, prezime, placa
FROM zaposlenik
WHERE placa > (SELECT AVG(placa) FROM zaposlenik);
```

   - *Ovaj upit pronalazi zaposlenike čija je plaća veća od prosječne plaće.*

Tablica 57. Rezultat podupita u WHERE dijelu

id	ime	prezime	placa
2	Ana	Kovačić	9500
5	Luka	Jurić	8500

3. **Podupit u FROM dijelu**

   Naziva se i "*inline view*". Podupit se tretira kao privremena tablica. Kada se koriste u FROM dijelu mogu vratiti više vrijednosti.

   Primjer:

```
SELECT prosjecna_placa
FROM (SELECT AVG(placa) AS prosjecna_placa
      FROM zaposlenik) AS privremena_tablica;
```

   - *Ovaj upit izračunava prosječnu plaću i prikazuje je kao rezultat.*
   - *(SELECT AVG(placa) AS prosjecna_placa FROM zaposlenik) → unutarnji upit izračunava prosječnu plaću i daje mu naziv stupca prosjecna_placa.*
   - *AS privremena_tablica → dodjeljuje se alias (privremeni naziv) za cijeli rezultat unutarnjeg upita, što znači da se na njega u vanjskom upitu referencira kao na tablicu.*
   - *SELECT prosjecna_placa FROM privremena_tablica → iz privremene tablice dohvaća se stupac prosjecna_placa.*

Tablica 58. Rezultat podupita u FROM dijelu

prosjecna_placa
7900

4. **Korelirani podupit**

   Korelirani podupit odnosi se na stupce iz glavnog upita. Izvodi se za svaki redak glavnog upita. U podupitu možemo referirati podatke iz vanjskog upita i zato se naziva korelirani podupit, jer je povezan sa svojim roditeljskim upitom i ne može se izvršavati samostalno kao drugi podupti.

Primjer:

```
SELECT id, ime, prezime, placa, odjel
FROM zaposlenik AS z1
WHERE placa > (SELECT AVG(placa)
               FROM zaposlenik AS z2
               WHERE z1.odjel = z2.odjel);
```

- *Ovaj upit pronalazi zaposlenike koji imaju plaću veću od prosječne plaće unutar svog odjela.*

Tablica 59. Rezultat koreliranog podupita

id	ime	prezime	placa	odjel
2	Ana	Kovačić	9500	IT
5	Luka	Jurić	8500	Marketing

5. **Podupit s HAVING**

   Kada se podupiti koriste u HAVING dijelu moraju vratiti uvijek jednu vrijednost koja se odabire ili koju vraća agregatna funkcija.

Primjer:

```
SELECT odjel, AVG(placa) AS prosjecna_placa
FROM zaposlenik
GROUP BY odjel
HAVING AVG(placa) > (SELECT AVG(placa) FROM zaposlenik);
```

- *Ovaj upit pronalazi odjele čija prosječna plaća prelazi ukupni prosjek plaća svih zaposlenika.*
- *Unutarnji upit (SELECT AVG(placa) FROM zaposlenik) vraća ukupni prosjek plaća svih zaposlenika (u ovom slučaju 7900).*
- *Vanjski upit grupira podatke prema odjelima i računa prosječnu plaću po odjelu.*
- *Klauzula HAVING osigurava da se prikažu samo oni odjeli čija je prosječna plaća veća od ukupnog prosjeka.*

Tablica 60. Rezultat podupita s HAVING

odjel	prosjecna_placa
IT	8750

Podupiti i pridruživanja (JOIN) u SQL-u koriste se za kombiniranje podataka, ali se razlikuju u načinu primjene. Podupit je upit unutar drugog upita, izvršava se neovisno i njegov rezultat koristi se kao ulaz u glavni upit. Najčešće se primjenjuje kada je potrebno izračunati vrijednost ili filtrirati podatke koji nisu izravno dostupni. S druge strane, pridruživanje (JOIN) kombinira podatke iz više tablica na temelju zajedničkih stupaca, omogućujući paralelnu

analizu podataka u jednom koraku. Podupiti se obično koriste za dodatne ili specifične izračune, dok je JOIN učinkovitiji kada treba povezati logički povezane tablice. JOIN se često preferira nad podupitima kod pisanja naredbi jer baze podataka obično optimiziraju JOIN operacije bolje nego podupite, što može rezultirati bržim izvođenjem. Ovo je zato što JOIN omogućava bazama podataka da koriste indekse i optimizirane algoritme za spajanje tablica.

4.6.1. Operatori podupita (engl. *Subquery operators*)

Operatori podupita se koriste za integraciju rezultata podupita u glavni SQL upit. Ovi operatori omogućuju usporedbu rezultata podupita s vrijednostima u glavnom upitu.

1. **IN / NOT IN**
   Koristi se za provjeru pripadnosti vrijednosti skupu rezultata koji vraća podupit. Provjerava se da li je vrijednost jednaka bilo kojem retku koji vraća podupit. Ukoliko se vraća NULL cijeli izraz je NULL.

   Primjer:

```
SELECT ime, prezime
FROM zaposlenik
WHERE odjel_id IN (SELECT id FROM odjeli WHERE naziv =
'IT');
```

   - *Uspoređuje odjel_id s popisom ID-ova iz podupita.*

2. **EXISTS**
   Koristi se za provjeru postoji li barem jedan redak koji zadovoljava uvjete podupita. Vraća TRUE ili FALSE.

   Primjer:

```
SELECT ime, prezime
FROM zaposlenik
WHERE EXISTS (SELECT 1 FROM odjeli WHERE odjeli.id =
zaposlenik.odjel_id);
```

   - *Provjerava postoji li odgovarajući zapis u tablici odjeli.*

3. **ANY / SOME**
   Koristi se za usporedbu vrijednosti s bilo kojom vrijednošću iz rezultata podupita i ukoliko se bilo koja usporedba podudara vraća se TRUE.

Primjer:

```
SELECT ime, prezime, placa
FROM zaposlenik
WHERE placa > ANY (SELECT placa FROM zaposlenik WHERE
odjel_id = 1);
```

- *Dohvaća zaposlenike čija je plaća veća od bilo koje plaće zaposlenika iz odjela s ID-jem 1.*

4. **ALL**
   Koristi se za usporedbu vrijednosti sa svim vrijednostima iz rezultata podupita. Ukoliko se sve usporedbe podudaraju vraća se TRUE.

   Primjer:

```
SELECT ime, prezime, placa
FROM zaposlenik
WHERE placa > ALL (SELECT placa FROM zaposlenik WHERE
odjel_id = 1);
```

- *Dohvaća zaposlenike čija je plaća veća od svih plaća zaposlenika iz odjela s ID-jem 1.*

5. **Usporedba s jednom vrijednošću (=, <, >, <=, >=)**
   Koriste se za usporedbu vrijednosti iz glavnog upita s jednom vrijednošću iz podupita. Podupit mora vratiti točno jedan rezultat (skalar).

   Primjer:

```
SELECT ime, prezime, placa
FROM zaposlenik
WHERE placa = (SELECT MAX(placa) FROM zaposlenik);
```

- *Pronalazi zaposlenika s najvišom plaćom.*

4.7. Naredba EXPLAIN ANALYZE

EXPLAIN ANALYZE u SQL-u je naredba koja kombinira funkcionalnosti naredbi EXPLAIN i ANALYZE. Koristi se za analizu izvedbe upita i dobivanje detaljnih informacija o tome kako je baza podataka izvršila upit. Kada se koristi EXPLAIN ANALYZE, baza podataka izvršava upit i prikazuje stvarne performanse za svaki korak u izvršnom planu, uključujući stvarno vrijeme izvršavanja i broj redaka koji su obrađeni.

Sintaksa:

```
EXPLAIN ANALYZE <SQL_UPIT>;
```

Izlaz sadrži korake potrebne za izvršenje upita, hijerarhiju operacija, stvarno vrijeme izvršavanja i vrijeme u milisekundama za svaki korak, očekivani i stvarni broj redaka obrađenih na svakoj razini te procijenjene troškove izvršenja koje optimizator koristi za odabir plana.

Primjer:

```
EXPLAIN ANALYZE SELECT * FROM zaposlenik WHERE gender =
'M';
```

Slika 15. Izgled rezultata naredbe iz primjera EXPLAIN ANALYZE u Valentina Studio[5]

	QUERY PLAN
1	Seq Scan on employees (cost=0.00..6054.30 rows=180284 width=31) (actual time=0.018..33.335 rows=179973 loops=1)
2	Filter: (gender = 'M'::gender)
3	Rows Removed by Filter: 120051
4	Planning Time: 0.340 ms
5	Execution Time: 36.731 ms

[5] Valentina Studio je besplatan alat za upravljanje bazama podataka koji omogućuje korisnicima stvaranje, administraciju, upite i istraživanje različitih baza podataka, uključujući MySQL, MariaDB, PostgreSQL, MS SQL Server, Valentina DB i SQLite te je dostupan na https://www.valentina-db.com/en/download-valentina-studio/current

5. UPRAVLJANJE BAZOM PODATAKA

Za upravljanje bazom podataka koriste se DDL (engl. *data definition language*), DML (engl. *data manipulation language*) te DCL (engl. *data control language*).

5.1. Izrada baze podataka

Naredba CREATE DATABASE u SQL-u spada pod DDL (engl. *data definition language*) te se koristi za izradu nove baze podataka unutar sustava za upravljanje bazama podataka (DBMS). Iako je sintaksa naredbe u osnovi slična u različitim sustavima (npr. MySQL, MariaDB, PostgreSQL i dr.), svaki sustav ima svoje specifične opcije i značajke.

Sintaksa:

```
CREATE DATABASE ime_baze
[ WITH ]
[ OWNER = korisnik ]
[ ENCODING = 'enkodiranje' ]
[ LC_COLLATE = 'pravilo_sortiranja' ]
[ LC_CTYPE = 'pravilo_znakova' ]
[ CONNECTION LIMIT = broj_veza ];
```

- *ime_baze: Ime nove baze podataka koje mora biti jedinstveno na serveru.*
- *OWNER: korisnik koji će biti vlasnik baze. Ako nije specificirano, koristi se trenutni korisnik.*
- *ENCODING: Određuje enkodiranje znakova koje će se koristiti u bazi (npr. UTF8, LATIN1).*
- *LC_COLLATE: Pravilo sortiranja teksta prema jezičnim pravilima (npr. abecedni redoslijed).*
- *LC_CTYPE: Pravilo za kategorizaciju znakova (velika i mala slova, klasifikacija znakova).*
- *CONNECTION LIMIT: Maksimalan broj veza prema bazi podataka.*

CREATE DATABASE u MySQL-u/MariaDB-u

U MySQL-u i MariaDB-u, sintaksa je jednostavnija, jer ti sustavi nemaju toliko složenih opcija prilikom izrade baza. Evo kako izgleda osnovna sintaksa:

Sintaksa:

```
CREATE DATABASE ime_baze
[ CHARACTER SET 'znakovni_skup' ]
[ COLLATE 'pravilo_sortiranja' ];
```

- *CHARACTER SET: Omogućuje postavljanje skupa znakova za bazu*
- *COLLATE: Omogućuje postavljanje pravila sortiranja i usporedbe (npr. utf8mb4_general_ci za neosjetljivost na velika/mala slova)*

CREATE DATABASE u PostgreSQL-u

PostgreSQL pruža naprednije opcije za izradu baza, uključujući mogućnost korištenja predložaka i više parametara konfiguracije. Sintaksa:

```
CREATE DATABASE ime_baze
[ WITH ]
[ OWNER = korisnik ]
[ TEMPLATE = predložak_baze ]
[ ENCODING = 'enkodiranje' ]
[ LC_COLLATE = 'pravilo_sortiranja' ]
[ LC_CTYPE = 'pravilo_znakova' ]
[ TABLESPACE = 'tablespace' ]
[ CONNECTION LIMIT = broj_veza ];
```

- *TEMPLATE: Omogućuje korištenje predloška (npr. template1 ili template0).*
- *ENCODING, LC_COLLATE, LC_CTYPE: Veća kontrola nad jezičnim postavkama baze.*
- *TABLESPACE: Mogućnost određivanja fizičkog mjesta pohrane baze.*

5.2. Sheme

Sheme su prisutne u većini SQL sustava, ali se njihova implementacija i značenje mogu razlikovati. U PostgreSQL-u omogućuje se njihova upotreba za organizaciju podataka unutar jedne baze, dok se u MySQL-u umjesto shema koriste odvojene baze.

U **PostgreSQL**-u shema predstavlja logički skup objekata baze podataka, poput tablica, pogleda (views), indeksa i tipova podataka. Može se zamisliti kao "kutija" unutar baze podataka u kojoj se organiziraju podaci. Svaka baza podataka može sadržavati više shema, što omogućuje bolju organizaciju i upravljanje podacima. Sheme se često koriste za grupiranje povezanih podataka unutar iste baze podataka. Na primjer, u poslovnom okruženju mogu se definirati zasebne sheme za različite odjele, poput prodaja, ljudski resursi i tehnologija. Na taj način se osigurava razdvajanje podataka unutar iste baze, ali se i dalje mogu jednostavno dohvatiti i povezivati kroz upite.

Svaka nova baza podataka automatski dobiva zadanu shemu *public*. Ukoliko se pri upitu ne specificira shema, utoliko se podrazumijeva korištenje *public*. Na primjer, naredba:

```
SELECT * FROM zaposlenik;
```

jednaka je naredbi:

```
SELECT * FROM public.zaposlenik;
```

Ukoliko postoji tablica istog imena u drugoj shemi, primjerice *prodaja.zaposlenik*, utoliko će se bez navođenja sheme uvijek prvo pristupati tablici u *public*.

Nova shema može se stvoriti naredbom:

```
CREATE SCHEMA prodaja;
```

Nakon izrade, tablice i ostali objekti mogu se dodavati unutar te sheme koristeći odgovarajuće SQL naredbe. Sheme u PostgreSQL-u omogućuju fleksibilnu organizaciju podataka unutar baze podataka. One olakšavaju upravljanje podacima, omogućuju izolaciju među korisnicima i smanjuju potrebu za stvaranjem više baza podataka. Korištenje shema preporučuje se u svim situacijama gdje je potrebno logički razdvojiti podatke, a istovremeno ih zadržati u istoj bazi radi jednostavnijih upita i održavanja.

5.3. Uloge/korisnici u bazama podataka

Uloge (engl. *roles*) predstavljaju ključan aspekt sigurnosti u sustavima za upravljanje bazama podataka (DBMS). One određuju što korisnici mogu raditi unutar baze podataka te osiguravaju kontrolu pristupa nad resursima. Pravilno postavljanje uloga i pripadajućih dozvola ključno je za sprječavanje neovlaštenog pristupa i održavanje integriteta podataka.

Većina modernih sustava za upravljanje bazama podataka podržava neku varijantu uloga ili korisničkih dozvola. Glavne značajke uključuju određivanje korisnika i grupa i njihovih uloga. Privilegije se dodjeljuju ulogama kako bi definirale što korisnici mogu raditi (čitanje, pisanje, administracija baze). Neke baze omogućuju da jedna uloga nasljeđuje prava od druge, čime se olakšava upravljanje pristupima. Različiti DBMS sustavi implementiraju upravljanje ulogama na različite načine.

PostgreSQL koristi koncept ***uloga*** (engl. *roles*), gdje svaka uloga može biti korisnik ili grupa korisnika. Uloge su fleksibilne i mogu imati različite privilegije te mogu nasljeđivati prava od drugih uloga. MySQL s druge strane koristi ***korisnike*** (engl. *users*), gdje su korisnici zasebni entiteti s dodijeljenim pravima. Od verzije 8.0, MySQL je uveo mehanizam uloga koji omogućuje definiranje skupa dozvola i njihovu dodjelu korisnicima, dok su ranije privilegije morale biti dodjeljivane svakom korisniku pojedinačno.

PostgreSQL ima SUPERUSER atribut, koji omogućuje punu kontrolu nad bazom podataka. Ostale važne uloge su CREATEDB, CREATEROLE, REPLICATION itd. MySQL koristi ***root*** korisnika s punim administrativnim pravima. Ostale dozvole, poput GRANT OPTION omogućuju delegiranje privilegija drugim korisnicima.

5.3.1. CREATE ROLE/ USER

Razlika između CREATE ROLE i CREATE USER u PostgreSQL-u više je stvar semantike, nego funkcionalnosti. PostgreSQL tretira korisnike kao posebnu vrstu uloga.

Uloga po zadanim vrijednostima nema pravo prijave, osim ako se izričito doda atribut LOGIN. Može se koristiti za definiranje grupnih prava koja se dodjeljuju drugim korisnicima ili ulogama. Može se koristiti i kao pojedinačni korisnik, ako ima LOGIN atribut.

Primjer:

```
CREATE ROLE samo_citanje WITH LOGIN ENCRYPTED PASSWORD
'lozinka';
```

- *uloga samo_citanje koja ima pravo prijave i koristi se kao korisnik*
- *ova uloga još uvijek nema nikakve privilegije osim mogućnosti prijave. Kako bi korisnik s ovom ulogom mogao samo čitati podatke iz tablica, potrebno mu je dodijeliti odgovarajuće privilegije.*

Kad izrađujemo korisnike PostgreSQL, automatski dodjeljuje atribut LOGIN. Tehnički, svaki korisnik stvoren s CREATE USER samo je posebna uloga s dodijeljenim atributom LOGIN.

Primjer:

```
CREATE USER samo_citanje ENCRYPTED PASSWORD 'lozinka';
```

Tablica 61. Ključne razlike CREATE ROLE i CREATE USER

Aspekt	CREATE ROLE	CREATE USER
Primjena	Opće uloge (grupne ili korisničke)	Isključivo za pojedinačne korisnike
Pravo prijave	Nije zadana postavka (dodaje se s LOGIN).	Zadana je postavka prava prijave
Preporuka za korištenje	Koristi se za fleksibilnije scenarije, npr. stvaranje grupa koje sadrže prava i koje nasljeđuju druge uloge	Fokusira se na stvaranje korisnika baze

U konačnici, oboje rade slično, jer su korisnici i uloge u PostgreSQL-u dio istog sustava "roles". Razlika je više u svrsi i čitljivosti.

Uloge se mogu dodjeljivati korisnicima preko **GRANT** naredbe. Naredba GRANT omogućuje korisniku ili ulozi da naslijedi prava i privilegije od određene uloge. Korisnik kojem se dodijeli uloga nasljeđuje prava koja su već dodijeljena toj ulozi (npr. pristup tablicama, izvršavanje funkcija itd.). Na ovaj način mogu se grupirati prava i jednostavno ih dodjeljivati korisnicima ili ulogama.

Sintaksa:

```
GRANT <ime_role> TO <korisnik_ili_rola>;
```

5.3.2. Atributi uloga

Prilikom kreiranja uloge u bazama podataka (npr. u PostgreSQL-u, MySQL-u ili SQL Serveru), ulozi se mogu dodijeliti različiti atributi koji definiraju njezina prava i ograničenja, oni definiraju dijelove privilegija uloge. Atributi uloge određuju što uloga može raditi unutar baze podataka. Na ulogu se dodaju nakon ključne riječi **WITH**. Češći atributi koji se koriste kod definiranja uloga su:

1. **LOGIN/ NOLOGIN** – određuje da li se uloga može prijaviti u bazu podataka. Uloge mogu postojati i bez prava prijave, služeći samo za dodjelu privilegija drugim ulogama.
2. **SUPERUSER / NOSUPERUSER** – određuje da li se ulozi dodjeljuju administratorska prava nad cijelom bazom podataka.
3. **CREATEDB/ NOCREATEDB** – mogućnosti kreiranja novih baza podataka.

4. **CREATEROLE / NOCREATEROLE** – mogućnosti kreiranja i upravljanje drugim ulogama.
5. **INHERIT** – određuje nasljeđuje li uloga privilegije drugih uloga kojima pripada.
6. **NOREPLICATION** – zabranjuje ulozi korištenje replikacijskih funkcionalnosti.
7. **CONNECTION LIMIT** – postavlja maksimalan broj istovremenih konekcija koje uloga može imati.
8. **BYPASSRLS** – omogućava zaobilaženje pravila sigurnosti na razini redaka (engl. *Row Level Security - RLS*).

Primjeri:

```
CREATE ROLE admin_uloga WITH
    LOGIN
    SUPERUSER
    CREATEDB
    CREATEROLE
    CONNECTION LIMIT 5;
```

5.3.3. Privilegije

Prilikom definiranja uloge, atributima se mogu odrediti određene privilegije koje uloga ima nad bazom podataka, poput mogućnosti prijave, kreiranja drugih uloga ili baza podataka. Međutim, osim ovih atributa, postoje i dodatne privilegije koje se dodjeljuju izvan okvira atributa uloge. Te privilegije mogu se precizno definirati za pojedine korisnike ili uloge, omogućujući granularnu kontrolu nad pristupom i operacijama unutar baze podataka. Privilegije korisnika spadaju pod DCL (engl. *data control language*) te obuhvaćaju prava na pristup, manipulaciju podacima te administrativne ovlasti koje im omogućuju izvršavanje određenih radnji nad tablicama, pogledima, funkcijama i drugim objektima u sustavu baze podataka.

Privilegije se uvijek dodjeljuju prema načelu najmanjih prava kako bi se osiguralo da korisnici ili uloge imaju samo one ovlasti koje su nužne za izvršavanje njihovih zadataka. Time se smanjuje rizik od zloupotrebe ili nenamjernog narušavanja sigurnosti i integriteta baze podataka.

Prema zadanim postavkama, samo kreator baze podataka ili superkorisnik ima pristup njezinim objektima. Ostalim ulogama se prava moraju dodijeliti. Ono što nije eksplicitno dodijeljeno je zabranjeno. Prava se dodjeljuju pomoću naredbi poput **GRANT** za dodjelu prava i **REVOKE** za njihovo oduzimanje.

Primjer:

```
CREATE ROLE samo_citanje WITH LOGIN ENCRYPTED PASSWORD
'lozinka';
```

```
GRANT CONNECT ON DATABASE moja_baza TO samo_citanje;
```

- *Kreiranoj ulozi omogućuje se prijava u bazu i povezivanje s bazom moja_baza*

```
GRANT USAGE ON SCHEMA public TO samo_citanje;
```

- *omogućava pristup objektima unutar sheme (npr. tablicama, funkcijama), ali ne daje automatska prava na čitanje ili izmjene podataka.*

```
GRANT SELECT ON ALL TABLES IN SCHEMA public TO
samo_citanje;
```

- *pokriva samo postojeće tablice. Ako se u shemi public u budućnosti kreiraju nove tablice, sa navedenom naredbom neće imati pravo čitanja nad tim tablicama*

```
ALTER DEFAULT PRIVILEGES IN SCHEMA public GRANT SELECT ON
TABLES TO samo_citanje;
```

- *naredba osigurava da korisnik samo_citanje automatski dobiva pravo čitanja na svim budućim tablicama stvorenim u shemi public.*

GRANT ALL PRIVILEGES - sva prava uključuju:

1. **SELECT** (čitanje podataka);
2. **INSERT** (umetanje novih redaka);
3. **UPDATE** (izmjena podataka);
4. **DELETE** (brisanje podataka);
5. **TRUNCATE, REFERENCES i TRIGGER** (ako je primjenjivo).

Sintaksa:

```
GRANT ALL PRIVILEGES ON <table> TO <user>;
```

- *Ova naredba daje sva prava na specificiranu tablicu korisniku ili ulozi.*

GRANT ALL ON ALL TABLES –dodjeljuje sva prava na sve tablice unutar specificirane sheme korisniku ili ulozi. Ukoliko se ne navede shema, podrazumijeva se *public* shema.

Sintaksa:

```
GRANT ALL ON ALL TABLES [IN SCHEMA <schema>] TO <user>;
```

GRANT [SELECT, UPDATE, INSERT, ...] - dodjeljuje specifične privilegije (npr. SELECT, INSERT, UPDATE...) na određenu tablicu unutar određene sheme korisniku ili ulozi. Omogućuje precizniju kontrolu nad pravima.

Sintaksa:

```
GRANT [SELECT, UPDATE, INSERT, ...] ON <table> [IN SCHEMA
<schema>] TO <user>;
```

ALTER DEFAULT PRIVILEGES IN SCHEMA <schema> **GRANT** [SELECT, UPDATE, INSERT, ...] **ON TABLES –** dodjeljuje privilegije za nove tablice i ukoliko se želi da korisnik automatski ima prava nad njima

Sintaksa:

```
ALTER DEFAULT PRIVILEGES IN SCHEMA public GRANT [SELECT,
UPDATE, INSERT, ...] ON TABLES TO <user>;
```

6. TIPOVI PODATAKA

Tipovi podataka ključni su element u dizajnu baza podataka jer određuju vrstu vrijednosti koje stupci u tablicama mogu pohraniti. Pravilan odabir tipova podataka značajno utječe na performanse, integritet i optimizaciju baze podataka. Tip podatka je ograničenje postavljeno na polje u tablici kako bi se omogućilo unos samo te vrste podataka.

Kada odredimo tip podatka za stupac, ograničava se vrsta podataka koja se može u njega unijeti, čime se sprječavaju nepravilni unosi. Odabir optimalnog tipa podataka smanjuje potrošnju memorije i prostora na disku, čime se poboljšava rad baze i moguće je povećanje brzine upita. Loš odabir tipova podataka može dovesti do:

- prekomjerne potrošnje memorije (npr. korištenje VARCHAR(255) umjesto VARCHAR(50) bez potrebe);
- gubitka preciznosti (npr. korištenje FLOAT umjesto DECIMAL za financijske podatke);
- problema s integritetom podataka (npr. pohrana datuma kao VARCHAR može dovesti do problema pri filtriranju i sortiranju);
- sporijih upita zbog preopterećenja indeksa i nepotrebne konverzije podataka.

Pravilnim izborom tipova podataka osigurava se optimalan rad baze podataka, smanjuju se troškovi pohrane i povećava preciznost analiza.

Tipove podataka možemo podijeliti u sljedeće kategorije za baze podataka:

1. **Numerički tipovi (engl. *Numeric Types*)**
   Tipovi podataka za pohranu numeričkih vrijednosti.
2. **Znakovni tipovi (engl. *Character Types*)**
   Tipovi podataka za pohranu tekstualnih vrijednosti.
3. **Tipovi datuma/vremena (engl. *Date/Time Types*)**
   Tipovi podataka za pohranu datuma, vremena ili kombinacije datuma i vremena.
4. **Logički tipovi (engl. *Boolean Types*)**
   Tip podataka za pohranu logičkih vrijednosti.
5. **Jedinstveni identifikator (engl. *UUID Type*)**
   Koristi se za jedinstvenu identifikaciju zapisa u sustavima gdje se traži globalna jedinstvenost (npr. distribuirane baze podataka).
6. **Binarni tipovi (engl. *Binary Types*)**
   Tipovi podataka za pohranu binarnih objekata poput slika, audio/video datoteka ili drugih binarnih podataka.
7. **Polja (engl. *Arrays*)**
   Tipovi podataka koji omogućuju pohranu više vrijednosti istog tipa u jednom polju.
8. **JSON tipovi (engl. JSON Types)**
   Tip podatka za pohranu strukturiranih podataka u formatu *JavaScript Object Notation* (JSON).

Ove kategorije ne obuhvaćaju sve dostupne tipove podataka, jer postoje specifični tipovi koji se manje koriste ili su svojstveni pojedinim bazama podataka. Na primjer, PostgreSQL podržava dodatne tipove podataka poput **Geometrijskih tipova** (engl. *Geometric Types*) za pohranu prostornih podataka, **Opsega** (engl. *Range Types*) koji omogućuju definiranje intervala numeričkih vrijednosti ili datuma, **Citext** tipa koji omogućuje pohranu tekstualnih vrijednosti neosjetljivih na velika/mala slova te **TSVector** tipa koji je optimiziran za pretragu teksta. [6]

Tablica 62. Pregled učestalijih tipova podataka u PostgreSQL i MySQL bazama podataka

Tip podataka	PostgreSQL	MySQL	Veličina zapisa
Numerički tipovi			
Cijeli brojevi	SMALLINT	SMALLINT	2 bajta
	INTEGER	INT	4 bajta
	BIGINT	BIGINT	8 bajta
Decimalni brojevi	NUMERIC (1-131.072, 0-16.383)	NUMERIC(1-65, 0-30)	NUMERIC (M, D), DECIMAL (M, D) M- (*Maximum Digits*) – ukupan broj znamenki koje broj može imati (prije i poslije decimalne točke); D-(*Decimal Places*) – broj znamenki koje mogu biti iza decimalne točke.
	DECIMAL (1-131.072, 0-16.383)	DECIMAL(1-65, 0-30)	
	REAL	FLOAT	4 bajta, približno 6 znamenki preciznosti
	DOUBLE PRECISION	DOUBLE	8 bajta, približno 15 znamenki preciznosti
Serijski brojevi	SERIAL	INT AUTO_INCREMENT	4 bajta
	BIGSERIAL	BIGINT AUTO_INCREMENT	8 bajta
Znakovni tipovi			
Fiksna duljina	CHAR(1-255)	CHAR(1-255)	1 do 255 znakova
Varijabilna duljina	VARCHAR(1-10.485.760)	VARCHAR(1-65.535)	n znakova +1-2 bajtova za dužinu
Neograničena duljina	TEXT	TEXT	Maksimalno 1 GB u PostgreSQL-u, 64 KB do 4 GB u MySQL-u
Tipovi datuma/vremena			
Datum	DATE	DATE	PostgreSQL - 4 bajta - raspon 4713. pr. Kr. – 5874897.

[6] Detaljan popis tipova podataka u PostgreSQL bazi može se pronaći na poveznici: https://www.postgresql.org/docs/current/datatype.html

			MySQL - 3 bajta - s mogućnošću spremanja od 1000. – 9999.
Vrijeme	TIME(0-6)	TIME(0-6)	3-5 bajta, preciznost sekundi od 0 do 6 decimala
Datum i vrijeme	TIMESTAMP(0-6)	TIMESTAMP(0-6)	MySQL - Unix vrijeme – 4 bajta (za TIMESTAMP(0)) do 7 bajta (za TIMESTAMP(6)) Raspon spremanja: 1970-01-01 00:00:01 do 2038-01-19 03:14:07 PostgreSQL - 8 bajta TIMESTAMP WITH TIME ZONE: 4713. pr. Kr. do 294.276. n. e. TIMESTAMP WITHOUT TIME ZONE: 4713. pr. Kr. do 587.4897. n. e.
	Nema ekvivalent	DATETIME(0-6)	MySQL - 8-11 bajta - 1000-01-01 do 9999-12-31
Interval	INTERVAL	Nema ekvivalent	12 bajtova, fleksibilan format
Godina	Nema ekvivalent	YEAR	1 bajt - s dopuštenim vrijednostima za godine od 1901. – 2155.
Logički tipovi			
Logički	BOOLEAN	BOOLEAN (alias za TINYINT(1))	1 bajt
Binarni tipovi			
Manji binarni objekti	BYTEA	TINYBLOB/ BLOB/ MEDIUMBLOB/ LONGBLOB	Maksimalno 1 GB u PostgreSQL-u, 255 B do 4 GB u MySQL-u
Veliki objekti	Large Object (LOB)	LONGBLOB	LOB: 4TB, LONGBLOB: 4 GB
Jedinstveni identifikator			
UUID	UUID	CHAR(36) ili BINARY(16)	16 bajtova
Polja			
Polja	ARRAY	Nema ekvivalent; koristi se JSON	Ograničenje ovisi o ukupnoj veličini retka
JSON tipovi			
JSON tipovi	JSON	JSON	Maksimalno 1 GB u PostgreSQL-u, 64 KB u MySQL-u
	JSONB	JSON (implicitno kao JSONB)	Maksimalno 1 GB u PostgreSQL-u

6.1. Numerički tipovi podataka

6.1.1. Cijeli brojevi

Cijeli brojevi (engl. *integer*) brojčani su tipovi podataka bez decimalnog dijela, koriste se za spremanje diskretnih vrijednosti poput ID-eva, količina i brojača. U MySQL-u podržani su različiti tipovi cijelih brojeva, a svaki zauzima određenu količinu memorije i ima svoj raspon.

Tablica 63. Tipovi cijelih brojeva u MySQL bazi i veličina pohrane

Tip podatka	Veličina (bajtovi)	Minimalna vrijednost (SIGNED)	Maksimalna vrijednost (SIGNED)	Maksimalna vrijednost (UNSIGNED)
TINYINT	1 bajt	-128	127	0 – 255
SMALLINT	2 bajta	-32.768	32.767	0 – 65.535
MEDIUMINT	3 bajta	-8.388.608	8.388.607	0 – 16.777.215
INT (INTEGER)	4 bajta	-2.147.483.648	2.147.483.647	0 – 4.294.967.295
BIGINT	8 bajta	-9.223.372.036.854.775.808	9.223.372.036.854.775.807	0 – 18.446.744.073.709.551.615

SIGNED (zadana opcija) dopušta pozitivne i negativne vrijednosti. UNSIGNED koristi samo pozitivne vrijednosti, ali dopušta dvostruko veći raspon. PostgreSQL podržava samo SMALLINT (2bajta), INTEGER (4 bajta) i BIGINT (8 bajta).

Svaki cijeli broj koristi uvijek isti broj bajtova bez obzira na stvarnu vrijednost broja. Time se brže izvršavaju upiti jer je podatke lako indeksirati i pretraživati. Operacije na cijelim brojevima su brže jer ne zahtijevaju dodatne konverzije (kao što je slučaj s FLOAT ili DECIMAL). Za uštedu prostora uvijek treba birati najmanji odgovarajući cjelobrojni tip.

6.1.2. Decimalni tipovi podataka

Tipovi podataka s fiksnom preciznošću DECIMAL (M,D), NUMERIC (M,D)

U bazama podataka, tip podataka DECIMAL(M, D) definira numeričke vrijednosti s fiksnom preciznošću i skalom, gdje M predstavlja ukupan broj znamenki, a D broj decimalnih mjesta. DECIMAL i NUMERIC su isti tip podataka – oba su fiksne preciznosti i koriste se za pohranu točnih decimalnih brojeva, primjerice u financijama. PostgreSQL dopušta veći raspon preciznosti (131072 znamenke), dok je MySQL ograničen na 65 znamenki.

Primjer:

```
DECIMAL(10,4)
```

- *Definira se tip podatka koji može imati ukupno 10 znamenki, od čega su 4 znamenke nakon decimalne točke, a preostalih 6 znamenki su prije decimalne točke.*

Kada se pokuša unijeti vrijednost koja premašuje definirane granice ponašanje sustava ovisi o bazi podataka. U PostgreSQL-u, unos koji premašuje dopuštenu preciznost ili broj decimalnih mjesta automatski se odbacuje uz grešku. Na primjer, ako je stupac definiran kao DECIMAL(10,4), a pokušava se unijeti broj s više od 10 znamenki ukupno ili više od 4 decimalna mjesta, PostgreSQL generirat će *„numeric field overflow error"* i neće spremiti vrijednost. Ovaj pristup osigurava strogu provjeru podataka, čime se sprječava gubitak preciznosti ili pojava nepredviđenih vrijednosti.

U MySQL-u, ponašanje ovisi o postavljenom SQL modu. Ako je STRICT MODE uključen, MySQL se ponaša slično PostgreSQL-u i odbija unos koji premašuje definirane granice. Međutim, ako STRICT MODE nije aktivan, MySQL neće odbiti unos, već će automatski zaokružiti broj na dopušteni broj decimalnih mjesta bez generiranja greške, već samo s upozorenjem. Primjerice, unos vrijednosti 12345.6789123 u stupac DECIMAL(10,4) rezultirat će pohranjenom vrijednošću 12345.6789. Kad bi unijeli 12345678.1234, MySQL će se pokušati prilagoditi broju dopuštenog raspona (-999999.9999 do 999999.9999) te će upisati 999999.9999 uz upozorenje, ali će ipak spremiti skraćenu vrijednost.

Ova razlika u ponašanju između PostgreSQL-a i MySQL-a ima značajan utjecaj na integritet podataka. PostgreSQL-ov pristup osigurava striktnu točnost podataka i eliminira rizik od nenamjernog zaokruživanja, što je ključno u financijskim i znanstvenim aplikacijama. S druge strane, MySQL omogućuje veću fleksibilnost, ali u okruženjima gdje je točnost presudna preporučuje se korištenje STRICT MODE kako bi se izbjegli potencijalni problemi s nenamjernim promjenama vrijednosti.

PostgreSQL pohranjuje podatke za DECIMAL/NUMERIC dinamički, dok MySQL koristi fiksni format, što znači da u MySQL-u uvijek zauzimaju istu količinu memorije bez obzira na stvarne vrijednosti. Kada se koristi fiksna veličina brže se pristupa podacima jer se lakše pretražuje memorija a nedostatak je da se troši memorija čak i ako broj ima manje znamenki od onog što je zadano u definiciji.

Tipovi podataka s pokretnim zarezom – REAL - FLOAT, DOUBLE PRECISION - DOUBLE

Tipovi podataka REAL (ili FLOAT4 - PostgreSQL) / FLOAT (MySQL) i DOUBLE PRECISION (ili FLOAT8 - PostgreSQL)/ DOUBLE (MySQL) koriste se za spremanje decimalnih brojeva s pokretnim zarezom. Za razliku od DECIMAL/NUMERIC, koji pohranjuju brojeve s točno određenom preciznošću, ovi tipovi koriste binarnu reprezentaciju s određenim brojem bajtova, što može dovesti do gubitka preciznosti.

Brojevi s pokretnim zarezom pohranjuju se prema IEEE 754 standardu, koji definira:

- Znak (1 bit) – označava je li broj pozitivan ili negativan.
- Eksponent (8 bitova za REAL/FLOAT, 11 bitova za DOUBLE PRECISION/DOUBLE) – definira pomak decimalne točke.

- Mantisa (23 bita za REAL/FLOAT, 52 bita za DOUBLE PRECISION/DOUBLE) – sadrži vrijednost broja u binarnom obliku.

Ovaj format omogućava rad s vrlo velikim i vrlo malim brojevima, ali dolazi uz nedostatak preciznosti, jer ne može točno pohraniti sve decimalne vrijednosti.

Ovi tipovi podataka su fiksnih veličina, što znači da:

- REAL/FLOAT uvijek zauzimaju 4 bajta bez obzira na vrijednost.
- DOUBLE PRECISION/DOUBLE uvijek zauzima 8 bajta.

Primjeri:

```
CREATE TABLE mjerenja (
temperatura REAL,   -- 4 bajta, preciznost ~6-7 znamenki
tlak DOUBLE PRECISION -- 8 bajta, preciznost ~15-16
znamenki
);
```

- *zadavanje u PostgreSQL*

Tipovi podataka s pokretnim zarezom mogu izgubiti preciznost, jer mogu spremiti samo ograničen broj bitova mantise. Zbog binarnog prikaza, neki brojevi se ne mogu savršeno pohraniti, što uzrokuje male greške. Tipovi podataka s pokretnim zarezom dobri su za znanstvene podatke, ali ne za financije. Kada se zahtijevaju točni brojevi i 100% preciznost (npr. kod novca), koristi se DECIMAL.

U ranijim verzijama MySQL-a mogla se koristiti opcija FLOAT(M, D) i DOUBLE(M, D). Ova sintaksa omogućuje definiranje ukupnog broja znamenki (M) i broja decimalnih mjesta (D), primjerice, FLOAT(5,2) dopušta vrijednosti od -999.99 do 999.99. U novijim verzijama MySQL-a (od verzije 8.0.17.), podrška za ovakve definicije je ukinuta, dok se u MariaDB-u i dalje koristi.

Ako uneseni broj premašuje ovaj raspon, u STRICT MODE-u unos se odbacuje, dok se bez njega broj zaokružuje na najveću dopuštenu vrijednost. Pri unosu brojeva s više decimalnih mjesta nego što je definirano, MariaDB ih automatski zaokružuje. Tako će unos 0.02345 u FLOAT(5,2) rezultirati vrijednošću 0.02. Zbog mogućeg gubitka preciznosti, za financijske izračune preporučuje se korištenje DECIMAL(M, D), umjesto FLOAT(M, D).

6.1.3. Serijski brojevi u bazama podataka (SERIAL, AUTO_INCREMENT)

Serijski brojevi (autoinkrementirajući brojevi) koriste se za jedinstvenu identifikaciju redaka u tablicama, obično kao primarni ključevi. Serijski brojevi su cjelobrojni (integer) tipovi koji se automatski povećavaju prilikom umetanja novog retka.

INT AUTO_INCREMENT / SERIAL koriste 4 bajta i podržava brojeve do 2.147.483.647 dok BIGINT AUTO_INCREMENT / BIGSERIAL koristi 8 bajta i podržava brojeve do 9,2 kvintilijuna (prikladno za jako velike baze).

*Primjer **MySQL**:*

```
CREATE TABLE korisnici (
    id INT AUTO_INCREMENT PRIMARY KEY,
    ime VARCHAR(100) NOT NULL
);
```

- *AUTO_INCREMENT radi samo s cijelim brojevima (INT ili BIGINT)*
- *Početna vrijednost može se postaviti pomoću AUTO_INCREMENT = X*
- *Ako se izbriše redak, ID se ne ponavlja automatski.*

```
ALTER TABLE korisnici AUTO_INCREMENT = 1;
```

- *Resetiranje auto_incerement*

*Primjer **PostgreSQL**:*

```
CREATE TABLE korisnici (
    id SERIAL PRIMARY KEY,
    ime VARCHAR(100) NOT NULL
);
```

- *Automatski se kreira sekvenca (korisnici_id_seq) koja povećava vrijednost id za svaki novi unos*

```
ALTER SEQUENCE korisnici_id_seq RESTART WITH 1;
```

- *Resetiranje SERIAL*

6.2. Znakovni tipovi podataka

Znakovni tipovi podataka u bazama podataka koriste se za pohranu teksta. Razlikuju se prema maksimalnoj duljini, načinu pohrane i učinkovitosti pretrage.

Tablica 64. Vrste znakovnih tipova podataka u bazama podataka

Kategorija	PostgreSQL	MySQL / MariaDB	Veličina pohrane	Korištenje
Fiksna duljina	CHAR(n)	CHAR(n)	1 do 255 znakova	Fiksna duljina, uvijek zauzima n znakova, a broj bajtova ovisi o kodiranju znakova
Varijabilna duljina	VARCHAR(n)	VARCHAR(n)	n znakova + 1-2 bajtova	Varijabilna duljina, zauzima onoliko bajtova koliko je potrebno + dodatni bajt za duljinu.
Neograničena duljina	TEXT	TEXT	do 1 GB - (PostgreSQL), do 4 GB - (MySQL)	Koristi se za velike tekstualne podatke, ali ima ograničenja kod indeksiranja i performansi.

CHAR tipovi u bazama imaju fiksnu duljinu (1 do 255 znakova) te se koriste za kratke, fiksne podatke (npr. oznake država HR, USA, spol M/F). Svaki zapis u memoriji zauzima isti zadani broj bajtova bez obzira na broj spremljenih znakova što daje prednost u bržoj obradi u indeksima ali troši više prostora nego je potrebno ukoliko se često sprema kraći tekst. Ukoliko je u polje tipa CHAR unesen manji broj znakova od definiranog kapaciteta, ono se automatski nadopunjuje prazninama (razmacima) do definirane duljine.

Primjer:

```
CREATE TABLE korisnici (
    oznaka CHAR(3)  -- Sprema točno 3 znaka
);
```

VARCHAR se sprema kao varijabilna duljina (ograničen je maksimalan broj znakova na 65.535 u MySQL-u i do 10.485.760 u PostgreSQL). Koristi se kada duljina teksta varira (npr. ime, adresa, e-mail). Pohranjuje samo stvarni broj znakova + dodatni bajt koji definira duljinu zapisa (ili 2 bajta ako duljina prelazi 255). Štedi prostor u usporedbi s CHAR-om. Sporija je obrada od CHAR-a jer se mora čitati duljina zapisa prije obrade podataka.

Primjer:

```
CREATE TABLE korisnici (
    email VARCHAR(255)
);
```

- *Maksimalno 255 znakova, ali pohranjuje samo stvarnu duljinu koja je unesena u polje*

TEXT omogućuje spremanje podataka s velikim brojem znakova (maksimalno 1 GB u PostgreSQL-u, 64 KB do 4 GB u MySQL-u). Koristi se za vrlo velike tekstove (npr. opisi, blogovi, dokumenti) i ne sprema se u glavnu tablicu (osim u PostgreSQL-u) nego u zaseban segment memorije. U glavnoj tablici se sprema samo pokazivač (pointer) na podatke. Zato pretraga po TEXT poljima može biti sporija nego po VARCHAR poljima. Ne može se indeksirati u potpunosti jer se indeksira samo početni dio (do 1.000 znakova) (osim FULLTEXT indeksa u MySQL-u).

Primjer:

```
CREATE TABLE blog (
    naslov VARCHAR(255),
    sadrzaj TEXT  -- Sprema velike tekstualne podatke
);
```

VARCHAR je bolji za kraće tekstove koji se često pretražuju dok je TEXT bolji za duge tekstove, ali može biti sporiji pri pretragama. VARCHAR je fleksibilan te se koristi za većinu tekstualnih podataka. CHAR je fiksne duljine, koristi se za kratke kodove i brže pretrage.

MYSQL osim navedenih znakovnih tipova podržava **TINYTEXT, MEDIUMTEXT, LONGTEXT**. Glavna razlika između TINYTEXT, TEXT, MEDIUMTEXT i LONGTEXT je maksimalna veličina podataka koje mogu pohraniti.

Tablica 65. Pregled tekstualnih tipova podataka u MySQL-u i veličina pohrane

Tip podatka	Maksimalan broj znakova	Maksimalna veličina	Veličina pokazivača
TINYTEXT	255 znakova	255 bajtova	1 bajt
TEXT	65.535 znakova	64 KB (65.535 bajtova)	2 bajta
MEDIUMTEXT	16.777.215 znakova	16 MB (16.777.215 bajtova)	3 bajta
LONGTEXT	4.294.967.295 znakova	4 GB (4.294.967.295 bajtova)	4 bajta

Broj znakova koji se mogu pohraniti u tekstualne tipove podataka ovise o kodiranju. Primjerice ukoliko se koristi utf8mb4 (4 bajta po znaku), TEXT može pohraniti samo do

16.383 znakova (64 KB / 4) dok u latin1 (1 bajt po znaku), TEXT može pohraniti 65.535 znakova.

6.3. Rad s datumima i vremenskim zonama

Kada se govori o bilježenju vremena ima više metoda i standarda koji se mogu upotrijebiti. Danas najčešće korišteni način bilježenja vremena u bazama podataka je **UTC (*Coordinated Universal Time*)**. UTC je univerzalno standardizirano vrijeme koje se koristi kao osnovna referenca za računanje vremena diljem svijeta. Ne ovisi o vremenskim zonama ili ljetnom/zimskom računanju vremena. UTC je moderni standard baziran na atomskom vremenu, iznimno precizan i konstantan.

GMT (*Greenwich Mean Time*) je sustav je osmišljen prije UTC sustava. GMT je vremenska zona koja se koristi kao referenca i podudara s UTC-om u zimskom razdoblju. GMT je često sinonim za UTC, ali povijesno manje precizan. GMT se ne prilagođava prijestupnim sekundama, dok UTC koristi prijestupne sekunde za usklađivanje s astronomskim vremenom. GMT se ne mijenja za ljetno računanje vremena već prelazi na drugi način računanja vremena. Primjerice, u Europi imamo (CET – *Central European Time*) koji zimi iznosi CET = GMT+1. Ljeti prelazi na CEST (*Central European Summer Time*) = GMT+2.

Iako su često međusobno zamjenjivi, preporučuje se korištenje UTC u bazama podataka jer je precizniji i standardiziran.

Lokalno vrijeme predstavlja vrijeme prilagođeno vremenskoj zoni korisnika. Potrebno je za prikaz podataka korisnicima u njihovoj vremenskoj zoni. Rad s lokalnim vremenom može biti kompliciran zbog promjena na ljetno/zimsko vrijeme. **Globalno vrijeme** obično označava vrijeme spremljeno u UTC formatu. Spremanje podataka u UTC omogućuje standardizaciju vremena na globalnoj razini i jednostavno prilagođavanje za različite vremenske zone. Spremanje u UTC omogućuje jedinstvenu referencu, neovisnu o vremenskim zonama.

Za pohranjivanje vremena u baze podataka često se koristi **ISO 8601 standard**. ISO 8601 je međunarodni standard za prikaz datuma i vremena. Standard definira format kao:

- Datum: YYYY-MM-DD (npr. 2024-12-14);
- Vrijeme: hh:mm:ss (npr. 15:45:30);
- Kombinacija: YYYY-MM-DD**T**hh:mm:ss**Z** (npr. 2024-12-14T15:45:30Z za UTC).

Koriste se dodatne oznake za vremenske zone, npr. +01:00 ili Z za UTC. Vrijednost 2024-12-14T15:45:30Z označava vrijeme u UTC vremenu. Kada se ovo konvertira u vremensku zonu Europe/Zagreb, koja je zimi GMT+1, vrijeme postaje: 2024-12-14T16:45:30**+01:00**. Europe/Zagreb zimi je u vremenskoj zoni UTC+1 (srednjoeuropsko vrijeme, CET). Ljeti bi to bila vremenska zona UTC+2 (srednjoeuropsko ljetno vrijeme, CEST).

ISO 8601 standard nije univerzalan standard za sve baze, ali većina modernih baza podataka (npr. PostgreSQL, MongoDB, SQL Server) ga podržava. Neke baze, poput MySQL-a i SQLite-a, koriste vlastite standarde za prikaz, ali mogu prihvatiti ISO 8601 za unos.

Preporuka je koristiti ISO 8601 za interoperabilnost među sustavima i konzistentnost. Ako je to potrebno, baze često nude funkcije za konverziju.

6.3.1. Tipovi podataka za spremanje datuma/vremena

Baze podataka podržavaju različite tipove podataka za rad s datumima i vremenom. Ključne razlike su u rasponu vrijednosti, preciznosti, memorijskoj veličini i funkcionalnosti.

1. **DATE**
   DATE omogućuje spremanje samo datuma (bez vremena). U PostgreSQL DATE se sprema u 4 bajta i omogućuje godine od 4713. pr. Kr. – 5.874.897 a u MySQL sa 3 bajta s mogućnošću spremanja od 1000. – 9999. Format pohranjivanja je 'YYYY-MM-DD'. Podržane su operacije zbrajanja dana i računanja razlike između datuma.
2. **TIME**
   TIME (0-6) omogućuje spremanje samo vremena (bez datuma). Tip podatka TIME koristi se za pohranu vremenskih vrijednosti u formatu HH:MM:SS, ali također može uključivati milisekunde i mikrosekunde ako je definiran s preciznošću. TIME(3) bi predstavljao zapis HH:MM:SS.ddd tj. 3 mikrosekunde.
3. **TIMESTAMP**
   TIMESTAMP je podatkovni tip koji pohranjuje datum, vrijeme i vremensku zonu. Postoje dvije glavne vrste u PostgreSQL bazi podataka:
   - TIMESTAMP WITHOUT TIME ZONE - pohranjuje vrijeme bez vremenske zone. Pogodno za lokalne aplikacije gdje vremenska zona nije potrebna.
   - TIMESTAMP WITH TIME ZONE - pohranjuje vrijeme u UTC formatu i omogućuje konverziju u lokalno vrijeme. Preporučuje se za globalne aplikacije.

   Primjer:

```
CREATE TABLE vremenskeZone (
     ts TIMESTAMP WITHOUT TIME ZONE,
     tz TIMESTAMP WITH TIME ZONE
);
```

   - *Kreiranje tablice vremenskeZone s 2 stupca ts bez vremenske zone i tz sa vremenskom zonom*

```
INSERT INTO vremenskeZone VALUES (
TIMESTAMP WITHOUT TIME ZONE '2024-12-15 10:00:00+1',
TIMESTAMP WITH TIME ZONE '2024-12-15 10:00:00+1'
);
```

   - *Provođenjem insert naredbe u ts bi dobili zapis '2024-12-15 10:00:00' a u tz '2024-12-15 09:00:00+00'. U ts izbrisana je informacija o vremenskoj zoni a u tz je vrijeme pretvoreno u UTC zapis*

Kako bi zadali spremanje podataka u UTC formatu u bazi podataka bi zadali stupac pri kreaciji tablice s atributima kao u primjeru:

```
vrijeme_kreiranja TIMESTAMP DEFAULT CURRENT_TIMESTAMP AT TIME ZONE 'UTC'
```

a kada želimo lokalno vrijeme zadali bi:

```
lokalno_vrijeme TIMESTAMP DEFAULT CURRENT_TIMESTAMP
```

MySQL TIMESTAMP nema ugrađenu podršku za vremenske zone te ne podržava TIMESTAMP WITH/ WITHOUT TIME ZONE kao što to radi PostgreSQL. Vrijednosti se uvijek spremaju u UTC i konvertiraju pri dohvaćanju, ali ne čuva vremensku zonu unutar samog zapisa.

Kako bi dohvatili lokalno vrijeme iz UTC zapisa potrebno je napraviti konverziju:

*Primjer (**PostgreSQL**):*

```
SELECT vrijeme_kreiranja AT TIME ZONE 'UTC' AT TIME ZONE 'Europe/Zagreb' AS lokalno_vrijeme FROM narudzbe;
```

```
SELECT timezone('Europe/Zagreb', vrijeme_kreiranja AT TIME ZONE 'UTC') AS lokalno_vrijeme FROM narudzbe;
```

- *Pretvorba u lokalno vrijeme*

```
SELECT timezone('Europe/Zagreb', vrijeme_kreiranja) AS lokalno_vrijeme FROM narudzbe;
```

- *PostgreSQL automatski prepoznaje da je vrijeme_kreiranja spremljen u UTC i konvertira ga u traženu vremensku zonu.*

*Primjer (**MySQL**):*

```
SELECT CONVERT_TZ(vrijeme_kreiranja, 'UTC', 'Europe/Zagreb') AS lokalno_vrijeme FROM narudzbe;
```

PostgreSQL baza zadano sprema sva vremena u UTC formatu. Vrijednosti koje vraća ovise o sesiji i korisniku koji pristupa bazi. Postavljanje vremenske zone za sesiju (PostgreSQL):

```
SET TIME ZONE 'Europe/Zagreb';
SET TIME ZONE 'UTC';
```

Ukoliko želimo promijeniti kako baza komunicira s određenim korisnikom napisali bi SQL:

```
ALTER USER postgres SET timezone = 'UTC';
```

Dohvat podataka unutar određenog vremenskog raspona:

```
WHERE vrijeme_kreiranja BETWEEN '2024-01-01 00:00:00'::TIMESTAMP AT TIME ZONE 'UTC' AND '2024-12-31 23:59:59'::TIMESTAMP AT TIME ZONE 'UTC';
```

TIMESTAMP(0-6) može također zadati preciznost i u PostgreSQL zauzima 8 bajta te omogućuje unos od 4713. pr. Kr. – 294.276. a u MySQL 4 bajta s rasponom od 1970. – 2038. TIMESTAMP u MySQL-u ima ograničen raspon vezan uz Unix vrijeme dok u PostgresSQL-u ne ovisi o Unix vremenu. TIMESTAMP može automatski koristiti trenutni vremenski žig (CURRENT_TIMESTAMP).

Primjer:

```
CREATE TABLE logovi (
    id INT PRIMARY KEY AUTO_INCREMENT,
    vrijeme TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

4. **DATETIME**
   DATETIME predstavlja zapis datuma i vremena (bez vremenske zone) te vrlo sličan TIMESTAMP zapisu. Glavna razlika između DATETIME i TIMESTAMP je da TIMESTAMP prati vremensku zonu, dok DATETIME pohranjuje vrijednost fiksno.
5. **INTERVAL**
   PostgreSQL podržava INTERVAL tip podatka, dok MySQL nema ekvivalentan tip. INTERVAL u PostgreSQL-u zauzima 12 bajta i koristi se za spremanje vremenskih perioda.
6. **YEAR**
   YEAR pohranjuje samo godine te se koristi u MySQL bazi. Veličina spremanja je 1 bajt te je moguće spremanje godina od 1901.–2155.

6.3.2. DATE/TIMESTAMP funkcije

Kako bi dobili trenutačno vrijeme možemo koristiti **now()** funkciju. Navedena funkcija vraća trenutno vrijeme kao timestamp ili se pretvara **DATE** zapis kao u primjeru. Datum se može formatirati sa **to_char()** funkcijom.

Primjer:

```
SELECT now()::date;
SELECT CURRENT_DATE;
```

- *Dobivanje trenutačnog datuma*

```
SELECT TO_CHAR(CURRENT_DATE, 'dd.mm.yyyy.') AS datum;
```

- *Formatiranje datuma*

Opis uzoraka za formatiranje datuma i vremena mogu se pogledati na službenim stranicama baze podataka.[7]

[7] Formatiranje datuma i vremena za PostgreSQL može se vidjeti na https://www.postgresql.org/docs/current/functions-formatting.html

6.3.3. Razmak između datuma, računanje godina, izvlačenje podataka

Razmak između datuma može se dobiti jednostavnim oduzimanjem 2 datuma, čime se dobiva razlika u danima.

Primjer:

```
SELECT now() - '2023-5-15';
```

- *Dobivanje razmaka u danima*

Godine se mogu preračunati korištenjem **AGE()** funkcije.

Primjer:

```
SELECT AGE(date '2008-5-15');
SELECT AGE('2008-5-15'::date);
```

- *Dobivanje godina, mjeseci i dana*

```
SELECT AGE('2024/5/15'::date, '2008-5-15'::date);
```

- *Dobivanje godina, mjeseci i dana za razdoblje između zadanih datuma*

Pomoću funkcije **EXTRACT()** moguće je izvući dijelove datuma.

Primjer:

```
SELECT EXTRACT(DAY FROM '2008-5-15'::date);
SELECT EXTRACT(MONTH FROM '2008-5-15'::date);
SELECT EXTRACT(HOUR FROM '2008-5-15 15:03:17'::timestamp);
```

- *Dobivanje dijelova datuma i vremena*

Pomoću funkcije **DATE_TRUNC()** moguće je zaokruživati datume.

Primjer:

```
SELECT date_trunc('month', '2008-5-15'::DATE); --2008-05-
01
SELECT date_trunc('year', '2008-5-15'::DATE); --2008-01-01
SELECT date_trunc('day', '2008-5-15 10:11:12'::TIMESTAMP);
--2008-01-01 00:00:00
```

- *Zaokruživanje vrijednosti*

6.3.4. Interval

Interval omogućuje dohvaćanje podataka u zadanom intervalu. Kao ključne riječi mogu se koristiti *Years, Months, Days, Hours i Minutes*.

Primjer:

```
SELECT * FROM orders WHERE
purchaseSate <=now()-INTERVAL '30 days';
```

6.4. Logički tipovi podataka

Logički tipovi podataka koriste se za pohranu *boolean* vrijednosti koje mogu biti istinite (TRUE) ili neistinite (FALSE). U nekim jezicima i sustavima mogu imati i treću vrijednost – NULL, što označava nepoznatu ili nedostupnu vrijednost. U različitim bazama podataka logički tipovi mogu biti implementirani na različite načine. Iako logičke vrijednosti koriste samo jedan bit za pohranu (0 ili 1), većina baza podataka ih sprema u najmanju moguću jedinicu pohrane, što je obično 1 bajt.

U MySQL-u **BOOLEAN** i **BOOL** su samo aliasi za **TINYINT(1)**, što znači da se pohranjuju kao cijeli broj (0 ili 1). Vrijednost TRUE se interno sprema kao 1, a FALSE kao 0.

BIT(1) se može koristiti kao alternativa BOOLEAN jer pohranjuje jedan bit podataka (0 ili 1). Međutim, BIT(1) se ponaša kao binarni tip, što znači da ga neki SQL upiti ne interpretiraju automatski kao TRUE ili FALSE.

MySQL interno koristi TINYINT(1) za BOOLEAN, ali dopušta pohranu bilo koje vrijednosti od -128 do 127, iako se u logičkom kontekstu koristi samo 0 i 1.

Primjer:

```
CREATE TABLE korisnici (
    id INT AUTO_INCREMENT PRIMARY KEY,
    aktivan BOOLEAN NOT NULL DEFAULT TRUE
);
```

U PostgreSQL-u postoji pravi BOOLEAN tip podataka koji pohranjuje samo vrijednosti TRUE, FALSE i NULL. Interno koristi 1 bajt. Podržava različite reprezentacije ulaznih vrijednosti pa tako TRUE, 't', 'true', 'y', 'yes', 1 označavaju TRUE a FALSE, 'f', 'false', 'n', 'no', označavaju FALSE.

6.5. Binarni tipovi podataka

Binarni tipovi podataka koriste se za pohranu binarnih podataka poput slika, zvuka, videa, dokumenata i drugih datoteka koje nisu običan tekst. Ti podaci mogu biti pohranjeni kao BYTEA, BLOB ili Large Object (LOB), ovisno o sustavu baze podataka.

PostgreSQL koristi BYTEA i Large Objects (LOB) za pohranu binarnih podataka. BYTEA pohranjuje binarne podatke do 1 GB po polju dok LOB može pohraniti do 4TB i pohranjuje se izvan tablice, čime omogućuje bolje performanse pri radu s velikim datotekama.

*Primjer **PostgreSQL**:*

```
CREATE TABLE slike (
    id SERIAL PRIMARY KEY,
    naziv TEXT,
    podaci BYTEA
);
INSERT INTO slike (naziv, podaci)
VALUES ('logo', pg_read_binary_file('/path/to/file.png'));
```

- *Unošenje u bazu*

```
SELECT naziv, encode(podaci, 'hex') FROM slike;
```

- *Dohvaćanje podataka*

```
CREATE TABLE dokumenti (
    id SERIAL PRIMARY KEY,
    naziv TEXT,
    podaci OID
);
```

- *LOB-ovi se spremaju pomoću OID-a (Object Identifier).*

```
INSERT INTO slike (naziv, podaci) VALUES ('logo',
lo_import('/path/to/file.png'));
```

- *Unošenje u bazu*

```
SELECT id, naziv, podaci FROM slike;
```

- *Dohvaćanje OID-a slike*

```
SELECT lo_export(podaci, '/path/to/save/image.png') FROM
slike WHERE naziv = 'logo';
```

- *Dohvaćanje slike natrag na disk*

MySQL koristi BLOB (engl. Binary Large Object) tipove podataka za pohranu binarnih podataka. Ovisno o veličini podataka, postoje četiri vrste BLOB tipova: TINYBLOB koji prima

do 255 bajtova, BLOB 65.535 bajtova (64 KB), MEDIUMBLOB 16.777.215 bajtova (16 MB) i LONGBLOB 4.294.967.295 bajtova (4 GB).

*Primjer **MySQL**:*

```
CREATE TABLE slike (
    id INT AUTO_INCREMENT PRIMARY KEY,
    naziv VARCHAR(255),
    podaci BLOB
);
INSERT INTO slike (naziv, podaci) VALUES ('logo',
LOAD_FILE('/path/to/file.png'));
```

- *Unošenje u bazu*

```
SELECT naziv, HEX(podaci) FROM slike;
```

- *Dohvaćanje podataka*

```
SELECT podaci FROM slike WHERE naziv = 'logo' INTO
DUMPFILE '/path/to/save/image.png';
```

- *Dohvaćanje slike natrag na disk*

6.6. Jedinstveni identifikator – UUID (engl. *Universally Unique Identifier*)

UUID (engl. *Universally Unique Identifier*) je 128-bitni jedinstveni identifikator koji se koristi za jedinstveno označavanje zapisa u bazama podataka. UUID-ovi su globalno jedinstveni i nisu vezani za određenu bazu podataka, što ih čini korisnima za distribuirane sustave gdje se zapisi kreiraju na različitim mjestima bez centralne baze koja dodjeljuje ID-jeve. UUID je toliko velik da je vjerojatnost ponavljanja gotovo nula. UUID ima 128 bita (16 bajtova) i obično se zapisuje kao niz od 36 znakova u hex formatu s crtama (-). PostgreSQL ima nativnu podršku za UUID kroz tip podataka UUID dok se u MySQL bazi reprezentira kao CHAR(36) ili BINARY(16).

Primjer UUID-a (standardni zapis, 36 znakova): 550e8400-e29b-41d4-a716-446655440000.

*Primjer **PostgreSQL**:*

```
CREATE TABLE korisnici (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    ime TEXT NOT NULL
);
```

6.7. Polja u bazama podataka

Polja (engl. *array*) su strukture podataka koje omogućuju pohranu više vrijednosti istog tipa unutar jednog atributa u tablici baze podataka. Umjesto da se isti podaci spremaju u više redaka ili kroz dodatne povezane tablice, polja omogućuju jednostavniju organizaciju podataka. Prednosti korištenja polja uključuju veću fleksibilnost u modeliranju podataka, manji broj redaka u tablicama te lakše dohvaćanje povezanih podataka. Međutim, polja mogu otežati indeksiranje i filtriranje podataka jer standardne SQL operacije nisu uvijek optimizirane za rad s njima.

PostgreSQL nudi nativnu podršku za ARRAY tip podatka. Polja u PostgreSQL-u mogu sadržavati elemente bilo kojeg tipa, a moguće ih je indeksirati i pretraživati pomoću ugrađenih funkcija.

Primjer PostgreSQL:

```
CREATE TABLE zaposlenik (
    id SERIAL PRIMARY KEY,
    ime TEXT,
    vjestine TEXT[]
);
INSERT INTO zaposlenik (ime, vjestine) VALUES ('Ana',
ARRAY['SQL', 'Python', 'Java']);
```

- *Spremanje u polje u bazu*

```
SELECT * FROM zaposlenik WHERE 'Python' = ANY(vjestine);
```

- *Dohvaćanje podataka iz polja*

PostgreSQL omogućuje rad s poljima kroz funkcije kao što su ***array_length()*** za duljinu polja i ***unnest()*** za rastavljanje polja u zasebne retke.

MySQL ne podržava ARRAY tip podatka. Umjesto toga, često se koristi JSON tip podatka za pohranu više vrijednosti unutar jednog atributa.

*Primjer **MySQL**:*

```
CREATE TABLE zaposlenik (
    id INT AUTO_INCREMENT PRIMARY KEY,
    ime VARCHAR(100),
    vjestine JSON
);
INSERT INTO zaposlenik (ime, vjestine) VALUES ('Ana',
'["SQL", "Python", "Java"]');
```

- *Spremanje u JSON u bazu*

```
SELECT * FROM zaposlenik WHERE JSON_CONTAINS(vjestine,
'"Python"');
```

- *Pretraživanje JSON polja*

Iako JSON omogućuje fleksibilnost, on ne nudi istu razinu indeksiranja i performansi kao nativni ARRAY tip u PostgreSQL-u. Dodatno, manipulacija JSON podacima može biti složenija zbog potrebe za specifičnim funkcijama.

6.8. JSON tipovi podataka u bazama podataka

JSON (engl. *JavaScript Object Notation*) je široko korišten format za pohranu i prijenos strukturiranih podataka. Njegova fleksibilnost omogućuje pohranu složenih struktura unutar jednog atributa, što ga čini korisnim za aplikacije koje zahtijevaju dinamične podatkovne modele. Rad s JSON podacima može biti sporiji u usporedbi s relacijskim pristupom jer zahtijeva posebne funkcije za dohvaćanje i manipulaciju.

PostgreSQL podržava dva različita tipa podataka za pohranu JSON sadržaja a to su:

- JSON – pohranjuje podatke u obliku tekstualnog zapisa, bez optimizacije za pretragu;
- JSONB – binarno pohranjeni JSON, optimiziran za brže pretrage i indeksiranje.

Maksimalna veličina JSON/JSONB podataka u PostgreSQL-u iznosi 1 GB po polju.

*Primjer **PostgreSQL**:*

```
CREATE TABLE proizvodi (
    id SERIAL PRIMARY KEY,
    naziv TEXT,
    podaci JSON,
    podaci_b JSONB
);
```

- *Definiranje tablice s JSON i JSONB tipovima u PostgreSQL-u*

```
INSERT INTO proizvodi (naziv, podaci, podaci_b) VALUES
('Laptop', '{"procesor": "Intel i7", "RAM": 16, "disk":
"512GB SSD"}', '{"procesor": "Intel i7", "RAM": 16,
"disk": "512GB SSD"}');
```

- *Umetanje podataka*

```
SELECT podaci->>'procesor' FROM proizvodi WHERE naziv =
'Laptop';
```

- *podaci->>'procesor' sintaksa koristi operatore za rad s JSON podacima.*
- *->> dohvaća vrijednost JSON ključa kao običan tekstualni (STRING) podatak.*

MySQL podržava JSON tip podataka, ali nema odvojene JSON i JSONB tipove – svi podaci se implicitno pohranjuju u binarnom obliku, slično PostgreSQL-ovom JSONB tipu.

Primjer:

```
CREATE TABLE proizvodi (
    id INT AUTO_INCREMENT PRIMARY KEY,
    naziv VARCHAR(100),
    podaci JSON
);
```

```
INSERT INTO proizvodi (naziv, podaci) VALUES ('Laptop',
'{"procesor": "Intel i7", "RAM": 16, "disk": "512GB
SSD"}');
```

- *umetanje podataka*

```
SELECT podaci->>'$.procesor' FROM proizvodi WHERE naziv =
'Laptop';
```

- *dohvaćanje podataka iz JSON-a*

Maksimalna veličina JSON polja u MySQL-u je 64 KB, što je značajno manje od PostgreSQL-ovog ograničenja od 1 GB.

7. CRUD (engl. *create, read, update, delete*)

CRUD je akronim koji označava osnovne operacije koje se mogu izvršavati nad podacima u bazama podataka:

- Create (Kreiranje) – umetanje novih podataka u bazu.
- Read (Čitanje) – dohvaćanje podataka iz baze.
- Update (Ažuriranje) – izmjena postojećih podataka.
- Delete (Brisanje) – uklanjanje podataka iz baze.

7.1. Kreiranje tablica – CREATE TABLE naredba

Naredba CREATE TABLE omogućuje definiranje strukture tablice, uključujući stupce, njihove tipove podataka i ograničenja.

Sintaksa:

```
CREATE TABLE naziv_tablice (
    naziv_stupca1 tip_podatka [OGRANIČENJE],
    naziv_stupca2 tip_podatka [OGRANIČENJE],
    ...
    [OGRANIČENJE_TABLICE]
);
```

Primjer:

```
CREATE TABLE studenti (
    id INTEGER PRIMARY KEY,
    ime VARCHAR(50) NOT NULL,
    prezime VARCHAR(50) NOT NULL,
    datum_rodenja DATE,
    email VARCHAR(100) UNIQUE
);
```

7.1.1. Kreiranje privremenih tablica

Privremene tablice (engl. *Temporary tables*) koriste se za pohranu privremenih podataka koji su potrebni tijekom trajanja sesije ili unutar određenog procesa baze podataka. One omogućuju učinkovitu obradu podataka bez utjecaja na glavne (trajne) tablice u bazi podataka te se koriste za povećanje performansi kod složenih upita. Privremene tablice korisne su u situacijama kada se obrađuju podaci koji nisu trajni, ali su potrebni za izvođenje složenih upita i analize. Različiti sustavi baza podataka implementiraju ih na različite načine, ali osnovni koncept ostaje isti – omogućuju privremenu pohranu podataka s automatskim uklanjanjem nakon završetka sesije.

Postoje dvije glavne vrste privremenih tablica:

1. **Lokalne privremene tablice**
   Vidljive su samo unutar trenutne sesije korisnika. Automatski se brišu nakon zatvaranja sesije. U PostgreSQL-u definiraju se s ključnom riječi TEMP ili TEMPORARY.
2. **Globalne privremene tablice**
   Dostupne su svim korisnicima baze podataka, ali se podaci unutar njih čuvaju samo tijekom trajanja sesije korisnika koji ih je unio. PostgreSQL i MySQL ne podržavaju globalne privremene tablice na isti način kao primjerice SQL Server, ali se slična funkcionalnost može postići putem običnih tablica i brisanja podataka nakon korištenja.

Privremene tablice često se koriste za:

- Složene upite – kada se obrađuje velik broj podataka i potrebno je privremeno pohraniti međurezultate;
- ETL procese (engl. *Extract, Transform, Load*) – za privremeno skladištenje podataka tijekom prijenosa i transformacije podataka;
- Grupiranje i filtriranje podataka – za optimizaciju performansi složenih upita;
- Povećanje brzine obrade – jer se privremene tablice obično pohranjuju u memoriji.

U PostgreSQL-u, privremena tablica može se stvoriti pomoću TEMP ili TEMPORARY, a u MySQL-u se koristi TEMPORARY. Privremene tablice automatski se brišu kada se sesija zatvori.

*Primjer **PostgreSQL**:*

```
CREATE TEMP TABLE privremena_tablica (
    id SERIAL PRIMARY KEY,
    naziv TEXT,
    vrijednost NUMERIC
);
```

*Primjer **MySQL**:*

```
CREATE TEMPORARY TABLE privremena_tablica (
    id INT AUTO_INCREMENT PRIMARY KEY,
    naziv VARCHAR(100),
    vrijednost DECIMAL(10,2)
);
```

Ako je potrebno, privremena tablica može se ručno izbrisati naredbom:

```
DROP TABLE privremena_tablica;
```

7.1.2. Ograničenja stupaca i tablice

Ograničenja stupaca definiraju opcionalna pravila za stupac, kao što su:

- **PRIMARY KEY**: Jedinstveni identifikator svakog retka.
  Primjer: `id INTEGER PRIMARY KEY`
- **NOT NULL**: Sprečava unos NULL vrijednosti.
  Primjer: `ime VARCHAR(50) NOT NULL`
- **UNIQUE**: Osigurava da su sve vrijednosti u stupcu jedinstvene.
  Primjer: `email VARCHAR(100) UNIQUE`
- **CHECK**: Definira uvjete koje vrijednosti moraju zadovoljiti.
  Primjer: `ocjena INTEGER CHECK (ocjena BETWEEN 1 AND 5)`
- **DEFAULT**: Postavlja zadanu vrijednost za stupac ako korisnik ne unese ništa.
  Primjer: `status VARCHAR(20) DEFAULT 'neaktivan'`
- **REFERENCES**: Ograničenje koje povezuje stupac s drugim tablicama te određuje da vrijednost stupca mora biti valjana vrijednost primarnog ključa u drugoj tablici.
  Primjer: `predmet_id INTEGER REFERENCES predmeti(id)`

Ograničenja tablice definira pravila na razini cijele tablice, kao što su primarni ili strani ključevi. Ova ograničenja djeluju na razini cijele tablice, često kombinirajući više stupaca. Svako ograničenje stupca može biti napisano kao ograničenje tablice.

- **PRIMARY KEY** (na razini tablice)
  Definira primarni ključ za tablicu, može uključivati više stupaca (složeni ključ).
  Primjer: `CONSTRAINT pk_primarni PRIMARY KEY (id, godina)`
- **REFERENCES/ FOREIGN KEY** (strani ključ na razini tablice)
  Povezuje više stupaca s ključem iz druge tablice.
  Primjer: `CONSTRAINT fk_upisi FOREIGN KEY (student_id, predmet_id) REFERENCES upisi(student_id, predmet_id)`
- **UNIQUE** (na razini tablice)
  Osigurava da kombinacija vrijednosti u više stupaca bude jedinstvena.
  Primjer: `CONSTRAINT un_ime_prezime UNIQUE (ime, prezime)`
- **CHECK** (na razini tablice)
  Omogućuje definiciju složenih pravila za više stupaca.
  Primjer: `CONSTRAINT chk_vremenski_razmak CHECK (pocetak <= kraj)`

Provjere i ograničenja mogu se ostvariti i pomoću regularnih izraza. **Regularni izrazi** (engl. *Regular Expressions* ili kraće *regex*) moćan su alat za pretraživanje, provjeru i manipulaciju tekstualnih podataka. Omogućuju definiranje uzoraka koji opisuju strukturu teksta koji se traži ili obrađuje. Koriste se u raznim programskim jezicima i alatima.

Primjer:

```
CREATE TABLE korisnici (
    id SERIAL PRIMARY KEY,
    email VARCHAR(100) NOT NULL,
    CONSTRAINT chk_email_format CHECK (email ~
'^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$')
);
```

PostgreSQL ima izvrsnu podršku za regularne izraze kroz operator ~ (*sensitive*- podudara se s obzirom na verzale i kurente) i ~* (*case-insensitive* – mala i velika slova su nebitna). MySQL ne podržava regularne izraze unutar ograničenja CHECK jer se CHECK u MySQL-u često ignorira. Međutim, možete koristiti REGEXP u upitima za validaciju.

Preporučene poveznice za učenje regularnih izraza:

- Regex for regular folk: https://refrf.dev/en/chapters/escapes - učenje pisanja regularnih izraza;
- Regex Crossword; https://regexcrossword.com/ – vježbanje regularnih izraza;
- Regex Search: https://ihateregex.io/ - alat za traženje regularnih izraza;
- Regex Tester: https://regex101.com/ - alat za testiranje regularnih izraza.

7.1.3. Prilagođeni tipovi podataka i domene

PostgreSQL baza omogućuje definiranje prilagođenih (engl. *Custom Data Types*) tipova podataka i domena (engl. *domains*) kako bi se pojednostavilo rukovanje podacima i osigurala dosljednost. Navedene mogućnosti nisu dostupne u MySQL bazi.

Domena u SQL-u predstavlja prilagođeni tip podataka koji uključuje definiciju osnovnog tipa podataka zajedno s pravilima i ograničenjima. Domenama se dodjeljuju pravila kako bi se osigurala valjanost podataka. Kod definiranja koristi se ključna riječ **DOMAIN**.

Primjer:

```
CREATE DOMAIN telefon AS VARCHAR(15)
CONSTRAINT chk_telefon CHECK
(VALUE ~ '^\+?[0-9]{10,15}$');
```

- *Kreiranje domene za validaciju telefonskih brojeva*

```
CREATE TABLE kontakti (
    id SERIAL PRIMARY KEY,
    broj_telefona telefon
);
```

- *Korištenje domena*

```
DROP DOMAIN broj_telefona;
```

- *Brisanje domene*

Domena se može koristiti u više tablica bez ponavljanja pravila. Promjenom domene ažuriraju se pravila u svim tablicama koje je koriste.

Prilagođeni tipovi podataka omogućuju definiranje složenih struktura podataka koje nisu standardno podržane u SQL-u. Koriste se za organizaciju podataka ili za rad s objektima. Kod definiranja koristi se ključna riječ **TYPE**.

Vrste prilagođenih tipova podataka:

1. **ENUM (Enumerirani tipovi)**
   Enumerirani tipovi omogućuju definiranje niza dozvoljenih vrijednosti.
   Primjer:

```
CREATE TYPE status AS ENUM ('aktivno', 'neaktivno',
'na_cekanju');
```

- *definiranje*

```
CREATE TABLE zadaci (
    id SERIAL PRIMARY KEY,
    naziv VARCHAR(100) NOT NULL,
    stanje status NOT NULL
);
```

- *u ovom primjeru, stupac stanje može sadržavati samo vrijednosti: aktivno, neaktivno ili na_cekanju.*

2. **Korisnički definirane strukture (engl. *Composite Types*)**
   Koriste se za definiranje složenih tipova koji sadrže više atributa.
   Primjer:

```
CREATE TYPE adresa AS (
    ulica VARCHAR(100),
    grad VARCHAR(50),
    drzava VARCHAR(50),
    postanski_broj VARCHAR(10)
);
```

7.2. Promjene tablica - ALTER TABLE naredba

ALTER TABLE naredba omogućava promjenu postojeće tablice bez potrebe za njenim brisanjem i ponovnim kreiranjem. Može se koristiti za dodavanje, izmjenu i brisanje stupaca, ograničenja (CONSTRAINT), tipova podataka i drugih svojstava tablice. U nastavku slijede najčešći primjeri korištenja ALTER naredbe.

- Dodavanje novog stupca (**ADD COLUMN**)

  Primjer:

  ```
  ALTER TABLE studenti ADD COLUMN email VARCHAR(100);
  ALTER TABLE studenti ADD COLUMN status VARCHAR(20) NOT
  NULL DEFAULT 'aktivan';
  ```

- Brisanje stupca (DROP COLUMN)

  Primjer:

  ```
  ALTER TABLE studenti DROP COLUMN email;
  ```

- Promjena tipa podataka stupca (ALTER COLUMN SET DATA TYPE)

  Primjer:

  ```
  ALTER TABLE studenti ALTER COLUMN email TYPE TEXT;
  ```

  - *Mijenja tip email stupca u TEXT.*

  PostgreSQL: Ako novi tip nije kompatibilan sa starim, treba eksplicitna konverzija:

  Primjer:

  ```
  ALTER TABLE studenti ALTER COLUMN datum_rodjenja TYPE
  VARCHAR USING datum_rodjenja::VARCHAR;
  ```

- Promjena zadanih vrijednosti **(SET/DROP DEFAULT)**

  Primjer:

  ```
  ALTER TABLE studenti ALTER COLUMN status SET DEFAULT
  'neaktivan';
  ```

  - *Postavlja 'neaktivan' kao zadanu vrijednost za buduće unose.*

  ```
  ALTER TABLE studenti ALTER COLUMN status DROP DEFAULT;
  ```

  - *Brisanje zadane vrijednosti*

- Dodavanje i brisanje ograničenja **(ADD/DROP CONSTRAINT)**

  Primjer:

  ```
  ALTER TABLE studenti ADD CONSTRAINT un_email UNIQUE
  (email);
  ```

  - *Dodavanje UNIQUE ograničenja*

```
ALTER TABLE studenti ADD CONSTRAINT chk_ocjena CHECK
(ocjena BETWEEN 1 AND 5);
```

- *Dodavanje CHECK ograničenja*

```
ALTER TABLE studenti ADD CONSTRAINT fk_predmet FOREIGN KEY
(predmet_id) REFERENCES predmeti(id);
```

- *Dodavanje stranog ključa (FOREIGN KEY)*

```
ALTER TABLE studenti DROP CONSTRAINT chk_ocjena;
```

- *Brisanje ograničenja*

- Preimenovanje stupca **(RENAME COLUMN)**

  Primjer:

  ALTER TABLE studenti RENAME COLUMN email TO e_mail;

- Preimenovanje tablice **(RENAME TO)**

  Primjer:

  ```
  ALTER TABLE studenti RENAME TO ucenici;
  ```

- Aktiviranje i deaktiviranje ograničenja (ENABLE/DISABLE TRIGGER - PostgreSQL)

  Primjer:

  ```
  ALTER TABLE studenti DISABLE TRIGGER ALL;
  ```

  - *Onemogućavanje svih FOREIGN KEY provjera*

  ```
  ALTER TABLE studenti ENABLE TRIGGER ALL;
  ```

  - *Ponovno omogućavanje*

ALTER TABLE omogućuje fleksibilnu modifikaciju tablica bez brisanja podataka. Postoje i druge operacije koje se mogu izvesti nad tablicom, ovisno o sustavu za upravljanje bazom podataka (DBMS-u). Gotovo sve što se definira pri izradi tablice može se naknadno izmijeniti pomoću ALTER TABLE. PostgreSQL pruža više mogućnosti, dok MySQL ima ograničenu podršku za neke operacije (npr. CHECK).

7.3. Umetanje podataka u bazu – INSERT naredba

INSERT je SQL naredba koja služi za dodavanje novih redaka u tablicu baze podataka.

Sintaksa:

```
INSERT INTO naziv_tablice (stupac1, stupac2, stupac3, ...)
VALUES (vrijednost1, vrijednost2, vrijednost3, ...);
```

- *naziv_tablice – ime tablice u koju se unosi novi zapis.*
- *stupac1, stupac2, ... – popis stupaca u koje će se umetnuti podaci.*
- *VALUES (vrijednost1, vrijednost2, ...) – vrijednosti koje će biti umetnute u stupce.*

Redoslijed stupaca u VALUES mora odgovarati onome u INSERT INTO. AUTO_INCREMENT / SERIAL stupci ne moraju biti eksplicitno navedeni. Efikasnost umetanja može se poboljšati grupiranjem više redaka u jednom INSERT pozivu.

Primjeri:

```
INSERT INTO zaposlenik (ime, prezime, placa)
VALUES ('Marko', 'Novak', 7500);
```

- *Umetanje jednog zapisa*

```
INSERT INTO zaposlenik (ime, prezime, placa)
VALUES
    ('Ana', 'Kovač', 8200),
    ('Ivan', 'Horvat', 9000),
    ('Luka', 'Marić', 6700);
```

- *Umetanje više redaka odjednom poboljšava performanse baze podataka.*

```
INSERT INTO zaposlenik
VALUES (5, 'Petra', 'Babić', 8800);
```

- *Umetanje zapisa bez navođenja stupaca*
- *redoslijed mora odgovarati redoslijedu u tablici u bazi podataka*

```
INSERT INTO arhiva_zaposlenika (ime, prezime, placa)
SELECT ime, prezime, placa FROM zaposlenik WHERE placa <
7000;
```

- *Ovo kopira sve zaposlenike s plaćom manjom od 7000 u tablicu arhiva_zaposlenika.*

```
INSERT INTO zaposlenik (ime, prezime, placa)
VALUES ('Tomislav', 'Radić', DEFAULT);
```

- *Umetanje podataka s podrazumijevanim vrijednostima (DEFAULT)*
- *Ako je placa definirana s DEFAULT 5000, Tomislav Radić će dobiti plaću od 5000.*

Specifičnosti za PostgreSQL

PostgreSQL podržava nekoliko dodatnih mogućnosti kod INSERT operacija:

Primjeri:

```
INSERT INTO zaposlenik (ime, prezime, placa)
VALUES ('Ema', 'Jurčić', 7200)
RETURNING id;
```

- *Ova naredba vraća ID novog zapisa, što je korisno za aplikacije koje odmah trebaju raditi s novim podacima.*

```
INSERT INTO zaposlenik (id, ime, prezime, placa)
VALUES (1, 'Ema', 'Jurčić', 7500)
ON CONFLICT (id) DO NOTHING;
```

- *Ako se pokuša umetnuti redak s već postojećim ID-jem, naredba neće izazvati grešku, nego će biti ignorirana.*

```
INSERT INTO zaposlenik (id, ime, prezime, placa)
VALUES (1, 'Ema', 'Jurčić', 7800)
ON CONFLICT (id) DO UPDATE SET placa = EXCLUDED.placa;
```

- *Ovdje se plaća ažurira na novu vrijednost (7800) ako redak s istim id već postoji.*

```
INSERT INTO zaposlenik (ime, placa)
VALUES ('Luka Marić', CAST('7500' AS DECIMAL));
INSERT INTO zaposlenik (ime, placa)
VALUES ('Luka Marić', '7500'::DECIMAL);
```

- *Ovo osigurava da se VARCHAR vrijednost '7500' pretvori u DECIMAL*
- *Konverzijski operatori kod INSERT naredbe*

Specifičnosti za MySQL

```
INSERT IGNORE INTO zaposlenik (id, ime, prezime, placa)
VALUES (1, 'Ema', 'Jurčić', 7500);
```

- *Ako id već postoji, naredba neće izazvati grešku, nego će se jednostavno preskočiti.*

```
INSERT INTO zaposlenik (id, ime, prezime, placa)
VALUES (1, 'Ema', 'Jurčić', 7800)
ON DUPLICATE KEY UPDATE placa = VALUES(placa);
```

- *Ako postoji konflikt na PRIMARY KEY ili UNIQUE indeksu, može se izvršiti ažuriranje umjesto umetanja*
- *Ovo će ažurirati plaću ako id = 1 već postoji.*

```
INSERT INTO zaposlenik (ime, placa)
VALUES ('Luka Marić', CAST('7500' AS DECIMAL(10,2)));
INSERT INTO zaposlenik (ime, placa)
VALUES ('Luka Marić', CONVERT('7500', DECIMAL(10,2)));
```

- *Konverzijski operatori kod INSERT naredbe*

7.4. Brisanje podataka iz baze – DELETE naredba

DELETE je SQL naredba koja se koristi za brisanje redaka iz tablice baze podataka.

Sintaksa:

```
DELETE FROM naziv_tablice WHERE uvjet;
```

- *naziv_tablice – ime tablice iz koje se brišu podaci.*
- *WHERE uvjet – uvjet koji određuje koje retke treba obrisati.*

Ukoliko se WHERE klauzula izostavi, svi redovi iz tablice će biti obrisani!

Primjeri:

```
DELETE FROM zaposlenik WHERE id = 5;
```

- *Brisanje pojedinog zapisa*
- *Ova naredba briše zaposlenika s id = 5.*

```
DELETE FROM zaposlenik WHERE placa < 5000;
```

- *Brisanje više redaka odjednom*
- *Ova naredba briše sve zaposlenike koji imaju plaću manju od 5000.*

```
DELETE FROM zaposlenik;
```

- *Brisanje svih zapisa iz tablice*

ON DELETE pravila za povezane tablice

Ako tablica sadrži strane ključeve, potrebno je definirati kako će se ponašati povezani zapisi kada dođe do brisanja.

Tablica 66. Opcije za *ON DELETE* kod povezanih tablica

Opcija	Opis
ON DELETE CASCADE	Briše redak u povezanoj tablici ako se glavni redak izbriše.
ON DELETE SET NULL	Postavlja strani ključ na NULL ako se glavni redak izbriše.
ON DELETE RESTRICT	Sprječava brisanje ako postoje povezani zapisi.

Primjer:

```
CREATE TABLE studenti (
    id SERIAL PRIMARY KEY,
    ime VARCHAR(50) NOT NULL
);
CREATE TABLE upisi (
    id SERIAL PRIMARY KEY,
```

```
    student_id INTEGER REFERENCES studenti(id)
    ON DELETE CASCADE,
   predmet VARCHAR(50) NOT NULL
);
```

- *Ako se student izbriše iz studenti, svi njegovi upisi će se automatski izbrisati.*
- *Ako koristimo ON DELETE CASCADE, kada se student obriše iz studenti, svi njegovi upisi se također brišu.*
- *Ako koristimo ON DELETE SET NULL, onda će se student_id u tablici upisi postaviti na NULL umjesto da se redak briše.*
- *Ako koristimo ON DELETE RESTRICT, baza neće dopustiti brisanje studenta ako postoje povezani upisi.*

DELETE vs. *TRUNCATE*

Tablica 67. Opcije za brisanje podataka

Naredba	Opis
DELETE	Briše redove prema WHERE uvjetu. Može se vratiti ROLLBACK-om.
TRUNCATE	Briše sve redove iz tablice bez mogućnosti vraćanja podataka.

Primjer:

```
TRUNCATE TABLE zaposlenik;
```

- *TRUNCATE je brži, ali ne može koristiti WHERE ni ROLLBACK!*

Specifičnosti za PostgreSQL

Primjer:

```
DELETE FROM zaposlenik WHERE placa < 5000
RETURNING *;
```

- *Ova naredba briše sve zaposlenike s plaćom manjom od 5000 i vraća njihove podatke.*

Specifičnosti za MySQL

Primjer:

```
DELETE FROM zaposlenik
WHERE placa < 5000
LIMIT 2;
```

- *MySQL omogućuje brisanje ograničenog broja redaka*
- *Ova naredba briše samo dva zaposlenika koji imaju plaću manju od 5000.*

7.5. Ažuriranje podataka u bazi – UPDATE naredba

UPDATE je SQL naredba koja se koristi za ažuriranje postojećih podataka u tablici baze podataka.

Sintaksa:

```
UPDATE naziv_tablice
SET stupac1 = vrijednost1, stupac2 = vrijednost2, ...
WHERE uvjet;
```

- *naziv_tablice – ime tablice u kojoj se podaci ažuriraju.*
- *SET stupac = vrijednost – definira koje vrijednosti će se promijeniti.*
- *WHERE uvjet – određuje koje redove treba ažurirati.*

Bez WHERE klauzule, svi redovi u tablici će biti ažurirani!

Primjer:

```
UPDATE zaposlenik
SET placa = 8500 WHERE id = 5;
```

- *Ažuriranje jednog zapisa*
- *Plaća zaposlenika s id = 5 ažurira se na 8500.*

```
UPDATE zaposlenik
SET placa = placa * 1.1 WHERE placa < 7000;
```

- *Ažuriranje više redaka*
- *Povećava plaću za 10% za sve zaposlenike s plaćom manjom od 7000.*

```
UPDATE zaposlenik  SET placa = 10000;
```

- *Ažuriranje svih redaka*
- *Svim zaposlenikma postavlja plaću na 10000.*

ON UPDATE **pravila za povezane tablice**

Ako tablica koristi strane ključeve, moguće je definirati ponašanje kada se primarni ključ ažurira.

Tablica 68. Opcije za ON UPDATE kod povezanih tablica

Opcija	Opis
ON UPDATE CASCADE	Ažurira strani ključ u povezanoj tablici ako se promijeni primarni ključ.
ON UPDATE SET NULL	Postavlja strani ključ na NULL ako se primarni ključ promijeni.
ON UPDATE RESTRICT	Sprječava ažuriranje ako postoje povezani zapisi.

Primjer:

```
CREATE TABLE studenti (
    id SERIAL PRIMARY KEY,
    ime VARCHAR(50) NOT NULL
);
CREATE TABLE upisi (
    id SERIAL PRIMARY KEY,
     student_id INTEGER REFERENCES studenti(id)
     ON UPDATE CASCADE,
    predmet VARCHAR(50) NOT NULL
);
Povezane tablice s ON UPDATE akcijama
```

- *Ako se id studenta promijeni u studenti, automatski će se ažurirati i u upisi.*

Specifičnosti za MySQL

MySQL podržava REPLACE INTO, koji briše postojeći zapis i unosi novi.

Primjer:

```
REPLACE INTO zaposlenik (id, ime, prezime, placa)
VALUES (5, 'Marko', 'Novak', 9000);
```

- *ako postoji zapis s id = 5, on će biti obrisan i zamijenjen novim.*
- *može dovesti do gubitka podataka jer REPLACE briše cijeli red!*

```
UPDATE zaposlenik SET placa = placa * 1.1
WHERE placa < 7000 LIMIT 2;
```

- *Ova naredba povećava plaću samo za dva zaposlenika s plaćom manjom od 7000.*

```
UPDATE zaposlenik  SET placa = placa + 1000
ORDER BY id ASC LIMIT 1;
```

- *Povećava plaću samo za zaposlenika s najmanjim ID-jem.*
- *Ažuriranje najstarijih ili najnovijih podataka*

```
UPDATE zaposlenik
SET placa = CAST('7500' AS DECIMAL(10,2))
WHERE id = 5;
UPDATE zaposlenik
SET placa = CONVERT('7500', DECIMAL(10,2))
WHERE id = 5;
```

- *Osigurava da je podatak pravilno konvertiran u numerički tip.*

Specifičnosti za PostgreSQL

Primjer:

```
UPDATE zaposlenik SET placa = 9000
WHERE id = 5 RETURNING *;
```

- *Ova naredba vraća ažurirani redak.*

```
UPDATE zaposlenik  SET placa = CAST('7500' AS DECIMAL)
WHERE id = 5;
UPDATE zaposlenik SET placa = '7500'::DECIMAL WHERE id =
5;
```

- *Osigurava da se VARCHAR vrijednost '7500' pretvori u DECIMAL.*

8. TRANSAKCIJE U BAZAMA PODATAKA

8.1. Rad sa transakcijama u bazama podataka

Transakcije u bazama podataka predstavljaju niz operacija koje se izvršavaju kao jedna logička cjelina. Njihova je svrha osigurati dosljednost i integritet podataka, čak i u slučaju nepredviđenih situacija, poput kvara sustava ili istovremenog pristupa više korisnika. Transakcije su ključne za održavanje integriteta baze podataka. Razumijevanje ACID svojstava, razina izolacije i problema koji mogu nastati pri čitanju pomaže u pravilnom dizajniranju sustava. Odabir pravog modela zaključavanja i distribuiranih transakcija može značajno poboljšati performanse sustava u okruženjima s velikim brojem korisnika.

Kako bi se transakcije ispravno koristile, u SQL-u se upotrebljavaju sljedeće naredbe:

Primjer:

```
BEGIN;
UPDATE racuni SET stanje = stanje - 500 WHERE id = 1;
UPDATE racuni SET stanje = stanje + 500 WHERE id = 2;
COMMIT;
```

Ako dođe do problema, može se izvršiti *rollback*:

Primjer:

```
BEGIN;
UPDATE racuni SET stanje = stanje - 500 WHERE id = 1;
UPDATE racuni SET stanje = stanje + 500 WHERE id = 2;
ROLLBACK;
```

Kako bi se osigurala ispravnost transakcija, one moraju zadovoljiti **ACID** svojstva:

- **Atomicity (atomarnost)** – transakcija se izvršava u potpunosti ili se uopće ne izvršava. Ako dođe do prekida, sve promjene se poništavaju (naredba ROLLBACK).
- **Consistency (dosljednost)** – nakon završetka transakcije, baza podataka mora ostati u valjanom stanju, poštujući sva pravila integriteta.
- **Isolation (izolacija)** – istovremene transakcije ne smiju utjecati jedna na drugu na način koji bi narušio konzistentnost podataka.
- **Durability (trajnost)** – kada se transakcija jednom potvrdi (naredba COMMIT), njezini se rezultati trajno pohranjuju u bazu, čak i u slučaju pada sustava.

U sustavima koji koriste više baza podataka ili distribuirane baze podataka, potrebno je koristiti distribuirane transakcije. Jedan od uobičajenih protokola za osiguravanje konzistentnosti u distribuiranim transakcijama je **dvofazno potvrđivanje (engl. *Two-phase commit - 2PC*)**.

- **Faza pripreme** – glavni poslužitelj šalje zahtjev svim sudionicima da pripreme transakciju.
- **Faza izvršenja** – ako svi sudionici potvrde, transakcija se dovršava; u suprotnom, poništava se.

8.2. Razine izolacije transakcija

Različite baze podataka omogućuju definiranje razine izolacije transakcija kako bi se balansirale performanse i sigurnost podataka. Postoje sljedeće razine izolacije:

- **Čitanje nepredanih podataka (engl. *Read uncommitted*)** – transakcija može čitati podatke iz drugih transakcija koje još nisu potvrđene (COMMIT), što može dovesti do problema s nedosljednim čitanjem.
- **Čitanje predanih podataka (engl. *Read committed*)** – transakcija može čitati samo podatke koji su potvrđeni, čime se sprječava čitanje nevaljanih podataka.
- **Ponovljivo čitanje (engl. *Repeatable read*)** – sigurava da podaci ostaju isti unutar transakcije, čak i ako ih druge transakcije mijenjaju. Međutim, ne sprječava pojavu fantomskog čitanja (engl. *phantom read*) problema.
- **Serijalizirano čitanje (engl. *Serializable*)** – Najstroža razina izolacije u kojoj se transakcije izvršavaju kao da su serijalizirane, čime se sprječavaju sve anomalije, ali se smanjuje učinkovitost sustava.

Što je razina izolacije viša, to je bolja zaštita podataka i konzistentnost, ali također dolazi do većeg opterećenja sustava. *Read Uncommitted* se koristi kada su performanse prioritet, a greške u čitanju su prihvatljive. *Serializable* osigurava maksimalnu konzistentnost, ali smanjuje paralelizam i može uzrokovati situacije zastoja (engl. *deadlock*).

U PostgreSQL-u, MySQL-u i drugim relacijskim bazama, razina izolacije se može postaviti ovako:

Primjer:

```
BEGIN;
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
UPDATE korisnici SET saldo = saldo - 100 WHERE id = 1;
COMMIT;
```

- *Primjeri za postavljanje unutar transakcija*

```
SET SESSION CHARACTERISTICS AS TRANSACTION ISOLATION LEVEL
SERIALIZABLE;
```

- *Ova naredba mijenja zadanu razinu izolacije za sve transakcije u trenutnoj sesiji, osim ako neka transakcija eksplicitno postavi drugu razinu izolacije.*

```
ALTER DATABASE moja_baza SET default_transaction_isolation
TO 'repeatable read';
```

- *Ova promjena vrijedi za sve buduće sesije koje se spoje na bazu, osim ako korisnik ručno postavi drugu razinu izolacije unutar sesije ili transakcije.*

Zaključavanje podataka može se podijeliti na dvije razine podjele:

- Prema strategiji upravljanja konfliktima (optimističko i pesimističko zaključavanje);
- Prema tipu pristupa podacima (dijeljeno i ekskluzivno zaključavanje).

Strategije zaključavanja i upravljanja konfliktima

Ove strategije definiraju kada i kako se pristupa zaključavanju podataka:

1. **Optimističko zaključavanje**
   Pretpostavlja da neće biti sukoba među transakcijama.
   Transakcija čita podatke i obrađuje ih bez zaključavanja. Prilikom COMMIT-a provjerava jesu li se podaci promijenili od trenutka kada su pročitani. Ako podaci nisu promijenjeni, transakcija se potvrđuje. Ako jesu, transakcija se poništava (ROLLBACK), a korisnik mora ponoviti operaciju.
   Ovakvo zaključavanje koristi se:
   - kada je vjerojatnost istovremenih sukoba niska;
   - u sustavima gdje je čitanje puno češće od pisanja (npr. analitičke baze podataka).

   Optimističko zaključavanje koristi se kada odaberemo niže razine izolacije, primjerice *Read committed* i *Repeatable read*.
2. **Pesimističko zaključavanje**
   Pretpostavlja da može doći do sukoba, pa odmah zaključava podatke koje transakcija koristi. Kada transakcija pristupi podacima za izmjenu, zaključava ih tako da ih druge transakcije ne mogu koristiti.
   Drugi korisnici moraju čekati dok se zaključavanje ne oslobodi (COMMIT ili ROLLBACK).
   Ovakvo zaključavanje koristi se:
   - Kada je velika vjerojatnost istovremenog pristupa istim podacima;
   - U financijskim sustavima gdje se ne smije dogoditi da se isti resurs mijenja u isto vrijeme.

   Pesimističko zaključavanje koristi se na višim razinama izolacije, poput *Serializable* i u nekim bazama podataka *Repeatable Read*.

Vrste zaključavanja prema tipu pristupa podacima

Ove vrste definiraju kako transakcije mogu koristiti zaključane podatke:

1. **Dijeljeno zaključavanje (engl. *Shared lock*)**
   Omogućuje višestruko čitanje, ali ne dopušta pisanje.

Više transakcija može istovremeno čitati iste podatke, ali nijedna ih ne može mijenjati dok postoji zaključavanje.

2. **Ekskluzivno zaključavanje (engl. *Exclusive lock*)**
   Omogućuje samo jednom procesu pristup podacima – nijedna druga transakcija ne može ni čitati ni pisati dok je zaključavanje aktivno.

8.2.1. Problemi pri čitanju (engl. *Read phenomena*)

Pri istovremenom izvršavanju transakcija mogu se pojaviti sljedeći problemi:

- **Prljavo čitanje (engl. *Dirty read*)** – transakcija čita podatke iz druge transakcije koja još nije završena. Ako se ta druga transakcija poništi, pročitani podaci postaju nevažeći.
- **Neponovljivo čitanje (engl. *Non-repeatable read*)** – ako transakcija ponovi čitanje istih podataka, može dobiti različite rezultate jer je druga transakcija u međuvremenu promijenila ili obrisala podatke.
- **Fantomsko čitanje (engl. *Phantom read*)** – transakcija dobije različite skupove podataka prilikom ponovljenih upita jer je druga transakcija umetnula ili izbrisala retke koji odgovaraju kriterijima upita.

9. DIZAJN BAZE PODATAKA

9.1. Faze dizajna baze podataka

Faze dizajna baze podataka unutar procesa životnog ciklusa razvoja softvera (engl. *Software Development Life Cycle – SDLC*) ključne su za izgradnju učinkovite, skalabilne i optimizirane baze podataka. Navedene faze prate strukturiran pristup, osiguravajući da baza podataka ispunjava zahtjeve korisnika i sustava.

Glavne faze dizajna baze podataka s pripadajućim zadacima u svakoj fazi su:

1. **Analiza zahtjeva**
   - Prikupljanje i dokumentiranje poslovnih i tehničkih zahtjeva.
   - Identifikacija ključnih entiteta, atributa i odnosa među podacima.
   - Definiranje zahtjeva sigurnosti i performansi.
2. **Dizajn baze podataka**
   - Konceptualni dizajn: Izrada ER modela s entitetima, atributima i odnosima.
   - Logički dizajn: Pretvaranje ER modela u relacijski model, definiranje primarnih i stranih ključeva, normalizacija podataka.
   - Fizički dizajn: Implementacija modela u DBMS-u, optimizacija indeksima, definiranje sigurnosnih mjera i strategija za pohranu podataka.
3. **Implementacija i testiranje**
   - Stvaranje baze podataka i tablica prema definiranom dizajnu.
   - Punjenje baze testnim podacima i provjera integriteta.
   - Testiranje performansi upita i optimizacija.
4. **Održavanje i optimizacija**
   - Praćenje performansi i provođenje optimizacija.
   - Upravljanje sigurnosnim kopijama i strategijama oporavka.
   - Nadogradnje i migracije baze podataka prema potrebama sustava.
   - Ovo je faza u kojoj se koriste razne SQL naredbe.

9.2. Pristupi u dizajnu baze podataka

Pri dizajniranju baze podataka koriste se dvije glavne strategije: **Top-Down** (odozgo prema dolje) i **Bottom-Up** (odozdo prema gore). Ove metode definiraju način na koji se podaci organiziraju i modeliraju.

Top-Down (odozgo prema dolje) pristup

Top-Down pristup počinje od šire slike, tj. općeg konceptualnog modela. Sustav se promatra kao cjelina, a zatim se razlaže na manje entitete i odnose. Koristi se kod velikih i složenih sustava gdje je potrebno osigurati konzistentnost i jasnoću podataka. Izrađuje se uobičajeno pomoću ER modela.

Koraci:

1. **Definiranje općih kategorija podataka** – određuju se ključni entiteti i njihovi odnosi.
2. **Podjela entiteta na manje cjeline** – razrađuju se entiteti, atributi i odnosi.
3. **Pretvaranje konceptualnog modela u relacijski model** – definira se struktura tablica.
4. **Normalizacija podataka** – uklanja se redundantnost i povećava integritet podataka.

Top-Down pristup je bolji za velike, složene sustave s jasno definiranim zahtjevima. Optimalan je izbor kada se kreira nova baza podataka.

Bottom-Up (odozdo prema gore) pristup

Počinje od manjih komponenti baze podataka (atributi, tablice) i postupno se spaja u veće cjeline. Pogodan za sustave koji se razvijaju inkrementalno, kada nisu svi zahtjevi unaprijed definirani. Postoji sistem ili specifični podaci koji se koriste te se oko njih kreira novi sistem. Često se koristi u manjim projektima gdje se baza razvija na temelju postojećih podataka. Izrađuje se uobičajeno normalizacijom podataka.

Koraci:

1. **Identifikacija pojedinačnih podataka** – definiraju se tablice i atributi.
2. **Grupiranje podataka u povezane cjeline** – uspostavljaju se odnosi među podacima.
3. **Spajanje tablica u veće cjeline** – povezuju se u logički koherentne sustave.
4. **Optimizacija strukture** – provodi se normalizacija i optimizacija performansi.

Bottom-Up praktičan je pristup za manje projekte i sustav koji raste postupno. Omogućuje fleksibilnost jer se sustav može lako proširiti. Omogućuje se brži početak razvoja bez potrebe za cjelokupnim planiranjem te je optimalan izbor kod migracije baze podataka.

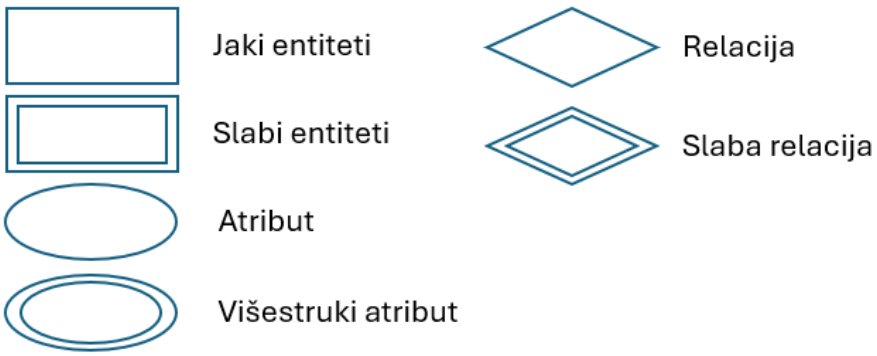
Hibridni pristup

Kombinacija *Top-Down* i *Bottom-Up* metoda. Počinje se s općim modelom, ali se neki dijelovi baze dizajniraju od manjih elemenata prema gore. Koristi se u većim sustavima gdje je potrebno zadržati fleksibilnost, ali i osigurati konzistentnost.

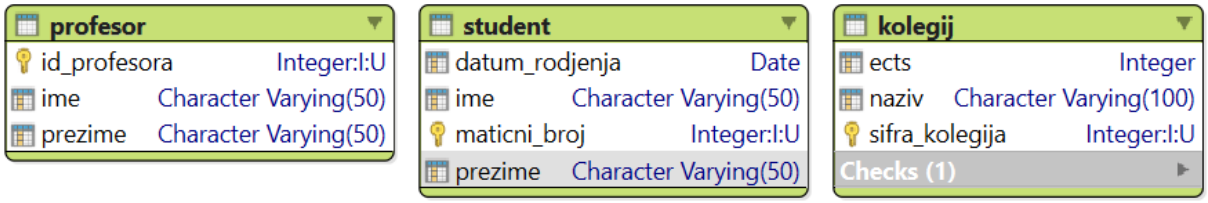
Hibridni pristup pruža fleksibilnost i koristi najbolje od oba modela.

9.3. ER (engl. *Entity-Relationship*) dijagrami

ER dijagrami (engl. *Entity-relationship*) predstavljaju vizualni model podataka koji prikazuje? kako su entiteti u bazi podataka međusobno povezani. Koriste se u fazi dizajniranja baze podataka kako bi se definirala struktura sustava prije implementacije u relacijski model.



Slika 16. Simboli ER dijagrama



Slika 17. Primjer prikaza entiteta i njihovih atributa u ER dijagramu

ER dijagram se sastoji od nekoliko ključnih elemenata:

1. **Entitet (engl. *Entity*)**
   Predstavlja stvarni objekt ili koncept iz domene podataka. Prikazuje se kao pravokutnik. Entiteti se obično imenuju u jednini kako je prikazano na slici 17.
   Može biti:
   - Slabi entitet (engl. *Weak Entity*) – nema vlastiti primarni ključ i ovisi o drugom entitetu.
   - Jaki entitet (engl. *Strong Entity*) – ima vlastiti primarni ključ.
2. **Atributi/ značajke (engl. *Attribute*)**
   Svojstvo entiteta koje opisuje njegovu karakteristiku.
   Može biti:
   - Jednostavni (engl. *Simple*) – ne mogu se dalje dijeliti (npr. ime, prezime).
   - Složeni (engl. *Composite*) – sastoje se od više podatributa (npr. adresa može imati ulicu, broj, grad).

- Višestruki (engl. *Multivalued*) – entitet može imati više vrijednosti za isti atribut (npr. telefonski brojevi).
- Derivirani (engl. *Derived*) – izračunava se iz drugih atributa (npr. starost iz datuma rođenja).

3. **Relacija/ odnos (engl. *Relationship*)**
   Predstavlja vezu između entiteta. Prikazuje se kao romb između entiteta.
   Tipovi odnosa:
   - 1:1 (Jedan-prema-jedan) – jedan entitet je povezan s točno jednim drugim entitetom.
   - 1:M (Jedan-prema-više) – jedan entitet može biti povezan s više drugih entiteta.
   - M:N (Više-prema-više) – više entiteta može biti povezano s više drugih entiteta (zahtijeva dodatnu tablicu u relacijskom modelu).

Relacijski ključevi u ER dijagramu jedinstveno definiraju redak i odnos – relaciju. Ključevi u ER dijagramu mogu biti:

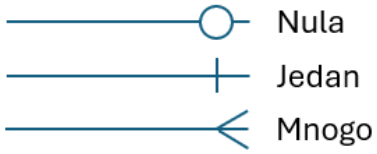
- **Superključ** (engl. *Super Key*) – skup atributa koji može jednoznačno identificirati zapis. U praksi se češće koristi jedan stupac koji može jedinstveno definirati redak.
- **Kandidatni ključ** (engl. *Candidate Key*) – minimalan skup atributa koji može služiti kao primarni ključ, najbolje ukoliko imamo samo jedan stupac zbog jednostavnosti određivanja relacija i dohvaćanja podataka.
- **Složeni ključ** (engl. *Compound key*) - je primarni ključ koji se sastoji od više od jednog atributa (stupca). Koristi se kada nijedan pojedinačni atribut nije jedinstven, ali njihova kombinacija jedinstveno identificira svaki redak u tablici.
  - *Composite key* (složeni ključ) bilo koja kombinacija dva ili više atributa koji zajedno mogu jedinstveno identificirati zapis. Može, ali ne mora biti primarni ključ.
- **Zamjenski ključ** (engl. *Surrogate key*) je umjetni (generirani) ključ koji služi kao primarni ključ u tablici baze podataka, ali nema poslovno značenje. Najčešće je to automatski generirani identifikator, poput autoinkrement broja ili UUID-a.
- **Primarni ključ** (engl. *Primary Key, PK*) – jednoznačno identificira svaki zapis u tablici.
- **Strani ključ** (engl. *Foreign Key, FK*) – referencira primarni ključ druge tablice kako bi se stvorila veza između entiteta.

Prema konvenciji imenovanja, nazivi entiteta trebaju biti u jednini kako bi precizno opisivali jedan zapis u bazi podataka, primjerice "student", "kolegij" ili "profesor". Nazivi atributa također trebaju biti u jednini, napisani malim slovima, pri čemu se za razdvajanje

riječi koristi donja crtica (engl. *snake_case*). Na primjer, atribut koji predstavlja ime studenta trebao bi biti nazvan "ime", dok bi prezime bilo "prezime", a matični broj "maticni_broj". Ova konvencija osigurava konzistentnost i čitljivost modela baze podataka, čineći ga intuitivnijim za korištenje i održavanje. Osim navedenog koristi se i konvencija pri kojoj koristimo veliko slovo kod svakog prvog slova u riječi (engl. *camel case*) primjerice „maticniBroj“ ili „datumRodjenja“.

Kod definiranja veza koristi se notacija odnosa koja se na engleskom naziva ***Crow's Foot Notation***. To je popularan način prikazivanja odnosa u ER dijagramima. Ova notacija koristi simbolične oznake za prikaz tipova veza između entiteta u bazi podataka.

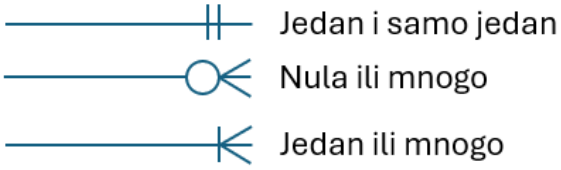
Kada se ER dijagram dizajnira, sljedeći korak njegova je transformacija u relacijsku bazu podataka. Kod transformacije svaki entitet postaje tablica s atributima kao stupcima, relacije se mapiraju pomoću stranih ključeva, a M:N veze se razdvajaju u zasebne tablice s primarnim ključevima povezanih entiteta.



Slika 18. Prikaz mogućih kardinalnosti

Kardinalnost određuje odnos između entiteta u bazi podataka. Za svaki entitet postoji minimalni i maksimalni broj veza koje definiraju njegov odnos s drugim entitetom. Crow's Foot notacija koristi simbolične oznake za prikaz kardinalnosti prikazano na Slici 18. Osnovni simboli su:

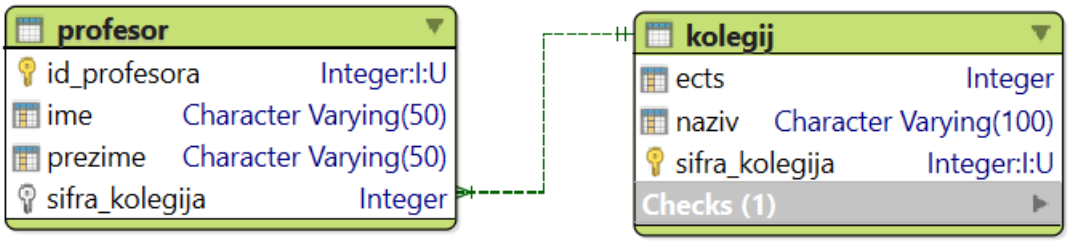
- Nula (simbol kruga)- koja označava da entitet može, ali ne mora imati povezanih instanci u drugom entitetu.
- Jedan (simbol okomita linija)- označava da entitet mora biti povezan s točno jednom instancom drugog entiteta.
- Više (simbol poput "vrana stopala")- označava da jedan entitet može imati više povezanih instanci drugog entiteta.



Slika 19. Minimalna i maksimalna kardinalnost

Kada govorimo o minimalnoj i maksimalnoj kardinalnosti prva oznaka do entiteta određuje odnos a druga linija određuje ograničenje kako je prikazano na Slici 19. Primjerice nula ili mnogo kardinalnost određuje da je odnos mnogo ali može biti i nula.

U primjeru na slici 20. prikazan je odnos entiteta za dva entiteta u bazi podataka – *profesor* i *kolegij*. Za svakog profesora definirano je da može predavati samo jedan kolegij. Kardinalnost ovog odnosa je točno jedan. Minimalni broj kolegija koji profesor može predavati je 1, a maksimalni je također 1. Za svaki kolegij zadaje se jedan ili više profesora koji ga mogu predavati. To znači da jedan kolegij može imati jednog ili više profesora. Minimalni broj profesora koji mogu predavati kolegij je 1, dok maksimalni broj nije ograničen (može biti više profesora). Studenti mogu birati između više profesora. Kardinalnost ovog odnosa je 1:M (jedan-prema-više).

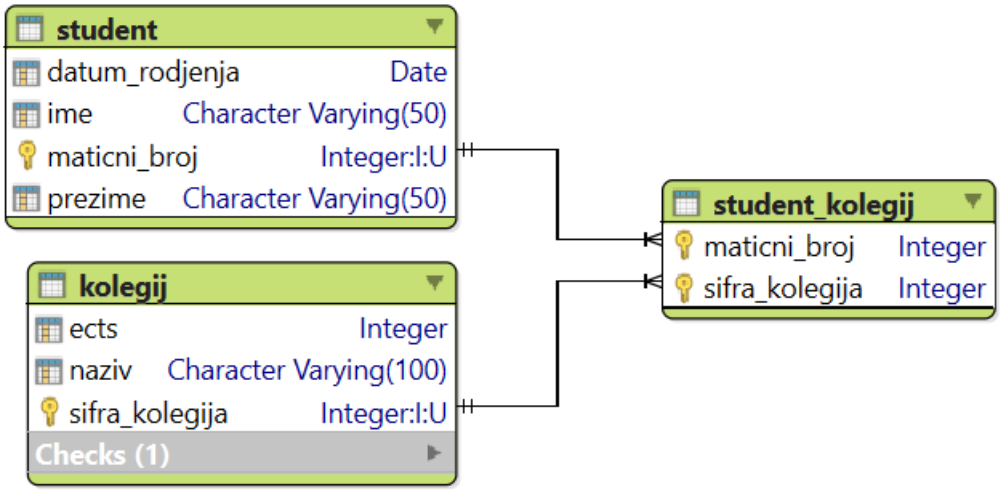


Slika 20. Primjer odnosa jedan prema više

U primjeru odnosa entiteta na slici 21. postoje dva entiteta u bazi podataka *student* i *kolegij*. Budući da jedan student može upisati više kolegija, a svaki kolegij može upisati više studenata, riječ je o M:N (više-prema-više) vezi. Minimalni broj kolegija koje student može upisati je 1, a maksimalni nije ograničen (može upisati više kolegija). Minimalni broj studenata koji mogu biti upisani na kolegij je 1, a maksimalni nije ograničen (više studenata može slušati isti kolegij).

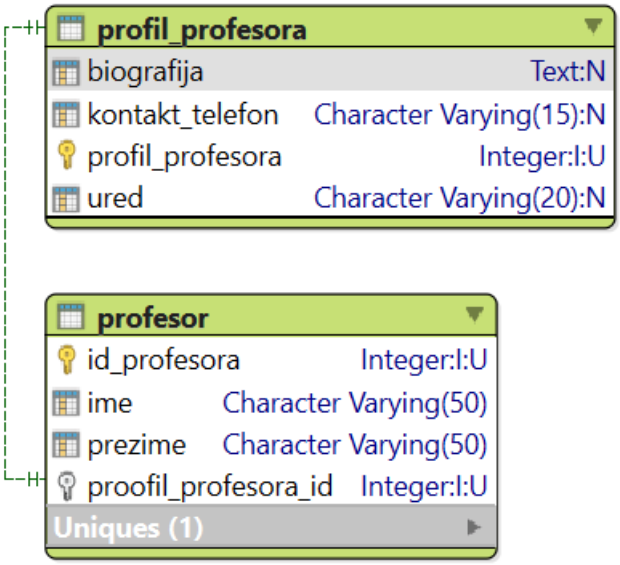
Budući da relacijske baze podataka ne mogu izravno modelirati M:N odnose, koristi se asocijativna tablica *student_kolegij* kako bi se pravilno povezali studenti i kolegiji. Ova tablica omogućava fleksibilno i jasno praćenje koji studenti pohađaju koje kolegije. Ona služi kao posrednik između tablica *student* i *kolegij*. Sadrži strane ključeve iz obje tablice kako bi povezala studente s kolegijima koje su upisali. Primarni ključ tablice *student_kolegij* sastoji se od kombinacije *maticni_broj* i *sifra_kolegija*.

Odnos jedan-prema-više postoji između entiteta student i tablice *student_kolegij*, što znači da jedan student može biti povezan s više zapisa u tablici *student_kolegij*. Slično tome, postoji odnos jedan-prema-više između entiteta kolegij i tablice *student_kolegij*, što znači da jedan kolegij može biti povezan s više zapisa u tablici *student_kolegij*. Posljedično, između entiteta student i kolegij postoji posredni odnos više-prema-više, jer jedan student može upisati više kolegija, a jedan kolegij može biti upisan od strane više studenata. Ova struktura ispravno modelira složeni odnos između studenata i kolegija kroz asocijativnu tablicu.



Slika 21. Primjer posrednog odnosa više prema više

U prikazanom modelu na slici 22. uspostavljen je odnos 1:1 između tablica *profesor* i *profil_profesora*. Tablica *profesor* predstavlja osnovne informacije o profesorima, uključujući njihove jedinstvene identifikatore (*id_profesora*), *ime* i *prezime*. S druge strane, tablica *profil_profesora* sadrži dodatne specifične detalje za profesore, poput biografije, kontaktnog telefona i broja ureda. Veza između ovih dviju tablica ostvarena je pomoću stranog ključa *profil_profesora_id* u tablici *profesor*, koji referencira primarni ključ tablice *profil_profesora*. Time je osigurano da svaki profesor može imati najviše jedan pridruženi zapis u tablici *profil_profesora*, čime se ostvaruje odnos 1:1. Ovaj model omogućuje logičko razdvajanje osnovnih podataka o profesorima od njihovih dodatnih detalja, čuvajući povezanost kroz strani ključ u tablici *profesor*.



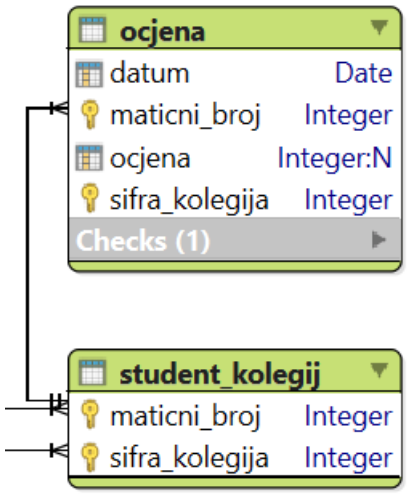
Slika 22. Primjer odnosa jedan prema jedan

U primjeru na slici 23. tablica *ocjena* pohranjuje informacije o ocjenama koje studenti dobivaju na određenim kolegijima. Budući da se *ocjena* ne može definirati bez prethodno

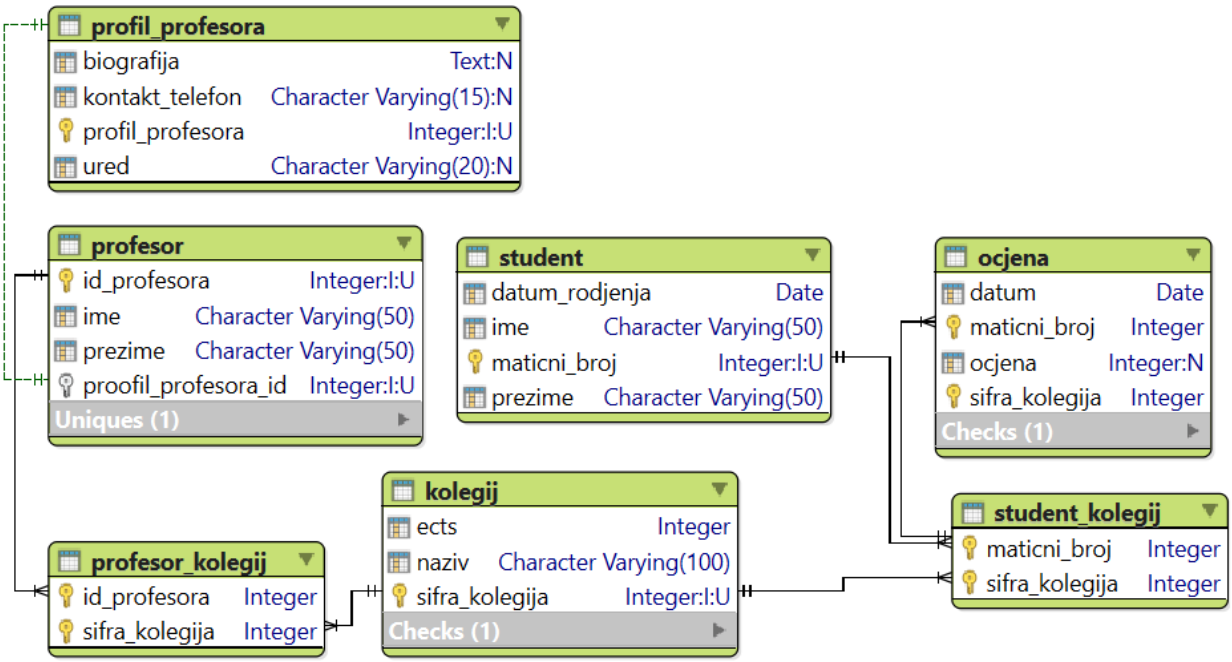
postojeće veze između *student* i *kolegij* (koja je definirana u tablici *student_kolegij*), ona je slabi entitet jer:

- Ne postoji samostalno – mora se referencirati na zapis u tablici *student_kolegij*;
- Primarni ključ je složen – sastoji se od stranih ključeva koji dolaze iz *student_kolegij* (kombinacija *maticni_broj* i *sifra_kolegija*) uz dodatak atributa poput *ocjena* i *datum*.

Jedan zapis u *student_kolegij* može biti povezan s jednom ili više ocjena. Na primjer, student može imati više ocjena za isti kolegij (npr. negativne i pozitivne). Kardinalnost je 1:M (jedan-prema-više).



Slika 23. Primjer slabog entiteta



Slika 24. Primjer cijelog ER dijagrama baze podataka

9.3.1. Koraci izrade ER dijagrama

Za izradu ER (*Entity-Relationship*) dijagrama koriste se određene procedure koje olakšavaju modeliranje baze podataka. Proces se može podijeliti u nekoliko koraka:

Korak 1: Određivanje entiteta u sustavu

- Entitet predstavlja osobu, mjesto, stvar ili koncept u sustavu.
- Naziv entiteta se zapisuje u jednini (npr. student, knjiga).
- Svaka instanca entiteta predstavlja jedan zapis (redak) u tablici.
- Entitet mora imati jedinstveni identifikator (primarni ključ).

Korak 2: Određivanje atributa za entitete

- Atributi su svojstva entiteta (npr. ime, datum rođenja).
- Svaki atribut treba biti atomski, tj. sadržavati jedinstvene vrijednosti zapisane do potrebne granularnosti (npr. ime, prezime umjesto puno ime).
- U ovom koraku određuju se i ključevi:
    - Primarni ključ (PK) – jedinstveno identificira svaku instancu entiteta;
    - Strani ključ (FK) – koristi se za povezivanje s drugim entitetima.

Korak 3: Određivanje odnosa (relacija) između entiteta

- Identificiraju se logičke veze između entiteta (npr. student upisuje kolegij).
- Definira se vrsta odnosa (jedan-prema-jedan, jedan-prema-više, više-prema-više).
- Ako je potrebno, uvode se atributi odnosa (npr. Datum upisa u odnosu Student upisuje kolegij).
- Razmatraju se obveznosti i kardinalnost odnosa (obavezna ili opcionalna veza).

Korak 4: Finalizacija i normalizacija modela

- Relacijski model baze podataka ne podržava izravno odnose više-prema-više pa ih je potrebno razložiti na dvije jedan-prema-više veze uvođenjem posredne (povezne) tablice.
- Provjera redundancije – uklanjaju se suvišni atributi i odnosi kako bi se optimizirala baza podataka.
- Primjena normalizacije – koristi se proces normalizacijskih formi (1NF, 2NF, 3NF, BCNF...) kako bi se eliminirala redundantnost i osigurala konzistentnost podataka.
- Validacija modela – provjerava se je li ER dijagram u skladu s poslovnim zahtjevima i očekivanim podacima.
- Razmatranje izvedbe – analizira se moguće optimizacije, poput uvođenja indeksa ili denormalizacije gdje je potrebno radi poboljšanja performansi.
- Dovršavanje dokumentacije – svi entiteti, atributi, odnosi i pravila integriteta trebaju biti jasno dokumentirani prije implementacije baze podataka.

9.4. Modifikacijske anomalije i normalizacija podataka

Modifikacijske anomalije predstavljaju probleme koji se javljaju prilikom umetanja, ažuriranja ili brisanja podataka u loše dizajniranim relacijskim bazama podataka. One su posljedica redundancije podataka i nedostatka normalizacije. Postoje tri glavne vrste modifikacijskih anomalija:

1. **Anomalija umetanja (engl. *Insertion anomaly*)**
   Javlja se kada nije moguće umetnuti nove podatke bez dodavanja dodatnih, nepotrebnih podataka.
   Primjer: Ako se u tablici Zaposlenik čuvaju i podaci o odjelima, a odjel ne može biti unesen dok ne postoji barem jedan zaposlenik u njemu, nastaje anomalija umetanja.
2. **Anomalija ažuriranja (engl. *Update anomaly*)**
   Pojavljuje se kada promjena jednog podatka zahtijeva ažuriranje više zapisa, što može dovesti do nekonzistentnosti baze podataka.
   Primjer: Ako se adresa odjela ponavlja u više redaka unutar tablice Zaposlenik, promjena adrese odjela zahtijeva ažuriranje svih redaka u kojima se ta adresa pojavljuje. Ako neki redci ostanu nepromijenjeni, dolazi do nekonzistentnosti podataka.
3. **Anomalija brisanja (engl. *Deletion anomaly*)**
   Nastaje kada brisanjem jednog retka iz tablice nehotice gubimo i važne informacije koje ne bi smjele biti obrisane.
   Primjer: Ako u tablici Zaposlenik postoji zapis o jedinom zaposleniku nekog odjela, njegovo brisanje može rezultirati gubitkom podataka o postojanju tog odjela.

Rješenje anomalija je **normalizacija** podataka. Normalizacija uključuje primjenu **normalnih formi (NF)**, pri čemu se:

- podaci dijele u više tablica kako bi se minimizirala redundancija;
- osiguravaju pravilne veze između tablica pomoću stranih ključeva.

9.5. Funkcionalne ovisnosti u bazama podataka

Funkcionalna ovisnost (engl. *functional dependency, FD*) je pravilo koje određuje vezu između atributa u relacijskoj bazi podataka. Kažemo da postoji funkcionalna ovisnost između dva skupa atributa ako vrijednost jednog skupa atributa jednoznačno određuje vrijednost drugog skupa.

Za neku relaciju R (entitet), kažemo da postoji funkcionalna ovisnost X → Y ako vrijednost atributa X jednoznačno određuje vrijednost atributa Y.

- X je determinant (lijeva strana ovisnosti, određuje vrijednost Y);
- Y je ovisni atribut (desna strana ovisnosti, određena vrijednošću X), on funkcijski ovisi o X.

Tablica 69. Tablica Zaposlenik

OIB	ime	prezime	placa
12345678901	Ivan	Horvat	1200€
98765432109	Ana	Kovač	1500€

U tablici Zaposlenik možemo imati sljedeću funkcionalnu ovisnost:

OIB → ime, prezime, placa

- Ako znamo OIB, možemo jedinstveno odrediti ime, prezime i plaću zaposlenika.
- OIB je determinant, a ime, prezime i placa su ovisni atributi.
- Kad bi preokrenuli vezu ne bi dobili ispravan rezultat, primjerice iz ime ne možemo saznati OIB

Preko funkcionalnih ovisnosti:

- prepoznaje se redundancija u bazama podataka;
- pomaže se u procesu normalizacije;
- omogućuje se bolji dizajn baze podataka i sprječavanje anomalija;
- definira se osnova za normalne forme, koje pomažu u organizaciji podataka u relacijskim bazama.

9.5.1. Vrste funkcionalnih ovisnosti

1. **Potpuna funkcionalna ovisnost**
   Svaki atribut s desne strane funkcionalne ovisnosti ovisi o cijelom primarnom ključu, a ne samo o njegovom dijelu.
   Primjer:

   (Student_ID, Predmet_ID) → Ocjena
   - Ocjena ovisi o kombinaciji Student_ID i Predmet_ID, a ne samo o jednom od njih.

2. **Djelomična funkcionalna ovisnost (krši 2NF)**
   Ovisni atribut ovisi samo o dijelu primarnog ključa, a ne o cijelom ključu.
   Primjer:

   (Student_ID, Predmet_ID) → Ime_Studenta
   - Ovdje Ime_Studenta ovisi samo o Student_ID, što znači da imamo djelomičnu funkcionalnu ovisnost.

3. **Tranzitivna funkcionalna ovisnost (krši 3NF)**
   Ako X → Y i Y → Z, tada postoji tranzitivna ovisnost X → Z.
   Primjer:

   OIB → Poštanski broj
   Poštanski broj → Grad
   OIB → Grad (tranzitivna ovisnost, jer grad nije izravno određen OIB-om).

4. **Trivijalna funkcionalna ovisnost**
   Ovisnost je trivijalna ako desna strana već pripada lijevoj strani.
   Primjer:

   (Ime, Prezime) → Ime (trivijalno, jer je "Ime" već dio skupa na lijevoj strani).

9.6. Normalne forme u relacijskim bazama podataka

Normalne forme predstavljaju skup pravila koja se primjenjuju na dizajn relacijskih baza podataka kako bi se eliminirala redundancija i spriječile modifikacijske anomalije (umetanja, ažuriranja i brisanja). Proces primjene ovih pravila naziva se normalizacija baze podataka. Postoje različite normalne forme od 0 NF do 6 NF. 0 NF predstavlja nenormaliziranu formu gdje podaci nisu organizirani prema pravilima relacijskog modela dok se 6 NF - šesta normalna forma fokusira na temporalne baze podataka (rijetko se koristi) i nije do kraja standardizirana. Svaka sljedeća normalna forma teži boljoj organizaciji podataka kako bi se minimizirale redundancije i spriječile anomalije pri manipulaciji podacima. NF od 0 do BCNF su najčešće normalne forme (BCNF se pozicionira nakon 3NF). NF4 i NF5 često odvode normalizaciju u ekstreme dodatnim smanjivanjem anomalija.

9.6.1. Prva normalna forma (1NF)

Zahtjevi prve normalne forme:

- Svaki atribut mora sadržavati samo atomske (nedjeljive) vrijednosti.
- Eliminacija stupaca sa istim podacima, tablica ne smije imati ponavljajuće stupce, dva ili više stupca koji su usko povezani (primjerice knjiga, autor1, autor2, autor3...)
- Svaka tablica mora imati primarni ključ.
- Redoslijed redaka i stupaca ne smije utjecati na ispravnost podataka.

Tablica 70. Primjer kršenja 1NF (telefon nije atomska vrijednost)

ID	Ime	Telefonski broj
1	Ivan	098123456, 091654321
2	Ana	097789456

Rješenje kršenja 1NF je razdvajanje ponavljajućih vrijednosti u zasebne retke ili stvaranje nove tablice.

Tablica 71. Rješenje razdvajanjem ponavljajućih vrijednosti u zasebne retke što uzrokuje redundantnost podataka

ID	Ime	Telefonski broj
1	Ivan	098123456
1	Ivan	091654321
2	Ana	097789456

Tablica 72. Rješenje stvaranjem nove tablice

ID	Ime
1	Ivan
2	Ana

ID_osobe	Telefonski broj
1	098123456
1	091654321
2	097789456

9.6.2. Druga normalna forma (2NF)

Zahtjevi druge normalne forme:

- Mora zadovoljiti 1NF.
- Svaki neključni atribut mora biti **potpuno funkcionalno ovisan** o cijelom primarnom ključu, a ne samo o njegovom dijelu.

Tablica 73. Primjer kršenja 2NF (tablica koja sadrži složeni primarni ključ)

Student_ID	Predmet_ID	Ime studenta	Naziv predmeta	Profesor
1	101	Ivan	Baze podataka	Prof. A
1	102	Ivan	Algoritmi	Prof. B
2	101	Ana	Baze podataka	Prof. A

Ovdje atributi Ime studenta i Naziv predmeta ovise samo o dijelu primarnog ključa (Student_ID ili Predmet_ID), a ne o cijelom ključu.

Tablica 74. Rješenje – razdvajanje tablica

Tablica Student

Student_ID	Ime studenta
1	Ivan
2	Ana

Tablica Predmet

Predmet_ID	Naziv predmeta	Profesor
101	Baze podataka	Prof. A
102	Algoritmi	Prof. B

Tablica Student_Predmet (relacija)

Student_ID	Predmet_ID
1	101
1	102
2	101

9.6.3. Treća normalna forma (3NF)

Zahtjevi treće normalne forme:

- Mora zadovoljiti 2NF.
- Ne smije postojati **tranzitivna ovisnost**, tj. neključni atributi ne smiju biti ovisni o drugim neključnim atributima, već samo o primarnom ključu.

Tablica 75. Primjer kršenja 3NF

ID	Ime	Grad	Poštanski broj
1	Ivan	Zagreb	10000
2	Ana	Split	21000

Ovdje atribut Poštanski broj ovisi o Gradu, a ne izravno o primarnom ključu.

Tablica 76. Rješenje – razdvajanje tablica

Tablica Osoba

ID	Ime	Grad_ID
1	Ivan	1
2	Ana	2

Tablica Grad

Grad_ID	Grad	Poštanski broj
1	Zagreb	10000
2	Split	21000

Najčešće se baze dizajniraju prema 3NF jer pruža optimalan balans između normalizacije i performansi.

9.6.4. Boyce-Codd normalna forma (BCNF)

Zahtjevi:

- Mora zadovoljiti 3NF.
- Svaki funkcionalni ovisan atribut mora biti ovisan o superključu, a ne samo o primarnom ključu.
- BCNF je stroža verzija 3NF i primjenjuje se kada postoji više kandidacijskih ključeva s ovisnostima koje narušavaju pravila.
- Prepoznajemo često da nismo u BCNF kada imamo složene ključeve (nije nužno).
- Postoji više kandidatnih ključeva.
- Neki atributi imaju zajedničke ključeve.

BCNF se primjenjuje kada postoji funkcionalna ovisnost u kojoj neključni atribut određuje dio primarnog ključa. Drugim riječima, BCNF osigurava da svaka funkcionalna ovisnost bude određena samo superključem.

Pretpostavimo da imamo tablicu s podacima o dodijeljenim učionicama za određene kolegije i profesore:

Tablica 77. Primjer tablice koja nije u BCNF

Profesor	Predmet	Učionica
Prof. A	Baze podataka	101
Prof. B	Algoritmi	102
Prof. A	Strukture podataka	101
Prof. C	Mreže	103

Analiza funkcionalnih ovisnosti:

- (Profesor, Predmet) → Učionica (Primarni ključ: Profesor + Predmet);
- Učionica → Profesor (Svaka učionica je dodijeljena samo jednom profesoru);
- Problem je što Učionica → Profesor predstavlja funkcionalnu ovisnost neključnog atributa (Učionica) o nekom drugom neključnom atributu (Profesor), što narušava BCNF.

Rješenje je podijeliti tablicu na dvije kako bismo uklonili nepravilnu funkcionalnu ovisnost.

Tablica 78. Rješenje – razdvajanje tablica

Predmeti i učionice

Predmet	Učionica
Baze podataka	101
Algoritmi	102
Strukture podataka	101
Mreže	103

Profesori i učionice

Učionica	Profesor
101	Prof. A
102	Prof. B
103	Prof. C

Uklonili smo funkcionalnu ovisnost Učionica → Profesor iz originalne tablice. Svaka funkcionalna ovisnost sada polazi od superključa. Više ne postoji funkcionalna ovisnost u kojoj neključni atribut određuje ključni atribut. Ovo pokazuje ispravno normaliziran dizajn koji zadovoljava BCNF i uklanja anomalije koje su mogle nastati u prethodnoj verziji tablice.

9.6.5. Daljnje normalizacije (4NF i 5NF)

Četvrta normalna forma (4NF) uklanja višezavisnosti, što znači da ne smiju postojati višestruke neovisne veze između primarnog ključa i više atributa.

Peta normalna forma (5NF) osigurava da ne postoje podaci koji se mogu rekonstruirati samo spajanjem više tablica.

10. EKOSUSTAV BAZA PODATAKA

Ekosustav baza podataka predstavlja cjelokupno okruženje koje obuhvaća tehnologije, alate, metode i prakse usmjerene na učinkovito upravljanje podacima. On uključuje različite tipove baza podataka, poput relacijskih i NoSQL sustava, kao i ključne koncepte poput replikacije, shardinga, sigurnosti i skalabilnosti. Osim baza podataka, ekosustav obuhvaća i dodatne komponente poput tehnologija pohrane, zaštite podataka te distribuciju podataka u centraliziranim ili distribuiranim sustavima. Zajedno, ove komponente omogućuju prilagodbu baza modernim izazovima i tehnologijama.

10.1. Skalabilnost

Skalabilnost baze podataka odnosi se na sposobnost sustava da učinkovito upravlja rastućim količinama podataka i povećanim zahtjevima za performansama. To je ključno obilježje modernih baza, osobito u kontekstu velikih i dinamičnih sustava koji moraju prilagoditi svoje kapacitete u stvarnom vremenu.

Postoje dva glavna pristupa skalabilnosti:

1. **Vertikalna skalabilnost:**
   Kod ovog pristupa povećava se kapacitet pojedinačnog servera dodavanjem resursa poput memorije, procesorske snage ili prostora za pohranu. Jednostavnija je implementacija uz ograničenja maksimuma hardverskih resursa.
2. **Horizontalna skalabilnost:**
   Ovaj pristup podrazumijeva dodavanje više servera ili čvorova u sustav kako bi se teret ravnomjerno raspodijelio. Horizontalna skalabilnost omogućuje bazama da obrađuju velike količine podataka i istovremeno podržavaju velik broj korisnika.

Slika 25. Skalabilnost baze podataka

Tehnike koje podržavaju skalabilnost:

1. **Replikacija (engl. *Replication*):**
   Stvaranje kopija baze podataka na više servera kako bi se poboljšala dostupnost i otpornost na kvarove.
   Replikacija može biti:
   - **Sinkrona** – kod upisivanja na glavni server podaci se automatski prenose i na ostale servere te se potvrda o upisu vraća tek nakon što su svi čvorovi ažurirani. Ovaj pristup osigurava dosljednost podataka, ali može usporiti performanse zbog čekanja potvrda sa svih replika.
   - **Asinkrona** – glavni server ne čeka potvrdu da su podaci zapisani na replikama, što omogućuje brže upisivanje podataka, ali postoji rizik kašnjenja u ažuriranju i potencijalnog gubitka podataka u slučaju kvara glavnog servera.

   Replikacija se često koristi za poboljšanje skalabilnosti čitanja, omogućujući da se zahtjevi za dohvaćanje podataka raspodijele na više servera, čime se smanjuje opterećenje glavnog čvora i poboljšava odziv sustava. Također doprinosi visokoj dostupnosti (engl. *High Availability*) i omogućuje oporavak u slučaju kvara glavnog servera.

   Replikacija (engl. *Replication*) i sigurnosna kopija (engl. *Backup*) dvije su različite tehnike zaštite podataka koje služe različitim svrhama. Replikacija podrazumijeva kontinuirano sinkroniziranje podataka između više servera kako bi se osigurala visoka dostupnost i otpornost sustava na kvarove. U slučaju da jedan server prestane raditi, drugi može preuzeti njegovu ulogu bez gubitka podataka. S druge strane, sigurnosna kopija (*backup*) odnosi se na periodično spremanje podataka na sigurnu lokaciju, neovisnu o glavnom sustavu, kako bi se omogućio oporavak u slučaju gubitka podataka, oštećenja baze ili napada poput *ransomwarea*. Dok replikacija pruža brzu redundanciju, ona ne štiti od logičkih pogrešaka (npr. slučajnog brisanja podataka), jer se promjene odmah propagiraju na sve replike, dok *backup* omogućuje vraćanje podataka iz prethodnih snimki, ali nije uvijek ažuran u stvarnom vremenu. Zbog mogućnosti gubitka podataka uslijed kvarova, ljudskih pogrešaka ili zlonamjernih napada, svaka baza podataka mora imati sigurnosnu kopiju (*backup*) kao ključnu zaštitnu mjeru, jer replikacija sama po sebi ne može spriječiti trajni gubitak podataka u slučaju neželjenih izmjena ili korupcije.

2. **Sharding**
   Razdvajanje podataka na manje dijelove (shardove) koji se distribuiraju među različitim čvorovima, smanjujući opterećenje na jednom serveru.
3. **Keširanje (engl. *Caching*):**
   Privremeno pohranjivanje često korištenih podataka u brzu memoriju (npr. RAM) kako bi se smanjilo opterećenje baze podataka i poboljšale performanse sustava.

Skalabilnost omogućuje bazama da se prilagode rastućim potrebama aplikacija, čineći ih ključnim dijelom modernih distribucijskih sustava i tehnologija poput oblaka (*Cloud*). Dobar dizajn skalabilnog sustava omogućuje bolje performanse, veću pouzdanost i manju latenciju, čak i u uvjetima intenzivnog rasta korisnika ili podataka.

10.1.1. Horizontalna skalabilnost u relacijskim bazama podataka

Horizontalna skalabilnost (engl. *Horizontal Scaling*) u relacijskim bazama podataka suočava se s izazovima zbog njihove stroge konzistentnosti i složenih transakcijskih pravila. Problemi do kojih dolazi kod horizontalne skalabilnosti u relacijskim bazama podataka:

1. **Sharding i kompleksni upiti**
   Relacijske baze koriste ACID (engl. *Atomicity, Consistency, Isolation, Durability*) principe, što otežava podjelu podataka na više čvorova. Upiti koji zahtijevaju agregacije i spajanja (JOIN) preko distribuiranih podataka smanjuju učinkovitost jer sustav mora dohvaćati i kombinirati podatke iz više izvora, što povećava latenciju.
2. **Distribuirane transakcije i konzistentnost**
   Stroga konzistentnost zahtijeva usklađenost svih čvorova, što povećava latenciju. Dvoslojni *commit* (engl. *Two-Phase Commit, 2PC*) osigurava konzistentnost, ali uvodi značajna kašnjenja.
3. **Referencijalni integritet i povezani podaci**
   Održavanje vanjskih ključeva preko više čvorova otežava skaliranje, jer provjera integriteta zahtijeva dodatne upite i sinhronizaciju podataka.
4. **Složenost upravljanja sustavom**
   Skaliranje relacijskih baza često zahtijeva ručno upravljanje shardingom i replikacijom, dok automatsko skaliranje (poput onog u NoSQL bazama) nije standardna mogućnost, što povećava troškove održavanja.

Zbog ovih izazova, relacijske baze podataka se češće skaliraju vertikalno (dodavanjem više resursa serveru). Kada je horizontalna skalabilnost nužna, često se koriste hibridna rješenja poput replikacije, particioniranja i *cache* sustava (npr. Redis, Memcached). Alternativno, neke aplikacije prelaze na NoSQL baze (engl. NoSQL Databases) koje su dizajnirane s naglaskom na horizontalnu skalabilnost.

10.2. Centralizirane i distribuirane baze podataka

Baze podataka mogu se organizirati na dva osnovna načina: centralizirano i distribuirano, ovisno o tome gdje se podaci pohranjuju i kako se njima upravlja.

Centralizirane baze podataka nalaze se na jednom glavnom serveru ili u jednom podatkovnom centru, gdje svi korisnici pristupaju istom sustavu. Ovakav model omogućuje jednostavno upravljanje podacima, bolju kontrolu sigurnosti i konzistentnosti, ali može predstavljati usko grlo kada raste broj korisnika i opterećenje sustava. Također, kvar glavnog servera može dovesti do nedostupnosti cijele baze.

Distribuirane baze podataka (engl. *Distributed Databases*) pohranjuju podatke na više servera, često smještenih na različitim lokacijama. Ovakav pristup poboljšava skalabilnost, otpornost na kvarove i omogućuje brži pristup podacima, posebno u globalno rasprostranjenim sustavima. Međutim, distribuirane baze su složenije za održavanje jer zahtijevaju mehanizme za sinkronizaciju podataka, osiguranje konzistentnosti i upravljanje mrežnim komunikacijama.

Glavna razlika između ova dva modela leži u načinu pohrane i upravljanja podacima – dok centralizirane baze nude veću jednostavnost i kontrolu, distribuirane baze pružaju bolju otpornost i prilagodljivost velikim sustavima s globalnom distribucijom korisnika.

10.3. Sigurnost baza podataka

Sigurnost baza podataka (engl. *Database Security*) ključan je aspekt zaštite informacija pohranjenih u sustavima baza podataka. U digitalnom dobu, curenje podataka i hakerski napadi predstavljaju ozbiljnu prijetnju, posebno za velike organizacije koje upravljaju osjetljivim korisničkim informacijama. Glavni izazovi u sigurnosti baza podataka odnose se na sprječavanje neovlaštenog pristupa, zaštitu od malicioznih napada te osiguranje integriteta, povjerljivosti i dostupnosti podataka.

Za učinkovitu zaštitu baza podataka potrebno je primijeniti više sigurnosnih mjera koje pokrivaju različite aspekte sigurnosti:

1. **Kontrola pristupa i upravljanje korisnicima**
   Korisnici baze podataka trebaju imati ograničen pristup prema načelu najmanjih privilegija (engl. *Principle of Least Privilege*), što znači da svaka osoba ili sustav ima samo ona prava koja su nužna za obavljanje njihovih zadataka. Autentifikacija i autorizacija ključni su mehanizmi zaštite, a dodatno osiguranje može se postići implementacijom dvofaktorske autentifikacije (engl. *Two-Factor Authentication – 2FA*).
2. **Šifriranje podataka (engl. *Encryption*)**
   Osjetljivi podaci trebaju biti šifrirani kako bi bili nečitljivi čak i ako napadač dobije pristup bazi. Šifriranje podataka u mirovanju (engl. *Encryption at Rest*) štiti podatke dok su pohranjeni, dok šifriranje podataka u prijenosu (engl. *Encryption in Transit*) osigurava zaštitu tijekom komunikacije između klijenata i servera. Lozinke se uobičajeno pohranjuju u kriptiranom obliku korištenjem algoritama za hashiranje (engl. *Hashing*) poput bcrypt, Argon2 ili PBKDF2, čime se onemogućava njihovo jednostavno vraćanje u izvorni oblik. Osim lozinki, osjetljivi podaci poput email adresa ili brojeva kreditnih kartica mogu se šifrirati na razini stupaca baze podataka. Primjerice, PostgreSQL pruža ekstenziju pgcrypto, koja omogućuje šifriranje stupaca u bazama podataka kako bi se dodatno zaštitili podaci korisnika.
3. **Praćenje aktivnosti i revizija sigurnosti**
   Redovito analiziranje logova baze podataka omogućuje prepoznavanje sumnjivih obrazaca ponašanja i potencijalnih prijetnji. Sigurnosni auditi i penetracijsko testiranje (engl. *Penetration Testing*) pomažu u identifikaciji ranjivosti koje bi napadači mogli iskoristiti.
4. **Upravljanje sigurnosnim zakrpama (engl. *Patch Management*)**
   Napadači često iskorištavaju ranjivosti u zastarjelim verzijama baza podataka. Redovito ažuriranje baze i aplikacija ključno je za zatvaranje sigurnosnih propusta i sprječavanje napada koji koriste poznate ranjivosti.
5. **Zaštita od napada i sigurnost podataka**
   Baze podataka često su meta napada poput SQL injekcija (engl. *SQL Injection*), gdje napadač iskorištava ranjivosti kako bi izvršio neovlaštene upite. Zaštita od ovakvih napada uključuje pravilnu validaciju ulaznih podataka, korištenje pripremljenih

upita (engl. *Prepared Statements*) te implementaciju sigurnosnih politika na razini aplikacije.

6. **Sigurnosne kopije i plan oporavka od katastrofe**
   Svaka baza podataka mora imati redovite sigurnosne kopije (*backup*) kako bi se omogućio oporavak u slučaju gubitka podataka, oštećenja baze ili cyber napada. Osim toga, organizacije bi trebale imati plan oporavka od katastrofe (engl. *Disaster Recovery Plan – DRP*) koji definira korake za vraćanje sustava u funkciju uz minimalan gubitak podataka i vremena.
7. **Zaštita od DoS i DDoS napada**
   Napadi usmjereni na preopterećenje baze podataka mogu uzrokovati prekide u radu i smanjenje dostupnosti sustava. Mjere poput rate *limiting*-a, *firewall* pravila i distribuiranih sustava za zaštitu od napada mogu pomoći u obrani od ovih prijetnji.

Sigurnost baza podataka zahtijeva višeslojni pristup koji uključuje kontrolu pristupa, šifriranje podataka, praćenje aktivnosti, zaštitu od napada te strategije za oporavak od katastrofa. Sve navedene sigurnosne mjere obično provodi administrator baze podataka (engl. Database Administrator – DBA), koji je odgovoran za pravilnu konfiguraciju pristupa, održavanje sustava i provođenje sigurnosnih politika. Pravilnim planiranjem i implementacijom sigurnosnih mehanizama moguće je minimizirati rizike i osigurati da podaci ostanu povjerljivi, cjeloviti i dostupni samo ovlaštenim korisnicima.

10.3.1. SQL injekcija (engl. SQL *Injection*)

SQL injekcija jedna je od najčešće korištenih metoda napada na baze podataka. Napadači koriste ovaj pristup kako bi manipulirali SQL upitima aplikacije, često s ciljem pristupa osjetljivim podacima, izmjene baze ili čak njezinog uništenja. SQL injekcija i dalje ostaje jedna od najčešćih ranjivosti u web aplikacijama.

Glavni razlog popularnosti ovog napada je nepravilna validacija korisničkog unosa. Ukoliko aplikacija omogućuje korisnicima da unesu podatke bez adekvatne provjere i sanitizacije, napadač može umetnuti zlonamjerni SQL kod i promijeniti način na koji baza podataka izvršava upite.

Primjeri SQL injekcije:

BRISANJE TABLICE

```
$username = $_GET['user'];
$query = "SELECT * FROM korisnici WHERE username =
'$username'";
```

- *Primjer nepravilno napisanog upita u PHP-u*

```
' OR 1=1; DROP TABLE korisnici; --
```

- *napadačev unos u polje za unos korisničkog imena*

```
SELECT * FROM korisnici WHERE username = '' OR 1=1; DROP
TABLE korisnici; --'
```

- *Način izvršavanja naredbe*
- *OR 1=1 uvijek vraća true, što znači da će se dohvatiti svi korisnici.*
- *DROP TABLE korisnici; briše cijelu tablicu korisnici.*
- *-- komentira ostatak upita kako bi izbjegao sintaktičke pogreške.*
- *Ako aplikacija nema zaštitu od SQL injekcije, tablica korisnici može biti trajno izbrisana.*

ZAOBILAŽENJE AUTENTIFIKACIJE

```
' OR '1'='1' --
```

- *Unos u polje za username*

```
SELECT * FROM korisnici WHERE username = '' OR '1'='1' --'
AND password = '';
```

- *Budući da 1=1 uvijek vraća true, napadač se može prijaviti bez valjanih vjerodajnica.*
- *Lozinka se ignorira na kraju s --*

EKSTRAKCIJA PODATAKA POMOĆU UNION NAPADA

```
SELECT ime, email FROM korisnici WHERE username =
'$username';
```

- *primjer upita u aplikaciji*

```
' UNION SELECT ime, lozinka FROM korisnici; --
```

- *napadačev unos*

```
SELECT ime, email FROM korisnici WHERE username = '' UNION
SELECT ime, lozinka FROM korisnici; --
```

- *izmijenjeni unos*
- *Time napadač može dohvatiti korisnička imena i lozinke.*

Korištenjem pripremljenih upita, validacije unosa i ograničavanja korisničkih privilegija, moguće je značajno smanjiti rizik od SQL injekcije.

10.4. Usporedba relacijskih i NoSQL baza podataka

Relacijske baze podataka (RDBMS) i NoSQL baze podataka predstavljaju dva različita pristupa upravljanju podacima. Relacijske baze koriste strukturirani model temeljen na tablicama s definiranim odnosima, dok NoSQL baze nude fleksibilniji model pohrane podataka, često temeljen na dokumentima, ključ-vrijednost strukturama, stupčanim bazama ili grafovima.

1. **Struktura podataka i način pohrane**

   Relacijske baze podataka

   - Podaci se organiziraju u tablicama s definiranim stupcima i redovima.
   - Normalizacija podataka osigurava eliminaciju redundancije i održavanje konzistentnosti.
   - Koriste SQL za definiranje i manipulaciju podacima.
   - Struktura baze mora biti unaprijed definirana kroz sheme koje određuju tipove podataka i odnose između tablica.

   NoSQL baze podataka

   - Umjesto tablica koriste fleksibilne strukture, poput dokumenata (MongoDB), ključ-vrijednost parova (Redis), stupčane pohrane (Cassandra) ili grafova (Neo4j).
   - Podaci često nisu strogo normalizirani, čime se omogućava brža obrada određenih tipova upita.
   - Shema nije strogo definirana, što omogućava jednostavnije dodavanje novih atributa bez značajne reorganizacije podataka.
   - Različiti NoSQL sustavi koriste vlastite jezike umjesto standardnog SQL-a.

2. **Skalabilnost i performanse**

   Relacijske baze podataka

   - Vertikalna skalabilnost – tradicionalno su dizajnirane za rad na jednom serveru, pri čemu se skaliranje postiže dodavanjem resursa (CPU, RAM, disk).
   - Indeksi i optimizacija upita omogućuju učinkovito izvršavanje složenih upita.
   - ACID pravila osiguravaju visoku razinu konzistentnosti podataka, što je ključno za transakcijske sustave (bankarstvo, ERP).

   NoSQL baze podataka

   - Horizontalna skalabilnost – omogućuje jednostavno raspoređivanje podataka na više čvorova, čime se postiže otpornost na opterećenje i veći kapacitet pohrane.
   - Brže dohvaćanje podataka kada su upiti jednostavni (npr. pristup korisničkim podacima iz jednog JSON dokumenta).
   - Eventualna konzistentnost – u mnogim NoSQL sustavima podaci nisu odmah dosljedni na svim čvorovima, što omogućava brže pisanje i čitanje, ali može dovesti do privremenih neslaganja u podacima.

3. **Fleksibilnost i održavanje**

   Relacijske baze podataka

   - Promjene u shemi su složenije i često zahtijevaju vrijeme kada je baza nedostupna (engl. *downtime*).
   - Stroga pravila osiguravaju visoku pouzdanost, ali smanjuju fleksibilnost kod aplikacija s brzo promjenjivim podacima.

   NoSQL baze podataka

   - Fleksibilnija shema omogućava dodavanje novih polja bez potrebe za mijenjanjem cijele strukture baze.
   - Lakše upravljanje podacima u aplikacijama gdje se često mijenja struktura podataka.

Tablica 79. Preporučena primjena različitih tipova baza podataka

Tip sustava	Preporučena baza podataka	Obrazloženje
Bankarski sustavi	Relacijska (PostgreSQL, MySQL)	ACID svojstva osiguravaju dosljednost podataka.
Društvene mreže	NoSQL (MongoDB, Neo4j)	Brzo dohvaćanje korisničkih podataka i preporuka.
E-trgovina	Relacijska ili kombinacija (PostgreSQL + Redis)	Relacijska baza za transakcije, Redis za keširanje podataka.
Blogovi i CMS sustavi	Relacijska (MySQL, PostgreSQL)	Struktura podataka dobro odgovara normaliziranom modelu.
IoT i Big Data aplikacije	NoSQL (Cassandra, MongoDB)	Efikasno skaliranje i obrada velikih količina podataka.

Odabir između relacijske i NoSQL baze ovisi o specifičnim zahtjevima projekta. Relacijske baze podataka pružaju dosljednost i strogu strukturu, što ih čini idealnim za aplikacije koje zahtijevaju visoku razinu integriteta podataka. S druge strane, NoSQL baze nude fleksibilnost i bolje skaliranje za velike, distribuirane sustave. U mnogim modernim aplikacijama koristi se hibridni pristup, gdje se kombiniraju prednosti oba modela.

10.4.1. Veliki podaci (engl. *Big Data*)

Pojam veliki podaci (engl. *Big Data*) odnosi se na ogromne količine podataka koje se generiraju, prikupljaju i analiziraju u digitalnom svijetu. Ovi podaci dolaze iz različitih izvora, uključujući društvene mreže, senzore, poslovne transakcije, zdravstvene zapise, IoT (engl. *Internet of Things*) uređaje i druge digitalne sustave.

Veliki podaci karakteriziraju se kroz tri glavna svojstva (tzv. 3V model):

- Volumen (engl. *Volume*) – ogromne količine podataka koje se prikupljaju svakodnevno.
- Brzina (engl. *Velocity*) – podaci se generiraju i obrađuju u stvarnom vremenu ili gotovo u stvarnom vremenu.
- Raznolikost (engl. *Variety*) – podaci dolaze u različitim oblicima (strukturirani, polustrukturirani i nestrukturirani podaci).

Uz ova tri osnovna svojstva, ponekad se dodaju još dva:

- Istinitost (engl. *Veracity*) – pouzdanost i kvaliteta podataka.
- Vrijednost (engl. *Value*) – korisnost podataka za donošenje odluka.

Veliki podaci koriste se u raznim industrijama za poboljšanje poslovnih procesa, donošenje odluka i optimizaciju resursa. Primjeri primjene:

- Zdravstvo – analiza medicinskih podataka za predviđanje bolesti i poboljšanje liječenja pacijenata.
- Financije – otkrivanje prijevara u bankarstvu i optimizacija investicijskih strategija.
- Marketing – personalizirane reklame temeljene na ponašanju korisnika na internetu.
- IoT– obrada podataka sa senzora u pametnim gradovima i industriji.
- Društvene mreže – analiza korisničkog ponašanja na platformama poput Facebooka i Twittera.

Veliki podaci postavljaju izazove za tradicionalne relacijske baze podataka koje su dizajnirane za strukturirane podatke i ne skaliraju se lako na više servera. NoSQL baze podataka rješavaju ove probleme i bolje su za velike podatke zbog mogućnosti horizontalnog skaliranja, prirodno su distribuirane, imaju fleksibilne sheme podataka, a podaci mogu biti strukturirani, polustrukturirani i nestrukturirani, što omogućuje veću brzinu upisa i čitanja. Veliki podaci zahtijevaju baze podataka koje mogu brzo pohranjivati, analizirati i dohvaćati podatke na distribuiran način.

10.4.2. Strojno učenje (engl. *Machine Learning*)

Veliki podaci i NoSQL baze podataka igraju ključnu ulogu u strojnom učenju (engl. *Machine Learning – ML*) jer omogućuju prikupljanje, obradu i pohranu podataka potrebnih za treniranje modela strojnog učenja. Strojno učenje se oslanja na analizu velikih količina podataka kako bi otkrilo obrasce i donosilo predikcije, a NoSQL baze omogućuju učinkovito upravljanje tim podacima.

Kako bi se trenirali modeli strojnog učenja, potrebni su veliki i raznoliki skupovi podataka, što tradicionalne relacijske baze često ne mogu učinkovito obraditi. NoSQL baze podataka nude sljedeće prednosti koje ih čine idealnim za primjenu u strojnom učenju:

1. **Pohrana velikih količina nestrukturiranih podataka**
   Većina podataka korištenih u strojnom učenju su nestrukturirani ili polustrukturirani (tekst, slike, audio, video, senzorski podaci), a NoSQL baze poput MongoDB-a omogućuju njihovu fleksibilnu pohranu bez unaprijed definirane sheme.
2. **Brza obrada podataka za treniranje modela**
   Strojno učenje zahtijeva brzi pristup velikim skupovima podataka, a NoSQL baze su optimizirane za paralelnu obradu i brzo dohvaćanje podataka.
3. **Skalabilnost za velike skupove podataka**
   Veliki modeli strojnog učenja rade s terabajtima ili petabajtima podataka, a NoSQL baze omogućuju horizontalnu skalabilnost, što znači da se novi podaci mogu lako dodavati bez smanjenja performansi.
4. **Integracija s alatima za strojno učenje**
   NoSQL baze podataka podržavaju integraciju s alatima za strojno učenje poput TensorFlow-a, PyTorcha i Apache Sparka, omogućujući analizu podataka izravno iz baze bez potrebe za njihovim premještanjem u druge sustave.

Veliki podaci, NoSQL baze podataka i strojno učenje međusobno su povezani i zajedno omogućuju naprednu analizu podataka. NoSQL baze postale su neizostavan dio modernih sustava strojnog učenja, omogućujući tvrtkama i istraživačima da efikasno koriste velike podatke za donošenje pametnijih odluka i razvoj naprednih AI sustava.

LITERATURA:

1. Beaulieu, A. (2020). Learning SQL. O'Reilly Media.
2. Douglas, K. (2020). PostgreSQL: Up and Running. O'Reilly Media.
3. DuBois, P. (2013). MySQL. Addison-Wesley Professional.
4. Elmasri, R., & Navathe, S. B. (2015). Fundamentals of Database Systems. Pearson.
5. Taylor, A. G. (2018). SQL For Dummies. For Dummies.
6. PostgreSQL dokumentacija: https://www.postgresql.org/docs/
7. MySQL dokumentacija: https://dev.mysql.com/doc/
8. W3Schools PostgreSQL Tutorial: https://www.w3schools.com/postgresql/
9. W3Schools MySQL Tutorial: https://www.w3schools.com/MySQL/