Departamentul Automatică şi Informatică Industrială
Facultatea Automatică şi Calculatoare
Universitatea Națională de Știință și Tehnologie POLITEHNICA
din Bucureşti

LUCRARE DE DIPLOMĂ

Analiză comparativă a bazelor de date noSQL de tip graf față de bazele de date relaționale pe bază de metrici de performanță. Studiu de caz: aplicație de navigare într-o clădire

Coordonator
Sl. Dr. Ing. Adriana Olteanu

Absolvent
Elena Lupu

2024

Cuprins

1. Introducere

Bazele de date reprezintă o componentă esențială în cadrul aplicațiilor software moderne. Primele astfel de sisteme au apărut în anii ’70, urmând ca acestea să evolueze până în prezent.

Prima categorie care se utilizează în această lucrare este cea a bazelor de date relaționale. După cum se afirmă și în [3], popularitatea acestora este datorată abilității de a stoca și gestiona o cantitate foarte mare de date în mod eficient, în formă tabelară, în care fiecare rând reprezintă un obiect, iar fiecare coloană reprezintă un atribut pentru fiecare obiect înregistrat. De asemenea, se concentrează pe asigurarea consistenței și integrității informației. O altă caracteristică ce a stat la baza succesului a fost impunerea limbajului standardizat SQL la nivelul tuturor bazelor de date relaționale existente pe piață.

Cu toate acestea, bazele de date relaționale au întâmpinat în ultimii ani probleme în ceea ce privește capacitatea de modelare a datelor, dupa cum observă și autorii din [2]. Una din acestea o reprezintă creșterea exponențială a cantității de date, care sunt stocate cu precădere pe sisteme distribuite, asigurate de serviciile cloud, pentru a putea face față cererilor masive ale utilizatorilor. Un alt aspect important îl reprezintă interdependențele datelor. Cu toate că acestea se pot reprezenta și ca interdependențe între tabele cu ajutorul cheilor străine, performanțele încep să scadă atunci când avem de interogat mai multe tabele odată pentru obținerea unui set de date.

Bazele de date noSQL sunt o categorie relativ nouă. Așa cum se precizează în [2], conceptul a luat naștere în anii ’90, însa abia la finalul anilor 2000 s-a putut face o clasificare clară a tipurilor de astfel de baze de date: graf, cheie-valoare, familie de coloane și orientat-document. Așa cum sugerează și numele, sunt sisteme care nu folosesc limbajul SQL pentru interogare și care au ca obiectiv păstrarea și gestionarea datelor în formate alternative, mai potrivite pentru cerințele diversificate ale industriei.

O altă caracteristică majoră pe care o au aceste noi sisteme o reprezintă aderarea la un nou set de principii. Orice bază de date relațională se știe că aderă la princpiile ACID (*Atomic, Consistent, Isolated, Durable*), care enunță faptul că fiecare tranzacție este atomică, se execută complet ori deloc, nu trebuie să afecteze integritatea datelor sau alte tranzacții, și că datele trebuie să fie persistente într-o bază de date în urma unei tranzacții. Odată cu evoluția sistemelor distribuite și a noilor modalități de reprezentare, stocare și gestionare a datelor, a luat naștere și teoria CAP (*Consistency, Availability, Partition tolerance*). Aceasta enunță principiile la care se dorește să adere orice bază de date: operațiile să fie consistente, sistemul trebuie să fie disponibil în orice moment pentru a deservi o cerere și trebuie să suporte distribuirea pe mai multe rețele, fără pierderi în comunicare, dar specifică faptul că nicio bază de date nu poate acoperii toate cele 3 principii simultan [4]. Bazele de date relaționale se consideră, conform acestei teoreme, de tipul CA, asigurând consistența și disponibilitatea. Cele noSQL pot fi ori CP ori AP, ele asigurând toleranța la partiționare între rețele, dar în detrimentul fie a disponibilității fie a consistenței, fie chiar de tip CA, asemenea celor relaționale. În urma acestui compromis, proprietățile ACID devin dificil de îndeplinit, așa că s-a întocmit un set de principii mai flexibile, ce pot fi acoperite de bazele de date noSQL, abreviat BASE (*Basically Available,*

Soft state, Eventually consistent). Acesta specifică faptul că un sistem trebuie să fie disponibil în cea mai mare parte din timp și să fie capabil să se recupereze cât mai repede în urma unei perioade de indisponibilitate, trebuie să suporte schimbările datelor în urma proceselor în desfășurare fără pierderi sau chiar coruperea datelor și trebuie să fie consistent atunci când nu mai primește cereri (nu neapărat instant, așa cum obligă ACID).

În lucrarea de față, mă voi concentra asupra bazelor de date noSQL de tip graf. Consider că merită un studiu aprofundat, deoarece, spre deosebire de celelalte tipuri de baze de date noSQL, acestea nu sunt atât de cunoscute publicului larg, cu toate că sunt folosite în domenii diverse, unde există foarte multe relații de interdependență între datele, ele modelându-se natural sub formă de graf. De asemenea, ele au atras atenția și comunității științifice, fiind redactate în cursul anilor o mulțime de articole științifice în care se evidențiază avantajele, dar și limitările acestora în comparație cu bazele de date relaționale.

În capitolele următoare voi detalia procesul prin care fac o comparație între acestea și bazele de date relaționale, folosind ca studiu de caz o aplicație comună a grafurilor, un sistem de navigare într-o clădire.

În secțiunea următoare voi prezenta domeniile în care se utilizează grafurile și cum sunt folosite acestea pentru stocarea datelor și voi aduce și exemple din industrie în acest sens, punând accentul pe cele relevante studiului meu de caz. În secțiunea 3 voi detalia pașii de lucru și criteriile de evaluare pe baza cărora voi face comparația. În secțiunea 4 voi oferi detalii cu privire la tehnologiile software folosite, apoi voi prezenta în secțiunea 5 rezultatele obținute. În cele din urmă, în secțiunea 6, voi trage concluziile și voi enumera posibilitățile de dezvoltare ulterioare.

2. Prezentarea domeniului din care face parte lucrarea

Bazele de date de tip graf sunt o categorie de baze de date noSQL care, după cum sugerează și numele, păstrează datele în structuri de tip graf. Se renunță la structura tabelară, optându-se pentru reprezentarea prin noduri și arce direcționate.

Cel mai comun model de graf utilizat pentru reprezentarea datelor, conform [1], este modelul grafului cu proprietăți etichetate, în care atât arcele cât și nodurile au câte un identificator unic și pot avea proprietăți stocate sub formă cheie-valoare. Mai mult, nodurile și arcele pot avea etichete prin care se poate face distincția între rolurile fiecărui element în graful respectiv. Spre exemplu, se poate modela harta unui oraș într-un graf considerând obiectivele de interes noduri și drumurile între ele arce bidirecționale. Pentru a clasifica obiectivele se poate creea un set de etichete care să definească tipul fiecărui nod: obiectiv turistic, clădire administrativă sau bloc de locuințe... Pentru a păstra alte detalii referitoare fie la drum, cum ar fi ponderi, fie la clădiri, se pot adăuga nodurilor (sau arcelor) proprietați suplimentare.

Mai există un model utilizat în practică, numit Resource Description Framework (abreviat și ca RDF). Acesta este un standard W3C pentru schimbul de date pe Internet. Numit și „Stocare triplă” (*Triplestore*), modelul presupune reprezentarea datelor ca și piese de cunoaștere într-o formă semantică, ca și fapte, prin relații de tipul Subiect-predicat-obiect, noile piese de cunoaștere fiind apoi deduse prin inferențe pe baza grafului existent. Un exemplu de graf ar putea fi o rețea socială, în care fiecare nod reprezintă o persoană, iar un arc poate fi de tipul „Este prieten cu”. Astfel se obțin relații de felul „Prietenul X **Este prieten cu** prietenul Y”. Acest model este folosit cu precădere în aplicațiile în care se lucrează cu seturi de date complexe și care au nevoie să facă statistici pe baza lor, interogările fiind mai eficiente în această formulă.

Pentru interogare s-au dezvoltat mai multe limbaje, fiecare adaptat pentru diverse tipuri de aplicații. Cel mai folosit este considerat a fi Cypher, care este un limbaj declarativ conceput pentru interogarea grafurilor structurate pe modelul proprietăților etichetate și care folosește artă ASCII pentru ilustrarea nodurilor și relațiilor căutate într-un graf, acesta bazându-se pe potriviri da șabloane pentru a întoarce un rezultat. Următorul cel mai popular limbaj este Gremlin, care este tot un limbaj declarativ, orientat spre grafurile cu proprietăți etichetate, dar care se concentrează pe procesul de parcurgere al grafului pentru a returna un rezultat. Mai este, de asemenea, folosit și SPARQL, un alt limbaj, de aceasta dată orientat spre interogarea grafurilor structurate pe model RDF, dar și multe altele.

2.1. Domenii de utilizare

Grafurile și-au găsit utilizarea în practică acolo unde în mod natural datele se modelează în acest fel. În această secțiune voi enumera o parte dintre domeniile în care sunt utilizate:

2.1.1. Aplicații de navigare

Această categorie cuprinde aplicații folosite la scară largă. Fie că vorbim de navigatoare la nivel global, cum ar fi cele integrate pe mașini sau disponibile ca aplicații pentru mobile, sau de sisteme concepute special pentru navigarea în spații interioare, aplicațiile de navigare folosesc pentru calculul traseelor algoritmii de navigare prin grafuri, cum ar fi Dijkstra sau A*.

Cel mai natural mod de a reprezenta o problemă de navigare este prin intermediul grafurilor. O locație se poate reprezenta ca nod, iar traseul prin arce, urmând a se calcula traseul folosind algoritmi standard de navigare.

Pe lângă algoritmii deosebit de eficienți existenți pentru calcularea traseelor, grafurile pot reține și proprietăți ale locațiilor și traseelor, cum ar fi ponderi sau alte detalii folosite la calcularea ponderilor, ceea ce face ca acest mod de reprezentare să fie unul suficient de versatil pentru a acoperi toate nevoile unui sistem de navigare.

Despre acest tip de aplicații urmează, de asemenea, să detaliez în subcapitolul **2.2**, acesta fiind și domeniul în care am ales să îmi dezvolt studiul de caz.

2.1.2. Soluții pentru detecția de fraudă

Fraudele sunt forme de infracțiuni care pot conduce la pierderi financiare masive în cadrul unei firme. Din acest motiv, cererea este tot mai mare pentru soluții software care să prevină această situație.

Grafurile în acest caz reprezintă soluția cea mai potrivită, deoarece se pot identifica ușor șabloanele suspecte prin urmărirea în timp real a legăturilor care se formează între nodori, cum ar fi activitățile repetate, prelungite sau orice altă formă de activitate ieșită din comun [5]. Mai mult, aceste baze de date pot fi integrate cu sistemele de Învățare Automată pentru a spori performanțele prin identificarea mai rapidă a activității frauduloase.

De exemplu, o strategie simplă pentru detectarea spălării de bani poate consta în implementarea unui graf pe baza tranzacțiilor ce se efectuează între diverse entități, care au câte un set de date personale (adresă de e-mail, parolă, domiciliu...), iar apoi, prin interogarea bazei de date, să se identifice șabloanele în formă de inele care pot astfel evidenția cum circulă în realitate banii.

2.1.3. Analize de tip 360-Degree Customer View

Conceptul se referă la stocarea tuturor informațiilor despre un client, de la date de bază, cum ar fi nume, vârstă, adresă de e-mail sau telefon până la interacțiunile pe care le are cu un furnizor de servicii, aici fiind incluse detalii despre preferințele în alegerea unui serviciu sau produs și achizițiile trecute, cu scopul de a contura un profil al clientului [6].

Grafurile aici oferă posibilitatea de a stoca cu ușurință o cantitate mare de informații, puternic interconectate între ele, care pot fi apoi accesate ușor pentru a efectua

analize, creea statistici în vederea îmbunătățirii produselor sau serviciilor sau chiar pentru a creea oferte personalizate fiecărui client.

2.1.4. Motoare de recomandare

Folosite de la platformele de streaming până la rețelele sociale, cu ajutorul motoarelor de recomandare, fiecărui client i se oferă recomandări personalizate, în funcție de produsele pe care le achiziționează, filmele și serialele pe care le urmărește sau prietenii pe care deja îi are.

Principiul are la bază legăturile între date. Atâta timp cât se cunosc legăturile unui nod X cu alte noduri, atunci se pot prezice legăturile viitoare ale unui nod care are legătură cu X. Pe baza preferințelor utilizatorilor se construiesc așa-numitele grafuri de cunoștințe în care se rețin preferințele fiecăruia sub formă de relații. Atunci când se generează o recomandare, se urmăresc, de fapt, șabloane în acest graf.

Spre exemplu, în cadrul unei rețele sociale, unui utilizator i se pot face recomandări de persoane pe care să le adauge în lista de prieteni analizându-se legăturile pe care le au deja prietenii acelui utilizator. Astfel, recomandările vor fi, în primul rând, persoanele care au cei mai mulți prieteni comuni cu utilizatorul și pe care cel mai probabil le și cunoaște deja.

2.1.5. Învățare automată

Inteligența artificială este în prezent un element de mare interes pentru dezvoltatori datorită capacității de a îndeplinii sarcinii din ce în ce mai complexe, sporind astfel eficiența și profitul. Aceasta se bazează pe modele, care, cu cât sunt mai complexe, cu atât sunt mai performante. Creearea de modele se face pe baza unui set de date de la programator. Datele trebuiesc colectate, clasificate ca apoi să poată fi folosite pentru antrenarea unui model de inteligență artificală.

Se dorește antrenarea unui model de inteligență artificială într-un timp cât mai scurt, motiv pentru care se dorește ca datele, deși complexe, să fie structurate într-un mod cât mai ușor de parcurs. Grafurile pot constitui o soluție pentru optimizarea procesului de învățare automată, deoarece datele se pot reprezenta într-o formă ușor de parcurs de către un algoritm de învățare automată, iar adugarea de noi date în timp este mai facilă decât dacă s-ar utiliza tabele.

2.1.6. Protecția datelor

În ultimii ani, odată cu creșterea exponențială a volumului de date care circulă în mediul virtual, protecția datelor utilizatorilor a devenit o prioritate pentru toți furnizorii de servicii. În general, datele sunt stocate în baze de date, de unde pot fi mutate, modificate și apoi folosite de către alte procese pentru prestarea unor servicii. Pentru a spori securitatea, nu sunt ținute într-o structură fixă, asftel încât să permită un acces ușor, ci se folosesc ierarhii fluide.

Problema care poate apărea este aceea de a îngreuna accesul la date atunci când este necesar. Prin grafuri se pot creea ierarhii care, atunci când sunt parcurse, se poate asigura în timp real accesul la date ușor, dar doar pentru entitățile autorizate [6].

2.2. Stadiul actual în domeniul studiului de caz

Aplicațiile navigare, așa cum am menționat și mai sus în capitolul **2.1.1**, reprezintă probleme ce se modelează natural sub formă de graf. În egală măsură, se pot folosi și baze de date relaționale, dar și noSQL pentru stocarea datelor referitoare la trasee sau la punctele de interes. Ce trebuie remarcat este faptul că aceste aplicații nu se rezumă doar la aplicarea simplă a algoritmului de navigare pe o hartă fixă dată. Acestea trebuie să ia în considerare și factorii de mediu, cum ar fi aglomerarea la anumite intervale orare sau posibilitatea de acces pentru persoanele cu dizabilități, fapt care presupune utilizarea unor algoritmi suplimentari în momentul calculării traseului optim.

În această lucrare am ales pentru efectuarea studiului de caz această categorie de aplicații, deoarece se poate implementa astfel încât să îndeplinească funcționalitățile propuse folosind ambele tehnologii (baze de date relaționale și noSQL). De asemenea, calculul traseelor, dar și alți algoritmi pentru calculul ponderilor drumurilor și actualizarea datelor se pot face în baza de date, incluzând astfel operații complexe din partea bazei de date în urma cărora se poate face o analiză comparativă.

Aplicațiile de navigare existente pe piață se pot distinge în mai multe categorii, în funcție de scopul pentru care au fost proiectate. Navigarea nu se discută doar pentru un spațiu exterior, asemenea unei hărți fizice, ci poate presupune și navigarea în cadrul unei incinte. Există chiar și aplicații concepute special pentru persoanele cu dizabilități, care au nevoie de trasee calculate după criterii speciale. În următoarele paragrafe voi oferi exemple de diverse aplicații de navigare folosite, pentru care voi evidenția și funcționalitățile pe care le prezintă.

2.2.1. Aplicații pentru navigarea exterioară

Această categorie cuprinde aplicații folosite pentru navigarea în spațiu deschis, fie pietonală, fie cu un autovehicul. Ele funcționează folosind localizare prin GPS, seturi de date predefinite dispuse sub formă de hărți, dar și date colectate continuu, folosite la calcularea traseelor în timp real, cum ar fi informații despre trafic.

- **Google Maps**

Google Maps este unul dintre cele mai populare navigatoare, datorită multitudinii de funcționalități oferite.

În primul rând, este disponibil de pe orice dispozitiv care are acces la internet, putând fi folosit fie din browser, fie din aplicația mobilă, care rulează atât pe iOS cât și pe Android, principalele cele mai folosite sisteme de operare pentru mobil.

O altă facilitate oferită o reprezintă posibilitatea de a vizualiza hărțile atât ca și reprezentări standard, similare cu o hartă fizică, cât și ca imagini așa cum sunt văzute din satelit, ceea ce poate oferi utilizatorului și mai multe informații despre o locație. Mai mult, această funcționalitate a fost extinsă pentru a da posibilitatea utilizatorului să vizualizeze o

locație de la nivelul de pieton și de a se deplasa pe hartă, așa cum se poate și în Google Earth, o altă aplicație dezvoltată de Google tot pentru navigare, însă orientată spre experiența utilizatorului de la nivelul de pieton.

O funcționalitate, foarte utilă de altfel pentru șoferi, este aceea de a colecta în timp real date cu privire la starea traficului. Atunci când se calculează un traseu, se ia în considerare și traficul de pe traseul respectiv, care poate adăuga un timp considerabil. În acest caz, algoritmul de calcul oferă și rute ocolitoare, mai puțin aglomerate.

În momentul în care utilizatorul își construiește traseul, poate specifica dacă se deplasează cu un autovehicul, cu transport în comun, sau ca pieton, poate selecta puncte intermediare și poate specifica diverse preferințe, cum ar fi dacă dorește să evite autostrăzile, drumurile cu taxe suplimentare sau dorește un traseu mai economic din punct de vedere al combustibilului. Punctele de interes, de altfel, marcate și pe hartă, cum ar fi restaurante sau magazine, au incluse și detalii oferite de către alți utilizatori, cum ar fi imagini, descrieri sau recenzii. În plus, pentru a ușura căutarea unui punct dintr-o anumită categorie, sunt disponibile filtre, cum ar fi să arate toate benzinăriile sau restaurantele din zona în care se află utilizatorul. Atunci când se calculează traseul dorit, se oferă și rute alternative, împreună cu timpul aferent parcurgerii calculat de către algoritm. Astfel, se poate spune că aplicația nu se rezumă doar la simpla navigare între puncte, ci creează o întreagă experiență utilizatorului prin posiblitățile de particularizare a traseelor și a detaliilor suplimentare pe care le furnizează cu privire la trasee și la punctelee de interes.

2.2.2. Aplicații pentru navigarea interioară

Așa cum am precizat mai sus, navigarea nu se referă doar la cea în exterior. Este la fel de necesară și navigația interioară, spre exemplu, într-o clădire cu foarte multe etaje și încăperi și care este vizitată de un număr mare de persoane.

Principala diferență față de aplicațiile destinate navigației exterioare constă în modul în care se face localizarea utilizatorului în incintă. Dacă pentru localizarea exterioară se folosește GPS, pentru interior este nevoie de sisteme alternative care să permită localizarea cât mai precisă într-un spațiu restrâns.

Tehnologia Bluetooth, respectiv Low Energy Bluetooth, reprezintă una dintre soluțiile populare utilizate în prezent. Metoda constă în instalarea în tot perimetrul incintei de balize Bluetooh, care emit semnale ce sunt apoi recepționate de către utilizator prin imtermediul aplicației, determinându-se poziția acestuia [7]. Este o soluție utilizată datorită consturilor reduse ale balizelor, care de asemenea sunt capabile să transmită semnale pe distanțe mari, de ordinul zecilor de metri, și pot estima poziția cu acuratețe de 2-3 metri. Pe deasupra, durata de viață a bateriilor este foarte mare, ajungând și până la 10 ani de funcționare.

Tehnologia WiFi poate fi de asemenea folosită pentru localizare. Nu are aceeași acuratețe precum sistemul cu balize Bluetooh, iar costul routerelor este considerabil mai mare față de cel al balizelor Bluetooth, însă este o soluție folosită cu precădere de locațiile în care deja există o infrastructură amplă, cu un număr mare de routere în funcțiune [7]. Principiul constă în calcularea poziției pe baza semnalului WiFi recepționat de către mai

multe routere, ceea ce implică faptul că, cu cât sunt mai multe routere, cu atât mai precisă este localizarea utilizatorului.

UWB (*Ultra WideBand*) este considerată tehnologia care oferă cea mai mare precizie de localizare. Principiu se bazează pe transmisia de semnal pe distanțe scurte la frecvențe mari tot de la balize, asemenea celor folosite de sistemul Bluetooth menționat mai sus. Este o metodă folosită mai mult pentru localizarea și monitorizarea obiectelor valoroase, fiind nu numai costisitoare implementarea sistemului ci și pentru că nu este o tehnologie suportată decât de o gamă restrânsă de dispozitive mobile [8].

- **Pointr**

Pointr este o companie care oferă soluții dedicate pentru proiectarea de hărți interactive, dar și sisteme de navigație interioară pentru magazine, aeroporturi, clădiri de birouri sau hoteluri.

Sistemul de localizare utilizat de aplicație se bazează pe balize Bluetooth sau WiFi, dar se îmbină și cu algoritmi de învățare automată și inteligență artificială pentru a spori precizia. Pe deasupra, acoperă chiar și cazul tranziției interior-exterior, ceea ce este util în cazul în care în zona care se dorește reprezentată pe hartă cuprinde mai multe clădiri, putând fii posibilă deci și localizarea prin GPS. Pe baza locației utilizatorului, aplicația poate schimba automat mecanismul de localizare asftel încât să folosească fie GPS, fie sistemul interior.

O altă funcționalitate oferită este navigarea folosind realitatea augmentată, care constă în reprezentarea indicațiilor pentru navigare direct în mediul în care navighează utilizatorul. De asemenea, aplicația poate reda indicațiile pentru navigare și sub formă de msaje vocale.

2.2.3. Aplicații de navigare destinate persoanelor cu dizabilități

Acestea pot fi considerate o categorie specială, deoarece indicațiie de navigare trebuie să cuprindă mai multe detalii pentru a oferi o experiență de calitate utilizatorului, cum ar fi avertizmente pentru diverse obstacole, cum ar fi trecerile pentru pietoni, sau detalii cu privire la punctele de interes din zonă pentru persoanele nevăzătoare sau rute alternative, spre exemplu, care să evite scările, pentru persoanele aflate în scaun cu rotile. Se pot proiecta astfel de aplicații atât pentru spații interioare cât și pentru exterior.

- **Lazarillo**

Lazarillo este un furnizor de soluții pentru navigare dedicată persoanelor nevăzătoare atât pentru navigare exterioară, cât și interioară, cum ar fi hoteluri, magazine, centre medicale sau campusuri universitare.

Una dintre facilitățile sale este aceea că se pot crea hărți ce pot fi personalizate, clientul având posibilitatea de a adăuga detaliile pe care le consideră necesare oricărui punct de pe hartă, oferind astfel utilizatorului o experiență cât mai sigură și plăcută prin intermediul mesaje audio îndrumătoare. În plus, ghidul vocal poate fi personalizat de către fiecare utilizator în funcție de nevoile sale, putând fii setate limba, unitățile de măsură și vocea folosită pentru comunicarea mesajelor.

3. Descrierea problemei abordate și a metodei de rezolvare propuse

Prin lucrarea de față îmi propun să realizez o analiză comparativă cât mai completă între bazele de date relaționale și cele noSQL de tip graf, folosind ca studiu de caz o aplicație simplă de navigare într-o clădire. Bazele de date utilizate vor fi Microsoft SQL Server, care este una relațională, și Neo4j, bază de date noSQL de tip graf, iar aplicația va fi construită în limbajul de programare .NET (C#). Comparația se va face atât din punct de vedere al performanțelor la rulare, cât și al caracteristicilor specifice ale fiecărui tip de baze de date, evidențiând astfel punctele atât tari cât și slabe ale acestora.

Aplicația, fiind proiectată special pentru efectuarea studiului de caz, nu m-am concentrat să implementez funcționalități complexe pentru utilizator, așa cum se regăsesc în alte aplicații de navigare menționate în capitolul anterior. În schimb, m-am concentrat să implementez funcționalitățile de bază, care presupun solicitarea bazelor de date, acetsea reprezentând obiectul acestui studiu. De asemenea, pentru a ajuta procesul de analiză, am implementat funcționalități suplimentare, dedicate exclusiv temei de față, mai exact selectarea bazei de date utilizate la rulare și vizualizarea informațiilor de rulare înregistrare în jurnale atât în formă tabelară cât și ca grafice.

3.1. Funcționalitățile aplicației

În momentul în care se rulează aplicația, se deschide în browser pagina principală, în care se configurează traseul dorit. În Figura 1 se pot vedea harta, meniul de sus din care se pot selecta harta, baza de date utilizată, dar de unde se pot și căuta puncte folosind spațiul dedicat *Caută...,* meniul de configurare al traseului din partea stângă, panoul din partea dreaptă în care se afișează traseul calculat și panoul de jos în care se scriu informațiile despre rulare care sunt folosite la statistici.

Fig. 1. Pagina principală

Meniul de configurare a traseului începe cu 2 elemente pentru selectarea punctelor de plecare și sosire. Fiecare dintre ele afișează câte o listă cu toate sălile spre care se poate naviga, grupate pe etaje. De asemenea, punctul de sosire se poate selecta și din bara de căutare din partea de sus a pagini. În Figura 2 se poate vedea și butonul *Calculează traseu*, care trimite cererea la server de calcul al traseului, împreună cu punctele alese ca parametrii.

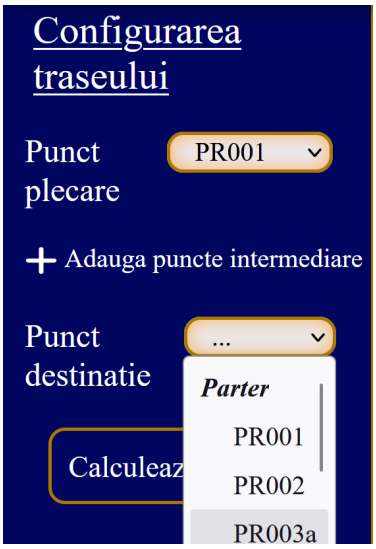

Fig. 2. Setarea punctelor de sosire și destinație

Pe lângă cele 2 puncte de plecare și sosire se mai pot adăuga și puncte intermediare. Aplicația permite maxim 5 la număr și se pot adăuga apăsând butonul în formă de plus unde scrie „Adăugare puncte intermediare”. Atunci când se calculează traseul, se va trimite la server și lista punctelor intermediare adăugate de către utilizator.

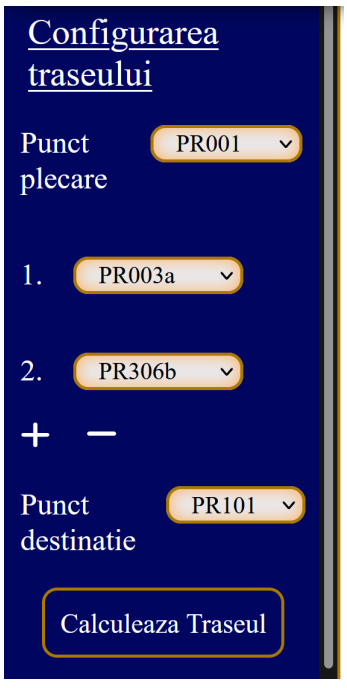

Fig. 3. Adăugarea punctelor intermediare

O altă facilitate ce se regăsește și în aplicațiile de navigare disponibile pe piață este aceea de a adăuga filtre pentru a evita fie o categorie de puncte de pe hartă fie puncte alese de către utilizator. În aplicație sunt implementate 2 filtre: unul pentru evitarea scărilor (se presupune scenariul în care aplicația este folosită de o persoană cu dizabilități care nu poate folosi scările) și unul prin care utilizatorul își poate crea o listă personalizată de puncte pe care dorește să le evite. La rularea algoritmului de calculare a traseului, se vor trimite și aceste puncte. În cazul în care, din cauza configurărilor utilizatorului, nu se poate forma niciun traseu, atunci în panoul dedicat traseului se va afișa mesajul „Nu există traseu”.

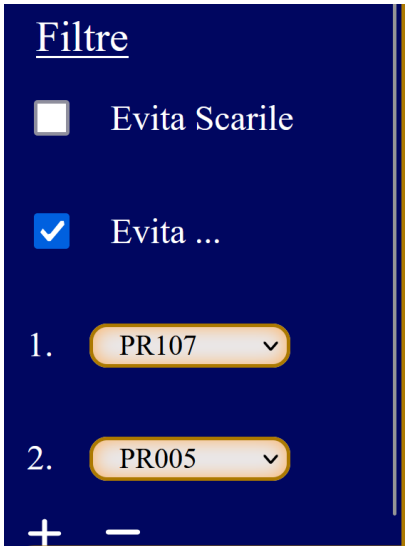

Fig. 4. Filtre

După ce se calculează traseul și este întors de către server, pașii sunt afișați în panoul din partea dreaptă a ecranului. Mai mult, traseul este reprezentat și pe hartă pentru a ghida utilizatorul prin incintă.

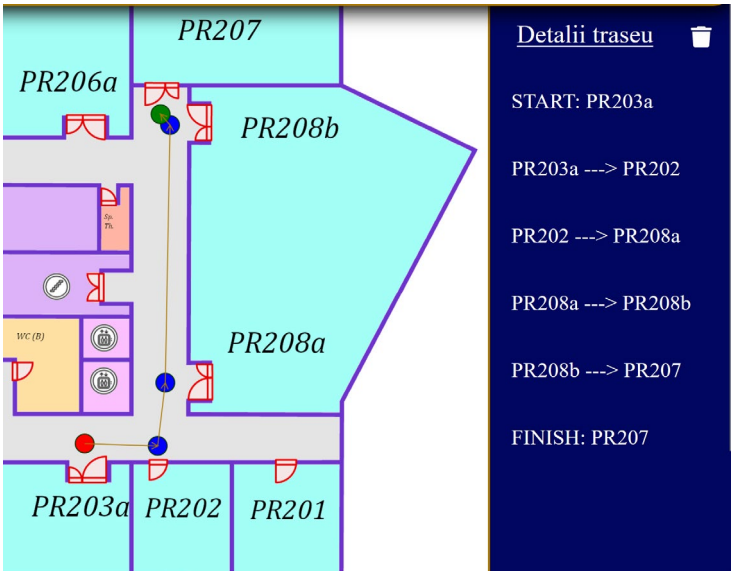

Fig. 5. Traseul și reprezentarea pe hartă

Așa cum am precizat mai sus, aplicația de față cuprinde și funcționalități dedicate studiului de caz, putând fi astfel considerată o unealtă în cadrul acestuia. Una dintre ele o

reprezintă posibilitatea de a selecta baza de date utilizată. Dacă într-o aplicație reală nu este necesar ca utilizatorul să cunoască mecanismul din spate, aici din contră, din moment ce reprezintă obiectul acestei lucrări, am considerat necesar să introduc posibilitatea de a schimba ușor între cele 2 sisteme folosite.

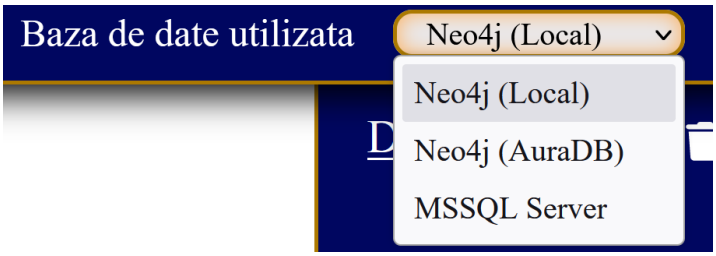

Fig. 6. Alegerea bazei de date

Pentru a vizualiza rapid informațiile folosite la statistici obținute în urma unei rulări, acestea se scriu în panoul de jos. Tot acolo se observă și un buton numit „Statistică”. La apăsarea acestuia se deschide pagina de statistici.

În cele din urmă, pentru a putea formula observații și concluzii pe baza celor măsurate, se poate deschide o pagină dedicată, în care, pentru fiecare bază de date, se specifică câte cereri s-au efectuat în ziua respectivă, se enumeră în formă tabelară toate acestea și se desenează câte 3 grafice care redau utilizarea CPU, a memoriei și timpii de execuție, aceștia fiind metricii măsurați în momentul rulării. Toate aceste date care sunt afișate sunt extrase din jurnale care sunt salvate pe server și care sunt completate în urma fiecărei cereri.

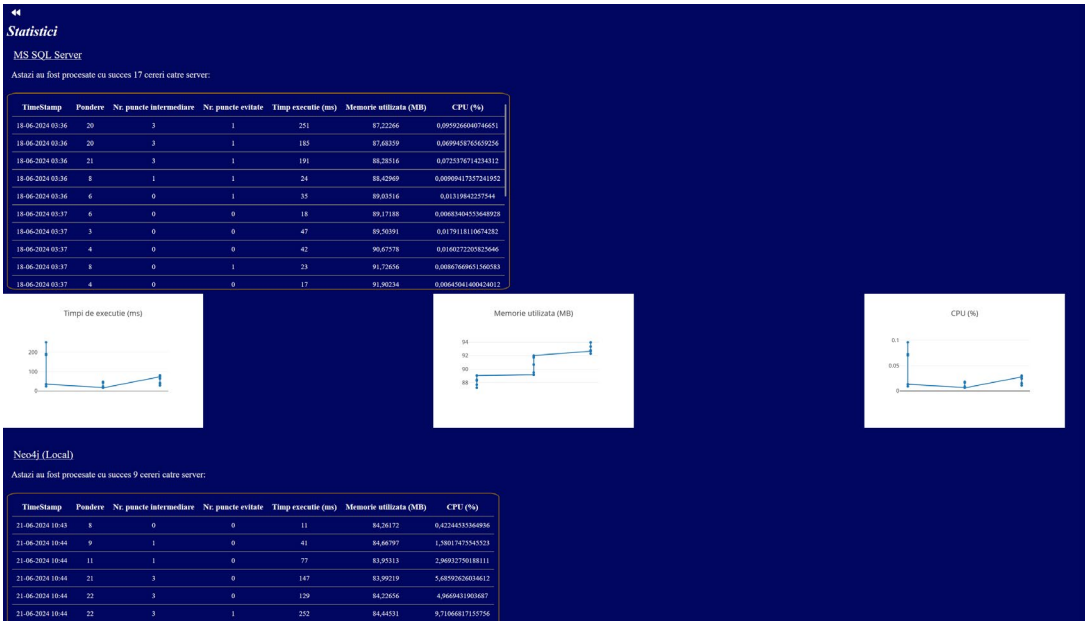

Fig. 7. Pagina de statistici

3.2. Metricii analizați

Este important de menționat faptul că metricii avuți în vedere pentru această comparație se pot clasifica în 3 categorii. În primul rând, în urma rulării aplicației, se pot măsura utilizarea CPU, utilizarea memoriei și timpul de execuție al calculului traseului.

Acești metrici sunt influențați de calculele efectuate în momentul rulării algoritmului, se pot calcula în mod identic pentru ambele baze de date, din codul aplicației, și, deci, se măsoară în urma fiecărei rulări și se salvează în jurnal. Apoi, atât Neo4j cât și SQL Server dispun de unelte dedicate analizării interogărilor, din care se pot extrage informații suplimentare, cum ar fi numărul de accesări al bazei de date sau operațiile I/O. Nu în ultimul rând, pentru o comparație completă trebuie discutate și diferențele din punct de vedere al modului în care pot fi integrate bazele de date în aplicații complexe, ceea ce implică menționarea instrumentelor auxiliare pe care le pun fiecare la dispoziție, posibilităților de conectare prin intermediul unui limbaj de programare și utilizare de cod extern, elementelor de securitate și a soluțiilor de integrare în cadrul serviciilor de tip Cloud.

- **Timpul de execuție**

Acesta reprezintă timpul în care se efectuează calculul traseului. Măsurarea acestuia, astfel încât să se poată cronometra în mod identic pentru ambele baze de date, se face în cadrul codului aplicației. În C# se poate folosi o variabilă de tip *Stopwatch*, dedicată calculului diferențelor de timp. Se pornește (*Start()*) la începutul efectuării tranzacției sau atunci când este invocată procedura stocată și se oprește (*Stop()*) la final. Variabila respectivă va conține valoarea în milisecunde a timpului scurs.

- **Memoria utilizată**

Constituie memoria ocupată de către procesul unei baze de date. Pentru determinarea acestei valori se folosește clasa *PerformanceCounter* din biblioteca *System.Diagnostics.* Aceasta întoarce direct valoarea căutată.

- **CPU utilizat**

Acesta nu se poate obține într-atât de ușor precum memoria și timpul așa că în acest sens a trebuit implementată o procedură mai amplă. Întâi, folosind *PerformanceCounter,* se obține procentul din timpul total în care toate procesele utilizează CPU, se selectează apoi toate procesele care țin de baza de date dorită, folosind clasa *Process* tot din biblioteca *System.Diagnostics* și se calculează timpul total alocat proceselor, apoi se obține valoarea dorită, procentul de utilizare a CPU, aplicând formula *timp_execuție* * *timp_total_procese / timp_procese_interes.* Toate aceste calcule se efectuează imediat după finalizarea unei tranzacții în baza de date.

- **Planurile de execuție din SQL Server**

SQL Server Management Studio pune la dispoziție la rândul său unelte pentru analizarea interogărilor: *Display Estimated Execution Plan* și *Include Actual Execution Plan*. Astfel, pentru orice interogare sau procedură stocată ce se dorește executată se pot vizualiza planurile de execuție, care pot fi fie estimate, adică calculate pe bază de estimări fără a executa interogările, fie reale, calculate după execuție, pe baza datelor reale. Aceste planuri constituie reprezentări grafice ale etapelor de excuție, pentru fiecare fiind disponibile câte o listă cu informațiile de execuție calculate, cele mai importante fiind numărul de linii citite (*Number of rows read*), utilizarea CPU (*CPU cost)*, numărul de execuții (*Number of executions*) și Costul I/O (*Estimated I/O cost*). Se pot face observații aici pe baza diferențelor dintre planurile estimate și cele reale.

- **Planurile de execuție din Neo4j**

Similar cu SQL Server, și Neo4j pune la dispoziția utilizatorului instrumente de analiză. Acestea sunt 2 comenzi Cypher care se pot adăuga la începutul unei interogări pentru a returna planul de execuție: PROFILE și EXPLAIN. La fel ca în SQL Server, se face distincția între planurile reale, obținute cu ajutorul comenzii PROFILE, și planurile estimate, obținute prin comanda EXPLAIN. Planurile constau, de asemenea, în reprezentări grafice care ilustrează etapele. Diferența care se poate remarca totuși între Neo4j și SQL Server este că Neo4j nu face estimări numerice în ceea ce privește informațiile de rulare. PROFILE întoarce planul împreună cu informațiile de rulare, care includ aici memoria utilizată și numărul de accesări al bazei de date. În plus, sunt afișate în interfață și numărul total de accesări ale bazei de date, împreună cu timpul total de execuție. EXPLAIN în schimb întoarce doar reprezentarea grafică. Și aceasta poate fi utilă, deoarece se construiește fără să execute initerogarea în cauză și se pot vizualiza pașii pe care îi parcurge interogarea.

3.3. Structura interogărilor

Atât în Cypher (pentru Neo4j) cât și în SQL am implementat pentru calcularea traseului algoritmul Dijkstra, care primește ca parametrii punctul de plecare, punctul destinație, o valoare booleană care specifică dacă trebuie evitate nodurile de tip scări, o listă (care poate fi și vidă) de puncte intermediare și o listă de puncte evitate alese de către utilizator (care poate fi deasemenea vidă). În ambele baze de date, calculul se face nu direct pe tabelul / graful original cu noduri, ci se construiesc tabele temporare / grafuri virtuale, în acest fel oferind posibilitatea de a lucra cu hărți personalizate din care se elimină automat punctele evitate.

```
Dacă (este folosită o hartă completă sau doar cu filtrul FarăScări aplicat) atunci
{
    Verifică dacă există deja în memorie graful virtual / tabelul temporar

    Dacă (graf virtual / tabel temporar nu există) atunci
    {
        Creează unul corespunzător
    }
}
Altfel    //Exista puncte evitate
{
    Creează un graf virtual / tabel temporar care să excludă punctele evitate
}

[tr, pond] = Aplică Dijkstra
Traseu = tr
Pondere = pond
```

Fig. 8. Pseudocod calcul traseu fără puncte intermediare

Pentru SQL Server, codul este scris în totalitate într-o procedură stocată, care este doar invocată de către aplicație prin intermediul codului C#. Pentru Neo4j, codul se regăsește integrat în codul aplicației, care, cu ajutorul librăriei dedicate, pornește o tranzacție în baza de date. Datorită acestei particularități, algoritmul este implementat combinând interogările Cypher cu codul C#. De asemenea, se remarcă faptul că dacă

pentru SQL a trebuit realizată o implementară manuală a algoritmului Dijkstra, Cypher pune la dispoziție o implementare în cadrul plugin-ului *Graph Data Science* (abreviat GDS), care primește ca parametrii un graf virtual, realizat la rândul său folosind tot GDS, nodul de plecare, nodul destinație și denumirea proprietății relațiilor în care se regăsește ponderea.

Așa cum am precizat mai sus, algoritmul rulează pe harta virtuală construită pe loc. Astfel, pentru a lua în considerare orice punct care se dorește evitat, este suficient ca acesta să nu fie inclus in harta virtuală.

Pentru cazul punctelor intermediare, din păcate algoritmul Dijkstra nu are o soluție în acest sens, și nu este inclusă nici în implementarea din GDS. Astfel, în acest sens am completat cele 2 implementări cu o buclă repetitivă în care se rulează de mai multe ori algoritmul Dijkstra. Pentru a găsi drumul cel mai scurt între 2 puncte, care să includă și o listă de puncte intermediare, trebuie identificat, în primul rând, traseul cel mai scurt între perechile de puncte intermediare. Am presupus că ordinea în care se parcurg punctele intermediare nu este impusă, fapt care sporește complexitatea algoritmului. Se presupune că punctele intermediare se află într-o vector. Numărul de iterații este dat de dimensiunea acestui vector. La fiecare iterație, se calculează traseul de la punctul de plecare la primul punct intermediar din vector, apoi de la fiecare pe rând se calculează traseul de la primul punct din vector la cel de-al doilea, apoi de la al doilea la cel de al treilea și tot așa până să formează traseul până la ultimul punct intermediar, apoi se calculează traseul de la ultimul punct la punctul destinație. Se salvează acest traseu împreună cu ponderea lui și se trece la următoarea iterație. La final, traseul rezultat va fi cel care are ponderea cea mai mică.

```
Dacă (este folosită o hartă completă sau doar cu filtrul FarăScări aplicat) atunci
{
    Verifică dacă există deja în memorie graful virtual / tabelul temporar

    Dacă (graf virtual / tabel temporar nu există) atunci
    {
        Creează unul corespunzător
    }
}
Altfel    //Exista puncte evitate
{
    Creează un graf virtual / tabel temporar care să excludă punctele evitate
}

Dacă (nu există puncte intermediare) atunci
{
    Aplică Dijkstra
    Extrage rezultatul
}
Altfel    //punctele intermediare sunt în forma unui vector: V = [x1, x2, x3]
{
    Pondere = 999

    For (i = 0; i < len(vectorPuncteIntermediare); i++)
    {
        [tr, pond] = Aplică Dijkstra punctStart-V[0]
        PondereTemp = pond
        TraseuTemp = tr
        For (i = 0; i < len(vectorPuncteIntermediare) - 1; i++)
        {
            [tr, pond] = Aplică Dijkstra V[i]-V[i+1]
            PondereTemp += pond
            TraseuTemp += tr
        }
        [tr, pond] = Aplică Dijkstra V[len(vectorPuncteIntermediare) - 1]-punctFinish
        PondereTemp += pond
        TraseuTemp += tr

        Dacă (PondereTemp < Pondere) atunci
        {
            Pondere = PondereTemp
            Traseu = TraseuTemp
        }

        Permută la circular spre stânga => V = [x2, x3, x1]
    }
}
```

Fig. 9. Pseudocod calcul traseu cu puncte intermediare

4. Documentație tehnică

4.1. Tehnologii folosite

Aplicația este construită folosind ca limbaj de programare pentru server .NET (C#). Pentru interfață am folosit javaScript, în combinație cu biblioteca React.js.

Pentru simularea utilizatorilor care accesează simultan aplicația am folosit Apache JMeter, o aplicație prin care se pot creea și executa planuri de testare pentru aplicații web. În cadrul acestei lucrări, am creat planuri de testare prin care se simulează un număr mare de utilizatori prin crearea de threaduri care trimit simultan cereri către server, fiecare având câte un set de date unice de intrare.

Baza de date noSQL de tip graf pe care am ales să o folosesc este Neo4j, care este cel mai utilizat sistem de acest tip la momentul actual. Acesta a fost lansat pe piață în 2007, putând spune deci că este un sistem relativ nou, însă a căptătat popularitate considerabilă în ultimii ani, împreună cu alte baze de date noSQL. Neo4j este construit în Java, fiind deci necesară instalarea JVM pentru utilizarea acestuia, și folosește ca limbaj de interogare exclusiv Cypher. Implicit, modelul utilizat este cel de graf cu proprietăți etichetate, dar, cu ajutorul unui plugin dedicat, se poate folosi și modelul RDF. Este important, de asemenea, de menționat faptul că Neo4j, conform teoriei CAP, este o bază de date de tip CA, asemenea uneia relaționale, fiind capabil să asigure consistența și disponibilitatea.

Baza de date relațională folosită în această lucrare este Microsoft SQL Server. Lansat pe piață în 1989, în prezent este una dintre cele mai populare sisteme de acest tip, împreună cu MySQL și Oracle Database, datorită multitudinii de funcționalități pe care le oferă și a performanțelor deosebite. MS SQL Server este construit în C și C++, putând astfel să ruleze pe orice calculator fără a necesita o mașină virtuală și folosește ca limbaj de interogare SQL, unicul limbaj de interogare a bazelor de date relaționale de altfel.

În cadrul aplicației .NET, pentru a utiliza prin intermediul codului aceste baze de date, am inclus bibliotecile *System.Data.SqlClient,* respectiv *Neo4j.Driver.* Pentru calcularea metricilor prezentați în capitolul **3** am utilizat biblioteca *System.Diagnostics.* Pentru scrierea în jurnale am folosit biblioteca *Zlogger.*

4.2. Structura aplicației

Aplicația web este construită pe modeul MVC (*Model-View-Controller*). Astfel, pentru partea de interfață cu utilizatorul am avut un singur View numit „Index.cshtml”, în care a fost încărcat conținutul paginii, fie ea cea principală, fie cea de statistici, iar pentru calculele operațiile la server, care au implicat și comunicarea cu bazele de date, am folosit 3 controller-e: „HomeController.cs”, care încarcă pagina web dorită, „Neo4jCntroller.cs”, care gestionează operațiile cu Neo4j, și „SQLServerController.cs”, care gestionează operațiile cu SQL Server.

Pentru construcția interfeței, am concentrat toate operațiile pe partea clientului, asftel încât la server să revină doar operațiile care țin de comunicarea cu bazele de date. În

acest sens, am cuprins în fișierul „ReactComps.jsx” toate componentele React definite și folosite în întreaga aplicație, în „Funcs.jsx” toate funcțiile javaScript care sunt folosite de către diverse componte și în Constants.js constantele care sunt folosite de componentele DropDown din aplicație. În acest fel se pot refolosi la maxim toate componentele React, fiecare putând fi instanțiată cu funcții sau constante diferite. O schemă a implementării se poate vedea în Figura 10.

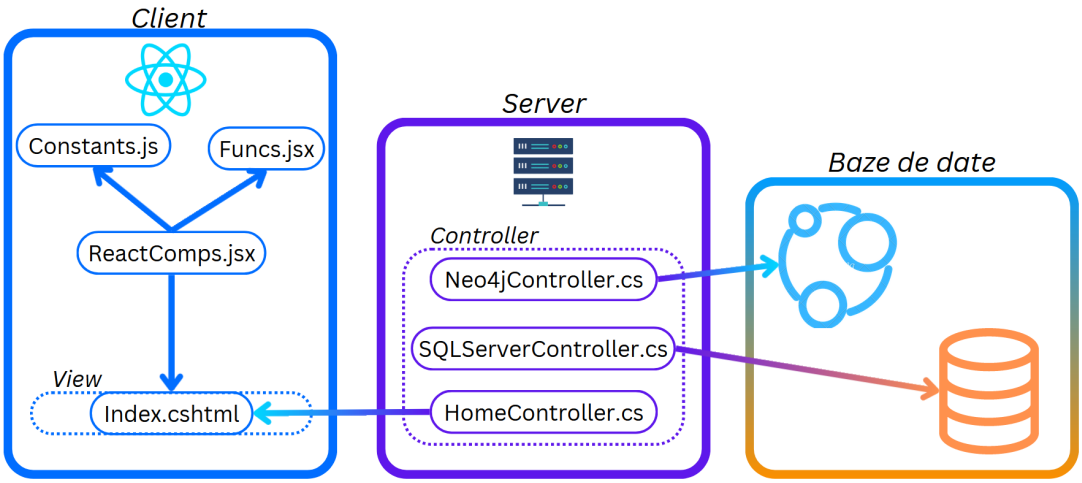

Fig. 10. Schema implementării

4.3. Proiectarea bazelor de date

4.3.1. Neo4j

Am presupus că fiecare sală este echivalentă cu un nod în graf și traseele între săli sunt echivalente cu relațiile dintre noduri. În acest sens am creat câte un nod pentru fiecare sală, care are eticheta „sala” și câte nu nume unic ca și proprietate, legate între ele prin perechi de relații care au ponderile ca și proprietăți, deoarece traseele sunt bidirecționale (iar o relație în graf este mereu unidirecțională). Ca noduri am reprezentat de asemenea și scările și liftul prezente în clădire, dar și ieșirile din clădire, acestea fiind incluse ca și puncte atunci când se calculează traseul, și le-am atribuit la fel un nume și etichete corespunzătoare. Schema realizată pentru unul dintre etaje se poate vizualiza în Figura 11. Am selectat doar un etaj pentru a obține o imagine simplă și clară.

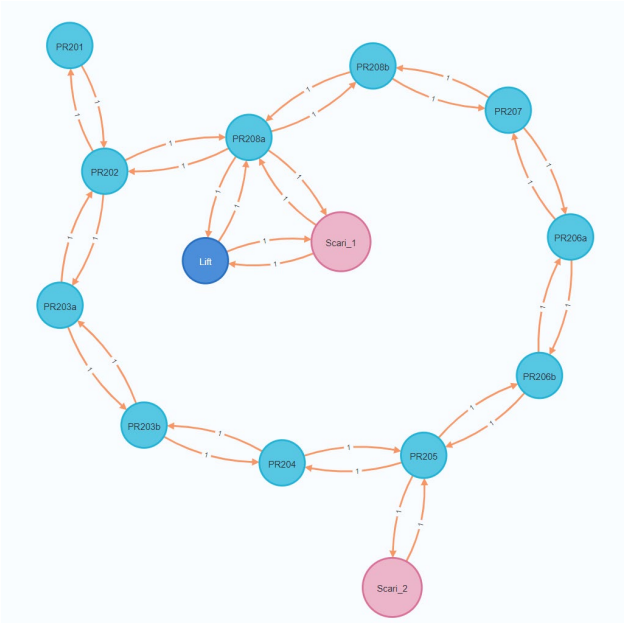

Fig. 11. Schema unui etaj

4.3.2. MS SQL Server

Pentru a avea o reprezentare echivalentă cu cea din Neo4j, am creat 2 tabele, unul pentru noduri și unul pentru relațiile dintre noduri, conform diagramei din Figura 12. Tabelul de noduri conține fiecare nod împrună cu un identificator și cu tipul acestuia (sala, scari, lift, iesire din clădire), iar tabelul de relații cuprinde perechi de noduri împreună cu ponderea traseului.

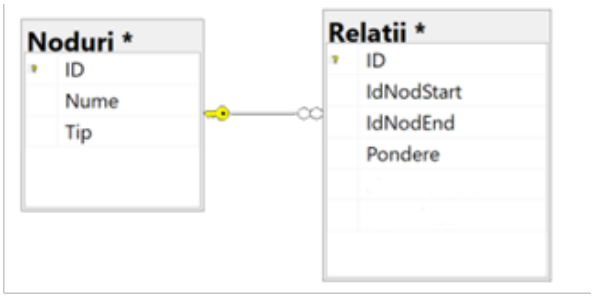

Fig. 12. Diagrama bazei de date din SQL Server

4.4. Specificații tehnice ale calculatorului folosit pentru rulare

Calculatorul pe care rulează cele aplicația construită are următoarele specificații tehnice:

- Sistem de operare: Microsoft Windows 11;
- Procesor: Intel Core i9-10885H CPU @ 2.40GHz, 8 nuclee, 16 procesoare logice;
- Memorie RAM: 32 GB

Versiunile bazelor de date:

- Neo4j: 5.12.0
- Microsoft SQL Server: 2019

5. Rezultate obținute

5.1. Rezultate obținute în timpul rulării

Pentru a simula un număr de utilizatori virtuali care accesează simultan aplicația, am creat pentru fiecare bază de date câte un plan de test (*Test Plan*) în Apache JMeter. În figura 13 se poate vedea planul întocmit pentru Neo4j. Similar este realizat și pentru SQL Server. Fiecare plan are inclus un grup de fire de execuție (*Thread Group*), în care pot stabili câte fire de execuție să trimită simultan cereri la server, o cerere HTTP (*HTTP Request*), în care se specifică protocolul, tipul de cerere, calea, portul, serverul și lista de parametrii, o serie de variabile generate aleator ce vor fi folosite în lista de parametrii astfel încât fiecare cerere să fie unică, și o listă de rezultate (*View Results Tree*) în care se poate vizualiza fiecare cerere împreună cu răspunsul de la server asociat.

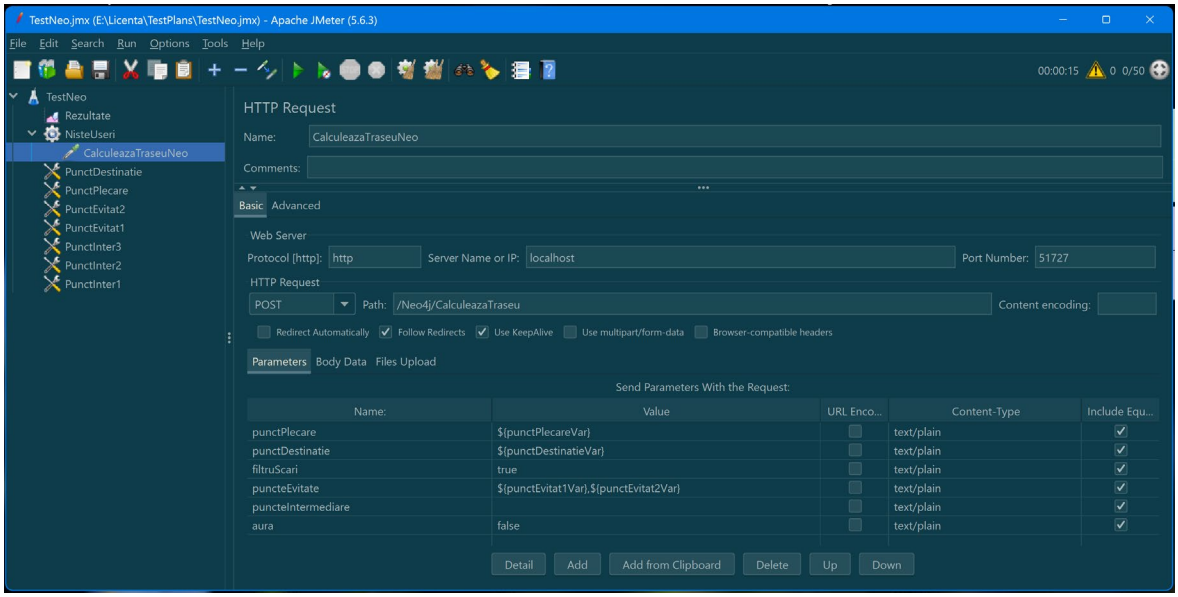

Fig. 13. Planul de test din Apache JMeter pentru testarea Neo4j

Pentru a acoperi cât mai multe cazuri posibile, am stabilit să rulez aceste planuri de testare cu 4 configurări diferite ale listei de parametrii: cu 3 puncte intermediare și 2 evitate, fără niciun punct intermediar sau evitat, cu un singur punct intermediar și fără niciun punct intermediar, dar cu 2 puncte evitate, de fiecare dată având setate 50 de fire de execuție pe grup. De asemenea, am efectuat și un test manual, în care, am trimis cereri din aplicația web. În acest fel, am observat cum este influențat răspunsul de complexitatea algoritmilor rulați pe fiecare bază de date atât în cazul în care există puține cereri la un interval mare de timp (de ordinul secundelor) cât și în cazul în care există mai multe cereri simultan, care implică diverse complexități ale algoritmilor de calcul.

Dacă pentru SQL Server nu a fost nevoie de nicio configurare suplimentară, pentru Neo4j a fost nevoie de modificarea fișierului de setări, *neo4j.config,* ca acesta să funcționeze corect în timpul testării. La rularea manuală din aplicație a funcționat fără probleme, însă atunci când au fost trimise mai multe cereri simultane, baza de date răspundea cu eroare (*java.lang.OutOfMemoryError)*, deoarece nu avea suficientă memorie

heap în JVM. În [10] se descrie modul în care este organizată memoria de către Neo4j. Principalele regiuni sunt considerate off-heap și on-heap. Dacă prin on-heap se definește memoria heap din JVM, care este responsabilă pentru stocarea datelor la runtime, execuția interogărilor și gestionarea grafurilor și tranzacțiilor, prin off-heap sunt definite memoria page-cache, care păastrează în memorie datele stocate în graf, memoria directă, care conține buffer-ele, și resursele interne ale JVM. Din toate acestea, se pot configura manual dimensiunea inițială și cea maximă a memorie heap și page-cache. Așadar, modificările efectuate au constat în modificarea următorilor parametrii din fișierul de configurare:

- *Server.memory.heap.initial_size:* reprezintă dimensiunea inițială a memoriei heap din JVM și implicit se alocă dinamic pe baza resurselor disponibile ale sistemului pe care rulează → Setat manual la 2 Gb
- *Server.memory.heap.max_size:* reprezintă dimensiunea maximă pe care poate să o atingă memoria heap din JVM și implicit se alocă dinamic pe baza resurselor disponibile ale sistemului pe care rulează → Setat manual la 8 Gb
- *Server.memory.pagecache.size:* reprezintă dimensiunea memoriei utilizate la maparea fișierelor și implicit are valoarea egală cu 50% din dimensiunea memorie RAM disponibile minus dimensiunea maximă a memoriei heap și nu va depăși o dimensiune de 70 de ori mai mare decât cea a memoriei heap → Setat la 10 Gb
- *Dbms.memory.transaction.total.max:* reprezintă cantitatea maximă de memorie pe care o pot consuma toate tranzacțiile simultan și implicit se alocă o dimensiune egală cu 70% din dimensiunea memoriei heap → Setat manual la 2 Gb
- *Db.memory.transaction.max:* reprezintă cantitatea de memori maximă pe care o poate consuma o tranzacție → Setat manual la 16 Mb

În figurile 14 - 18 se pot vedea timpi de execuție pentru SQL Server:

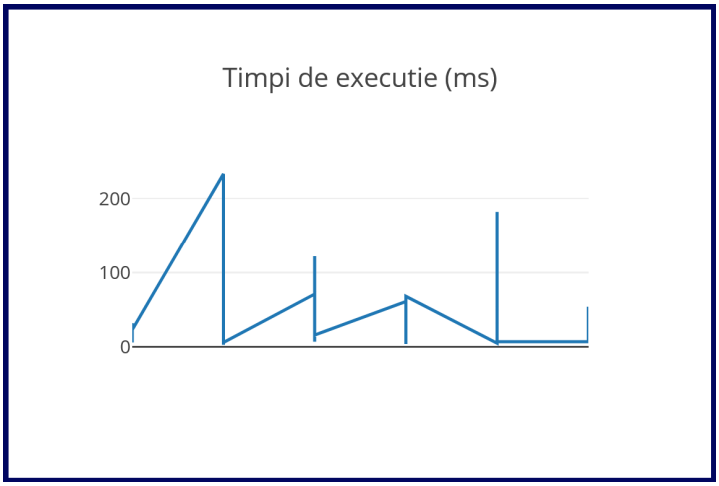

Fig. 14. Timpi de execuție pentru SQL Server pentru cazul fără puncte intermediare/evitate

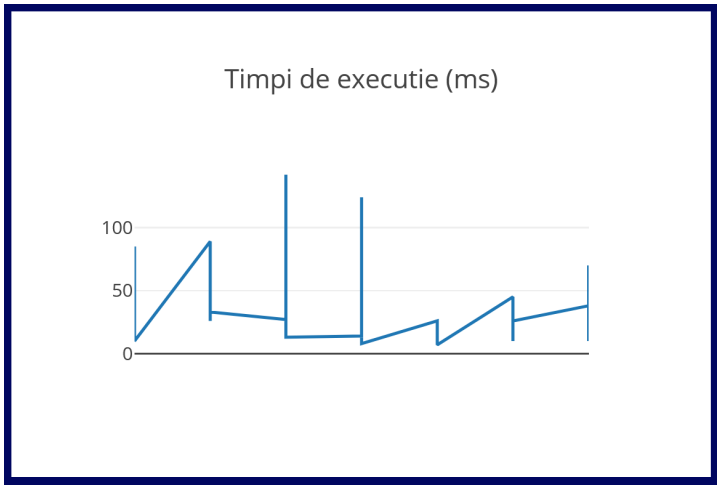

Fig. 15. Timpi de execuție pentru SQL Server pentru cazul cu 2 puncte evitate

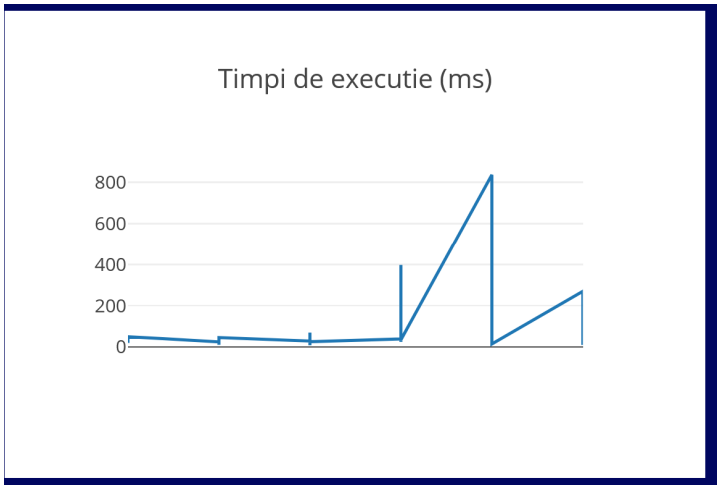

Fig. 16. Timpi de execuție pentru SQL Server pentru cazul cu un punct intermediar

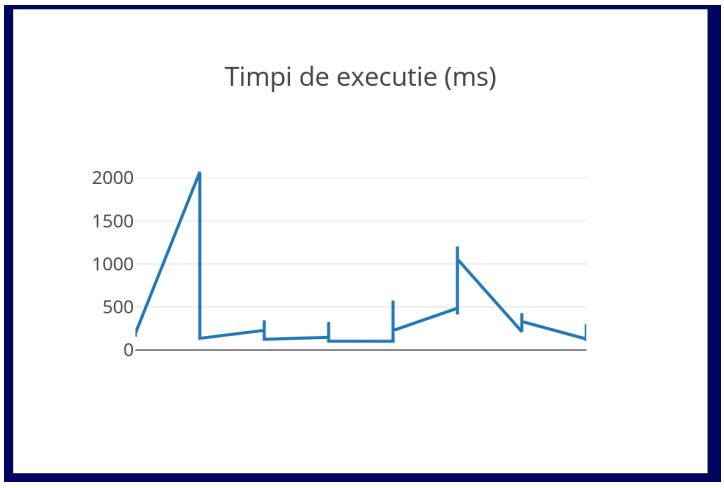

Fig. 17. Timpi de execuție pentru SQL Server pentru cazul cu 3 puncte intermediare și 2 puncte evitate

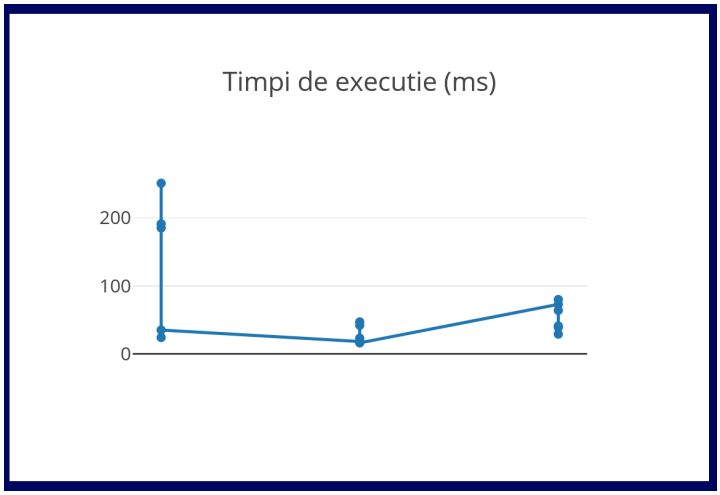

Fig. 18. Timpi de execuție pentru SQL Server pentru cazul cu rulări manuale aleatorii

Timpii de execuție fluctuează în funcție de lungimea traseului obținut, fiind influențați direct de numărul de iterații făcute în cadrul algoritmului. Creșterea este vizibilă atunci când se introduc punctele intermediare, fiind necesară deci repetiția algoritmului de calculare a traseului între perechile de puncte. Urmărind și cazul cererilor trimise manual din aplicație se poate afirma faptul că SQL Server reușește să păstreze o medie a timpilor de răspuns chiar și în cazul solicitării de către mai mulți utilizatori simultan.

În figurile 19 - 23 se pot vedea timpi de execuție pentru Neo4j:

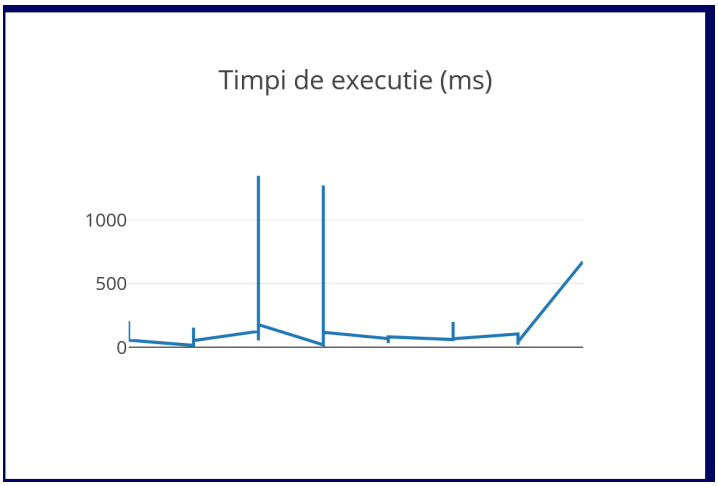

Fig. 19. Timpi de execuție pentru Neo4j pentru cazul fără puncte intermediare/evitate

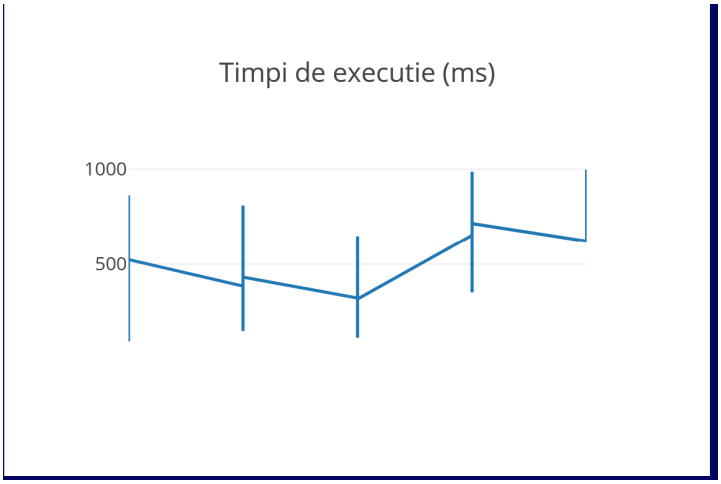

Fig. 20. Timpi de execuție pentru Neo4j pentru cazul cu 2 puncte evitate

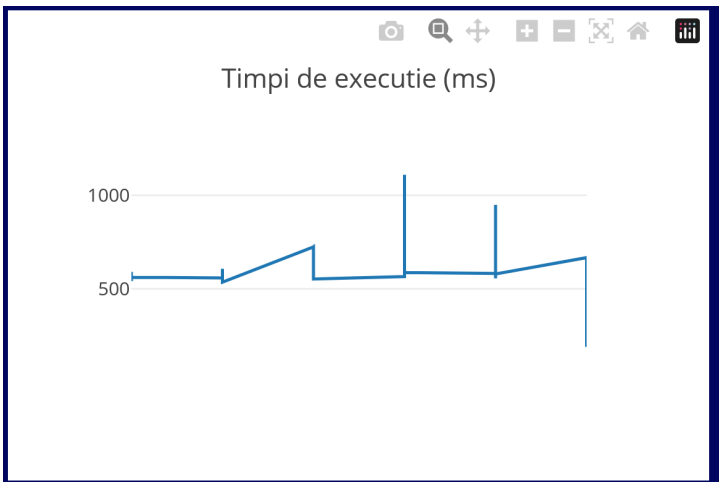

Fig. 21. Timpi de execuție pentru Neo4j pentru cazul cu un punct intermediar

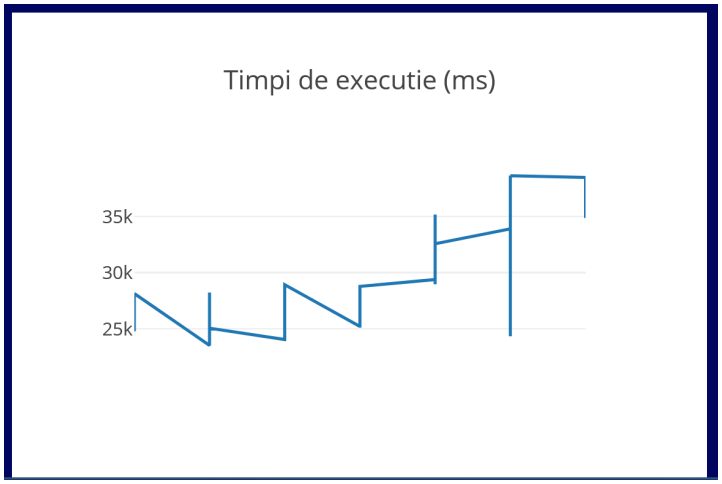

Fig. 22. Timpi de execuție pentru Neo4j pentru cazul cu 3 puncte intermediare și 2 puncte evitate

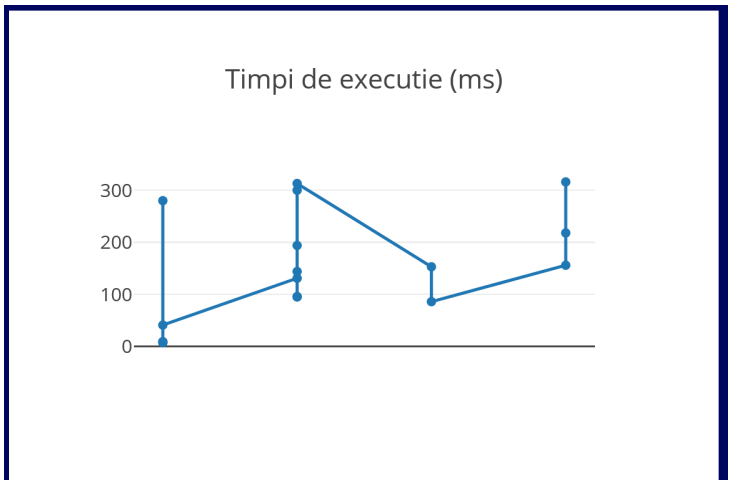

Fig. 23. Timpi de execuție pentru Neo4j pentru cazul cu rulări manuale aleatorii

Timpii de execuție pentru Neo4j sunt clar mai mari față de SQL Server, inclusiv la cazul rulărilor manuale din aplcație. În cazul accesului mai multor utilizatori simultan, diferențele sunt din ce în ce mai mari pe măsură ce calculul devine mai complex.

În figurile 24 - 28 se poate vedea procentul de utilizare a CPU pentru SQL Server:

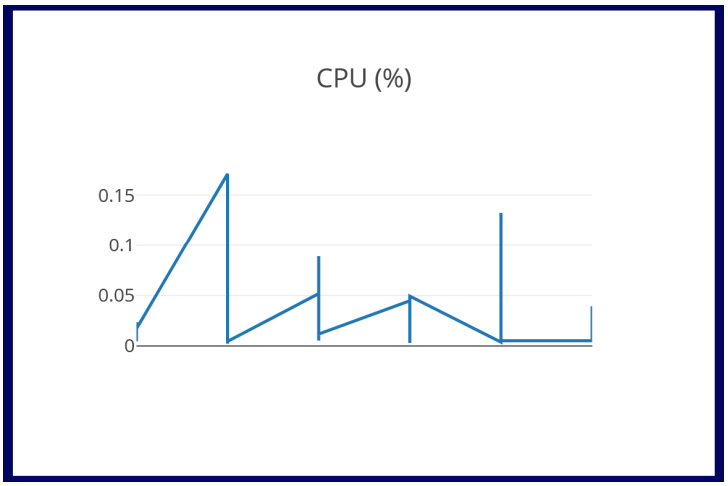

Fig. 24. Utilizarea CPU pentru SQL Server pentru cazul fără puncte intermediare/evitate

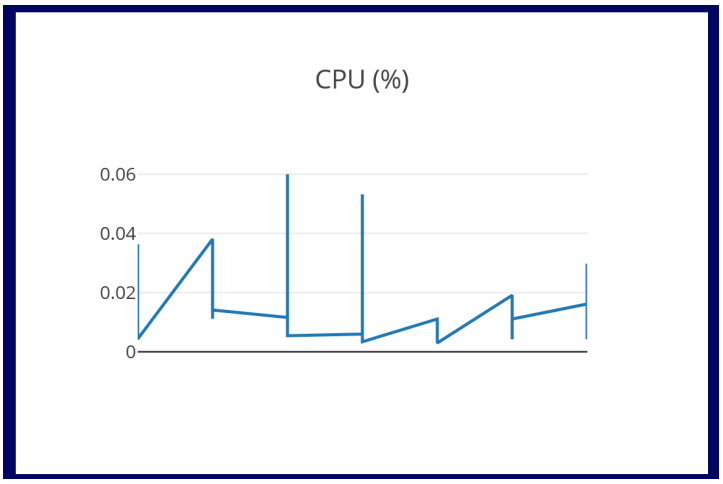

Fig. 25. Utilizarea CPU pentru SQL Server pentru cazul cu 2 puncte evitate

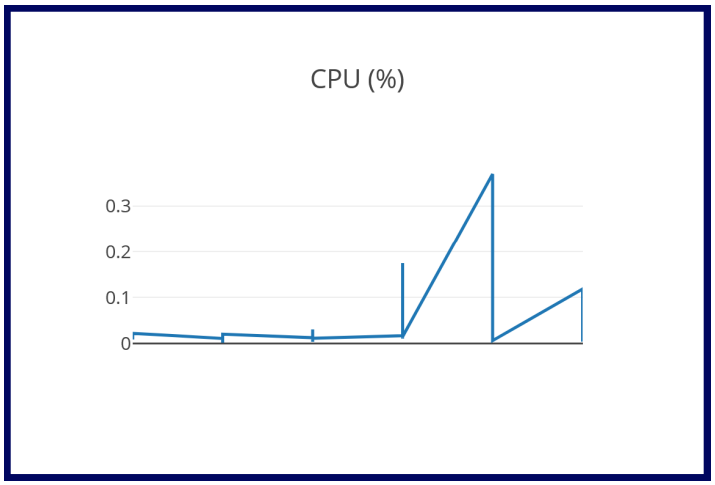

Fig. 26. Utilizarea CPU pentru SQL Server pentru cazul cu 1 punct intermediar

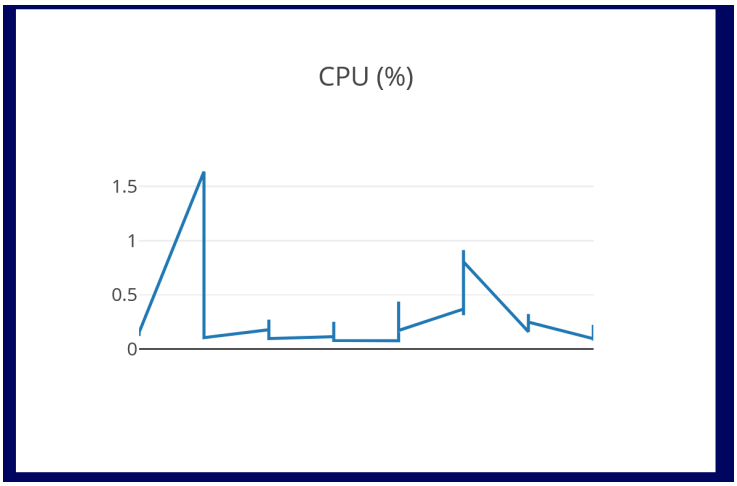

Fig. 27. Utilizarea CPU pentru SQL Server pentru cazul cu 3 puncte intermediare și 2 puncte evitate

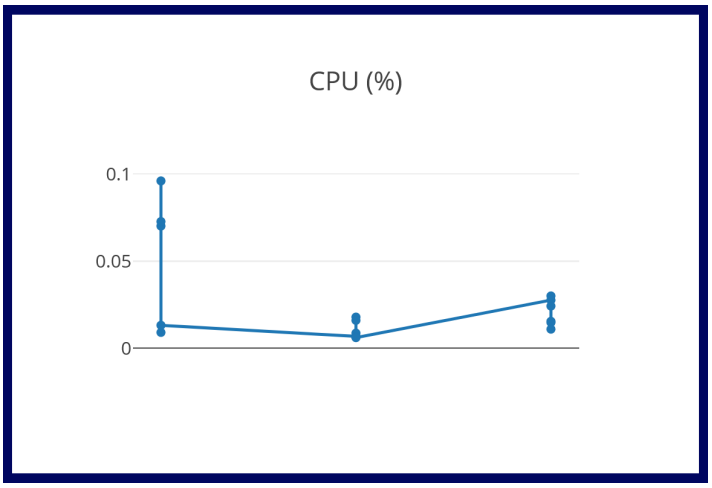

Fig. 28. Utilizarea CPU pentru SQL Server pentru cazul cu rulări manuale aleatorii

Este de remarcat faptul că SQL Server, chiar și atunci când este puternic solicitat de mai mulți utilizatori și are de realizat calcule mai complexe reușește să păstreze constant și la un nivel scăzut procentul de utilizare al CPU, ceea ce demonstrează robustețea sa și eficiența algoritmilor de optimizare din construcția sa.

În figurile 29 - 33 se poate vedea utilizarea CPU pentru Neo4j:

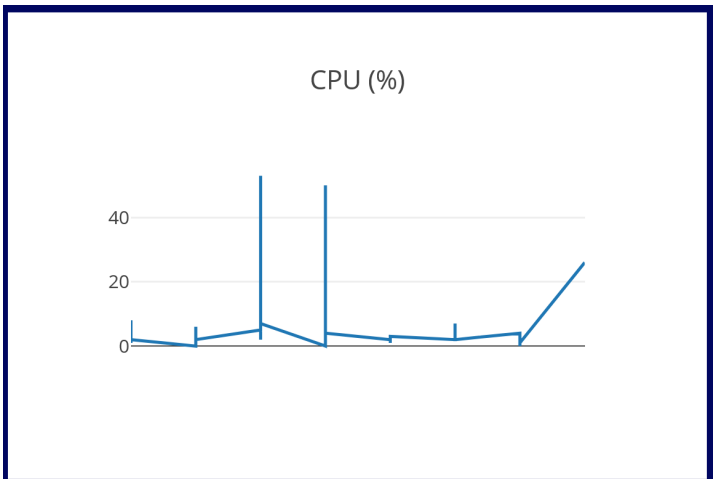

Fig. 29. Utilizarea CPU pentru Neo4j pentru cazul fără puncte intermediare/evitate

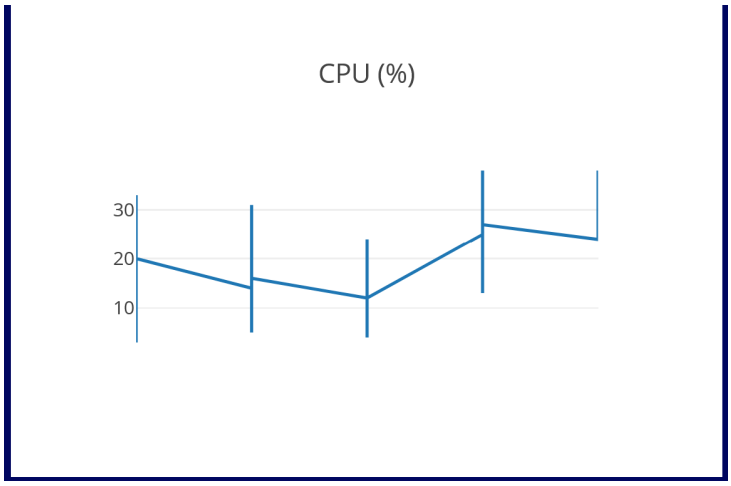

Fig. 30. Utilizarea CPU pentru Neo4j pentru cazul cu 2 puncte evitate

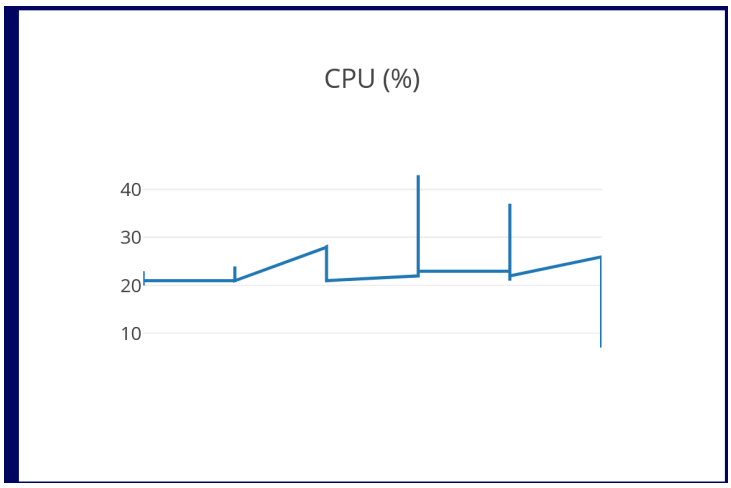

Fig. 31. Utilizarea CPU pentru Neo4j pentru cazul cu un punct intermediar

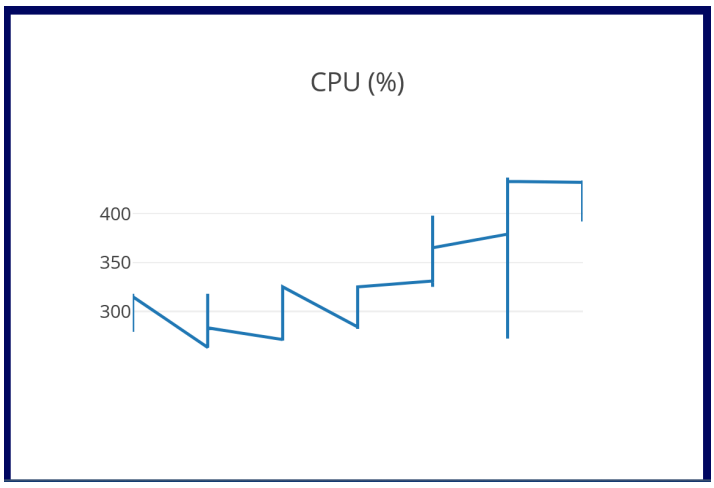

Fig. 32. Utilizarea CPU pentru Neo4j pentru cazul cu 3 puncte intermediare și 2 puncte evitate

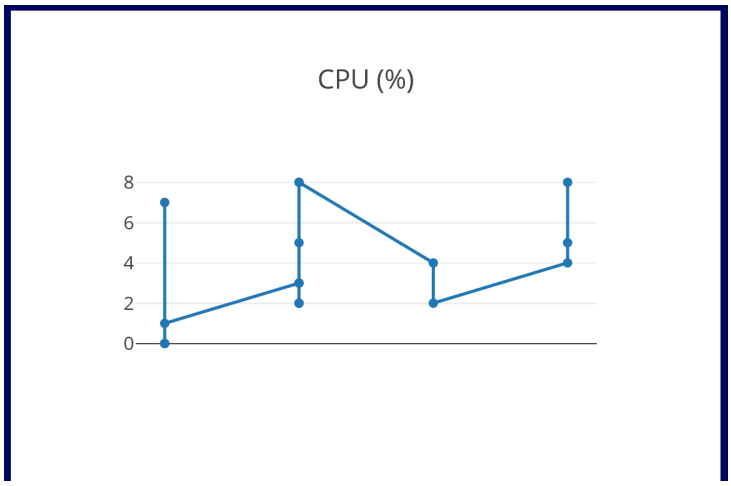

Fig. 33. Utilizarea CPU pentru Neo4j pentru cazul cu rulări manuale aleatorii

Este evident faptul că Neo4j solicită foarte mult CPU. Dată fiind și problema susmenționată a memoriei heap, se observă că întâmpină probleme atunci când primește mai multe cereri simultane, fapt ce are consecințe și asupra performanțelor. În scenariul cu 3 puncte intermediare, unde calculele devin mai complexe si mai solicitante, se pot vedea pe grafic valori ce depășesc teoretic capacitatea maximă a CPU. În realitate, există momente în care se atinge valoarea maximă de utilizare a CPU, ceea ce arată o suprasolicitare a sistemului. Calculele făcute de către aplicație, care sunt afișate și pe grafic, sunt influențate direct și de către timpii foarte mari de execuție prezentați anterior. Mai grav este faptul că dacă se mărește numărul de utilizatori care trimit simultan cereri, revine eroarea *OutOfMemory*, ceea ce conduce direct la concluzia că Neo4j nu face față la accesul concurențial, spre deosebire de SQL Server, care a făcut față chiar și la un test cu peste 100 de utilizatori simultan și a oferit rezultate similare.

În figurile 34 - 38 se poate vedea memoria utilizată pentru SQL Server:

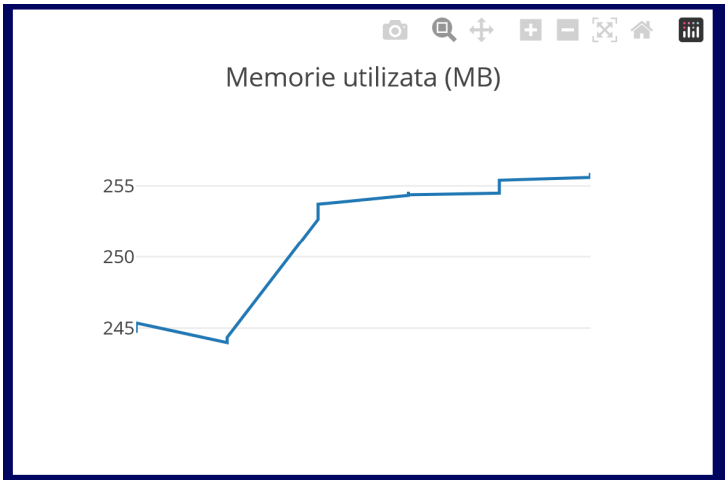

Fig. 34. Utilizarea memorie pentru SQL Server pentru cazul fără puncte intermediare/evitate

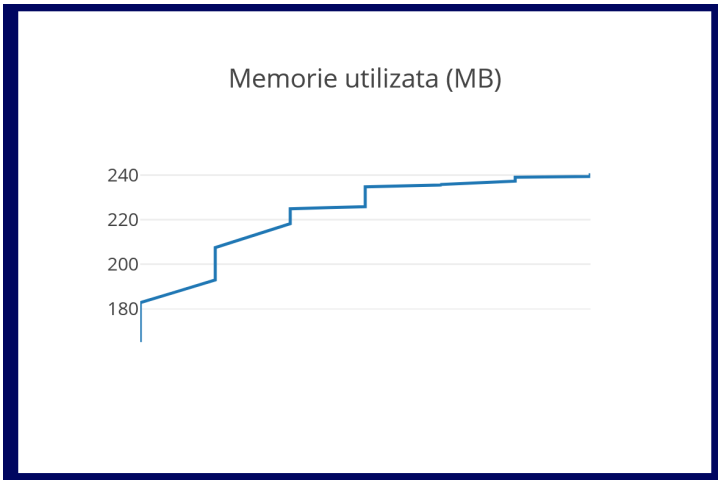

Fig. 35. Utilizarea memorie pentru SQL Server pentru cazul cu 2 puncte evitate

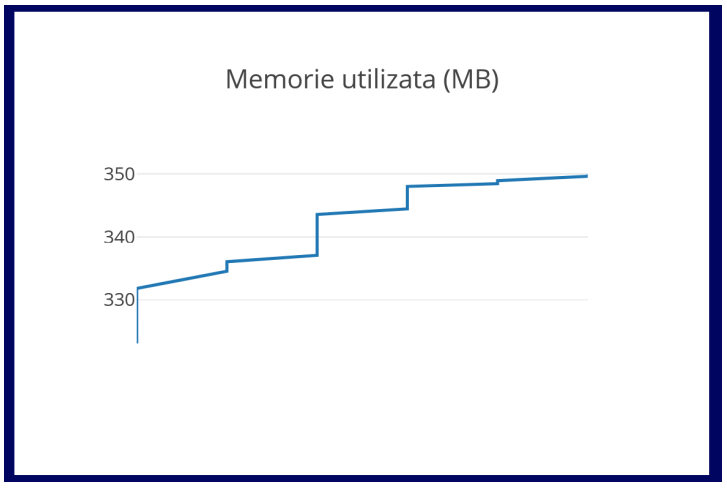

Fig. 36. Utilizarea memorie pentru SQL Server pentru cazul cu un punct intermediar

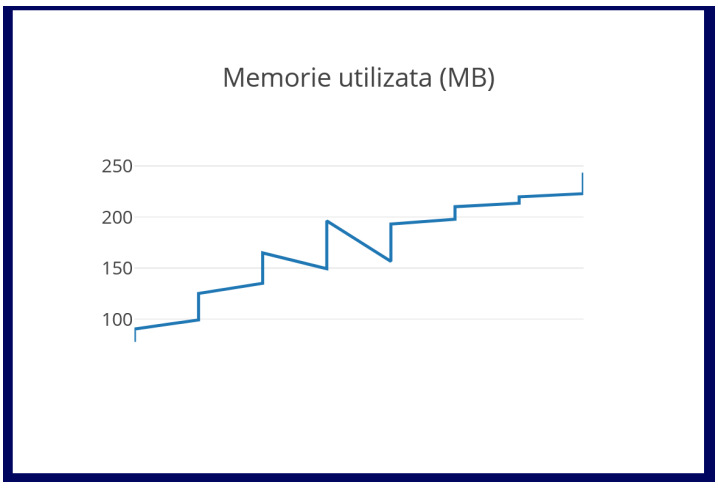

Fig. 37. Utilizarea memorie pentru SQL Server pentru cazul cu 3 puncte intermediare și 2 puncte evitate

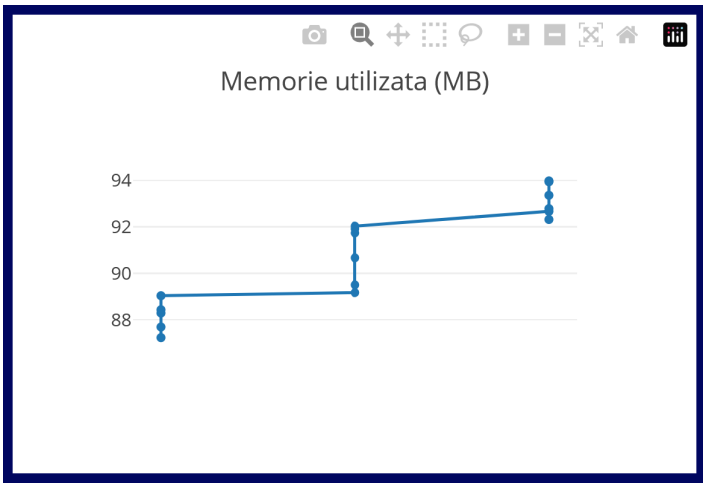

Fig. 38. Utilizarea memorie pentru SQL Server pentru cazul cu rulări manuale aleatorii

Memoria utilizată de către SQL Server se obsevă cum crește constant pe măsură ce se execută mai multe cereri. Creșterea este mai abruptă în cazurile în care există puncte evitate, deoerece acolo se creează tabele intermediare suplimentare, personalizate pentru fiecare utilizator.

În figurile 39 - 43 se poate vedea memoria utilizată pentru Neo4j:

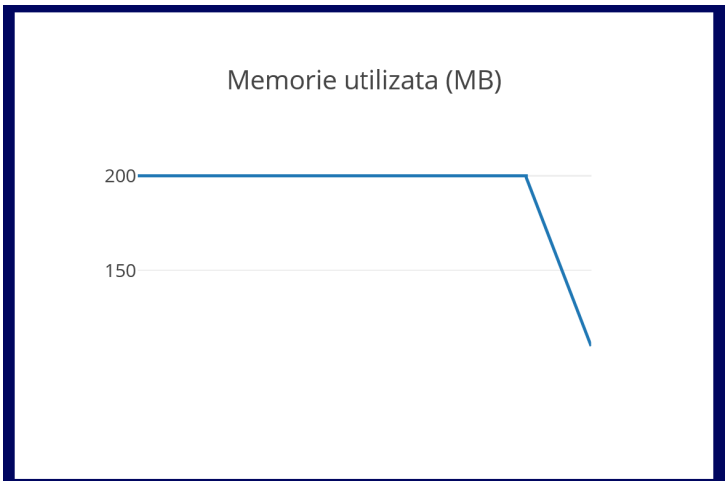

Fig. 39. Utilizarea memorie pentru Neo4j pentru cazul fără puncte intermediare/evitate

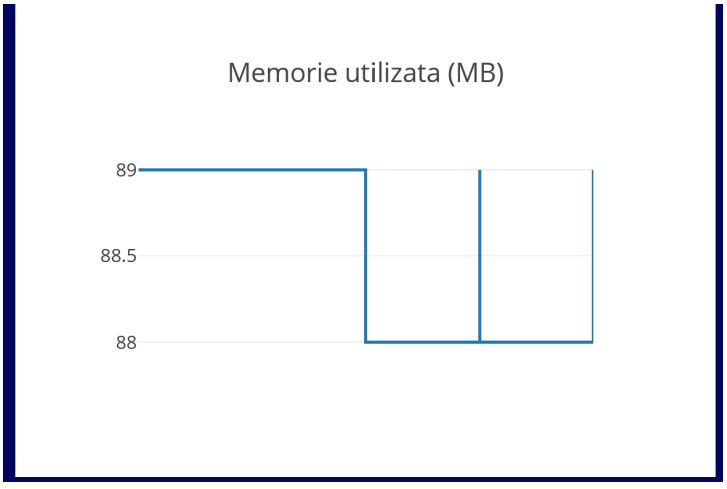

Fig. 40. Utilizarea memorie pentru Neo4j pentru cazul cu 2 puncte evitat

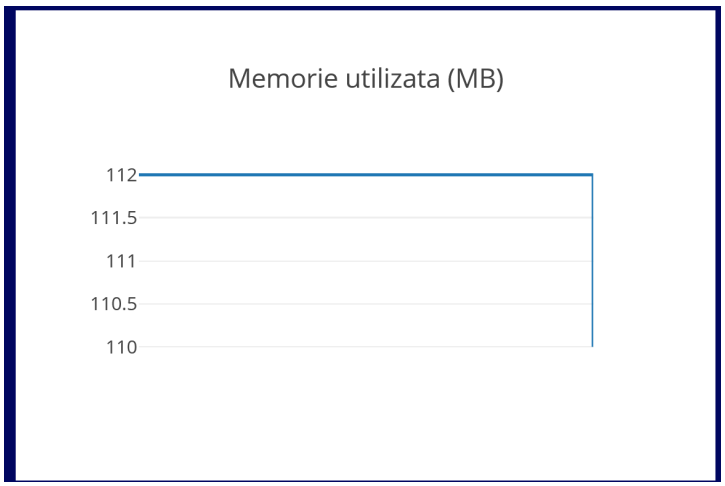

Fig. 41. Utilizarea memorie pentru Neo4j pentru cazul cu un punct intermediar

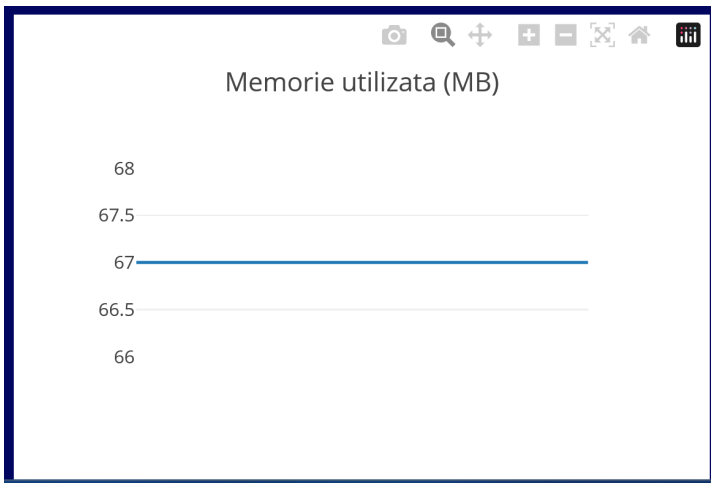

Fig. 42. Utilizarea memorie pentru Neo4j pentru cazul cu 3 puncte intermediare și 2 puncte evitate

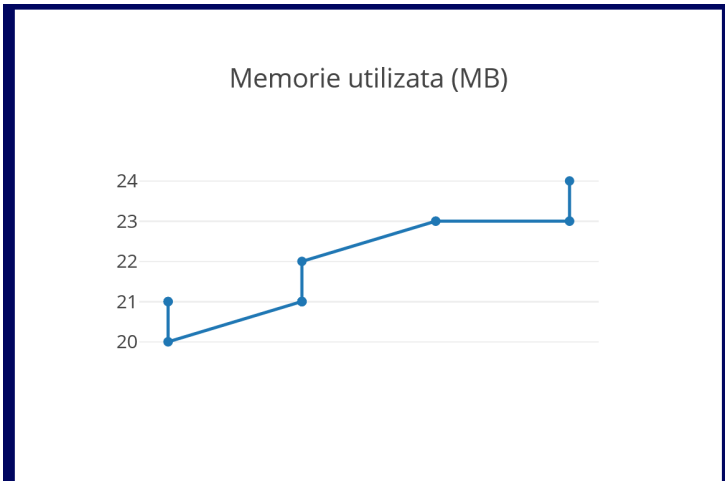

Fig. 43. Utilizarea memorie pentru Neo4j pentru cazul cu rulări manuale aleatorii

Primul lucru care se observă la Neo4j este faptul că memorie este constantă în toate scenariile, eventualele fluctuații fiind foarte mici. Fenomenul se explică prin faptul că Neo4j folosește pentru stocarea variabilelor memoria heap din JVM.

Până acum, Neo4j nu a depășit din puncte de vedere al metricilor de rulare SQL Server, ba chiar s-a dovedit mai ineficient și chiar incapabil să deservească în anumite situații toți utilizatorii. Încă o discuție în acest capitol s-ar putea face totuși și despre comportamentul lui Neo4j în aceleași scenarii de testare, dar nu rulat pe calculatorul unde este dezvoltată și aplicația, ci pe o instanță de pe o platformă de Cloud, în care resursele sunt mai flexibile, iar monitorizarea este mai precisă și detaliată, deoarece, în contextul unei aplicații reale, trendul este reprezentat de integrarea serviciilor Cloud. Pentru Neo4j, AuraDB reprezintă o platformă de Cloud ce pune la dispoziție instanțe dedicate pentru rularea acestei baze de date.

Am creat instanța în AuraDB folosind un fișier .dump al bazei de date aflate local, astfel încât să se regăsească exact același graf. Instanța are ca și resurse 2 CPU, 8Gb de memorie și 16 Gb de spațiu de stocare și rulează Neo4j versiunea 5. În interfața aplicației, am introdus ca opțiune la selecția bazei de date și posibilitarea de a rula Neo4j din AuraDB și am modificat și codul din Neo4jController.cs asftel încât să poată lucra atât cu varianta locală cât și cu cea de pe Cloud.

Baza de date aflându-se de data aceasta pe o instanță Cloud, nu se mai pot extrage date în aplicație decât pentru timpii de utilizare. Utilizarea CPU și memoria vor fi urmărite direct din consola AuraDB. Rezultatele de această dată au fost următoarele, vizibile în figurile 44 - 48:

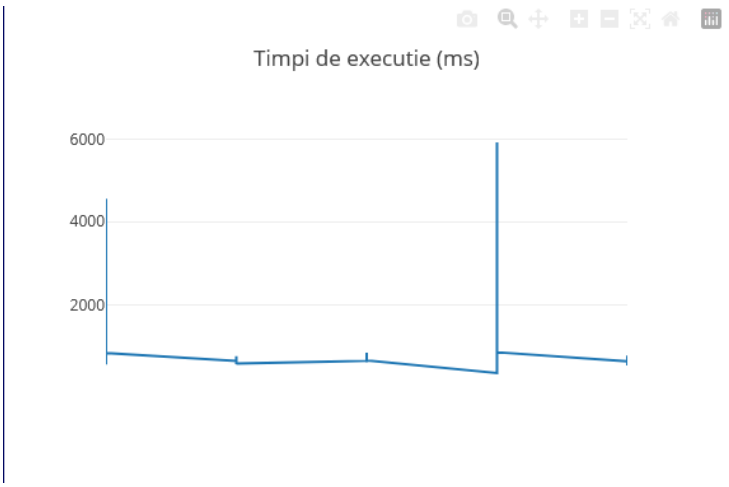

Fig. 44. Timpi de execuție pentru Neo4j (AuraDB) pentru cazul fără puncte intermediare/evitate

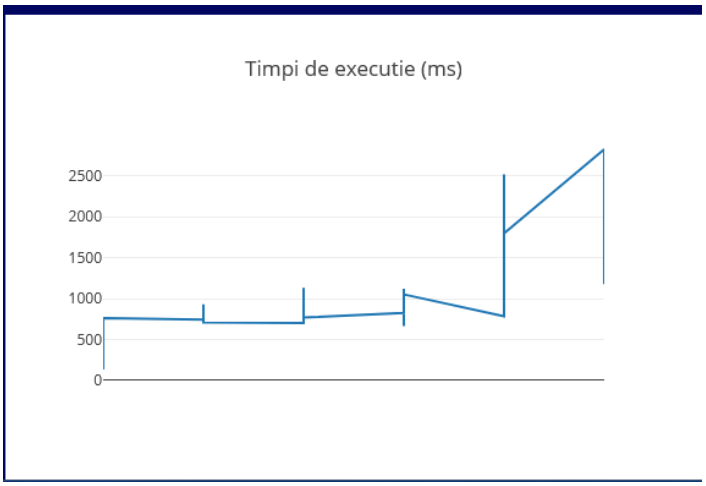

Fig. 45. Timpi de execuție pentru Neo4j (AuraDB) pentru cazul cu 2 puncte evitate

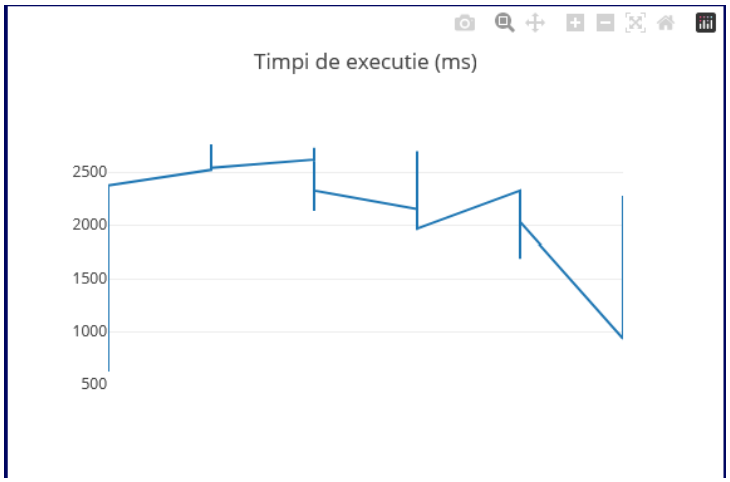

Fig. 46. Timpi de execuție pentru Neo4j (AuraDB) pentru cazul cu un punct intermediar

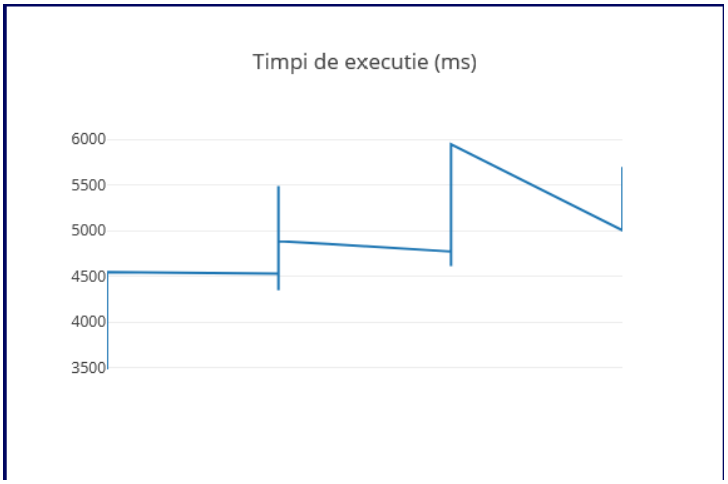

Fig. 47. Timpi de execuție pentru Neo4j (AuraDB) pentru cazul cu 3 puncte intermediare și 2 puncte evitate

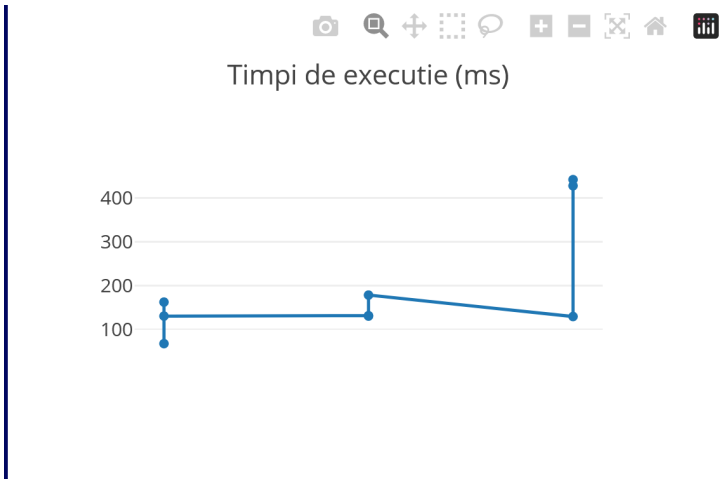

Fig. 48. Timpi de execuție pentru Neo4j (AuraDB) pentru cazul cu rulări manuale aleatorii

Timpii se observă că sunt ceva mai bine față de versiunea locală, însă tot sunt mai slabi față de SQL Server. Pentru utilizarea CPU, am extras o singură captură de ecran, figura 49, cu evoluția pe parcursul tuturor testelor:

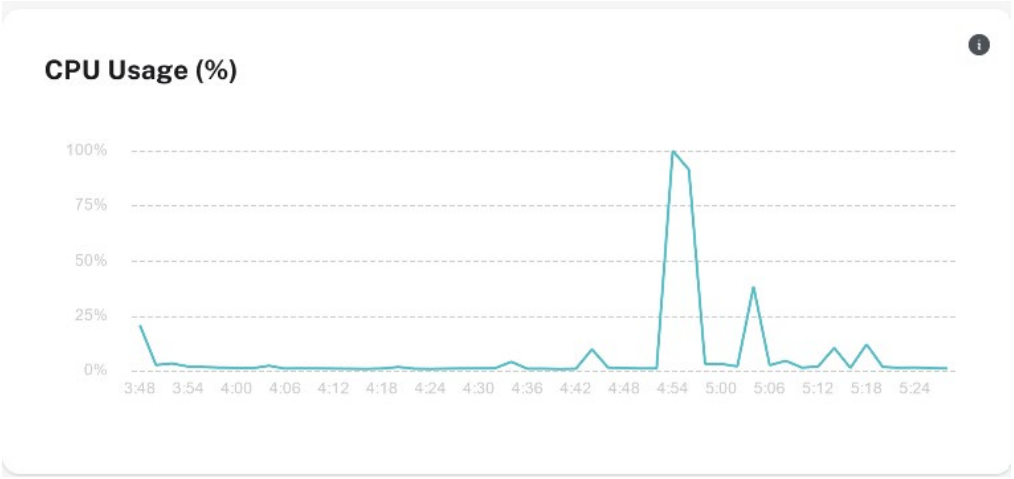

Fig. 49. Procentul de utilizare CPU pentru Neo4j (AuraDB)

Cu excepția scenariului cu 3 puncte intermediare, pentru celelalte performanța se arată mai bună față de varianta locală. Chiar dacă mai există salturi în valori, se păstrează o limită decentă. Pentru cazul cu 3 puncte intermediare se poate vedea acel salt la 100% al CPU. Testul, la fel ca pentru celelalte scenarii, a fost rulat cu 50 de utilizatori simultan. Din păcate, instanța a înregistrat saturarea CPU, dar și erori *OutOfMemory*, și trecut subit în modul offline. Și-a revenit rapid după încheierea testului, dar nu a putut deservi niciun utilizator corespunzător. A trebuit rulat testul doar cu 30 de utilizatori ca să funcționeze corect.

O altă metrică care este monitorizată de AuraDB este utilizarea memoriei heap, care se poate vedea în figura 50:

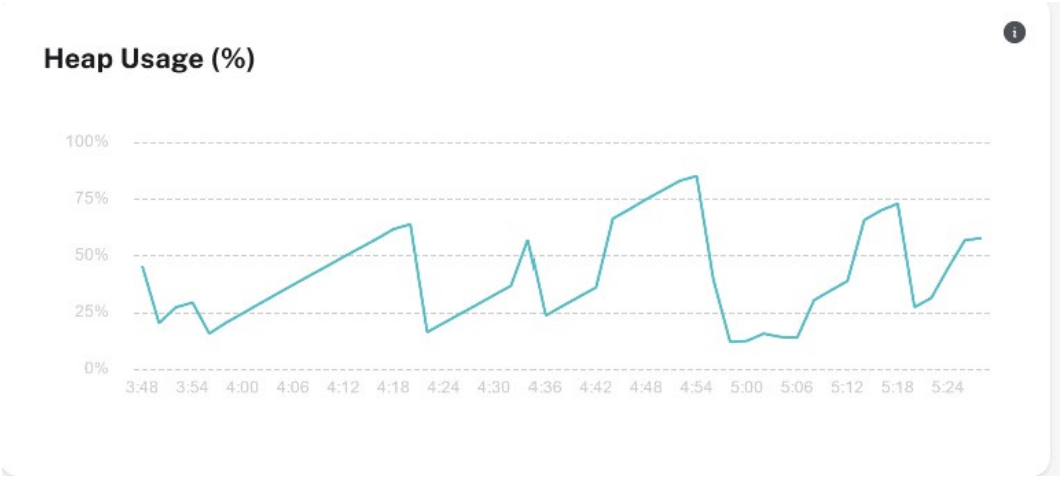

Fig. 50. Procentul de utilizare a memoriei heap pentru Neo4j (AuraDB)

Analizând graficul, se poate afirma faptul că Neo4j este un sistem consumator de resurse, având în vedere cantitatea mică de informații stocată în baza de date pentru care nu ar putea fi folosiți mai puțini de 8Gb de memorie pentru a asigura o funcționare corespunzătoare.

Un al grafic pe care îl oferă AuraDB este o evidență a erorilor de tip *OutOfMemory.* În Neo4j aceasta este o problemă comună, care trebuie monitorizată pentru a determina momentul în care trbuie adăugate noi resurse sistemului. În figura 51 am făcut o captură de ecran a acestui grafic. Înregistrările vizibile în grafic sunt asociate momentului în care am testat scenariu cu 3 puncte intermediare.

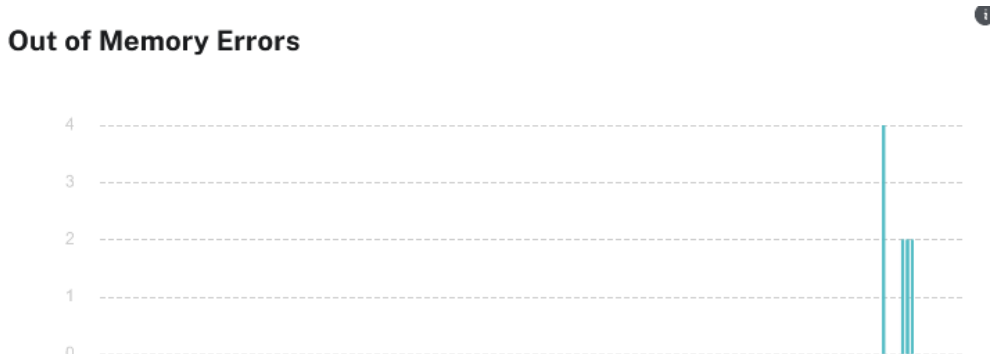

Fig. 51. Statistica erorilor de tip *OutOfMemory* Pentru Neo4j (AuraDB)

5.2. Analiză individuală a fiecărei baze de date

Ambele baze de date, așa cum am menționat și în capitolul **3.2,** pun la dispoziție unelte prin care se pot analiza interogările întocmai cu scopul de a analiza performanțele și a identifica procesele cele mai costisitoare din pnct de vedere al memoriei sau al CPU utilizate.

5.2.1. SQL Server

Deja am precizat faptul că SQL Server dispune de 2 unelte pentru analiza de interogări, și anume *Display Estimated Execution Plan* și *Include Actual Execution Plan.*

Fiecare din ele prezintă statistici pentru fiecare interogare se execută în cadrul procedurii stocate. Pentru a îmi creea a imagine despre cum se distribuie resursele în cadrul rulării și ce procedee sunt cele mai consumatoare de resurse, am rulat ambele planuri pentru procedura stocată scrisă, folosind diverși parametrii pentru rulare, și am observat operațiile care prezintă procente diferite de 0.

Display Estimated Execution Plan prezintă planul de execuție al procedurii stocate fără însă să o și ruleze. Planul este ilustrat sub forma unui arbore, așa cum se poate vedea și în figura 23 și se pot vedea toate structurile decizionale și operațiile împreună cu procentul de cost asociat. SQL Server calculează aceste procente pe baza resurselor pe care estimează acesta că le folosește și aproximează o valoare egală cu procentul din totalul setului de operații. Se observă faptul că majoritatea operațiilor sunt estimate cu 0%, la deschiderea detaliilor afișând valori infime ale tuturor metricilor, de ordinul 10^{-4} sau 10^{-5}. Operațiile cele mai consumatoare de resurse sunt cele de Cluster Index Update/Scan/Seek/Insert, fiecare având alocat aproximativ 1-3% din totalul procedurii stocate. Am rulat cu mai multe seturi de parametrii și estimările au fost aceleași de fiecare dată.

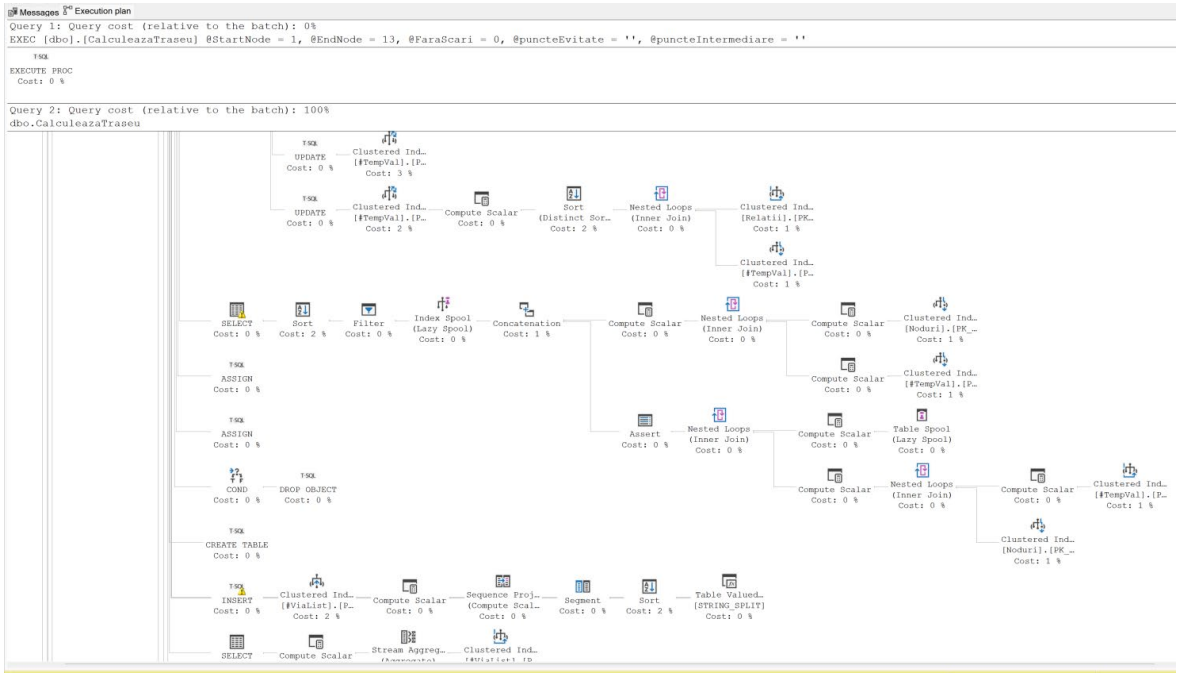

Fig. 52. Rezultat *Display Estimated Execution Plan*

Include Actual Execution Plan prezintă planul de execuție întocmit în urma rulării, oferind date reale cu privire la resursele utilizate la fiecare interogare. Prima diferență observată este că informația nu se mai arată sub formă de arbore, ci ca o listă în care se prezintă fiecare pas împreună cu procentul de cost raportat la costul total, cu o schemă în detaliu cu pași intermediari, la rândul lor cu procentele de cost asociate, așa cum se poate vedea în figura 53.

Trebuie remarcat faptul că acest procedeu de extragere al planului real de rulare este extrem de costisitor pentru SSMS. Atunci când am rulat doar cu punctul de plecare și punctul destinație ca și parametrii, timpul de răspuns a fost unul rezonabil, însă în momentul în care am încercat să testez scenariul în care se trimit 2 puncte evitate și 3 puncte intermediare ca și parametrii suplimentari, nici după zeci de minute de așteptare nu a mai venit un răspuns de la SSMS, fapt care, din păcate, poate fi un impediment în depanarea unei proceduri complexe într-o situație reală. În cele din urmă, ce am putut observa din rezultatele pe care am putut să le obțin faptul că resursele se distribuie uniform între pași, fiecare având alocat 1-2% din resursele utilizate.

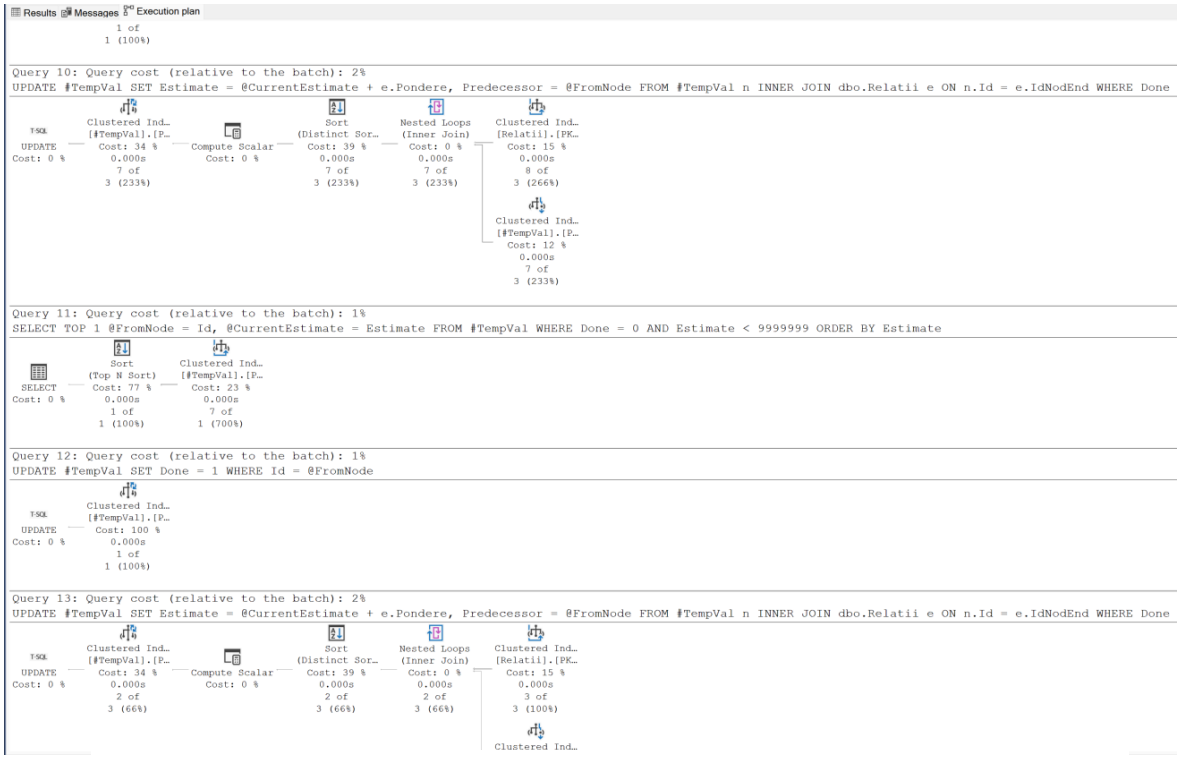

Fig. 53. Rezultat *Display Actual Execution Plan*

Din rezultatele obținute, făcând o comparație între cele 2 planuri, constat că SQL Server reușește să își optimizeze din mers resursele în funcție de complexitatea pe care o are setul de interogări pe care trebuie să le proceseze.

5.2.2. Neo4j

Pentru Neo4j interogările se pot analiza cu ajutorul comenzii PROFILE. Neo4j nu oferă posibilitatea de a scrie proceduri stocate precum bazele de date relaționale, algoritmul fiind implementat printr-o combinație de interogări Cypher cu cod C#, așa că nu se poate extrage din Neo4j nicio statistică cu privire la întreg procedeul, ci se analizează pe rând interogările de interes. În cazul meu, interogarea de interes ce merită analizată este cea care invocă algoritmul de calcul al traseului din GDS.

Spre deosebire de SSMS, PROFILE oferă rezultate foarte rapid indiferent de setul de parametrii, cu toate că acest lucru poate fi argumentat și prin faptul că nu analizează decât o interogare, nu o procedură întreagă. Informațiile oferite aici sunt timpul total de rulare, numărul de accesări ale bazei de date și memoria utilizată la fiecare pas. O altă diferență majoră în măsurătorile făcute cu PROFILE este că nu se poate verifica decât un singur scenariu, acela în care nu există puncte intermediare sau evitate. Punctele evitate nu influențează, deoarece ele nu sunt incluse de la început în graful virtual pe care se calculează traseul, iar punctele intermediare nu sunt incluse in interogarea Cypher, ci în logica C# care trimite interogări repetate către Neo4j.

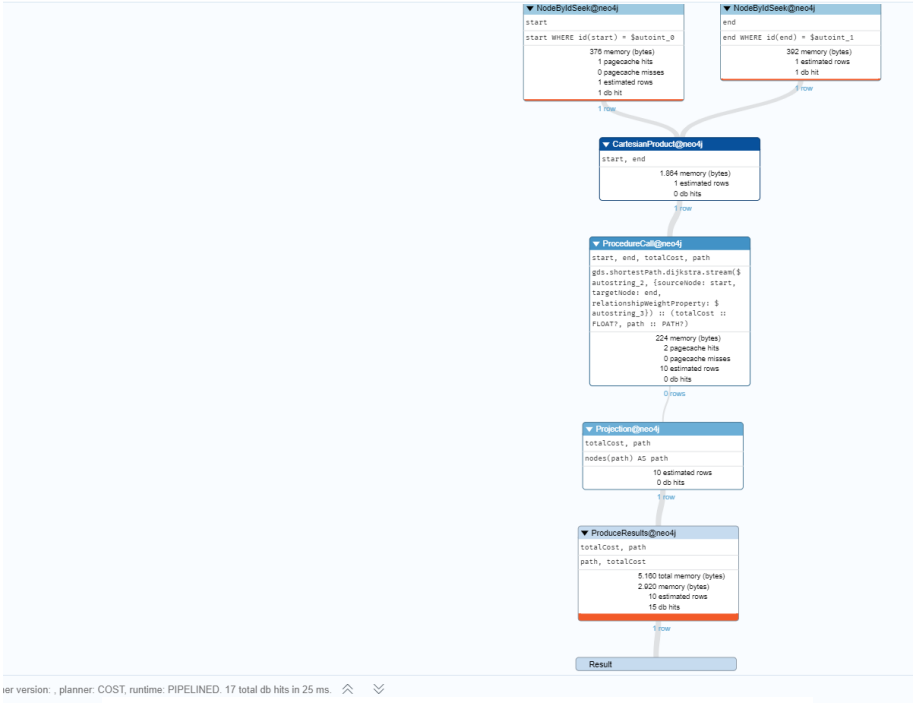

Fig. 54. Rezultatul PROFILE

Ce se poate observa în urma unor rulări repetate cu diferite puncte de plecare și destinație este faptul că lungimea traseului rezultat influențează numărul de intrări în baza de date, timpul de execuție și memoria alocată, însă tind către valori medii, vizibile chiar în figura 54.

5.3. Limbaje de interogare

SQL Server lucrează folosind limbajul de interogare SQL, la fel ca orice bază de date relațională. Neo4j lucrează exclusiv cu Cypher, unul din limbajele de interogare folosite pentru bazele de date de tip graf.

SQL este limbajul unic de interogare al bazelor de date relaționale, în oricare sistem fiind valabil standardul T-SQL. Orice interogare sau procedură stocată se poate scrie la fel în orice bază de date de acest tip. Într-adevăr, mai pot apărea mici diferențe între sisteme atunci când vine vorba de sintaxă sau funcții predefinite. Spre exemplu, clauza GROUP BY în MySQL poate fi omisă în anumite interogări, spre deosebire de SQL Server care obligă folosire ei în interogări ce cuprind funcții agregate.

Cypher, pe de altă parte, este unul din multele limbaje ce au fost dezvoltate pentru bazele de date de tip graf. Nu există un standard la care să adere toate. Fiecare limbaj de interogare are caracteristicile lui. Principalul concurent al Cypher este Gremlin, un limbaj care se concentrează pe procesul de parcurgere al grafului decât pe rezultatul în sine. O bază de dată care folosește acest limbaj este JanusGraph. AQL este un alt limbaj, folosit de către ArangoDB, un alt sistem concurent pentru Neo4j.

5.3.1. Gestionarea tranzacțiilor

Unul dintre cele mai importante aspecte ale oricărui limbaj de interogare reprezintă modul în care sunt gestionate tranzacțiile. Atât SQL cât și Cypher dispun de mai multe nivele de izolare a tranzacțiilor, astfel încât să asigure consistența datelor. SQL suportă 4 nivele de izolare [11]:

- *read uncommitted:* cel mai slab nivel de izolare, deoarece nu blochează citirea datelor din înregistrări care au fost modificate de alte tranzacții, dar care nu au fost comise încă, permițând astfel citirile murdare (*Dirty read*);
- *read committed:* spre deosebire de *read uncommitted,* se previn citirile murdare din baza de date, deoarece se blochează înregistrările care sunt modificate de alte tranzacții și nu se permite citirea acestora până când nu sunt comise; Este și nivelul de izolare implicit al tranzacțiilor din SQL Server;
- *repeatable read:* o vulnerabilitate a tranzacțiilor neizolate corespunzător poate fi citirea nonrepetată (*nonrepeatable reads*) a unei înregistrări care este accesată de mai multe tranzacții, situație care poate duce la returnarea de date diferite, ceea ce contrazice principiul consistenței; Pentru combaterea acestei probleme, se poate utiliza acest nivel de izolare care blochează o înregistrare atunci când este citită de o tranzacție până când aceasta se finalizează;
- *serializable*: cel mai puternic nivel de izolare; se poate întâmpla ca în timpul execuției unei tranzacții să fie introduse noi înregistrări de către alte tranzacții care să poată fi găsite de prima tranzacție la o nouă citire deoarece îndeplinesc condiții comune, fenomen numit citiri fantomă (*phantom reads*); acest nivel de izolare blochează introducerea de noi date ce ar putea conduce la această situație.

În Cypher se pot întâlni deasemenea problemele menționate mai sus, citirile murdare, fantomă sau nerepetabile, însă nu există un standard al nivelelor de izolare specific Cypher, ci depinde de baza de date care este utilizată. Spre exemplu, Memgraph, o altă bază de date care folosește Cypher ca limbaj de interogare, suportă ca nivele de izolare *read uncommited, read committed,* care previne citirile murdare, și *snapshot isolation,* un nivel implicit și specific acestui sistem care este echivalent cu *serialization* din SQL [12]. Neo4j, baza de date studiată în lucrarea de față, suportă următoarele nivele, conform dicumentației [9]:

- *read commited:* nivelul implicit, cel mai slab, poate fi considerat echilavent cu *read uncommitted* din SQL, însă documentația Neo4j precizează faptul că există posibilitatea de a bloca totuși scrierea într-o resursă dacă există o dependență directă, spre exemplu, o comandă de forma *SET n.prop = n.prop + 1;*

- *serializable:* blochează explicit nodurile și relațiile utilizate de o tranzacție și este echivalent cu nivelul cu același nume din SQL.

5.3.2. Operații CRUD

Orice limbaj de interogare are la bază operațiile CRUD (*Create-Read-Update-Delete*), cu ajutorul cărora se construiește orice interogare. Din punct de vedere al sintaxei, Cypher nu diferă foarte mult de SQL, fapt care îl face ușor de învățat, însă are și particularități specifice tipului de structură pe care trebuie să o interogheze

- **Inserarea datelor („Create”)**

În SQL, introducerea de noi date în baza de date se face folosind doar clauza INSERT, în interogare specificându-se apoi tabelul în care se inserează, coloanele care trebuie completate și valorile. Cypher oferă 2 opțiuni în acest sens. Prima clauză utilizată este CREATE, care poate crea un nod sau o relație între 2 noduri cu un set de proprietăți date. Nu este nevoie să fie specificat graful, pentru că se consideră că acesta reprezintă întreaga bază de date în care se lucrează, ci doar setul de proprietăți ale elementului respectiv și eventual și o etichetă pentru a putea fi inclus într-o categorie. Cea de a 2-a clauză ce poate fi folosită este MERGE. Și în SQL există această clauză, doar că în cazul său este folosit mai degrabă pentru sincronizarea conținutului a 2 tabele decât pentru a introduce informații noi în baza de date. În Cypher, se poate specifica un șablon și apoi să se folosească MERGE pentru a adăuga noduri sau relații care să-l completeze, executându-se astfel și o operație de citire înainte astfel încât să se evit creerea de elemente duplicate. Față de CREATE, mai poate fi nevoie de o operație de citire. Spre exemplu, dacă există 2 noduri izolate între care doresc să creez o relație, MERGE poate fi alegerea potrivită, deoarece pot specifica șablonul, menționând și identificatorii celor 2 noduri pentru a fi clar unde se dorește creată relația, fără riscul de a crea noduri duplicate în graf.

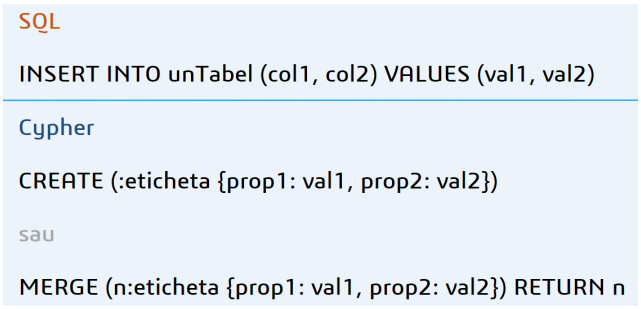

Fig. 55. SQL INSERT vs Cypher CREATE/MERGE

- **Citirea datelor („Read”)**

Pentru a citi una sau mai multe înregistrări dintr-unul sau mai multe tabele, în SQL se folosește clauza SELECT ... FROM, eventual împreună cu JOIN în cazul în care există relații între tabele și este nevoie de o alăturare a acestora pentru a obține informația căutată. Echivalentul din Cypher este MATCH ... RETURN. Câștigul cel mai mare al Cypher și, de fapt, al bazelor de date de tip graf în general, este faptul că relațiile stau la baza reprezentării datelor și nu mai este, deci, nevoie de operații suplimentare prin care să se extragă informații mai complexe atunci când există conexiuni. O interogare simplă cu o

clauză MATCH ... RETURN cuprinde doar un șablon în care se poate specifica dacă să întoarcă un nod, o relație, sau întregul șablon, indiferent de câte conexiuni există între date, fapt care reduce semnificativ o interogare la o singură linie, spre deosebire de SQL unde, pe o bază de date complexă în care se pot găsi și zeci de tabele, o interogare poate avea câteva clauze JOIN, în funcție de relațiile dintre tabele.

Desigur, atât în SQL cât și în Cypher se pot adăuga condiții pe baza cărora să fie extrasă o anumită informație și să fie afișată într-o anumită ordine. Pentru ușurința utilizatorului, Cypher folosește exact aceleași clauze ca și SQL, WHERE și ORDER BY, cu toate că în Cypher o parte din condiții se pot scrie și direct în clauza MATCH, cum se poate vedea în figura 56. În cazul funcțiilor agregate, cum ar fi COUNT(), AVG(), MIN() sau MAX(), prezente în Cypher așa cum sunt și în SQL, nu mai este nevoie de clauza GROUP BY în Cypher, așa cum este în SQL, gruparea datelor fiind realizată automat de către funcția agregată.

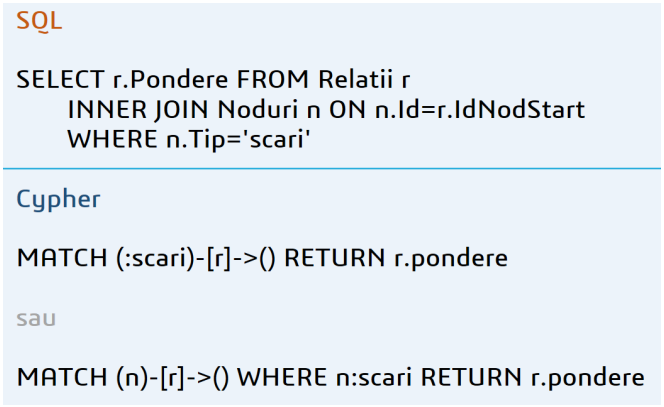

Fig. 56. SQL SELECT vs Cypher MATCH

- **Actualizarea datelor („Update”)**

Pentru a actualiza informația, în SQL se folosește clauza UPDATE. În Cypher clauza se numește SET, iar sintaxa completă cuprinde și o clauză MATCH care selectează nodul sau relația ce se dorește actualizată. Clauza WHERE în ambele limbaje poate fi prezentă astfel încât să se actualizeze un set de date care îndeplinesc anumite condiții.

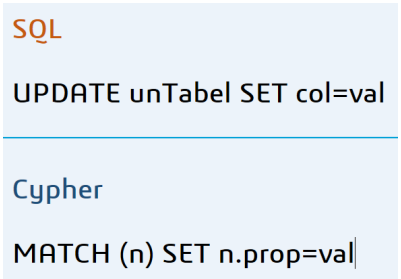

Fig. 57. SQL UPDATE vs Cypher SET

- **Ștergerea datelor („Delete”)**

SQL oferă o soluție simplă, clauza DELETE pentru ștergerea de înregistrări dintr-un tabel. Din nou, Cypher oferă o gamă mai largă de posibilități. Prima variantă este clauza DELETE, folosită pentru ștergerea unei relații fără a afecta nodurile pe care le leagă sau a unui nod izolat. Pentru ștergerea unui nod împreună cu toate relațiile conectate la acesta se folosește clauza DETACH DELETE. Pentru ștergerea unor proprietăți sau etichete ale unui nod sau ale unei relații se folosește clauza REMOVE. Toate cele 3 variante au nevoie de o clauză MATCH pentru selectarea nodurilor și/sau a relațiilor pentru a alcătui sintaxa corectă.

```
SQL
DELETE FROM unTabel
-----------------------------------------------------
Cypher
MATCH (n) DELETE n              //Nodurile nu sunt conectate între ele
sau
MATCH (n) DETACH DELETE n   //Se șterg și relațiile prin care sunt legate
sau
MATCH (n) REMOVE n.prop      //Se șterge doar proprietatea "prop" a nodului
```

Fig. 58. SQL DELETE vs Cypher DELETE/DETACH DELETE/REMOVE

5.3.3. Programarea procedurală

Pentru a implementa algoritmi complecși sunt necesare, pe lângă clauzele pentru interogări, și elemente de programare procedurală, care presupun instrucțiuni pentru blocuri decizionale de tip *if/switch* și bucle repetitive de tip *while* și *for/foreach.* Dacă pentru SQL există instrucțiuni simple IF ... ELSE, CASE și WHILE, similare cu cele regăsite în limbajele de programare procedurale, cum ar fi C++ sau Python, pentru Cypher sintaxa este mai dificilă și mai greu de implementat de către utilizator. Din acest motiv, dacă se interoghează baza de date prin intermediul unei aplicații, asemenea celei prezentate în această lucrare, o soluție poate fi combinarea interogărilor Cypher cu instrucțiuni din limbajul aplicației pentru a ușura implementarea de algoritmi complecși, iar în Cypher să se folosească instrucțiuni, fie native fie incluse în plugin-uri, în măsura în care condițiile depind direct de datele din tabel. Un exemplu de acest fel am utilizat în implementarea aplicației mele, vizibil în figura 59, pentru a verifica dacă există deja un graf virtual în baza de date, iar în cazul în care nu există să fie creat unul. Aici am folosit implementarea oferită de plugin-ul APOC.

```
CALL apoc.when(NOT gds.graph.exists("hartaCompleta"),
    "MATCH (n)-[r]→(m)
    WITH gds.graph.project('hartaCompleta',n,m,{ relationshipProperties: r { .pondere } })
    AS grafComplet RETURN 'ok'"
)
```

Fig. 59. Structură de decizie Cypher cu *apoc.when()*

Pentru scrierea unei instrucțiuni de decizie în Cypher, codul din figura 29 nu arată singura posibilitate. Din contră, această versiune nu permite decât citirea, nu și scrierea în graf, în acest caz fiind nevoie de apelarea *apoc.do.when()*. Documentația Neo4j [9] precizează faptul că Cypher are nativ instrucțiunea CASE pentru scrierea de expresii

condiționale, însă se poate spune că este echivalentă mai degrabă cu instrucțiunea CASE din SQL, permițând implementarea de condiții multiple față de care se verifică o expresie. Și APOC cuprinde o implementare a instrucțiuni CASE, *apoc.case()* și *apoc.do.case(),* doar cel din urmă având drept de scriere în graf.

Pentru scrierea de bucle repetitive în SQL există teoretic instrucțiunea FOR LOOP, însă aceasta este construită diferit pentru fiecare sistem. Pentru SQL Server, se folosește instrucțiunea WHILE. De remarcat este faptul că în Cypher nu există un echivalent pentru WHILE. Există însă 2 instrucțiuni pentru crearea de bucle repetitive cu număr dat de pași, FOREACH și UNWIND, la rândul lor acestea neavând echivalent în SQL. FOREACH, asemenea instrucțiunii cu același nume din limbaje de programare precum C# sau Java, poate parcurge o listă de noduri, spre exemplu, și poate executa interogări la fiecare iterație. UNWIND oferă rezultate similare, dar folosește un element de tip listă pe care îl convertește într-un set de date ce se pot trata separat. În SQL un rezultat asemănător se poate obține declarând un cursor cu care se parcurge un tabel. Figura 60 ilustrează sintaxa și modul de utilizare al fiecărei instrucțiuni.

```
UNWIND [1, 2, 3] AS x RETURN X    // --> Va afișa pe câte o linie separată valorile din listă

MATCH (p:scari)
FOREACH (n IN nodes(p) | SET n.name="scariNoi")  //--> Va redenumi toate nodurile cu eticheta "scari" în "scariNoi"
```

Fig. 60. Instrucțiunile FOREACH și UNWIND

Un caz prezentat în [10] și interesant de menționat ar fi faptul că instrucțiunea FOREACH din Cypher mai poate fi utilizat împreună cu CASE pentru construcția unui bloc de decizie. Nu este foarte intuitivă pentru utilizator această combinație, însă se poate folosi pentru a executa un set de interogări doar în cazul în care există elemente care îndeplinesc o anumită condiție.

În cazul particular al Neo4j, fiind un sistem construit în Java, mai există posibilitatea de a crea proceduri ce pot fi chemate în Cypher sub formă de plugin-uri, scrise direct în Java. Poate fi o alternativă bună pentru implementarea unui algoritm lung și complex, deoarece se folosește în întregime un limbaj comun de programare, ușor de structurat, însă nu numai că trebuie instalate dependențele necesare pentru a creea plugin-uri pentru Neo4j, dar pentru a include procedura în sistem, codul Java trebuie de fiecare dată arhivat și plasată arhiva în folderul care conține plugin-urile folosite de către Neo4j. Nu există în acest sens o funcționalitate similară cu cea de creere de proceduri stocate din SQL Server Management Studio care să faciliteze procesul.

5.3.4. Gestionarea indecșilor

În orice bază de date, indecșii sunt folosiți pentru a îmbunătății performanțele, reducând cantitatea de date care trebuie citite pentru obținerea unei informații.

SQL are posibilitatea de a crea 6 tipuri de indecși:

- *Clustered Index:* implementat ca o structură de indexare de forma arbore-B [11], stochează înregistrările ordonate după o cheie;

- *Non-cluesterd Index:* se poate defini pe un tabel sau vedere și fiecare linie a sa conține cheia non-clustered și un pointer către înregistrarea aferentă; se ordonează doar indexul nu și datele fizice din tabel;
- *Index Unic:* are rolul de a asigura unicitatea unei înregistrări în tabel;
- *Index Filtrat:* Este util atunci când un număr mic de înregistrări cuprinde informația relevantă pentru majoritatea interogărilor și este considerată o versiune optimizată a *non-clustered index;*
- *Columnstore Index:* Utilizat cu precădere în bazele de date asupra cărora se execută interogări ce presupun cantități mari de date citite;
- *Index Hash:* Datele sunt accesate prin intermediul unui tabel hash aflat în memorie, unde consumă o dimensiune fixă, îmbunătățind astfel performanța

Cypher are doar 4 tipuri de indecși descriși de către documentația Neo4j [9]:

- *Range Index:* indexarea implicită creată de Neo4j;
- *Text Index:* Optimizează predicatele care lucrează cu valori de tip STRING;
- *Point Index:* Optimizează predicatele care lucrează cu valori de tip POINT;
- *Token Lookup Index:* 2 astfel de indecși sunt prezenți în momentul creării bazei de date și privește doar predicatele de tip etichetă de nod sau tip de relație.

5.4. Posibilități de integrare în aplicații complexe

Bazele de date în general nu se folosesc decât integrate în aplicații mai complexe, ca sisteme pentru stocarea datelor. În acest sens, atunci când se discută despre 2 baze de date trebuie vorbit și despre posibilităție pe care fiecare le oferă pentru integrarea cu alte mecanisme.

5.4.1. Interogarea prin intermediul unei aplicații

Pentru SQL Server se găsesc drivere pentru aproape toate limbajele de programare, aceasta nefiind astfel o restricție pentru programator. Neo4j însă nu este atât de bine extins, fiind disponibile oficial și întreținute de către dezvoltatorii Neo4j drivere doar pentru .NET (C#), Java, JavaScript, Go și Python. Pentru alte limbaje de programare există doar versiuni dezvoltate open source. În acest fel, se poate menține pe piață la fel de competitiv ca și SQL Server.

5.4.2. Instrumente auxiliare

Prin instrumente auxiliare se înțeleg acele programe care oferă utilizatorului posibilitatea de a interacționa cu baza de date printr-o interfață grafică.

SQL Server Management Studio reprezintă programul dezvoltat de Microsoft prin care se poate opera în SQL Server prin intermediul interfeței grafice. Este un sistem complet care facilitează nu numai vizualizarea și executarea de interogări simple, ci se pot redacta și gestiona proceduri stocate, funcții și scripturi, tabele fără a scrie manual interogări SQL. Cum s-a evidențiat și în capitolul **5.2.1**, sunt cuprinse și instrumente

speciale pentru analizarea interogărilor cu scopul identificării de operații costisitoare ce ar putea fi optimizate. Practic, orice operație posibiliă asupra unei baze de date poate fi făcută din SSMS.

Și Neo4j pune la dispoziție aplicații pentru interacțiunea vizuală cu baza de date însă, spre deosebire de SQL Server, nu este un singur program care să centralizeze toate operațiile posibile. Aplicația de bază este Neo4j Desktop, în care se pot vizualiza și gestiona bazele de date existente pe calculator. De aici, oricare bază de date se poate deschide într-o aplicație auxiliară. Aceste aplicații auxiliare, denumite *Graph Apps,* se pot descărca ca extensii, oferind funcționalități suplimentare. Cel care se instaleaxă implicit odată cu Neo4j Desktop este Neo4j Browser, care cuprinde funcționalitățile de bază de vizualizare și interogare a unei baze de date. Un alt exemplu de astfel de aplicație auxiliară este Neo4j ETL Tool, care poate fi folosită pentru importarea datelor dintr-o bază de date relațională în Neo4j.

5.4.3. Integrarea în servicii de tip Cloud

Din ce în ce mai mult, dezvoltatorii de aplicații se concentrează pe integrarea serviciilor Cloud în produsele software, datorită posibilității de extindere rapidă și eficientă la un cost avantajos și a securității sporite. Pentru utilizarea unei baze de date în mediul Cloud o soluție poate fi utilizarea unui mașini pe care să se instaleze manual sistemul dorit și să se realizeze configurările necesare, însă există și servicii care gestionează servere dedicate pentru diverse baze de date, care au toate configurările făcute și care pun la dispoziție și instrumente de monitorizare pentru client ca să își poată monitoriza starea bazei de date în timp real.

Pentru SQL Server Microsoft a dezvoltat Azure SQL Database, un serviciu gestionat în întregime, disponibil pe platforma Cloud Microsoft Azure, care poate fi folosit ca parte a unei aplicații complexe dezvoltate folosind arhitectura Cloud. Și alți competitori ai Microsoft care dețin platforme Cloud oferă servicii similare: în Google Cloud există Cloud SQL, pe care pot rula SQL Server, MySQL și PostgreSQL, iar în AWS se poate folosi RDS (*Relational Database Service*) în care se poate rula SQL Server, MySQL, PostgreSQL, Amazon Aurora, MariaDB, Db2 sau Oracle. Pentru Neo4j principala platformă Cloud unde se poate gestiona o bază de date este AuraDB, folosită și în lucrarea de față, care își stochează serverele tot în Google Cloud, AWS sau Microsoft Azure.

5.4.4. Securitate

În momentul în care se adaugă un nou sistem la o aplicație complexă, unul din primele aspecte care trebuie avute în vedere este menținerea securității aplicației. Noul sistem introdus fie are în componența sa mecanisme de protecție fie trebuie suplimentate de aplicația care o deservește.

Din acest punct de vedere, niciuna din cele 2 baze de date nu aduce probleme de securitate. Atât în SQL Server cât și în Neo4j există posibilitatea de a crea și gestiona utilizatori multipli cu drepturi diferite, astfel încât să fie protejate bazele de date de către accesul neautorizat. Cel mult se poate considera o vulnerabilitate pentru Neo4j faptul că

sunt folosite plugin-uri externe pentru scrierea interogărilor, care pot cuprinde ele probleme în cadrul implementărilor lor.

5.4.5. Utilizarea de cod extern

În orice limbaj de programare, posibilitatea de a folosi cod extern reprezintă un mod esențial de eficientizare a muncii, refolosindu-se un cod deja existent pentru anumite proceduri, programatorul concentrându-se pe conceperea noilor implementări.

În SQL nu se pune problema de importări de biblioteci sau alte elemente care să conțină un cod deja scris și care să fie refolosit precum în alte limbaje de programare. Cel mult se pot creea proceduri stocate sau funcții, care sunt stocate în baza de date, sau scripturi ce pot fi salvate pe calculator ca fișiere .sql și reutilizate la nevoie.

În Cypher însă, un aspect important constă în utilizarea așa numitelor plugin-uri. Acestea sunt arhive Java care conțin implementări menite să suplinească lipsurile acestui limbaj de interogare, cum ar fi limbajul procedural prezentat la capitolul **5.3.3**. De asemenea, se mai pot găsi implementări pentru algoritmi uzuali astfel încât programatorul să poată reutiliza un cod existent, așa cum este *gds.shortestPath.dijkstra.stream(),* pe care l-am folosit și în lucrarea de față pentru a rula algoritmul Dijkstra. și pot fi comparate cu bibliotecile folosite în alte limbaje de programare. Cele mai utilizate plugin-uri sunt APOC și GDS, dar există multe altele pe internet sau chiar se pot crea de către utilizator, așa cum am descris la capitolul **5.3.3.**

6. Concluzii și dezvoltări ulterioare

În concluzie, prin lucrarea de față am reușit să compar atât din punct de vedere al performanțelor de rulare, cât și din puncte de vedere al integrării în contextul altor aplicații, cele mai populare sisteme de baze de date, reprezentative pentru categoriile din care fac parte, SQL Server și Neo4j. Rezultatul la care am ajuns confirmă faptul că bazele de date noSQL de tip graf se dezvoltă și de la un an la altul cresc în popularitate datorită avantajelor considerabile atunci când se lucrează cu date puternic interconectate în aplicațiile care se concentrează pe statistici și analize și care nu sunt accesate de un număr mare de utilizatori, însă și cele relaționale încă domină piața datorită robusteții lor în contextul aplicațiilor accesate masiv și a algoritmilor de optimizare utilizați.

Cum am precizat și în introducere, există numeroase articole științifice pe tema aceasta, fiecare realizând propriul lui studiu de caz și analizând după criteriile pe care le consideră cele mai relevante, cu scopul de a scoate în evideță și de a populariza tot mai mult grafurile ca și reprezentare a datelor, care cu siguranță se vor dezvolta până la punctul la care vor concura cu clasicele metode de stocare.

Referitor la aplicația dezvoltată și prezentată în această lucrare bineînțeles că există loc pentru dezvoltări ulterioare, prima și poate și cea mai importantă fiind adaptarea ca funcționare sub formă de aplicație mobilă, navigarea putând fii realizată folosind tehnologii de localizarea interioară asemenea aplicațiilor existente pe piață. Cum am precizat, tendința actuală este de construire a aplicațiilor folosind arhitectură Cloud, așa că și aplicația de față ar putea integra servicii Cloud, cum ar fi rularea pe un server Cloud a aplicației propriu-zise.

7. Bibliografie

[1] J. Pokorny: Graph Databases: Their Power and Limitations, în K. Saeed and W. Homenda (Eds.): CISIM 2015, LNCS 9339, pp. 58–69, 2015.

[2] S. Ataky, T. Mpinda, L. C. Ferreira, M. Xavier Ribeiro, M. T. Prado Santos: Evaluation of Graph Databases Performance Through Indexing Techniques, în International Journal of Artificial Intelligence & Applications, pp. 87-98, Septembrie 2015

[3] C. Rodrigues, M. R. Jain, A. Khanchandani: Performance Comparison of Graph Database and Relational Database, DOI:10.13140/RG.2.2.27380.32641

[4] S. Gilbert, N. A. Lynch: Perspectives on the CAP Theorem, în *Computer* 45.2, pp. 30-36, 2012

[5] J. Webber: The Top 10 Use Cases of Graph Database Technology, E-Book. Online: https://go.neo4j.com/rs/710-RRC-335/images/Neo4j-Top-10-Use-Cases-EN-US.pdf, accesat la data de 16.04.2024

[6] Oracle, 17 Use Cases for Graph Databases and Graph Analytics , E-Book. Online: graph-database-use-cases-ebook.pdf (oracle.com), accesat la data de 16.04.2024

[7] Indoor navigation, everything you need to know, https://www.pointr.tech/blog/indoor-navigation-everything-you-need-to-know

[8] Indoor navigation, a comprehensive guide, https://navigine.com/blog/indoor-navigation-a-comprehensive-guide/

[9] Documentație Neo4j, https://neo4j.com/docs/

[10] Neo4j Knowledge Base, https://neo4j.com/developer/kb/

[11] T-SQL reference, https://learn.microsoft.com/en-us/sql/t-sql/

[12] Memgraph transactions, https://memgraph.com/docs/fundamentals/transactions