

## Cuprins

<b>Introducere</b> .....	2
<b>Capitolul 1 – Relațional, non-relațional, post-relațional</b> .....	4
1.1 Evoluția bazelor de date.....	4
1.2 Era Big Data.....	6
1.3 Trendul NoSQL.....	7
1.4 Tipuri de baze de date non-relaționale.....	10
1.5 MapReduce: simplificarea procesării datelor din clustere mari.....	14
<b>Capitolul 2 - Modelul relațional în Oracle vs. modelul graf în Neo4j</b> .....	19
2.1 Oracle vs. Neo4j : model de date, tipuri de date, tipuri de interogări.....	19
2.2 Operații CRUD în Oracle și Neo4j.....	22
2.3 Operații de intersecție, reuniune, diferență .....	27
2.4 Funcții scalare.....	31
2.5 Funcții agregate .....	34
2.6 Funcții pentru șiruri de caractere.....	39
2.7 Alte clauze și cuvinte-cheie .....	44
2.8 Constrângeri (restricții) în Oracle și Neo4j.....	57
2.9 Elemente distincte în Neo4j .....	57
<b>Capitolul 3 - Utilizarea Apache Jmeter în analiza performanței Oracle server vs. Neo4j server</b> .....	60
<b>Concluzii</b> .....	69
<b>Bibliografie</b> .....	72

## Introducere

În mod categoric, suntem dependenți de informații, iar pentru a le obține avem nevoie de date. Din orice domeniu ar fi ele, datele necesită a fi stocate și prelucrate. Și cum tehnologia tinde să parcurgă un trend ascendent și să ofere provocări prin noi produse și servicii, nici domeniul bazelor de date nu rămâne mai prejos, astfel încât remarcăm o tendință de migrare dinspre relațional spre non-relațional. Înconjurați de sisteme care se confruntă cu probleme de scalabilitate, performanță și ghidați tot mai mult de fenomenul Big Data ancorat în cei trei “V” (volum, viteză, varietate), marii furnizori de baze de date, și nu numai, au decis că este momentul să uite de modelul propus cândva de E. Codd și să se adapteze noilor cerințe ale pieței. Astfel, NoSQL a devenit noul trend capabil să angajeze marii jucători din IT. Deși rețelele sociale au fost prima țintă, furnizorii de NoSQL promit să-și extindă aria, chiar dacă obstacolele includ “semi-prezența” sau lipsa suportului tranzacțiilor (ACID) și folosirea limbajelor de interogare nestandardizate.

Bazele de date NoSQL au propriul lor mod de accesare și manipulare a datelor și sunt non-relaționale pentru că tranzacțiile ACID și restricțiile referențiale împiedicau scalarea și gestionarea seturilor imense de date. Cu toate acestea, se dorește prezența SQL-ului în lumea NoSQL și, ca rezultat, au fost create limbaje de interogare ce se aseamănă în sintaxă și stil cu SQL-ul. Așadar, trecerea de la relațional la non-relațional nu ar fi atât de “dureroasă” din acest punct de vedere. Problema mai importantă, mai ales în cazul marilor site-uri de comerț electronic, ar fi consistența datelor. Se pare, totuși, că se concretizează o soluție alternativă, o “semi-îmbinare” între SGBD-ul relațional și NoSQL. Este vorba despre NewSQL, o clasă de SGBD-uri moderne care încearcă să combine scalabilitatea NoSQL-ului cu capabilitățile ACID ale bazelor de date relaționale. Rămâne însă de văzut dacă marea competiție va elimina RDBMS-ul de pe piață sau dacă toate produsele vor co-exista armonios în scopul rezolvării problemelor actuale și viitoare.

Alegerea realizării unei analize comparative a relaționalului și non-relaționalului a pornit din provocarea lansată odată cu acceptarea de către piață a noului trend NoSQL. Din dorința de a pune în balanță plusurile și minusurile aduse de fiecare tip de bază de date și din necesitatea de a decide dacă merită să ne îndreptăm spre NoSQL, fără niciun regret pentru modelul lui Codd, am decis să compar soluțiile oferite de Oracle și Neo4j în materie de interogare a bazei de date. Domeniul în care activează cei doi furnizori, va simți mereu nevoia de a se adapta la cerințele pieței, și cum necesitățile actuale s-au desprins din popularul fenomen Big Data, satisfacerea lor a devenit provocarea celor care au lansat curentul NoSQL.

Lucrarea de față se consideră a fi punctul de pornire în concretizarea unor idei privitoare la studiul mecanismelor de interogare a bazelor de date relaționale și non-relaționale. Analiza prevede o prezentare succintă a istoricului parcurs de bazele de date, a fenomenului Big Data - provocarea ultimei decade, a noului concept, NoSQL, și a tipurilor de baze de date pe care le oferă, încheierea aparținând modelului MapReduce – ingredient

esențial al cunoscutului Hadoop.

Toate aceste amănunte sunt sintetizate într-un prim capitol, urmând ca în cel de-al doilea să fie prezentată o analiză comparativă între Oracle (ca sistem relațional) și Neo4j (ca sistem NoSQL): model de date, tipuri de date, tipuri de interogări și exemple comparative de interogări.

În plus față de acestea, pentru o precizie mai bună în analiza comparativă a performanței ambelor tipuri de baze de date, am ales folosirea unui set mare de date, care, în urma măsurărilor realizate cu Apache JMeter, a scos la iveală faptul că Oracle rămâne în top la capitolul “timp de execuție” și “latență redusă”. Însă și aici lucrurile rămân discutabile, dat fiind faptul că alți factori pot influența semnificativ performanța bazei noastre de date.

Astfel, fie că alegem modelul relațional, sau renunțăm la el în detrimentul noului trend NoSQL, avantajele și dezavantajele puse în balanță, dar mai ales nevoile și tipul afacerii vor fi ghid în alegerile noastre.

# Capitolul 1 – Relațional, non-relațional, post-relațional

## 1.1 Evoluția bazelor de date

Odată cu apariția primelor calculatoare electronice (mijlocul sec.XX) s-a ivit oportunitatea de a dezvolta instrumente capabile să stocheze și să prelucreză informații cu o viteză mult superioară omului. Deși mulți au fost sceptici, temerile lor au fost risipite când calculatoarele electronice și-au dovedit utilitatea și au determinat o explozie în domeniul tehnologiei informației.

Ascensiunea în domeniul bazelor de date s-a concretizat prin trecerea progresivă de la era navigațională (anii '60) la cea relațională (anii '70) și ulterior post-relațională (începând cu anii '90).

În primă etapă, datele erau organizate sub forma unor fișiere secvențiale stocate pe bandă magnetică, fiecare dată fiind descrisă autonom în fiecare fișier, lucru ce a generat o serie de inconveniente precum: inconsistența și redundanța datelor, izolare și acces dificil, complexitate în actualizare, probleme legate de integritate, securitate și cost.

Însă, evoluția tehnologică și-a pus amprenta câțiva ani mai târziu, odată cu apariția hard-disk-ului (anii '70), printr-un nou model de date: cel relațional. Edgar Codd a propus o structură de date tabelară, independentă de tipul de echipamente și software de sistem pe care este implementată, ce asigură optimizarea accesului la date, o îmbunătățire a integrității și confidențialității datelor, o separare dintre structura logică și fizică a datelor, asigurând independență fizică.

În scurt timp, anii '80 au adus limbaje de programare orientate pe obiect, ceea ce a generat o necesitate a stocării informațiilor conținute de obiectele din memorie. Astfel au apărut baze de date orientate pe obiect, eliminând o mare parte din inconvenientele “strămoșilor” din erele precedente, în sensul că problemele de redundanță au fost eliminate prin identificatorul de obiecte (OID) – folosit ca mecanism pentru identitatea obiectelor, integritatea e asigurată prin caracterul unic al întregului sistem, dispar dificultățile de integrare cu programele OOP.

În anii '90, odată cu apariția Internetului, cantitatea de informații a crescut considerabil și cerințele pentru prelucrarea datelor au suferit modificări. Apariția conceptului Web 2.0 a pus în pericol SGBD-ul relațional, incapabil să mai gestioneze cantitățile imense de date. Continua creștere a volumului de date a ridicat probleme de scalabilitate ce puteau fi rezolvate prin adăugare de memorie RAM sau hard-disk-uri mai performante, însă costurile erau substanțiale și defectarea sistemului central conducea la pierderea datelor. Drept răspuns, conceptul de scalabilitate orizontală a permis replicarea datelor pe mai multe mașini/noduri și gestionarea lor prin intermediul unui load balancer.

Deși sistemele distribuite devin din ce în ce mai complexe odată cu adăugarea de noi noduri, tehnicile de recuperare după dezastre, redundanță, partiționare și consistență a datelor au fost dezvoltate în beneficiul acestor tipuri de sisteme.

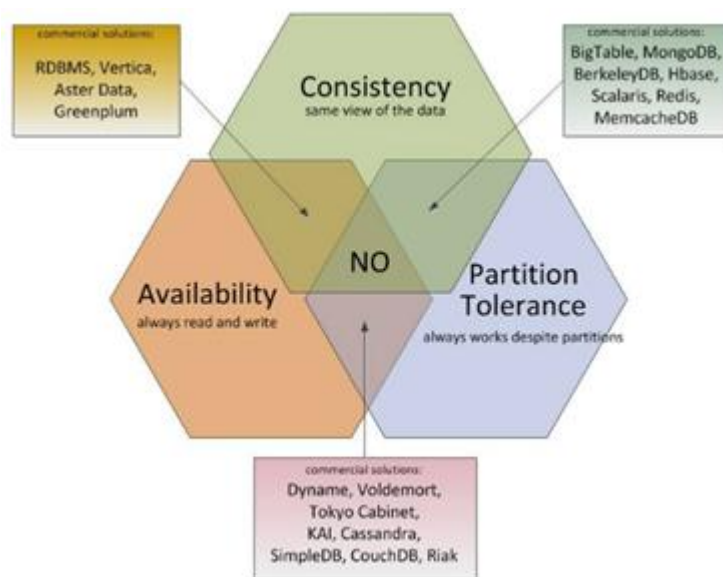
Sistemele de baze de date distribuite trebuie să satisfacă proprietăți precum consistență, disponibilitate și toleranță la partiționare, însă a fost demonstrat că acestea nu

pot fi îndeplinite concomitent. Teoremă CAP (cunoscută ca și teorema lui Brewer) abordează acest aspect. Consistency, Availability și Partition tolerance nu pot fi satisfăcute în același timp de către un sistem distribuit.

Consistența înseamnă că datele sunt aceleași în întregul cluster, astfel încât citirea lor din orice nod va returna același rezultat.

Disponibilitatea constă în abilitatea de a accesa clusterul chiar dacă un nod nu mai este funcțional.

Toleranța la partiționare se rezumă prin faptul că un cluster continuă să funcționeze chiar dacă există o “partiționare” (pauză de comunicare) între două noduri (ambele fiind funcționale, dar neputând comunica).



**Fig. 1 Reprezentarea teoremei CAP**

Sursa: <http://ctrl-d.ro/development/resurse-development/introducere-in-tematica-bazelor-de-date-in-cloud/>

Cele trei puncte nu pot fi îndeplinite concomitent, prin urmare, combinațiile posibile sunt:

- CA - datele sunt consistente între toate nodurile, cu siguranța obținerii acelorași date odată cu citirea din orice nod, dar, la apariția unei partiționări, datele nu vor mai fi sincronizate și nici nu se vor sincroniza după rezolvarea partiționării.
- CP - datele sunt consistente între toate nodurile și se menține toleranță la partiționare (prevenind nesincronizarea datelor) prin indisponibilitatea clusterului atunci când un nod nu mai e funcțional.
- AP - nodurile rămân online chiar dacă nu pot comunica între ele și datele vor fi sincronizate odată ce partiționarea este rezolvată, dar nu există garanția că toate nodurile vor conține aceleași date.

## 1.2 Era Big Data

Volumul de date a început să crească exponențial și analizarea seturilor imense de date - concept cunoscut ca Big Data - va deveni cheia concurenței, stând la baza noilor valuri de creștere a productivității, inovației și excedentului de consum, potrivit cercetărilor realizate de MGI și McKinsey's Business Technology Office.<sup>1</sup> Liderii din fiecare sector vor trebui să se confrunte cu implicațiile Big Data. Volumul tot mai mare și detaliile informațiilor deținute de întreprinderi, creșterea conținutului multimedia, rețelele sociale vor alimenta multiplicarea cantității de date în viitorul apropiat.

Big Data poate fi definit ca orice set de date care depășește câțiva terabytes. Aceasta este limita la care setul de date este considerat destul de mare încât să fie împărțit între mai multe unități de stocare. De asemenea, este mărimea la care tehnicile tradiționalelor RDBMS încep să dea semne de stres.

Doug Laney<sup>2</sup>, pioner în domeniul depozitelor de date și totodată analist cunoscut la Gartner, afirma în 2001 că la baza noțiunii Big Data stau trei "V": volum, viteză, varietate.

- *Volum.* În ceea ce privește prima dimensiune, suntem conștienți că dimensiunea fișierelor a crescut exponențial și că tendința este, în continuare, una de creștere. Mai multe surse de date sunt adăugate în mod continuu. În trecut, toate datele pentru companii erau generate doar intern, de către angajați. Acum, însă, datele provin și de la parteneri sau clienți, sau, în unele cazuri, de la instrumente. Evoluția noilor tehnologii a adus cu sine îmbinarea mai multor surse de date de dimensiuni mari, ceea ce a declanșat creșterea volumului de date care trebuia analizat. Dacă Petabyte a devenit ceva obișnuit, nu mai e mult până când Exabyte îi va lua locul. Un exemplu concret în acest sens este proiectul celor de la Google, LSST (Large Synoptic Survey Telescope)<sup>3</sup>, prin care peste 30 TB de imagini vor fi generate în fiecare noapte, în timpul studiului, timp de 10 ani. Exemplele pot continua cu Youtube, care declară în statistici că 100 de ore de video sunt încărcate în fiecare minut.<sup>4</sup>
- *Viteză.* Inițial, companiile analizau datele folosind procese pe loturi. Cum, în prezent, orice întârziere este un mare minus, s-a recurs la procesarea în timp real. Cu timpul, întârzierile pentru rezultate și analiză vor continua să scadă, pentru a ajunge în timp real. Un exemplu e cazul celor de la Twitter care înregistrează aproximativ 9000 de tweet-uri pe secundă.<sup>5</sup>
- *Varietate.* De la tabele Excel și baze de date, structura datelor a suferit modificări în urma adăugării a sute de alte formate: text, video, foto, web, date GPS, baze de date relaționale, SMS, flash etc. Nu mai există control asupra formatului datelor de intrare. Structura nu mai poate fi impusă, ca în trecut, pentru a menține controlul

<sup>1</sup>Laney Douglas, *3D Data Management: Controlling Data Volume, Velocity and Variety*, 2001 (<http://blogs.gartner.com/doug-laney/files/2012/01/ad949-3D-Data-Management-Controlling-Data-Volume-Velocity-and-Variety.pdf>)

<sup>2</sup><http://www.forbes.com/sites/gartnergroup/2013/03/27/gartners-big-data-definition-consists-of-three-parts-not-to-be-confused-with-three-vs/>

<sup>3</sup> <http://lsst.org/lst/google>

<sup>4</sup> <http://www.youtube.com/yt/press/statistics.html>

<sup>5</sup> <http://www.statisticbrain.com/twitter-statistics/>

asupra analizei. Odată cu apariția unor noi aplicații, sunt introduse noi formate. Un exemplu în acest sens este un proiect la care participă Universitatea Berkeley, Nokia și NAVTEQ, în care telefoanele inteligente sunt folosite pentru determinarea condițiilor din trafic.<sup>6</sup>

Trăim în epoca creșterii galopante a datelor. Camere digitale, bloguri, actualizări zilnice ale rețelelor de socializare, tweet-uri, documente electronice, conținuturi scanate, fișiere audio și video, toate presupun un volum de date în creștere. Consumăm și producem cantități imense de date. Prin urmare, provocările nu întârzie să apară:

- devin dificile stocarea și accesarea eficientă a marilor cantități de date; cererile adiționale de toleranță la erori și backup-uri complică lucrurile mai mult.
- manipularea seturilor mari de date implica rularea unor procese paralele extinse; devin dificile recuperarea datelor în urma unei căderi și furnizarea rezultatelor într-un timp rezonabil.
- o problemă complicată este gestionarea schemei aflate într-o evoluție continuă și a metadatelor pentru date semi-structurate sau nestructurate generate din diferite surse.

Prin urmare, căile și mijloacele de stocare și extragere a cantităților mari de date au nevoie de abordări noi, dincolo de metodele noastre actuale. NoSQL și soluțiile referitoare la Big Data sunt un prim pas înainte, în această direcție.

### 1.3 Trendul NoSQL

Deși am tinde să credem că NoSQL este un termen recent inventat, istoria acestor tipuri de baze de date<sup>7</sup> începe în anii '60, odată cu dezvoltarea de către cei de la TRW(1965) a unor baze de date multi-valoare, Pick Database Management System, folosite de către armata americană pentru inventarierea pieselor pentru elicoptere Cheyenne. Tot în aceeași perioadă (1966-1967), o echipă de la MGH (Massachusetts General Hospital) dezvoltă MUMPS (M)<sup>8</sup>, un limbaj de programare care asigura procesarea tranzacțiilor ACID. Caracteristica unică era baza de date încorporată care permitea acces pentru stocare pe disc, folosind variabile simbolice (mătrici subscrise) similare variabilelor folosite de majoritatea limbajelor pentru a accesa memoria centrală. Bazele de date M sunt de tipul cheie-valoare, optimizate pentru procesarea tranzacțiilor mari. M stochează datele în matrici ierarhice (cunoscute ca noduri chei-valoare, sub-arbori sau memorie asociativă), fiecare având până la 32 de indici/dimensiuni. Un scalar poate fi considerat un element matriceal cu indice 0. Nodurile cu număr variat de indici (inclusiv un nod fără indici) pot co-exista în aceeași matrice. Însă, aspectul cel mai neobișnuit este faptul că baza de date e accesată prin variabile, și nu interogări sau extrageri. Acest lucru înseamnă că accesul la memoria volatilă și non-volatilă se face folosind sintaxa de bază, permițând unei funcții să lucreze cu variabile locale (volatile) sau globale (non-volatile). Practic, acest lucru asigură performanță ridicată privind accesul la date. Proiectat inițial în

---

<sup>6</sup> <http://traffic.berkeley.edu/>

<sup>7</sup> <http://blog.knuthaugen.no/2010/03/a-brief-history-of-nosql.html>

<sup>8</sup> <http://www.cs.uni.edu/~okane/source/MUMPS-MDH/>

1966 pentru industria medicală, M continuă să fie folosit și astăzi de spitale mari sau bănci pentru a asigura procesarea tranzacțiilor mari de date.

În aceeași perioadă (1966), cei de la IBM proiectau IMS (Information Management System)<sup>9</sup> pentru programul Apollo, cu scopul gestionării facturilor imense de materiale pentru racheta Saturn V și vehiculului spațial Apollo. Baza de date IMS folosea un model ierarhic, destul de diferit de DB2-ul lansat mai târziu de IBM. Acest model presupunea folosirea unor blocuri de date, cunoscute ca și segmente. Fiecare segment conținea mai multe bucăți de date - câmpuri. Un segment-rădăcină avea mai multe segmente-frunză, care la rândul lor conțineau alte segmente-frunză, ajungând astfel până la ultimul nivel din ierarhie. Dimensiunea unui segment atingea 40 bytes și trebuia definit un câmp de 6 bytes ca și câmp-cheie, pe baza căruia să se facă ulterior interogări.

Nu peste mult timp, anii '70 au continuat cu apariția ISM (InterSystems M), cu standardizarea limbajului M, cu proiectarea unui motor simplu de baze de date (biblioteca DBM - Database Manager) de către Ken Thompson și lansat de At&T (1979). DBM stoca date prin folosirea unei singure chei (o cheie primară) și a tehnicilor de hashing pentru căutarea datelor după cheie. Câțiva ani mai târziu, istoria a continuat cu TDBM (baze de date de tipul DBM, cu tranzacții atomice inaltuite), NDBM (New DBM), SDBM (o clonă a NDBM, pentru UNIX), GT.M (prima versiune de baze de date cheie-valoare axată pe performanța ridicată privind procesarea tranzacțiilor), Berkeley DB, Lotus Notes. Și produsele dezvoltate de companiile din IT au continuat să apară unul după altul. Și continuă să apară. Însă, trebuie precizat că termenul de NoSQL a fost folosit pentru prima dată acum mulți ani în urmă, în 1998, când Carlo Strozzi a utilizat această denumire pentru baza lui de date open-source care nu avea interfața SQL, datele fiind stocate ca fișiere ASCII și accesate prin shell scripting. Peste 11 ani, la un meeting unde se discutau noile tehnologii din IT, Johan Oskarsson<sup>10</sup> introducea pe piață noi produse precum BigTable și Dynamo. Pentru că era necesară folosirea unui termen scurt pentru descrierea acestei categorii de produse, Eric Evans de la RackSpace a sugerat pentru discuția de atunci, termenul general de "NoSQL". Însă, la scurt timp, noua denumire s-a răspândit la nivel global, ca o publicitate virală, devenind un trend în IT.

Proliferarea bazelor de date non-relaționale în sectorul tehnologic ne face să credem că instrumentele de gestionare a datelor (BD NoSQL) vor face ca bazele de date tradiționale să dispară. Însă nu se dovedește a fi astfel. Fiecare din aceste tipuri de baze de date este potrivit pentru diferite tipuri de sarcini de lucru și acest lucru împiedică dispariția unuia din ele.

Bazele de date NoSQL sunt unice pentru că, de obicei, sunt independente de SQL-ul întâlnit în bazele de date relaționale. Acestea din urmă folosesc SQL ca și limbaj specific domeniului pentru interogări ad-hoc, în timp ce bazele de date non-relaționale nu au un astfel de limbaj standardizat, încât pot folosi orice doresc, inclusiv SQL.

Bazele de date NoSQL sunt concepute să exceleze în viteză și volum. În schimb, soft-urile NoSQL nu promit tot timpul că datele din sistem vor fi consistente. Aici aduc un plus bazele de date relaționale, pentru că atunci când se efectuează o tranzacție financiară,

<sup>9</sup> [http://en.wikipedia.org/wiki/IBM\\_Information\\_Management\\_System](http://en.wikipedia.org/wiki/IBM_Information_Management_System)

<sup>10</sup> <http://leopard.in.ua/2013/11/08/nosql-world/>



cum ar fi o achiziție de pe Amazon, bazele de date trebuie să se asigure că un cont este debitat cu aceeași sumă cu care, în același timp, alt cont este creditat.

Pentru că într-o singură tranzacție sunt necesare multe activități de citire-scriere, un RDBMS nu ar putea ține pasul cu viteza și scalarea necesare funcționării unei companii ca Amazon.

Potrivit lui Bob Wiederhold, CEO al furnizorului de baze de date NoSQL Couchbase<sup>11</sup>, arhitectura și caracteristicile bazelor de date NoSQL aduc 4 avantaje cheie pentru utilizatorii NoSQL:

- *Scalabilitate*

Scalabilitatea facilă este primul aspect subliniat de Wiederhold. Bazele de date precum Couchbase sau MongoDB pot fi extinse pentru a gestiona volume mari de date cu mai multă ușurință.

Dacă o companie este invadată peste noapte de un succes răsunător, de exemplu, clienții ar intra în masă pe site-ul firmei, o bază de date relațională ar trebui să fie replicată și re-partiționată astfel încât extinderea să asigure noua cerere.

Wiederhold oferă ca exemplu al acestei situații furnizorii de rețele sociale și jocuri pe mobil.

Pentru că bazele de date non-relaționale sunt distribuite, pentru extindere nu este nevoie decât de adăugarea unor mașini la cluster pentru a satisface cererea.

- *Performanță*

Performanța este un alt punct la care bazele de date NoSQL excelează. De fiecare dată când se adaugă un nou server la cluster are loc o scalare de performanță.

Dacă o bază de date relațională are zeci sau sute de mii de tabele, procesarea datelor poate genera blocaje și degrada performanța bazei de date.

Pentru că bazele de date NoSQL au modele mai slabe de consistență a datelor, ele renunță la consistență pentru eficiență.

În exemplul cu jocuri sociale, Wiederhold afirma că în momentul în care un utilizator își actualizează profilul, nu se produce o degradare a performanței jocului dacă noile informații din profil nu sunt actualizate instant în întreaga bază de date. Acest lucru înseamnă că resursele pot fi alocate altor lucruri, cum ar fi atacarea de către adversarul de joc.

- *Obiecte*

Când dezvoltatorii de aplicații trebuie să lucreze cu baze de date relaționale, poate fi uneori dificil datorită mapării datelor și problemelor de impedanță.

În bazele de date NoSQL acest lucru nu este o problemă pentru că datele nu sunt stocate în același mod. De exemplu, cu bazele de date orientate-document, datele sunt stocate doar în format document. Și din moment ce documentele sunt obiecte, programatorii care gândesc orientat-obiect vor fi mult mai familiarizați cu manipularea unor astfel de date.

---

<sup>11</sup> The Future of NoSQL Market: An Interview with Bob Wiederhold, CEO of Couchbase (<http://www.marketanalysis.com/?p=329>)

Modelul slab de consistență ajută programatorii deoarece aplicațiile lor nu trebuie să se conformeze strict la cerințele de consistență a datelor. Acest lucru conduce la simplitate și rapiditate în programare.

- *Lipsa downtime-ului*

Al patrulea exemplu al lui Wiederhold este nou în lista de avantaje NoSQL. Este vorba despre un lucru pentru care bazele de date non-relaționale nu au fost special concepute pentru a-l îndeplini, însă s-au dovedit competente.

Pentru că sunt distribuite, bazele de date NoSQL au uptime apropiat de limita maximă de 100%. Acesta este un avantaj imens pentru afacerile bazate pe web și mobil, ce nu își pot permite să fie oprite pentru cel mai scurt moment.

Cu unele planificări avansate, actualizările de soft și upgrade-urile de hard pot fi efectuate în timp ce baza de date este încă în desfășurare. Dacă se încearcă acest lucru cu o bază de date relațională, fără ca ea să fie oprită, problemele nu vor întârzia să apară.

## **1.4 Tipuri de baze de date non-relaționale**

În paralel cu creșterea rapidă a volumului de date, acestea au devenit semi-structurate și împrăștiate. Acest lucru înseamnă că tehnicile tradiționale de gestionare a datelor din zona definirii schemei și referințelor relaționale sunt sub semnul întrebării. Căutarea pentru a rezolva problemele legate de date semi-structurate și voluminoase a dus la apariția unei clase de noi tipuri de baze de date. Această clasă nouă constă în baze de date orientate pe coloane, baze de date chei-valoare, baze de date orientate document, toate cunoscute ca și baze de date NoSQL.

Bazele de date non-relaționale numără în jur de 150 de tipuri, însă câteva dintre acestea au o popularitate ridicată și o utilizare pe scară largă.

Cele mai importante categorii includ: baze de date columnare, baze de date document, baze de date cheie-valoare, baze de date graf.

Nivelurile de performanță, scalabilitate, flexibilitate, complexitate și funcționalitatea bazelor de date non-relaționale și relaționale pot fi analizate în tabelul de mai jos.

Tipul bazei de date	Performanță	Scalabilitate	Flexibilitate	Complexitate	Funcționalitate
Cheie-valoare	ridică	ridică	ridică	moderată	vectori asociativi
Columnară	ridică	ridică	moderată	scăzută	baze de date columnare
Document	ridică	variabilă (ridică)	ridică	scăzută	model obiectual
Graf	variabilă	variabilă	ridică	ridică	teoria grafurilor
Relațională	variabilă	variabilă	scăzută	moderată	algebra relațională

**Fig.2 NoSQL vs. RDBMS**

Sursa: [http://www.academia.edu/5352898/NoSQL\\_Database\\_New\\_Era\\_of\\_Databases\\_for\\_Big\\_Data\\_Analytics\\_-\\_Classification\\_Characteristics\\_and\\_Comparison](http://www.academia.edu/5352898/NoSQL_Database_New_Era_of_Databases_for_Big_Data_Analytics_-_Classification_Characteristics_and_Comparison)

### **Baze de date columnare**

Depozitele columnare nu sunt ceva nou. TAXIR a fost primul sistem de baze de date orientate-coloană, dezvoltat în 1969 pentru biologie. Șapte ani mai târziu sistemul RAPID era folosit în domeniul statisticii, în zona Canadei, continuând a fi utilizat până în anii '90. Pentru mulți ani, Sybase IQ a fost singurul DBMS columnar disponibil comercial. Mai târziu, când OLAP a devenit cunoscut, au fost dezvoltate produse care au preluat tehnici folosite pentru cuburi și operațiuni rollup și le-au aplicat mai multor baze de date generale.

Produsul celor de la Google, Bigtable<sup>12</sup>, adoptă un model în care datele sunt stocate în mod orientat pe coloană. Marele avantaj presupune evitarea consumului de spațiu pentru stocarea valorilor nule, renunțându-se pur și simplu la o coloană atunci când nu există nicio valoare pentru ea. De menționat este faptul că aceste coloane pot stoca orice tip de date atât timp cât datele sunt un șir de biți.

Fiecare unitate de date este gândită ca o pereche de cheie-valoare și unitatea însăși este identificată prin așa numita "cheie primară", întâlnită în Bigtable sub numele de "row-key". Unitățile sunt stocate și ordonate pe baza acestei row-key. În baze de date columnare asemănătoare Bigtable, datele sunt stocate sub forma unor familii de coloane. Aceasta este definită de obicei la configurarea sau pornirea bazei de date. În realitate, familiile de

<sup>12</sup> Shashank Tiwari, *Professional NoSQL*, John Wiley & Sons, Indiana, 2011, p. 11

coloane nu sunt izolate fizic pentru o anumită linie. Column-family acționează ca o cheie pentru coloanele pe care le conține și ca o cheie pentru întregul set de date.

Datele sunt stocate în Bigtable și clonele sale, într-o manieră secvențială adiacentă. Când datele cresc până la capacitatea maximă a unui nod, ele vor fi împărțite între mai multe noduri și sortate, ordonate ca un set secvențial, și nu doar la nivelul fiecărui nod.

Căutarea datelor după row-key este extrem de eficientă. Accesul la date este mai puțin întâmplător și ad-hoc, iar căutarea e simplă din moment ce a fost găsit nodul ce conține datele necesare. Nu de puține ori, modificările presupun adăugări de versiuni noi ale datelor, în loc de înlocuirea datelor vechi cu cele noi. De aceea, mai tot timpul vom găsi câteva versiuni ale fiecărei celule.

Un mare avantaj al acestui tip de baze de date este inexistența unui blocaj în momentul când doi sau mai mulți utilizatori încearcă accesarea diferitelor subseturi de coloane. Acest lucru este facilitat de RAID (Redundant Array of Independent Disks), care combină mai multe discuri într-o unitate logică. Datele sunt stocate în mai multe șabloane (nivele), cu diferite grade de redundanță.

Dezavantajele ar putea consta în lipsa tranzacțiilor și unele limitări pentru dezvoltatorii obișnuiți să folosească RDBMS.

Acest tip de baze de date este folosit, în general, pentru sisteme analitice, BI, depozite de date analitice. Câteva exemple sunt: HBase, Cassandra, Amazon SimpleDB, Hypertable.

Multe sarcini de lucru sunt columnar selective și, prin urmare, beneficiază extrem de mult de acest model. Bazele de date columnare gestionează foarte bine cantități mari de date și interogări I/O, aduc beneficii în materie de performanță și au capacități unice de comprimare a datelor. Însă două motive au determinat saltul lor în piața IT: hardware-ul îmbunătățit (SSD) și algoritmi eficienți (algoritmi paraleli).

### ***Baze de date document***

MongoDB face parte din categoria bazelor de date orientate-document, unde documentele sunt grupate în colecții. Colecțiile pot fi asemănate cu tabelele bazelor de date relaționale. Colecțiile nu impun un format strict, așa cum este cazul tabelelor relaționale. Într-o singură colecție pot fi grupate documente arbitrare. Totuși, pentru o indexare eficientă, documentele din colecție ar trebui să fie cât de cât similare. Fiecare document este stocat în format BSON. BSON este o reprezentare binară a formatului JSON, unde structura este apropiată de setul înălțuit de perechi chei-valoare și suportă tipuri adiționale precum expresii regulate, date binare și date calendaristice. Fiecare document are un identificator unic generat automat de MongoDB, în cazul în care nu este specificat explicit la inserarea datelor în colecție.

Deși nu există o formulă pentru determinarea numărului optim de colecții într-o bază de date, e recomandabil ca datele diferite să nu fie amestecate într-o singură colecție întrucât crește complexitatea pentru indecși. O colecție poate atinge limita de mărime de 2 GB. Din acest motiv e utilă folosirea colecțiilor limitate (capped collections). Acestea sunt

ca o stivă cu o mărime predefinită, iar în momentul când se atinge limita, datele vechi sunt șterse. Câmpul `_id` indexează fiecare colecție MongoDB. Când sunt interogate, documentele dintr-o colecție sunt returnate în ordinea naturală a `_id`-ului. Doar colecțiile limitate folosesc ordinea bazată pe metoda LIFO, care reprezintă și ordinea de inserare.

MongoDB oferă performanță sporită, dar face acest lucru în detrimentul fiabilității. Acest tip de baze de date nu respectă întotdeauna atomicitatea și nu stabilește niveluri de integritate sau izolare tranzacțională în timpul operațiilor simultane. Deci este posibil ca procesele să se suprapună în timpul actualizării colecției. Doar o clasă de operații (operații modificatoare) oferă consistență atomică. Includem aici : `$inc`, `$set`, `$unset`, `$push`, `$pushAll`, `$addToSet`, `$pop`, `$pull`, `$pullAll`, `$rename`.

În această categorie sunt incluse și: Couchbase, CouchDB, RethinkDB.

### ***Baze de date cheie-valoare***

Acest tip de baze de date permit stocarea sub formă de perechi cheie-valoare și citirea valorilor prin intermediul unei valori unice (cheia perechii). Cheia poate fi sintetică sau autogenerată, iar valoarea poate fi de orice tip: String, JSON, BLOB etc. Un concept nou este cel de ‘bucket’ - o grupare logică de chei-valori. Bucketuri diferite pot conține chei identice, însă cu valori diferite.

Spre deosebire de bazele de date relaționale, cele de tipul cheie-valoare pot gestiona cu ușurință volume mari de înregistrări și un număr mare de schimbări pe secundă, cu milioane de utilizatori simultan, prin prelucrări și stocări distribuite.

Folosite frecvent pentru gestionarea informațiilor din sesiunile aplicațiilor web, utilizate în aplicații pentru jocuri online, comerț electronic, aceste baze de date sunt eficiente și pentru analiza în timp real a tranzacțiilor din domeniul financiar.

Riak, un produs open-source al celor de la Basho Technologies, este proiectat pe clustere de noduri fizice sau virtuale, iar datele și operațiile sunt distribuite pe aceste clustere. Cu cât numărul de noduri este mai mare, cu atât funcționalitatea este mai bună și timpul de răspuns scurt. Comunicarea în cluster se face prin intermediul unui protocol special, Gossip, care stochează informații despre statusul clusterului și distribuie informații despre bucket-uri. Soluția asigură performanță, costuri scăzute și scalabilitate.

În această categorie amintim și Redis, Memcached, Hamster DB, Amazon Dynamo DB.

### ***Baze de date graf***

Pornind de la noțiunea de graf- o colecție de noduri și relațiile dintre ele - și de la teoria grafurilor, bazele de date de tip graf au reușit să se consacre în companii cunoscute precum Google, Facebook, Twitter, Glassdoor etc, în special datorită faptului că oferă

stocare și interogare optimizată pentru date ce pot fi reprezentate sub forma unui graf (exemplu: o rețea socială).

Tratând relațiile ca obiecte de primă clasă, bazele de date graf au devenit o inovație într-o continuă extindere, popularitatea lor crescând cu 300% în ultimul an (conform unui raport realizat de DB-Engines)<sup>13</sup>. În baza de date nu sunt stocate doar informații despre obiecte individuale, ci și relațiile dintre ele. Această capacitate face mai ușoară exprimarea unor întrebări sofisticate și obținerea unor răspunsuri, decât dacă am interoga o bază de date relațională pentru a obține aceleași rezultate.

Bazele de date graf sunt în general folosite cu sisteme OLTP. Punctele tari pe care le oferă sunt: performanță, flexibilitate, scalabilitate. În general, sunt utilizate în aplicații geospațiale, web analitic, BI, bioinformatica, rețele sociale.

## 1.5 MapReduce: simplificarea procesării datelor din clustere mari

Apariția conceptului de Big Data și nevoia de procesare paralelă pe scară largă pentru manipularea datelor au condus la răspândirea adoptării infrastructurilor bazate pe scalabilitate orizontală. O parte din ele (cum sunt cazurile Google, Amazon, Facebook, eBay, Yahoo!) implică un număr foarte mare de servere, ajungându-se la mii sau sute de mii de servere. Procesarea datelor dintr-un cluster poate fi complexă.

Google publica în 2004 o lucrare despre MapReduce, un model de programare care permite procesarea rapidă a cantităților masive de date<sup>14</sup>. În 2005 începea proiectul Apache Hadoop, care, în prezent este cea mai cunoscută implementare open-source a framework-ului MapReduce. Creșterea popularității modelului MapReduce a dus la o “revoluție” în procesarea Big Data. Au fost dezvoltate instrumente și limbaje auxiliare pentru facilitarea analizelor Big Data. Modelul MapReduce oferă una din cele mai bune metode de procesare a datelor din clustere scalabile orizontal.

MapReduce este un model de programare paralelă care permite procesarea distribuită pe seturi mari de date dintr-un cluster de mașini. MapReduce a preluat ideile din conceptele prezente în lumea programării funcționale. Map și Reduce sunt funcții frecvent întâlnite în programarea funcțională: prin funcția Map se aplică o operație fiecărui element din listă (ex: multiplicarea cu 2 a unor numere), obținându-se tot o listă, iar prin funcția Reduce se aplică o operație asupra unei liste și se obține un singur rezultat (ex.: suma unor numere). Această idee simplă a fost extinsă asupra seturilor mari de date din colecții de tupluri sau perechi chei-valoare. Astfel, funcția Map se aplică pentru fiecare pereche cheie-valoare și generează o nouă colecție. Funcția Reduce acționează asupra noii colecții prin aplicarea unei funcții agregate, pentru a calcula un rezultat final.<sup>15</sup>

---

<sup>13</sup> <http://neo4j.com/stories/telenor/>

<sup>14</sup> J. Dean, S. Ghemawat, *MapReduce: Simplified Data Processing on Large Clusters*, Sixth Symposium on Operating System Design and Implementation, San Francisco, 2004 (<http://research.google.com/archive/mapreduce.html>)

<sup>15</sup> Shashank Tiwari, *Professional NoSQL*, John Wiley & Sons, Indiana, 2011, p. 10

De exemplu, avem o colecție de perechi chei-valoare, unde cheia este codul poștal al unei persoane și valoarea este prenumele persoanei:

```
[{"700701" : "Ana"}, {"700701" : "Ioana"}, {"700705" : "George"}, {"700707" : "Marius"}]
```

O funcție *Map* ne-ar putea genera o colecție cu persoanele care locuiesc la un anumit cod poștal:

```
[{"700701" : ["Ana", "Ioana"]}, {"700705" : ["George"]}, {"700707" : ["Marius"]}]
```

O funcție *Reduce* aplicată noii colecții ar putea restitui numărul de persoane ce locuiesc la un anumit cod poștal:

```
[{"700701" : 2}, {"700705" : 1}, {"700707" : 1}]
```

*MapReduce* este fundamentul prelucrării datelor în Hadoop-ului, este paradigma de programare care permite scalabilitate sporită pentru sute sau mii de servere dintr-un cluster. Termenul *MapReduce* se referă de fapt la două sarcini separate, distincte, desfășurate de programele *Hadoop*. *Map* preia un set de date și îl convertește într-un alt set de date ce are elemente de forma tuplurilor (perechi cheie-valoare). *Reduce* preia rezultatul obținut anterior și combină datele într-un set mai mic de tupluri. Acest proces va avea loc întotdeauna după activitatea *Map*.

Cu toate că Hadoop este cel mai cunoscut în momentul de față, nu au întârziat să apară și produse colaterale. Intră în această categorie și *Pig*, *Hive*, *Dremel*, *Drill* și altele asemenea. Mai jos, puținele noțiuni de bază oferă o idee de ansamblu și un punct comun al acestor produse.

**Apache Pig** este o platformă pentru analiză seturilor mari de date, ce constă într-un limbaj pentru programele de analiză a datelor și o infrastructură pentru evaluarea acestor programe. O proprietate remarcabilă a programelor *Pig* este capacitatea de paralelizare a structurii de bază. Infrastructura *Pig* constă într-un compilator care produce secvențe de programe *MapReduce*, pentru care există deja implementări paralele pe scară largă (de exemplu, subproiectul Hadoop). Limbajul *Pig Latin* are următoarele proprietăți-cheie:

- Ușurință în programare. Execuția paralelă a sarcinilor simple de analiză a datelor este o activitate banală. Sarcinile complexe, compuse din mai multe transformări de date interconectate, sunt codificate explicit ca secvențe de fluxuri de date, făcând ușoară scrierea, înțelegerea și păstrarea lor.
- Oportunități de optimizare. Modul de codificare a sarcinilor permite sistemului să optimizeze în mod automat executarea lor, permițând utilizatorului să se concentreze mai mult pe semantică, decât pe eficiență.
- Extensibilitate. Utilizatorii pot crea propriile funcții pentru procesări particularizate.

### **Pig vs SQL**

În comparație cu SQL, *Pig* folosește evaluarea întârziată (*lazy evaluation*), ETL, e capabil să stocheze date în orice punct în timpul unui pipeline (lanț de procese, fire de

execuție, rutine), declară planuri de execuție, suportă împărțiri de *pipeline*.<sup>16</sup> Evaluarea întârziată (lazy evaluation) este o strategie care presupune faptul că expresiile sunt evaluate abia atunci când rezultatele lor sunt necesare în alte calcule. Deci, argumentele nu sunt evaluate înainte de a fi păsate unei funcții, ci doar când se vor folosi valorile lor.

Pe de altă parte, s-a susținut că SGBD-urile sunt mult mai rapide decât sistemele *MapReduce* odată ce datele sunt încărcate, dar această încărcare durează în mod semnificativ mai mult în sistemele de baze de date. S-a susținut de asemenea că SGBD-urile relaționale oferă suport pentru stocare columnară, lucrând cu date comprimate, întrucât pentru accesul eficient la date aleatoare și toleranță la eroare - la nivel de tranzacție.<sup>17</sup>

Omniprezența limbajului SQL este convenabilă. Totuși *Pig Latin* este o alegere mult mai naturală în a construi pipeline-uri de date pentru că:

- este un limbaj procedural, pe când SQL este declarativ;
- permite programatorilor să decidă în ce punct din *pipeline* se va face controlul datelor;
- permite programatorului să aleagă anumite implementări, în locul optimizatorului;
- suportă împărțirile (*splits*) în cadrul *pipeline*-ului;
- permite programatorilor să însereze propriul cod în aproape orice loc din *pipeline*-ul de date.

Prin definiție, un limbaj declarativ permite programatorului să specifice ce trebuie făcut, și nu cum trebuie făcut. În SQL, utilizatorii pot specifica faptul că trebuie aplicată o clauza JOIN pentru a lega datele din două tabele, însă, fără a alege și algoritmul potrivit pentru *join*. În cazul *Pig*, programatorii se bazează pe faptul că optimizatorul va face alegerea corectă pentru ei. Programarea în *Pig Latin* este asemenea specificării unui plan de execuție a unei interogări, facilitând controlul fluxului de sarcini privind procesarea datelor.<sup>18</sup>

**Apache Hive** este un sistem data warehouse pentru interogarea și analiza seturilor mari de date stocate în fișiere *Hadoop*. Are 3 funcții principale: centralizarea, interogarea și analiza datelor. Permite exprimarea interogărilor într-un limbaj numit *HiveQL*, care traduce, în mod automat, interogări SQL în job-uri *MapReduce* executate în *Hadoop*. În plus, *Hive* este mult mai potrivit pentru aplicațiile de tip depozite de date, unde sunt analizate date relativ statice și unde nu este obligatoriu un timp foarte scurt de răspuns. Spre deosebire de *Pig*, care permite încărcarea datelor în orice punct din pipeline, *Hive* solicită mai întâi importul datelor și apoi prelucrarea efectivă a lor.

---

<sup>16</sup> Yahoo Pig Development Team: Comparing Pig Latin and SQL for Constructing Data Processing Pipelines (<https://developer.yahoo.com/blogs/hadoop/comparing-pig-latin-sql-constructing-data-processing-pipelines-444.html>)

<sup>17</sup> M. Stonebraker, D. Abadi, D.J. Dewitt, S. Madden, E. Paulson, A. Pavlo, A. Rasin, *Communications of the ACM: MapReduce and Parallel DBMSs: Friends or Foes?*, vol. 53, nr. 1, New York, 2010, p. 64-71 (<http://database.cs.brown.edu/papers/stonebraker-cacm2010.pdf>)

<sup>18</sup> C. Olston, B. Reed, U. Srivastava, R. Kumar, A. Tomkins, *ACM SigMod 08: Pig Latin: A Not-So-Foreign Language for Data Processing*, Vancouver, 2008 (<http://infolab.stanford.edu/~olston/publications/sigmod08.pdf>)



Tabelele din *Hive* sunt similare cu cele din bazele de date relaționale, fiind formate din partiții. Datele pot fi accesate printr-un limbaj simplu de interogare, *HiveQL*, similar cu SQL. *Hive* permite suprascrierea și adăugarea datelor, dar nu și modificarea sau ștergerea lor. Datele din tabele sunt serializate și fiecare tabelă are un registru *HDFS* (*Hadoop Distributed File System*). Tabelele sunt divizate în partiții și ulterior în *bucket-uri*. *Hive* permite folosirea tipurilor de date primitive precum *Timestamp*, *String*, *Float*, *Boolean*, *Decimal*, *Binary*, *Double*, *Int*, *Tinyint*, *Smallint* și *Bigint*. Combinația lor poate genera tipuri de date complexe precum structuri, hărți și matrici.

Avantajele *Hive* pot fi enumerate astfel:

- este familiar. Sute de utilizatori pot interoga simultan baza de date, folosind un limbaj familiar utilizatorilor SQL.
- este rapid. Comparativ cu interogările rulate pe aceleași seturi imense de date, în *Hive* timpul de răspuns este mai scurt.
- este scalabil. Când volumul de date crește, se pot adăuga mai multe mașini în cluster, fără scăderea performanței.

Deși scopul este același, *Hive* și *Pig* prezintă diferențe concrete la nivel de flux de date, de sintaxă, de mentenanță, persistență și alte aspecte. Dacă *Pig* este un limbaj procedural, *Hive* se aseamănă foarte mult cu SQL, fiind declarativ. De aici derivă și ușurința în învățare de care creatorii *Hive* sunt mândri, respectiv timpul mai îndelungat necesar învățării sintaxei *Pig*. *Hive* este folosit mai mult în aria de analiză a datelor, pe când *Pig* este recomandat programatorilor, mai ales în cazul interogărilor complexe, cu multe *join-uri* și filtre. Dacă *Pig* gestionează în mod eficient atât date structurate cât și nestructurate, *Hive*, ca și SQL, este mai mult destinat datelor structurate. La nivel de rezultate intermediare, *Hive* folosește tabele, iar *Pig* variabile. Dacă *Pig* permite depanarea locală, *Hive* reușește acest lucru cu greu și cu mult consum de timp. Mai mult, funcțiile definite de utilizator sunt prea complexe în *Hive*, spre deosebire de *Pig*. În schimb, *Hive* câștigă la capitolul mentenanță, unde procesul se realizează foarte ușor comparativ cu *Pig*. Pe lista clienților *Pig* se numără *Yahoo!*, *Twitter* și *LinkedIn*, în timp ce *Hive* este ales de gigantul *Facebook*.

**Google Dremel** este un sistem scalabil, de interogări ad-hoc, destinat *Big Data*, care depășește capacitățile *Hadoop*. Prin combinarea arborilor de execuție multi-nivel și aspectului datelor columnare, poate agrega în câteva secunde tabele de miliarde de linii.<sup>19</sup>

Dremel este o bază de date interactivă, cu un limbaj pentru date structurate, asemănător clasicului SQL. În locul tabelelor cu câmpuri fixe întâlnite în bazele de date relaționale, produsul prezintă fiecare linie ca un obiect JSON, iar valorile pot fi restricționate datorită unui format de protocol dezvoltat de *Google*. Pe plan intern, datele sunt stocate într-un format special care le curăță mereu într-un mod foarte eficient.

---

<sup>19</sup> S. Melnik, A. Gubarev, J.J. Long, G. Romer, S. Shivakumar, M. Tolton, T. Vassilakis, *Dremel: Interactive Analysis of Web-Scale Datasets*, 2010  
(<http://static.googleusercontent.com/media/research.google.com/ro/pubs/archive/36632.pdf>)

Interogările sunt trimise la server și agregate la returnare, folosind un format inteligent pentru performanță maximă.

Exemplele pot continua, însă ideea principală rămâne aceeași: volumul de date a crescut, structura lor s-a schimbat și cum nu există punct de oprire pentru aceste tendințe, tehnologiile și produsele care apar, au de fapt același scop și anume cel de a facilita procesarea și analizarea datelor.

## Capitolul 2 - Modelul relațional în Oracle vs. modelul graf în Neo4j

### 2.1 Oracle vs. Neo4j : model de date, tipuri de date, tipuri de interogări

#### Modelul de date în Oracle

Modelul relațional este baza conceptuală a bazelor de date relaționale. Propus de E.Codd în 1969, el reprezintă o metodă de structurare a datelor folosind relații, asemenea structurilor matematice de tip grilă, cu linii și coloane. Datele sunt stocate în tabele (relații) ce au drept antet lista de coloane și drept conținut setul de date organizat pe linii. La intersecția dintre o coloană și o linie întâlnim o valoare unică, denumită tuplu<sup>20</sup>.

O a doua caracteristică majoră a modelului relațional o reprezintă utilizarea cheilor. Acestea sunt definite ca și coloane dintr-o relație, folosite pentru ordonarea datelor sau pentru a lega date din tabele diferite. De departe, cea mai importantă este cheia primară, care identifică în mod unic fiecare linie, urmată de cheia străină care leagă datele dintr-o tabelă-copil cu datele dintr-o tabelă-părinte.

Întâlnite sub denumirea restricțiilor de integritate, seturile de reguli stabilite la nivel inter-relațional și intra-relațional asigură integritatea datelor. Constrângerile de domeniu (constrângerile de coloană) impun restricții la nivelul atributelor, astfel încât acestea să corespundă sensului pe care îl au în realitatea modelată. Cele mai cunoscute exemple includ: NOT NULL, CHECK, DEFAULT. Constrângerile de tuplu, întâlnite sub forma cheii primare și a celei secundare, asigură unicitatea tuplurilor. Regulile impuse la nivel inter-relațional aduc în discuție termenul de integritate referențială, asigurată prin definirea cheilor străine.

Ca parte integrantă a modelului relațional, normalizarea a fost propusă de Codd pentru obținerea unor forme normalizate, prin eliminarea valorilor non-atomice și datelor redundante, asigurând prevenirea anomaliilor de manipulare a datelor și pierderea integrității.<sup>21</sup>

O bază de date relațională poate stoca date de diferite tipuri: de la date numerice (întregi, reale, în virgulă fixă sau mobilă), la șiruri de caractere (char, varchar2), date de tip LOB (BFILE, BLOB, CLOB, NLOB), date de tip boolean (true, false, null), de tip data calendaristică (date, time, interval), de tip ROWID.

Pentru manipularea datelor a fost necesară adoptarea unui limbaj de interogare, universal valabil. Abreviat de la Structured Query Language, SQL este limbajul standardizat de interogare a datelor dintr-o bază de date relațională. Versiunea originală, numită SEQUEL (Structured English Query Language) a fost proiectată de un centru de cercetare IBM, în 1974 și 1975. Patru ani mai târziu a fost introdus de Oracle ca și sistem de baze de date comerciale. Istoric vorbind, SQL a fost limbajul de interogare preferat al

---

<sup>20</sup> E.F.Codd, *The relational model for database management: version 2*, Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1990, p. 82

<sup>21</sup> C.J.Date, Hugh Darwen, *Databases, Types And the Relational Model*, Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 2006, p. 253

SGBD-urilor ce rulau pe minicalculatoare și mainframe-uri. El s-a impus ca fiind un limbaj ușor de învățat, standardizat, declarativ, interactiv, case insensitive.

SQL este probabil cel mai simplu, dar cel mai puternic limbaj specific domeniului, creat până acum.

Este ușor de învățat pentru că are un vocabular limitat, o gramatică lipsită de ambiguitate și sintaxă simplă. Este concis și limitat în domeniul de aplicare, însă face exact ceea ce este menit să facă: permite manipularea seturilor de date structurate, ce pot fi ușor filtrate, sortate, împărțite. Bazat pe relații, se pot face intersecții și reuniuni; se poate rezuma un set de date și grupa după un anumit atribut, sau filtra pe baza unui anumit criteriu de grupare.

Toate frazele SQL folosesc un optimizator/instrument de optimizare care determină cele mai eficiente moduri de accesare a datelor menționate. Sarcinile SQL includ: interogare de date, inserare, modificare, ștergere de linii dintr-o tabelă, creare, înlocuire, modificare, ștergere de obiecte, controlul accesului la bază de date și obiectele sale, garantarea consistenței și integrității bazei de date.

Toate SGBD-urile mari suportă SQL și toate programele scrise în SQL pot fi transferate dintr-o bază de date în alta, cu mici modificări. În plus, față de datele tradiționale, SQL poate stoca, extrage și prelucra date mai complexe: obiecte, colecții, REF-uri, LOB-uri, XML-uri și suportul nativ al capacităților bazate pe standarde include caracteristici precum: suport pentru expresii regulate native (căutări pe bază de model), tipuri de date în virgulă mobilă (reduc spațiul necesar pentru datele numerice), funcții agregate și analitice (manipularea datelor din depozite de date și data marts).<sup>22</sup>

Pentru obținerea informațiilor din baza de date, apelăm la interogări construite pe baza unor cuvinte-cheie, domenii, criterii de selecție. Fie că sunt simple sau complexe, instrucțiunile de selecție debutează întodeauna cu magicul cuvânt-cheie “SELECT”.

Sintaxa unei interogări simple s-ar putea rezuma prin:

```
SELECT nume_campuri  
FROM nume_tabela  
[WHERE criteriu_selectie]  
[ORDER BY criteriu_ordonare [ASC/DESC]];
```

Odată cu introducerea funcțiilor agregat (COUNT, SUM, AVG, MIN, MAX), asocierilor (JOIN) sau combinărilor (UNION), interogarea se va transforma într-una complexă.

În cazul folosirii unei funcții agregat, sintaxa ar urma exemplul:

```
SELECT functie_agregat(nume_camp) AS alias  
FROM nume_tabela  
GROUP BY camp_grupare  
[HAVING criteriu_grupare]  
[ORDER BY criteriu_ordonare [ASC/DESC]];
```

---

<sup>22</sup> Alan Beaulieu, *Learning SQL*, O'Reilly Meduia, Inc, CA, USA, 2009, p.18

Selectarea datelor din mai multe tabele se poate realiza datorită clauzei JOIN, care asigură legătura dintre tabele pe baza unui câmp comun:

```
SELECT nume_campuri
FROM nume_tabela
{INNER/LEFT OUTER/RIGHT OUTER} JOIN nume_tabela
ON criteriu_asociere
[WHERE criteriu_selectie]
[ORDER BY criteriu_ordonare [ASC/DESC]];
```

Există uneori necesitatea unei afișări cumulate a rezultatelor obținute din două sau mai multe interogări. Pentru realizarea acestui lucru folosim clauza UNION:

```
SELECT nume_campuri
FROM nume_tabela
UNION
SELECT nume_campuri
FROM nume_tabela2
[GROUP BY câmp_de_grupare]
[HAVING criteriu_de_agregare]
[ORDER BY câmpuri_criteriu [ASC|DESC]];
```

## Modelul de date în Neo4j

Pornind de la Teoria grafurilor, Neo4j a construit o structură ce are la bază noduri, relații și proprietăți. Comparând modelul bazelor de date graf cu cel al bazelor de date relaționale, am putea generaliza că un nod e similar unei înregistrări dintr-o tabelă, iar o relație preia rolul cunoscutei clauze “JOIN”. În Neo4j, modelul de date este bazat pe elemente granulare conectate prin relații care contribuie la accelerarea anumitor operații.

Cum bazele de date graf permit stocarea datelor nestructurate, nu vom întâlni nicidecum constrângeri precum CHECK, ci totul se limitează doar la UNIQUE. Relațiile dintre noduri au dus la dispariția clasicele chei primare și străine extrem de necesare în bazele de date relaționale.

Ca limbaj de interogare a datelor este folosit Cypher, care are sintaxa cât de cât asemănătoare SQL-ului și clauzele folosite sunt intuitive.

Proprietățile pot avea diferite tipuri de date pe diferite noduri. Marele avantaj este că Cypher recunoaște singur tipul de dată asociat de noi unui nod sau unei relații, pe când SQL necesită specificarea tipurilor de date în mod explicit (ex: VARCHAR).

Pentru o mai bună asimilare a noțiunilor, exemplul practic de mai jos oferă o analiză comparativă a interogărilor efectuate asupra unei baze de date relaționale, folosind SQL și interogărilor din Neo4j, folosind Cypher.

Am ales colectarea informațiilor despre structura ierarhică a unei companii, pornind de la sediul central, filială, departament și ajungând până la cea mai mică parte:

angajatul. Cum bine se știe, o companie este condusă de un CEO și poate avea mai multe filiale, acestea având la rândul lor mai multe departamente cu angajați care răspund de gestionarea anumitor proiecte.

În Oracle, totul debutează prin crearea tabelelor și popularea lor cu date.

În Neo4j, am început prin crearea nodurilor și a relațiilor dintre ele.

## 2.2 Operații CRUD în Oracle și Neo4j

### 1. CREATE și INSERT vs. CREATE

Clauza CREATE este folosită pentru crearea tabelelor, iar INSERT contribuie la popularea acestora cu date.

În Oracle, începem prin crearea tabelelor și continuăm cu popularea lor. În Neo4j, în general, ambele operațiuni au loc simultan, odată cu crearea nodului. Dacă în Oracle definim câmpuri ale tabelii, în Neo4j vom avea, în mod echivalent, proprietăți ale nodului. Principala deosebire este faptul că în Neo4j, printr-o singură secvență de cod, vom putea crea nodul, defini proprietățile și valorile pentru el, pe când în Oracle vom putea crea doar structura, urmând ca mai apoi să introducem valori prin bine-cunoscutul INSERT.

```
CREATE TABLE companii (  
idCompanie NUMBER(3) NOT NULL PRIMARY KEY,  
denumire VARCHAR2(50) NOT NULL,  
domeniu VARCHAR2(50),  
tara VARCHAR2(30),  
oras VARCHAR2(25),  
adresa VARCHAR2(200),  
anInfiintare NUMBER(4)  
) ;
```

```
INSERT INTO companii (idCompanie, denumire, domeniu, tara, oras, adresa,  
anInfiintare)  
VALUES (1, 'Amazon', 'Comert electronic', 'Washington', 'Seattle', '1200  
12th Ave. South, Ste. 1200', 1994) ;
```

În Neo4j nu avem noțiunea de tabelă. În schimb, nodurile și relațiile sunt cele ce compun baza noastră de date. Odată cu crearea unui nod, acesta primește niște proprietăți și valori pentru ele.

Spre exemplu, vom defini un nod de tip Companie și un nod de tip Angajat. Fiecare din cele două va avea proprietăți (denumire, domeniu, țară etc.; nume, data\_nașterii, gen etc) și valori ('Amazon', 'Comert electronic', 'Washington'; 'Bezos Jeffrey', '1964-01-12', 'M').

Sintaxa necesară pentru crearea nodurilor arată astfel:

```
CREATE (amazon:Companie {denumire: 'Amazon', domeniu: 'Comert
electronic', tara: 'Washington', oras: 'Seattle', adresa: '1200 12th
Ave. South, Ste. 1200', anInfiintare: 1994})
CREATE (jbezoz:Angajat {nume: 'Bezos Jeffrey', data_nasterii: '1964-01-
12', gen: 'M', nationalitate: 'americana', data_angajarii: '1994-07-10',
post: 'CEO'})
```

Cum orice companie este condusă de un angajat (CEO), vom defini relația ‘CONDUCE’ între nodurile mai sus create:

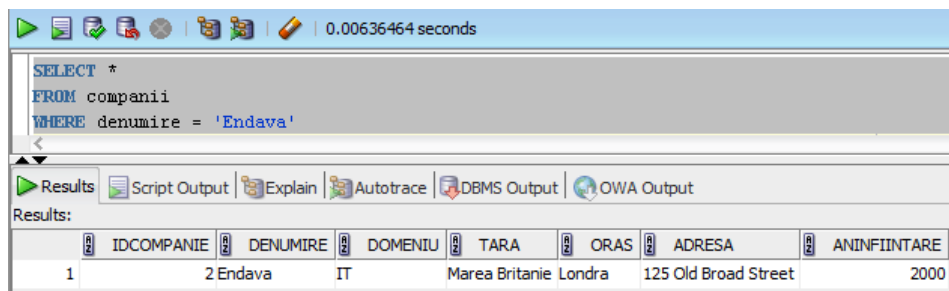
```
MATCH (a:Angajat),(b:Companie)
WHERE a.nume = 'Bezos Jeffrey' AND b.denumire = 'Amazon'
CREATE (a)-[r:CONDUCE]->(b)
```

## 2. a) SELECT vs. RETURN

Pentru obținerea informațiilor relevante dintr-o bază de date, vom apela întotdeauna la interogări cu ajutorul cărora avem posibilitatea de a extrage exact lucrurile care ne sunt de interes. În acest sens, Oracle folosește pentru citirea datelor clauza SELECT, pe când Neo4j are definită clauza RETURN.

Spre exemplu, în Oracle, obținerea datelor despre compania Endava o putem realiza prin:

```
SELECT *
FROM companii
WHERE denumire = 'Endava'
```



IDCOMPANIE	DENUMIRE	DOMENIU	TARA	ORAS	ADRESA	ANINFIINTARE
1	2 Endava	IT	Marea Britanie	Londra	125 Old Broad Street	2000

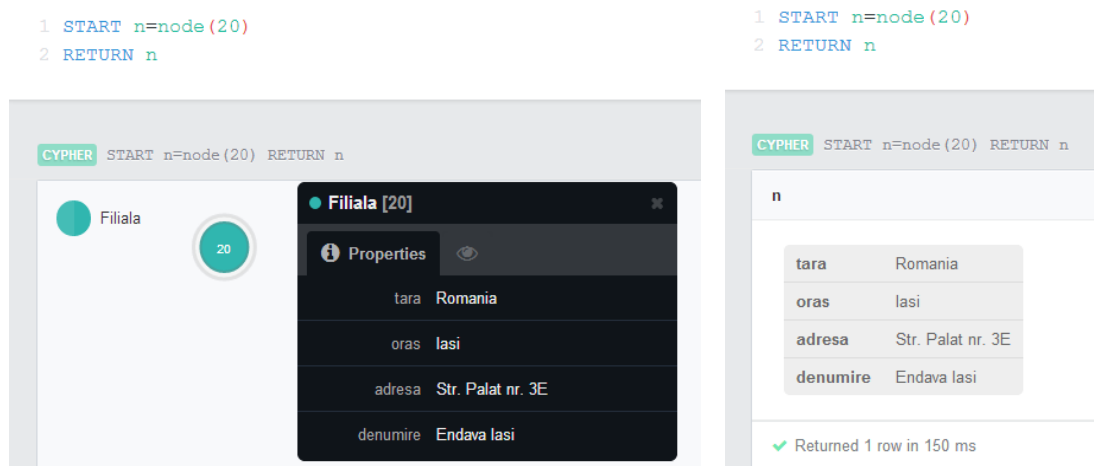
**Fig. 3 Clauza SELECT în SQL (Oracle)**

În Neo4j, una din modalitățile de afișare a cerinței de mai sus, poate arăta astfel:

```
START n=node(20)
RETURN n
```

Dacă știm că nodul aferent companiei Endava are indexul 20, putem construi interogarea apelând la clauza START, care definește punctul de start în care începe

căutarea, în cazul nostru `node(20)`, adică nodul aferent companiei Endava. Atributele clauzei `RETURN` sunt cele care se doresc a fi afișate, în cazul nostru fiind toate atributele nodului `n` (cel cu index 20).



**Fig. 4 Clauza RETURN în Neo4j**

## 2. b) `SELECT (...) FROM (...) WHERE (...)` vs. `MATCH (...) WHERE (...)` `RETURN (...)`

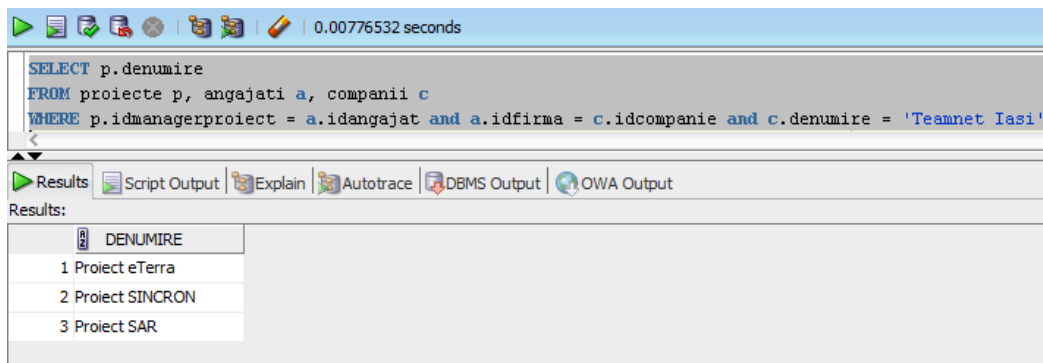
În Oracle, clauzele `SELECT`, `FROM` și `WHERE` sunt poate printre cele mai utilizate pentru extragerea datelor. Atributele ce precedă `SELECT`-ul sunt de fapt scopul întregii interogări (câmpurile ce se doresc a fi afișate), ceea ce urmează după `FROM` sunt de fapt tabelele din care se extrag datele și clauza `WHERE` ajută la filtrare în funcție de anumite condiții.

În mod similar, sintaxa Cypher are la bază clauzele `MATCH`, `WHERE` și `RETURN`. `MATCH` îl putem asemăna oarecum cu `FROM`, pentru că permite specificarea nodurilor și relațiilor participante la interogare, `WHERE` are același rol de stabilire a filtrelor, iar `RETURN`, după cum precizam mai sus, stabilește nodurile sau proprietățile ce se doresc a fi afișate.

Spre exemplu, dacă vom dori afișarea proiectelor derulate în cadrul filialei Teamnet din Iași, sintaxele și rezultatele vor arăta astfel:

```
SELECT p.denumire
FROM proiecte p, angajati a, companii c
WHERE p.idmanagerproiect = a.idangajat and a.idfirma = c.idcompanie and
c.denumire = 'Teamnet Iasi'
```

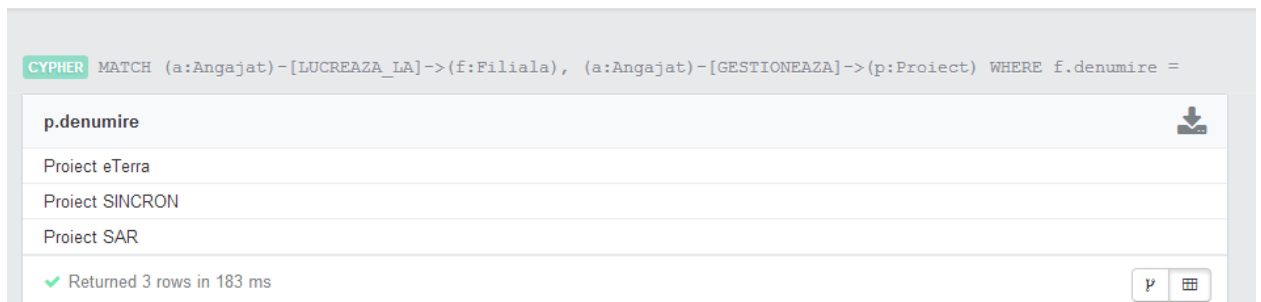




*Fig. 5 Clauzele SELECT, FROM, WHERE în SQL (Oracle)*

```
MATCH (a:Angajat)-[LUCREAZA_LA]->(f:Filiala), (a:Angajat)-[GESTIONEAZA]->(p:Proiect)
WHERE f.denumire = 'Teamnet Iasi'
RETURN p.denumire
```

```
1 MATCH (a:Angajat)-[LUCREAZA_LA]->(f:Filiala), (a:Angajat)-[GESTIONEAZA]->
  (p:Proiect)
2 WHERE f.denumire = 'Teamnet Iasi'
3 RETURN p.denumire
```



*Fig. 6 Clauzele MATCH, WHERE, RETURN în Neo4j*

### 3. UPDATE vs. SET

Deseori folosim clauza UPDATE, respectiv SET, atunci când dorim modificarea valorii deja stabilite pentru un câmp, respectiv o proprietate a nodului.

Presupunând că, în momentul populării cu date, pentru angajatul Caselden Jeff am greșit naționalitatea și vrem să o modificăm din “americană” în “britanică”, putem face acest lucru cu ajutorul clauzei UPDATE. Tabela pe care se dorește a se face modificarea este tabela “angajați”. Câmpul căruia îi vom modifica valoarea este ”naționalitate”. În clauza WHERE vom seta ca filtru numele celui pentru care dorim modificarea naționalității.

```
UPDATE angajati
SET nationalitate = 'britanica'
WHERE nume = 'Caselden Jeff'
```

În Neo4j apelăm la clauza SET atunci când dorim modificarea unei valori pentru o anumită proprietate. Selectăm nodurile țintă, adică angajații, prin clauza MATCH, filtrăm în funcție de numele angajatului (în clauza WHERE), modificăm naționalitatea (prin SET) și afișăm angajatul (RETURN).

```
MATCH (a:Angajat)
WHERE a.nume = 'Caselden Jeff'
SET a.nationalitate = 'britanica'
RETURN a
```

#### 4. DELETE vs. DELETE

Folosim clauza DELETE atunci când vrem să ștergem o înregistrare a unei tabele, spre exemplu un departament din tabela Departamente. Tot cu ajutorul aceleiași clauze putem șterge un nod și relațiile sale.

```
DELETE FROM Departamente
WHERE denumire = 'Suport'
```

Presupunând că filiala Amazon Dublin nu mai are departament de suport:

```
MATCH (f:Filiala)-[r:ARE_DEPARTAMENT]-(d:Departament)
WHERE a.denumire = 'Amazon Dublin' AND d.denumire = 'Suport'
DELETE d, r
```

#### REMOVE

Dacă dorim să ștergem doar o proprietate a nodului, vom folosi clauza REMOVE:

```
MATCH (a:Angajat)
WHERE a.nume = 'Caselden Jeff'
REMOVE a.nationalitate
RETURN a
```

Se remarcă faptul că în Neo4j modificările la nivel de nod sunt flexibile, în sensul că pentru un nod de tip Angajat se poate șterge proprietatea “naționalitate”, însă alt nod Angajat o poate conține fără probleme. Pe când în Oracle, ștergerea unei coloane (ex.: naționalitate) este valabilă pentru toate înregistrările tabelului Angajați, prin urmare niciun angajat nu va mai avea o naționalitate în baza noastră de date.

## 5. DROP vs. DELETE

Pentru ștergerea unui obiect al bazei de date, spre exemplu o tabelă, vom folosi clauza DROP:

**DROP** TABLE Proiecte

Pentru ștergerea tuturor nodurilor și a relațiilor dintre ele:

MATCH (n)-[r]-() **DELETE** n,r

## 2.3 Operații de intersecție, reuniune, diferență

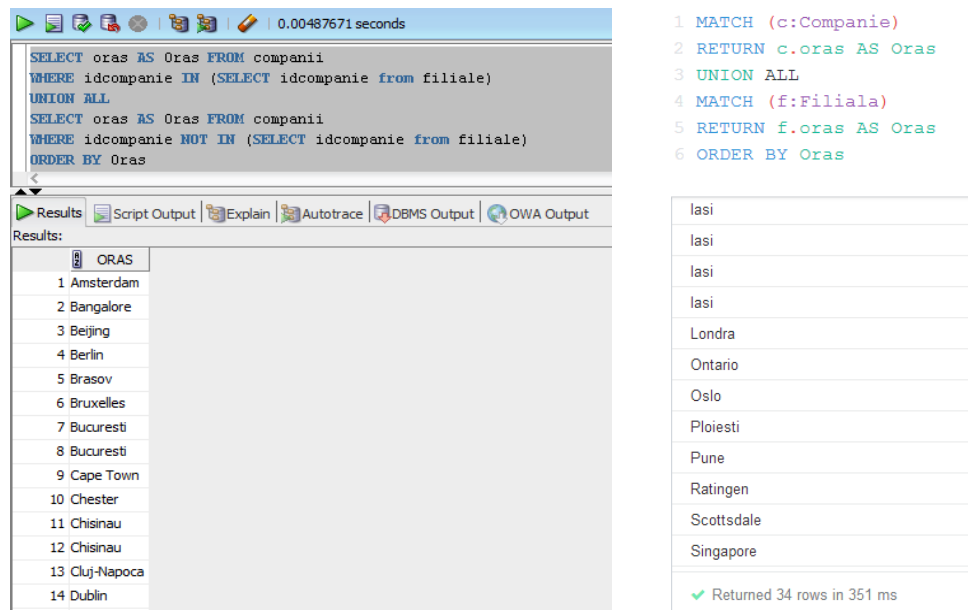
### 1. a) UNION ALL vs. UNION ALL

Când avem nevoie de afișarea combinată a seturilor de rezultate obținute în urma interogărilor, avem șansa de a folosi operatorul UNION ALL care ne permite realizarea cu ușurință a acestui lucru.

Dacă, spre exemplu, afișarea orașelor în care avem companii și filiale o putem realiza prin uniunea a două interogări: una care ne returnează orașele în care avem companii și una care ne va înapoia orașele în care există filiale.

```
SELECT oras AS Oras FROM companii
WHERE idcompanie IN (SELECT idcompanie from filiale)
UNION ALL
SELECT oras AS Oras FROM companii
WHERE idcompanie NOT IN (SELECT idcompanie from filiale)
ORDER BY Oras
```

```
MATCH (c:Comanie)
RETURN c.oras AS Oras
UNION ALL
MATCH (f:Filiala)
RETURN f.oras AS Oras
ORDER BY Oras
```

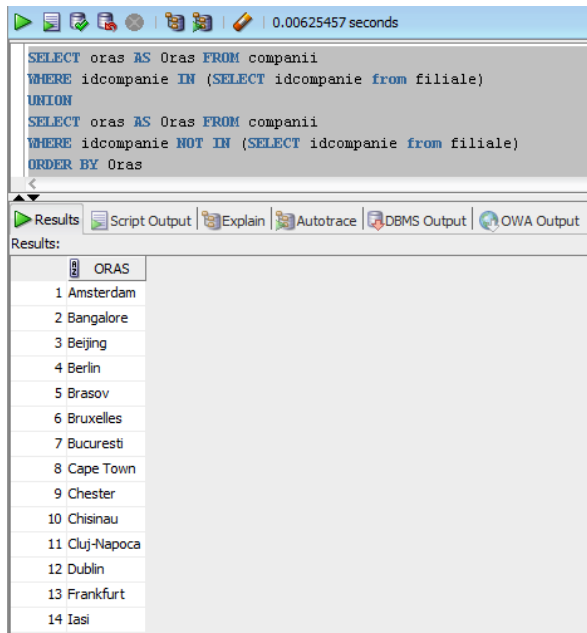


**Fig. 7 Operatorul UNION ALL în SQL - Oracle (stânga) și Neo4j (dreapta)**

După cum putem observa, cel mai mare inconvenient este faptul că valorile se repetă, lucru care nu ar fi de dorit. Dacă dorim ca valorile afișate să fie distincte, operatorul pe care îl vom folosi va fi UNION.

## b) UNION vs. UNION

```
SELECT oras AS Oras FROM companii
WHERE idcompanie IN (SELECT idcompanie from filiale)
UNION
SELECT oras AS Oras FROM companii
WHERE idcompanie NOT IN (SELECT idcompanie from filiale)
ORDER BY Oras
```



```

SELECT oras AS Oras FROM companii
WHERE idcompanie IN (SELECT idcompanie from filiale)
UNION
SELECT oras AS Oras FROM companii
WHERE idcompanie NOT IN (SELECT idcompanie from filiale)
ORDER BY Oras

```

```

1 MATCH (c:Companie)
2 RETURN c.oras AS Oras
3 UNION
4 MATCH (f:Filiala)
5 RETURN f.oras AS Oras
6 ORDER BY Oras

```

Dublin
Frankfurt
Iasi
Ontario
Oslo
Ploiesti
Pune
Ratingen
Scottsdale
Singapore
Stockholm
Tel-Aviv

✓ Returned 27 rows in 431 ms

**Fig. 8 Operatorul UNION în Oracle (stânga) și Neo4j (dreapta)**

```

MATCH (c:Companie)
RETURN c.oras AS Oras
UNION
MATCH (f:Filiala)
RETURN f.oras AS Oras
ORDER BY Oras

```

## 2. INTERSECT vs. IN

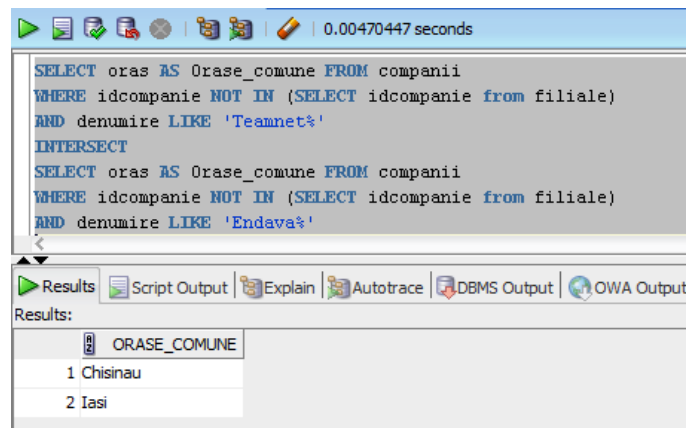
Atunci când dorim afișarea părții comune a două seturi de rezultate, apelăm la operatorul INTERSECT, respectiv IN.

Spre exemplu, dacă ni se cere afișarea oraselor comune în care Teamnet și Endava au filiale, vom recurge la:

```

SELECT oras AS Orase_comune FROM companii
WHERE idcompanie NOT IN (SELECT idcompanie from filiale)
AND denumire LIKE 'Teamnet%'
INTERSECT
SELECT oras AS Orase_comune FROM companii
WHERE idcompanie NOT IN (SELECT idcompanie from filiale)
AND denumire LIKE 'Endava%'

```



**Fig. 9 Operatorul INTERSECT în SQL (Oracle)**

```

MATCH (f:Filiala), (s:Filiala)
WHERE f.denumire=~'Teamnet.*' AND s.denumire=~'Endava.*'
RETURN FILTER(x IN f.oras WHERE x IN s.oras) AS Orase_comune

```

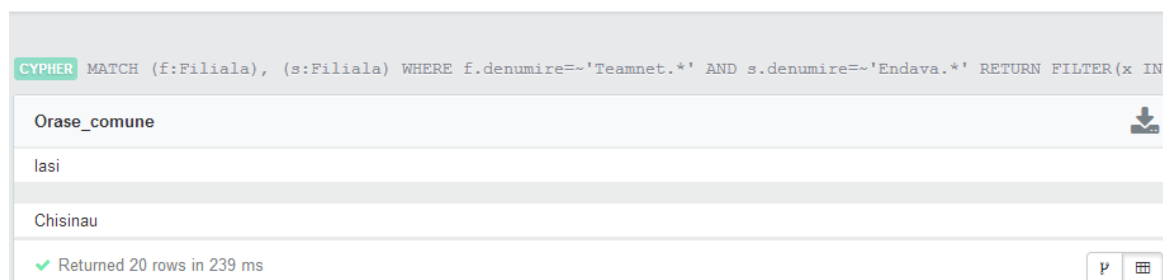
Operatorul “=~” are rol asemănător clauzei LIKE din SQL.

Ne interesează filialele Teamnet, așadar vom căuta denumirile care încep cu Teamnet (“.\*”), fără a mai ține cont de numele orașului (de exemplu, Teamnet Iași).

```

1 MATCH (f:Filiala), (s:Filiala)
2 WHERE f.denumire=~'Teamnet.*' AND s.denumire=~'Endava.*'
3 RETURN FILTER(x IN f.oras WHERE x IN s.oras) AS Orase_comune

```



**Fig. 10 Operatorul IN în Neo4j**

### 3. EXCEPT/ MINUS

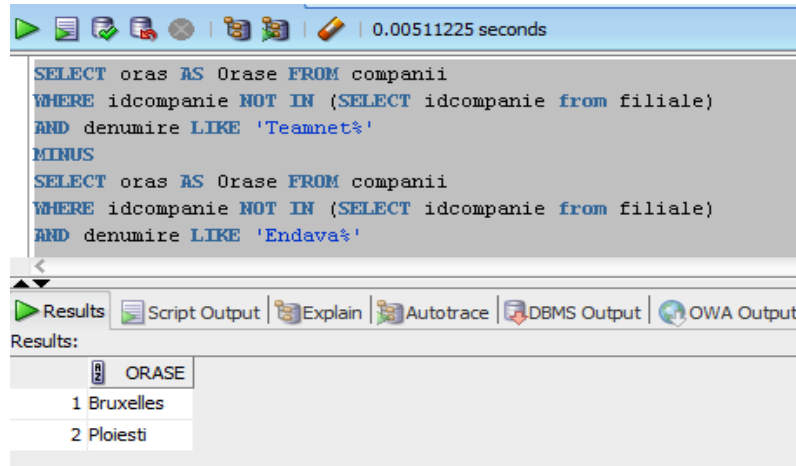
Opusul operației de mai sus, EXCEPT sau MINUS este operatorul care ne permite afișarea diferenței dintre două seturi de rezultate.

Presupunând că dorim afișarea orașelor în care sunt filiale Teamnet și nu sunt și filiale Endava, vom avea:

```

SELECT oras AS Orase FROM companii
WHERE idcompanie NOT IN (SELECT idcompanie from filiale)
AND denumire LIKE 'Teamnet%'
MINUS
SELECT oras AS Orase FROM companii
WHERE idcompanie NOT IN (SELECT idcompanie from filiale)
AND denumire LIKE 'Endava%'

```



*Fig.11 Operatorul MINUS în SQL (Oracle)*

## 2.4 Funcții scalare

### 1. COALESCE ( )

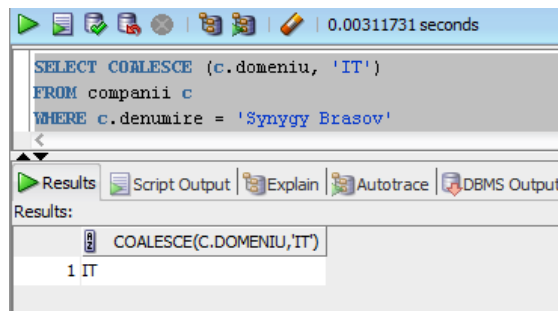
Spre deosebire de bazele de date graf, cele relaționale permit stocarea valorilor nule. În special, dar nu numai, când avem de realizat calcule pe baza valorilor cantitative, ne putem confrunta cu probleme tocmai datorită faptului că unele valori sunt nule și prin urmare rezultatul final nu va fi capabil să includă aceste valori nule. Datorită acestui fapt, clauza COALESCE ne permite transformarea valorilor NULL într-o valoare ne-nulă. Spre exemplu, dacă unele companii din bază de date nu au setat un domeniu din care fac parte și noi dorim să afișăm domeniile tuturor companiilor, rezultatul nu va fi tocmai plăcut când ne vor fi afișate valori nule.

Presupunând că toate companiile noastre sunt din domeniul IT, am decis ca pentru Synogy Brașov (ce nu are precizată o valoare pentru domeniu) să afișăm valoarea IT pentru câmpul “domeniu”.

```

SELECT COALESCE (c.domeniu, 'IT')
FROM companii c
WHERE c.denumire = 'Synogy Brasov'

```

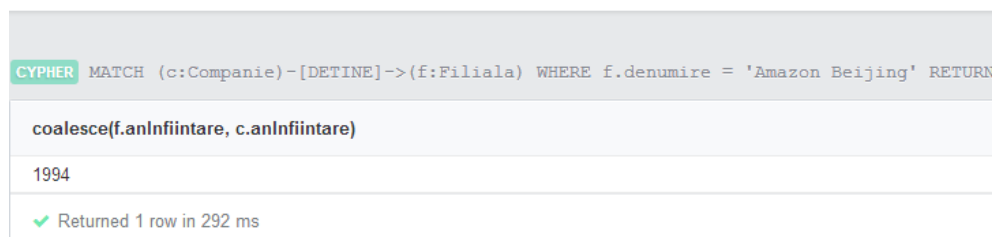


**Fig.12 Funcția COALESCE în SQL (Oracle)**

În mod asemănător, dacă filiala Amazon Beijing nu are setat un an de înființare, vom afișa pentru ea, anul de înființare a sediului central Amazon:

```
MATCH (c:Companie)-[DETINE]->(f:Filiala)
WHERE f.denumire = 'Amazon Beijing'
RETURN COALESCE(f.anInfiintare, c.anInfiintare)
```

```
1 MATCH (c:Companie)-[DETINE]->(f:Filiala)
2 WHERE f.denumire = 'Amazon Beijing'
3 RETURN coalesce(f.anInfiintare, c.anInfiintare)
```



**Fig.13 Funcția COALESCE în Neo4j**

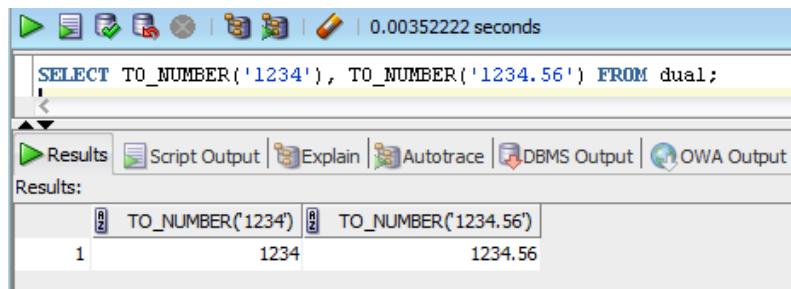
## 2. TO\_NUMBER () vs. TOINT () / TOFLOAT ()

Deseori, funcțiile de convertire a datelor de tip șir de caractere (string) în date numerice sunt foarte utile. În Oracle beneficiem de funcția TO\_NUMBER() care convertește datele CHAR/VARCHAR în numere. În Neo4j avem funcții separate pentru convertirea în numere întregi - TO\_INT() sau numere reale - TO\_FLOAT().

Transformarea șirurilor de caractere “1234” si “1234.56” în date numerice:

```
SELECT TO_NUMBER('1234'), TO_NUMBER('1234.56') FROM dual;
```





**Fig.14 Funcția TO\_NUMBER( ) în SQL (Oracle)**

Transformarea string-ului “1234” într-un număr întreg și a string-ului “1234.56” într-un număr real:

**RETURN TOINT("1234"), TOFLOAT("1234.56")**

`$ RETURN TOINT("1234"), TOFLOAT("1234.56")`

CYPHER RETURN TOINT("1234"), TOFLOAT("1234.56")	
TOINT("1234")	TOFLOAT("1234.56")
1234	1234.56
✓ Returned 1 row in 291 ms	

**Fig.15 Funcțiile TO\_INT(), TO\_FLOAT() în Neo4j**

### 3. TYPE ( )

O funcție specifică bazelor de date graf este TYPE() care ne returnează numele relației dintre două noduri.

Angajatul Huang Wei conduce o filială, prin urmare relația dintre el și un alt nod “subordonat” (filiala) este una de tip “CONDUCE”:

```
MATCH (a:Angajat)-[r]->()
WHERE a.nume='Huang Wei'
RETURN TYPE(r)
```

```

1 MATCH (a:Angajat)-[r]->()
2 WHERE a.num='Huang Wei'
3 RETURN TYPE(r)

```

CYPHER MATCH (a:Angajat)-[r]->() WHERE a.num='Huang Wei' RETURN TYPE(r)	
TYPE(r)	
CONDUCE	
✓ Returned 1 row in 197 ms	

**Fig.16 Funcția TYPE() în Neo4j**

### 3. ID()

O altă funcție specifică Neo4j, foarte utilă în unele cazuri, este funcția ID() care ne returnează indexul unui nod.

Dacă dorim aflarea indexului angajatului Hickler Tim, rulăm secvența:

```

MATCH (a:Angajat)
WHERE a.num='Hickler Tim'
RETURN ID(a)

```

```

1 MATCH (a:Angajat)
2 WHERE a.num='Hickler Tim'
3 RETURN ID(a)

```

CYPHER MATCH (a:Angajat) WHERE a.num='Hickler Tim' RETURN ID(a)	
ID(a)	
7	
✓ Returned 1 row in 177 ms	

**Fig.17 Funcția ID() în Neo4j**

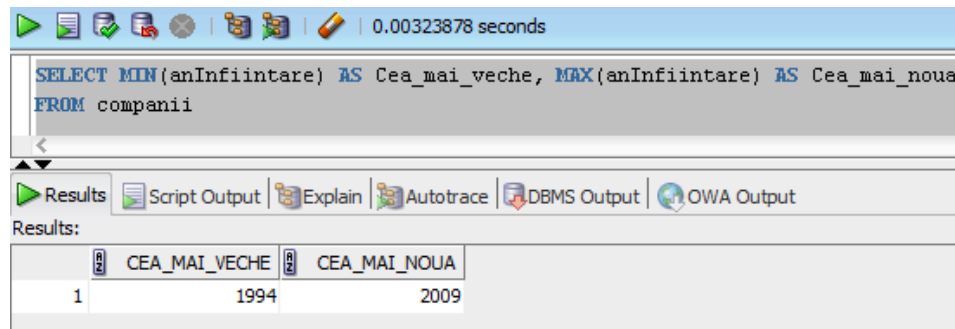
## 2.5 Funcții agregate

### 1. MIN() / MAX() vs. MIN() / MAX()

După cum le spune și numele, funcțiile MIN() și MAX() ne vor returna minimul valorii, respectiv maximul valorii pentru un anumit câmp precizat de noi.

Dacă dorim afișarea celei mai vechi, respectiv celei mai noi companii, vom căuta anul de înființare cel mai mic, respectiv cel mai mare:

```
SELECT MIN(anInfiintare) AS Cea_mai_veche, MAX(anInfiintare) AS
Cea_mai_noua
FROM companii
```



The screenshot shows the Oracle SQL Developer interface. The top toolbar includes icons for running, saving, and other database operations, along with a timer showing 0.00323878 seconds. The SQL editor contains the query: `SELECT MIN(anInfiintare) AS Cea_mai_veche, MAX(anInfiintare) AS Cea_mai_noua FROM companii`. Below the editor, the 'Results' tab is active, displaying a table with two columns: 'CEA\_MAI\_VECHЕ' and 'CEA\_MAI\_NOUA'. The first row shows the values 1994 and 2009.

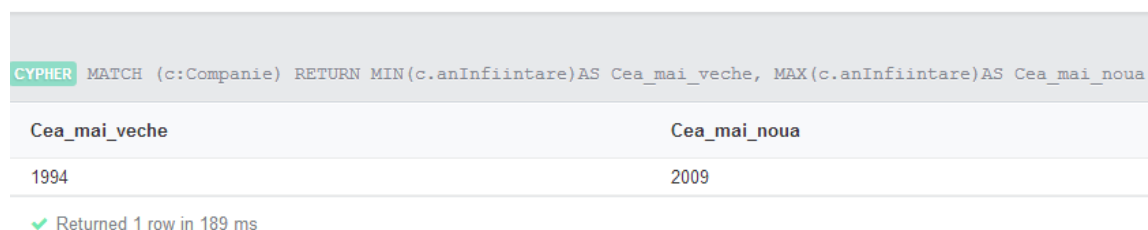
	CEA_MAI_VECHЕ	CEA_MAI_NOUA
1	1994	2009

*Fig.18 Funcțiile MIN(), MAX() în SQL (Oracle)*

Același rezultat obținut în Neo4j:

```
MATCH (c:Companie)
RETURN MIN(c.anInfiintare) AS Cea_mai_veche, MAX(c.anInfiintare) AS
Cea_mai_noua
```

```
1 MATCH (c:Companie)
2 RETURN MIN(c.anInfiintare)AS Cea_mai_veche, MAX(c.anInfiintare)AS
Cea_mai_noua
```



The screenshot shows the Neo4j Cypher query editor. The query is: `MATCH (c:Companie) RETURN MIN(c.anInfiintare)AS Cea_mai_veche, MAX(c.anInfiintare)AS Cea_mai_noua`. Below the query, the results are displayed in a table with two columns: 'Cea\_mai\_veche' and 'Cea\_mai\_noua'. The first row shows the values 1994 and 2009. At the bottom, a status bar indicates 'Returned 1 row in 189 ms'.

Cea_mai_veche	Cea_mai_noua
1994	2009

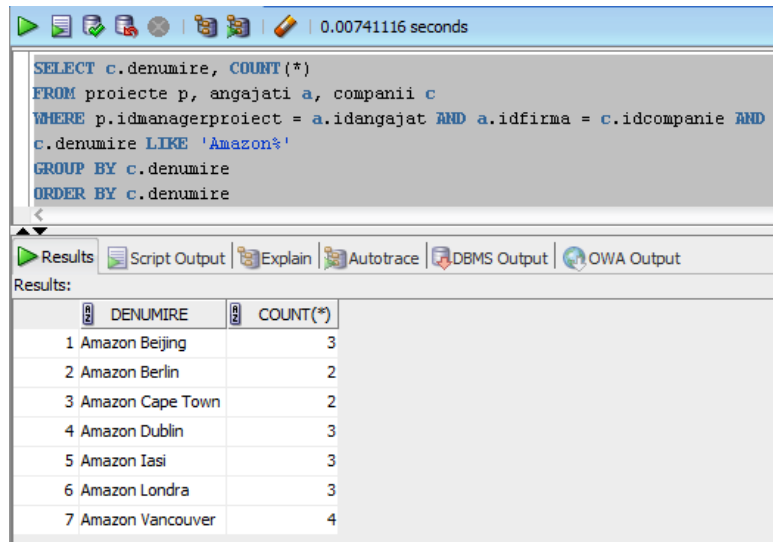
*Fig.19 Funcțiile MIN(), MAX() în Neo4j*

## 2. COUNT () vs. COUNT ()

Funcția COUNT () are ca scop returnarea numărului de linii a unei interogări. Spre exemplu, putem afișa numărul de proiecte al fiecărei filiale Amazon astfel:

```
SELECT c.denumire, COUNT(*)
FROM proiecte p, angajati a, companii c
WHERE p.idmanagerproiect = a.idangajat AND a.idfirma = c.idcompanie AND
```

```
c.denumire LIKE 'Amazon%'
GROUP BY c.denumire
ORDER BY c.denumire
```



The screenshot shows an Oracle SQL Developer window with a query editor at the top and a results grid at the bottom. The query is:
 

```
SELECT c.denumire, COUNT(*)
FROM proiecte p, angajati a, companii c
WHERE p.idmanagerproiect = a.idangajat AND a.idfirma = c.idcompanie AND
c.denumire LIKE 'Amazon%'
GROUP BY c.denumire
ORDER BY c.denumire
```

 The results grid shows 7 rows of data:
 

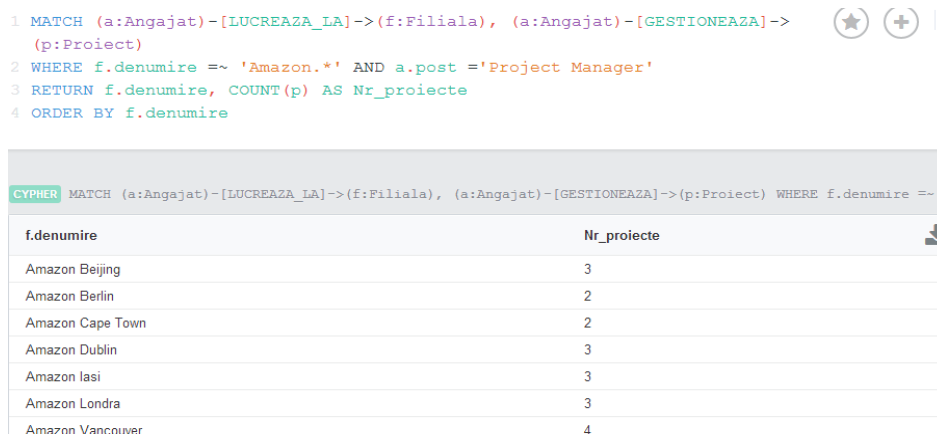
DENUMIRE	COUNT(*)
1 Amazon Beijing	3
2 Amazon Berlin	2
3 Amazon Cape Town	2
4 Amazon Dublin	3
5 Amazon Iasi	3
6 Amazon Londra	3
7 Amazon Vancouver	4

 The window title bar indicates a execution time of 0.00741116 seconds.

**Fig.20 Funcția COUNT() în SQL (Oracle)**

În mod similar, în Neo4j rezultatul va fi:

```
MATCH (a:Angajat)-[LUCREAZA_LA]->(f:Filiala), (a:Angajat)-[GESTIONEAZA]->(p:Proiect)
WHERE f.denumire =~ 'Amazon.*' AND a.post ='Project Manager'
RETURN f.denumire, COUNT(p) AS Nr_proiecte
ORDER BY f.denumire
```



The screenshot shows a Neo4j Cypher query editor with the following query:
 

```
1 MATCH (a:Angajat)-[LUCREAZA_LA]->(f:Filiala), (a:Angajat)-[GESTIONEAZA]->(p:Proiect)
2 WHERE f.denumire =~ 'Amazon.*' AND a.post ='Project Manager'
3 RETURN f.denumire, COUNT(p) AS Nr_proiecte
4 ORDER BY f.denumire
```

 Below the query editor, the results are displayed in a table:
 

f.denumire	Nr_proiecte
Amazon Beijing	3
Amazon Berlin	2
Amazon Cape Town	2
Amazon Dublin	3
Amazon Iasi	3
Amazon Londra	3
Amazon Vancouver	4

 The query editor interface includes a CYPHER label and a download icon for the results.

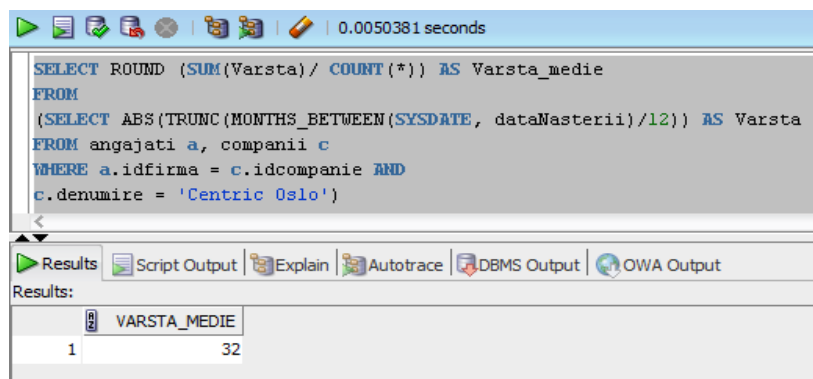
**Fig.21 Funcția COUNT() în Neo4j**

### 3. SUM () vs. SUM()

După cum îi spune și numele, funcția ne returnează suma valorilor pentru un anumit câmp cu date cantitative.

Să presupunem că vrem să afișăm vârsta medie a angajaților de la Centric Oslo. Similar logicii matematice, vom calcula media folosind SUM() /COUNT():

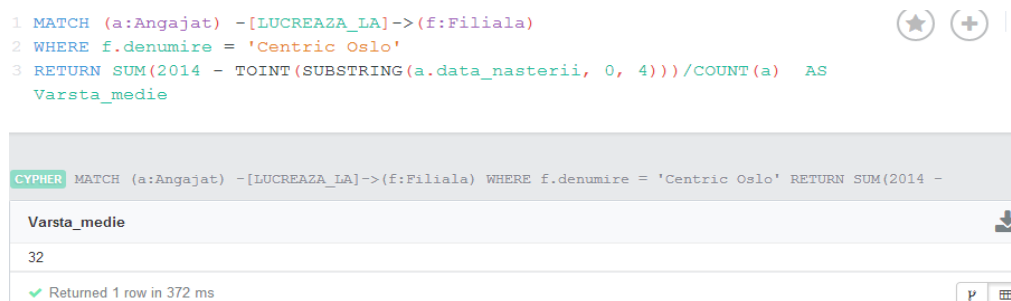
```
SELECT ROUND (SUM(Varsta)/ COUNT(*)) AS Varsta_medie
FROM
(SELECT ABS(TRUNC(MONTHS_BETWEEN(SYSDATE, dataNasterii)/12)) AS Varsta
FROM angajati a, companii c
WHERE a.idfirma = c.idcompanie AND
c.denumire = 'Centric Oslo')
```



**Fig.22 Funcția SUM() în SQL (Oracle)**

În Neo4j logica este aceeași:

```
MATCH (a:Angajat) -[LUCREAZA_LA]->(f:Filiala)
WHERE f.denumire = 'Centric Oslo'
RETURN SUM(2014 - TOINT(SUBSTRING(a.data_nasterii, 0, 4)))/COUNT(a) AS
Varsta_medie
```



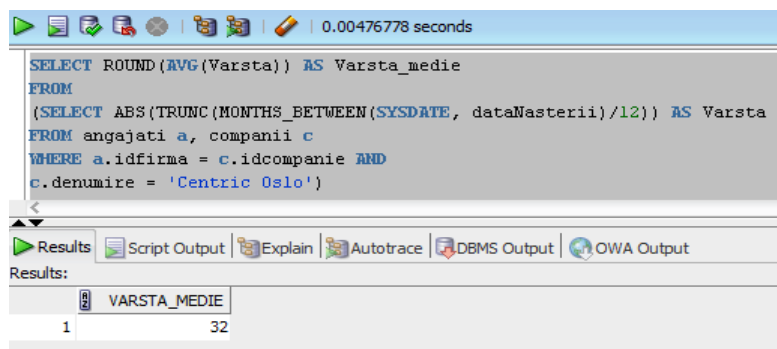
**Fig.23 Funcția SUM() în Neo4j**

#### 4. AVG() vs. AVG()

Această funcție returnează media valorilor unei expresii.

Acceași problemă de mai sus se poate rezolva, mai simplu, folosind funcția AVG ( ):

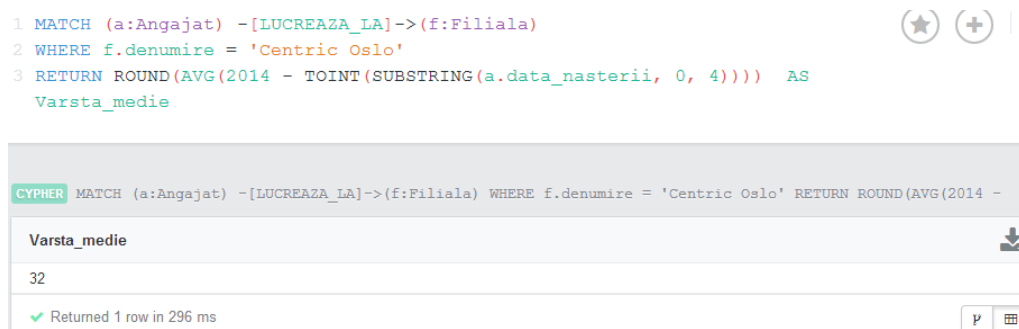
```
SELECT ROUND(AVG(Varsta)) AS Varsta_medie
FROM
(SELECT ABS(TRUNC(MONTHS_BETWEEN(SYSDATE, dataNasterii)/12)) AS Varsta
FROM angajati a, companii c
WHERE a.idfirma = c.idcompanie AND
c.denumire = 'Centric Oslo')
```



**Fig.24 Funcția AVG() în SQL (Oracle)**

Respectiv sintaxa Cypher:

```
MATCH (a:Angajat) -[LUCREAZA_LA]->(f:Filiala)
WHERE f.denumire = 'Centric Oslo'
RETURN ROUND(AVG(2014 - TOINT(SUBSTRING(a.data_nasterii, 0, 4)))) AS
Varsta_medie
```



**Fig.25 Funcția AVG() în SQL (Oracle)**

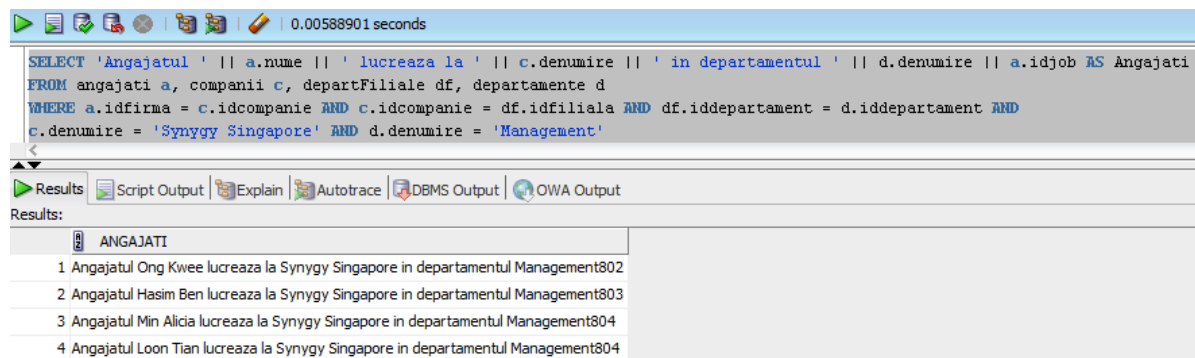
## 2.6 Funcții pentru șiruri de caractere

### Concatenarea șirurilor de caractere: “||” vs. “+”

În primul rând trebuie făcut cunoscut procedeul de concatenare a două șiruri de caractere. Dacă în Oracle se folosea “||” în acest scop, Neo4j a ales “+”-ul ca și caracter de alipire, utilizat pe scară largă și în multe limbaje de programare.

Spre exemplu, dorim o afișare a angajaților din departamentul de Management al filialei Syngy Singapore:

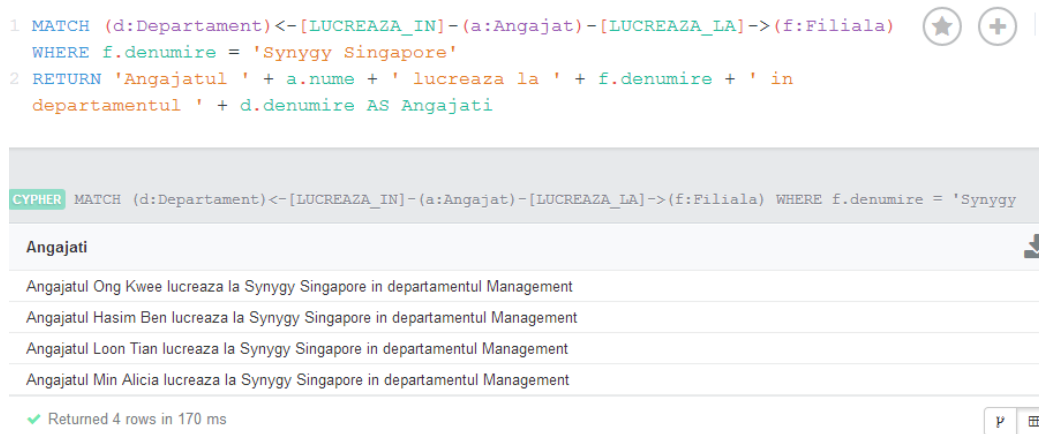
```
SELECT 'Angajatul ' || a.nume || ' lucreaza la ' || c.denumire || ' in departamentul ' || d.denumire || a.idjob AS Angajati
FROM angajati a, companii c, departFiliale df, departamente d
WHERE a.idfirma = c.idcompanie AND c.idcompanie = df.idfiliala AND df.iddepartament = d.iddepartament AND
c.denumire = 'Syngy Singapore' AND d.denumire = 'Management'
```



**Fig.26 Concatenarea șirurilor de caractere în SQL (Oracle)**

Sintaxa Cypher și rezultatul în Neo4j:

```
MATCH (d:Departament)<-[LUCREAZA_IN]-(a:Angajat)-[LUCREAZA_LA]->(f:Filiala) WHERE f.denumire = 'Syngy Singapore'
RETURN 'Angajatul ' + a.nume + ' lucreaza la ' + f.denumire + ' in departamentul ' + d.denumire AS Angajati
```



**Fig.27 Concatenarea șirurilor de caractere în Neo4j**

## 1. SUBSTR() vs SUBSTRING() / LEFT() /RIGHT ()

E necesar de precizat de la bun început că în SQL numerotarea caracterelor începe de la 1, pe când în Cypher începe de la 0. Ca și noutate, Neo4j are în plus funcțiile LEFT și RIGHT, neexistente în Oracle.

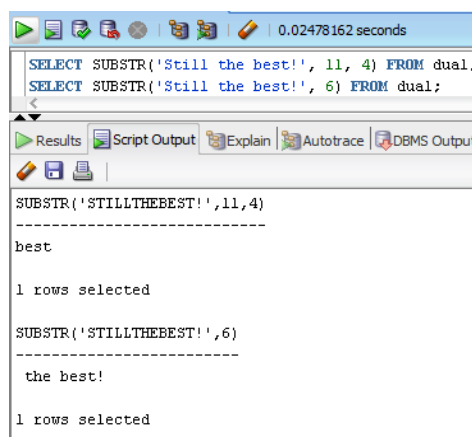
Intuiția nu ne înșală atunci când ne gândim la scopul funcției SUBSTR(), respectiv SUBSTRING(), pentru că într-adevăr ea este destinată extragerii unui set de caractere dintr-un șir de caractere.

În exemplul banal, dar concludiv, de mai jos, 11 este numărul caracterului de la care se începe extragerea sub-șirului, iar 4 este lungimea șirului extras.

```
SELECT SUBSTR('Still the best!', 11, 4) FROM dual;
```

În cazul în care nu specificăm lungimea șirului care se dorește a fi extras, în mod implicit, extragerea se va derula până la ultimul caracter.

```
SELECT SUBSTR('Still the best!', 6) FROM dual;
```



**Fig.28 Funcția SUBSTR() în SQL (Oracle)**



```
RETURN SUBSTRING("Still the best!",10 , 4), SUBSTRING("Still the best!",  
6)
```

```
$ RETURN SUBSTRING("Still the best!",10 , 4), SUBSTRING("Still the best!", 6)
```

CYPHER RETURN SUBSTRING("Still the best!",10 , 4), SUBSTRING("Still the best!", 6)	
SUBSTRING("Still the best!",10 , 4)	SUBSTRING("Still the best!", 6)
best	the best!
✓ Returned 1 row in 215 ms	

**Fig.29 Funcția SUBSTRING() în Neo4j**

Precizam mai sus că funcțiile LEFT() și RIGHT() sunt prezente doar în Neo4j.

LEFT() are rolul de a extrage caractere din partea stângă a șirului (a se urmări primul rezultat), iar RIGHT() caracterele din dreapta (a se vedea al doilea rezultat).

```
RETURN LEFT("Still the best!", 5)  
RETURN RIGHT("Still the best!", 5)
```

```
$ RETURN LEFT("Still the best!", 5)
```

CYPHER RETURN LEFT("Still the best!", 5)	
LEFT("Still the best!", 5)	
Still	
✓ Returned 1 row in 117 ms	

```
$ RETURN RIGHT("Still the best!", 5)
```

CYPHER RETURN RIGHT("Still the best!", 5)	
RIGHT("Still the best!", 5)	
best!	
✓ Returned 1 row in 126 ms	

**Fig.30 Funcțiile LEFT(), RIGHT() în Neo4j**

## 2. TO\_CHAR () vs. STR()

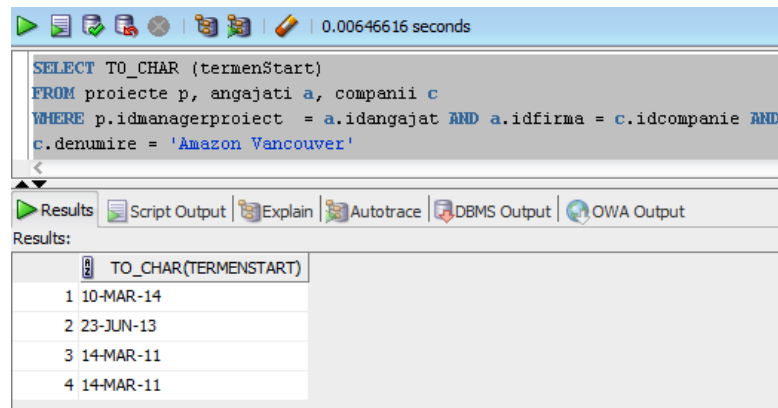
Când avem nevoie de o funcție care să convertească tipul nostru de date într-un tip String, apelăm la TO\_CHAR(), respectiv STR()

Data de start a proiectelor de la Amazon Vancouver:

```

SELECT TO_CHAR (termenStart)
FROM proiecte p, angajati a, companii c
WHERE p.idmanagerproiect = a.idangajat AND a.idfirma = c.idcompanie AND
c.denumire = 'Amazon Vancouver'

```



TO_CHAR(TERMENSTART)
1 10-MAR-14
2 23-JUN-13
3 14-MAR-11
4 14-MAR-11

**Fig.31 Funcția TO\_CHAR() în Oracle**

Echivalentul Neo4j este:

```

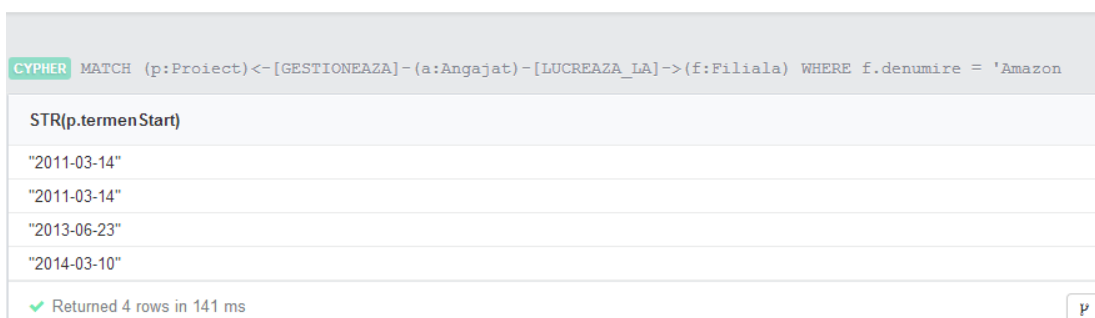
MATCH (p:Proiect)<-[GESTIONEAZA]-(a:Angajat)-[LUCREAZA_LA]->(f:Filiala)
WHERE f.denumire = 'Amazon Vancouver'
RETURN STR(p.termenStart)

```

```

1 MATCH (p:Proiect)<-[GESTIONEAZA]-(a:Angajat)-[LUCREAZA_LA]->(f:Filiala)
2 WHERE f.denumire = 'Amazon Vancouver'
3 RETURN STR(p.termenStart)

```



STR(p.termenStart)
"2011-03-14"
"2011-03-14"
"2013-06-23"
"2014-03-10"

Returned 4 rows in 141 ms

**Fig.32 Funcția STR( ) în Neo4j**

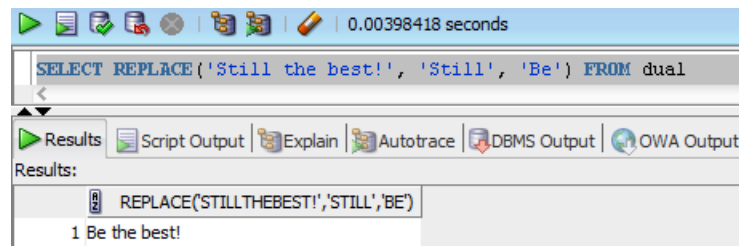
#### 4. REPLACE( ) vs. REPLACE( )

Funcția REPLACE( ) este utilă atunci când dorim înlocuirea unui grup de caractere cu un alt grup de caractere din cadrul unui șir.

```

SELECT REPLACE('Still the best!', 'Still', 'Be') FROM dual

```

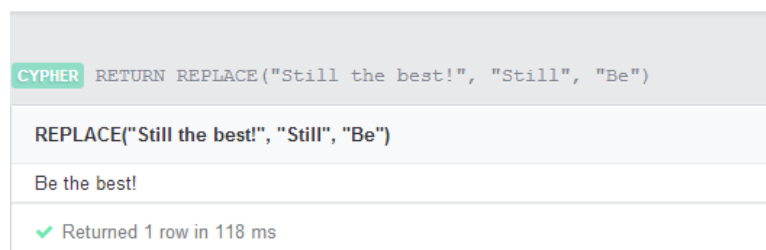


**Fig.33 Funcția REPLACE( ) în Oracle**

În mod similar, în Neo4j vom avea:

```
RETURN REPLACE("Still the best!", "Still", "Be")
```

```
$ RETURN REPLACE("Still the best!", "Still", "Be")
```

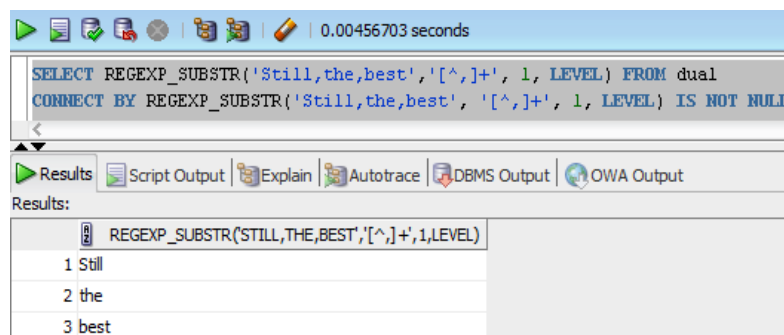


**Fig.34 Funcția REPLACE( ) în Neo4j**

## 5. REGEXP\_SUBSTR ( ) vs. SPLIT ( )

Folosim aceste funcții când este necesară afișarea șirurilor de caractere pe mai multe rânduri.

```
SELECT REGEXP_SUBSTR('Still,the,best','[^,]+', 1, LEVEL) FROM dual
CONNECT BY REGEXP_SUBSTR('Still,the,best', '[^,]+', 1, LEVEL) IS NOT
NULL
```

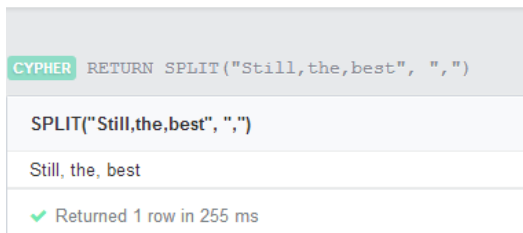


**Fig.35 Funcția REGEXP\_SUBSTR( ) în SQL (Oracle)**

În Neo4j vom avea:

```
RETURN SPLIT("Still,the,best", ",")
```

```
$ RETURN SPLIT("Still,the,best", ",")
```



**Fig.36 Funcția SPLIT( ) în Neo4j**

Avem însă și un set de funcții care nu implică deosebiri, decât bineînțeles la nivelul construcției sintactice:

**5. TRIM()/ LTRIM( )/ RTRIM( ) vs TRIM( )/ LTRIM( )/ RTRIM( )** - trunchiază caractere din partea stângă (LTRIM) sau dreaptă (RTRIM), sau din ambele părți (TRIM) ale unui șir de caractere precizat.

**6. LOWER ( ) /UPPER( ) vs. LOWER ( ) /UPPER( )** - aplicate unui șir de caractere, transformă literele mari în mici (LOWER) și invers (UPPER).

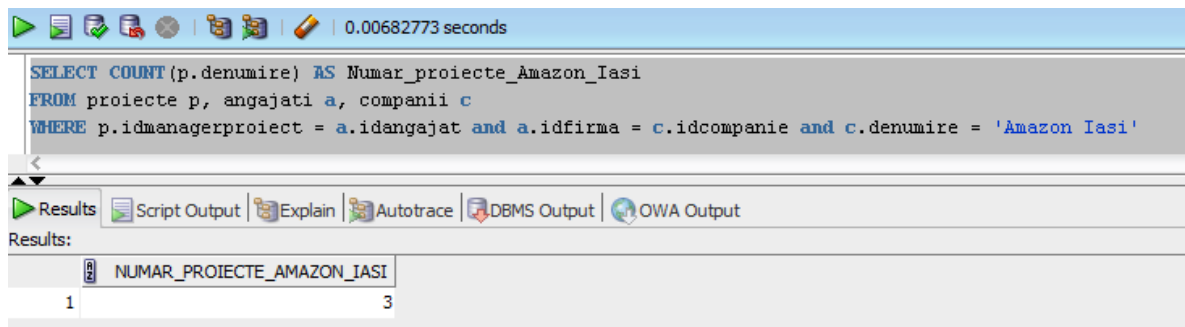
## 2.7 Alte clauze și cuvinte-cheie

### 1. AS vs. AS

Și în Oracle, dar și în Neo4j, pentru stabilirea unui alias folosim AS.

Dacă, spre exemplu, dorim să afișăm numărul de proiecte pe care Amazon Iași îl deține până în prezent:

```
SELECT COUNT(p.denumire) AS Numar_proiecte_Amazon_Iasi
FROM proiecte p, angajati a, companii c
WHERE p.idmanagerproiect = a.idangajat and a.idfirma = c.idcompanie and
c.denumire = 'Amazon Iasi'
```



*Fig.37 Clauza AS în SQL (Oracle)*

În mod similar, în Neo4j:

```
MATCH (a:Angajat)-[LUCREAZA_LA]->(f:Filiala), (a:Angajat)-[GESTIONEAZA]->(p:Proiect)
WHERE f.denumire = 'Amazon Iasi' AND a.post = 'Project Manager'
RETURN COUNT(p) AS Proiecte_Amazon_Iasi
```



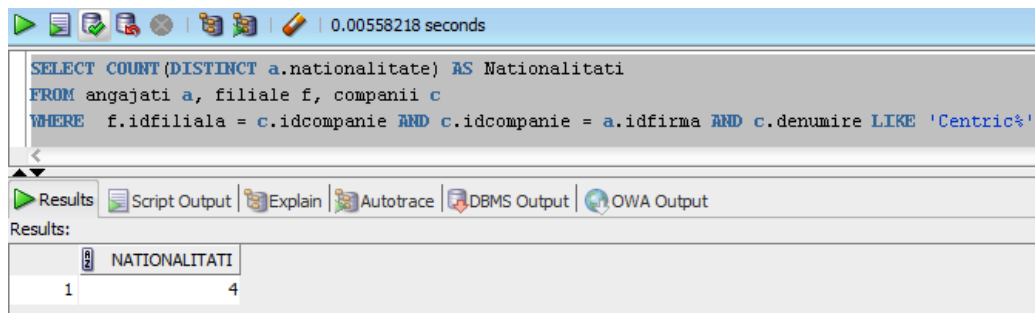
*Fig.38 Clauza AS în Neo4j*

## 2. DISTINCT vs. DISTINCT

Pentru îndepărtarea duplicatelor, atât în Oracle cât și în Neo4j avem posibilitatea de a utiliza clauza DISTINCT.

Afișarea numărului distinct de naționalități întâlnite în filialele companiei Centric:

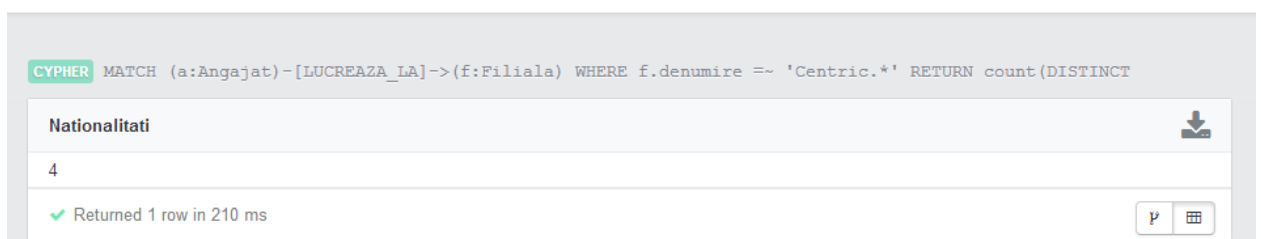
```
SELECT COUNT(DISTINCT a.nationalitate) AS Nationalitati
FROM angajati a, filiale f, companii c
WHERE f.idfiliala = c.idcompanie AND c.idcompanie = a.idfirma AND
c.denumire LIKE 'Centric%'
```



**Fig.39 Clauza DISTINCT în SQL (Oracle)**

```
MATCH (a:Angajat)-[LUCREAZA_LA]->(f:Filiala)
WHERE f.denumire =~ 'Centric.*'
RETURN count(DISTINCT a.nationalitate) AS Nationalitati
```

```
1 MATCH (a:Angajat)-[LUCREAZA_LA]->(f:Filiala)
2 WHERE f.denumire =~ 'Centric.*'
3 RETURN count(DISTINCT a.nationalitate) AS Nationalitati
```



**Fig.40 Clauza DISTINCT în Neo4j**

### 3. NOT vs. NOT

Operatorul logic NOT este folosit pentru negarea unei condiții.  
Dacă dorim afișarea companiilor care nu au fost înființate în România:

```
SELECT c.denumire
FROM companii c
WHERE NOT(c.tara = 'Romania')
and idcompanie IN (
SELECT c.idcompanie FROM companii c
MINUS
SELECT f.idfiliala FROM filiale f)
```

**Oracle SQL Query:**

```
SELECT c.denumire
FROM companii c
WHERE NOT(c.tara = 'Romania')
and idcompanie IN (
SELECT c.idcompanie FROM companii c
MINUS
SELECT f.idfiliala FROM filiale f)
```

**Neo4j Cypher Query:**

```
1 MATCH (c:Companie)
2 WHERE NOT (c.tara = 'Romania')
3 RETURN c.denumire
```

**Results:**

DENUMIRE
1 Amazon
2 Endava
3 Centric
4 Syngy
5 Ness

**Cypher Results:**

c.denumire
Amazon
Endava
Centric
Syngy
Ness

Returned 5 rows in 307 ms

**Fig.41 Operatorul NOT în SQL - Oracle (stânga) și Neo4j (dreapta)**

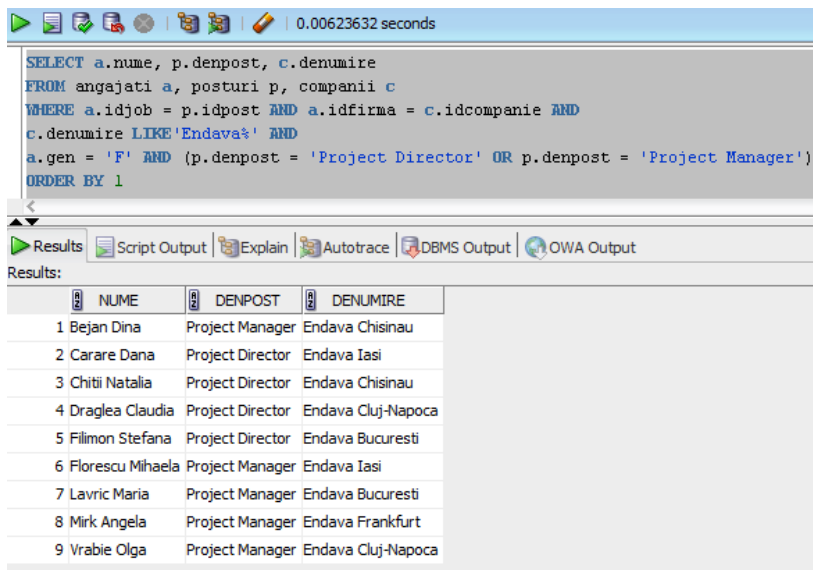
```
MATCH (c:Companie)
WHERE NOT (c.tara = 'Romania')
RETURN c.denumire
```

#### 4. AND vs. AND si OR vs. OR

Ca și NOT, operatorii logici AND și OR manipulează rezultatul unei condiții.

Afișarea tuturor femeilor care ocupă un post de Project Director sau Project Manager în cadrul filialelor companiei Endava:

```
SELECT a.numa, p.denpost, c.denumire
FROM angajati a, posturi p, companii c
WHERE a.idjob = p.idpost AND a.idfirma = c.idcompanie AND
c.denumire LIKE 'Endava%' AND
a.gen = 'F' AND (p.denpost = 'Project Director' OR p.denpost = 'Project
Manager')
ORDER BY 1
```



0.00623632 seconds

```

SELECT a.num, p.denpost, c.denumire
FROM angajati a, posturi p, companii c
WHERE a.idjob = p.idpost AND a.idfirma = c.idcompanie AND
c.denumire LIKE 'Endava%' AND
a.gen = 'F' AND (p.denpost = 'Project Director' OR p.denpost = 'Project Manager')
ORDER BY 1

```

Results:

	NUME	DENPOST	DENUMIRE
1	Bejan Dina	Project Manager	Endava Chisinau
2	Carare Dana	Project Director	Endava Iasi
3	Chitii Natalia	Project Director	Endava Chisinau
4	Draglea Claudia	Project Director	Endava Cluj-Napoca
5	Filimon Stefana	Project Director	Endava Bucuresti
6	Florescu Mihaela	Project Manager	Endava Iasi
7	Lavric Maria	Project Manager	Endava Bucuresti
8	Mirk Angela	Project Manager	Endava Frankfurt
9	Vrabie Olga	Project Manager	Endava Cluj-Napoca

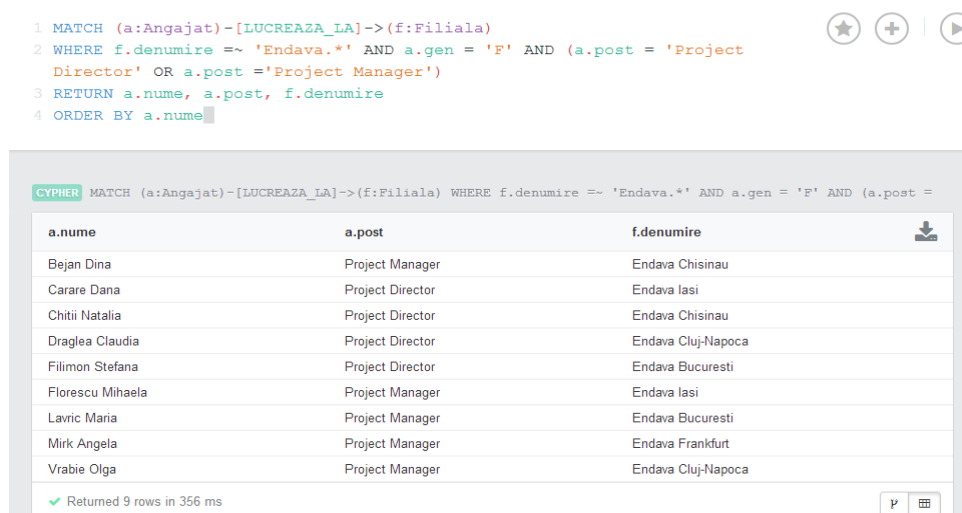
**Fig.42 Operatorii AND și OR în SQL (Oracle)**

Respectiv în Neo4j:

```

MATCH (a:Angajat)-[LUCREAZA_LA]->(f:Filiala)
WHERE f.denumire =~ 'Endava.*' AND a.gen = 'F' AND (a.post = 'Project
Director' OR a.post ='Project Manager')
RETURN a.num, a.post, f.denumire
ORDER BY a.num

```



```

1 MATCH (a:Angajat)-[LUCREAZA_LA]->(f:Filiala)
2 WHERE f.denumire =~ 'Endava.*' AND a.gen = 'F' AND (a.post = 'Project
3 Director' OR a.post ='Project Manager')
4 RETURN a.num, a.post, f.denumire
5 ORDER BY a.num

```

CYPHER MATCH (a:Angajat)-[LUCREAZA\_LA]->(f:Filiala) WHERE f.denumire =~ 'Endava.\*' AND a.gen = 'F' AND (a.post =

a.num	a.post	f.denumire
Bejan Dina	Project Manager	Endava Chisinau
Carare Dana	Project Director	Endava Iasi
Chitii Natalia	Project Director	Endava Chisinau
Draglea Claudia	Project Director	Endava Cluj-Napoca
Filimon Stefana	Project Director	Endava Bucuresti
Florescu Mihaela	Project Manager	Endava Iasi
Lavric Maria	Project Manager	Endava Bucuresti
Mirk Angela	Project Manager	Endava Frankfurt
Vrabie Olga	Project Manager	Endava Cluj-Napoca

Returned 9 rows in 356 ms

**Fig.43 Operatorii AND și OR în Neo4j**

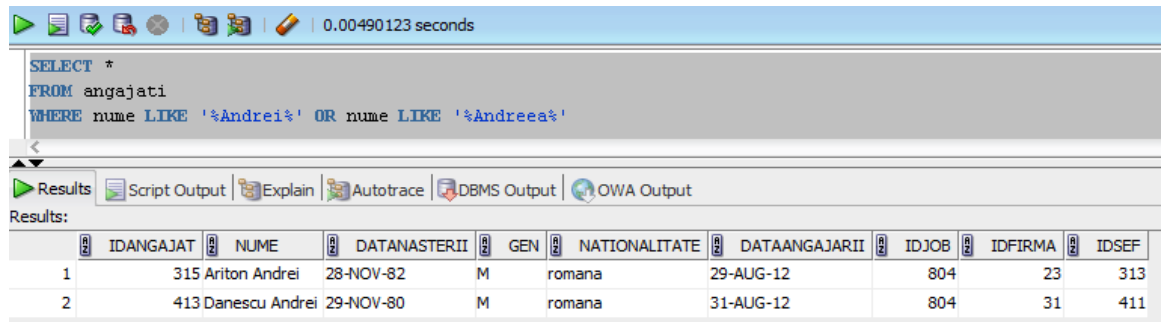


## 5. LIKE vs. “=~”

Condiția LIKE din Oracle returnează True atunci când prima expresie se încadrează în tiparul celei de-a doua.

Spre exemplu, afișarea angajaților care își serbează ziua numelui de Sfântul Andrei:

```
SELECT *  
FROM angajati  
WHERE nume LIKE '%Andrei%' OR nume LIKE '%Andreea%'
```



0.00490123 seconds

```
SELECT *  
FROM angajati  
WHERE nume LIKE '%Andrei%' OR nume LIKE '%Andreea%'
```

Results | Script Output | Explain | Autotrace | DBMS Output | OWA Output

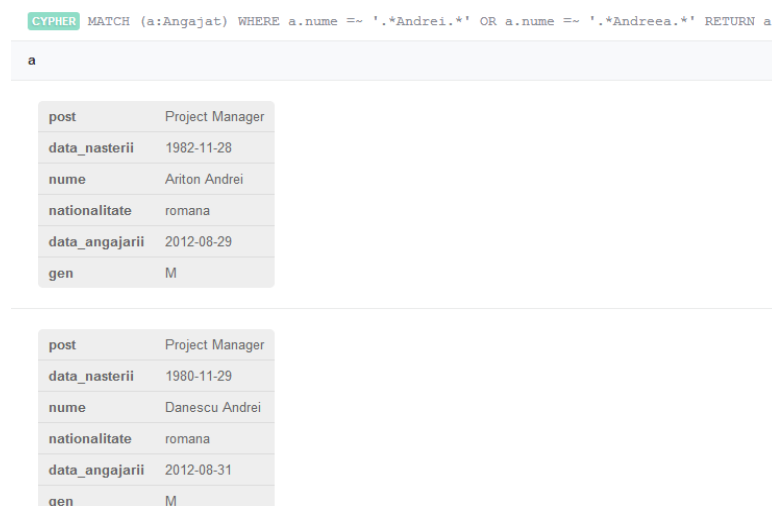
Results:

	IDANGAJAT	NUME	DATANASTERII	GEN	NATIONALITATE	DATAANGAJARII	IDJOB	IDFIRMA	IDSEF
1	315	Ariton Andrei	28-NOV-82	M	romana	29-AUG-12	804	23	313
2	413	Danescu Andrei	29-NOV-80	M	romana	31-AUG-12	804	31	411

**Fig.44 Condiția LIKE în SQL (Oracle)**

În Neo4j, putem realiza același lucru folosind “=~”:

```
MATCH (a:Angajat)  
WHERE a.nume =~ '.*Andrei.*' OR a.nume =~ '.*Andreea.*'  
RETURN a
```



CYPHER MATCH (a:Angajat) WHERE a.nume =~ '.\*Andrei.\*' OR a.nume =~ '.\*Andreea.\*' RETURN a

a

post	Project Manager
data_nasterii	1982-11-28
nume	Ariton Andrei
nationalitate	romana
data_angajarii	2012-08-29
gen	M

post	Project Manager
data_nasterii	1980-11-29
nume	Danescu Andrei
nationalitate	romana
data_angajarii	2012-08-31
gen	M

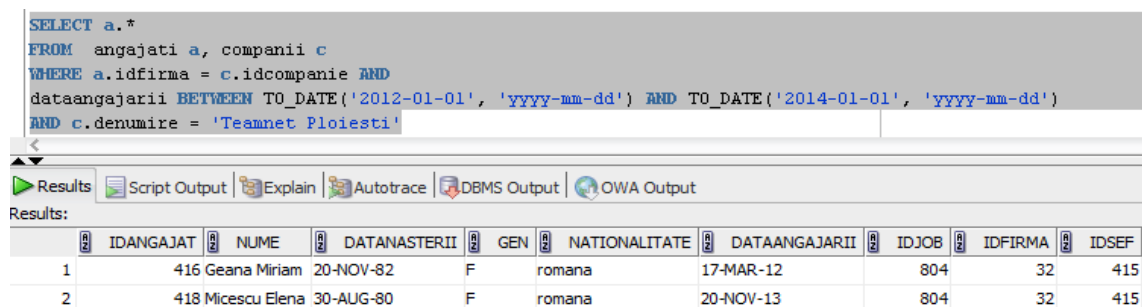
**Fig.45 Operatorul “=~” în Neo4j**

## 6. BETWEEN vs. “<=” și “>=”

Obținerea valorilor încadrate într-un interval, se poate face apelând la operatorul BETWEEN din Oracle, respectiv “<=” și “>=” din Neo4j.

Presupunând că dorim să afișăm angajările de la Teamnet Ploiești din ultimii 2 ani:

```
SELECT a.*
FROM angajati a, companii c
WHERE a.idfirma = c.idcompanie AND
dataangajarii BETWEEN TO_DATE('2012-01-01', 'yyyy-mm-dd') AND
TO_DATE('2014-01-01', 'yyyy-mm-dd')
AND c.denumire = 'Teamnet Ploiesti'
```

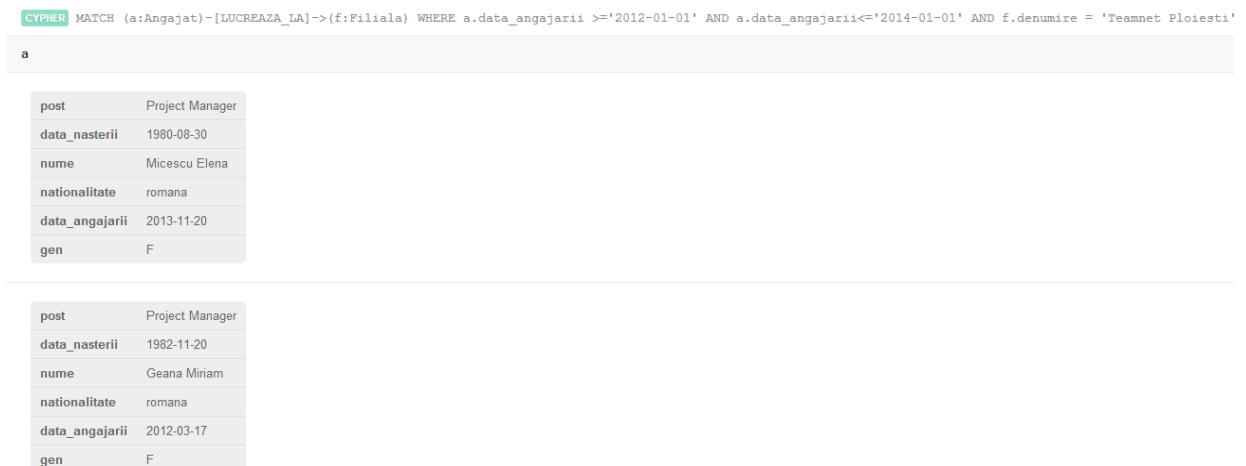


	IDANGAJAT	NUME	DATANASTERII	GEN	NATIONALITATE	DATAANGAJARII	IDJOB	IDFIRMA	IDSEF
1	416	Geana Miriam	20-NOV-82	F	romana	17-MAR-12	804	32	415
2	418	Micescu Elena	30-AUG-80	F	romana	20-NOV-13	804	32	415

*Fig.46 Operatorul BETWEEN în SQL (Oracle)*

În Neo4j interogarea echivalentă va fi:

```
MATCH (a:Angajat)-[LUCREAZA_LA]->(f:Filiala)
WHERE a.data_angajarii >= '2012-01-01' AND a.data_angajarii <= '2014-01-01' AND f.denumire = 'Teamnet Ploiesti'
RETURN a
```



CYPHER MATCH (a:Angajat)-[LUCREAZA\_LA]->(f:Filiala) WHERE a.data\_angajarii >='2012-01-01' AND a.data\_angajarii<='2014-01-01' AND f.denumire = 'Teamnet Ploiesti'

a

post	Project Manager
data_nasterii	1980-08-30
nume	Micescu Elena
nationalitate	romana
data_angajarii	2013-11-20
gen	F

post	Project Manager
data_nasterii	1982-11-20
nume	Geana Miriam
nationalitate	romana
data_angajarii	2012-03-17
gen	F

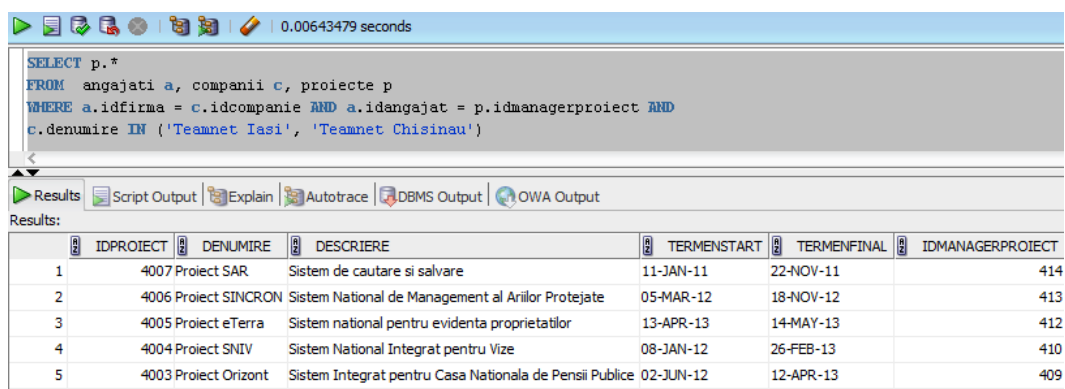
*Fig.47 Operatorii “<=” și “>=” în Neo4j*

## 7. IN ( values ) vs. IN [ values ]

Operatorul IN, folosit în ambele tipuri de baze de date, are scopul de a evalua dacă fiecare membru al listei este prezent ca și valoare pentru un anumit câmp specificat de noi.

Atunci când dorim afișarea proiectelor din cadrul filialelor Teamnet Iași și Teamnet Chișinău, vom rula secvența:

```
SELECT p.*  
FROM angajati a, companii c, proiecte p  
WHERE a.idfirma = c.idcompanie AND a.idangajat = p.idmanagerproiect AND  
c.denumire IN ('Teamnet Iasi', 'Teamnet Chisinau')
```



The screenshot shows an Oracle SQL Developer window with a query editor at the top and a results grid at the bottom. The query is: `SELECT p.* FROM angajati a, companii c, proiecte p WHERE a.idfirma = c.idcompanie AND a.idangajat = p.idmanagerproiect AND c.denumire IN ('Teamnet Iasi', 'Teamnet Chisinau')`. The results grid displays 5 rows of data with columns: IDPROIECT, DENUMIRE, DESCRIERE, TERMENSTART, TERMENFINAL, and IDMANAGERPROIECT.

IDPROIECT	DENUMIRE	DESCRIERE	TERMENSTART	TERMENFINAL	IDMANAGERPROIECT
1	4007 Proiect SAR	Sistem de cautare si salvare	11-JAN-11	22-NOV-11	414
2	4006 Proiect SINCRON	Sistem National de Management al Arilor Protejate	05-MAR-12	18-NOV-12	413
3	4005 Proiect eTerra	Sistem national pentru evidenta proprietatilor	13-APR-13	14-MAY-13	412
4	4004 Proiect SNIV	Sistem National Integrat pentru Vize	08-JAN-12	26-FEB-13	410
5	4003 Proiect Orizont	Sistem Integrat pentru Casa Nationala de Pensii Publice	02-JUN-12	12-APR-13	409

*Fig.48 Operatorul IN în Oracle*

În Neo4j, singura diferență este folosirea parantezelor pătrate în locul celor rotunde.

```
MATCH (f:Filiala)<-[LUCREAZA_LA]-(a:Angajat)-[GESTIONEAZA]->(p:Proiect)  
WHERE f.denumire IN ['Teamnet Iasi', 'Teamnet Chisinau']  
RETURN p
```

**CYPHER** MATCH (f:Filiala)->[LUCREAZA\_LA]-(a:Angajat)->[GESTIONEAZA]-(p:Proiect) WHERE f.denumire IN ['Teamnet Iasi', 'Teamnet Chisinau'] RETURN p

**p**

denumire	Proiect Orizont
descriere	Sistem Integrat pentru Casa Națională de Pensii Publice
termenStart	2012-08-02
termenFinal	2013-04-12

---

denumire	Proiect SNIV
descriere	Sistem Național Integrat pentru Vize
termenStart	2012-01-08
termenFinal	2013-02-28

---

denumire	Proiect eTerra
descriere	Sistem national pentru evidenta proprietatilor
termenStart	2013-04-13
termenFinal	2013-05-14

---

denumire	Proiect SINCRON
descriere	Sistem National de Management al Ariilor Protejate
termenStart	2012-03-05
termenFinal	2012-11-18

---

denumire	Proiect SAR
descriere	Sistem de cautare si salvare
termenStart	2011-01-11
termenFinal	2011-11-22

---

✓ Returned 5 rows in 3209 ms

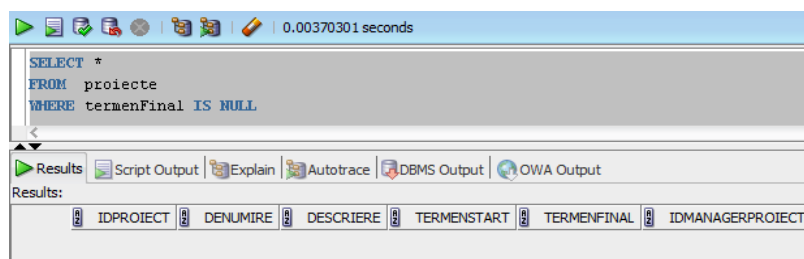
**Fig.49 Operatorul IN în Neo4j**

## 6. IS NULL vs. IS NULL

În cazul în care dorim filtrarea valorilor nule ale unuia sau mai multor câmpuri, vom utiliza operatorul IS NULL.

Practic, afișarea proiectelor care nu au un termen final, se va rezuma la:

```
SELECT *
FROM proiecte
WHERE termenFinal IS NULL
```



**Fig.50 Operatorul IS NULL în SQL (Oracle)**

Rezultatul ne demonstrează că nu avem proiecte fără descriere.

În Neo4j folosim același operator și obținem același rezultat:

```
MATCH (p:Proiect)
WHERE p.termenFinal IS NULL
RETURN p
```

```
1 MATCH (p:Proiect)
2 WHERE p.termenFinal IS NULL
3 RETURN p
```

CYPHER MATCH (p:Proiect) WHERE p.termenFinal IS NULL RETURN p

✓ Returned 0 rows in 224 ms

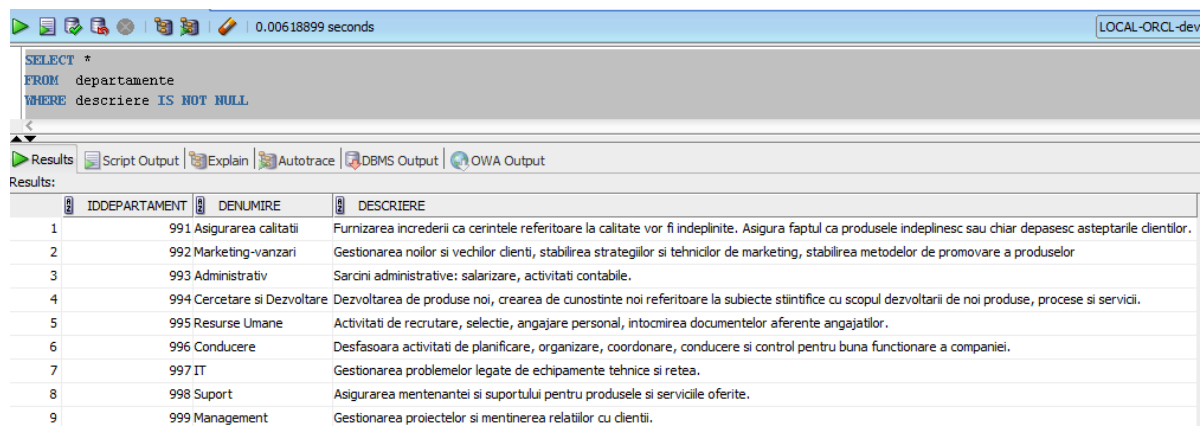
**Fig.51 Operatorul IS NULL în Neo4j**

## 9. IS NOT NULL vs. IS NOT NULL

Opusul operatorului prezentat anterior, IS NOT NULL, va avea ca finalitate afișarea valorilor ne-nule pentru un anumit câmp.

Să presupunem că dorim să aflăm care sunt departamentele care au o descriere:

```
SELECT *
FROM departamente
WHERE descriere IS NOT NULL
```



The screenshot shows the Neo4j interface with a Cypher query executed. The results are displayed in a table with 9 rows. The columns are IDDEPARTAMENT, DENUMIRE, and DESCRIERE. The data is as follows:

IDDEPARTAMENT	DENUMIRE	DESCRIERE
1	991 Asigurarea calitatii	Furnizarea increderii ca cerintele referitoare la calitate vor fi indeplinite. Asigura faptul ca produsele indeplinesc sau chiar depasesc asteptarile clientilor.
2	992 Marketing-vanzari	Gestionarea noilor si vechilor clienti, stabilirea strategiilor si tehnicilor de marketing, stabilirea metodelor de promovare a produselor
3	993 Administrativ	Sarcini administrative: salarizare, activitati contabile.
4	994 Cercetare si Dezvoltare	Dezvoltarea de produse noi, crearea de cunostinte noi referitoare la subiecte stiintifice cu scopul dezvoltarii de noi produse, procese si servicii.
5	995 Resurse Umane	Activitati de recrutare, selectie, angajare personal, intocmirea documentelor aferente angajatilor.
6	996 Conducere	Desfasoara activitati de planificare, organizare, coordonare, conducere si control pentru buna functionare a companiei.
7	997 IT	Gestionarea problemelor legate de echipamente tehnice si retea.
8	998 Suport	Asigurarea mentenantei si suportului pentru produsele si serviciile oferite.
9	999 Management	Gestionarea proiectelor si mentinerea relatiilor cu clientii.

**Fig.52 Operatorul IS NOT NULL în Neo4j**

În mod similar, în Neo4j:

```
MATCH (d:Departament)
WHERE d.descriere IS NOT NULL
RETURN d
```

```
1 MATCH (d:Departament)
2 WHERE d.descriere IS NOT NULL
3 RETURN d
```

The screenshot shows the Neo4j Cypher query editor with the query: `MATCH (d:Departament) WHERE d.descriere IS NOT NULL RETURN d`. Below the query, the results are displayed in a table format. The table has two columns: `d` (the variable name) and the details of the department. The results show two departments: 'Asigurarea calitatii' and 'Marketing-vanzari'.

d				
<table border="1"><tr><td>denumire</td><td>Asigurarea calitatii</td></tr><tr><td>descriere</td><td>Departament concentrat pe furnizarea increderii ca cerintele referitoare la calitate vor fi indeplinite. Asigura faptul ca produsele indeplinesc sau chiar depasesc asteptarile clientilor.</td></tr></table>	denumire	Asigurarea calitatii	descriere	Departament concentrat pe furnizarea increderii ca cerintele referitoare la calitate vor fi indeplinite. Asigura faptul ca produsele indeplinesc sau chiar depasesc asteptarile clientilor.
denumire	Asigurarea calitatii			
descriere	Departament concentrat pe furnizarea increderii ca cerintele referitoare la calitate vor fi indeplinite. Asigura faptul ca produsele indeplinesc sau chiar depasesc asteptarile clientilor.			
<table border="1"><tr><td>denumire</td><td>Marketing-vanzari</td></tr><tr><td>descriere</td><td>Atributiile principale: gestionarea noilor si vechilor clienti, stabilirea startegiilor si tehnicilor de marketing, stabilirea metodelor de promovare a produselor.</td></tr></table>	denumire	Marketing-vanzari	descriere	Atributiile principale: gestionarea noilor si vechilor clienti, stabilirea startegiilor si tehnicilor de marketing, stabilirea metodelor de promovare a produselor.
denumire	Marketing-vanzari			
descriere	Atributiile principale: gestionarea noilor si vechilor clienti, stabilirea startegiilor si tehnicilor de marketing, stabilirea metodelor de promovare a produselor.			

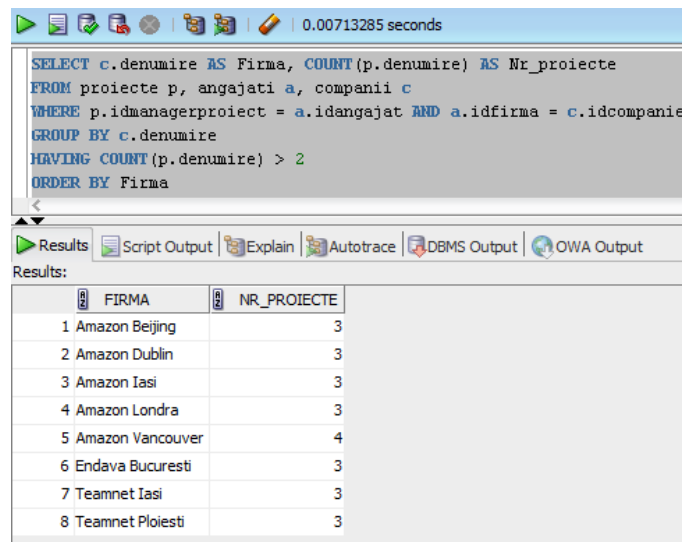
Returned 9 rows in 272 ms

*Fig.53 Operatorul IS NOT NULL în Neo4j*

## 10. HAVING vs. WITH

Clauza HAVING, respectiv WITH, filtrează rezultatul obținut în urma unei grupări. Dorim să aflăm care sunt filialele Amazon care “dețin” mai mult de 2 proiecte:

```
SELECT c.denumire AS Firma, COUNT(p.denumire) AS Nr_proiecte
FROM proiecte p, angajati a, companii c
WHERE p.idmanagerproiect = a.idangajat AND a.idfirma = c.idcompanie
GROUP BY c.denumire
HAVING COUNT(p.denumire) > 2
ORDER BY Firma
```



```

SELECT c.denumire AS Firma, COUNT(p.denumire) AS Nr_proiecte
FROM proiecte p, angajati a, companii c
WHERE p.idmanagerproiect = a.idangajat AND a.idfirma = c.idcompanie
GROUP BY c.denumire
HAVING COUNT(p.denumire) > 2
ORDER BY Firma

```

FIRMA	NR_PROIECTE
1 Amazon Beijing	3
2 Amazon Dublin	3
3 Amazon Iasi	3
4 Amazon Londra	3
5 Amazon Vancouver	4
6 Endava Bucuresti	3
7 Teamnet Iasi	3
8 Teamnet Ploiesti	3

**Fig.54 Clauza HAVING în SQL (Oracle)**

În mod similar, în Neo4j, folosind clauza WITH:

```

MATCH (a:Angajat)-[LUCREAZA_LA]->(f:Filiala), (a:Angajat)-[GESTIONEAZA]->(p:Proiect)
WITH f.denumire AS Firma , COUNT(p) AS Nr_proiecte
WHERE Nr_proiecte > 2
RETURN Firma, Nr_proiecte
ORDER BY Firma

```

```

1 MATCH (a:Angajat)-[LUCREAZA_LA]->(f:Filiala), (a:Angajat)-[GESTIONEAZA]->(p:Proiect)
2 WITH f.denumire AS Firma , COUNT(p) AS Nr_proiecte
3 WHERE Nr_proiecte > 2
4 RETURN Firma, Nr_proiecte
5 ORDER BY Firma

```

Firma	Nr_proiecte
Amazon Beijing	3
Amazon Dublin	3
Amazon Iasi	3
Amazon Londra	3
Amazon Vancouver	4
Endava Bucuresti	3
Teamnet Iasi	3
Teamnet Ploiesti	3

✓ Returned 8 rows in 382 ms

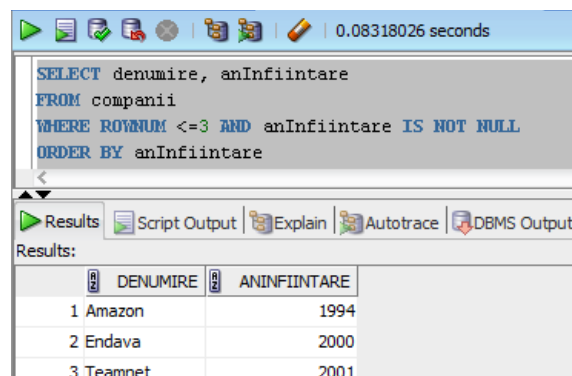
**Fig.55 Clauza WITH în Neo4j**

## 11. ROWNUM vs. LIMIT

Există situații în care dorim limitarea numărului de linii dintr-o interogare. În acest caz, avem șansa de a folosi clauza ROWNUM în Oracle, respectiv LIMIT în Neo4j.

În Oracle, cele mai vechi 3 companii:

```
SELECT denumire, anInfiintare
FROM companii
WHERE ROWNUM <=3 AND anInfiintare IS NOT NULL
ORDER BY anInfiintare
```



0.08318026 seconds

```
SELECT denumire, anInfiintare
FROM companii
WHERE ROWNUM <=3 AND anInfiintare IS NOT NULL
ORDER BY anInfiintare
```

Results:

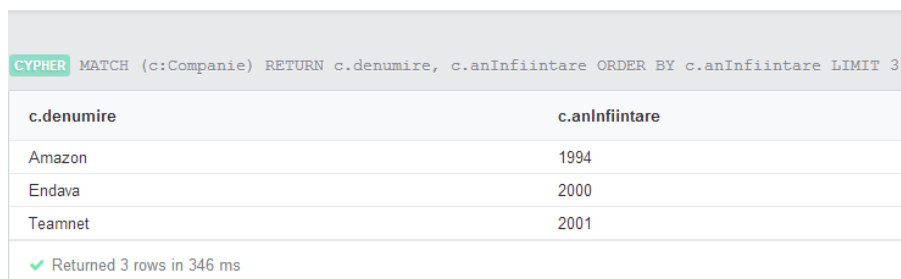
	DENUMIRE	ANINFIINTARE
1	Amazon	1994
2	Endava	2000
3	Teamnet	2001

*Fig.56 Clauza ROWNUM în SQL (Oracle)*

Acceași cerință în Neo4j:

```
MATCH (c:Companie)
RETURN c.denumire, c.anInfiintare
ORDER BY c.anInfiintare
LIMIT 3
```

```
1 MATCH (c:Companie)
2 RETURN c.denumire, c.anInfiintare
3 ORDER BY c.anInfiintare
4 LIMIT 3
```



CYPHER MATCH (c:Companie) RETURN c.denumire, c.anInfiintare ORDER BY c.anInfiintare LIMIT 3

c.denumire	c.anInfiintare
Amazon	1994
Endava	2000
Teamnet	2001

✓ Returned 3 rows in 346 ms

*Fig.57 Clauza LIMIT în Neo4j*



## 2.8 Constrângeri (restricții) în Oracle și Neo4j

În Oracle restricțiile de integritate se pot încadra în următoarele tipuri: NOT NULL, UNIQUE, PRIMARY KEY, FOREIGN KEY, CHECK și pot fi definite în momentul creării tabelor. Restricția de mai jos impune introducerea uneia din cele două valori: 'M' (pentru angajații de gen masculin) sau 'F' (pentru angajații de gen feminin):

```
CREATE TABLE angajati (  
  idAngajat NUMBER(3) NOT NULL PRIMARY KEY,  
  nume VARCHAR2(50) NOT NULL,  
  dataNasterii DATE,  
  gen CHAR(1) CHECK (gen IN ('M', 'F')),  
  nationalitate VARCHAR2(20),  
  dataAngajarii DATE,  
  idJob NUMBER(3),  
  FOREIGN KEY (idJob)  
  REFERENCES posturi(idPost),  
  idFirma NUMBER(3),  
  FOREIGN KEY (idFirma)  
  REFERENCES companii(idCompanie),  
  idSef NUMBER(3),  
  FOREIGN KEY (idSef)  
  REFERENCES angajati(idAngajat)  
);
```

În Neo4j nu avem chei primare sau străine, însă beneficiem de constrângeri precum UNIQUE, ca în exemplul de mai jos:

```
CREATE CONSTRAINT ON (c:Companie) ASSERT c.denumire IS UNIQUE  
DROP CONSTRAINT ON (c:Companie) ASSERT c.denumire IS UNIQUE
```

## 2.9 Elemente distincte în Neo4j

### 1. FOREACH

Asemenea FOR-ului din SQL (Oracle), în Neo4j putem folosi FOREACH. Spre exemplu, un Project Director supervizează mai mulți Project Manageri. Pentru crearea acestor relații de subordonare putem folosi clauza FOREACH:

```
MATCH (s:Angajat), (a:Angajat)-[:LUCREAZA_LA]-(f:Filiala)  
WHERE s.nume = 'Francusi Carmen' AND a.post = 'Project Manager' AND  
f.denumire = 'Synygy Iasi'  
WITH s, COLLECT(a) AS angajati
```

```
FOREACH(i IN RANGE(0, length(angajati)-1) |
FOREACH(ai IN [angajati[i]] |
  CREATE (s)-[:SUPERVIZEAZA_PE]->(ai)))
```

## 2. Parametri

Parametrii pot fi folosiți pentru expresiile (șiruri de caractere, expresii regulate) din clauza WHERE sau START și sunt recomandați pentru optimizarea interogărilor.

Spre exemplu, putea crea noduri cu proprietățile definite în cadrul parametrului “props”:

```
FOREACH (props IN [{ nume:"Vasilescu Daniel", nationalitate:"romana" },
{ nume:"Murs Oliver", nationalitate:"britanica" }]|
  CREATE (a:Angajat{
    nume:props.nume,nationalitate:props.nationalitate })))
```

După creare, putem afișa nodurile folosind:

```
MATCH (a:Angajat) WHERE a.nume IN ["Vasilescu Daniel", "Murs Oliver"]
RETURN a
```



**Fig.58 Parametri în Neo4j**

- 3. Funcții scalare** care returnează tipul relației (TYPE), nodul de start (STARTNODE), nodul final (ENDNODE), primul (HEAD) sau ultimul (LAST) element dintr-o colecție etc.
- 4. Funcția HAS( ),** care primește ca parametru o proprietate a nodului și returnează True dacă nodul conține proprietatea specificată.

```

1 MATCH a
2 WHERE HAS (a.nume) AND a.nume = 'Bezos Jeffrey'
3 RETURN a

```

**CYPHER** MATCH a WHERE HAS (a.nume) AND a.nume = 'Bezos Jeffrey' RETURN a

**a**

post	CEO
data_nasterii	1964-01-12
nume	Bezos Jeffrey
nationalitate	americana
data_angajarii	1994-07-10
gen	M

***Fig.59 Funcția HAS( ) în Neo4j***

## Capitolul 3 - Utilizarea Apache Jmeter în analiza performanței Oracle server vs. Neo4j server

Performanța unei baze de date depinde de mai mulți factori, însă o analiză detaliată ne confirmă sau nu faptul că un anumit tip de bază de date este potrivit nevoilor noastre.

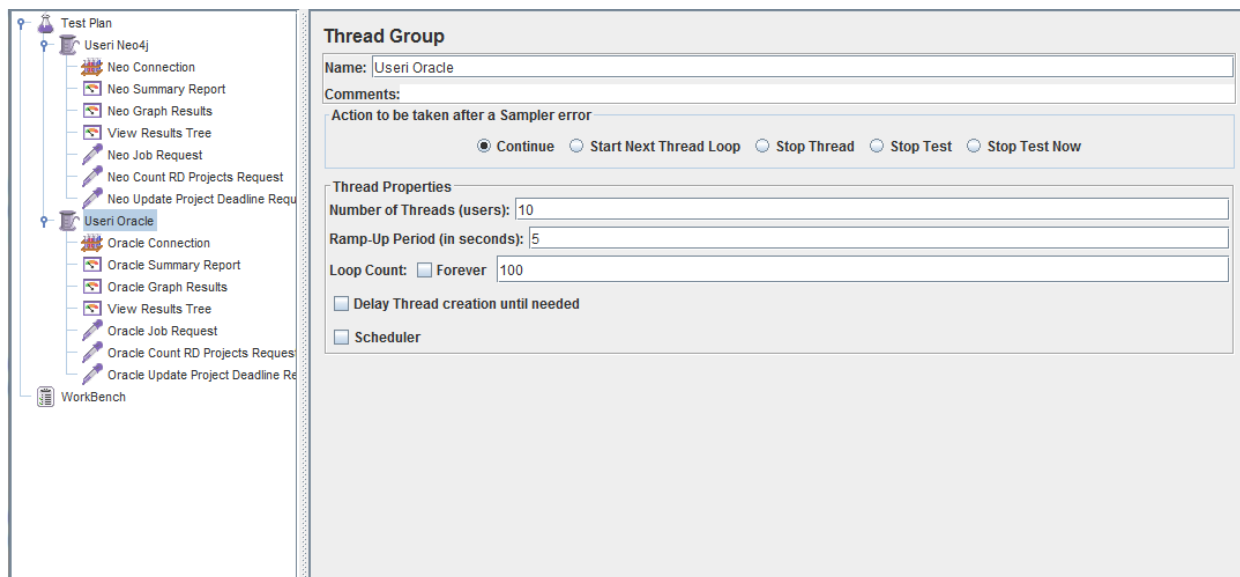
Apache JMeter este un produs destinat testării comportamentului și măsurării performanței unei baze de date. Pentru o analiză mai bună, am ales încărcarea unui set mare de date cu privire la companii, angajați, departamente, posturi, proiecte. Datele au fost stocate atât în serverul Oracle, cât și în Neo4j, pentru realizarea ulterioară a unei analize comparative a timpilor de răspuns și latenței în cele două servere.

Timpul de răspuns reprezintă de fapt perioada de timp dintre cererea trimisă și ultimul răspuns primit, iar latența se referă la timpul scurs între trimiterea cererii și sosirea primului răspuns. Interogările folosite în continuare sunt din categoria celor simple, așadar trebuie ținut cont că analiza nu se axează pe interogări recursive.

Pentru construirea unui plan de test, pașii parcurși în JMeter au constat în:

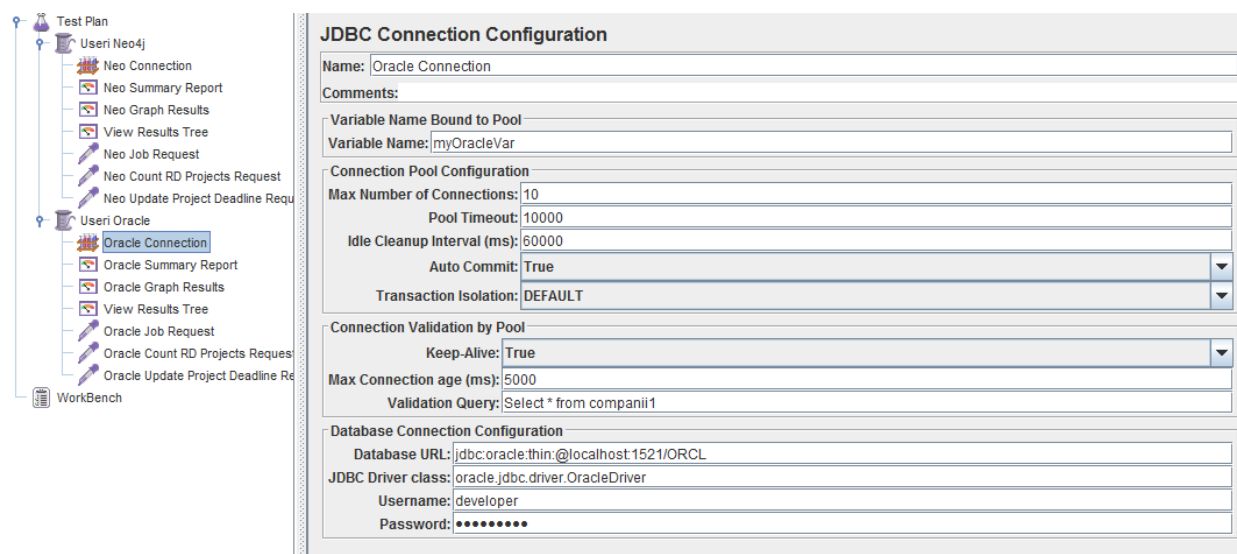
1) Definirea a două Thread Group-uri care conțin ca și elemente: numărul de utilizatori care vor trimite cereri, perioada de timp (în secunde) dintre cererile fiecărui utilizator și, nu în ultimul rând stabilirea numărului de solicitări trimise.

Atât în Oracle (Fig. 60), cât și în Neo4j am ales ca 10 utilizatori să trimită 100 de cereri la un interval de 5 secunde unul față de altul.

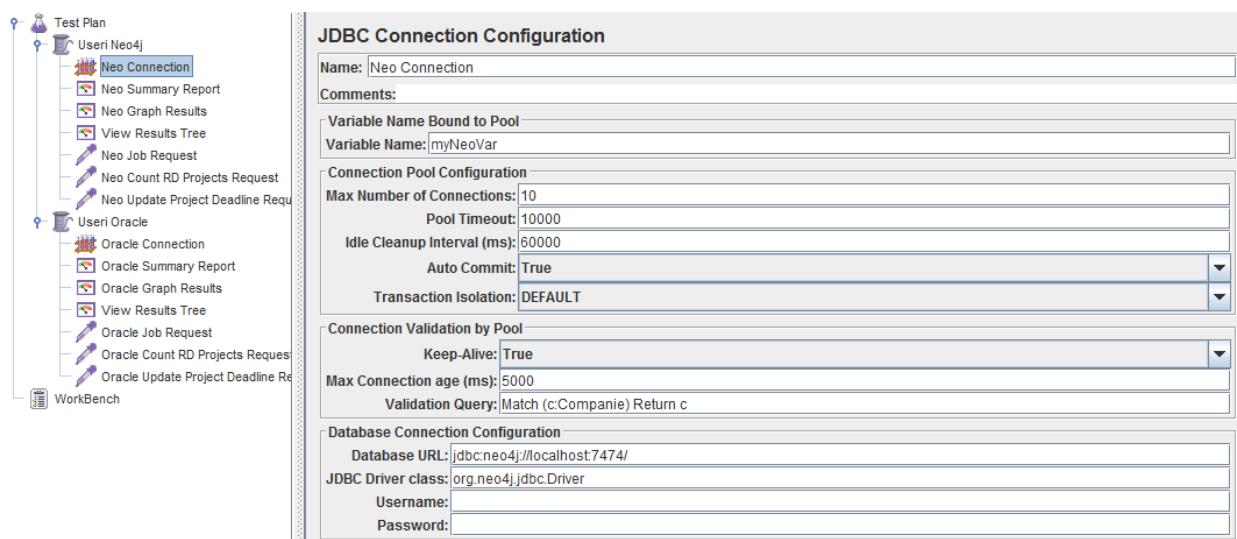


**Fig.60 Definirea unui Thread Group în Apache JMeter**

2) Definirea conexiunilor JDBC la bazele noastre de date din Oracle (Fig. 61), respectiv Neo4j (Fig. 62). Pentru conectare am folosit drivere JDBC specifice fiecărui tip de server în parte.



**Fig. 61 Definirea conexiunii la serverul Oracle**



**Fig. 62 Definirea conexiunii la serverul Neo4j**

3) Adăugarea cererilor JDBC, ce pot fi de tip SELECT, UPDATE, COMMIT, ROLLBACK ș.a.. Am ales definirea a trei cereri: două de tip SELECT și una de tip UPDATE.

Prima cerere a presupus obținerea angajaților de la Macromedia care au o funcție de “Tester”.

În Oracle:

```
SELECT a.nume AS Nume_Angajat, a.prenume AS Prenume_Angajat
FROM angajati1 a, companii1 c, posturi1 p
WHERE a.idfirma = c.id AND a.idjob = p.id
```

```
AND p.denumire = 'Tester'  
AND c.denumire = 'Macromedia'
```

În Neo4j:

```
MATCH (a:Angajat)-[:LUCREAZA_LA]->(c:Companie)  
WHERE a.post = 'Tester' AND c.denumire = 'Macromedia'  
RETURN a.ume AS Nume_Angajat, a.prenume AS Prenume_Angajat
```

Prin a doua cerere se dorește afișarea numărului de proiecte din cadrul departamentului de “Cercetare și dezvoltare”.

În Oracle:

```
SELECT COUNT(p.id) AS Nr_proiecte_RD  
FROM proiecte1 p, departamente1 d  
WHERE p.iddepart = d.id AND d.denumire = 'Research and  
Development'
```

În Neo4j:

```
MATCH (d:Departament{denumire:'Research and Development'})-  
[:GESTIONEAZA]->(p:Proiect)  
RETURN COUNT(p) AS Nr_Proiecte_RD
```

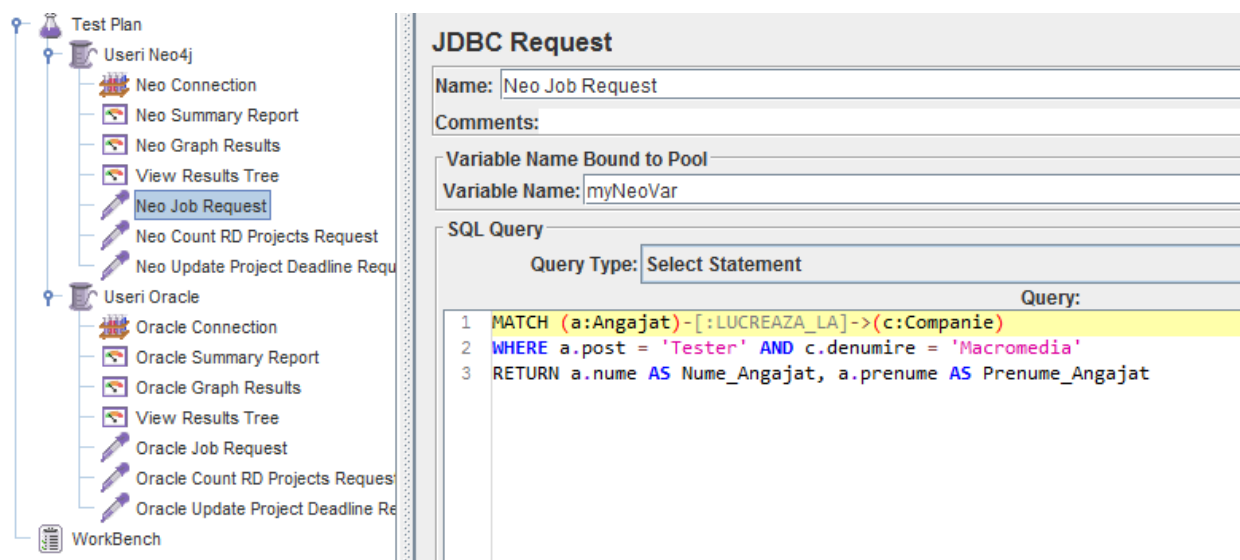
Prin cererea de UPDATE am dorit să modificăm termenul final pentru proiectul “Lovaza”.

În Oracle:

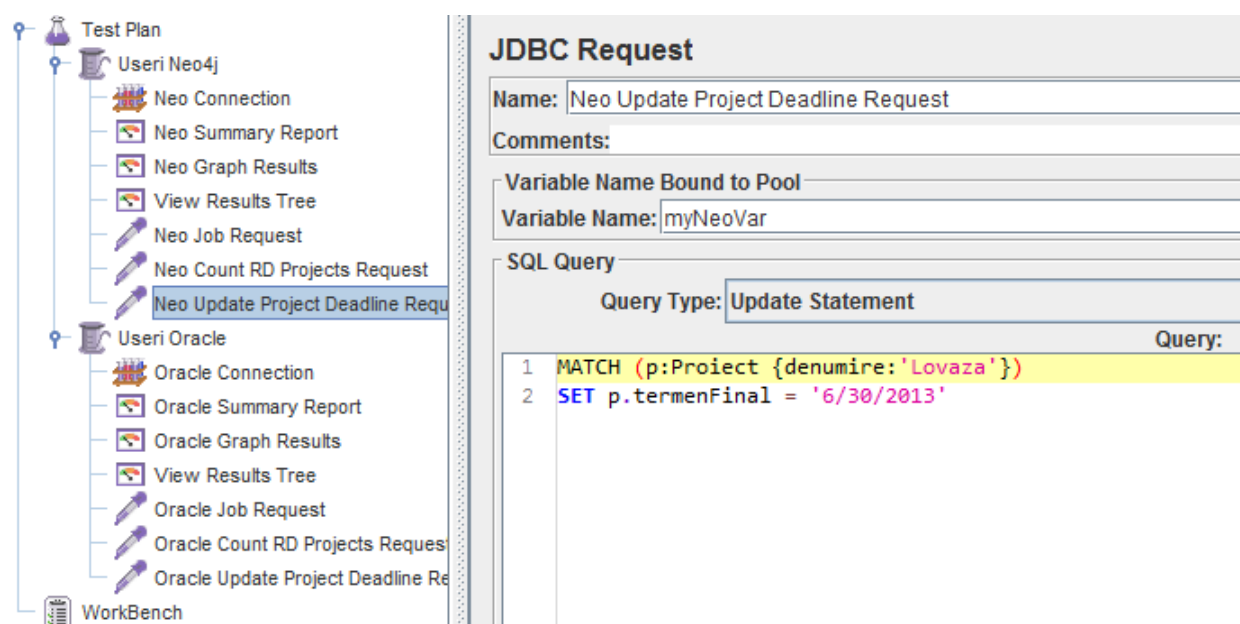
```
UPDATE proiecte1  
SET termenFinal = TO_DATE('2013/06/30', 'yyyy/mm/dd')  
WHERE denumire = 'Lovaza'
```

În Neo4j:

```
MATCH (p:Proiect {denumire:'Lovaza'})  
SET p.termenFinal = '6/30/2013'
```

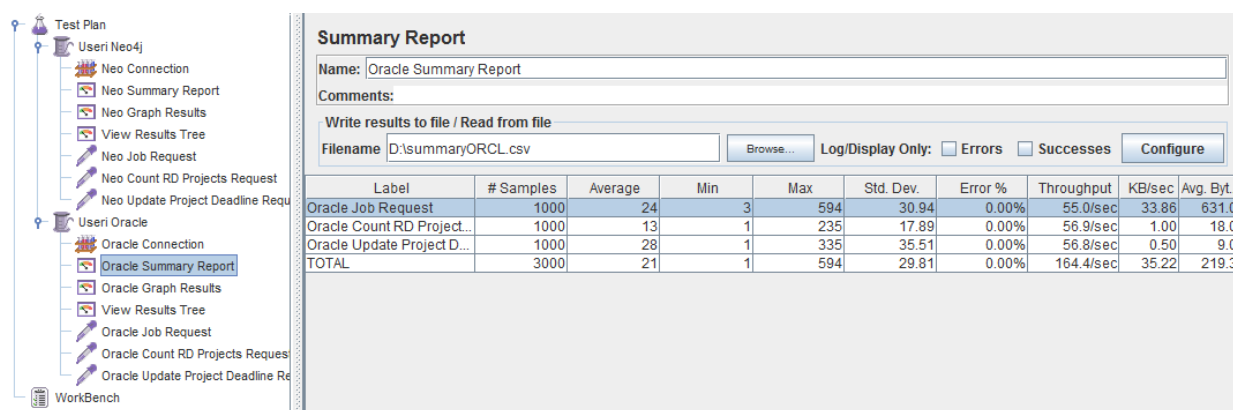


**Fig. 63** Definirea unei cereri de tip **SELECT** în Neo4j

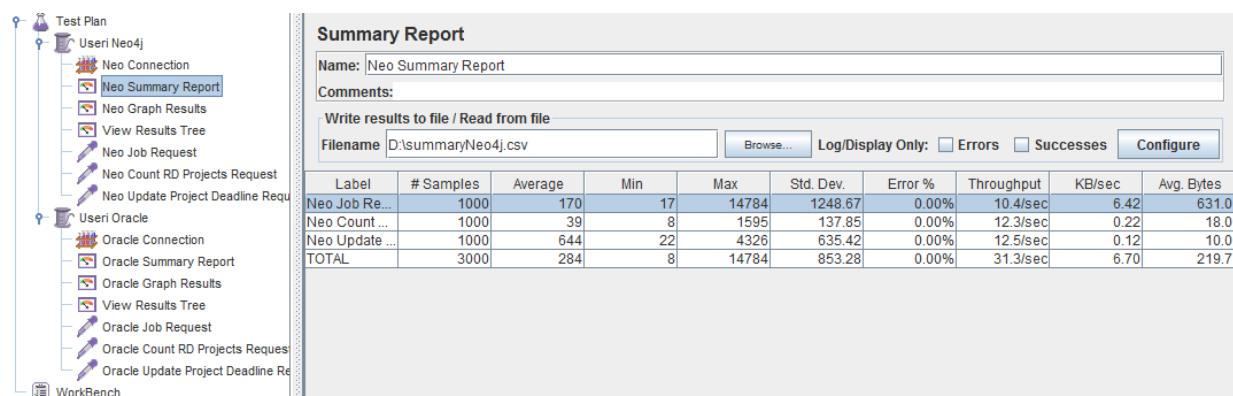


**Fig. 64** Definirea unei cereri de tip **UPDATE** în Neo4j

4) Pentru vizualizarea rezultatelor obținute am ales definirea unui Listener de tip Summary Report (Fig. 65 și Fig. 66).



**Fig. 65 Summary Report pentru testul Oracle**



**Fig. 66 Summary Report pentru testul Neo4j**

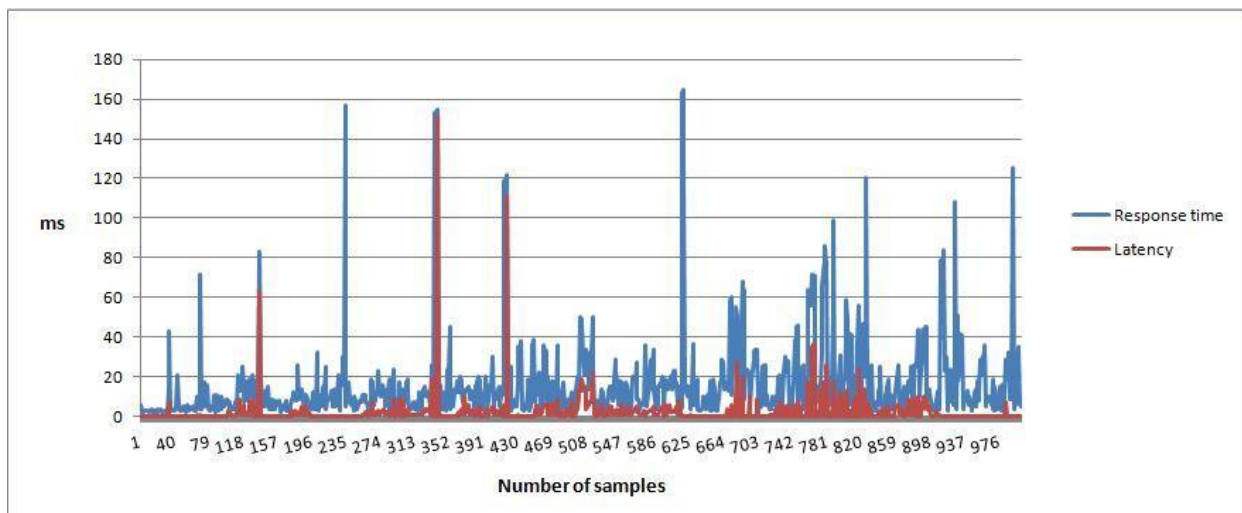
Încă din interfață ne putem da seama că Oracle reușește satisfacerea a peste 50 de cereri/secundă, pe când Neo4j abia atinge un număr de 12 cereri/secundă (câmpul Throughput). Chiar și la timpul minim, respectiv maxim petrecut pentru fiecare grup de cereri în parte, se pare că Oracle câștigă lupta în fața Neo4j (câmpurile Min, Max).

E necesar să precizăm că putem scrie rezultatele detaliate într-un fișier .csv, în care putem vizualiza elemente precum: timeStamp, elapsed, responseMessage, threadName, success, failureMessage, bytes, allThreads, Latency ș.a.

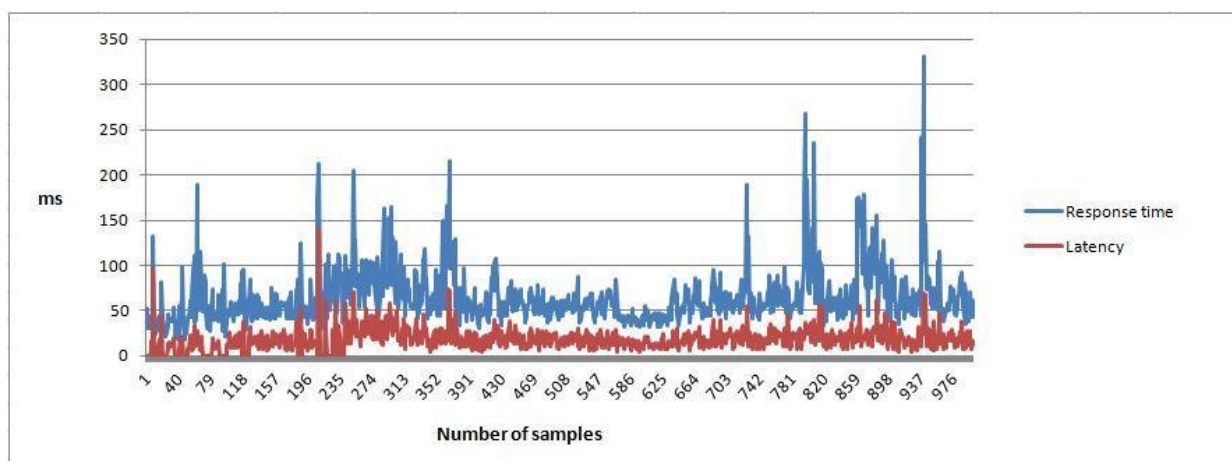
Pe baza rezultatelor detaliate am realizat grafice de tip Line Chart pentru fiecare tip de cerere în parte.

Prima cerere de tip SELECT, cea referitoare la afișarea angajaților care ocupă o poziție de “Tester” la compania Macromedia, s-a sintetizat în graficele de mai jos.





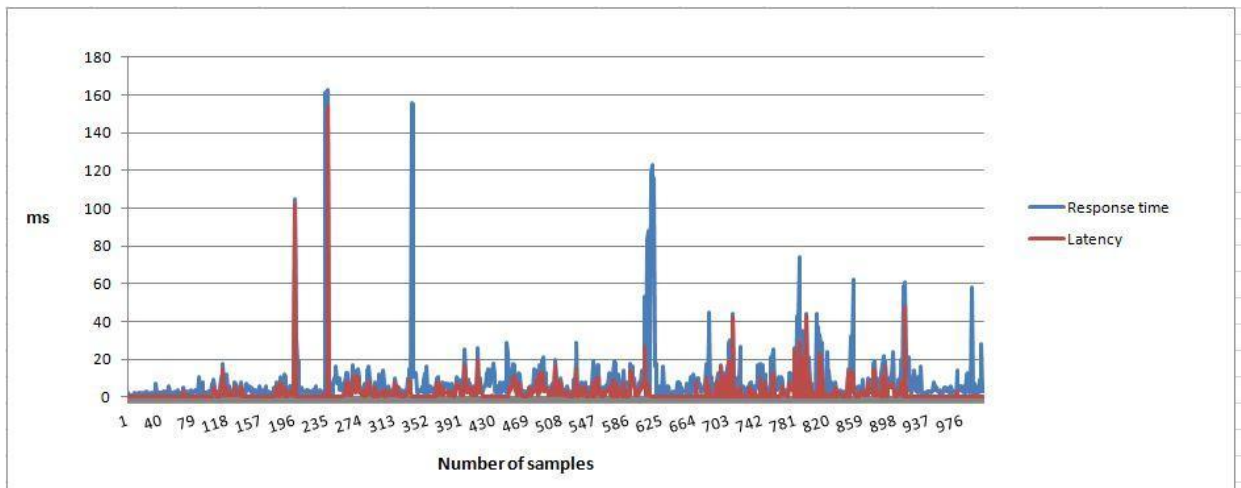
**Fig. 67 Prima cerere SELECT în Oracle**



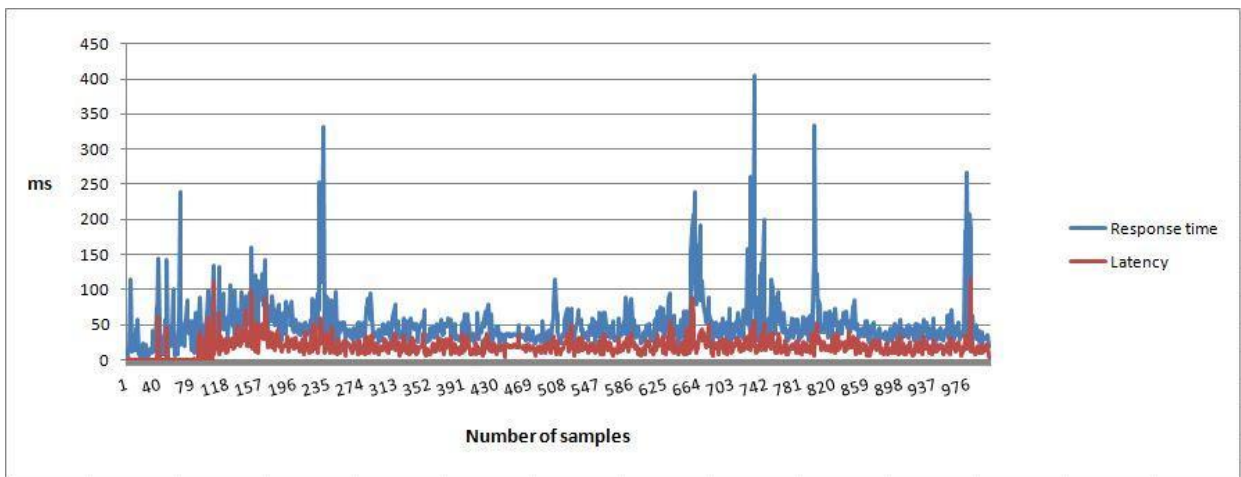
**Fig. 68 Prima cerere SELECT în Neo4j**

Concluzia e simplă: în Oracle timpul de răspuns se încadrează majoritar până în 40 ms, pe când în Neo4j este depășită cu mult această limită (marea majoritate a sample-urilor răspund în peste 50 ms). Latența e simțitor mai mare la serverul Neo4j, ceea ce ne dă de înțeles că din nou Oracle aduce un avantaj în plus.

Cea de-a doua cerere de tip SELECT, prin care se solicită numărul de proiecte din cadrul departamentului R&D, poate fi analizată urmărind graficele de mai jos.



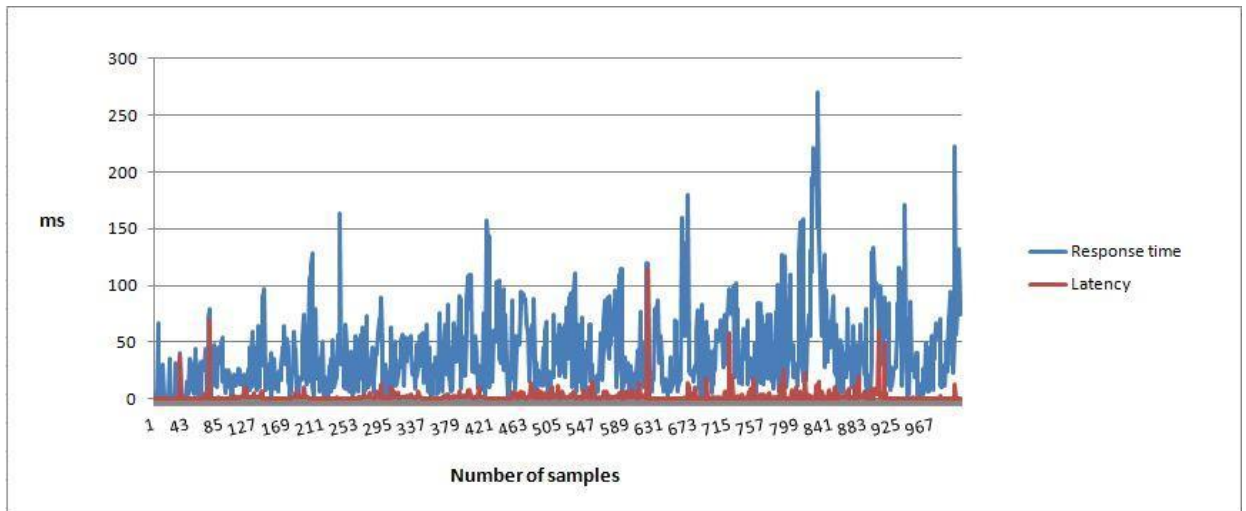
**Fig. 69** A doua cerere SELECT în Oracle



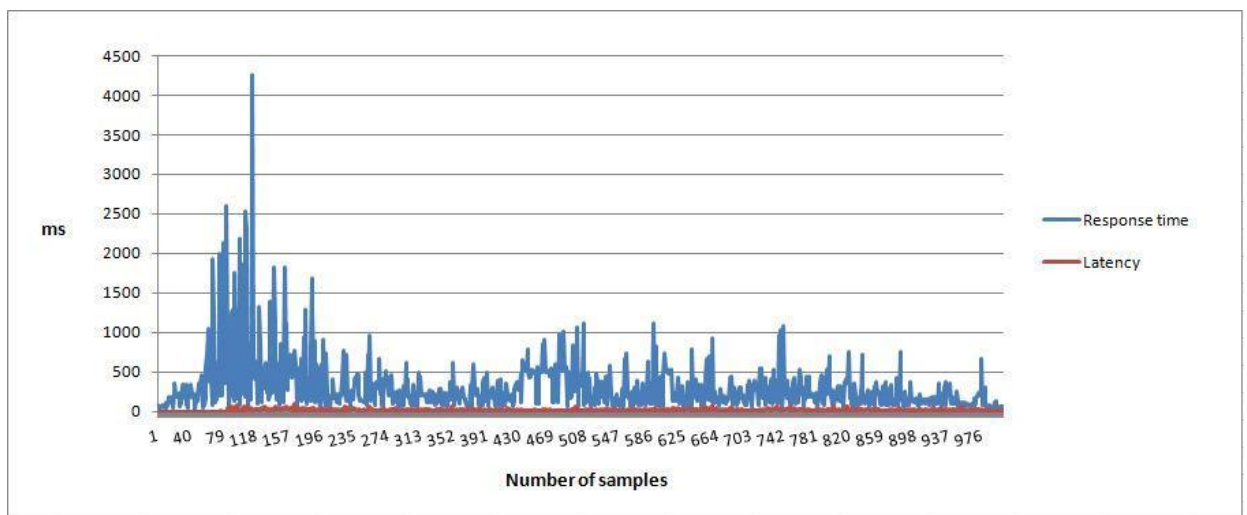
**Fig. 70** A doua cerere SELECT în Neo4j

Concluzionând, Oracle câștigă din nou prin timp de răspuns destul de redus, însă observăm o latență destul de persistentă până aproape de ultimele sample-uri procesate. Prezentă și la serverul Neo4j, latența pare să fie simțitor mai redusă decât timpul de răspuns, însă rămâne în continuare în parametri destul de ridicați, față de cei obținuți în cazul serverului de baze de date relaționale (peste 20 ms atinse majoritar în Oracle).

Ultima cerere, cea de tip UPDATE, prin care solicităm modificarea datei finale pentru proiectul “Lovaza”, e sintetizată în cele două grafice de mai jos.

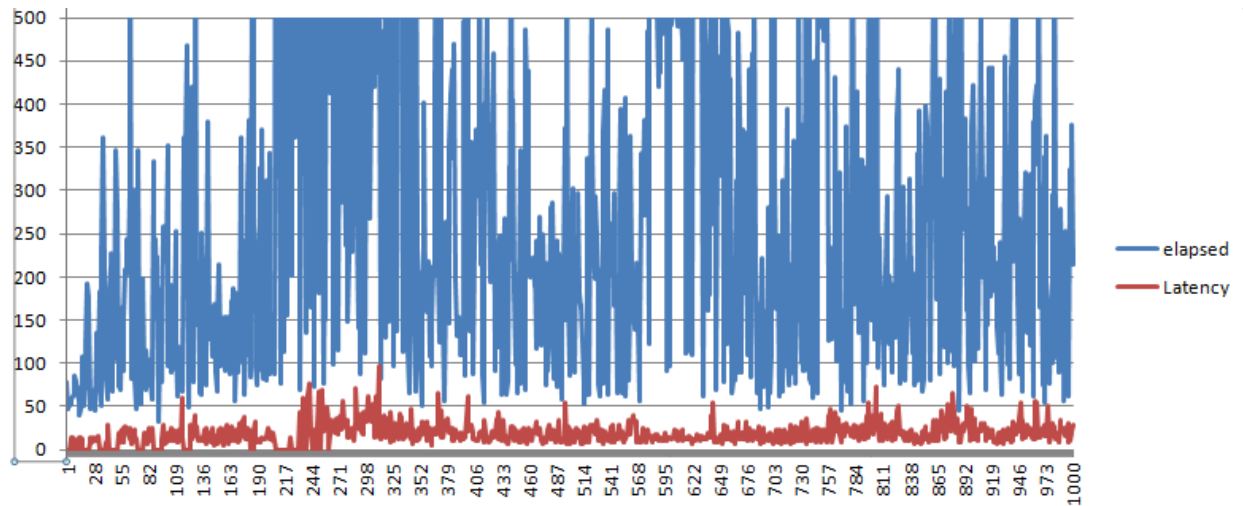


**Fig. 71 Cererea UPDATE în Oracle**



**Fig. 72 Cererea UPDATE în Neo4j**

Dacă în cazul cererilor SELECT, latența era destul de evidentă, se pare că la UPDATE este relativ redusă, însă timpul de răspuns a crescut semnificativ. Per ansamblu, valorile timpului de răspuns din Oracle se încadrează până în 100 ms (exceptând câteva teste care depășesc acest prag), însă în Neo4j sunt semnificativ mai mari (marea majoritate până în 500 ms), iar în ceea ce privește latența, Neo4j pierde din nou teren, atingând valori foarte apropiate de 50 ms (Fig.73), valoare foarte rar întâlnită în Oracle.



***Fig. 73 Cererea UPDATE în Neo4j (la scară largă)***

Așadar, putem afirma că Oracle încă își menține titlatura în materie de timp de răspuns scurt și latență redusă, însă nu putem infirma faptul că cei de la Neo4j nu vor face tot posibilul să demonstreze că non-relaționalul va câștiga bila albă la acest capitol și că va reuși să acopere punctele slabe de care încă mai dă dovadă.

## Concluzii

Viitorul este cert și destul de previzibil. Volumul de date are trend ascendent, tehnologiile nu întârzie să vină în ajutorul manipulării acestor bunuri strategice, care se alătură informațiilor și cunoștințelor. Fie că sunt relaționale sau nu, bazele de date continuă să stocheze volume impresionante de date și reușesc să uimească prin viteza, precizia și cantitatea rezultatelor pe care le returnează. Cum fiecare produs tehnologic este folosit cu un scop și se ghidează după anumite cerințe, și bazele de date relaționale vor fi alese în detrimentul bazelor NoSQL pentru asigurarea anumitor criterii, sau invers.

Faptul că a crescut exponențial volumul de consum și producție de date, fapt datorat în special evoluției tehnologice, nu ne duce cu gândul decât la extinderea spațiului de stocare, la facilitarea manipulării datelor și la realizarea sarcinilor paralele cu scop de optimizare a timpului. NoSQL tinde să ne asigure că aceste puncte pot fi îndeplinite cu ușurință, odată cu alegerea unui tip de baze de date din această categorie.

Ca administrator de sistem sau ca manager e uneori dificil să compari diferitele tipuri de instrumente NoSQL. În primul rând trebuie să iei în considerare nevoile informaționale, să le potrivești cu ceea ce piața îți oferă, să alegi ce e bun pentru organizație și să iei o decizie finală. NoSQL a fost proiectat pentru data store-uri distribuite, pentru nevoia de stocare a unor volumuri imense de date (de exemplu, Facebook și Twitter acumulează zilnic TB de date pentru milioane de utilizatori), pentru ușurinta modificărilor (flexibilitatea schemei). NoSQL are ca avantaj ceea ce întâlnim sub denumirea de “scalabilitate orizontală” - împărțirea sarcinii între mai multe sisteme.

Un punct forte al bazelor de date graf este existența relațiilor. Comparativ cu JOIN-urile “obositoare” din Oracle, relațiile simplifică lucrurile în Neo4j. Să presupunem că avem în Oracle două tabele cu o relație de tip Many-to-Many, care ne va cere implicit crearea unei tabele suplimentare, deținătoare a celor două id-uri din tabelele “părinte”. În Neo4j structura va fi simplificată prin prezența a două noduri și a relației dintre ele. Dacă relaționarea celor trei tabele din Oracle ar necesita folosirea a două clauze JOIN și identificarea câmpurilor de legătură, în Neo4j e de ajuns să specificăm în clauza MATCH cele două noduri și relația dintre ele. Dacă am compara din punct de vedere sintactic, în multe cazuri Cypher câștigă bătălia în fața SQL-ului, luând în considerare numărul de linii de cod și gradul de dificultate în a scrie o interogare. În cele mai multe cazuri Cypher implică mai puțin cod și mai multă intuiție.

În funcție de necesitățile noastre, o schemă fixă poate fi un avantaj și chiar o cerință obligatorie, dar poate fi un inconvenient dacă dorința noastră este de a manipula date nestructurate, structuri “flexibile”. Spre exemplu, pentru un site de comerț online este strict necesară popularea câmpului “Telefon” din baza de date. Pentru o bază de date destinată gestiunii angajaților dintr-o unitate, e posibil ca unele persoane să aibă definită proprietatea “Telefon” și valori pentru dânsa sau e posibil să lipsească cu desăvârșire. Acesta poate fi un caz tipic bazelor de date graf.

Să nu uităm și de punctele mai puțin bune de care dă dovadă Neo4j. Faptul că nu există o schemă, că datele introduse sunt nestructurate, că avem șansa să inserăm ”orice” în baza de date, ne poate costa mult în momentul interogării datelor. Practic, dacă interogarea noastră nu este corectă sintactic, există posibilitatea returnării unei erori (uneori destul de

clară, alteori mai puțin concluzivă), dar există și posibilitatea returnării unui set gol de date, nespecificându-se faptul că s-a strecurat o greșeală la tipărirea interogării.

Dacă ne interesează în mod special timpul de execuție al unei interogări, inclusiv latența, analiza de mai sus ne demonstrează că, per ansamblu, Oracle încă își menține titlul în ceea ce privește rapiditatea în afișarea rezultatelor unor interogări simple. Acest lucru poate fi un criteriu de luat în seamă atunci când dezvoltăm o aplicație web (gen magazin online) care necesită afișarea informațiilor într-un timp cât mai scurt.

Când ne orientăm spre un tip de baze de date, trebuie să stabilim mai întâi natura datelor. Dacă datele au o structură tabulară (în genul unei foi de calcul cu date contabile), atunci modelul relațional ar fi cel mai adecvat. Pe de altă parte, datele geo-spațiale, modelele moleculare, datele din inginerie tind să fie foarte complexe. Pot avea multe niveluri de imbricări și modelul complet de date poate fi complicat. Dacă în trecut, acestea erau stocate în tabele relaționale, acum ele nu se mai potrivesc structurii bidimensionale linie-coloană. De aceea, bazele de date NoSQL pot fi considerate o opțiune, căci imbricarea multi-nivel și ierarhiile sunt ușor reprezentabile în format JSON.

Un alt punct de luat în considerare este volatilitatea modelului de date. Deși nu se poate ști în momentul proiectării dacă modelul de date se va modifica, va evolua sau va rămâne același, este nevoie de puțină flexibilitate. Aici, se pare că bazele de date relaționale pierd puțin teren în fața noilor ”concurente”. Până nu demult, folosirea schemei statice era ceva foarte potrivit pentru modul de construire a unui nucleu de baze de date, însă, cum actualmente schimbările sunt făcute zilnic sau din oră în oră, e necesară adaptarea la o schemă dinamică. Și nu e de mirare că mulți utilizatori NoSQL sunt cei cu afaceri centrate pe web, unde se cere flexibilitate sporită.

Dacă e să ne gândim la conceperea produsului, dezvoltatorul aplicației a devenit cel mai important utilizator, rolul sau nemaifiind clar delimitat de cel al unui administrator de baze de date. Se pune accent pe viteză mare de codare și agilitate sporită în întregul proces de creare a aplicației. Bazele de date NoSQL s-au dovedit a fi soluția potrivită din acest punct de vedere, folosind tehnologii orientate-obiect, precum JSON, de exemplu. Se estimează că programatorii pot construi un prototip în zile sau săptămâni.

Un alt punct despre care ar trebui ținut cont este cel legat de problemele operaționale (scalabilitate, performanță, accesibilitate). Odată cu creșterea volumului de date sau creșterea numărului de utilizatori, o bază de date relațională poate întâmpina probleme serioase de performanță. Recomandarea consultanților, scalarea verticală, va implica un cost prea mare și uneori poate genera blocaje. Multe produse comerciale de baze de date relaționale oferă scalare orizontală, dar această suplimentare poate fi foarte scumpă și complexă. Dacă o organizație se confruntă cu astfel de probleme, poate lua în considerare tehnologiile NoSQL ca o alternativă. Deși sunt construite pentru a găzdui baze de date distribuite pentru sisteme online, de multe ori cedează consistența în favoarea accesibilității. Însă și în acest caz lucrurile rămân discutabile de la caz la caz.

Pe parte de raportare și analiză, Oracle primește un plus. Dacă ne confruntăm cu interogări și analize complexe sau dacă ne axăm pe depozite de date, cu siguranță o alegere bună înseamnă o bază de date relațională. Pe de altă parte, analizele în timp real pentru date operaționale sunt mult mai potrivite pentru NoSQL, mai ales când datele provenite din mai multe sisteme contribuie la crearea unei aplicații. În plus, deși instrumentele BI pentru

NoSQL sunt o noutate, creșterea e destul de rapidă.

Așadar, deși SQL poate urca în top prin faptul că este un limbaj standardizat, în același timp, poate fi defavorizat pentru că nu ajută la stocarea și manipularea datelor nestructurate, cele care dețin un procentaj important în prezent. Pe de altă parte, dacă scopul nostru este consistența datelor, ne vom orienta tot către tre clasicele baze de date relaționale, care întrec NoSQL-ul la acest capitol.

Când vom avea de gestionat date structurate sau vom acorda interes deosebit celor patru puncte esențiale ale modelului ACID ori vom gestiona seturi de date relativ reduse, vom alege cu ușurință bazele de date relaționale, în detrimentul celor NoSQL. Dacă, în schimb, volumul și viteza sunt pe primul plan, la care adaugăm scalabilitate, performanță și downtime minimizat, ne putem îndrepta cu siguranță spre baze de date NoSQL.

Dacă mulți consideră că NoSQL a fost creat pentru a elimina din joc SQL-ul, în realitate el are mai degrabă înțeles de “Not Only SQL” (“Nu doar SQL”) decât “No SQL” (“Fără SQL”). Ambele tehnologii (NoSQL și RDBMS) pot co-exista și fiecare are destinația ei.

Nu vom putea categorisi una din aceste tipuri de baze de date că fiind general valabilă pentru un tip de afacere. Așadar nu ne rămâne decât să analizăm bine cerințele, să punem în balanță plusurile și minusurile oferite de fiecare tip de baze de date în parte și să alegem soluția care ni se potrivește cel mai bine.

## Bibliografie

### Cărți și articole:

1. A., Beaulieu, *Learning SQL*, O'Reilly Media, California, 2009
2. E.F., Codd, *The relational model for database management: version 2*, Addison-Wesley Longman Publishing Co., Boston, 1990
3. C.J., Date, H., Darwen, *Databases, Types And the Relational Model*, Addison-Wesley Longman Publishing Co., Boston, 2006
4. J., Dean, S., Ghemawat, *MapReduce: Simplified Data Processing on Large Clusters*, Sixth Symposium on Operating System Design and Implementation, San Francisco, 2004 (<http://research.google.com/archive/mapreduce.html>; accesat la data de 15.03.2014)
5. L., Douglas, *3D Data Management: Controlling Data Volume, Velocity and Variety*, 2001 (<http://blogs.gartner.com/doug-laney/files/2012/01/ad949-3D-Data-Management-Controlling-Data-Volume-Velocity-and-Variety.pdf>; accesat la data de 17.04.2014)
6. I. Robinson, J., Webber, E., Eifrem, *Graph Databases*, O'Reilly Media, Sebastopol, 2013
7. S., Melnik, A., Gubarev, J.J., Long, G., Romer, S., Shivakumar, M., Tolton, T., Vassilakis, *Dremel: Interactive Analysis of Web-Scale Datasets*, 2010 (<http://static.googleusercontent.com/media/research.google.com/ro//pubs/archive/36632.pdf>; accesat la data de 15.03.2014)
8. C., Olston, B., Reed, U., Srivastava, R., Kumar, A., Tomkins, *ACM SigMod 08, Pig Latin: A Not-So-Foreign Language for Data Processing*, Vancouver, 2008 (<http://infolab.stanford.edu/~olston/publications/sigmod08.pdf>; accesat la data de 10.05.2014)
9. M., Stonebraker, D., Abadi, D.J. Dewitt, S., Madden, E., Paulson, A., Pavlo, A., Rasin, *Communications of the ACM: MapReduce and Parallel DBMSs: Friends or Foes?*, vol.53, nr.1, New York, 2010, (<http://database.cs.brown.edu/papers/stonebraker-cacm2010.pdf>; accesat la data de 15.03.2014)
10. S. Tiwari, *Professional NoSQL*, John Wiley & Sons, Indiana, 2011

### Curs online:

*Introduction to Data Science* (<https://class.coursera.org/datasci-001>)



**Site-uri:**

1. <http://leopard.in.ua/2013/11/08/nosql-world/> (accesat la data de 23.02.2014)
2. [http://en.wikipedia.org/wiki/IBM\\_Information\\_Management\\_System](http://en.wikipedia.org/wiki/IBM_Information_Management_System) (accesat la data de 14.03.2014)
3. <http://blog.knuthaugen.no/2010/03/a-brief-history-of-nosql.html/> (accesat la data de 14.03.2014)
4. <http://www.cs.uni.edu/~okane/source/MUMPS-MDH/> (accesat la data de 14.03.2014)
5. <http://research.google.com/archive/mapreduce.html> (accesat la data de 15.03.2014)
6. [http://www.mckinsey.com/insights/business\\_technology/big\\_data\\_the\\_next\\_frontier\\_for\\_innovation](http://www.mckinsey.com/insights/business_technology/big_data_the_next_frontier_for_innovation) (accesat la data de 22.03.2014)
7. <http://www.forbes.com/sites/gilpress/2013/05/09/a-very-short-history-of-big-data> (accesat la data de 12.04.2014)
8. <http://docs.neo4j.org/chunked/milestone/cypher-query-lang.html> (accesat la data de 12.04.2014)
9. <http://research.google.com/pubs/pub36632.html> (accesat la data de 24.04.2014)
10. <http://neo4j.com/stories/telenor/> (accesat la data de 25.04.2014)
11. <http://www.forbes.com/sites/gartnergroup/2013/03/27/gartners-big-data-definition-consists-of-three-parts-not-to-be-confused-with-three-vs/> (accesat la data de 25.04.2014)
12. <http://www.statisticbrain.com/twitter-statistics/> (accesat la data de 17.05.2014)
13. <http://www.youtube.com/yt/press/statistics.html> (accesat la data de 17.05.2014)
14. <http://lsst.org/lsst/google> (accesat la data de 17.05.2014)
15. <http://traffic.berkeley.edu/> (accesat la data de 17.05.2014)
16. <http://jmeter.apache.org/usermanual/build-db-test-plan.html> (accesat la data de 19.06.2014)
17. [http://docs.neo4j.org/chunked/milestone/cypherdoc-importing-csv-files-with-cypher.html?\\_ga=1.30054519.977901647.1402160895](http://docs.neo4j.org/chunked/milestone/cypherdoc-importing-csv-files-with-cypher.html?_ga=1.30054519.977901647.1402160895) (accesat la data de 19.06.2014)