



UNIVERZITET U NIŠU
ELEKTRONSKI FAKULTET



Obrada transakcija, planovi izvršavanja transakcija, izolacija i zaključavanje kod MySQL baze podataka

Seminarski rad

Studijski program: Računarstvo i informatika

Modul: Softversko inženjerstvo

Mentor:

Doc. dr Aleksandar Stanimirović

Student:

Vuk Cvetković 1667

Niš, maj 2024.

Sadržaj

Uvod	2
Transakcije	3
Svojstva transakcije	3
InnoDB Storage Engine	4
Pisanje transakcija u MySQL-u	4
Autocommit	7
Ostali parametri	11
Naredbe koje se ne mogu poništiti (ROLLBACK)	12
Savepoint	13
Lock instance	15
Lock i Unlock Table Statements	17
Zaključavanje tabela	18
Otključavanje tabela	20
Zaključavanje tabela sa transakcijama	21
Zaključavanje tabela sa okidačima (triggers)	22
Konkurentnost	24
Problemi u konkurenciji	25
Lost updates	25
Dirty Reads	26
Non-repeatable reads	26
Phantom reads	27
Nivoi izolacije	29
Deadlocks	31
XA Transakcije	33
Zaključak	35
Literatura	36

Uvod

Baze podataka su ključni elementi savremenih informacionih sistema, omogućavajući organizacijama i aplikacijama da efikasno skladište, upravljaju i manipulišu velikim količinama podataka. U većini slučajeva, baza podataka služi kao centralni izvor podataka kojem pristupaju različiti korisnici i aplikacije, često istovremeno. Ova mogućnost zajedničkog pristupa bazi podataka od strane više korisnika stvara potencijalne izazove, posebno u pogledu integriteta i konzistentnosti podataka.

Kada više korisnika pristupa istoj bazi podataka istovremeno, može doći do konflikata ili neželjenih efekata kao što su gubitak podataka, duplikati ili nekonzistentnost u podacima. Ovi problemi mogu nastati usled konkurentnog pristupa istim podacima, što može dovesti do grešaka i nesigurnosti podataka.

Postoje različita rešenja za ove izazove. Na primer, sistemi baza podataka koriste mehanizme zaključavanja i izolacionih nivoa kako bi kontrolisali konkurentni pristup podacima. Zaključavanje omogućava kontrolu pristupa tako što sprečava druge korisnike da istovremeno menjaju iste podatke. Izolacijski nivoi definišu koliko su transakcije izolovane jedna od druge, što može pomoći u sprečavanju sukoba i održavanju konzistentnosti podataka.

Takođe, mehanizmi kao što su transakcije sa ACID svojstvima (atomicity, consistency, isolation, durability) pružaju garantovanu i pouzdanu obradu podataka, čak i u okruženjima sa visokim stepenom paralelizma pristupa podacima. Ove transakcije osiguravaju da su promene u podacima trajne i da su baze podataka konzistentne čak i u slučaju grešaka ili nepredviđenih događaja.

U suštini, iako paralelni pristup podacima može doneti izazove, moderni sistemi baza podataka imaju alate i mehanizme koji pomažu u efikasnom upravljanju tim izazovima i osiguravanju pouzdanosti i integriteta podataka. Ove prakse su ključne za stabilnost i pouzdanost aplikacija i sistema koji zavise od baza podataka.

Transakcije

Transakcija [1] predstavlja osnovnu jedinicu ili niz radnji koje se izvode u bazi podataka. Ona uključuje niz operacija koje se moraju izvršiti u bazi podataka kako bi se postigao određeni cilj. Transakcije mogu biti izvršene ručno od strane korisnika ili automatski, korišćenjem programa baze podataka.

Transakcija podrazumeva unošenje, ažuriranje ili brisanje zapisa u tabelama baze podataka. Na primer, dodavanje novog zapisa, menjanje postojećeg zapisa ili brisanje zapisa predstavlja transakciju u toj tabeli. Upravljanje transakcijama je ključni aspekt za održavanje integriteta podataka i efikasno rukovanje greškama.

Kontrola transakcija obuhvata koordinaciju više operacija u jedan jedinstveni proces, koji se može sprovesti ili kao celina (izvršava se sve u okviru jedne transakcije), ili se ništa iz transakcije ne izvršava, u slučaju greške. To znači da se sve uključene operacije moraju završiti uspešno, ili se, u suprotnom, poništavaju sve promene.

U praksi, grupe SQL upita se često kombinuju i izvršavaju zajedno kao deo jedne transakcije. Na taj način, radnje koje čine transakciju postaju povezane i međusobno zavisne, što pomaže u osiguranju konzistentnosti podataka i smanjenju rizika od grešaka. Upravljanje transakcijama je ključno za postizanje visokih performansi i pouzdanosti u sistemima upravljanja bazama podataka.

Svojstva transakcije

Transakcije u bazama podataka imaju četiri osnovna svojstva [1] koja su obično poznata pod akronimom ACID. Ova svojstva obezbeđuju integritet i pouzdanost podataka, posebno kada se transakcije izvršavaju u okruženju sa više korisnika.

- **Atomicity** znači da sve operacije unutar jedne transakcije moraju biti uspešno završene kako bi se transakcija smatrala potpunom. Ako dođe do bilo kakvog problema tokom izvršenja transakcije, ona će biti prekinuta i sve prethodne operacije u transakciji će biti poništene i vraćene u prvobitno stanje. Ova osobina je ključna za očuvanje konzistentnosti podataka u slučaju grešaka ili nepredviđenih događaja.
- **Consistency** obezbeđuje da se baza podataka nalazi u ispravnom i konzistentnom stanju pre i nakon transakcije. Ako se transakcija uspešno potvrdi, podaci se menjaju na način koji osigurava da su sva ograničenja baze podataka ispunjena. Ovo svojstvo je ključno za održavanje tačnosti i integriteta podataka u sistemu.
- **Isolation** omogućava transakcijama da se izvršavaju nezavisno jedna od druge, bez uticaja na međusobne rezultate. Ovo osigurava da se istovremene transakcije ne mešaju, što smanjuje rizik od sukoba i osigurava da svaka transakcija ima dosledne podatke.
- **Durability** znači da se rezultati potvrđenih transakcija trajno čuvaju i ostaju u bazi podataka čak i u slučaju kvara sistema. Nakon potvrde transakcije, promene postaju trajne i ne mogu se lako poništiti, što obezbeđuje stabilnost i pouzdanost sistema podataka.

Ova svojstva zajedno obezbeđuju mehanizam za upravljanje transakcijama, osiguravajući da su podaci uvek pouzdani i konzistentni uprkos složenosti sistema i potencijalnim greškama.

InnoDB Storage Engine

InnoDB [2] je jedan od najpopularnijih i najkorišćenijih mehanizama za upravljanje skladištenjem podataka (eng. “Storage Engine”) u MySQL bazama podataka. Poznat je po svojoj podršci za transakcije, pouzdanosti, efikasnosti i funkcijama za integritet podataka. Omogućava razne funkcije ključne za transakcione sisteme.

InnoDB podržava transakcije koristeći ACID svojstva. Ova svojstva osiguravaju da su sve promene u bazi podataka sigurne i dosledne. Korišćenjem zaključavanja na nivou reda, InnoDB omogućava bolju paralelizaciju i smanjuje mogućnost sukoba između različitih transakcija koje se istovremeno izvode.

Omogućava višestrukim korisnicima istovremeni pristup podacima u bazi, koristeći zaključavanje i kontrolu istovremenog pristupa, čime se osigurava konzistentnost podataka i izbegavaju greške. Takođe podržava oporavak podataka nakon pada sistema ili neispravnosti.

Redo log omogućava praćenje transakcija i obezbeđuju trajnost tako da se transakcije mogu vratiti u slučaju kvara sistema. InnoDB koristi MVCC (Multiversion Concurrency Control) za podršku izolacije transakcija, omogućavajući različitim transakcijama da vide različite verzije podataka i izvrše upite paralelno, a da pritom ne dolazi do zaključavanja celih tabela.

InnoDB pruža širok spektar funkcionalnosti za upravljanje podacima i transakcijama u MySQL bazama podataka. Njegove napredne karakteristike čine ga idealnim izborom za aplikacije koje zahtevaju visoke performanse i pouzdanost.

Pisanje transakcija u MySQL-u

Postoji određena sintaksa za pisanje transakcija u MySQL-u koja definiše kako se upravlja početkom, završetkom i poništavanjem transakcija kako bi se osigurala konzistentnost i integritet podataka. U nastavku je data slika koja pokazuje sve moguće komande za upravljanje transakcijama u MySQL-u, uključujući **START TRANSACTION**, **COMMIT**, **ROLLBACK**, i postavljanje opcije **autocommit**.

```

START TRANSACTION
    [transaction_characteristic [, transaction_characteristic] ...]

transaction_characteristic: {
    WITH CONSISTENT SNAPSHOT
    | READ WRITE
    | READ ONLY
}

BEGIN [WORK]
COMMIT [WORK] [AND [NO] CHAIN] [[NO] RELEASE]
ROLLBACK [WORK] [AND [NO] CHAIN] [[NO] RELEASE]
SET autocommit = {0 | 1}

```

Slika 1. Komande u okviru jedne transakcije [3]

Ovi izrazi pružaju kontrolu nad korišćenjem transakcija [3]:

- **START TRANSACTION** - Ova komanda započinje novu transakciju u MySQL bazi podataka. Transakcija predstavlja seriju SQL operacija koje se izvršavaju kao jedna celina. Kada se izvrši START TRANSACTION, MySQL počinje beležiti sve promene koje se dešavaju u bazi podataka tokom te transakcije. Ništa se ne potvrđuje sve dok se ne izvrši COMMIT.
- **COMMIT** – Ova komanda potvrđuje sve promene koje su napravljene tokom transakcije, čineći ih trajnim u bazi podataka. Kada se izvrši COMMIT, sve izmene koje su napravljene tokom transakcije postaju vidljive drugim korisnicima i aplikacijama koje pristupaju bazi podataka.
- **ROLLBACK** – Ova komanda poništava sve promene koje su napravljene tokom trenutne transakcije, vraćajući bazu podataka u stanje pre početka transakcije. Ovo je korisno kada je poželjno otkazati izmene zbog greške ili neželjenih rezultata.
- **SET autocommit** – Ova komanda omogućava ili onemogućava automatsko potvrđivanje za trenutnu sesiju. Kada je autocommit uključen, svaka pojedinačna SQL naredba se automatski potvrđuje kao zasebna transakcija. Ovo znači da svaka izmena postaje trajna odmah nakon što se izvrši. Kada je autocommit isključen, mora se eksplicitno koristiti COMMIT da bi se potvrdila izmena ili ROLLBACK da bi se poništila.

Svaka transakcija se skladišti u binarnom logu kao celina, prilikom potvrde (COMMIT). Transakcije koje se poništavaju (ROLLBACK) se ne beleže u log.

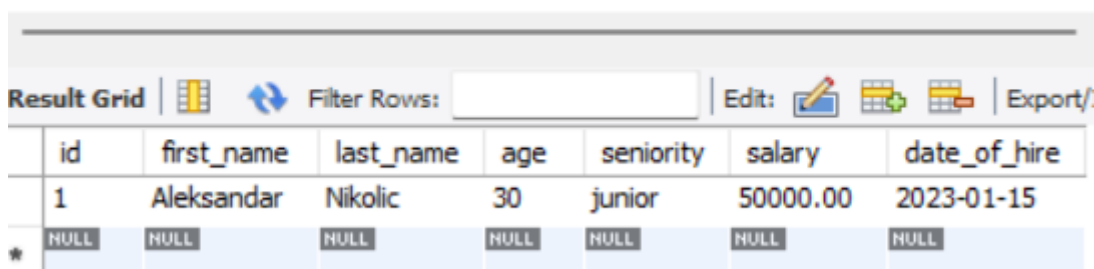
START TRANSACTION dozvoljava nekoliko modifikatora koji kontrolišu karakteristike transakcije. Da bi se navelo više modifikatora, potrebno je odvojiti ih zarezom.

- **Consistent Snapshot** je koncept koji se koristi u kontekstu transakcionih baza podataka kako bi se osiguralo da podaci čitani tokom transakcije predstavljaju dosledno stanje

baze u određenom trenutku. Ovo je posebno važno u okviru složenih sistema sa više korisnika i paralelnim transakcijama, gde je ključno da svaka transakcija vidi dosledne podatke. Kada se koristi Consistent Snapshot, baza podataka obezbeđuje kopiju podataka koja je konzistentna u smislu da predstavlja stanje podataka u određenom trenutku, često na početku transakcije. Ovaj snapshot se zatim koristi tokom trajanja transakcije za sve operacije čitanja, čime se garantuje da transakcija vidi dosledno stanje podataka, bez obzira na promene koje se dešavaju u bazi tokom izvršavanja transakcije. U kontekstu MySQL baze podataka, korišćenjem modifikatora "WITH CONSISTENT SNAPSHOT" uz START TRANSACTION, može se započeti transakcija sa konzistentnim snimcima. Ovo je posebno korisno za InnoDB Storage Engine, gde omogućava konzistentno čitanje podataka iz tabele čak i dok druge transakcije možda vrše izmene nad tim podacima. Kada se koristi Consistent Snapshot, transakcija će videti podatke u stanju koje je bilo tačno u trenutku pokretanja transakcije, bez obzira na to što se tokom trajanja transakcije događa sa podacima u bazi. Na primer, ako aplikacija želi da izvrši dugotrajnu analizu ili izveštaj nad podacima u bazi, korišćenje Consistent Snapshot može biti korisno zbog sigurnosti da analiza vidi dosledne podatke, čak i ako se tokom izvršavanja analize vrše izmene nad tim podacima od strane drugih korisnika ili aplikacija.

- **"READ WRITE"** i **"READ ONLY"** su modifikatori koji se koriste u okviru transakcija u MySQL-u kako bi se postavio režim pristupa transakciji.
 - **READ WRITE:** Ovaj modifikator označava transakciju koja ima dozvolu da čita i menja podatke u bazi podataka. To znači da transakcija može izvršavati SELECT, INSERT, UPDATE i DELETE operacije nad podacima. Kada je transakcija označena kao READ WRITE, ona može vršiti izmene nad podacima u tabelama baze podataka.
 - **READ ONLY:** Ovaj modifikator označava transakciju koja ima samo dozvolu za čitanje podataka iz baze, ali ne i za menjanje tih podataka. To znači da transakcija može samo izvršavati SELECT operacije, ali ne može izvršavati INSERT, UPDATE ili DELETE operacije. Kada je transakcija označena kao READ ONLY, ona neće moći da menja podatke u tabelama baze podataka, što je korisno kada je poželjno da se obezbedi da se podaci ne menjaju tokom izvršavanja određenih operacija.

4 • **SELECT * FROM employee;**



The screenshot shows a database interface with a 'Result Grid' tab. Above the grid is a 'Filter Rows' input field and an 'Edit' button. The grid contains one data row and one header row. The data row shows an employee with id 1, first_name Aleksandar, last_name Nikolic, age 30, seniority junior, salary 50000.00, and date_of_hire 2023-01-15. The header row shows NULL values for all columns. A star icon is visible in the bottom left corner of the grid.

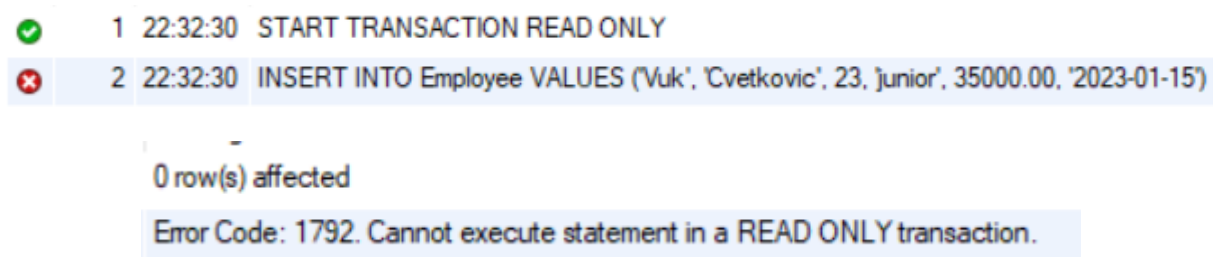
	id	first_name	last_name	age	seniority	salary	date_of_hire
	1	Aleksandar	Nikolic	30	junior	50000.00	2023-01-15
*	NULL	NULL	NULL	NULL	NULL	NULL	NULL

Slika 2. Početna baza

Ukoliko postavimo READ ONLY, i probamo da izvršimo neku izmenu u bazi, na primer da dodamo novi red, dobićemo sledeću grešku:

```
START TRANSACTION READ ONLY;
INSERT INTO Employee (first_name, last_name, age, seniority, salary, date_of_hire)
VALUES
('Vuk', 'Cvetkovic', 23, 'junior', 35000.00, '2023-01-15');
COMMIT;
```

Slika 3. READ ONLY transakcija



Slika 3. Greška pri unosu podataka u READ ONLY transakciju

MySQL omogućava dodatne optimizacije za upite na InnoDB tabelama kada je transakcija navedena kao read-only. Navođenje READ ONLY osigurava da se ove optimizacije primenjuju u slučajevima kada status read-only ne može automatski da se odredi.

Ako nije naveden način pristupa, primenjuje se podrazumevani način. Ukoliko podrazumevani način nije promenjen, on je read/write. Nije dozvoljeno navođenje i READ WRITE i READ ONLY u istoj naredbi.

U read-only modu, ostaje mogućnost menjanja tabela kreiranih sa TEMPORARY ključnom reči koristeći DML naredbe. Izmene napravljene sa DDL naredbama nisu dozvoljene, isto kao i kod trajnih tabela.

Autocommit

Da bi se videla trenutna konfiguracija autocommita, može se koristiti sledeća komanda:

```
SHOW VARIABLES like 'autocommit';
```

Variable_name	Value
autocommit	ON

Slika 4. Default vrednost autocommit-a

Možemo videti da je default autocommit postavljen na ON.

Autocommit je mod rada u MySQL-u koji automatski potvrđuje (commit) svaku pojedinačnu SQL naredbu odmah nakon njenog izvršenja. Kada je autocommit mod omogućen, svaka SQL naredba koja menja podatke (INSERT, UPDATE, DELETE) automatski se potvrđuje bez potrebe za eksplicitnim pozivanjem COMMIT naredbe.

Podrazumevano, MySQL radi sa omogućenim autocommit modom. To znači da je svaka SQL naredba koja menja podatke automatski potvrđena čim se izvrši. U ovom režimu, transakcije nisu eksplicitno definisane od strane korisnika. Svaka izmena podataka se tretira kao pojedinačna transakcija koja se automatski potvrđuje.

Primer sa isključenim autocommit-om

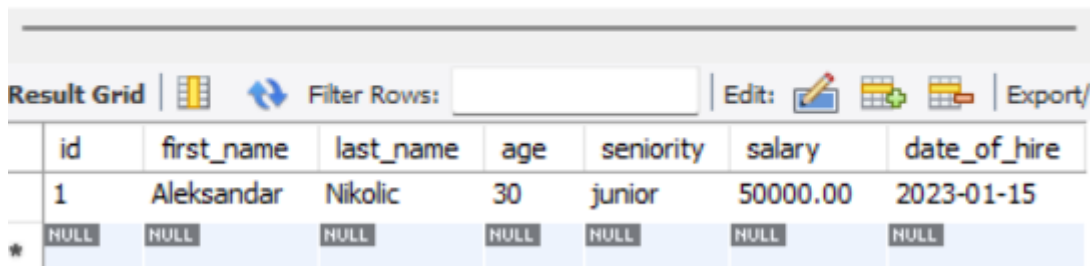
Komanda za postavljanje autocommita na 0:

```
SET autocommit = 0;
```

Slika 5. Postavljanje autocommit-a na 0

Izgled baze na početku:

4 • **SELECT * FROM employee;**



	id	first_name	last_name	age	seniority	salary	date_of_hire
	1	Aleksandar	Nikolic	30	junior	50000.00	2023-01-15
*	NULL	NULL	NULL	NULL	NULL	NULL	NULL

Slika 6. Početna baza i tabela employee

U nastavku će biti objašnjen kod koji se izvršava, zajedno sa rezultatom koji se dobija:

```

6 • SET autocommit = 0;
7 • START TRANSACTION;
8
9 • INSERT INTO Employee
10 (first_name, last_name, age, seniority, salary, date_of_hire)
11 VALUES
12 ('Vuk', 'Cvetkovic', 23, 'junior', 35000.00, '2023-01-15');
13 • SELECT * FROM employee;
14
15 • COMMIT;

```

id	first_name	last_name	age	seniority	salary	date_of_hire
1	Aleksandar	Nikolic	30	junior	50000.00	2023-01-15
2	Vuk	Cvetkovic	23	junior	35000.00	2023-01-15
*	NULL	NULL	NULL	NULL	NULL	NULL

#	Time	Action	Message
1	18:14:26	SET autocommit = 0	0 row(s) affected
2	18:14:28	START TRANSACTION	0 row(s) affected
3	18:14:30	INSERT INTO Employee (first_name, last_name, age, seniority, salary, date_of_hire) VALUES ('Vuk', 'C...	1 row(s) affected
4	18:14:48	SELECT * FROM employee LIMIT 0, 1000	2 row(s) returned

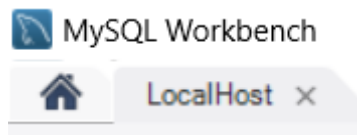
Slika 7. Transakcija bez izvršene COMMIT naredbe

U ovom SQL kodu izvršava se niz instrukcija redom kako bi se postigla željena transakcija. Prvo se postavlja autocommit mod na 0, čime se onemogućava automatsko potvrđivanje (commit) svake pojedinačne SQL naredbe. Ovo omogućava ručno upravljanje transakcijama.

Zatim se započinje nova transakcija korišćenjem START TRANSACTION naredbe. Ova naredba obeležava početak transakcije, u okviru koje se sve SQL naredbe tretiraju kao deo jedne celovite operacije.

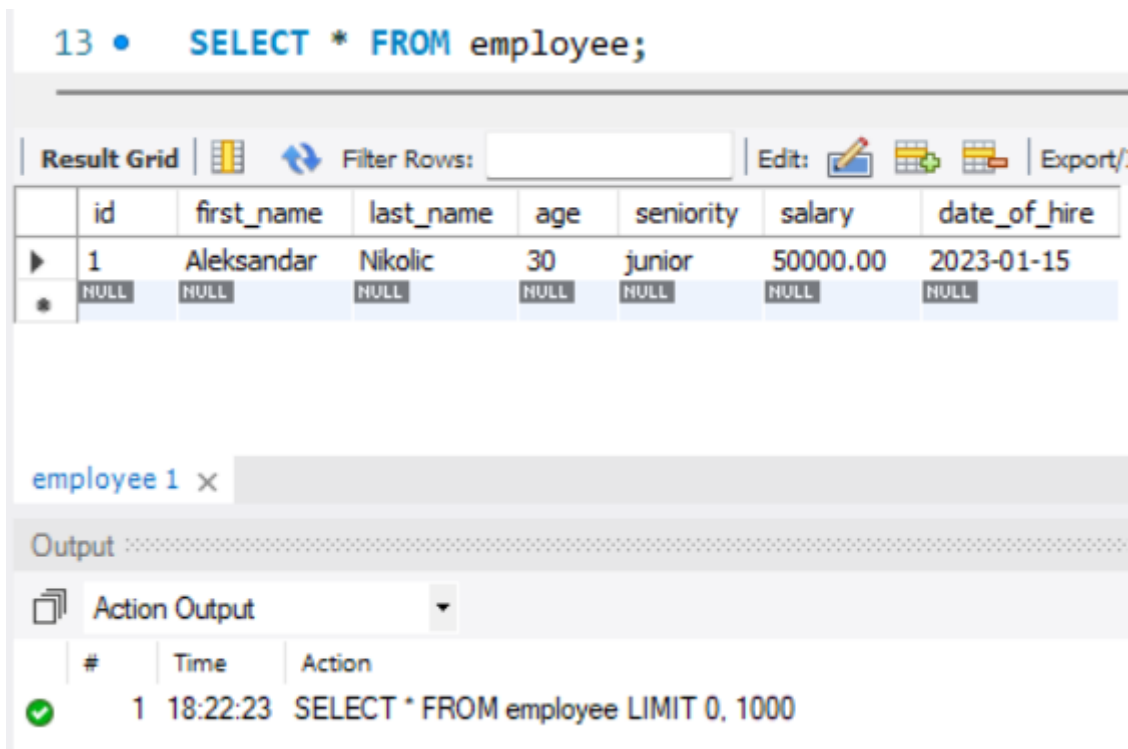
Sledeća instrukcija unosi novi red u tabelu *employee*. Nakon što se novi red uspešno doda, koristi se SELECT * FROM employee; naredba kako bi se prikazali svi zapisi iz tabele *employee*. Ovo je korisno za proveru unosa i uveravanje da su promene izvršene ispravno. Nakon onemogućavanja režima automatske potvrde postavljanjem promenljive autocommit na nulu, izmene na tabelama koje podržavaju transakcije (kao što su one za InnoDB ili NDB) ne postaju odmah trajne. Mora se koristiti COMMIT da bi se sačuvale izmene ili ROLLBACK da bi se ignorisale izmene.

U konkretnom slučaju, neće biti izvršena COMMIT naredba, nego će se simulirati greška (biće ugašena trenutna sesija, i ponovo će se upaliti). Sesiju možemo ugasiti kroz interfejs:



Slika 8. Sesija i dugme za gašenje sesije

Kada ugasimo sesiju, i ponovo pozovemo komandu za prikaz podataka, vidimo da podatak koji je unešen u okviru prethodne transakcije nije trajno sačuvan, zato što se sesija prekinula, i nije odrađen commit.



Slika 9. Izgled tabele employee nakon restartovanja sesije

Ukoliko smo želeli da ručno vratimo bazu u početno stanje, umesto COMMIT, moguće je izvršiti ROLLBACK naredbu koja će vratiti sve promene trenutne transakcije, i postaviti bazu u stanje u koje je bila na početku transakcije.

Sekvenca koda koja će da se izvrši:

```

START TRANSACTION;

SELECT * FROM employee;
INSERT INTO Employee
(first_name, last_name, age, seniority, salary, date_of_hire)
VALUES
('Vuk', 'Cvetkovic', 23, 'junior', 35000.00, '2023-01-15');
SELECT * FROM employee;

ROLLBACK;

```

Slika 10. Transakcija sa ROLLBACK naredbom

Rezultat izvršenja je sledeći:

✓	1	21:27:34	START TRANSACTION	0 row(s) affected
✓	2	21:27:34	SELECT * FROM employee LIMIT 0, 1000	1 row(s) returned
✓	3	21:27:36	INSERT INTO Employee (first_name, last_name, age, seniority, s...	1 row(s) affected
✓	4	21:27:37	SELECT * FROM employee LIMIT 0, 1000	2 row(s) returned
✓	5	21:27:39	ROLLBACK	0 row(s) affected
✓	6	21:27:39	SELECT * FROM employee LIMIT 0, 1000	1 row(s) returned

Slika 11. Rezultat prethodne transakcije

Možemo videti da na početku transakcije u bazi imamo samo jedan red. Nakon insertovanja novog reda, imamo 2 reda. Kada se odradi ROLLBACK, poništava se sve u okviru tekuće transakcije, i baza izgleda onako kako je izgledala na samom početku transakcije, odnosno neposredno pre početka transakcije (1 red u bazi).

Ostali parametri

Naredbe **BEGIN** i **BEGIN WORK** su podržane kao sinonimi za **START TRANSACTION** za pokretanje transakcije. **START TRANSACTION** je standardna SQL sintaksa i preporučeni način za započinjanje transakcije, omogućavajući modifikacije koje **BEGIN** ne podržava. U nastavku je dato objašnjenje svih parametara koji se mogu naći u okviru transakcije.

BEGIN

Ova SQL naredba označava početak nove transakcije u MySQL bazi podataka. Kada se započinje transakcija sa **BEGIN**, MySQL beleži sve izmene koje se naprave u okviru te transakcije, ali ne primenjuje ih odmah na bazu podataka. To omogućava da se izvrše više SQL naredba unutar transakcije pre nego što se odluči da li se trajno čuvaju

ili odbacuju. Naredba BEGIN se razlikuje od korišćenja ključne reči BEGIN koja započinje BEGIN ... END složenu naredbu.

BEGIN WORK Ovo je sinonim za BEGIN i takođe se koristi za započinjanje nove transakcije. Obično se koristi na isti način kao BEGIN.

WORK, CHAIN, RELEASE Ključna reč **WORK** je opcionalna za naredbe COMMIT i ROLLBACK, kao i klauzule CHAIN i RELEASE. CHAIN i RELEASE se mogu koristiti za dodatnu kontrolu nad završetkom transakcije. Vrednost sistemske varijable *completion_type* određuje podrazumevano ponašanje završetka transakcije.

AND CHAIN, RELEASE Klauzula **AND CHAIN** uzrokuje da nova transakcija počne odmah nakon završetka trenutne, pri čemu nova transakcija zadržava isti nivo izolacije kao i upravo završena transakcija. Nova transakcija takođe koristi isti način pristupa (READ WRITE ili READ ONLY) kao i upravo završena transakcija. Klauzula **RELEASE** uzrokuje da server prekine trenutnu klijentsku sesiju nakon završetka trenutne transakcije. Uključivanje ključne reči NO potiskuje završetak CHAIN ili RELEASE, što može biti korisno ako je sistemska varijabla *completion_type* podešena da podrazumevano uzrokuje završetak lanca ili prekida veze.

Naredbe koje se ne mogu poništiti (ROLLBACK)

Kada kažemo da neki izrazi ne mogu biti poništeni [4], to znači da njihove promene ne mogu biti otkazane ili vraćene na prethodno stanje. Uglavnom, ovo se odnosi na izraze jezika definicije podataka (DDL), koji su odgovorni za strukturu baze podataka. To uključuje radnje poput kreiranja ili brisanja baza podataka, tabela ili uskladištenih procedura, kao i njihovu izmenu.

Kada se radi o transakcijama, važno je izbegavati uključivanje ovih DDL izraza jer oni ne dopuštaju poništavanje promena. Na primer, ako se započne transakcija kreiranjem nove tabele i zatim se izvrši nekoliko operacija nad tom tabelom, bilo kakav neuspeh kasnije u transakciji neće omogućiti da se sve prethodne promene ponište korišćenjem standardne ROLLBACK naredbe. To znači da će ostati delimično izvršene promene koje se ne mogu jednostavno ukloniti.

Ovaj koncept je bitan jer može uticati na dizajn baza podataka i operacija nad njima. Kada se planiraju transakcije, potrebno je pažljivo razmisliti o tome koje operacije su deo transakcije i da li je moguće vratiti se na prethodno stanje u slučaju greške ili neuspeha. Izbegavanje DDL izraza u transakcijama može olakšati upravljanje promenama i održavanje konzistentnosti podataka.

Savepoint

InnoDB podržava SQL naredbe **SAVEPOINT**, **ROLLBACK TO SAVEPOINT**, **RELEASE SAVEPOINT**, kao i opcioni **WORK** ključ za **ROLLBACK**. [5]

```
SAVEPOINT identifier  
ROLLBACK [WORK] TO [SAVEPOINT] identifier  
RELEASE SAVEPOINT identifier
```

Slika 12. Komande u okviru SAVEPOINT-a

[5]

U nastavku je dato objašnjenje za svaku naredbu:

SAVEPOINT

SAVEPOINT naredba postavlja imenovanu tačku čuvanja u trenutnoj transakciji sa imenom identifikatora. Ako trenutna transakcija već ima tačku čuvanja sa istim imenom, stara tačka čuvanja se briše i postavlja se nova. Ovo omogućava korisnicima da redefinišu tačke čuvanja unutar iste transakcije.

ROLLBACK TO SAVEPOINT

ROLLBACK TO SAVEPOINT naredba vraća transakciju na imenovanu tačku čuvanja bez prekida same transakcije. Ovo znači da se transakcija ne završava, već se samo poništavaju promene napravljene nakon postavljanja tačke čuvanja. Promene koje je trenutna transakcija napravila nakon postavljanja tačke čuvanja se poništavaju.

RELEASE SAVEPOINT

RELEASE SAVEPOINT naredba uklanja imenovanu tačku čuvanja iz skupa tačaka čuvanja trenutne transakcije. Ova operacija ne dovodi do commit-a ili rollback-a transakcije. Ako tačka čuvanja ne postoji, to predstavlja grešku. Uklanjanje tačke čuvanja olakšava upravljanje memorijom i osigurava da tačke čuvanja koje više nisu potrebne ne zauzimaju resurse.

Efekat COMMIT i ROLLBACK

Sve tačke čuvanja trenutne transakcije se brišu kada se izvrši COMMIT ili ROLLBACK koji ne imenuje tačku čuvanja. Ovo znači da izvršavanje ove dve naredbe završava transakciju i oslobađa sve tačke čuvanja koje su bile postavljene tokom nje.

Ovi koncepti omogućavaju precizniju kontrolu nad transakcijama, omogućavajući da se ponište specifične promene bez uticaja na čitavu transakciju. Ovo je posebno korisno za velike i složene transakcije.

Korišćenjem SAVEPOINT naredba, administratori baza podataka mogu upravljati transakcijama na detaljnijem nivou, čime se poboljšava stabilnost i kontrola nad podacima u bazi. Ovo je kritično za održavanje integriteta podataka i omogućavanje efikasnog oporavka od grešaka unutar kompleksnih transakcija.

Primeri za ROLLBACK i RELEASE

Kod koji će biti izvršen je dat u nastavku:

```
START TRANSACTION;

SAVEPOINT sp;
SELECT * FROM employee;
INSERT INTO Employee
(first_name, last_name, age, seniority, salary, date_of_hire)
VALUES
('Vuk', 'Cvetkovic', 23, 'junior', 35000.00, '2023-01-15');
SELECT * FROM employee;
ROLLBACK TO SAVEPOINT sp1;
ROLLBACK TO SAVEPOINT sp;
SELECT * FROM employee;

RELEASE SAVEPOINT sp;
RELEASE SAVEPOINT sp;
```

Slika 13. Kod za transakciju sa SAVEPOINT i ROLLBACK to SAVEPOINT

Koraci su sledeći:

- Startovanje transakcije
- Postavljanje SAVEPOINT-a sp
- Izvršavanje SELECT naredbe (gde ćemo dobiti samo jedan red iz tabele)
- Izvršavanje INSERT naredbe (broj redova je trenutno 2)
- Izvršavanje SELECT naredbe (gde ćemo dobiti dva reda iz tabele)
- Prvi ROLLBACK će baciti grešku jer taj SAVEPOINT ne postoji
- Drugi ROLLBACK će vratiti izgled baze podataka na izgled koji je bio u trenutku definisanja SAVEPOINT-a sp
- Izvršavanje SELECT naredbe (gde ćemo dobiti samo jedan red iz tabele)
- Prvi RELEASE će ukloniti SAVEPOINT sp
- Drugi RELEASE će baciti grešku jer SAVEPOINT sp ne postoji

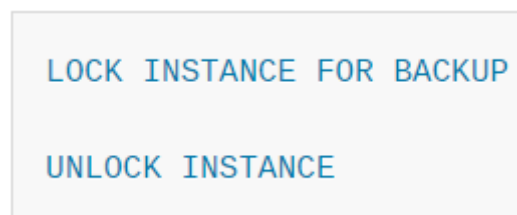
Sve ove korake takođe možemo ispratiti kroz output

✓	1	22:39:44	START TRANSACTION	0 row(s) affected
✓	2	22:39:46	SAVEPOINT sp	0 row(s) affected
✓	3	22:39:47	SELECT * FROM employee LIMIT 0, 1000	1 row(s) returned
✓	4	22:39:50	INSERT INTO Employee (first_name, last_name, age, seniority, s...	1 row(s) affected
✓	5	22:39:59	SELECT * FROM employee LIMIT 0, 1000	2 row(s) returned
✗	6	22:40:02	ROLLBACK TO SAVEPOINT sp1	Error Code: 1305. SAVEPOINT sp1 does not exist
✓	7	22:40:04	ROLLBACK TO SAVEPOINT sp	0 row(s) affected
✓	8	22:40:15	SELECT * FROM employee LIMIT 0, 1000	1 row(s) returned
✓	9	22:40:17	RELEASE SAVEPOINT sp	0 row(s) affected
✗	10	22:40:29	RELEASE SAVEPOINT sp	Error Code: 1305. SAVEPOINT sp does not exist

Slika 14. Rezultati prethodno izvršene transakcije

Lock instance

U nastavku će biti objašnjen segment koji se tiče LOCK INSTANCE FOR BACKUP i UNLOCK INSTANCE statement-a [6]



Slika 15. LOCK INSTANCE sintaksa [6]

LOCK INSTANCE FOR BACKUP je naredba koja postavlja zaključavanje na nivou instance za potrebe online bekapa, omogućavajući DML operacije (operacije manipulacije podacima) dok sprečava operacije koje bi mogle dovesti do neusklađenog snimka baze podataka. Ova naredba zahteva privilegiju BACKUP_ADMIN za izvršenje. Kada se nadograđuje na MySQL 8.0 iz starijih verzija, ova privilegija se automatski dodeljuje korisnicima sa privilegijom RELOAD.

Zaključavanje za bekap omogućava više sesija da istovremeno drže zaključavanje, što omogućava paralelno izvršavanje više bekapa. Međutim, dok je zaključavanje na snazi, operacije koje bi mogle kreirati, preimenovati ili brisati datoteke su blokirane. Ovo uključuje operacije kao što su REPAIR TABLE, TRUNCATE TABLE, OPTIMIZE TABLE, i upravljanje korisničkim nalogima. Takođe, operacije koje modifikuju InnoDB datoteke, a koje nisu zabeležene u InnoDB redo logu, su blokirane.

Uprkos zaključavanju, neke operacije su i dalje dozvoljene. Na primer, DDL operacije koje se odnose samo na korisnički kreirane privremene tabele su dozvoljene, uključujući kreiranje, preimenovanje i brisanje datoteka koje pripadaju ovim tabelama.

Kada se backup završi, naredba `UNLOCK INSTANCE` oslobađa zaključavanje koje drži trenutna sesija. Ako se sesija prekine, zaključavanje se automatski oslobađa, osiguravajući da zaključavanje ne ostane aktivno nakon završetka sesije. Ovo omogućava normalno funkcionisanje svih operacija unutar baze podataka nakon što se backup završi.

Zaključavanje instance za backup je ključno za održavanje integriteta i doslednosti podataka tokom online backupa. Omogućava administratorima baza podataka da balansiraju između potrebe za kontinuiranim pristupom podacima i sigurnog izvršavanja backupa. Na taj način, baza podataka ostaje pouzdana i dostupna, čak i u okruženjima sa visokim zahtevima za dostupnošću podataka.

Lock i Unlock Table Statements

LOCK i UNLOCK TABLE [7] su alati koji omogućavaju eksplicitno upravljanje zaključavanjem tabela u MySQL bazi podataka, ali se preporučuje oprezno korišćenje radi izbegavanja problema sa konkurentnošću i performansama sistema.

```
LOCK {TABLE | TABLES}
    tbl_name [[AS] alias] lock_type
    [, tbl_name [[AS] alias] lock_type] ...

lock_type: {
    READ [LOCAL]
    | [LOW_PRIORITY] WRITE
}

UNLOCK {TABLE | TABLES}
```

Slika 15. Sintaksa LOCK i UNLOCK TABLE naredbe [7]

MySQL omogućava sesijama klijenata da eksplicitno preuzmu zaključavanje tabela kako bi omogućile saradnju sa drugim sesijama pri pristupu tabelama ili kako bi sprečile druge sesije da modifikuju tabele tokom perioda kada jedna sesija zahteva ekskluzivni pristup. Jedna sesija može preuzeti ili otpustiti zaključavanje samo za sebe. Ne postoji mogućnost da jedna sesija preuzme zaključavanje za drugu sesiju niti da otpusti zaključavanje koja drži druga sesija.

Komanda LOCK TABLES eksplicitno preuzima zaključavanje tabela za trenutnu sesiju klijenta. Zaključavanje mogu biti preuzeta za osnovne tabele ili za poglede (views). Da bi se koristila ova komanda, korisnik mora imati privilegiju LOCK TABLES, kao i privilegiju SELECT za svaki objekat koji se zaključava.

Kada se zaključavaju pogledi, komanda LOCK TABLES automatski dodaje sve osnovne tabele koje se koriste u pogledu u skup tabela koje treba zaključati.

Ako se eksplicitno zaključa tabela pomoću LOCK TABLES, bilo koje tabele korišćene u okidačima (triggers) takođe će biti implicitno zaključane. Ako se eksplicitno zaključa tabela pomoću LOCK TABLES, bilo koje tabele povezane sa foreign key constraint biće automatski zaključane.

UNLOCK TABLES eksplicitno oslobađa bilo koje zaključavanje tabela koje drži trenutna sesija. Još jedna upotreba UNLOCK TABLES je oslobađanje globalnog zaključavanja za čitanje koje je preuzeto pomoću komande FLUSH TABLES WITH READ LOCK, koja omogućava zaključavanje svih tabela u svim bazama podataka.

LOCK TABLE je sinonim za LOCK TABLES; UNLOCK TABLE je sinonim za UNLOCK TABLES.

Zaključavanje tabele štiti samo od neprimerenih čitanja ili pisanja od strane drugih sesija. Sesija koja drži WRITE zaključavanje može izvršavati operacije na nivou tabele kao što su DROP TABLE ili TRUNCATE TABLE. Za sesije koje drže READ zaključavanje, operacije DROP TABLE i TRUNCATE TABLE nisu dozvoljene.

Zaključavanje tabela

Da bi se zaključala tabela unutar trenutne sesije, koristi se naredba LOCK TABLES. Postoje dva tipa zaključavanja [7]:

- **READ [LOCAL] Zaključavanje**
 - Sesija koja drži zaključavanje može čitati tabelu (ali ne i pisati u nju).
 - Više sesija može istovremeno steći READ zaključavanje za tabelu.
 - Druge sesije mogu čitati tabelu bez eksplicitnog sticanja READ zaključavanja.
 - Modifikator LOCAL omogućava neometano izvršavanje INSERT naredba (konkurentni unos) od strane drugih sesija dok je zaključavanje na snazi. Međutim, READ LOCAL se ne može koristiti ako će se manipulirati bazom podataka koristeći procese izvan servera dok držite zaključavanje. Za InnoDB tabele, READ LOCAL je isto što i READ.
- **[LOW_PRIORITY] WRITE Zaključavanje**
 - Sesija koja drži zaključavanje može čitati i pisati u tabelu.
 - Samo sesija koja drži zaključavanje može pristupiti tabeli. Nijedna druga sesija ne može pristupiti tabeli dok se zaključavanje ne oslobodi.
 - Zahtevi za zaključavanje tabele od strane drugih sesija se blokiraju dok je WRITE zaključavanje na snazi.
 - Modifikator LOW_PRIORITY nema efekta. U prethodnim verzijama MySQL-a, uticao je na ponašanje zaključavanja, ali to više nije slučaj. Sada je zastareo i njegova upotreba generiše upozorenje.

LOCK TABLES nareda imaju veći prioritet za WRITE zaključavanja nego za READ zaključavanja kako bi se osiguralo da se ažuriranja obrađuju što je brže moguće. To znači da ako jedna sesija dobije READ zaključavanje, a zatim druga sesija zatraži WRITE zaključavanje, kasniji READ zaključavanja će čekati dok sesija koja je zatražila WRITE zaključavanje ne dobije i ne oslobodi zaključavanje.

Ako LOCK TABLES naredba mora da čeka zbog zaključavanja koje drže druge sesije na bilo kojoj od tabela, blokiraće se dok se ne dobiju sva zaključavanja.

Sesija koja zahteva zaključavanja mora dobiti sva potrebna zaključavanja u jednoj LOCK TABLES naredbi. Dok su zaključavanja na taj način dobijena, sesija može pristupiti samo zaključanim tabelama. Na primer, u sledećem nizu naredbi, doći će do greške pri pokušaju pristupa tabeli *employee* jer nije bila zaključana u LOCK TABLES naredbi:

1	•	LOCK TABLES company READ;
2	•	SELECT * FROM company;
3	•	SELECT * FROM employee;

Output				
Action Output				
#	Time	Action	Message	
✓ 1	17:22:12	LOCK TABLES company READ	0 row(s) affected	
✓ 2	17:22:13	SELECT * FROM company LIMIT 0, 1000	1 row(s) returned	
✗ 3	17:22:14	SELECT * FROM employee LIMIT 0, 1000	Error Code: 1100. Table 'employee' was not locked with LOCK TABLES	

Slika 16. Greška pri čitanju tabele koja nije zaključana

Nije moguće više puta se pozivati na zaključanu tabelu u jednom upitu koristeći isto ime. Umesto toga, koristi se alias i dobija se posebno zaključavanje za tabelu i svaki alias:

1	•	LOCK TABLE sector_info WRITE, sector_info AS si READ;
2	•	INSERT INTO sector_info SELECT * FROM sector_info;
3		INSERT INTO sector_info SELECT * FROM sector_info AS si;

Output				
Action Output				
#	Time	Action	Message	
✓ 1	17:42:20	LOCK TABLE sector_info WRITE, sector...	0 row(s) affected	
✗ 2	17:42:22	INSERT INTO sector_info SELECT * FR...	Error Code: 1100. Table 'sector_info' was not locked with LOCK TABLES	
✓ 3	17:42:23	INSERT INTO sector_info SELECT * FR...	1 row(s) affected Records: 1 Duplicates: 0 Warnings: 0	

Slika 17. Zaključavanje preko aliasa

- Prva INSERT naredba generiše grešku zbog toga što u istom upitu postoje dve reference na isto ime zaključane tabele. Kada tabela bude zaključana, svaka referenca na nju mora koristiti isti alias ili ime kako bi bila dosledna. U ovom slučaju, obe reference u prvom INSERT upitu koriste isto ime "sector_info", što nije dozvoljeno jer MySQL zahteva da se ista tabela referencira koristeći isti alias ako je već zaključana.
- Druga INSERT naredba uspeva jer su reference na tabelu koristile različita imena, čime se izbegla konfuzija. Kada koristimo različite aliase ili imena za istu tabelu u jednom upitu, MySQL može jasno razlikovati o kojoj se referenci radi, što omogućava uspešno izvršavanje upita.

Ako naredbe referenciraju tabelu putem aliasa, mora se zaključati tabela koristeći isti alias. Neće funkcionisati zaključavanje tabele bez navođenja aliasa.

1 •	LOCK TABLE	employee	READ;
2 •	SELECT *	FROM employee AS e;	

#	Time	Action	Message
✓ 1	17:49:06	LOCK TABLE employee READ	0 row(s) affected
✗ 2	17:49:06	SELECT * FROM employee AS e LIMIT 0...	Error Code: 1100. Table 'e' was not locked with LOCK TABLES

Slika 18. Greška zbog nenavođenja aliasa u okviru LOCK TABLE naredbe

Obrnuto, ako se zaključa tabela koristeći alias, mora se referencirati na nju u naredbama koristeći taj isti alias.

1 •	LOCK TABLE	employee AS e	READ;
2 •	SELECT *	FROM employee;	
3 •	SELECT *	FROM employee AS e;	

#	Time	Action	Message
✓ 1	17:50:52	LOCK TABLE employee AS e READ	0 row(s) affected
✗ 2	17:50:52	SELECT * FROM employee LIMIT 0, 1000	Error Code: 1100. Table 'employee' was not locked with LOCK TABLES
✓ 3	17:51:00	SELECT * FROM employee AS e LIMIT 0...	1 row(s) returned

Slika 19. Greška zbog nenavođenja aliasa pri referenciranju zaključane tabele

Otključavanje tabela

Kada se zaključavanja tabela koja drži sesija oslobode, ona se sva oslobađaju istovremeno. Sesija može eksplicitno osloboditi svoja zaključavanja, ili zaključavanja mogu biti implicitno oslobođena pod određenim uslovima. [7]

- Sesija može eksplicitno osloboditi svoja zaključavanja koristeći naredbu **UNLOCK TABLES**.
- Ako sesija izda naredbu LOCK TABLES da stekne zaključavanje dok već drži zaključavanja, njena postojeća zaključavanja će biti implicitno oslobođena pre nego što se dodele nova zaključavanja.
- Ako sesija započne transakciju (na primer, sa START TRANSACTION), implicitno se izvršava UNLOCK TABLES, što uzrokuje oslobađanje postojećih zaključavanja. (Za dodatne informacije o interakciji između zaključavanja tabela i transakcija, pogledajte Interakcija Zaključavanja Tabela i Transakcija.)

Ako se konekcija za klijentsku sesiju prekine, server implicitno oslobađa sva zaključavanja tabela koja drži ta sesija (transakcionalna i netransakcionalna). Ako se korisnik ponovo poveže, zaključavanja više nisu na snazi. Pored toga, ako je korisnik imao aktivnu transakciju, server

će povući transakciju prilikom prekida veze, a ako se ponovo poveže, nova sesija počinje sa uključenim autocommit-om. Zbog ovoga, korisnici mogu želeti da onemoguće automatsko ponovno povezivanje. Sa uključenim automatskim ponovnim povezivanjem, korisnik nije obavešten ako dođe do ponovnog povezivanja, ali se sva zaključavanja tabela ili trenutna transakcija gube. Sa onemogućenim automatskim ponovnim povezivanjem, ako dođe do prekida veze, javlja se greška pri sledećoj izdatoj naredbi. Korisnik može detektovati grešku i preduzeti odgovarajuće mere, kao što je ponovno sticanje zaključavanja ili ponavljanje transakcije.

Zaključavanje tabela sa transakcijama

Naredbe **LOCK TABLES** i **UNLOCK TABLES** interaguju sa korišćenjem transakcija na sledeći način [7]:

- **LOCK TABLES** nije transakcijski siguran i implicitno potvrđuje bilo koju aktivnu transakciju pre nego što pokuša da zaključa tabele.
- **UNLOCK TABLES** implicitno potvrđuje bilo koju aktivnu transakciju, ali samo ako je **LOCK TABLES** korišćen za sticanje zaključavanja tabela. Na primer, u sledećem skupu naredbi, **UNLOCK TABLES** oslobađa globalno read zaključavanje, ali ne potvrđuje transakciju jer nijedno zaključavanje tabela nije na snazi:

```
FLUSH TABLES WITH READ LOCK;  
START TRANSACTION;  
SELECT * FROM employee;  
UNLOCK TABLES;
```

Slika 20. UNLOCK TABLES sa transakcijom

- Pokretanje transakcije (na primer, sa **START TRANSACTION**) implicitno potvrđuje bilo koju trenutnu transakciju i oslobađa postojeća zaključavanja tabela.
- **FLUSH TABLES WITH READ LOCK** stiče globalno read zaključavanje, a ne zaključavanja tabela, tako da nije podložno istom ponašanju kao **LOCK TABLES** i **UNLOCK TABLES** u pogledu zaključavanja tabela i implicitnih potvrda. Na primer, **START TRANSACTION** ne oslobađa globalno read zaključavanje.
- Druge naredbe koje implicitno uzrokuju potvrđivanje transakcija ne oslobađaju postojeća zaključavanja tabela.
- Ispravan način korišćenja **LOCK TABLES** i **UNLOCK TABLES** sa transakcionim tabelama, kao što su InnoDB tabele, je da se započne transakcija sa **SET autocommit = 0** (ne sa **START TRANSACTION**) nakon čega sledi **LOCK TABLES**, i da se ne pozivaju **UNLOCK TABLES** dok se ne potvrdi transakcija eksplicitno. Na primer, ako potreban upis u tabelu *employee* i čitate iz tabele *company*, može se to uraditi ovako:

```
SET autocommit=0;
LOCK TABLES employee WRITE, company READ;
#bizni logika
COMMIT;
UNLOCK TABLES;
```

Slika 21. Pravilan način za pisanje transakcija za zaključavanjem

ROLLBACK ne oslobađa zaključavanja tabela.

Zaključavanje tabela sa okidačima (triggers)

Ako se tabela eksplicitno zaključa pomoću naredbe LOCK TABLES, bilo koje tabele korišćene u okidačima takođe se implicitno zaključavaju [7]:

- Zaključavanja se vrše istovremeno kao i ona koja su eksplicitno stečena naredbom LOCK TABLES.
- Zaključavanje tabele korišćene u okidaču zavisi od toga da li se tabela koristi samo za čitanje. Ako je tako, dovoljno je zaključavanje za čitanje. U suprotnom, koristi se zaključavanje za pisanje.
- Ako je tabela eksplicitno zaključana za čitanje pomoću LOCK TABLES, ali treba da bude zaključana za pisanje jer može biti izmenjena unutar okidača, koristi se zaključavanje za pisanje umesto zaključavanja za čitanje. (To jest, implicitno zaključavanje za pisanje potrebno zbog pojavljivanja tabele u okidaču dovodi do toga da se eksplicitan zahtev za zaključavanje za čitanje tabele konvertuje u zahtev za zaključavanje za pisanje.)

Pretpostavimo da je potrebno zaključati dve tabele, *employee* i *employee_department*, korišćenjem ove naredbe:

```
LOCK TABLES employee WRITE, employee_department READ;
```

Slika 22. Naredba za zaključavanje tabela employee i employee_department

Ako *employee* ili *employee_department* imaju bilo kakve okidače, tabele korišćene unutar okidača takođe su zaključane. Pretpostavimo da *employee* ima definisan okidač na sledeći način:

```

CREATE TRIGGER employee_after_insert
AFTER INSERT ON employee
FOR EACH ROW
BEGIN
    UPDATE company
    SET location = 'Niš'
    WHERE id = 1 AND EXISTS (SELECT * FROM sector_info);
    INSERT INTO employee_department VALUES (1, 1);
END;

```

Slika 23. Definisanje trigger-a za tabelu employee

Rezultat LOCK TABLES naredbe je da su *employee* i *employee_department* zaključani jer se pojavljuju u naredbi, a *company* i *sector_info* su zaključani jer se koriste unutar okidača:

- *employee* je zaključan za pisanje zbog zahteva za WRITE zaključavanjem.
- *employee_department* je zaključan za pisanje, iako je zahtev za READ zaključavanjem. Do toga dolazi jer se *employee_department* ubacuje unutar okidača, pa se READ zahtev pretvara u WRITE zahtev.
- *sector_info* je zaključan za čitanje jer se samo čita unutar okidača.
- *company* je zaključan za pisanje jer bi mogao biti ažuriran unutar okidača.

Konkurentnost

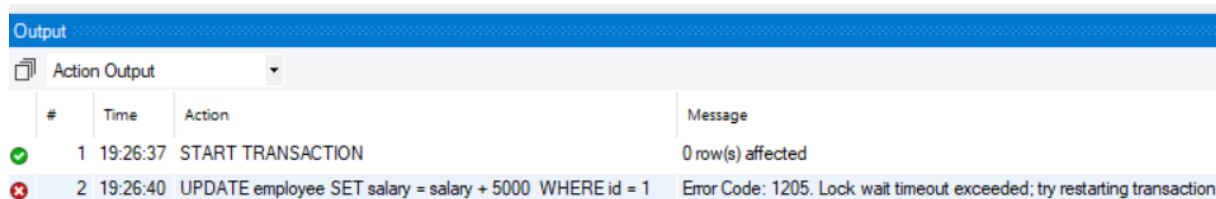
U realnom svetu, postoji više korisnika koji istovremeno pristupaju bazi podataka ili aplikaciji. Konkurentnost može postati problem kada jedan korisnik menja podatke, dok drugi korisnik istovremeno pokušava da pročita ili promeni iste te podatke. Ova situacija može dovesti do nedoslednosti podataka, sukoba u transakcijama i drugih problema vezanih za integritet podataka.

Zamislimo primer da postoje dva korisnika, koji žele da UPDATE-uju isti red u tabeli, i promene neku vrednost. Kada izvršimo prvi UPDATE na određeni red, MySQL stavlja zaključavanje (lock) na taj red koji se ažurira. Ovo zaključavanje sprečava druge transakcije da izvrše promene na istom redu dok je prva transakcija još uvek aktivna. Na taj način se osigurava da samo jedna transakcija u isto vreme može da menja određeni red, čime se sprečavaju sukobi i osigurava doslednost podataka.

Ako druga transakcija pokuša da ažurira isti red, moraće da sačeka dok se prva transakcija ne završi, bilo commit-om ili rollback-om. Ukoliko se prva transakcija ne završi u određenom vremenskom periodu, druga transakcija će se prekinuti zbog timeout-a. Na ovaj način, MySQL osigurava da se transakcije izvode serijski kada pristupaju istim redovima, čime se smanjuje mogućnost konflikata.

Konkretan primer isteka transakcije, zbog zaključavanja reda koji se UPDATE-uje:

```
1  START TRANSACTION;
2  •  UPDATE employee SET salary = salary + 5000
3  WHERE id = 1;
4  •  COMMIT;
```



#	Time	Action	Message
1	19:26:37	START TRANSACTION	0 row(s) affected
2	19:26:40	UPDATE employee SET salary = salary + 5000 WHERE id = 1	Error Code: 1205. Lock wait timeout exceeded; try restarting transaction

Slika 24. Timeout transakcije

Sa podrazumevanim ponašanjem zaključavanja u MySQL-u, uglavnom ne treba da se brinemo o konkurentnosti jer je sistem dizajniran da automatski rukuje većinom situacija. Međutim, postoje specijalni slučajevi kada podrazumevano ponašanje nije dovoljno da reši sve probleme. U tim situacijama, potrebno je dodatno konfigurisati zaključavanja i upravljanje transakcijama kako bi se osigurala ispravnost i doslednost podataka.

Na primer, kada je potrebno raditi sa veoma kompleksnim scenarijima konkurentnosti ili kada specifične aplikacije zahtevaju finiju kontrolu nad zaključavanjima, MySQL pruža napredne mehanizme poput različitih nivoa izolacije transakcija (READ UNCOMMITTED, READ COMMITTED, REPEATABLE READ, SERIALIZABLE) i specijalnih zaključavanja (LOCK TABLES, LOCK INSTANCE FOR BACKUP). Korišćenjem ovih mehanizama, možemo

preciznije kontrolisati kako i kada se podaci zaključavaju, čime se obezbeđuje efikasnije i pouzdanije upravljanje konkurentnim pristupom bazi podataka.

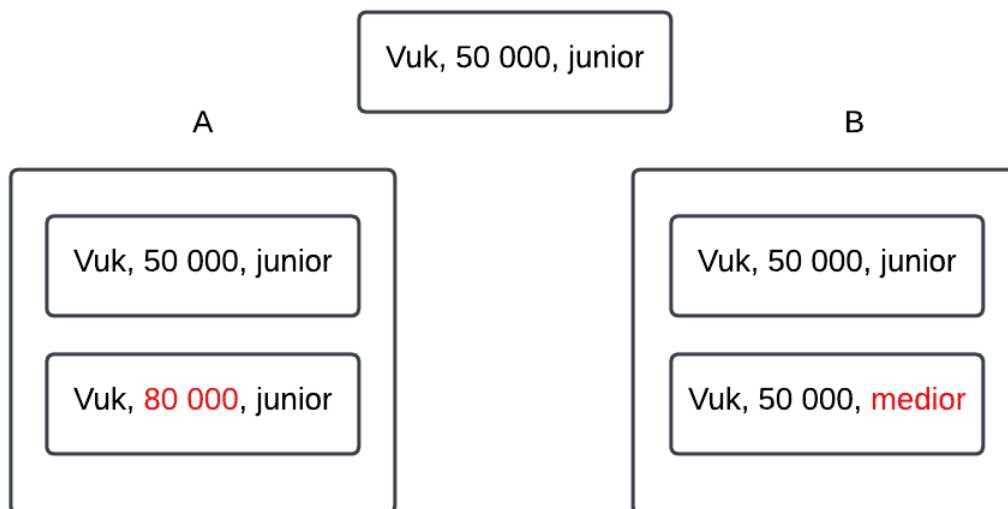
Problemi u konkurenciji

U nastavku će biti objašnjeni problemi koji se javljaju zbog konkurentnosti.

Lost updates

Jedan od glavnih problema je **Lost Updates** [8]. Ova situacija se dešava kada dve transakcije pokušavaju da ažuriraju isti podatak, ali se ne koristi lock. U tom slučaju, transakcija koja se izvrši poslednja će override-ovati promene koje je napravila prva transakcija.

Na primer, imamo dve transakcije koje žele da promene podatke o istom zaposlenom. Transakcija A želi da promeni platu, dok transakcija B želi da promeni senioritet. Zaposleni Vuk ima platu 50.000 i senioritet junior. Transakcija A želi da poveća platu, a transakcija B želi da promeni senioritet. Obe transakcije počinju sa istim početnim podacima. Ako transakcija A promeni platu, ali ne izvrši commit, i transakcija B promeni senioritet, ali takođe ne izvrši commit, transakcija koja se poslednja izvrši će override-ovati promene prve transakcije.

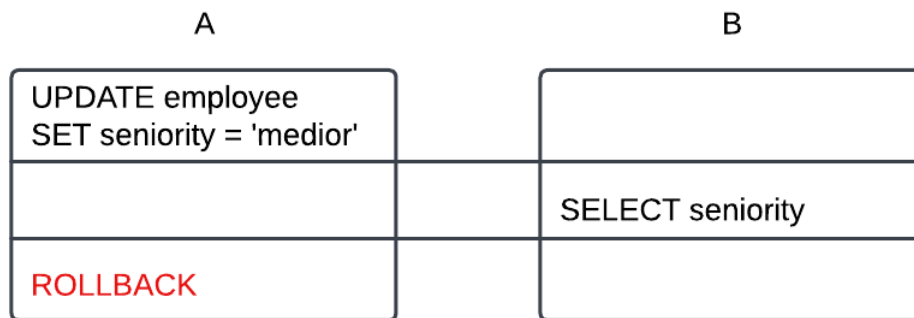


Slika 25. Primer za Lost Updates

Kako se ovo rešava? Korišćenjem lock mehanizma. Jedna transakcija mora biti završena pre nego što druga može da počne. Ne mogu se obe izvršavati istovremeno.

Dirty Reads

Dirty reads [8] se dešava kada transakcija čita podatke koji još nisu commitovani. Na primer, transakcija A promeni senioritet zaposlenog na medior (koji je inicijalno bio junior), ali ne izvrši commit, ostajući u transakciji. U tom trenutku, transakcija B pročitaj taj senioritet i na osnovu njega računa platu. Ako transakcija A kasnije izvrši rollback, podaci koje je pročitala transakcija B zapravo ne postoje, što znači da će izračunata plata biti netačna.

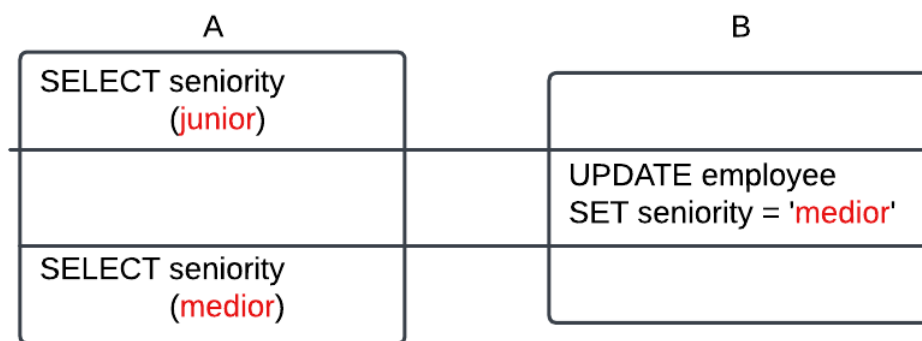


Slika 26. Primer za Dirty Reads

Da bi se ovaj problem rešio, mora se primeniti neki nivo izolacije između transakcija. Podaci koje je promenila transakcija A ne smeju biti dostupni ostalim transakcijama dok se te promene ne commituju. Ovo se postiže postavljanjem nivoa izolacije na Read Committed, o čemu će biti više reči kasnije. Kada se koristi ovaj nivo izolacije, transakcije mogu pročitati samo podatke koji su commitovani, čime se sprečava problem Dirty Reads.

Non-repeatable reads

Non-repeatable reads [8] se dešava kada u toku transakcije, isti podatak pročitat dva puta daje različite rezultate. Na primer, transakcija A pročitaj senioritet određenog zaposlenog i dobije vrednost "junior". Na osnovu ovog podatka, transakcija A donosi odluku, recimo određivanje plate. Pre nego što se transakcija A završi, transakcija B ažurira senioritet tog zaposlenog na "medior". Kada transakcija A ponovo pročitaj senioritet, dobija "medior". Dakle, za dva uzastopna čitanja iste kolone dobijamo različite podatke. Ovo se naziva **Non-repeatable Read** ili **Inconsistent Read**.



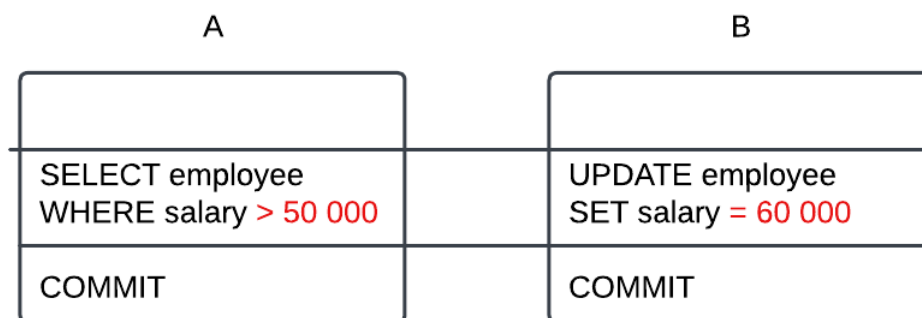
Slika 27. Primer za Non-repeatable Reads

Kako se ovo rešava? Potrebno je da odluke unutar transakcije budu zasnovane na konzistentnim podacima koji su važili na početku transakcije. Promene koje se dogode tokom trajanja transakcije ne smeju uticati na već donete odluke. Ovo se postiže povećanjem nivoa izolacije na **repeatable read**. Na ovom nivou izolacije, uzastopna čitanja istog podataka unutar jedne transakcije daju konzistentne rezultate, čak i ako se podaci u međuvremenu promene. Transakcija vidi snapshot podataka kako su bili na početku njenog izvršenja, čime se obezbeđuje konzistentnost čitanja.

Phantom reads

Phantom reads [8] se dešavaju kada jedna transakcija selektuje skup redova koji zadovoljavaju određeni uslov, a druga transakcija u međuvremenu doda ili izmeni redove koji bi zadovoljili taj uslov, ali nisu bili uključeni u prvobitni rezultat prve transakcije. Na primer, transakcija A selektuje sve zaposlene koji imaju platu veću od 50.000 radi neke poslovne logike. U isto vreme, transakcija B ažurira platu zaposlenog koji je imao manje od 50.000, čime on postaje kvalifikovan za selekciju transakcije A, ali se ne pojavljuje u njenom rezultatu. Kada se transakcije završe, i dalje postoji zaposleni koji ispunjava uslov transakcije A, ali nije bio uključen u njen rezultat. Ovo se naziva Phantom Read.

"Phantom" označava da se podatak pojavljuje kao duh - ima sve kvalifikacije da bude u rezultatu, ali se ne nalazi tamo jer je ažuriran posle izvršenja upita iz transakcije A.



Slika 28. Primer za Phantom Reads

Kako se ovo rešava? Zavisi od poslovnog problema koji se rešava i od toga koliko je bitno da taj zaposleni bude uključen u rezultat. Možemo ponovo izvršiti transakciju A kasnije i dobiti tog zaposlenog u listi. Međutim, ako je obavezno da se u bilo kom slučaju svi kvalifikovani podaci nađu u rezultatu, ne smeju se pokretati transakcije koje mogu menjati podatke koje čita neka druga transakcija.

Za ovo se koristi najviši nivo izolacije, Serializable. On garantuje da se transakcije koje menjaju podatke koje čita neka druga transakcija ne mogu pokretati dok se ta transakcija ne završi. Transakcije moraju čekati i izvršavaju se sekvencijalno. Ovaj nivo izolacije je vrlo efikasan u sprečavanju Phantom Read problema, ali ima svoju cenu. Što je više korisnika i transakcija, sistem se sve više usporava. Ovaj nivo izolacije negativno utiče na performanse i skalabilnost, zbog čega se koristi samo kada je apsolutno neophodno sprečiti Phantom Reads.

Nivoi izolacije

U nastavku je data tabela koja pokazuje koje probleme resave koji nivo izolacije [8]:

	Lost Updates	Dirty Reads	Non-repeating Reads	Phantom Reads
READ UNCOMMITTED				
READ COMMITTED		✓		
REPEATABLE READ	✓	✓	✓	
SERIALIZABLE	✓	✓	✓	✓

Slika 29. Tabela koja koji nivo izolacije rešava koji problem [8]

Ova slika prikazuje različite nivoe izolacije u MySQL-u i probleme sa konzistencijom podataka koje svaki nivo može da reši. Nivoi izolacije određuju koliko transakcije mogu biti izolovane jedna od druge, odnosno koliko jedna transakcija može da vidi promene koje su napravile druge transakcije pre nego što su te promene završene (commit-ovane).

Najniži nivo izolacije je "**READ UNCOMMITTED**", gde transakcije mogu da čitaju podatke koje druge transakcije još nisu završile. Ovaj nivo izolacije može prouzrokovati brojne probleme, kao što su dirty reads (čitanje podataka koji nisu završeni), non-repeatable reads (gde ponovljeno čitanje istog reda tokom transakcije daje različite rezultate), phantom reads (gde se prilikom ponovljenog čitanja pojave novi redovi koji nisu bili prisutni na početku transakcije), i lost updates (gde se promene napravljene u jednoj transakciji izgube zbog promena u drugoj). Međutim, "READ UNCOMMITTED" je najbrži nivo izolacije jer ne koristi nikakve lockove (zaključavanja).

Viši nivo izolacije, "**READ COMMITTED**", omogućava čitanje samo završenih transakcija. Ovo rešava problem dirty reads, ali ne rešava non-repeatable reads, phantom reads niti lost updates. Ovaj nivo izolacije je nešto sporiji od "READ UNCOMMITTED" jer koristi zaključavanja.

Nivo izolacije "**REPEATABLE READ**" obezbeđuje da transakcija uvek vidi iste podatke pri svakom čitanju tokom svog trajanja, čime rešava problem lost updates, dirty reads i non-repeatable reads. Međutim, phantom reads se mogu i dalje pojaviti. Ovaj nivo izolacije koristi više zaključavanja i sporiji je u poređenju sa "READ COMMITTED".

Najstrožiji nivo izolacije je "**SERIALIZABLE**". Ovaj nivo izolacije tretira transakcije kao da se izvode jedna po jedna, serijalizovano, što eliminiše sve prethodno pomenute probleme sa

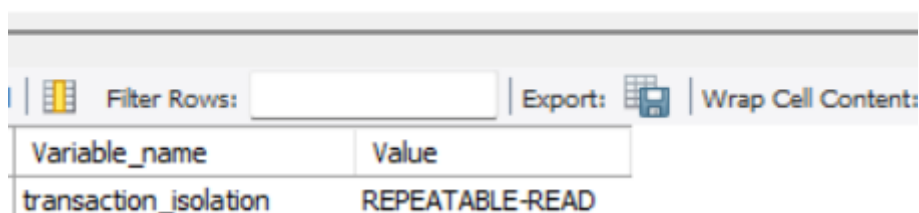
konzistencijom podataka, uključujući dirty reads, non-repeatable reads, phantom reads i lost updates. Međutim, "SERIALIZABLE" je najsporiji nivo izolacije jer koristi najviše zaključavanja i značajno usporava sistem zbog visoke izolacije između transakcija.

Ukratko, kako se povećava nivo izolacije, tako se smanjuje verovatnoća problema sa konzistencijom podataka, ali to može značajno uticati na performanse baze podataka. "READ UNCOMMITTED" je najbrži jer ne koristi zaključavanja, dok je "SERIALIZABLE" najsporiji zbog visoke izolacije.

U MySQL, podrazumevani nivo izolacije je Repeatable Reads. Većina slučajeva je dobro pokrivena. Ovaj nivo izolacije radi efikasno u mnogim situacijama. Brži je od Serializable nivoa izolacije i pruža prevenciju za mnoge probleme konkurentnosti, osim za phantom reads. Sve dok nije eksplicitno zahtevano da se spreče phantom reads, Repeatable Reads će obaviti posao.

Komanda kojom se proverava trenutni nivo izolacije:

```
SHOW VARIABLES LIKE 'transaction_isolation';
```



The screenshot shows a MySQL query result window. At the top, there is a toolbar with icons for 'Filter Rows', 'Export', and 'Wrap Cell Content'. Below the toolbar is a table with two columns: 'Variable_name' and 'Value'. The table contains one row with the values 'transaction_isolation' and 'REPEATABLE-READ'.

Variable_name	Value
transaction_isolation	REPEATABLE-READ

Slika 30. Komanda za proveru trenutnog nivoa izolacije

Ukoliko ništa nije menjano, ovo je i default nivo izolacije.

U nastavku je data komanda kojom se menja nivo izolacije:

```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;  
SET SESSION TRANSACTION ISOLATION LEVEL SERIALIZABLE;  
SET GLOBAL TRANSACTION ISOLATION LEVEL SERIALIZABLE;
```

Slika 31. Komanda za postavljanje nivoa izolacije

Nivo izolacije se može postaviti na 3 različita načina:

- Prva komanda postavlja nivo izolacije na serializable samo za narednu transakciju
- Druga komanda postavlja nivo izolacije na serializable za trenutno otvorenu sesiju
- Treća komanda postavlja nivo izolacije na serializable globalno, za sve sesije

Deadlocks

Deadlocks [8] je jedan od klasičnih problema u bazama podataka. Deadlock se dešava kada se različite transakcije ne mogu izvršiti zato što svaka transakcija drži lock na nekom podatku koji je potreban nekoj drugoj transakciji. Obe transakcije čekaju jedna drugu i nikada ne otpustaju svoje lockove.

Konkretan primer:

Sledeća slika pokazuje inicijalnu bazu, koja ima dve tabele, i svaka tabela po jedan red.

	id	first_name	last_name	age	seniority	salary	date_of_hire
▶	1	Aleksandar	Nikolic	30	junior	50000.00	2023-01-15
•	NULL	NULL	NULL	NULL	NULL	NULL	NULL

Slika 32. Employee tabela

	id	name	industry	location	founded_date	employee_count
▶	1	Kompanija	Informacione Tehnologije	Leskovac	2000-05-20	500
•	NULL	NULL	NULL	NULL	NULL	NULL

Slika 33. Company tabela

Imamo dve sesije, prva sesija izvršava kod sa sledeće slike:

```
START TRANSACTION;  
UPDATE company SET location = 'Niš' WHERE id = 1;  
UPDATE employee SET salary = 80000 WHERE id = 1;  
COMMIT;
```

Slika 34. Kod prve sesije

Druga sesija izvršava kod sa sledeće slike:

```
START TRANSACTION;  
UPDATE employee SET salary = 80000 WHERE id = 1;  
UPDATE company SET location = 'Niš' WHERE id = 1;  
COMMIT;
```

Slika 35. Kod druge sesije

Prva i druga sesija u isto vreme startuju transakciju. Prva transakcija će ažurirati tabelu *company* i postaviti lokaciju na Niš. Druga transakcija će ažurirati tabelu *employee* i postaviti salary na 80000. Te operacije će se uspešno izvršiti. Sada kad prva transakcija proba da ažurira *employee*, upašće u deadlock, zato što druga transakcija vec ima lock na tu tabelu. Prva transakcija će

pokušavati da izvrši azuriranje *employee*, ali neće uspeti. Kada druga transakcija proba da ažurira *company*, MySQL će odrediti tu drugu transakciju kao žrtvu (victim) i ona će odraditi ROLLBACK, a prva transakcija će se izvršiti.

✓	1	00:27:17	START TRANSACTION	0 row(s) affected
✓	2	00:27:18	UPDATE employee SET salary = 80000 WHERE id = 1	1 row(s) affected Rows matched: 1 Changed: 1 Warnings: 0
✗	3	00:27:24	UPDATE company SET location = 'Niš' WHERE id = 1	Error Code: 1213. Deadlock found when trying to get lock; try restarting transaction

Slika 36. Greška koju dobija druga transakcija

Deadlock nije toliko veliki problem, sve dok se on ne dešava i suviše često. Postoje neke stvari koje se mogu odraditi da bi se smanjio deadlock. Nikad ne mogu u potpunosti da se uklone, mogu samo da se umanje. Ako često detektujemo deadlock među dve transakcije, potrebno je pogledati kod. Možda može kod da se preuredi, tako da ne dolazi do deadlocka. Potrebno je da se prati isti redosled u tim transakcijama kako ne bi došlo do deadlocka. Druga stvar je da pravimo naše transakcije tako da budu male i da se relativno kratko izvršavaju, tako da ne stupaju u konflikt sa drugim transakcijama. Ako se neke transakcije izvršavaju na ogromnim tabelama, transakcije mogu duže da se izvršavaju, što povećava mogućnost za deadlock. Moguće je odraditi te transakcije u neko doba dana kada najmanje korisnika pristupa aplikaciji.

XA Transakcije

XA transakcije [9] su mehanizam koji omogućava podršku za distribuirane transakcije u MySQL bazi podataka, posebno za InnoDB storage engine. Osnova za implementaciju XA transakcija je X/Open CAE dokument pod nazivom "Distributed Transaction Processing: The XA Specification". Putem XA transakcija, MySQL omogućava više odvojenih transakcionih resursa, kao što su različite RDBMS (Relational Database Management System), da učestvuju u globalnoj transakciji.

Na klijentskoj strani, korišćenje XA transakcija zahteva slanje SQL naredbi koje počinju sa XA ključnom rečju. Ovo omogućava MySQL klijentskim programima da obavljaju operacije u sklopu globalnih transakcija.

Ključne karakteristike XA transakcija uključuju:

- Distribuirane transakcije: Omogućava više odvojenih transakcionih resursa da budu deo jedne globalne transakcije.
- ACID osobine: Transakcije su atomične, konzistentne, izolovane i trajne, kako bi se obezbedila pouzdanost i konzistentnost podataka.
- Podrška za InnoDB: XA transakcije podržane su za InnoDB storage engine, što omogućava upotrebu ovog mehanizma u MySQL bazama podataka koje koriste InnoDB za upravljanje podacima.

Za razliku od običnih transakcija, koje se odvijaju na nivou pojedinačnog servera, XA transakcije omogućavaju višestruku koordinaciju transakcionih resursa širom mreže, čime se olakšava razvoj složenih aplikacija koje zahtevaju distribuirane transakcije.

Evo nekoliko primera distribuiranih transakcija:

- Integracija messaging servisa i RDBMS-a: Aplikacija može delovati kao integracioni alat koji kombinuje messaging servis sa relacionalnom bazom podataka. Aplikacija osigurava da se transakcije koje se bave slanjem, preuzimanjem i obradom poruka, a koje takođe uključuju transakcione baze podataka, dešavaju u okviru globalne transakcije.
- Akcije koje uključuju različite serverske baze podataka: Aplikacija izvršava akcije koje uključuju različite serverske baze podataka, kao što su MySQL server i Oracle server (ili više MySQL servera), gde akcije koje uključuju više servera moraju da se odvijaju kao deo globalne transakcije, umesto kao odvojene transakcije lokalne za svaki server.
- Bankarske transakcije kroz bankomate (ATM): Banka čuva informacije o računima u relacionalnoj bazi podataka, a novac distribuira i prima putem bankomata. Neophodno je osigurati da akcije bankomata ispravno odražavaju stanje računa, ali to se ne može uraditi samo pomoću RDBMS-a. Globalni transakcioni menadžer integriše resurse bankomata i baze podataka kako bi se osigurala celokupna konzistentnost finansijskih transakcija.

Aplikacije koje koriste globalne transakcije uključuju jedan ili više upravljača resursima (Resource Managers) i jednog upravljača transakcijama (Transaction Manager):

- Upravljač resursima (**Resource Manager** - RM) pruža pristup transakcionim resursima. Baza podataka je jedan tip upravljača resursima. Moralo bi biti moguće potvrditi ili poništiti transakcije koje upravlja RM
- Upravljač transakcijama (**Transaction Manager** - TM) koordinira transakcije koje su deo globalne transakcije. On komunicira sa RM-ovima koji obrađuju svaku od ovih transakcija. Individualne transakcije unutar globalne transakcije su "grane" globalne transakcije

MySQL-ova implementacija XA omogućava MySQL serveru da deluje kao Upravljač resursima (Resource Manager) koji rukuje XA transakcijama unutar globalne transakcije. Klijentski program koji se povezuje na MySQL server deluje kao Upravljač transakcijama (Transaction Manager).

Da bi se izvršila globalna transakcija, potrebno je znati koje komponente su uključene i dovesti svaku komponentu do tačke kada može biti potvrđena ili poništena. U zavisnosti od toga šta svaka komponenta prijavljuje o svojoj sposobnosti da uspe, sve one moraju biti potvrđene ili poništene kao atomična grupa. Drugim rečima, ili sve komponente moraju biti potvrđene, ili sve komponente moraju biti poništene. Da bi se upravljalo globalnom transakcijom, potrebno je uzeti u obzir da bilo koja komponenta ili povezana mreža mogu da otkazu.

Proces izvršenja globalne transakcije koristi dvofaznu potvrdu (2PC). Ovo se dešava nakon što su akcije izvršene od strane grana globalne transakcije.

- U prvoj fazi, sve grane se pripremaju. To znači da im TM kaže da se pripreme za potvrdu. Tipično, ovo znači da svaki RM koji upravlja granom beleži akcije za granu u stabilnom skladištu. Grane ukazuju da li su u mogućnosti da to urade, i ovi rezultati se koriste za drugu fazu.
- U drugoj fazi, TM obaveštava RM-ove da li potvrditi ili poništiti. Ako su sve grane ukazale kada su bile pripremljene da su sposobne za potvrdu, svim granama se kaže da potvrde. Ako je bilo koja grana ukazala kada je bila pripremljena da nije u mogućnosti da potvrdi, svim granama se kaže da se ponište.

U nekim slučajevima, globalna transakcija može koristiti jednofaznu potvrdu (1PC). Na primer, kada Upravljač transakcijama otkrije da globalna transakcija sastoji se samo od jednog transakcionog resursa (to jest, jedne grane), taj resurs može biti upućen da se pripremi i potvrdi istovremeno.

Zaključak

U ovom radu smo detaljno istražili koncept transakcija u MySQL bazi podataka, fokusirajući se na njihova svojstva, upotrebu, parametre i ponašanje u različitim scenarijima. Upoznali smo se sa InnoDB storage engine-om koji pruža podršku za transakcije u MySQL-u i naučili kako pisati transakcije koristeći MySQL jezik.

Autocommit funkcionalnost je istaknuta kao važan aspekt koji utiče na upravljanje transakcijama, dok su savepoints omogućili fleksibilnost u upravljanju tačkama vraćanja transakcija. Razmotrili smo i različite parametre koji utiču na ponašanje transakcija, kao i naredbe koje nije moguće poništiti pomoću ROLLBACK-a.

U radu smo takođe obradili teme vezane za zaključavanje tabela i upravljanje konkurentnošću u bazi podataka. Upoznali smo se sa problemima konkurencije kao što su Lost Updates, Dirty Reads, Non-repeatable Reads i Phantom Reads, te smo istražili nivoe izolacije transakcija kao alate za rešavanje tih problema.

Pored toga, razmotrili smo pitanja vezana za deadlock-ove i XA transakcije. Kroz analizu svih ovih aspekata, stekli smo dublje razumevanje kako MySQL baza podataka upravlja transakcijama i kako se nosi sa različitim scenarijima koji mogu nastati u realnim aplikacijama.

U zaključku, možemo reći da su transakcije ključni element u svetu baza podataka, omogućavajući konzistentnost i pouzdanost podataka. Razumevanje njihovog ponašanja i pravilna implementacija pružaju osnovu za efikasno upravljanje podacima u MySQL bazi podataka.

Literatura

- [1] TutorialsPoint, SQL – transactions,
Datum pristupa: 29.04.2024.
Dostupno na: <https://www.tutorialspoint.com/sql/sql-transactions.htm>
- [2] MySQL_{TM}, InnoDB Locking and Transaction Model,
Datum pristupa: 08.05.2024.
Dostupno na: <https://dev.mysql.com/doc/refman/8.0/en/innodb-locking-transaction-model.html>
- [3] MySQL_{TM}, START TRANSACTION, COMMIT, and ROLLBACK Statements,
Datum pristupa: 10.05.2024.
Dostupno na: <https://dev.mysql.com/doc/refman/8.0/en/commit.html>
- [4] MySQL_{TM}, Statements That Cannot Be Rolled Back
Datum pristupa: 11.05.2024.
Dostupno na: <https://dev.mysql.com/doc/refman/8.0/en/cannot-roll-back.html>
- [5] MySQL_{TM}, SAVEPOINT, ROLLBACK TO SAVEPOINT, and RELEASE
SAVEPOINT Statements
Datum pristupa: 13.05.2024.
Dostupno na: <https://dev.mysql.com/doc/refman/8.0/en/savepoint.html>
- [6] MySQL_{TM}, LOCK INSTANCE FOR BACKUP and UNLOCK INSTANCE
Statements
Datum pristupa: 13.05.2024.
Dostupno na: <https://dev.mysql.com/doc/refman/8.0/en/lock-instance-for-backup.html>
- [7] MySQL_{TM}, LOCK TABLES and UNLOCK TABLES Statements
Datum pristupa: 16.05.2024.
Dostupno na: <https://dev.mysql.com/doc/refman/8.0/en/lock-tables.html>
- [8] Mosh Hamedani, Complete SQL Mastery
Datum pristupa: 18.05.2024.
Dostupno na: <https://codewithmosh.com/p/complete-sql-mastery>
- [9] MySQL_{TM}, XA Transactions
Datum pristupa: 21.05.2024.
Dostupno na: <https://dev.mysql.com/doc/refman/8.0/en/xa.html>