SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

ZAVRŠNI RAD br. 1166

PODACI INTERNETA STVARI OZNAČENI OZNAKAMA KOJE SU ORGANIZIRANE U HIJERARHIJU

Filip Fabris

Zagreb, lipanj 2023.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

ZAVRŠNI RAD br. 1166

PODACI INTERNETA STVARI OZNAČENI OZNAKAMA KOJE SU ORGANIZIRANE U HIJERARHIJU

Filip Fabris

Zagreb, lipanj 2023.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

Zagreb, 10. ožujka 2023.

ZAVRŠNI ZADATAK br. 1166

Pristupnik:	**Filip Fabris (0036532834)**
Studij:	Elektrotehnika i informacijska tehnologija i Računarstvo
Modul:	Računarstvo
Mentor:	prof. dr. sc. Mario Kušek

Zadatak: **Podaci Interneta stvari označeni oznakama koje su organizirane u hijerarhiju**

Opis zadatka:

U današnjem Internetu značajno raste broj jednostavnih uređaja poput senzorskih i aktuatorskih čvorova koji prikupljaju i odašilju podatke s različitih senzora te primaju naredbe koje izvršavaju u fizičkom svijetu. Takvi čvorovi su obično vezani za pojedine stvari i tvore Internet stvari (Internet of Things). U laboratoriju za Internet stvari napravljena je platforma koja služi za povezivanje uređaja u Internetu stvari, spremanje njihovih podatka i kroz aplikacijsko programsko sučelje dohvaćanje podataka pomoću REST sučelja. Vaš je zadatak nadograditi podatke koji se spremaju oznakama (engl. tag) koji su organizirani u hijerarhiju. Za to trebate koristiti bazu podataka koja se temelji na grafu, povezati s postojećom relacijskom bazom podataka i kroz REST sučelje pretraživati podatke prema oznakama. Svu potrebnu literaturu i uvjete za rad osigurat će Vam Zavod za telekomunikacije.

Rok za predaju rada: 9. lipnja 2023.

Zahvaljujem se mentoru prof. dr. sc Mariju Kušeku na pomoći i vodstvu prilikom izrade završnog rada.

Sadržaj

1. Uvod

U današnjem Internetu značajno raste broj jednostavnih uređaja poput senzorskih i aktuatorskih čvorova koji prikupljaju i odašilju podatke s različitih senzora te primaju naredbe koje izvršavaju u fizičkom svijetu. Takvi čvorovi su obično vezani za pojedine stvari i tvore Internet stvari (Internet of Things).

Kroz navedeni koncept interneta stvari uređaji dosežu sve veću razinu međusobne povezanosti pa se tako predviđa da će do 2025. godine biti preko 19 milijardi povezanih uređaja[12]. Samim porastom broja povezanih IoT uređaja raste količina generiranih podataka koji su u pojedinim segmentima usko povezani.

S obzirom na veliku količinu uređaja i podataka, postavlja se pitanje kako ih organizirati u hijerarhijsku strukturu koja omogućuje lakše upravljanje i analizu podataka.

Cilj ovog rada je istražiti koju vrstu baze podataka upotrijebiti u svrhu spremanja hijerarhije te također ostvariti:

- brzu responzivnost,
- skalabilnost,
- lagano upravljanje,
- grafički prikaz hijerarhije i
- mogućnost integracije s ostalim bazama podataka.

U sklopu FER-ovog istraživačkog projekta „IoT-polje“ te tijekom studentskog projekta „Mobilno/web IoT-polje„ dizajniran je i implementiran REST poslužitelj koji u relacijsku bazu podataka sprema informacije o scenama, prikazima i ključevima za pristup podatcima baze podataka InfluxDB.

Kroz REST sučelje korisnicima je omogućeno dohvaćanje scena koje sadrže podatke o ključevima i upitu kako dohvatiti podatke s baze podataka na temelju vremenskih sljedova, u ovom slučaju InfluxDB.

Trenutno je filtriranje scena ostvareno nizanjem oznakama (engl. tag), drugim riječima nije implementirana nikakva hijerarhija u tom smislu. U svrhu studijskog primjera uz sam istraživački moment rada, zadatak je nadograditi postojeći REST poslužitelj u svrhu dizajna i ostvarivanja hijerarhije pomoću spremanja oznaka (engl. tag).

Kao što je navedeno REST poslužitelj implementiran tijekom projekata „IoT-polje“ i „Mobilno/web IoT-polje“ koristi relacijsku bazu podataka u svrhu spremanja podataka. Stoga je, još jedan važan aspekt ovog rada, postizanje rješenja koje nije samo učinkovito već se i lako može implementirati u sklopu navedenih projekata.

Cilj je postignuti hibridno rješenje hijerarhije koje ne ovisi o postojećem dizajnu i implementaciji spremanja podataka. Drugim riječima želimo uvesti rješenje hijerarhije koje se s lakoćom implementira s postojećim projektima.

2. Princip hijerarhije među podatcima

Hijerarhiju među objektima možemo pronaći u svakodnevnom životu, na primjer hijerarhija radnih mjesta, hijerarhija u kategorijama proizvoda, hijerarhija u smislu distributivnih cenatara itd.

Uvođenjem hijerarhije među podatcima ostvarujemo nekoliko pogodnosti. Prvo, omogućuju nam brže filtriranje podataka jer se mogu primijeniti filtri na različite razine hijerarhijske strukture. Na taj način možemo efikasno pristupiti samo relevantnim podatcima u okviru određene grane hijerarhije.

Drugo, hijerarhijska struktura omogućuje nam lako određivanje položaja objekta unutar strukture. Na primjer, možemo saznati tko su roditelji određenog objekta i koja su mu djeca. To nam pruža uvid u odnose između objekata te nam omogućuje definiranje pristupa podatcima na temelju njihove pozicije u hijerarhiji. Na primjer, objekt može imati pristup podatcima svojih roditelja i svoje djece, ali ne i ostalim podatcima unutar strukture.

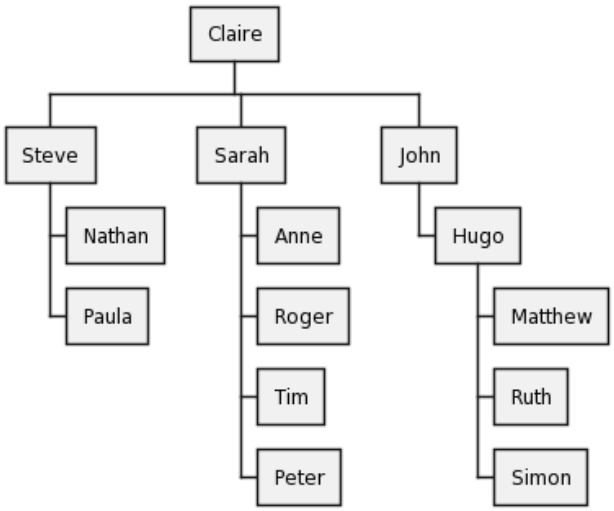
Uvođenje hijerarhije među podatcima omogućuje nam bolje organiziranje, upravljanje i analizu podataka. Omogućuje nam uspostavu jasnih odnosa između objekata, pruža nam kontekstualni uvid u njihovu povezanost i olakšava nam pristup relevantnim podatcima.

2.1. Problem hijerarhije u relacijskim bazama

U relacijskoj bazi podataka, hijerarhijski podaci mogu se modelirati tehnikama:

- model popisa susjedstva (engl. Adjacency List),
- model ugniježđenog skupa (engl. Nested Set),
- model ravne tablice (engl. Flat Table) i
- model putanje (engl. Path Enumeration).

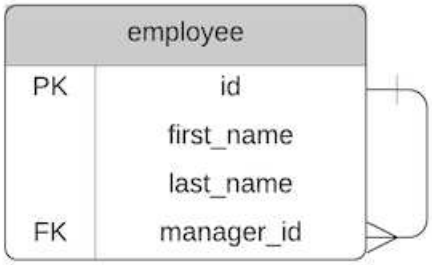
Prema primjerima iz knjige Beginning Oracle SQL for Oracle Database 18c [13] predstavit ću načine implementacije hijerarhije u relacijskoj bazi podataka. U navedenom opisu metoda koristit će se hijerarhija sa slike (Slika 2.1).



Slika 2.1 Radni primjer hijerarhije među zaposlenicima [1]

Osoba na vrhu hijerarhije je Claire, kojem su Steve, Sarah i John izravno podređeni. Također u primjeru Steve, Sarah i John također imaju svoje podređene članove.

Prvi korak je ostvariti način referenciranja odnosa roditelj – dijete. To je najlakše ostvariti modelom samoreferencirajućih tablica. Radi se o tablicama koje sadrže stupac stranog ključa koji se odnosi na tu istu tablicu. Taj strani ključ predstavlja odnos roditelj-dijete između redaka u tablici.



Slika 2.2 Samoreferencirajuća tablica radnog primjera [1]

Slika 2.2. predstavlja primer jedna takve samoreferencirajuće tablice. U ovom primjeru atribut *manager_id* predstavlja stupac stranog ključa koji pokazuje na jedan zapis koji mu je nadređen. Specifičnije zaposlenici su povezani s drugim zaposlenicima koji su im u ovom smislu nadređeni (nadalje menadžeri).

2.1.1. Model popisa susjedstva

Ovaj model naspram svih ostalih najlakše je implementirati jer izravno koristi model samoreferencirajucih tablica. Izravno koristimo stupac stranog ključa koji pokazuje na primarni ključ nadređenog stupca tablice. Primjeri takvih zapisa dan je u tablici 2.1.

id	first_name	role	manager_id
1	Claire	CEO	null
2	Steve	Head of Sales	1
3	Sarah	Head of Support	1
4	John	Head of IT	1
5	Nathan	Salesperson	2
6	Paula	Salesperson	2
7	Anne	Customer Support Officer	3
8	Roger	Customer Support Officer	3
9	Tim	Customer Support Officer	3
10	Peter	Assistant	3
11	Hugo	Team Leader	4
12	Matthew	Developer	11
13	Ruth	Developer	11
14	Simon	QA Engineer	11

Tablica 2.1 Prikaz zapisa elemenata prema modelu popisa susjedstva

Budući da je Claire na vrhu hijerarhije on nema nadređene osobe stoga je njegov atribut *manager_id null*. Također prema hijerarhiji na slici 2.1., nadređena osoba Steva je Claire stoga je *manager_id* stupca Steve 1, koji korespondira primarnom ključ od Claire.

Odabir zaposlenika i njihovih menadžera postižemo pomoću lijevog spajana na istoj tablici.

```
SELECT e.id, e.first_name, e.role, e.manager_id, m.first_name
FROM employee e
LEFT JOIN employee m ON e.manager_id = m.id;
```

id	first_name	role	manager_id	first_name
1	Claire	CEO	null	null
2	Steve	Head of Sales	1	Claire
3	Sarah	Head of Support	1	Claire
4	John	Head of IT	1	Claire
5	Nathan	Salesperson	2	Steve
6	Paula	Salesperson	2	Steve
7	Anne	Customer Support Officer	3	Sarah
8	Roger	Customer Support Officer	3	Sarah
9	Tim	Customer Support Officer	3	Sarah
10	Peter	Assistant	3	Sarah
11	Hugo	Team Leader	4	John
12	Matthew	Developer	11	Hugo
13	Ruth	Developer	11	Hugo
14	Simon	QA Engineer	11	Hugo

Tablica 2.2 Rezultat lijevog spajanja tablica

U svrhu prikaza stabla hijerarhije koristimo rekurzivnu klauzulu WITH poznatu kao i Recursive Common Table Expression (RCTE) koja je zapravo proširenje klauzule WITH u SQL-u koje nam omogućuje definiranje privremenog skupa rezultata koji se stvara izvršavanjem naredbe SELECT sve dok se ne ispuni uvjet zaustavljanja.

Drugim riječima, RCTE je CTE koji se referencira samog sebe i uključuje jednu ili više naredbi SELECT koje se referenciraju na CTE.

RCTE se sastoji od dva dijela:

- početni član
- rekurzivni član

Početni član je ne-rekurzivni dio CTE koji definira početni čvor pretrage.

Dok je rekurzivni član je dio CTE-a koji se ponavlja sve dok se ne ispuni uvjet zaustavljanja. Taj uvjet zaustavljanja definiran je pomoću anotacije WHERE unutar rekurzivnog člana CTE-a.

Sintaksa CTE-a:

```
WITH RECURSIVE <cte_name> (column1, column2, ...) AS (
    -- anchor member
    (SELECT column1, column2, ...
    FROM table_name
    WHERE condition)
    UNION ALL
    -- recursive member
    (SELECT column1, column2, ...
    FROM cte_name
    WHERE <resursive_condition>)
)
SELECT column1, column2, ...
FROM cte_name;
```

Elementi RCTE:

- *<cte_name>* predstavlja ime CTE izraza, preko tog imena ćemo kasnije referencirat CTE u upitu,
- *(column1, column2, ...)* specificira koji će stupci biti uključeni u CTE,
- dio označen s „anchor member“ predstavlja ne rekurzivni dio CTE, on definira inicijali čvor,
- dio „recursive member“ predstavlja rekurzivnog člana, taj upit se izvršava sve dok je uvijet *<resursive _condition>* zadovoljen.

Važno je primijeniti da je upravo WHERE filter unutar rekurzivnog člana element koji zaustavlja rekurziju.

Evo jednog primjera korištenja WITH klauzule sa svrhom generiranja hijerarhijskog stabla temeljenog na modelu popisa susjeda. U ovom primjeru unutar ne rekurzivnog člana (anchor node) definiramo inicijalni čvor. Ovdje se radi o zaposleniku koji ima id = 1, radi se o Claire koja je zapravo i vrh stabla naše hijerarhije. Također je važno prijetiti da unutar rekurzivnog člana nema WHERE filtera, stoga će se rekurzija obraditi sve čvorove hijerarhije stabla dok ne dođe do listova. Izvršavanjem sljedećeg upita kao rezultat dobivamo Tablicu 2.3.

```
WITH RECURSIVE wholeHierarchy AS (
  -- anchor member
  (SELECT id, first_name, role, manager_id, 1 AS level
  FROM employee
  WHERE id = 1)
  UNION ALL
 -- recursive member
  (SELECT this.id, this.first_name, this.role, this.manager_id, prior.level
       + 1  FROM empdata prior
  INNER JOIN employee this ON this.manager_id = prior.id)
)

SELECT h.id, h.first_name, h.role, h.manager_id, h.level
FROM wholeHierarchy h
ORDER BY h.level;
```

id	first_name	role	manager_id	first_name	level
1	Claire	CEO	null	Claire	1
2	Steve	Head of Sales	1	Steve	2
3	Sarah	Head of Support	1	Sarah	2
4	John	Head of IT	1	John	2
5	Nathan	Salesperson	2	Nathan	3
6	Paula	Salesperson	2	Paula	3
7	Anne	Customer Support Officer	3	Anne	3
8	Roger	Customer Support Officer	3	Roger	3
9	Tim	Customer Support Officer	3	Tim	3
10	Peter	Assistant	3	Peter	3
11	Hugo	Team Leader	4	Hugo	3
12	Matthew	Developer	11	Matthew	4
13	Ruth	Developer	11	Ruth	4
14	Simon	QA Engineer	11	Simon	4

Tablica 2.3 Izvršavanje RCTE upita za ispis cijelog stabla

Ukoliko za ne rekurzivni član stavimo na primjer čvor sa id-jem 4 dobivamo stablo hijerarhiju ispod osobe John koji ima ulogu „Head of IT“. Kod je dan u nastavku, a rezultat upita je prikazan u Tablici 2.4.

```
WITH RECURSIVE NodeHierarchy AS (
  -- anchor member
  (SELECT id, first_name, role, manager_id, 1 AS level
  FROM employee
  WHERE id = 4)
  UNION ALL
 -- recursive member
      ...
```

id	first_name	role	manager_id	first_name	level
4	John	Head of IT	1	John	2
11	Hugo	Team Leader	4	Hugo	3
12	Matthew	Developer	11	Matthew	4
13	Ruth	Developer	11	Ruth	4
14	Simon	QA Engineer	11	Simon	4

Tablica 2.4 Izvršavanje RCTE upita za ispis cijelog stabla

Dodavanje novog elementa

```
INSERT INTO employee (id, first_name, role, manager_id)
VALUES (15, 'Alex', 'Salesperson', 2);
```

Micanje elemenata u hijerarhiji

```
UPDATE employee
SET manager_id = 2
WHERE id = 10;
```

Kao što vidimo vrlo je jednostavnim upitom prebaciti element iz jednog stabla hijerarhije u drugi. U ovom slučaju osobu s ID-em 10 (Peter) prebacujemo iz odjela prodaje u odjel IT.

Brisanje elementa iz hijerarhije

```
DELETE FROM employee WHERE id = 10;
```

Ako je zaposlenik list stabla hijerarhije on se može direktno obrisati. Međutim, ako se radi o menadžeru, moramo sve zaposlenike kojima on upravlja prvo prebaciti na drugog menadžera. Naravno da želimo zabraniti mogućnost brisanja čvora koji ima elemente pod sobom, to lako ostvarimo dodavanjem ograničenjima stranog ključa, zbog toga ako korisnik pokuša napraviti tu operaciju dobit će pogrešku.

Sve u svemu prvo moramo koristiti gore napisan UPDATE upit za ažuriranje zaposlenikovog menadžera, kada navedeni zaposlenik postane list u stablu moguće ga je obrisati tj. kada više nije menadžer.

Prednosti koncepta popisa susjedstva:

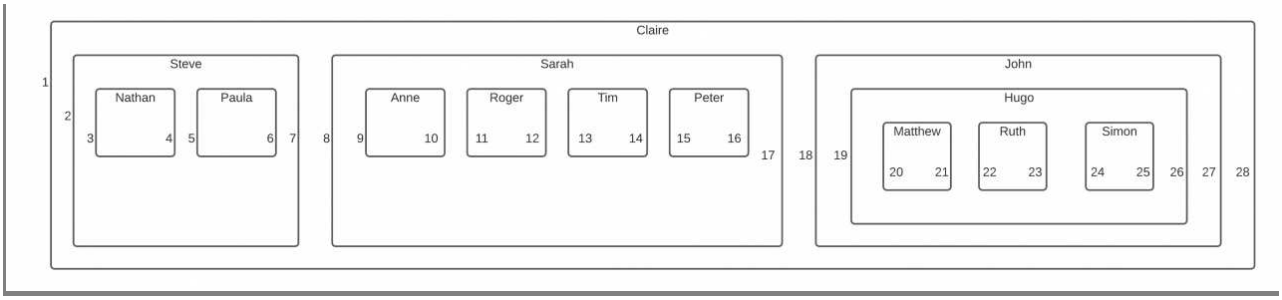
- Lagana implementacija, dodavanje novog stupca koji se odnosi na ID nadređenog zapisa u istoj tablici
- Lagano dodavanje novih elemenata
- Lako je premjestiti elemente na drugo mjesto u hijerarhiji
- Lako je ukloniti elemente.

Nedostatci koncepta popisa susjedstva:

- Upit za dohvaćanje podataka je kompliciran. U većini baza podataka trebate napisati WITH klauzulu s dva upita unutar nje.
- Upit za dohvaćanje cijelog stabla ili podskupa stabla sporo će se izvoditi s velikom količinom zapisa.
- Teško je pronaći put zapisa od korijena do trenutnog zapisa., loša vizualizacija
- Treba premjestili djecu novom roditelju kako bi se obrisao stari roditelj

2.1.2. Model ugniježdenog skupa

Model ugniježđenog skupa hijerarhijskih podataka dizajn je koji pohranjuje minimalne i maksimalne ID vrijednosti zapisa unutar njega. Model ugniježđenog skupa često se vizualizira pomoću serije kontejnera. Svaki kontejner reprezentira jednog zaposlenika, a unutar spremnika su svi zaposlenici kojima taj zaposlenik upravlja.



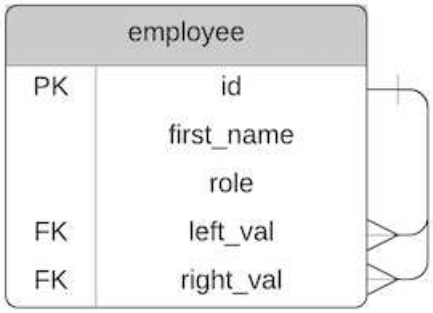
Slika 2.3 Vizualizacija modela ugniježđenog skupa pomoću serije kontejnera [1]

Također je potrebno numerirati lijevi i desni rub svakog kontejnera. To će nam biti potrebno za pisanje upita i održavanje podataka u tablici.

Numeracija se provodi tako da počinjemo s brojem 1, koji je lijevi rub najvećeg spremnika. Zatim dodajemo broj 2 sljedećem obrubu, bilo da je to lijevi ili desni obrub. Nastavljamo dok sve lijeve i desne granice ne budu numerirane.

Sada je važno primijeti da svaki kontejner ima broj s lijeve i broj s desne strane.

Kao i u modelu popisa susjeda koristimo model samoreferencirajućih tablice, ali dodajemo dva nova stupca *left_val* i *right_val* koji predstavljaju lijevi odnosno desni rub kontejnera. Model talice dan je na slici 2.4.



Slika 2.4 Radna tablica za implementaciju modela ugniježđenog skupa [1]

Ovakav model omogućava pisanje lakših upita te su također upiti puno brži budući da više ne trebamo koristiti rekurzivni WITH već se koristi jednostavno spajanje tablica i njihovo filtriranje.

Evo primjera:

```
SELECT e2.id, e2.first_name, e2.role, e2.left_val, e2.right_val
FROM employee e1
INNER JOIN employee e2 ON e2.left_val BETWEEN e1.left_val AND e1.right_val
WHERE e1.id = 1;
```

Ovaj primjer će nam ispisati cijelu našu hijerarhiju budući da smo za početni čvor stavili ID 1 pomoću WHERE filtera. Rezultat upita dan je tablicom 2.5.

Također za razliku od modela popisa susjeda SELECT upiti su mnogo kraći za pisati.

id	first_name	role	left_val	right_val
1	Claire	CEO	1	28
2	Steve	Head of Sales	2	7
3	Sarah	Head of Support	8	17

4	John	Head of IT	18	27
5	Nathan	Salesperson	3	4
6	Paula	Salesperson	5	6
7	Anne	Customer Support Officer	9	10
8	Roger	Customer Support Officer	11	12
9	Tim	Customer Support Officer	13	14
10	Peter	Assistant	15	16
11	Hugo	Team Leader	19	26
12	Matthew	Developer	20	21
13	Ruth	Developer	22	23
14	Simon	QA Engineer	24	25

Tablica 2.5 Ispis hijerarhije pomoću modela ugniježđenog skupa

Iz ovog primjera vidimo da se ispisuju svi elementi koji su između vrijednosti 1 i 28 *left_val*-a odnosno *right_val*-a specificiranog na temelju početnog čvora.

Za ispis hijerarhije za specifični početni čvor koristimo WHERE klauzulu u kojoj zadamo početni čvor pretrage.

```
SELECT e2.id, e2.first_name, e2.role, e2.left_val, e2.right_val
FROM employee e1
INNER JOIN employee e2 ON e2.left_val BETWEEN e1.left_val AND e1.right_val
WHERE e1.id = 4;
```

id	first_name	role	left_val	right_val
4	John	Head of IT	18	27
11	Hugo	Team Leader	19	26
12	Matthew	Developer	20	21
13	Ruth	Developer	22	23
14	Simon	QA Engineer	24	25

Tablica 2.6 Ispis hijerarhije ispod Head of IT-a

Na ovom primjeru vidimo ispis redaka tablice koji se nalaze između vrijednosti

18 (left_val) i 27 (right_val) budući da je sada početni čvor s ID-jem 4 koji ima rolu Head of IT.

Iz gore navedenog vidimo da je u modelu hijerarhije ugniježđenog skupa lako pisati SELECT upite koji su ujedno i brži naspram modela hijerarhije na temelju susjeda jer nije potrebno koristiti rekurzivnu WITH klauzulu već jednostavne JOIN-ove.

Za primjer dodavanja novog elementa dodat ćemo zaposlenika sljedećeg zaposlenika u hijerarhiju:

- ID: 15
- Name: Alex
- Role: Salesperson
- Manager: Steve (ID 2)

Dodajemo novog zaposlenika u diviziju Sales kojom upravlja Steve (ID 2).

Iz tablice 2.5 za redak zaposlenika Steve (ID 2) vidimo da je left_val 2 i right val 7, odnosno da on upravlja zaposlenicima u tom rangu.

Kako bismo dodali novi element potrebno je napraviti dva koraka:

1. Potrebno je „napraviti prostor" za novi zapis, povećanjem vrijednosti lijeve i desne strane svih elemenata većih od onoga gdje taj novi zapis ide.

   Novi zapis će imati lijevu vrijednost 7, jer ide nakon Paule koja ima desnu vrijednost 6.

   Prvi SQL upit će ažurirati tablicu kako bi se lijeve i desne vrijednosti povećale za 2 (tako da novi zapis može imati lijeve i desne vrijednosti)

2. Sada kada smo napravili prostor (povećavanjem lijevih i desnih vrijednosti za 2), možemo dodati novi zapis pomoću sljedećeg INSERT-a:

```
UPDATE employee
SET left_val = CASE WHEN left_val >= 6 THEN left_val + 2 ELSE left_val END,
right_val = right_val + 2
WHERE right_val >= 6;

INSERT INTO employee (id, first_name, role, left_val, right_val)
VALUES (15, 'Alex', 'Salesperson', 6, 7);
```

id	first_name	role	left_val	right_val
1	Claire	CEO	1	30
2	Steve	Head of Sales	2	9
3	Sarah	Head of Support	10	19
4	John	Head of IT	20	29
5	Nathan	Salesperson	3	4
6	Paula	Salesperson	5	6
...	...	...	...	...
15	Alex	Salesperson	7	8

Tablica 2.7 Rezultat dodavanja novog elementa

Iz navedenog vidimo da smo prije morali proći kroz sve čvorove hijerarhije, nad njima napraviti UPDATE stupaca left_val i right_val kako bismo oslobodili mjesta za novi čvor.

Micanje elementa iz hijerarhije

Proces je sličan kao i kod dodavanja novog elementa. Napravimo mjesta te UPDATE left_val i right_val atributa elementa koji prebacujemo. Na primjer ako želimo zaposlenika Peter (ID 10) prebaciti iz Support tima u Sales tim, moramo ažurirati lijevu i desnu vrijednost za retke između sadašnjeg retka i novog retka. Prvi SQL upit je napravio mjesta u Sales i ostalim divizijama. Nakon toga pomoću UPDATE-a zaposlenik se prebacuje u tu diviziju. Slijedi primjer:

```
UPDATE employee
SET left_val = CASE WHEN left_val >= 6 THEN left_val + 2 ELSE left_val END,
right_val = right_val + 2
WHERE right_val >= 6;

UPDATE employee
SET left_val = 7,
right_val = 8
WHERE id = 10;
```

Brisanje elementa je vrlo jednostavno ostvariti budući da nema ograničenja ključa (eng. key violation constraint) kada izbrišemo redak tablice čak i ako je on menadžer.

```
DELETE FROM employee
WHERE id = 14;
```

Upit za brisanje cjeline pod stabla hijerarhije vrlo je jednostavan. Na primjer ako se odlučimo obrisati cijelu diviziju Sales jednostavno u WHERE filter označimo glavnog menadžera te divizije i jednostavno na temelju left_val i right_val određuju se elementi hijerarhije koji će se obrisati.

```
SELECT e2.id, e2.first_name, e2.role, e2.left_val, e2.right_val
FROM employee e1
INNER JOIN employee e2 ON e2.left_val BETWEEN e1.left_val AND e1.right_val
WHERE e1.id = 2;
```

Prednosti koncepta popisa susjedstva:

- Brzi SELECT upiti
- Jednostavni SELECT upiti, lagani za razumjeti i pisati budući da se koristi samo JOIN s BETWEEN i WHERE filterom.
- Lako je premjestiti elemente na drugo mjesto u hijerarhiji

Nedostatci koncepta popisa susjedstva:

- Dodavanje novih zapisa je složeno, potrebno je druge zapise ažurirati.
- Brisanje i premještanje zapisa također je složeno iz istog razloga.

2.1.3. Usporedba Modela

Usporedba modela popisa susjeda s modelom ugniježđenog skupa:

Model	SELECT	INSERT	UPDATE	DELETE
Popis susjeda	Teško	Lagan	Lagan	Lagan *
Ugniježđeni skup	Lagan	Teško	Teško	Lagan

Tablica 2.8 Usporedba modela popisa susjeda s modelom ugniježđenog skupa

SELECT upite puno je lakše pisati u modelu ugniježđenog skupa za razliku u modela popisa susjeda.

Kod INSERT-a upita primijetili smo da je lakše dodati novi element kod modela popisa susjeda., potreban nam je samo ID čvora roditelja. U modelu ugniježđenog skupa osim što trebamo znati ID roditelja moramo znati i njegov left_val i right_val, još je ktome potrebno

napraviti mjesta za novi element koji se dodaje roditelju. Kod UPDATE ista je problematika kao i kod INSERTA kod modela ugniježđenog skupa.

Na kraju DELETE je vrlo jednostavno ostvariti kod ugniježđenog skupa budući da se elementi brišu na temelju left, right val. Zbog tih vrijednosti također se lako obrišu i djeca roditelja ako se to želi. Ako se žele ostaviti djeca jednostavno se obriše samo roditelj, a djeca koja ostanu bit će vezana u hijerarhiji na roditelja jednu hijerarhiju više.

DELETE kod modela popisa susjeda može biti kompleksan ako se želi obrisati roditelj koji ima djecu. Kao što smo pokazali u primjeru djeca se prvo trebaju prebaciti na drugog roditelja kako bi se ostvarilo brisanje samog roditelja, a da djeca ostanu u bazi.

3. Graf baze i problem hijerarhije

Baze podataka na temelju grafa spadaju u kategoriju NoSQL koje su posebno dizajnirane za rukovanje podacima sa složenim odnosima i međusobnim vezama. U domeni NoSQL-a graf baze posebno su prikladne za aplikacije koje zahtijevaju duboke i složene upite, poput društvenih mreža, mehanizama za preporuke i sustava za otkrivanje prijevara.
S bazama podataka na temelju grafa imamo mogućnost pohraniti odnose između kategorija kao što su interesi kupaca, prijatelji i povijest kupovine. Navedene podatke u bazi pak koristimo kako bismo korisniku preporučili proizvode na temelju proizvoda koje kupuju drugi korisnici koji prate isti kategorije proizvoda ili imaju sličnu povijest kupovine.
Također pomoću podataka iz graf baza lako se mogu identificirati ljudi koji imaju zajedničkog prijatelja, ali se još oni ne poznaju međusobno te im možemo ponuditi preporuku za prijateljstvo. Graf baze danas se također koriste za prevenciju sofisticiranih prijevara. Primjer toga je istraga Pandora Papers [7], koja je koristila graf bazu Neo4j za povezivanjem osoba s *offshore* kompanijama i pranjem novca.

Još jedna od glavnih prednosti graf baza je njihova fleksibilnost. Takve baze mogu rukovati podacima s promjenjivom strukturom i mogu se prilagoditi novim slučajevima upotrebe bez potrebe za značajnim promjenama same sheme baze podataka. To ih čini posebno korisnima za aplikacije s brzim promjenama strukture podataka.

Danas popularne baze podataka na temelju grafa su:

- Neo4J
- OrientDB
- ArangoDB

Danas graf baze imaju široku primjeru, neke od primjena su[2],[11]:

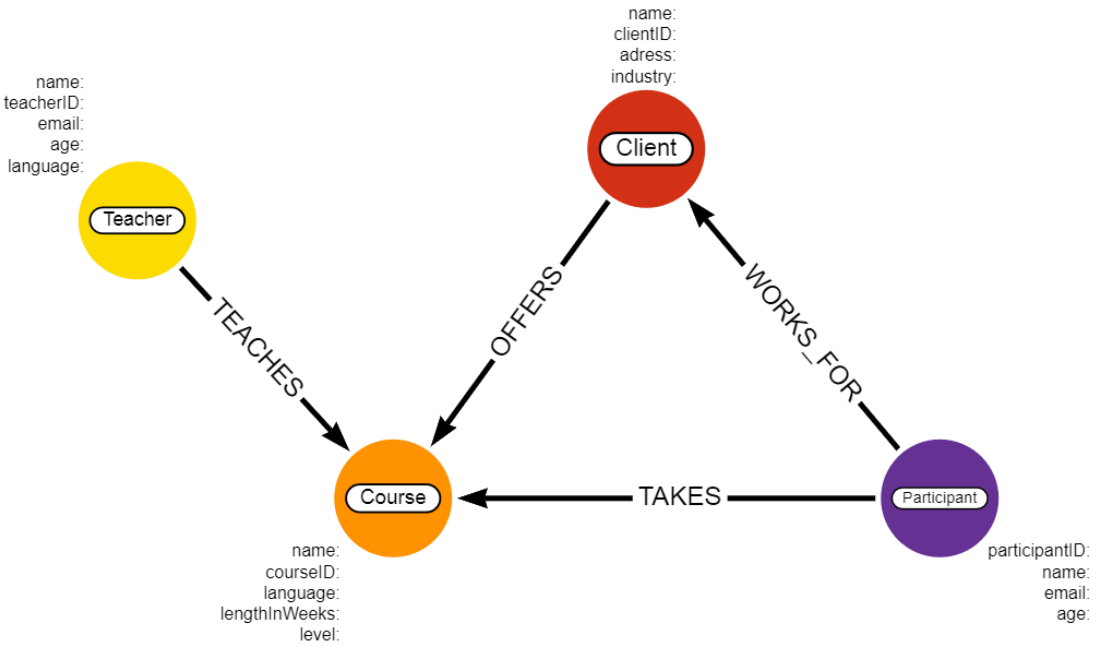
- Društvene mreže: Zbog svojih svojstava graf model podataka idealan je za modeliranje složenih odnosa između korisnika društvenih mreža. Na primjer prijateljstava, pratitelja, lajkova itd. Klasičan problem u relacijskom modelu tko su sve prijatelji mojih prijatelja.

- Pretraživanje web-stranica: Graf model podataka također se koristiti za pretraživanje weba. Google koristi grafove za svoju pretraživačku tehnologiju, koja temelji rezultate pretraživanja na mreži povezanih stranica.
- Karte, geografske lokacije: Grafa baze se također koriste za modeliranje složenih geografskih podataka, kao što su rute i putovanja. Na primjer, OpenStreetMap koristi grafove za modeliranje cesta i ruta.
- Preporučiteljski sustavi: Danas jedna od najčešćih upotreba grafa je izrada marketinškog identiteta korisnika u smislu personaliziranih reklama.

  Također graf baze koriste se u svrhu preporučivanje filmova i glazbe na temelju sličnosti između entiteta i korisnika. Netflix i Spotify su najpoznatiji primjeri tvrtki koji koriste graf baze u tu svrhu.

Model Grafa sastoji se od:

- čvorova (eng. Nodes) koji su temeljna jedinica podataka. Cijeli model dizajniramo oko ovih entiteta.
- veza (eng. Links) koje predstavljaju relacije/odnose između čvorova. Veze mogu biti jednosmjerne ili dvosmjerne



Slika 3.1 Primjer modela baze podataka na temelju grafa [8]

Prema slici 3.1 vidimo da svaki čvor ima značke (eng. Label). Na temelju tih znački možemo razlikovati tipove čvorova. Na istom primjeru možemo vidjeti da imamo čvorove tipa: Teacher, Course, Client i Participant.

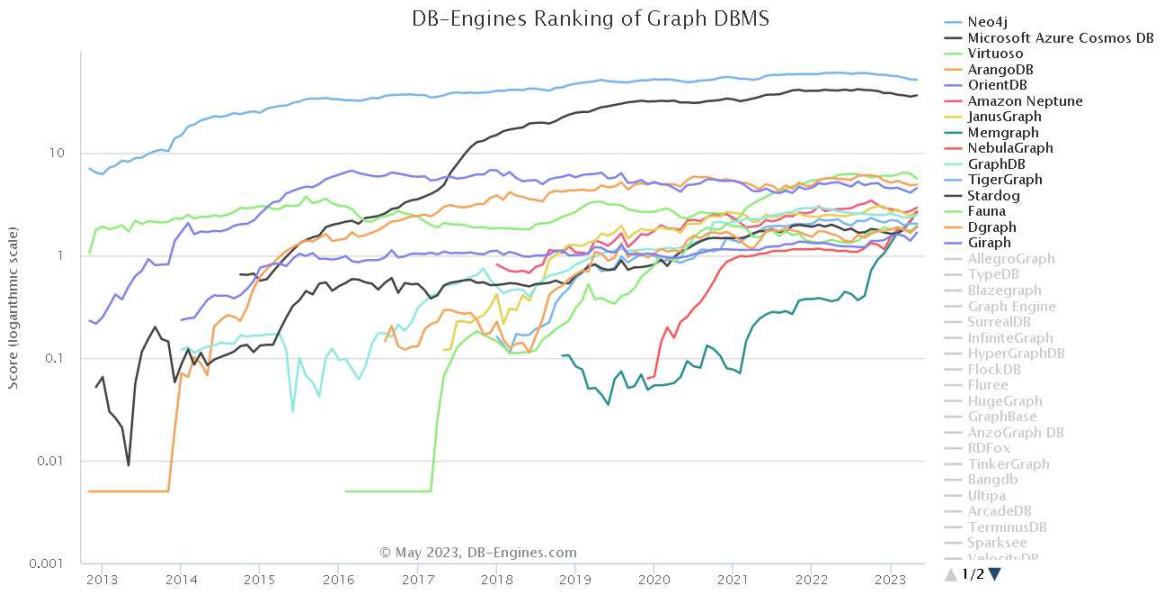
Uz postojanje znački svakom čvoru pridjeljujemo i svojstva (eng. properties) ključ-vrijednost parove koji imaju svrhu opisa čvora. Prema gore navedenom primjeru čvor tima Teacher ima svojstva: name, teacherID, email, age i language. Drugim riječima svojstva u modelu grafa su opisne karakteristike čvorova i veza, ali te karakteristike nisu dovoljno važne da postanu čvorovi .Na temelju tipova čvorova i njima pridijeljenim vrijednostima omogućeno nam je efikasno filtriranje između čvorova.

Kod prebacivanja iz relacijskog modela u model grafa dobro je zamisliti čvorove kao tablice, veze izemđu čvorova kao strane ključeve i svojstva čvorova kao atribute entiteta.

3.1. Neo4j i upitni jezik Cypher

Neo4j danas je najpopularniji sustav za upravljanje bazom podataka na temelju grafa dizajniranom za pohranu, upravljanje i pisanje upita nad podatcima.

Uz to graf bazu Neo4j u odnosu na druge NoSQL baze podataka izdvaja to što Neo4j podržava svojstva ACID: atomarnost (eng. atomicity), konzistentnost (eng. consistency), izolacija (eng. isolation), trajnost (eng. durability).



Slika 3.2. Popularnost pojedinih sustava za upravljanje graf

bazom podataka., 2023. godine[10]

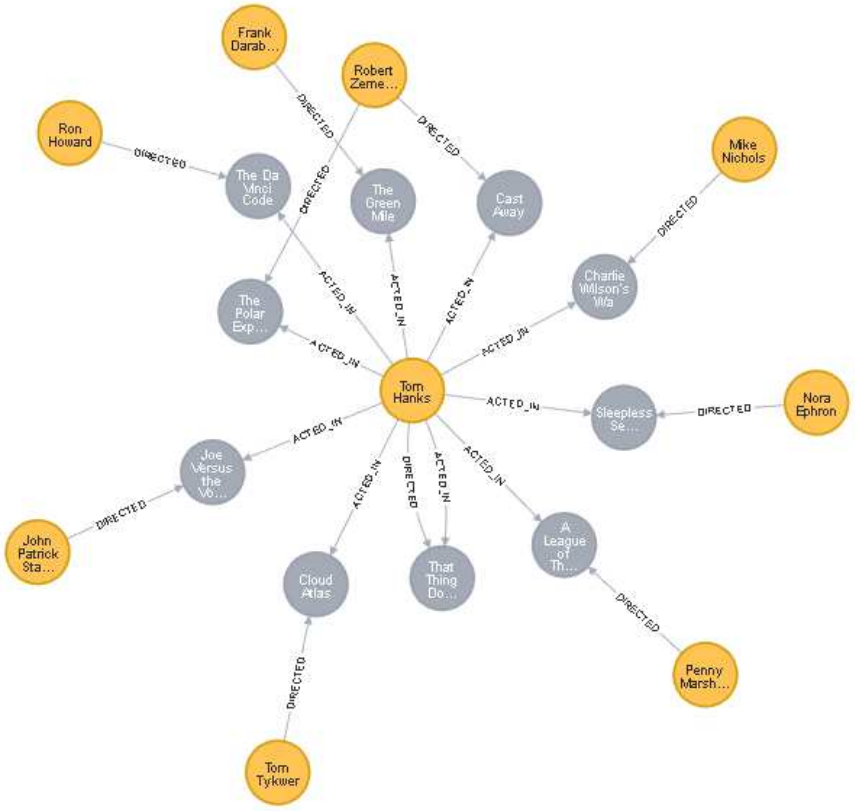
Kao što relacijske baze podržavaju SQL kao standardni upitni jezik tako Neo4j podržava upitni jezik Cypher, koji se koristi za izvođenje složenih upita sa svrhom analitike podataka grafovima.

Neo4j izrađen je u programskom jeziku Java i ima odličnu podršku za Spring Boot aplikacije. Također nudi niz alata i API-ja za integraciju s drugim sustavima i aplikacijama.

3.1.1. Upitni jezik Cypher

Cypher Query Language, ili jednostavnije Cypher, je deklarativni upitni jezik dizajniran na način koji omogućuje lako čitanje i razumijevanje svim dionicima uključenim u razvoj i korištenje baze podataka. Njegova jednostavnost korištenja proizlazi iz činjenice da se piše tako da intuitivno opisujemo grafove pomoću dijagrama. Fokusira se na rezultate upita, a ne na specifične metode ili načine za dobivanje tih rezultata. To znači da je jezik izrazito čitljiv i ekspresivan za ljude.

Za objašnjenje Cypher sintakse koristit ćemo gotovu Neo4j graf bazu movie-graph [5]. Baza podataka movie-graph sastoji se od dvije vrste čvorova: Movie i Person. Čvor Movie ima sljedeća svojstva: ID, released, tagline, title, dok čvor Person ima: ID, born, name. Čvor Person je povezan dvijama vrstama jednosmjernih relacija s čvorom Movie DIRECTED i ACTED_IN.



Slika 3.3 Prikaz dataseta movie-graph

Prvi radni primjer Cypher upita kojeg ćemo analizirati je sljedeći:

```
MATCH (tom:Person {name: "Tom Hanks"})-[rel:ACTED_IN]->(tomHanksMovies)
RETURN tom,rel,tomHanksMovies
```

Kao i većina upitnih jezika, Cypher se sastoji od ulaznih klauzula. Najjednostavniji upiti sastoji se klauzule MATCH iza koje slijedi klauzula RETURN. klauzulu MATCH možemo poistovjetiti sa klauzulom SELECT u SQL-u. Sam kostur klauzule MATCH Cypher upitnog jezika je sljedeći:

```
MATCH (first_node:LABEL)-[rel:RELATIONSHIP]->(second_node:LABEL)
```

Za sada primijetimo da imamo tri elementa:

- (first_node:LABEL)
- -[rel:RELATIONSHIP]->
- (second_node:LABEL)

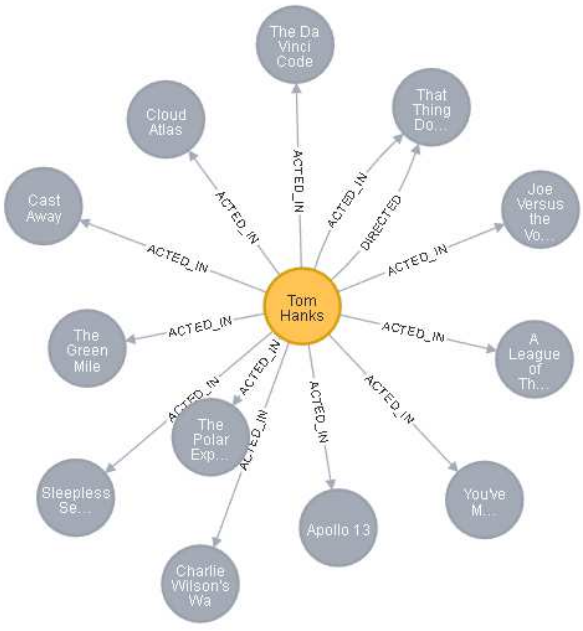
Unutar oblih zagrada () referenciramo čvorove (engl. Nodes). U primjeru imamo sljedeći primjer referenciranja prvog čvora:

(tom:Person {name: "Tom Hanks"})

Tražimo tip (engl. Label) čvora Person koji ima svojstvo (eng. property) *name: "Tom Hanks"*. I na kraju pronađene čvorove želimo spremiti u referencu koju smo nazvali *tom*.

Pogledajmo sada sliku 3.4, za sada smo pronašli žuti čvor tipa Person Tom Hanks.
U upitu sada slijedi *-[rel:ACTED_IN]->*. Navedena sintaksa predstavlja vezu (eng. Link) koja spaja čvorove. Tražimo vezu tipa *ACTED_IN* i to spremamo u referencu koju smo nazvali rel.
Važno je primijeti da imamo smjer reference iz prvog čvora u drugi čvor match klauzule *–[]->*.
U MATCH klauzuli još nam je ostalo da definiramo *second_node*. U ilustracijskom primjeru to je napravljeno pomoću sljedeće sintakse *(tomHanksMovies)*. Možemo primijetiti da u ovoj sintaksi nismo eksplicitno definirali tip krajnji čvor pomoću operatora *:* iza kojega se definira tip. U ovom primjeru dopuštamo da se ispitaju čvorovi bilo kojeg tipa i svi pronađeni čvorovi spremaju se u referencu nazvanu *tomHanksMovies*.

Da rekapituliramo:
Tražimo početne čvorove tima Person koji imaju svojstvo name: „Tom Hanks“ te koji imaju vezu ACTED_IN s bilo kojim drugim čvorovima.
Na kraju RETURN klauzuli dajemo pohranjene reference i dobivamo vizualizaciju koja je dana na slici 3.4.



Slika 3.4 Rezultat upita nad movie-graph datasetom

Kao i u ostalim upitnim jezicima pojedine upite je moguće napisati na različite načine. Gore pokazni primjer mogli smo također i ovako napisati:

```
MATCH (tom:Person)-[:ACTED_IN]->(tomHanksMovies)
WHERE tom.name = "Tom Hanks"
RETURN tom,tomHanksMovies
```

U Cypheru dobra je praksa jednostavne filtere čvorova pisati pomoću vitičastim zagradama na temelju *key:value* odmah iza referenciranja čvorova. No ništa nas ne brani da taj filter napišemo pomoću WHERE klauzule. U gornjem primjeru sve čvorove tipa PERSON spremamo u referencu tom te filtriramo ih pomoću WHERE klauzule.

WHERE tom.name = "Tom Hanks"

U upitu možemo primijetiti da ne spremamo referencu vezu *rel* između čvorova. Neo4J Desktop vizualizacijski paket automatski dodaje veze između čvorova prilikom izvođenja RETURN klauzule koje su nađene u MATCH klauzuli unutar *-[]->* slijeda.

3.1.1.1 Klauzule

Klauzula MATCH

Koristimo ju za pronalazak čvorove s određenim svojstvima

```
MATCH (city:City)-[:IN]-(country:Country)
WHERE city.name = "Zagreb"
RETURN country.name
```

MATCH (city:City)-[:IN]-(country:Country): klauzula MATCH specificira čvorove tipa City i Country koje pohranjuje u reference *city* i *country* te također specificira i vezu između ta dva povezana čvora tipa *IN*.

WHERE c.name = "Zagreb": klauzula WHERE filtrira rezultate sa svojstvom *name* varijable *c* tipa *City* sa vrijednošću *"London"*.

RETURN c.name: klauzula RETURN koristi se za vraćanje rezultata.

Navedeni upit također može biti napisan bez WHERE klauzule tako da filter direktno dodamo unutar MATCH klauzule:

```
MATCH (c:City {name: "Zagreb"})-[:IN]-(country:Country)
RETURN c.name
```

Klauzula CREATE/MERGE

Navedene klauzule koristimo za stvaranje čvorova i veza između njih:

```
CREATE (country:Country {name: "Hrvatska"}),
       (city:City {name: "Zagreb", population_size: 1000000})
       (country)<-[r:IN]-(city)
RETURN country, city, r;
```

city:City: stvara novi čvor tipa *City* i sprema referencu varijablu *city*.

{name: "Zagreb", population_size: 1000000}: novostvoreni čvoru dodjeljujemo dva svojstva *name* i *population_size*.

(country)<-[r:IN]-(city): Stvaranje veze između reference *country* i *city* tipa *IN*.

Za stvaranje veza između čvorova koji već postoje koristimo MERGE klauzulu. Ako čvorovi unutar MERGE klauzule ne postoje tada se MERGE ponaša kao i CREATE klauzula.

Ako navedeni čvorovi već postoje MERGE prvo napraviti MATCH tih čvorova te zatim dodaje veze koje smo specificirali. Evo primjera:

```
MATCH (country:Country {name: "Hrvatska"}),
      (city:City {name: "Zagreb", population_size: 1000000})
MERGE (city)-[:IN]->( country);
```

Klauzule SET i REMOVE

Ažurirajte svojstva čvora:

```
MATCH (c:Country {name: "HR"})
SET c.name = "Hrvatska";
```

SET c.name = "United Kingdom": pomoću SET varijabli c ažuriramo svojstvo name.

Dodavanje svojstva čvorovima:

```
MATCH (c:Country)
WHERE c.official_language IS null
SET c.official_language = c.language
```

WHERE c.official_language IS null: klauzula WHERE osigurava da novo svojstvo official_language dodamo samo čvorovima koje još nemaju to svojstvo.

SET n.official_language = n.language: stvaramo novo svojstvo official_language i da mu pridjeljujemo vrijednost od language svojstva.

Brisanje svojstva čvora:

```
MATCH (c:Country)
REMOVE c.language;
```

REMOVE n.language: klauzula REMOVE koristimo za brisanje svojstva language.

Klauzule DELETE i DETACH

Koristimo ju za brisanje čvorova i veza između njih, evo primjera:

```
MATCH (city:City)-[r:IN]-(country:Country)
WHERE city.name = "Zagreb"
DELETE city, r, country
```

DELETE city, r, country: brišemo čvorove i veze pohranjene u varijablama city, r i country

Moramo naglasiti da nije moguće obrisati čvor koji je povezan s ostalim čvorovima preko veza, da bismo obrisali takve čvorove koristimo DETACH DELETE:

```
MATCH (city:City)
WHERE city.name = "Zagreb"
DETACH DELETE city
```

Na ovaj način prvo se pobrišu sve veze između čvora Zagreb te se zatim obiše i sam čvor

Ograničenja (CONSTRAINTS)

Ograničenje jedinstvenosti (eng. uniqueness)

```
CREATE CONSTRAINT ON (c:City)
ASSERT c.location IS UNIQUE;
```

Ovim upitom osiguravamo da svaki čvor tipa City ima jedinstvenu vrijednost za svojstvo lokacije.

Brisanje ograničenja jedinstvenosti

```
DROP CONSTRAINT ON (c:City)
ASSERT c.location IS UNIQUE;
```

Ograničenje postojanja (eng. existence)

```
CREATE CONSTRAINT ON (c:City)
ASSERT exists (c.name);
```

Ovim upitom osiguravamo da svaki čvor tipa City ima svojstvo name.

Brisanje egzistencijalnog ograničenja

```
DROP CONSTRAINT ON (c:City)
ASSERT exists (c.name);
```

3.2. Neo4j i paket Graph Dana Science

Neo4j uz razvoj svog programskog alata Neo4j Desktop za vizualizaciju i izvršavanje Cyper upita razvio je i različite kategorije algoritma za rješavanje problema pretrage i izrade analiza nad grafovima.

Razvoj je započet sa APOC (Awesome Procedures on Cypher) programskom bibliotekom. To je omogućilo da korisnici koriste standardne biblioteke za uobičajene procedure.

APOC biblioteku naslijedio je paket Neo4j Graph Data Science. Radi se o skupu alata i algoritama dizajniranih da pomognu korisnicima u analizi podataka iz grafova. Navedeni paket pruža okvir za primjenu naprednih algoritama analitike nad podatcima te korisnicima pomaže identificirati obrasce i odnose između podataka koji nisu vidljivi korištenjem tradicionalnih analitičkih tehnika.

Radi se o algoritmima iz domena problema: centralnosti (eng. centrality), otkrivanje zajednice (eng. community detection), ugradnje čvorova (eng. node embeddings), mjere sličnosti (eng. similarity) i pronalaženje puta (eng. path finding).

Alat također uključuje algoritme grafova za strojno učenje, kao što je klasifikacija čvorova (eng. node classification), regresiju čvorova (eng. node regression), predviđanje veze (eng. link prediction) i grupiranje grafova (eng. graph clustering). Ovim algoritmi korisnicima je omogućeno da iskoriste tehnike strojnog učenja za prepoznavanje uzoraka i predviđanje ishoda iz podataka grafikona.

Neki od češće korištenih algoritama paketa Graph Data Science[4] su sljedeći:

- problem pronalaženja puta (eng. Path finding):

    - pronalazak puta između dva ili više čvorova, procijeniti dostupnost i cijenu putova
    - A* Shortest Path, Yen's Shortest Path, Dijkstra Single-Source Shortest Path, Delta-Stepping Single-Source Shortest Path Random Walk

- problem otkrivanja zajednica (eng. Community detection)
    - algoritmi za otkrivanje zajednice koriste se za procjenu načina na koji su grupe čvorova klasterirane ili podijeljene, kao i njihovu tendenciju jačanja ili raspadanja
    - Louvain, Label Propagation, Weakly Connected Components, Local Clustering Coefficient, K-1 Coloring, Modularity Optimization
- problem centralnosti (eng. Centrality)
    - algoritmi centralnosti koriste se za određivanje važnosti čvorova u mreži
    - Eigenvector Centrality, PageRank, Betweenness Centrality
- problem sličnosti (eng. Similarity)
    - algoritmi sličnosti izračunavaju sličnost parova čvorova na temelju njihovih susjeda ili njihovih svojstava.
    - aazličite metrike za definiranje sličnosti
    - Node Similarity, K-Nearest Neighbors
- problem pretvorbe čvorova (eng. Node embeddings)
    - predstavljanje čvorova baze kao vektora
    - algoritmi za pretvorbu čvorova izračunavaju vektorske reprezentacije čvorova u grafu. Kasnije se ti vektori koristiti za strojno učenje.
    - FastRP, GraphSAGE, HashGNN
- problem predviđanja veza (eng. Topological link prediction)
    - algoritmi određuju blizine para čvorova na temelju topologije grafa.
    - azračunati rezultati se zatim koristite za predviđanje odnosa i veza između dva vrha u paru.
    - Adamic Adar, Common Neighbors, Preferential Attachment

Uz graf algoritme Graph Dana Sceince pruža algoritme i modele za strojno učenje. Također prilikom treniranja teško je znati koliko nam je regulacije potrebno kako nam model ne bismo prenaučili naš model podatcima za treniranje. Iz toga razloga Neo4j daje nam opciju korištenja automatskog podešavanja (eng. auto tuning).

Implementirani modeli za treniranje:

- klasifikacija (eng. Classification)
  - predviđa diskretne vrijednosti (oznake klase)
  - Logistic regression, Random forest, Multilayer Perceptron
- regresija (eng. Regression)
  - predviđa kontinuirane vrijednosti
  - Random forest, Linear regression

Također unutar paketa nalaze se i cjevovodi koji pripremaju podatke za treniranje modela

- Training pipeliney (eng. Training pipeline)
  - ulazne podatke prethodno je skalirati i normalizirati.
  - cjevovodi nam omogućuju da specificiramo slijed koraka pretprocesiranja koji se primjenjuju na ulazne podatke. Na primjer ulazne podatke prethodno je potrebno skalirati, normalizirati, odrediti koje varijable se koriste itd.
  - Node Classification Pipelines, Link Prediction Pipelines, Node Regression Pipelines

U nastavku je dan primjer poziva procedure BFS iz GDS (eng. Graph Dana Science) paketa. Za ovaj primjer korišteni su čvorovi A, B, C, D i E čiji je kod dan u prilogu.

```
MATCH (a:Node{name:'A'}), (d:Node{name:'D'}), (e:Node{name:'E'})
WITH id(a) AS source, [id(d), id(e)] AS targetNodes
CALL gds.bfs.stream('myGraph', {
  sourceNode: source,
  targetNodes: targetNodes
})
YIELD path
RETURN path
```

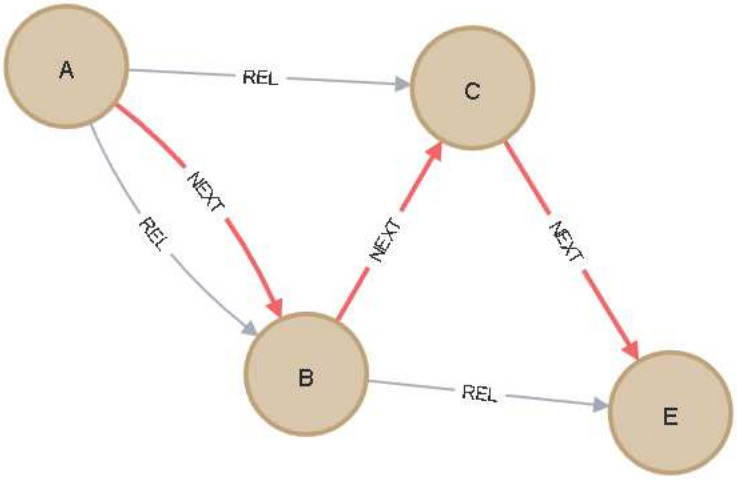
MATCH (a:Node{name:'A'}), (d:Node{name:'D'}), (e:Node{name:'E'}): na početku pomoću MATCH-a pronađemo čvorove koje ćemo kasnije koristiti kao početak (eng. source) i cilj (eng. target).

WITH id(a) AS source, [id(d), id(e)] AS targetNodes: ovime označujemo čvor A kao početni te čvorove D i E kao ciljne čvorove.

CALL gds.bfs.stream('myGraph', { sourceNode: source, targetNodes: targetNodes }): ovime zovemo proceduru bfs i dajemo joj argumente sourceNode i targetNodes.

YIELD path: eksplicitno odabire koja polja će se vratiti iz procedure.

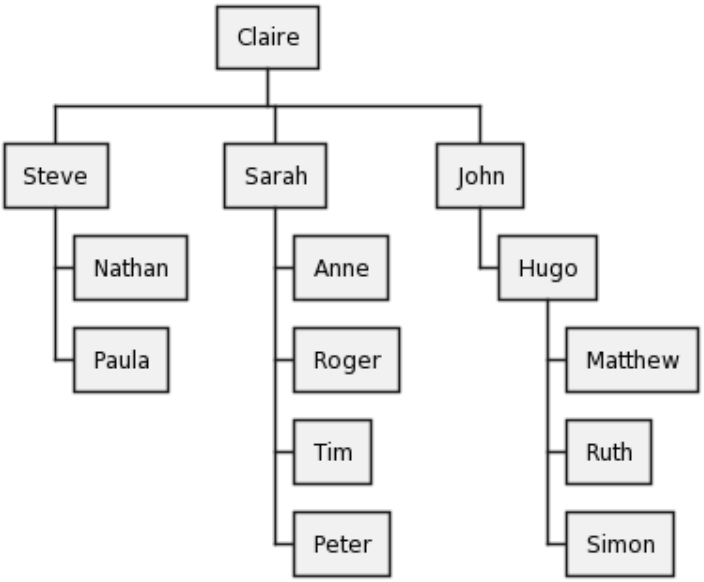
Izlaz upita dan je na slici 3.6.



Slika 3.6 Izlazna vrijednost BFS algoritma u obliku puta (eng. path) paketa GDS-a

3.3. Radni primjer hijerarhije među zaposlenicima

U ovom poglavlju ilustrirat će se izrada hijerarhije iz poglavlja **2.1. Problem hijerarhije u relacijskim bazama** te će se na kraju također napraviti usporedba modela iz graf baza s relacijskim modelom po pitanju težine traženja, dodavanja, brisanja i modifikacije elemenata.

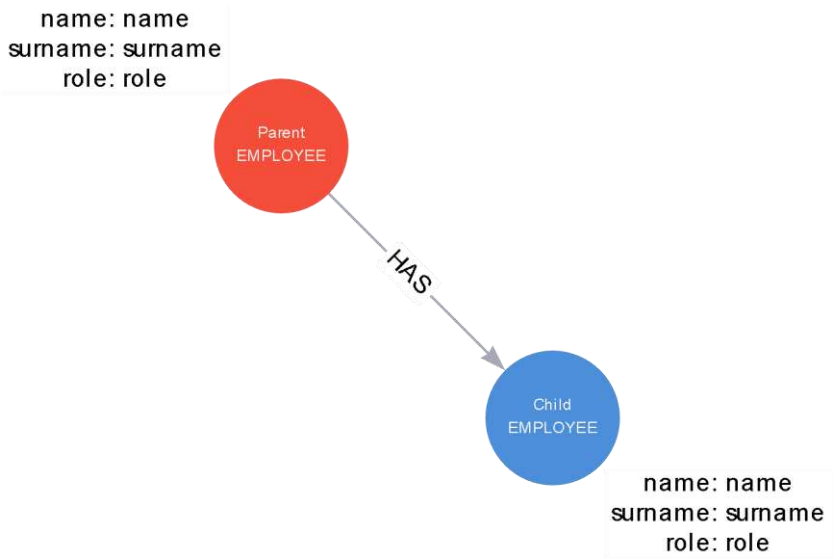


Slika 3.7 Ulazna hijerarhija [1]

3.3.1. Model graf baze

Na početku trebamo izraditi konceptualni model[9] za graf bazu. Započinjemo s određivanjem tipova potrebnih čvorova. Mogući kandidati za čvorove su Zaposlenici (eng. employee) i njihova funkcija (eng. role) na poslu. Naš zaposlenik ima svojstva ime i prezime, stoga prilikom modeliranja modela možemo ga odmah klasificirati kao čvor.

Sada dolazimo do kandidata funkcija na poslu, budući da rola nema svojih posebnih svojstava te da jedan zaposlenik može imati maksimalnu jednu rolu nema potrebe taj element postavljati kao čvor. Stoga ulogu ukomponiramo kao svojstvo *role* (eng. property) čvora zaposlenika. Da je na primjer čvor zaposlenika mogao imati više uloga onda bi element funkcije posla također bio čvor sa svojstvom imena *role*. Model prema gore navedenim postupcima modeliranja prikazan je na slici 3.8.



Slika 3.8 Model hijerarhije zaposlenika graf baze

3.3.2. Izrada i popunjavanje graf baze ulaznom hijerarhijom

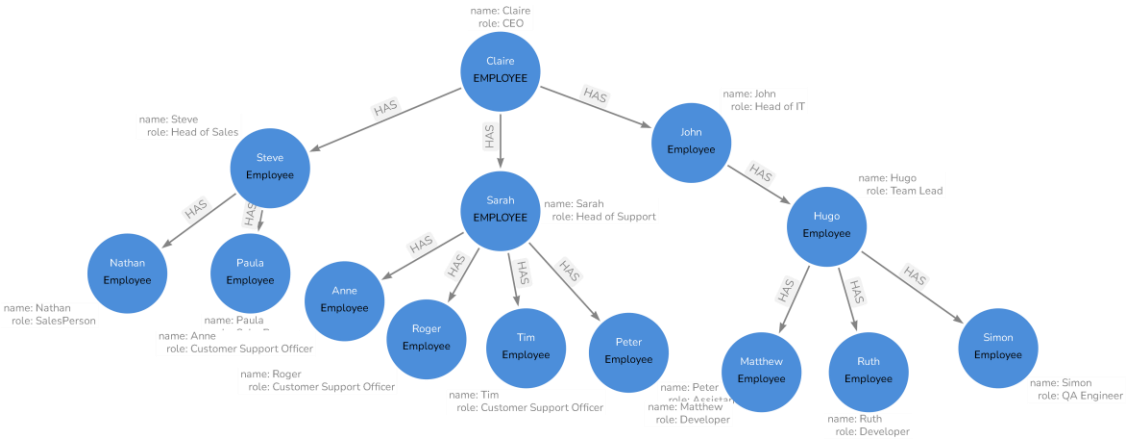
Za izradu čvorova i veza između njih koristimo sljedeću CREATE i po potrebi MERGE klauzule. U daljem tekstu objasnit će se dva načina generiranja stabla.

Prvi način, koristeći CREATE i MERGE klauzule

```
CREATE (claire:Employee {name: "Claire", role: "CEO"})
CREATE (steve:Employee {name: "Steve", role: "Head of Sales"})
CREATE (john:Employee {name: "John", role: "Head of IT"})
CREATE (sarah:Employee {name: "Sarah", role: "Head of Support"})

MERGE (claire)-[:HAS]->(steve)
MERGE (claire)-[:HAS]->(john)
MERGE (claire)-[:HAS]->(sarah)
```

Ovime je dan isječak koda za izradu dijela stabla hijerarhije. Odmah možemo primijetiti da je kod podijeljen u dva dijela. U prvom dijelu stvaramo čvorove pomoću CREATE klauzule i njihove reference spremamo u varijable *claire*, *steve*, *john* i *sarah*. Kasnije koristeći varijable na gore stvorene čvorove pomoću MERGE klauzule dodajemo veze između referenciranih čvorova.

Drugi način generiranja grafa je pomoću alata Neo4j Arrows. U navedenom alatu kroz grafičko sučelje crtamo graf tako što dodajemo čvorove i veze između njih. Kroz alat Arrows moguće je definiranje tipova i svojstva za čvorove i veze. To je prikazano na slici 3.9.

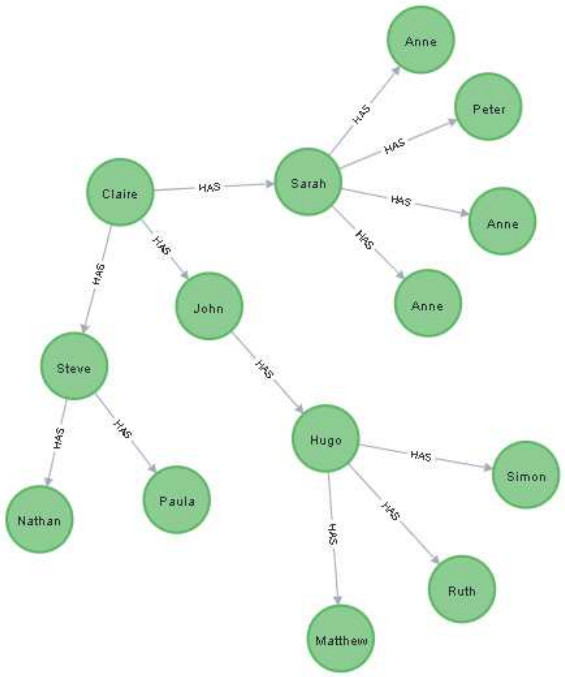


Slika 3.9 Izrada stabla hijerarhije pomoću Neo4j Arrows grafičkog alata

Na kraju nacrtani graf iz Arrows grafičkog alata exportamo kao Cypher kod koji je dan u nastavku:

```
CREATE (:EMPLOYEE {name: "Anne", role: "Customer Support Officer"})<-[:HAS]-
(Sarah:EMPLOYEE {name: "Sarah", role: "Head of Support"})<-[:HAS]-
(Claire:EMPLOYEE {name: "Claire", role: "CEO"})-[:HAS]->(Steve:EMPLOYEE
{name: "Steve", role: "Head of Sales"})-[:HAS]->(:EMPLOYEE {name: "Nathan",
role: "SalesPerson"}),
(Claire)-[:HAS]->(:EMPLOYEE {name: "John", role: "Head of IT"})-[:HAS]-
>(Hugo:EMPLOYEE {name: "Hugo", role: "Team Lead"})-[:HAS]->(:EMPLOYEE {name:
"Matthew", role: "Developer"}),
(Steve)-[:HAS]->(:EMPLOYEE {name: "Paula", role: "SalesPerson"}),
(:EMPLOYEE {name: "Tim", role: "Customer Support Officer"})<-[:HAS]-(Sarah)-
[:HAS]->(:EMPLOYEE {name: "Roger", role: "Customer Support Officer"}),
(Sarah)-[:HAS]->(:EMPLOYEE {name: "Peter", role: "Assistant"}),
(:EMPLOYEE {name: "Simon", role: "QA Engineer"})<-[:HAS]-(Hugo)-[:HAS]-
>(:EMPLOYEE {name: "Ruth", role: "Developer"})
```

Na kraju dobivamo graf prikazan na slici 3.10



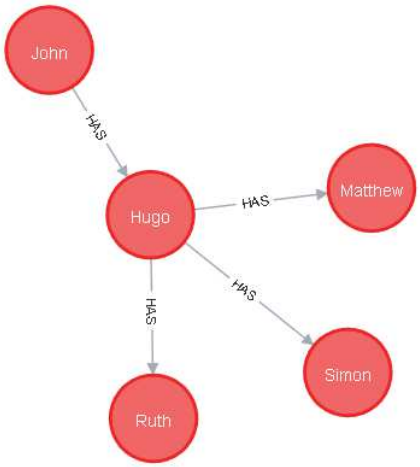
Slika 3.10 Prikaz grafa radnog primjera hijerarhije između zaposlenika

3.3.3. Upiti za pretraživanje stabla

Za razliku od relacijskih baza u graf bazama jednom linijom je moguće dohvatiti hijerarhiju pojedinog čvora, kodom danim u nastavku:

```
MATCH (parent: EMPLOYEE {name:"John"})-[:HAS*]->(child: EMPLOYEE)
RETURN parent,child
```

(parent:Employee {name:"John"})-[:HAS]->(child:Employee):* Početni čvor pretrage je John i pomoću sintakse *–[:HAS*]->* propagiramo traženje u dubinu do dna stabla po vezi tipa HAS. Izlaz ove MATCH klauzule je prikazan na slici 3.11



Slika 3.11 Rezultat pretraživanja čvora John u dubinu, grafička reprezentacija

Za razliku od relacijskih baza Neo4j ima odličnu vizualizaciju podataka samim time puno je lakše održavati podatke unutar baze. U Neo4j kao i u relacijskom modelu moguće je ispis napraviti u tabličnom obliku. Dani ispis dan je na slici 3.12.

"parent"	"child"
{"role":"Head of IT","name":"John"}	{"role":"Team Lead","name":"Hugo"}
{"role":"Head of IT","name":"John"}	{"role":"Developer","name":"Ruth"}
{"role":"Head of IT","name":"John"}	{"role":"QA Engineer","name":"Simon"}
{"role":"Head of IT","name":"John"}	{"role":"Developer","name":"Matthew"}

Slika 3.12 Rezultat pretraživanja čvora John u dubinu, tablična reprezentacija

Dodavanje novog čvora u hijerarhiju:

```
MATCH (parent:EMPLOYEE {name: "Steve"})
MERGE (parent)-[:HAS]->(child:EMPLOYEE {name: "Alex", role:"Salesperson"})
```

Brisanje čvora u hijerarhiji:

```
MATCH (employee:EMPLOYEE {name: "Alex"})
DETACH DELETE employee
```

Prebacivanje zaposlenika u drugi odjel kada se radi o zaposleniku koji je list stabla.

```
MATCH (sarah:EMPLOYEE {name: "Sarah"})-[oldRel:HAS]->(anne:EMPLOYEE {name:
"Anne"})
MATCH (hugo:EMPLOYEE {name: "Hugo"})
DELETE oldRel
MERGE (hugo)-[:HAS]->(anne)
```

U ovom primjeru cilj nam je prebaciti osobu Anne iz support odjela kojem je voditelj Sarah u odjel IT-a pod u tim koji vodi Hugo.

MATCH (sarah:EMPLOYEE {name: "Sarah"})-[oldRel:HAS]->(anne:EMPLOYEE {name: "Anne"}): dolazimo do reference stare veze koju spremamo u varijablu oldRel, budući da Anne više ne radi u odjelu koji vodi Sarah tu vezu brišemo *DELETE oldRel.*

MATCH (hugo:EMPLOYEE {name: "Hugo"}): pronalazimo čvor novog Anne-inog menadžera.

MERGE (hugo)-[:HAS]->(anne): dodavanje veze Anne i njezinog menadžera Huga.

Prebacivanje zaposlenika kada se radi o čvoru koji nisu listovi odnosno ima djecu.

```
WITH "John" AS parentNode, "Hugo" AS nodeToMove
MATCH (john: EMPLOYEE {name:parentNode})
MATCH (hugo: EMPLOYEE {name:nodeToMove})-[childrenRels:HAS*1]->(children:
EMPLOYEE)
MATCH (hugo: EMPLOYEE {name:nodeToMove})<-[parentRels:HAS*1]-(john)
UNWIND childrenRels as rels
UNWIND parentRels as rels2
DELETE rels,rels2
MERGE (john)-[:HAS]->(children);


WITH "Hugo" AS nodeToMove, "Claire" as newParent
MATCH (hugo: EMPLOYEE {name:nodeToMove})
MATCH (claire: EMPLOYEE {name:newParent})
MERGE (claire)-[:HAS]->(hugo)
```

Nazovimo čvor koji prebacujemo *nodeToMove*, njegovu djecu *children*, njegovog roditelja *parentNode* i novog roditelja kojem će *nodeToMove* biti pridodijeljen *newParent.*

Upit se konceptualno sastoji od dva dijela:

- dohvaćanje i brisanje veza između čvora *nodeToMove* s *children* čvorovima i *parentNod-a*. Zatim čvorove *children* pridodjeljujemo *parentNod-u*;
- drugi dio upita je postavljanje novog roditelja *newParent* čvoru *nodeToMove.*

WITH "John" AS parentNode, "Hugo" AS nodeToMove: koristimo kao lokalne varijable upita

UNWIND childrenRels as rels: Budući da MATCH klauzula vrača Listu referenci pomoću UNWIND klauzule transformiramo listu u pojedinačne retke.

DELETE rels,rels2: DELETE klauzuli moramo predati pojedinačne retke, nije moguće predati listu

WITH "Hugo" AS nodeToMove, "Claire" as newParent: budući da se radi o novom upitu nije moguće koristite lokalne varijable iz prošlog upita

4. Usporedba relacijskih modela s modelom iz baza na temelju grafa

U ovom poglavlju usporedit ćemo relacijski s graf modelom. Također ćemo usporediti kompleksnost pisanja upita za radni primjer izrade hijerarhije između zaposlenika.

Prema istraživačkom radu Marka Brice „Usporedba graf i relacijskih baza podataka“ usporedbom modela relacijskog i graf modela dobiveni su sljedeći podatci:

Parametar usporedbe	Relacijski model	Graf Model
Pohrana podataka	Podaci se pohranjuju u fiksne, unaprijed definirane tablice koje se sastoje od stupaca i redaka s povezanim podacima koji se često odvajaju između tablica, što smanjuje učinkovitost upita.	Struktura graf baza podataka sa svojstvom slobodnog indeksa rezultira bržim transakcijama i obradom podataka.
Modeliranje podataka	Model baze podataka se mora razviti modeliranjem i potrebno ga je prevesti iz konceptualnog u logički pa tek onda u fizički model. Sve vrste podataka i izvori podataka moraju bit unaprijed poznati, pa sve promjene zahtijevaju stanku u radu za implementaciju promjena.	Fleksibilan proces modeliranja i model podataka koji rezultira nepostojanjem neusklađenosti između logičkog i fizičkog modela. Vrste podataka i izvori podataka mogu se dodati ili mijenjati u bilo kojem trenutku, što dovodi do dramatično kraćih vremena razvoja i prave agilne iteracije.
Upitni jezik	SQL: Upitni jezik kojim se povećava složenost s porastom broja JOIN operacija potrebnih za povezivanje podataka	Cypher: Nativni upitni jezik za graf baze podataka koji pruža najučinkovitiji i najekspresivniji način za opis povezanosti upitima.
Performanse upita	Performanse obrade podataka padaju sa brojem i dubinom JOIN – ova.	Procesiranje grafa osigurava izvođenje u realnom vremenu, bez obzira na broj ili dubinu odnosa
Podrška za transakcije	Podrška za ACID transakcije koje zahtijevaju poslovne aplikacije za dosljedne i pouzdane podatke.	Zadržava podršku za ACID transakcije zbog dosljednosti i pouzdanosti podataka tokom cijelog vremena

Tablica 4.1 usporedba relacijskog sa graf modelom [3]

Relacijskom model organizira podatke u tablice s unaprijed definiranom shemom. Svaka tablica predstavlja vrstu entiteta, a svaki redak predstavlja jednu instancu tog entiteta. Odnosi između entiteta postižu se primjenom stranih ključeva.

Graf model pohranjuje podatke kao čvorove i veze. Čvorovi predstavljaju entitete, a veze odnose između tih entiteta. Jedna od ključnih prednosti graf modela je da možemo učinkovito pohranjivati i lako pretraživati jako povezane podatke.

Općenito, izbor između graf baze podataka i relacijske baze podataka ovisi o specifičnim zahtjevima aplikacije. Ako su podaci jako povezani i upiti zahtijevaju pretraživanje kroz više odnosa, graf model je bolji izbor. Međutim ako su podaci dobro strukturirani i upiti uključuju složena spajanja i agregacije, relacijska model je bolja opcija.

Model	SELECT	INSERT	UPDATE	DELETE	VIZUALIZACIJA
Popis susjeda	Teško	Lagan	Lagan	Lagan *	Loša
Ugniježđeni skup	Lagan	Teško	Teško	Lagan	Loša
Graf model	Lagan	Lagan	Lagan	Lagan	Odlična

Tablica 4.2. Usporedba složenosti upita između modela

Nasuprot relacijskom modelu, model na temelju grafa puno lakše rješava probleme koji uključuju hijerarhiju. Hijerarhija se može predstaviti stablom koje je zapravo graf. Stoga, za takve probleme bi se trebala koristiti graf baza, jer graf baza nudi fleksibilnost i jednostavnost u radu s hijerarhijskim podacima. Također nudi i odličnu vizualizaciju koja je i sama biti u radu s podatcima u hijerarhiji.

5. Hijerarhija podataka u projektu IoT-polje

5.1. Motivacija

Kao što smo prije naveli baza podataka projekta IoT-polje ostvarena je u relacijskom modelu. Hijerarhija podataka u projektu ostvarena je nizanjem oznaka (eng. Tag) u entitetima scena (eng. Scene). Ovakav model oznaka je prihvatljiv za mali skup no s povećanjem kompleksnosti i međuovisnosti podataka navedeni model više nije moguće optimalno koristiti. Kroz 5. poglavlje implementirat ćemo stablastu hijerarhija između podataka.

U prethodnim poglavljima rada odredili smo prednosti i nedostatke relacijskih i graf baza u poglavlju **4. Usporedba relacijskih modela s modelom iz baza na temelju grafa**.

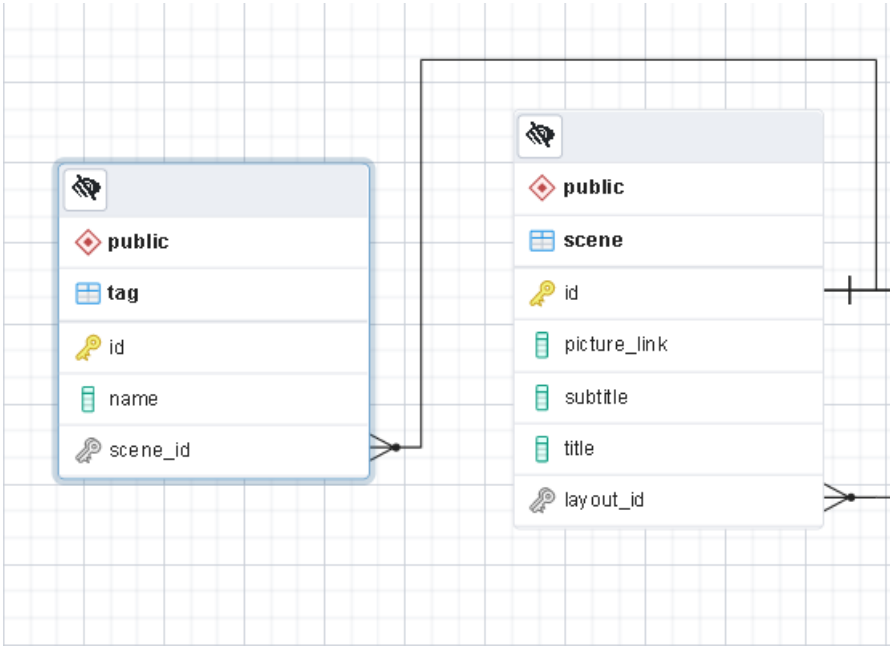
Projekt zahtjeva lako upravljanje, dobru vizualizaciju i lako dodavanje/brisanje elemenata u hijerarhiju baze podataka. Pomoću tablice 4. iz poglavlja **4 .Usporedba relacijskih modela s modelom iz baza na temelju grafa** možemo odrediti koji model nam najviše odgovara. Iz svih navedenih funkcijskih zahtjeva najviše nam odgovara model na temelju grafa. Shodno tome da nećemo koristiti relacijsku bazu u koju koristi projekt morat ćemo posegnuti hibridnoj opciji NoSQL-a s relacijskim modelom. Kako bi hibridni pristup bio učinkovit važno je osigurati sinkronizaciju između njih. To se može postići replikacijom podataka, migracija podataka ili koristeći API integracije. Jedan od popularnih alata i koji omogućuje hibridni pristup grafu i relacijskim bazama je Spring Data Neo4j [6] koji se temelji na JTA *transaction manageru*.

Jedan uobičajenih hibridnih pristupa je korištenje grafa za upravljanje odnosima između entiteta, dok se korištenje relacijske baze podataka koristi za pohranjivanje detaljnih informacija o tim entitetima.

5.2. Model relacijske baze podataka IoT-polja

Kao što smo opisali u poglavlju **5.1 Motivacija** dosadašnji entitet Scene koristi entitet Tag s vezom 1..N. Time ostvaruje da može imati više instanci Tag-ova kako bi se ostvarila

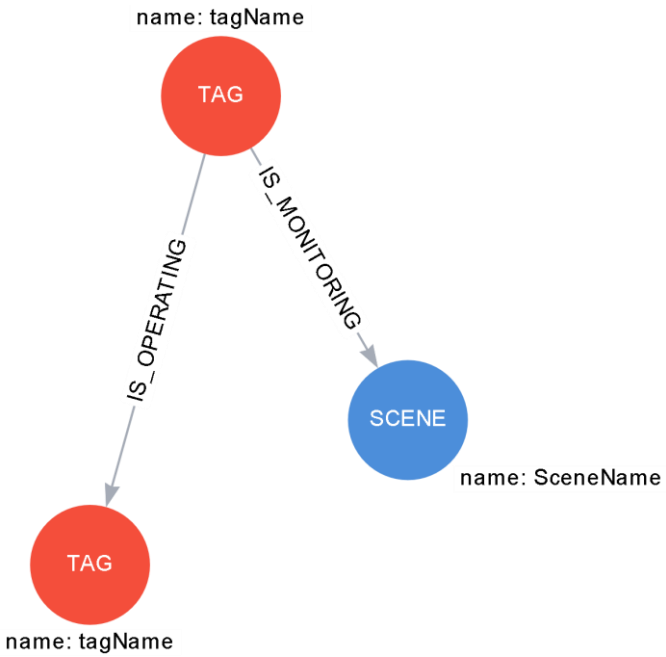
jednostavna hijerarhija. Dio ERD modela projekta koji pokazuje odnose Tag i Scene entiteta prikazan je na slici 5.1.



Slika 5.1 ERD model Tagova i Scene entiteta

Kao što smo vidjeli iz prošlih poglavlja Neo4j nam omogućuje odličnu vizualizaciju te su upiti za modifikaciju hijerarhije su vrlo jednostavni. Budući da je glavni funkcijski zahtjev projekta dobra vizualizacija i lako upravljanje hijerarhijom odlučujemo se za hibridnu opciju baza podataka. Na taj način kombiniramo prednosti oba modela. U relacijskoj bazi ostavljamo podatke s fiksnim i unaprijed određenim tablicama budući da je relacijski model s tim vrstama podataka odlično barata. Baza podataka na temelju grafa koristiti ćemo za spremanje hijerarhije odnosa između entiteta Scena i Tagova.

5.3. Izrada modela grafa u Neo4j za IoT-polje“ u Neo4j



Slika 5.2 model hijerarhije Tagova i Scena

Budući da smo već u poglavlju **3.3. Radni primjer hijerarhije među zaposlenicima** već obradili modeliranje grafa baze u ovom poglavlju samo ćemo proletjeti kroz izradu hijerarhije i naglasiti razlike.

Na slici 5.2 vidimo da hijerarhiju tvore čvorovi tipa Tag te da se čvorovi tipa Scene vežu na tu hijerarhiju. Također primijetimo da između čvorova Tag imamo vezu tipa *IS_OPERATING* te između čvorova Tag i Scene imamo vezu tipa *IS_MONITORING*.

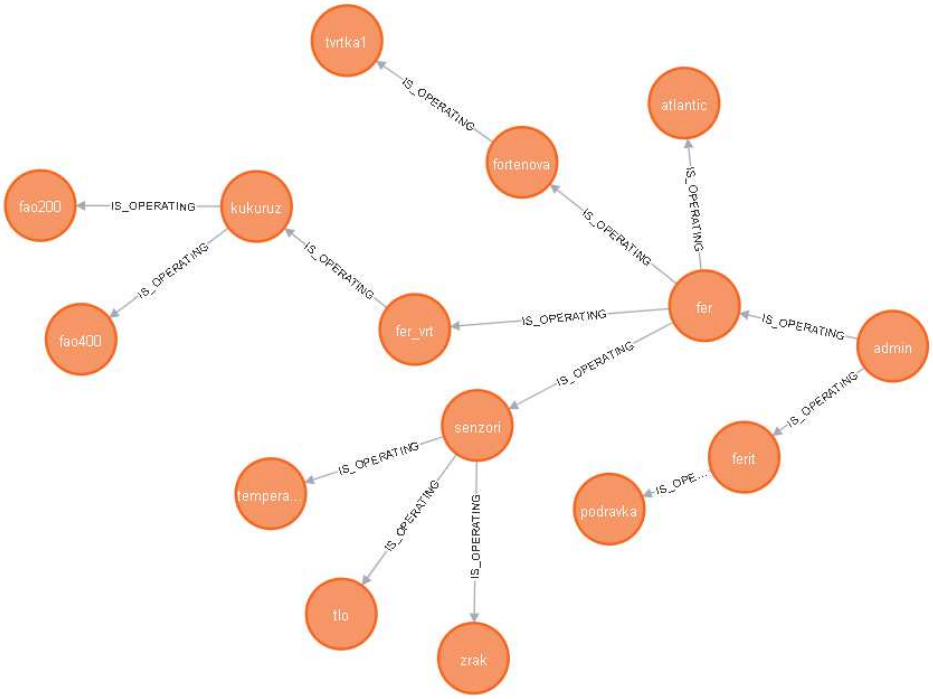
Za stvaranje čvorova tipa Tag koristimo sljedeći Cypher upit:

```
MERGE (n:TAG {name:"tvrtka1"})
```

Za stvaranje veza IS_OPERATING između Tag čvorova koristimo:

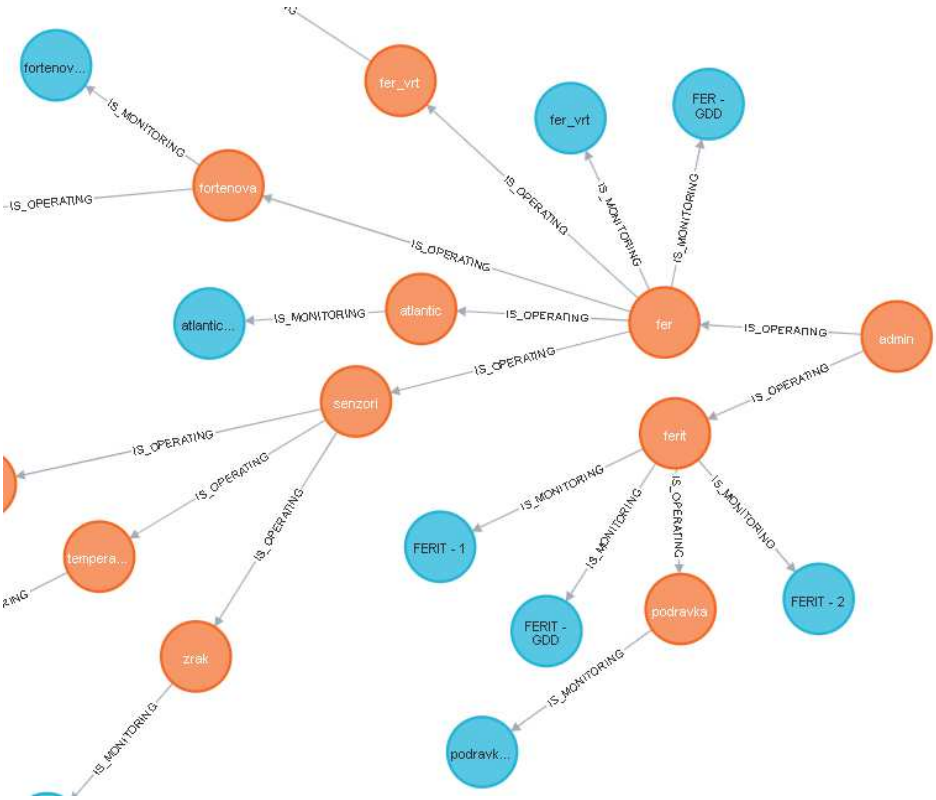
```
MATCH (n1:TAG {name:"admin"})
MATCH (n2:TAG {name:"atlantic"})
MERGE (n1)-[rel:OPERATES]->(n2)
```

Korištenjem navedenih upita dobivamo hijerarhiju koja se nalazi na slici 5.3.



Slika 5.3 hijerarhija između Tagova projekta IoT-polje napravljena u bazi Neo4j

Sada kada smo napravili početnu hijerarhiju temeljenu na čvorova tipa Tag na njih dodajemo čvorove tipa Scene kao djecu Tag čvorovima. To je prikazano na slici 5.4.



Slika 5.4 prikaz dijela hijerarhije projekta IoT-Polje napravljeno pomoću baze Neo4j

Slično kao i prije spajanje čvorova tipa Tag i čvorova tipa Scene sa vezom *IS_MONITORING* napravljen je pomoću sljedećeg upita:

```
MATCH (n1:Tag {name:"fortenova"})
MATCH (n2:Scene {name:"fortenova_vrt"})
MERGE (n1)-[rel:IS_MONITORING]->(n2)
```

U radnog primjera hijerarhije između zaposlenika samo smo imali čvorove tipa *Employee*.

Za razliku od tog primjera ovdje imamo čvorove tipa Tag koji tvore hijerarhiju i čvorove tipa Scene koji se vežu za tu hijerarhiju.

Zbog toga upit za odabir Scena na temelju čvora hijerarhije je sljedeći:

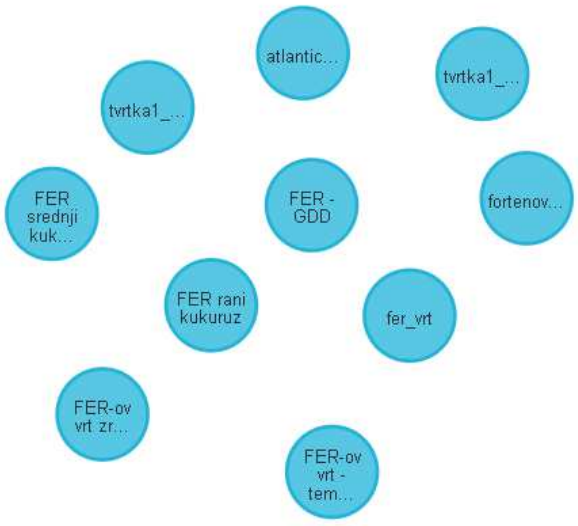
```
OPTIONAL MATCH (parent:Tag {name:"fer"})-[:IS_OPERATING*]->
(child:Tag)-[:IS_MONITORING]->(grandChildren:Scene)

OPTIONAL MATCH (parent2:Tag {name:"fer"})-[:IS_MONITORIN]->
(directChildren:Scene)

RETURN grandChildren,directChildren
```

Prvim MATCH-om tražimo sve Scene kojima je Tag *fer* roditelj

(parent:Tag{name:"fer"})-[:IS_OPERATING]->(child:Tag)-[:IS_MONITORING]-> (grandChildren:Scene)*

Drugim MATCH-om tražimo direktnu djecu Tag-a *fer*.

(parent2:Tag {name:"fer"})-[:IS_MONITORIN]-> (directChildren:Scene)



Slika 5.5 Izlaz upit za odabir svih Scena kojima je Tag fer nadležan

5.4. Integracija baze podataka Neo4j u SpringBootu

Kao što je ranije opisano, za implementaciju baze podataka Neo4j koristit ćemo JTA transaction manager. U Spring projekt dodajemo ga kao ovisnost (Gradle *dependency*).

```
implementation 'org.springframework.boot:spring-boot-starter-data-neo4j'
```

Rest poslužitelj projekta modeliran je klasičnim Controller, Service, Repository obrascem, stoga ćemo u nastavku implementirati Model, Repository, Service i Controller.

5.4.1. Izrada modela

Za izradu modela čvorova i veza između njih koristit ćemo modul dana-neo4j.

Ispod je dan kod za definiranje čvora tipa Tag na temelju modela graf baze danog na slici 5.2. U nastavku ćemo objasnit glavne dijelove koda.

```
@Node(labels = {"Tag"})
public class TagGraph {

    @Id
    @GeneratedValue
    private Long id;

    @Property(name = "name")
    private String tagName;

    @Relationship(type = "IS_OPERATING", direction =
                                            Relationship.Direction.OUTGOING)
    private Set<TagGraph> operates = new HashSet<>();

    @Relationship(type = "IS_MONITORING", direction =
                                            Relationship.Direction.OUTGOING)
    private Set<MonitoringRelationship> monitoring = new HashSet<>();

    public boolean addMonitoring(MonitoringRelationship
                                                isMonitoringRelationship) {
        return monitoring.add(isMonitoringRelationship);
    }
}
```

Pojedine metode poput getX, setX, equals, konstruktora itd. su izostavljene zbog preglednosti. Pomoću *@Node(labels = {"Tag"})* definiramo da će sljedeća klasa predstavljati čvor tipa Tag. Čvoru zatim dodajemo svojstvo name pomoću sljedeće notacije *@Property(name = "name")*. Za definiranje veza između čvorova koristimo sljedeću

notaciju *@Relationship(type = "IS_OPERATING", direction = Relationship.Direction.OUTGOING)* unutar koje definiramo tip notacije te smjer. Budući da neki čvor može imati više veza tipa IS_OPERATING navedene veze spremamo u skup. Primijetimo da skup definiramo tipom TagGraph koji je zapravo klasa čvora Tag.

Ono što bi nas moglo zbuniti je to što veza *IS_MONITORING* za tip podataka u skupu koristi klasu *MonitoringRelationship*, a ne klasu *SceneGraph* analogno prošloj vezi. Razlika je u tome što smo vezi *IS_MONITORING* dodali svojstvo, stoga smo primorani izraditi posebnu klasu *MonitoringRelationship* koja će predstavljati tu vezu zajedno sa svojstvom.

Klasa SceneGraph koja predstavlja čvor tipa Scene implementira se analogno klasi TagGraph stoga ćemo ju preskočiti u ovom razmatranju. Ali zbog cjelovitosti njen kod dajemo u prilogu ovog rada.

U nastavku dajemo kod za izradu veza između čvorova Tag i Scene.

```
@RelationshipProperties
public class MonitoringRelationship {

    @Id
    @GeneratedValue
    private Long id;

    @Property(name = "tbd")
    private String tbd;

    @TargetNode
    private SceneGraph scene;
}
```

Pojedine metode poput getera, setera, equals, constructora itd. su izostavljene zbog preglednosti.

Analogno prošlom primjeru na početku pomoću notacije *@RelationshipProperties* definiramo vezu predstavljenu razredom *MonitoringRelationship*.

Ovoj vezi također dodajemo svojstvo tbd (eng. to be determined). Navedeno svojstvo u ovom momentu projekta nema svrhu ali kasnije će se definirati njegova svrha u smislu pametnije pretrage i izradi višeslojne hijerarhije.

Također na kraju moramo definirati ciljni čvor na koji će veza pokazivati, u ovom slučaju SceneGraph.

5.4.2. Izrada Neo4j Repositorya

Repository definiramo klasično uporabom Spring *@Repository* notacije.

SceneGraphRepository definiramo u nastavku:

```
@Repository
public interface SceneGraphRepository extends Neo4jRepository<SceneGraph,
Long> {

    @Query("MATCH (s:Scene) RETURN s.name")
    List<String> findAllScenesNames();

    @Query( "OPTIONAL MATCH (parent:Tag {name:$tagName})-[:IS_OPERATING*]->
             (child:Tag)-[:IS_MONITORING]->(grandChildren:Scene)\n" +

             "OPTIONAL MATCH (parent2:Tag {name:$tagName})
              -[:IS_MONITORING]->(children:Scene)\n" +

             "RETURN COLLECT( distinct scene.name), COLLECT( distinct
                                                          novo.name)"
    )
    List<String> findAllScenesByTagsName(String tagName);

    SceneGraph findBySceneName(String name);
}
```

Neo4jRepository je sučelje koji dolazi iz programskog okvira Spring Data Neo4j, i pruža osnovne CRUD (create, read, update, delete) operacije za entitete koji se čuvaju u graf bazi podataka Neo4j. Proširivanjem sučelja Neo4jRepository, SceneGraphRepository nasljeđuje CRUD metode za upravljanje entitetimom SceneGraph.

Također u SceneGraphRepositoryu definiramo upite koje koristimo kasnije u servisima.

Od značaja nam je upit predstavljen metodom:

List<String> findAllScenesByTagsName(String tagName). Radi se o modifikaciji upita kojeg smo predstavili u poglavlju **5.3. Izrada modela grafa u Neo4j za IoT-polje“ u Neo4j**. Upit vraća listu Scena za kojeg je nadležan dani Tag kao argument.

Razlika između upita iz poglavlja 5.3 i ovog je to što ovdje vraćamo listu imena pronađenih čvorova dok smo prije vraćali listu objekata čvorova tipa Scene.

TagGraphRepository definiramo slično, kod je dan u nastavku:

```
@Repository
public interface TagGraphRepository extends Neo4jRepository<TagGraph, Long>
{

    @Query("MATCH (t:Tag) RETURN t.name")
    List<String> findAllTagsNames();

    TagGraph findByTagName(String name);

    List<TagGraph> findByTagNameIn(List<String> tags);

    @Query("MATCH (tracker:Tag) " +
            "WHERE tracker.name IN $tagNames " +
            "WITH tracker " +
            "OPTIONAL MATCH (tracker)-[r_issued_by:IS_OPERATING]->
                                                (organization:Tag) " +
            "WITH tracker, COLLECT(r_issued_by) AS organization_rels,
             COLLECT(DISTINCT organization) AS organizations " +
            "OPTIONAL MATCH (tracker)-[r_assigned_to:IS_MONITORING]->
                                                    (asset:Scene) " +
            "RETURN tracker, organization_rels, organizations,
                    COLLECT(r_assigned_to) AS asset_rels,
                    COLLECT(DISTINCT asset) AS assets")
    List<TagGraph> findByTagNames(List<String> tagNames);
}
```

U TagGraphRepository od značaja nam je upit dan metodom *List<TagGraph> findByTagNames(List<String> tagNames);* Koji nam na temelju dane liste imena tagova kao argument vraća pronađenih objekata Tagova.

Digresija, naizgled ovaj upit izgleda glupo i prekomplicirano s obzirom na njegovu zadaću. Treba pronaći listu TagGraph objekata na temelju tagNames liste.
Upitom u nastavku jednostavno dolazimo do objekata TagGraph:

```
WITH ["fer", "ferit"] AS tags
MATCH (tag:Tag)
WHERE tag.name IN tags
RETURN COLLECT(tag)
```

Ovim upitnom uistinu dolazimo do liste objekata TagGraphs, ali mu veze IS_MONITORING i IS_OPERATING spremljene u skupove *monitorning* i *operating* nisu povučene.
Drugim riječima implementacija Neo4jRepositorya povlačenjem objekta iz baze ne povlači i njegove veze.

Ukoliko bismo pomoću *repositoy.save()* pokušali ažurirati stablo zbog veza koje su null SpringBoot bi shvatio to kao da mi želimo pobrisati veze tamo gdje je skup *null* . Samim time sa svakim pozivom *repository.save()* naše stablo bi se raspadalo.

Rješenje možemo pronaći u JPA Provided and Derived Methods koju generira sam SpringBoot na temelju imena metode., radi se o metodi:

List<TagGraph> findByTagNameIn(List<String> tags);

Navedena metoda rekurzivno do dna stabla pobunjuje sve objekte i veze za tražene TagGraph čvorove.

Na kraju poglavlja bit će raspravljeno kako efikasnije riješiti navedeni problem.

5.4.3. Servisi i transakcije hibridne baze

Na početku ovog poglavlja odlučili smo se za hibridni model baza podatak stoga se moramo pobrinuti da se transakcije izvode na obje baze podataka te ako dođe do povrata u prijašnje stanje (eng. rollback) jedne transakcije to se treba propagira i na drugu bazu podataka.

U prvom dijelu objasnit ćemo servis za relacijsku bazu te zatim servis za graf bazu.

Stoga za početak doradimo servis relacijske baze podataka:

```
@Service
public class SceneService {

    @Autowired
    private SceneRepository sceneRepository;

    @Autowired
    private GraphService graphService;

    @Transactional
    public Scene addScene(Scene scene) {
        if (!sceneRepository.existsById(scene.getId())) {

            graphService.addToNeo4J( scene );

            return sceneRepository.save(scene);
        }
        throw new NoSuchElement("Scene "+ scene.getId()+ "already exists);
    }

    @Transactional
    public Scene editScene(Long id, Scene newScene) {
```

```
        if (sceneRepository.existsById(id)) {
            Scene oldScene = sceneRepository.findById(id).get();
            graphService.modifyToNeo4J( newScene, oldScene);
            return sceneRepository.save(newScene);
        }
        throw new NoSuchElement("Scene " + id + " does not exists!");
    }

    @Transactional
    public Scene deleteSceneById(Long id) {
        Scene scene = this.fetch(id);

        if(scene == null)
            throw new NoSuchElement("Scene " + id + " does not exists!");

        graphService.deleteFromNeo4J(scene);
        sceneRepository.delete(scene);

        return scene;
    }

    public List<Scene> getScenesByTagName(String tagName) {
       List<String> scenesNames = graphService.getScenesByTagName(tagName );
        return sceneRepository.getScenesByNames( scenesNames );
    }
}
```

Prikazane su samo varijable i metode od interese zbog ograničenog prostora za prikaz.

Na početku pomoć *@Autowired* anotacije injektiramo zavisnosti SceneRepository-a u varijablu sceneRepository i GraphService u varijablu graphService.

Pomoću notacije *@Transactional* iznad metoda osiguravamo da će se transakcija provesti ako se uspiju provesti sve naredbe u metodi.

Pogledajmo sada za primjer metodu *addScene*, ona kao svoj argument dobiva objekt Scene kojeg daje metodama *graphService.addToNeo4j* i *sceneRepository.save*.

Pomoću metode addToNeo4j navedenu scenu spremamo u Neo4j bazu podataka, dok pomoću *sceneRepository.save* istu scenu spremamo u relacijsku bazu podataka.

Specifičnosti metode addToNeo4j objasnit ćemo kasnije.

Metode *editScene* i *deleteSceneById* funkcioniraju na isti način kao i prethodno opisana metoda stoga ih nećemo ponovno objašnjavati.

Od velike zanimljivosti nam je metoda *List<Scene> getScenesByTagName(String tagName)*. Metoda vraća listu scena koja je nalaze ispod u hijerarhiji danog Taga kao argumenta metode.

Pogledajmo sada njenu implementaciju. Pozivom *graphService.getScenesByTagName (tagName)* iz neo4j baza podataka dobivamo listu imena scena koje se nalaze u hijerarhiji danog taga. Budući da sada imamo imena danih scena sada nam je jednostavno doći do objekata tih scena iz relacije baze podataka pozivom metode repozitorija *sceneRepository.getScenesByNames(scenesNames)* koja je implementirana na sljedeći način:

```
@Query(value="SELECT * FROM SCENE WHERE SCENE.title IN ?1",
                                                    nativeQuery = true)
    List<Scene> getScenesByNames(List<String> scenesNames);
```

Da rekapituliramo preko baze Neo4j dobivamo imena scena kojima je dani tag nadležan. Kasnije na temelju tih imena jednostavnim SQL upitom dohvaćamo objekte tih scena.

Na ovaj način iskorištavamo prednost baze podataka Neo4j koja lako barata s hijerarhijama i relacijskom bazom koja efikasno dohvaća strukturirane podatke.

Budući da posao traženja scena na temelju hijerarhije obrađuje neo4j baza podataka naša originalna PostgreSQL baza nije preopterećena. Njena glavna zadaća je samo dohvat podataka, dok zadaća neo4j baze je rekurzivno traženje imena scena.

Sada prođimo kroz glavne metode GraphService servisa:

```
    public boolean addToNeo4J(Scene scene) {
        SceneGraph sceneGraph;

        //Provjeri da li scena već postoji
       sceneGraph = sceneGraphRepository.findBySceneName( scene.getTitle());

        //Ako ne postoji napravi je
        if (sceneGraph == null) {
            sceneGraph = new SceneGraph( scene.getTitle() );
            sceneGraphRepository.save( sceneGraph );
        }
        this.saveToNeo4j( scene, sceneGraph );
        return true;
    }
```

Pomoću sceneGraphRepositorya spremamo scenu u graf bazu te zatim zovemo metodu saveToNeo4j koja je dana u nastavku:

```
    private void saveToNeo4j(Scene scene, SceneGraph sceneGraph) {
        //Tagovi za novu scenu
        List<String> tags = scene.getTags();

        //Ako već neki tagovi postoje u graf bazi dohvati ih
        List<TagGraph> selectedTags = tagRepository.findByTagNameIn( tags );

        //Takovi koje nisi upio pronaći napravi ih
        for (String tag : tags) {
            if (selectedTags.stream().noneMatch( tagGraph ->
                                tagGraph.getTagName().equals( tag ) )) {
                TagGraph tagGraph = new TagGraph( tag );
                selectedTags.add( tagGraph );
            }
        }

        //Dodavanje relaciaj između tagova i scena
        for (TagGraph tagGraph : selectedTags) {
            MonitoringRelationship monitoring = new
                                                MonitoringRelationship();
            monitoring.setScene( sceneGraph );
            tagGraph.addMonitoring( monitoring );
        }

        //Spremanje – modifikacija tagova
        for (TagGraph tagGraph : selectedTags) {
            tagRepository.save( tagGraph );
        }
    }
```

Scene predstavlja objekt koji je spremljen u relacijskom modelu, dok *SceneGraph* predstavlja objekt kojeg spremamo u graf model bazu podataka. S*cene* objekt kojeg dobivamo preko POST request controllera propagiranog preko *addScene*, *addToNeo4j* i na kraju *saveToNeo4j*. Navedeni objekt koristimo kako bismo iz njega izvadili listu imena tagova. Na temelju tih tagova spajamo naš novostvoreni čvor *SceneGraph*.

Još jedna metoda od značaja u *SceneService* je *deleteTag* koja briše tag u neo4j i relacijskoj bazi podataka.

```
    @Transactional
    public TagGraph deleteTag(String tagName) {
        TagGraph tagGraph = tagRepository.findByTagName( tagName );
        if (tagGraph == null) {
            return null;
        }
```

```
        tagRepository.delete( tagGraph );
        sceneRelationalRepository.deleteTagFromScenes( tagName );

        return tagGraph;
    }
```

Naravno ako se pobriše čvor taga koji nije list stablo će se raspasti, ali za to su dodane metode *addTag* i *connectTag* koje nam omogućuju dodavanje nove veze u hijerarhiji. Njihove implementacije nalaze se u prilogu rada.

5.4.4. Controller i krajnje točke

Kontroler je napravljen na temelju REST arhitektonskog stila tj. koristimo HTTP metode GET, POST, PUT, DELETE za izvršavanje akcija nad resursima i vraćamo odgovarajuće odgovore HTTP.

U ovom potpoglavlju prikazat ćemo samo neke metode koje su nam od interesa.

Prva od njih je *addEditScene* koja je vezana na krajnju točku (*endpoint*) */scene*. Navedena metoda dodaje novu scenu u našu bazu podataka. Metoda iz tijela HTTP zahtjeva preko *objectMappera* i DTO (eng. *Data Trensfer Object*) objekata izrađuje objekt *Scene* relacijskog modela. Kasnije se taj objekt propagira u metode *addScene*, *addToNeo4j* i *saveToNeo4j* kao što smo u prošlom potpoglavlju opisali.

```
@PostMapping("/scene")
    public ResponseEntity<?> addEditScene(@RequestBody String model) throws
JsonMappingException, JsonProcessingException {
        ...
        sceneDTO = objectMapper.readValue(model, sceneDTO.getClass());
        ...
        Scene scene = new Scene(sceneDTO);
        ...
        service.addScene(scene);
    }
```

Druga značajan metoda je *getScenesByTagNames* koja ne temelju danog imena taga vraća scene kojima je taj *tag* hijerarhijski nadređen.

```
@GetMapping("/scenes/tag/{tagName}")
    public ResponseEntity<List<Scene>>
              getScenesByTagName(@PathVariable("tagName") String tagName) {
        List<Scene> list = service.getScenesByTagName(tagName);
        return ResponseEntity.status(HttpStatus.OK).body(list);
    }
```

6. Zaključak

Baze podataka na temelju grafova lako rješavaju problem hijerarhije naspram relacijskih baza zbog mogućnosti direktnog modeliranja hijerarhije kao stablo. U relacijskom modelu razvili smo dva modela za ostvarivanje hijerarhije: model popisa susjeda i model ugniježdenih čvorova. Napravili smo njihovu usporedbu i zaključili da je u modelu popisa susjeda lako dodavati i brisati elemente, ali da je problem pretraživati stablo. Dok je u modelu ugniježdenih čvorova problem dodavati nove elemente, a lako je pretraživati postojeću hijerarhiju.

Također isti problem smo riješili u graf modelu i ostvarili lako pretraživanje, dodavanje i brisanje elementa u stablu hijerarhije. Uz to korištenjem programskog paketa Neo4J Desktop dobili smo odličnu vizualizaciju samih podataka što nam olakšava upravljanje hijerarhijom.

Ostvarena je hibridna opcija između baza podataka PostgreSQL i Neo4J koje zadržavaju ACID svojstva u projektu IoT-polje. Na taj način odvojili smo hijerarhiju od relacijskog modela i prebacili ga na graf model, dok podatke još uvijek spremamo i upravljamo od strane relacijskog modela. Time smo PostgreSQL oslobodili skupih upita filtriranja i upravljanja hijerarhijom i to prebacili na bazu podataka Neo4J koja takve upite izvršava brzo. Također time smo spriječili da ne dođe do zagušenja baze podataka PosgreSQL.

U ovom radu koristio se Spring Data Neo4j transaction manager, ali kao što smo vidjeli u poglavlju **5.4.3 Servisi i transakcije hibridne baze** navedeni paket na povlači sve reference čvorova djece što dovodi do razbijanja veza kod ažuriranja čvorova. Kao nastavak ovog rada mogu se istražiti alternativi transactional manageryi. Također mogu se provesti testiranja performansi nad ovako hibridno ostvarenom sustavu.

7. Literatura

[1] Brumm B., *Hierarchical Data in SQL*, Databasestar, (2022, rujan). Poveznica: https://www.databasestar.com/hierarchical-data-sql/; pristupljeno 8. ožujka 2023.

[2] Robinson I., Webber J., Eifrem E. *Graph Databases*. 2. izdanje. O'reilly, 2015.

[3] Brica, M. *Usporedba graf i relacijskih baza.* Diplomski rad. Sveučilište Josipa Jurja Strossmayera u Osijeku Fakultet elektrotehnike, računarstva i informacijskih tehnologija, 2018.

[4] Neo4j, Graph Data Science Library, (2023). Poveznica: https://neo4j.com/docs/graph-data-science; pristupljeno 2. svibnja 2023.

[5] Neo4j, Example Datasets, (2023). Poveznica: https://neo4j.com/developer/example-data/; pristupljeno 1. svibnja 2023.

[6] Neo4j, Spring Dana Neo4j, (2023). Poveznica: https://neo4j.com/docs/getting-started/languages-guides/java/spring-data-neo4j/; pristupljeno 12. travnja 2023.

[7] Neo4j, Pandora Papers, (2021). Poveznica: https://neo4j.com/developer-blog/exploring-the-pandora-papers-with-neo4j/; pristupljeno 25. travnja 2023.

[8] MemGraph, Graph modeling, (2023). Poveznica: https://memgraph.com/docs/memgraph/tutorials/graph-modeling; pristupljeno 24. travnja 2023.

[9] Disney A. Guide to creating graph data models, Cambridge Intelligence, (2020, siječanj). Poveznica: https://cambridge-intelligence.com/graph-data-modeling-101/; pristupljeno 4. svibnja 2023.

[10] Woodie A., Graph Databases Everywhere, Datanami (2015). Poveznica: https://www.datanami.com/2015/01/15/graph-databases-everywhere-2020-says-neo4j-chief/ pristupljeno 10. svibnja 2023.

[11] Sasaki Merkl B., Chao J. *Graph Databases for Beginners*. 1. izdanje. Neo4j, 2021.

[12] Weston G., IoT Connectivity Industry Forecast By 2030, (2023.). Poveznica: https://101blockchains.com/iot-connectivity-industry-forecast/; pristupljeno: 28. ožuljka 2023.

[13] Brumm B. *Beginning Oracle SQL for Oracle Database 18c*, 1. izdanje. Apress 2019.

8. Sažetak

Podaci Interneta stvari označeni oznakama koje su organizirane u hijerarhiju

Sažetak:

U ovom radu predstavljena je problematika hijerarhije među podatcima. Uz to objašnjeni su najčešći modeli ostvarivanje hijerarhije u SQL domeni relacijskih baza te također u domeni NOSQL-a na modelu grafa. U radu su uspoređeni navedeni modeli prema zahtjevima kompleksnosti pisanja upita, vizualizaciji te težini implementacije.

Kroz rad napravili smo uvod u Neo4j grafičku bazu podataka te u Cypher upitni jezik.

Kroz radni okvir SpringBoot napravljena je integracija Neo4j baze podataka s postojećim projektom IoT-Polje i dodatim uslugama koje su razvijene u sklopu projekta R.

Kroz rad razvijeno je hibridno rješenje hijerarhije koje ne ovisi o postojećem dizajnu i implementaciji spremanja podataka. Drugim riječima ostvarili smo rješenje hijerarhije koje se s lakoćom povezuje s postojećom relacijskom bazom podataka.

Ključne riječi: hijerarhija, Neo4j, Cypher, SpringBoot, hibridna baza

9. Summary

IoT data labeled with tags that are organized into a hierarchy

Summary:

In this paper, the issue of hierarchy among data is presented. In addition, the most common models for realizing hierarchy in the SQL domain of relational databases and also in the domain of NOSQL on the graph model are explained. The paper compares the mentioned models according to the requirements of the complexity of query writing, visualization and the difficulty of implementation.

Through the work, we made an introduction to the Neo4j graphic database and the Cypher query language.

Through the SpringBoot framework, the Neo4j database was integrated with the existing IoT Polje project and additional services, which were developed as part of the R project.

Through the work, a hybrid solution of the hierarchy was developed, which does not depend on the existing design and implementation of data storage. In other words, we have created a hierarchy solution that is easily integrated with existing relational database.

Keywords: hierarchy, Neo4j, Cypher, SpringBoot, hybrid database

10. Prilozi

```
CREATE TABLE employee (
  id NUMBER(5),
  first_name VARCHAR2(50),
  last_name VARCHAR2(50),
  manager_id NUMBER(5),
);
```

SQL kod za izrada tablica modela popisa susjeda

```
CREATE TABLE employee (
  id NUMBER(5),
  first_name VARCHAR2(50),
  manager_id NUMBER(5),
  left_val NUMBER(5),
  right_val NUMBER(5),
);
```

SQL kod za izradu tablica modela ugniježdenih čvorova

```
CREATE
       (nA:Node {name: 'A'}),
       (nB:Node {name: 'B'}),
       (nC:Node {name: 'C'}),
       (nD:Node {name: 'D'}),
       (nE:Node {name: 'E'}),

       (nA)-[:REL]->(nB),
       (nA)-[:REL]->(nC),
       (nB)-[:REL]->(nE),
       (nC)-[:REL]->(nD)
```

Inicijalizacija Neo4J baze podataka demonstracijskog primjera BFS-a, slika 3.6.

```
@Node(labels = {"Scene"})
public class SceneGraph {

    @Id
    @GeneratedValue
    private Long id;
```

```
    @Property(name = "name")
    private String sceneName;

    public SceneGraph() {
    }

}
```

Model izrade čvora Scene u SpringBootu pomoću Spring Neo4j paketa

```
    public TagGraph addTag(String tagName) {
        TagGraph tagGraph = tagRepository.findByTagName( tagName );
        if (tagGraph == null) {
            tagGraph = new TagGraph( tagName );
            tagRepository.save( tagGraph );
        }
        return tagGraph;
    }
```

Metoda za dodavanje novog čvora u Neo4J bazu podataka

```
    public TagGraph connectTags(String tagFrom, String tagTo) {
        TagGraph tagGraphFrom = tagRepository.findByTagName( tagFrom );
        TagGraph tagGraphTo = tagRepository.findByTagName( tagTo );

        if (tagGraphFrom == null || tagGraphTo == null) {
            return null;
        }

        tagGraphFrom.addOperates( tagGraphTo );
        tagGraphTo.addOperates( tagGraphFrom );

        tagRepository.save( tagGraphFrom );
        tagRepository.save( tagGraphTo );

        return tagGraphFrom;
    }
```

Metoda za spajane postojećih čvorova Tag u hijerarhiju