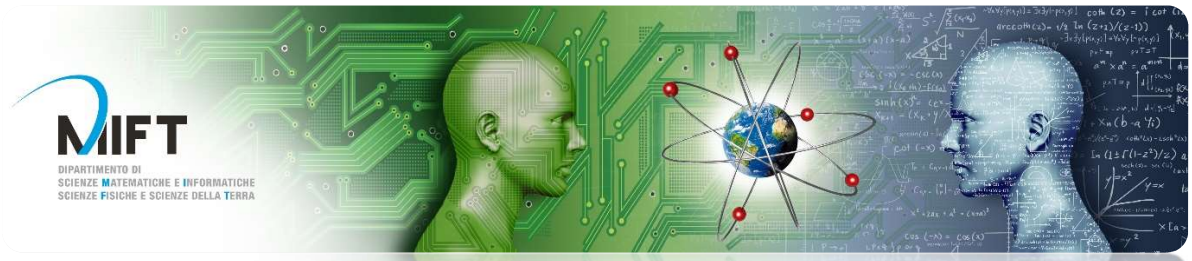


Progetto di basi di dati II.

A.A 2021/2022

Docente: Antonio Celesti

Studente: Giuseppe Primerano

Numero di matricola: 476933

Sommario

Capitolo 1: Prefazione.

La frode assicurativa è un inganno deliberato perpetrato contro una compagnia assicurativa o un agente a scopo di lucro. La frode può essere commessa in diversi punti della transazione. Anche agenti assicurativi, dipendenti della società e avvocati possono commettere frodi assicurative. Le frodi comuni consistono nel travisare i fatti su una domanda di assicurazione, presentare reclami per lesioni o danni mai verificatisi e infine messa in scena di incidenti.

Le persone che commettono frodi assicurative includono:

- criminali organizzati che rubano ingenti somme attraverso attività commerciali fraudolente;
- professionisti e tecnici che gonfiano i costi del servizio o addebitano i servizi non resi;
- persone comuni che vogliono coprire la loro franchigia o considerano la presentazione di un reclamo come un'opportunità per fare un po' di soldi.

Molto spesso coloro che commettono delle frodi, creano delle identità false per poi presentarsi alle compagnie assicurative per divenire clienti. Capita però che i truffatori "riciclino" delle identità.

Alcune stime dichiarano che il costo totale delle frodi assicurative (esclusa l'assicurazione sanitaria) superi i 40 miliardi di dollari all'anno. La frode assicurativa costa alla famiglia americana in media tra $ 400 e $ 700 all'anno. La Coalition Against Insurance Fraud (CAIF) stima che la sola frode assicurativa contro le indennità dei lavoratori costi agli assicuratori e ai datori di lavoro sei miliardi di dollari l'anno.

Uno dei mezzi più efficaci per combattere le frodi è l'adozione di tecnologie dei dati che riducono il tempo necessario per riconoscere le frodi. I progressi nella tecnologia analitica sono cruciali nella lotta contro le frodi per stare al passo con anelli sofisticati che sviluppano costantemente nuove truffe.

Gli approcci tradizionali sono stati potenziati dalla modellazione predittiva e dall'analisi dei collegamenti, che esaminano le relazioni tra elementi come persone, luoghi ed eventi. L'intelligenza artificiale può essere utilizzata, tra gli altri strumenti, per scoprire le frodi prima che venga effettuato un pagamento.

I programmi che scansionano i sinistri assicurativi sono stati migliorati grazie al consolidamento dei database dei sinistri del settore assicurativo. I sistemi che

identificano le anomalie in un database possono essere utilizzati per sviluppare algoritmi che consentono a un assicuratore di interrompere automaticamente i pagamenti dei sinistri.

Circa il 40% degli assicuratori intervistati in un sondaggio ha affermato che i propri budget tecnologici per il 2019 saranno più grandi, con modelli predittivi e analisi dei collegamenti o dei social network i due tipi più probabili di programmi considerati per gli investimenti.

Capitolo 2: Tecnologie implementative.

Per lo sviluppo del progetto sono state utilizzate le seguenti tecnologie:

- Python 3.9;
- L' editor di testo PyCharm Community Edition 2021.2;
- MongoDB e MongoDBCompass;
- Neo4j Desktop;
- Microsoft Excel® per la creazione dei fogli di calcolo relativi all'analisi prestazionale;
- Microsoft Word® per la stesura della relazione;
- Draw.io per la creazione dei diagrammi.

Capitolo 3: Neo4j e MongoDB.

In questo capitolo verrà fatta una piccola descrizione dei due database utilizzati per l'analisi prestazionale fatta in questo progetto.

Neo4j.

Neo4j viene definito graph-database, è un servizio implementato in Java da un'azienda svedese la Neo Technology e la sua prima versione è stata rilasciata nel 2010. Aziende come eBay, Walmart, Telenor, UBS, Cisco, Hewlett-Packard e Lufthansa hanno fatto affidamento sulle qualità di Neo4j per migliorare i propri servizi. Neo4j utilizza i grafici per rappresentare i dati e le relazioni tra di essi. Un grafico è definito come qualsiasi rappresentazione grafica costituita da vertici (mostrati da cerchi) e bordi (mostrati con linee di intersezione). I database grafici come Neo4j hanno come caratteristica principale le alte prestazioni. La chiave è che, anche se le query di dati aumentano in modo esponenziale, le prestazioni di Neo4j non diminuiscono. Neo4j è anche molto reattivo. Altre caratteristiche di tale database sono sicuramente la flessibilità e la scalabilità in questo modo

quando le esigenze aumentano le possibilità di aggiungere più nodi e relazioni ad un grafo esistente sono enormi. Infine, i graph-database rispondono alle richieste aggiornando il nodo e le relazioni di quella ricerca e non l'intero grafo completo, ciò ottimizza il processo.

MongoDB.

MongoDB è un database nato intorno alla metà degli anni 2000. E' un database NoSQL orientato ai documenti (document-oriented) utilizzato per l'archiviazione di dati ad alto volume. Invece di utilizzare tabelle e righe come nei tradizionali database relazionali, MongoDB fa uso di raccolte (collection) e documenti. I documenti sono costituiti da coppie chiave-valore che sono l'unità di base dei dati in MongoDB. Le raccolte o collection, contengono insiemi di documenti e funzioni equivalenti alle tabelle dei database relazionali. Tra le caratteristiche principali di questo database NoSQL troviamo lo schema-less ovvero non è presente uno schema come, ad esempio, una tabella ma tutto viene organizzato mediante l'utilizzo di documenti, abbiamo la mancanza di join complessi che potrebbero diminuire le prestazioni, questo database ha anche una capacità di interrogazione profonda. MongoDB supporta query dinamiche sui documenti utilizzando un linguaggio di query basato su quest'ultimi. Infine, un'altra caratteristica è la sua facile scalabilità. Inoltre, possiamo dire che grazie alla sua struttura è uno dei database NoSQL con la maggiore velocità in scrittura.

Capitolo 4: Implementazione.

Nei seguenti paragrafi verranno illustrati tutti i passaggi che sono stati eseguiti per lo svolgimento del progetto.

4.1 Creazione dataset.

Per la creazione del dataset e relativo file .*csv* è stata utilizzata la libreria *Faker* messa a disposizione dal linguaggio Python, essa permette di generare in maniera casuale dei record. Nel caso in questione *faker* è servita per generare nomi, indirizzi e codici fiscali casuali e fittizi. Mediante il ciclo *for* è possibile iterare il numero di volte in cui avverrà la creazione, ciò ha permesso di creare un numero di record prestabilito dal programmatore. Grazie a ciò sono stati generati i dataset utilizzati per fare il confronto tra i due database. Sono stati creati 4 diversi dataset rispettivamente da cento, mille, dieci mila e cento mila record. In figura 4 è possibile vedere lo snippet relativo a ciò che è stato descritto in precedenza.

```python
import csv
from faker import Faker
import random
from faker_vehicle import VehicleProvider

falso = Faker(('it_IT'))
falso.add_provider(VehicleProvider)
lista_nomi = []
lista_cognomi = []
lista_compagnia = []
lista_avvocati = []
lista_avvocati2 = []
lista_tipologia = ["Danno da incendio", "Furto", "Incidente stradale", "Incidente sul lavoro", "Calamita naturale", "Fenomeno elettrico",
                   "Incidente aereo", "Danno materiale", "Danno fisico", "Danno punitivo", "Danno patrimoniale", "Danno biologico"]
lista_stato = ["Risolto", "Non risolto"]




for w in range (20):
    compagnia_ripetuta = falso.company()
    lista_compagnia.append(compagnia_ripetuta)
for z in range (15):
    nome_ripetuto = falso.first_name()
    cognome_ripetuto = falso.last_name()
    lista_nomi.append(nome_ripetuto)
    lista_cognomi.append(cognome_ripetuto)
    lista_avvocati.append(nome_ripetuto)
    lista_avvocati2.append(cognome_ripetuto)

with open('dataset100.csv', mode='w', newline='') as csv_file:
    fieldnames = ['NAME', 'CF', 'EMAIL', 'CELL', 'ADDRESS',  'CLAIM', 'EVALUATED', 'LAWYER', 'COMPANY', 'TYPE', 'STATE', 'DATE']
    writer = csv.DictWriter(csv_file, fieldnames=fieldnames)

    writer.writeheader()
    id = 1
```

```
    claim = 0
    for x in range(100):
        name = random.choice(lista_nomi) + " " + random.choice(lista_cognomi)
        cf = falso.ssn()
        email = falso.email()
        cell = falso.phone_number()
        address = falso.address()
        claim += 1
        evaluated = falso.name()
        lawyer = random.choice(lista_avvocati) + " " + random.choice(lista_avvocati2)
        company = random.choice(lista_compagnia)
        type = random.choice(lista_tipologia)
        state = random.choice(lista_stato)
        date = falso.date()
        writer.writerow(
            {
                'NAME': name,
                'CF': cf,
                'EMAIL': email,
                'CELL': cell,
                'ADDRESS': address,
                'CLAIM': claim,
                'EVALUATED': evaluated,
                'LAWYER': lawyer,
                'COMPANY': company,
                'TYPE': type,
                'STATE': state,
                'DATE': date
            })
id+=1
```

Figura 4 Funzione per la creazione del dataset.

Nel dataset saranno presenti i seguenti indici:

NAME: nome e cognome falso dell'utente creato mediante *faker* e scelto in maniera casuale dalle liste create.

CF: codice fiscale fittizio creato in maniera automatica da *faker*.

EMAIL: e-mail falsa dell'utente, creata mediante *faker*.

CELL: numero di telefono con prefisso italiano generato in maniera casuale e fittizia da *faker*.

ADDRESS: indirizzo falso dell'utente generato in maniera del tutto casuale da *faker*.

CLAIM: indica l'identificativo univoco di un richiamo, esso viene generato in maniera crescente e iterativa mediante un ciclo *for* partendo da 1 fino alla fine del dataset.

EVALUATED: nome e cognome falsi del perito assicurativo creati dalla libreria *faker*.

LAWYER: nome e cognome falsi dell'avvocato creati tramite la libreria *faker,* in maniera controllata tramite un ciclo *for,* inseriti in due liste e successivamente generati in maniera casuale e ripetuta prelevandoli tramite la funzione *random*.

COMPANY: indica la compagnia assicurativa per cui lavorano gli avvocati ed i periti. Essa viene generata in maniera casuale e controllata tramite *faker* inserita in una lista ed infine prelevata dalla lista (*lista_compagnia*) in maniera casuale tramite *random*.

TYPE: indica la tipologia del richiamo. Viene creata prelevando in maniera casuale un elemento che fa parte della lista (lista_tipologia) che contiene diverse tipologie di sinistri.

STATE: indica lo stato del reclamo. Viene generato scegliendo in maniera casuale un valore dalla lista (lista_stato) che contiene gli stati.

DATE: indica la data in cui un richiamo è stato fatto dall'utente. Viene generata in maniera casuale usando *faker*.

4.2 Datamodel.

Nella figura 4.1 possiamo notare il datamodel e le relazioni che lo compongono utilizzato nel caso di studio in questione.

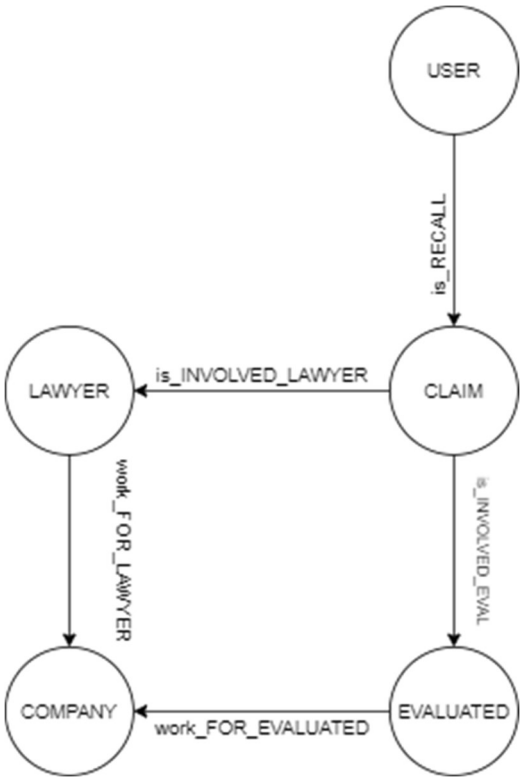


Figura 4.1 Datamodel

Queste sono le relazioni che collegano i vari nodi:

User-> [:is_RECALL]-> Claim

Claim-> [:is_INVOLVED_EVAL]-> Evaluated

Claim-> [:is_INVOLVED_LAWYER]-> Lawyer

Lawyer->[:work_FOR_LAWYER]->Company

Evaluated->[:work_FOR_EVAL]->Company

In MongoDB il datamodel è stato implementato seguendo la diversa natura del database. È stata creata una raccolta che consente di racchiudere i dati elencati in precedenza in modo che sia possibile avere gli stessi collegamenti e la stessa accessibilità.

4.3 Connessione ai database.

La connessione ai database è stata fatta come riportato nelle seguenti figure, infatti in figura 4.2 è mostrata la connessione per il database Neo4j invece in figura 4.3 è mostrata la connessione per il database MongoDB.

```
uri = "bolt://localhost:7687"
user = "neo4j"
psw =

driver = GraphDatabase.driver(uri, auth=(user, psw))
session = driver.session()
```

Figura 4.2

```
client = MongoClient('localhost', 27017)

nomedb = 'dati100'

miodb = client[nomedb]


dat100 = miodb.dat100
```

Figura 4.3

4.4 Creazione database.

Neo4j.

Per la creazione in Neo4j la prima cosa che è stata fatta è la creazione di un database locale su Neo4jDesktop, dopodiché all'interno di tale DB è stato importato il dataset in formato .csv per avere a disposizione i record creati. Successivamente sul client e quindi mediante l'utilizzo del linguaggio Cypher, è stata fatta la creazione di tutti i nodi e in seguito delle relazioni. Tutto ciò è stato possibile grazie all'importazione della libreria Python dedicata, chiamata appunto ***neo4j***.

Ecco (figura 4.4) la creazione dei nodi fatta su Python mediante Cypher.

```
#vengono creati i nodi###
session.run("""LOAD CSV WITH HEADERS FROM "file:///dataset100.csv"  AS row
MERGE (a:USER  {cf: row.CF})
ON CREATE SET a.name = row.NAME, a.email= row.EMAIL, a.cell = row.CELL , a.address=row.ADDRESS
ON MATCH SET a.name =row.NAME,  a.email= row.EMAIL, a.cell = row.CELL , a.address=row.ADDRESS

MERGE (b:EVAL {id_eval : row.EVALUATED})

MERGE (c:CLAIM {id_cla : row.CLAIM})
ON CREATE SET c.type = row.TYPE, c.state = row.STATE, c.date = row.DATE
ON MATCH SET  c.type = row.TYPE, c.state = row.STATE, c.date = row.DATE

MERGE (d:LAWYER {id_law: row.LAWYER})
MERGE (e:COMPANY {id_company: row.COMPANY})
""")
```

Figura 4.4

Qui (figura 4.5) invece possiamo vedere la creazione delle relazioni.

```
session.run("""LOAD CSV WITH HEADERS FROM "file:///dataset100.csv" AS row
MATCH (a:USER {cf: row.CF}), (c:CLAIM {id_cla: row.CLAIM})
CREATE (a)-[:is_RECALL]->(c)
""")

session.run("""LOAD CSV WITH HEADERS FROM "file:///dataset100.csv" AS row
MATCH  (d:LAWYER {id_law: row.LAWYER}), (c:CLAIM {id_cla: row.CLAIM})
CREATE (d)-[:is_INVOLVED_LAWYER]->(c)
""")

session.run("""LOAD CSV WITH HEADERS FROM "file:///dataset100.csv" AS row
MATCH  (b:EVAL {id_eval: row.EVALUATED}), (c:CLAIM {id_cla: row.CLAIM})
CREATE (b)-[:is_INVOLVED_EVAL]->(c)
""")

session.run("""LOAD CSV WITH HEADERS FROM "file:///dataset100.csv" AS row
MATCH  (d:LAWYER {id_law: row.LAWYER}), (e:COMPANY {id_company: row.COMPANY})
CREATE (d)-[:work_FOR_LAWYER]->(e)
""")

session.run("""LOAD CSV WITH HEADERS FROM "file:///dataset100.csv" AS row
MATCH  (b:EVAL {id_eval: row.EVALUATED}), (e:COMPANY {id_company: row.COMPANY})
CREATE (b)-[:work_FOR_EVAL]->(e)
""")
```

Figura 4.5

Eseguendo i precedenti snippet avremmo il nostro database Neo4j creato e pronto all'uso. Come si può vedere viene usata la clausola LOAD che permette di caricare i record dal dataset specificato.

Se si vogliono controllare gli snippet in questione bisogna andare nel file *creazione_db_neo4j_"numerorecord".py*

MongoDB.

In questo database è stata creata un'unica raccolta denominata in base al numero di record considerato (dati100, dati1000, dati10000, dati100000) in cui sono presenti tutti i dati del nostro dataset nel formato utilizzato da Mongo. Per la creazione del database è stata importate la libreria *pymongo* che permette di gestire i database Mongo su Phyton 3.9.

Come si può vedere dalla figura 4.6, innanzitutto viene specificato quale sarà il file da cui dovranno essere prelevati i record. Dopodiché è importante specificare il delimitatore (nel mio caso la virgola) che separerà un valore da un altro.

Per poter fare l'inserimento è stata utilizzata la funzione *insert_one()* tale funzione andrà ad inserire per ogni chiave indicata il valore corrispondente prelevandolo dal dataset creato.

Per fare ciò bisogna indicare la riga che deve essere prelevata all'interno della clausola *row[],* in questo modo verranno prelevati i valori della riga indicata fino al delimitatore.

Per quanto concerne la creazione del database bisogna stare molto attenti e fare in modo che nel *path* indicato all'interno della clausola *whit open()* ci sia l'esatto indirizzo del posto in cui il dataset è stato generato e poi memorizzato, altrimenti la creazione del database non avverrà.

Quindi si raccomanda il lettore di stare attento all'indirizzo di salvataggio del proprio dataset se si vuole evitare l'incombere degli errori e la mancata creazione del database.

```
16  with open(r'C:\Users\giuse\PycharmProjects\Prova\1_Cento_record\dataset100.csv') as csv_file:
17      csv_reader = csv.reader(csv_file, delimiter=',')
18      line_count = 0
19      for row in csv_reader:
20          if line_count == 0:
21              line_count += 1
22              continue
23          dat100.insert_one({'NAME': row[0], 'CF': row[1], 'EMAIL': row[2], 'CELL': row[3], 'ADDRESS': row[4],
24                             'CLAIM': int(row[5]), 'EVALUATED': row[6], 'LAWYER': row[7],
25                             'COMPANY': row[8], 'TYPE': row[9], 'STATE': row[10], 'DATE': (row[11])})
26          line_count += 1
```

Figura 4.6

Se si vogliono controllare gli snippet in questione bisogna andare nel file *creazione_db_mongoj_"numerorecord".py*.

Capitolo 5: Query.

Lo scopo principale di questo progetto è quello di sottoporre i nostri due database ad un'analisi prestazionale.

Per poter fare l'analisi sono state create cinque query d'interrogazione ognuna con difficoltà crescente ed ogni query è stata eseguita per 31 volte in maniera iterativa.

Per quanto riguarda le query in Neo4j è stato utilizzato un linguaggio che prende il nome di *Pypher* (Cypher + Python) tale linguaggio è una versione ibrida che mette insieme le caratteristiche del linguaggio tipico del graph-database Neo4j con quelle del linguaggio di programmazione orientato agli oggetti Python.

Le query in MongoDB sono state create anch'esse utilizzando il linguaggio proprietario del suddetto database, che prende il nome di MQL (MongoDB Query Language).

Nella creazione delle query sono stati utilizzati gli operatori logici, tra cui AND e OR, funzioni di conteggio come *count()* e di ordinamento come *ORDER BY* e *sort()*.

Relativamente alla progettazione e poi implementazione delle query è possibile fare delle piccole considerazioni personali sulle diverse difficoltà che sono nate. Riguardo a Neo4j posso dire che le difficoltà che ho trovato stanno nella differenza di sintassi che c'è tra il client e il server, è capitato spesso che le query scritte nel server andavano modificate nel momento in cui venivano scritte sul client. Dopo diverse prove sono arrivato ad un compromesso e sono riuscito ad avere gli stessi risultati (o errori) sia sul server che sul client. Uno dei consigli per evitare le mie stesse problematiche è quello di andare a capo ogni volta che si scrive una condizione.

In MongoDB le problematiche sono state uguali, infatti, nella versione desktop di tale database ovvero MongoDBCompass, le query si scrivono in maniera molto semplice, cosa che invece non accade nel momento in cui le query devono essere scritte sul client. Un altro difetto del linguaggio dal database Mongo sta nell'utilizzo delle parentesi; infatti, molto spesso gli errori che sono sopraggiunti nascevano da un errato utilizzo e/o posizionamento delle parentesi.

Fatte queste piccole considerazioni personali nei seguenti paragrafi verranno spiegate e poi mostrate le query implementate.

5.1 Prima query.

La prima query fa una ricerca e il conteggio di tutti gli utenti che hanno il nome ed il cognome uguali a quelli inseriti.

Neo4j.

```
16  def query_1(name):
17      query_1 = """MATCH (a:USER) WHERE a.name = $name
18                   RETURN count(a.name) AS conta"""
19      risultati = session.run(query_1, name = name)
20      dati = [dato["conta"] for dato in risultati]
21      return dati
```

MongoDB.

```
17  def query_uno(name):
18      lista = []
19      for x in dat100.aggregate([{"$count": name}]):
20          lista.append(x)
21      return lista
```

5.2 Seconda query.

Ricerca di tutti gli utenti che hanno il nome uguale a quello inserito oppure il codice fiscale uguale a quello inserito. Il risultato viene ordinato in base al nome in maniera crescente.

Neo4j.

```
23  def query_2(name, cf):
24      query_2 =       """MATCH (a:USER)-[r:is_RECALL]->(c:CLAIM)
25                      WHERE  a.name = $name OR
26                             a.cf = $cf
27
28                      RETURN r AS tipo_relazione, c AS richiamo, a AS utente
29                      ORDER BY a.name"""
30      risultati = session.run(query_2, name = name, cf = cf)
31      dati = [dato["utente"] for dato in risultati]
32
33      return dati
```

MongoDB.

```
23    def query_due(name, cf):
24        lista = []
25        for x in dat100.find( {"NAME": name}, {"CF": cf} ).sort("NAME", 1):
26            lista.append(x)
27        return lista
```

5.3 Terza query.

Ricerca di tutti gli utenti che hanno il nome, il codice fiscale e l'e-mail uguali a quelli inserti. Ci viene restituito il richiamo fatto e le relazioni che coinvolgono i nodi in questione.

Neo4j.

```
35    def query_3(name, cf, email):
36        query_3= """        MATCH   (a:USER)-[r1:is_RECALL]->(c:CLAIM),
37                                    (b:EVAL)-[r2:is_INVOLVED_EVAL]->(c:CLAIM)<-[r3:is_INVOLVED_LAWYER]-(d:LAWYER)
38                            WHERE   a.name = $name AND
39                                    a.cf = $cf AND
40                                    a.email = $email
41                            RETURN a AS utente, r1, c AS richiamo, r2, r3
42                            """
43
44        risultati = session.run(query_3, name = name, cf = cf, email = email)
45        dati = [dato["utente"] for dato in risultati]
46
47        return dati
```

MongoDB.

```
28    def query_tre(name, cf, email):
29        lista = []
30        for x in dat100.find({"NAME": name, "CF": cf, "EMAIL": email}):
31            lista.append(x)
32        return lista
```

5.4 Quarta query.

Ricerca degli utenti che hanno nome, cognome, e-mail e numero di cellulare uguali a quelli inseriti. Questa query restituisce l'utente, il richiamo fatto dall'utente, il perito e l'avvocato coinvolti nel richiamo e infine una relazione.

Neo4j.

```
49  def query_4(name, cf, email, cell):
50      query_4 = """MATCH  (a:USER)-[r1:is_RECALL]->(c:CLAIM),
51                           (b:EVAL)-[r4:is_INVOLVED_EVAL]->(c:CLAIM)<-[r5:is_INVOLVED_LAWYER]-(d:LAWYER),
52                           (b:EVAL)-[r6:work_FOR_EVAL]->(e:COMPANY)<-[r7:work_FOR_LAWYER]-(d:LAWYER)
53                  WHERE  a.name = $name AND
54                         a.cf = $cf AND
55                         a.email = $email AND
56                         a.cell = $cell
57
58                  RETURN a AS utente,  b as perito, c AS richiamo, d as avvocato, r1
59                  """
60
61
62      risultati = session.run(query_4, name = name, cf = cf, email = email,cell = cell )
63      dati = [dato["utente"] for dato in risultati]
64
65      return dati
```

MongoDB.

```
34  def query_quattro (name, cf, email, cell):
35      lista = []
36      for x in dat100.find({"NAME": name, "CF": cf, "EMAIL": email, "CELL": cell}):
37          lista.append(x)
38
```

5.5 Quinta query.

Ricerca degli utenti che hanno nome, cognome, e-mail e numero di cellulare uguali a quelli inseriti e l'indirizzo che inizia con la parola chiave immessa. Tale query restituisce tutte le relazioni e tutti i nodi coinvolti nell'interrogazione.

Neo4j.

```
67  def query_5(name, cf, email, cell, address):
68      query_5 = """MATCH  (a:USER)-[r1:is_RECALL]->(c:CLAIM),
69                          (b:EVAL)-[r4:is_INVOLVED_EVAL]->(c:CLAIM)<-[r5:is_INVOLVED_LAWYER]-(d:LAWYER),
70                          (b:EVAL)-[r6:work_FOR_EVAL]->(e:COMPANY)<-[r7:work_FOR_LAWYER]-(d:LAWYER)
71
72                  WHERE  a.name = $name AND
73                         a.cf = $cf AND
74                         a.email = $email AND
75                         a.cell = $cell AND
76                         a.address STARTS WITH $address
77
78
79                  RETURN a AS utente, b AS Perito, c AS richiamo, d AS avvocato, e AS Compagnia, r1, r4, r5, r6, r7
80                  ORDER BY a.cf
81              """
82
83      risultati = session.run(query_5, name=name, cf=cf, email=email, cell=cell, address=address)
84      dati = [dato["utente"] for dato in risultati]
85      return dati
```

MongoDB.

```
40  def query_cinque (name, cf, email, cell, address):
41      lista = []
42      for x in dat100.find({"NAME": name, "CF": cf, "EMAIL": email, "CELL": cell, "ADDRESS":{"$regex": address}}).sort("CF", 1):
43          lista.append(x)
44
45      return lista
```

Per una maggiore comprensione delle query è possibile visionare i file *query_neo4j.py* e *query_mongodb.py* i file sono stati divisi in base al numero di record da interrogare.

Capitolo 6: Analisi prestazionale.

Nel seguente capitolo verranno mostrati nel dettaglio la media dei tempi espressa in millisecondi, la deviazione standard e infine la confidenza al 95% relative ad ogni query. Per calcolare la media, in entrambi i database, sono stati utilizzati i tempi partendo dalla seconda iterazione, ciò viene fatto perché i database alla prima iterazione hanno dei meccanismi di caching. Per l'ottenimento dei tempi è stata importata la libreria Python *time()*, tutti i tempi sono espressi in millisecondi. I tempi sono stati scritti in maniera automatica grazie all'utilizzo della libreria *xlswriter()* essa permette di creare un file Excel e di scrivere all'interno di esso i tempi per ogni query e per ogni iterazione. In figura 6.1 possiamo vedere il codice Python che permette di calcolare i tempi e di scrivere i tempi all'interno dei file.

```
47  workbook = xlsxwriter.Workbook('Risultati100mongo.xlsx')
48  worksheet = workbook.add_worksheet()
49  row = 1
50  col = 0
51
52  for y in range(31):
53      a1 = tempo()
62      a2 = tempo()
63      worksheet.write(row, col, a2 - a1)
64      row += 1
65
66      print("abbiamo ottenuto il (", y + 1, "°) risultato in", a2 - a1, "millisecondi\n")
67
68
69  worksheet.write(row, col, a2-a1)
70  workbook.close()
```

Figura 6.1 Calcolo dei tempi.

Nel seguente capitolo verranno inoltre presentati degli istogrammi in cui viene mostrato la differenza nei primi tempi di esecuzione e la differenza della media dei tempi tra i due database considerati. Per la realizzazione degli istogrammi sull'asse delle ascisse sono state inserite le dimensioni del dataset considerato (100, 1000, 10000 e 100000), sull'asse delle ordinate sono stati inseriti, in un grafico i primi tempi e nell'altro la media dei tempi, si è tenuto in considerazione una scala logaritmica in base dieci.

Per la realizzazione dell'analisi, quindi dei calcoli della media, deviazione standard e confidenza al 95% e dei grafici da cui derivano le immagini che

saranno presentate di seguito è stato utilizzato un foglio di calcolo Excel. Perciò se si vuole avere una maggiore comprensione di quanto prodotto si consiglia di visionare il file allegato alla relazione denominato *Analisi_prestazionale.xlsx*.

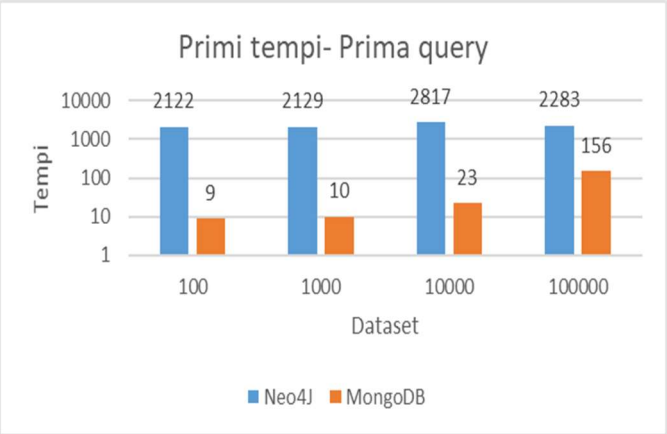
6.1 Analisi prestazionale prima query.

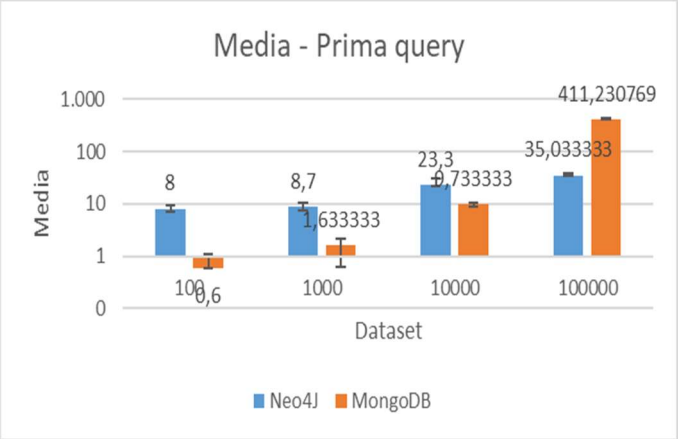
Risultati.

Primi tempi	100	1000	10000	100000
Neo4J	2122	2129	2817	2283
MongoDB	9	10	23	156

Media	100	1000	10000	100000
Neo4J	8	8,7	23,3	35,033333
MongoDB	0,6	1,633333	9,733333	411,230769

Grafici.





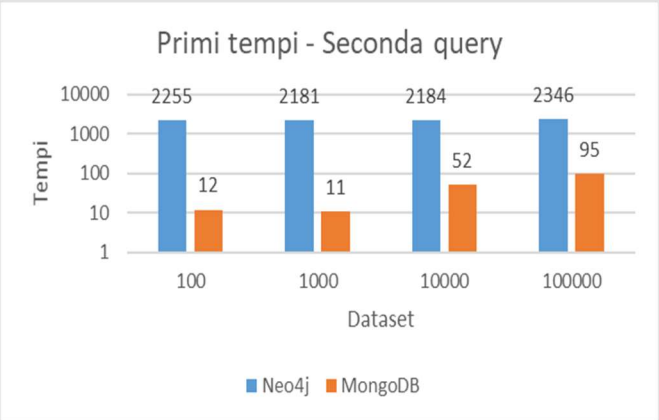
6.2 Analisi prestazionale seconda query.

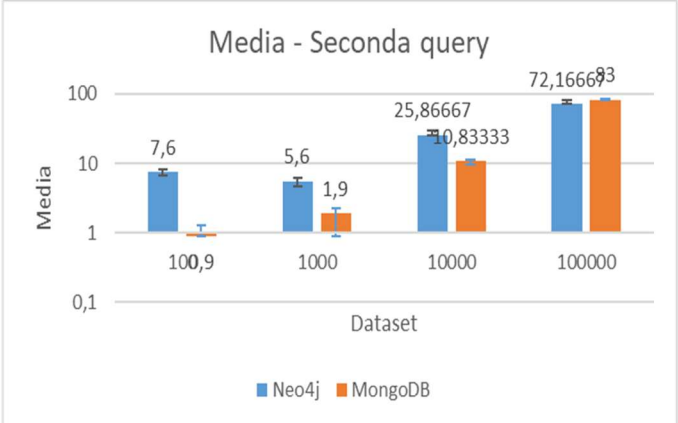
Risultati.

Primi tempi	100	1000	10000	100000
Neo4j	2255	2181	2184	2346
MongoDB	12	11	52	95

Media	100	1000	10000	100000
Neo4j	7,6	5,6	25,86667	72,16667
MongoDB	0,9	1,9	10,83333	83

Grafici.





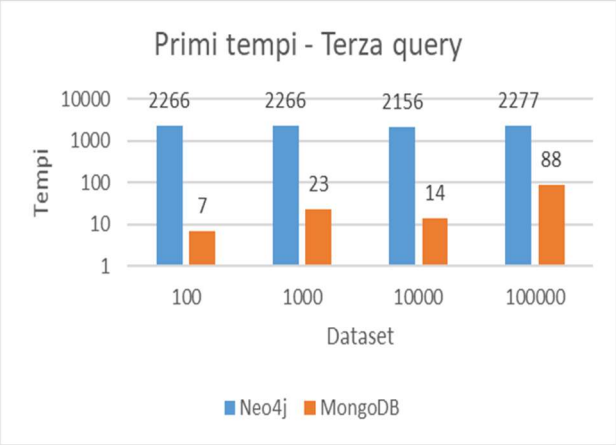
6.3 Analisi prestazionale terza query.

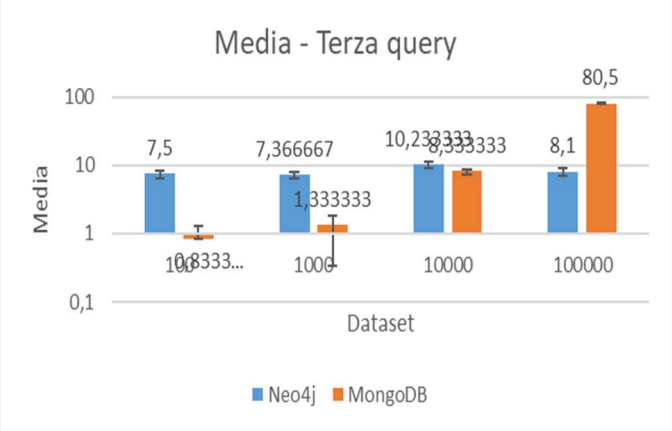
Risultati.

Primi tempi	100	1000	10000	100000
Neo4j	2266	2266	2156	2277
MongoDB	7	23	14	88

Media	100	1000	10000	100000
Neo4j	7,5	7,366667	10,233333	8,1
MongoDB	0,833333	1,333333	8,333333	80,5

Grafici.





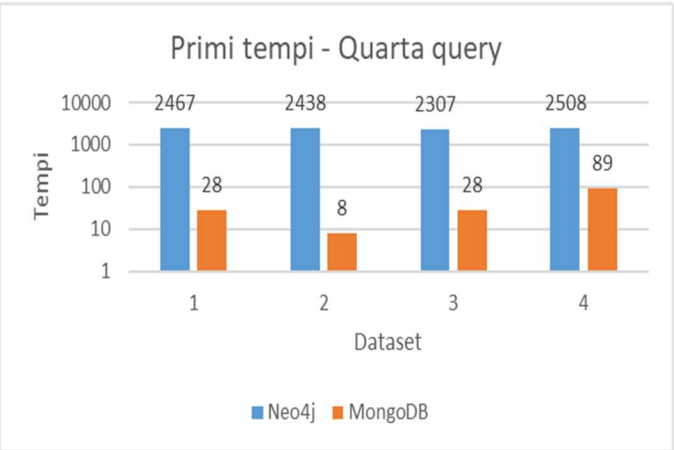
6.4 Analisi prestazionale quarta query.

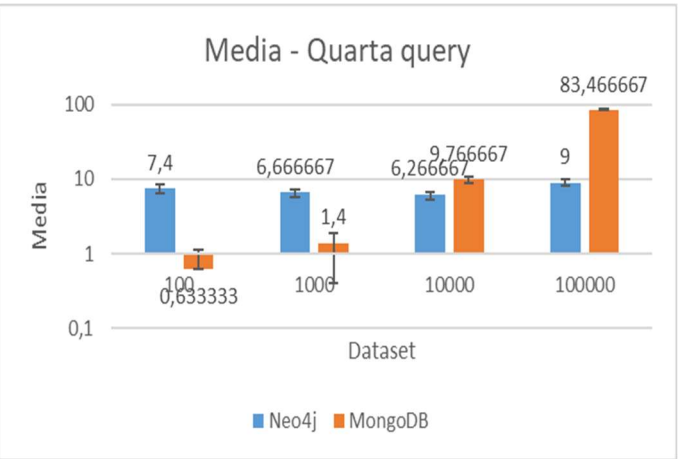
Risultati.

Primi tempi	100	1000	10000	100000
Neo4j	2467	2438	2307	2508
MongoDB	28	8	28	89

Media	100	1000	10000	100000
Neo4j	7,4	6,666667	6,266667	9
MongoDB	0,633333	1,4	9,766667	83,466667

Grafici.





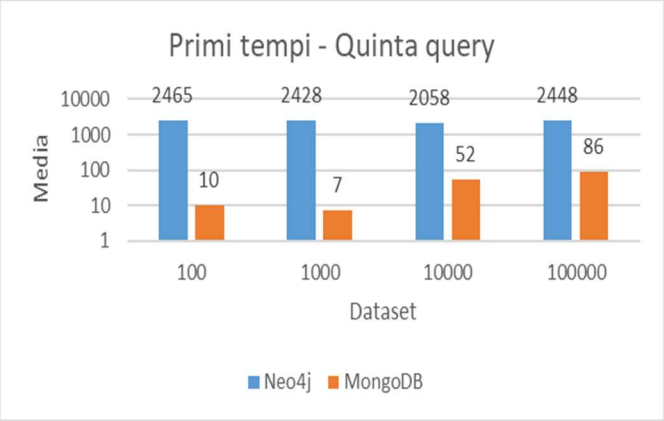
6.5 Analisi prestazionale quinta query.

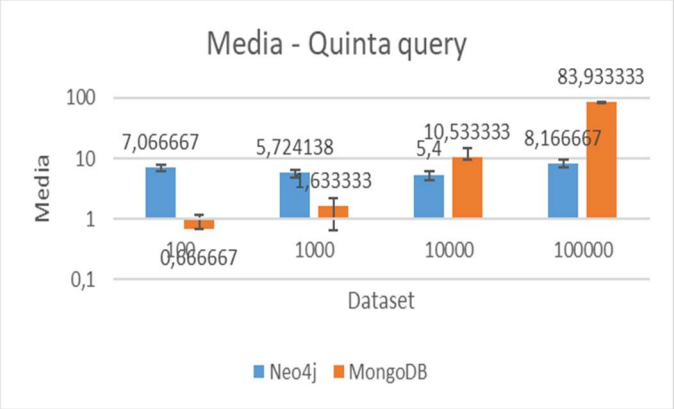
Risultati.

Primi tempi	100	1000	10000	100000
Neo4j	2465	2428	2058	2448
MongoDB	10	7	52	86

Media	100	1000	10000	100000
Neo4j	7,066667	5,724138	5,4	8,166667
MongoDB	0,666667	1,633333	10,533333	83,933333

Grafici.





Capitolo 7: Conclusioni.

Portata a termine l'analisi prestazionale possiamo dire che nel caso considerato il confronto tra i due database va a favore del graph-database, Neo4j. Quindi anche il caso in questione analizzato in questo progetto conferma tutto ciò che è scritto all'interno della documentazione ufficiale di tale database, dove gli sviluppatori affermano con decisione che la loro creatura è una delle soluzioni migliori sul mercato quando si vogliono interrogare grandi quantità di dati.

A parità di dataset possiamo notare come i tempi siano dalla parte di MongoDB, solo nel momento in cui ci troviamo a processare i dataset composti da cento e mille record, le prestazioni del sopracitato database, sono poco migliori di quelle del graph-database Neo4J. Possiamo quindi notare che quando vengono fatte le query in cui si considerano tutte le relazioni e un gran numero di record Neo4J è nettamente superiore rispetto a MongoDB.

Da questo studio possiamo notare come Neo4j si presta bene a situazioni in cui ci sono molti record da analizzare e tante relazioni da considerare. Invece nel caso in cui i record da considerare fossero pochi è preferibile l'utilizzo del database document-oriented MongoDB. Perciò dallo studio condotto si evince che se si vogliano considerare situazioni in cui abbiamo a nostra disposizione dei Big-Data, ovvero grosse mole di dati, è preferibile risolvere il problema sfruttando le caratteristiche offerte da Neo4J.

Dal punto di vista della creazione, quindi considerando operazioni di scrittura le prestazioni sono pressocché uguali. Il tempo trascorso per la generazione del database differisce di pochi millisecondi. Un piccolo problema è sorto durante la creazione del database composto da un milione di record. Per testare ancora di più le prestazioni dei due database si era deciso di creare un dataset e successivamente un database composto da un milione di record, in MongoDB dopo circa due minuti di attesa la generazione del database è avvenuta con successo, la stessa cosa non si può dire per Neo4j. Nel graph-database è scaturito un errore dovuto al fatto che la macchina in mio possesso non riesce ad avere lo spazio necessario per gestire un database di tali dimensioni, nonostante siano stati cambiati dei parametri di configurazione come suggerito dalla guida. Il problema sopraggiunto è dovuto al fatto che un database di questa grandezza con la struttura da me realizzata aveva un numero di nodi superiore al milione e un numero di relazioni uguale a cinque milioni, quindi la gestione dal punto di vista prestazionale risulta troppo onerosa da gestire dal personal computer a mia disposizione.

Bibliografia.

Per la realizzazione della relazione sono stati consultate le seguenti fonti:

- Il piccolo libro di MongoDB di Karl Seguin;
- Documentazione ufficiale di MongoDB, Neo4j e Python 3.9;
- Vari siti internet;
- Slide del docente Antonio Celesti presentate a lezione.