Report Basi di dati NoSQL

Orbitello Robin William (501648)

1. Problema affrontato

Il caso di studio scelto per confrontare i due DBMS è stato preso da un articolo in un sito web.
Il sito in questione è “Linkurious”. L’articolo in considerazione riguarda la semplificazione delle indagini svolte dalle forze dell’ordine utilizzando l’analisi dei collegamenti (link analysis).
https://linkurious.com/blog/rs21-data-investigation-criminal-justice/
Tutto questo è stato possibile grazie ad un’azienda di data science sede negli Stati Uniti.
L’azienda RS21 si è messa a lavoro per creare Quaro, uno strumento per ottimizzare le indagini sui dati utilizzando la potenza della tecnologia dei grafici e l’analisi avanzata dei link.
L’utilizzo dei dati può aiutare ad individuare degli elementi importanti come ad esempio delle reti organizzate (gang, gruppi terroristici, criminali informatici ecc.).
Per ottenere le giuste intuizioni dai dati e necessario riunire fonti di dati eterogenee e trovare le connessioni che contano.

Il problema principale quindi è quello di trovare delle connessioni tra determinate entità per stabilire se sono presenti delle reti organizzate, e questo è possibile farlo utilizzando dei DBMS come Neo4j o Oracle che facilitano la ricerca nei dati. Verranno eseguiti anche delle considerazioni riguardanti le tempistiche dei due DBMS a parità di dataset e complessità delle query.

2. Soluzione DBMS considerata (breve descrizione delle principali caratteristiche del database utilizzato)

La soluzione considerata per risolvere il problema è la creazione di un database che contenga tutti i dati necessari per la ricerca.
I database presi in considerazione sono Neo4j e Oracle.

Neo4j
Neo4j è un tipo di DBMS grafico (NoSQL), organizza i dati come nodi (entità), archi (relazioni) e proprietà (attributi di nodi o relazioni).
Utilizza un modello native graph storage che rappresenta i dati direttamente come grafi ottimizzando traversamenti e query di relazione.
Il linguaggio utilizzato è Cypher.
Neo4j è eccellente per analisi e navigazione su relazioni complesse, riduce drasticamente il tempo di elaborazione rispetto ai database relazionali ed è molto efficiente in caso di dataset moderatamente grandi (fino a miliardi di nodi).
Neo4j mantiene una cache in memoria delle pagine più frequentemente accessibili per ridurre l’I/O disco.
Lo schema che utilizza è lo schema-less che offre flessibilità nella struttura dei dati.
Un caso di uso ideale è ad esempio l’analisi di reti criminali.

Oracle
Oracle è un tipo di DBMS relazionale (RDBMS) ma supporta anche funzionalità NoSQL e dati non strutturati.
È basato su tabelle relazionali con supporto per gerarchie, JSON, XML e altri tipi di dati avanzati.
Il linguaggio utilizzato è SQL/PLSQL.
Richiede uno schema definito (schema-based) con validazione rigorosa.
Altamente scalabile sia orizzontalmente che verticalmente, adatto a grandi sistemi enterprise.

Caratteristica	Neo4j	Oracle
Tipo di DBMS	Grafico	Relazionale
Modello Dati	Nodi e relazioni	Tabelle relazionali
Schema	Schema-less	Schema-based
Linguaggio di Query	Cypher	SQL/PLSQL
Caso d'uso ideale	Relazioni complesse	Sistemi transazionali e analitici
Scalabilità	Moderata (grafi grandi)	Elevata (enterprise-scale)

3. Progettazione (contenente descrizione del modello dati utilizzato)

Neo4j

Il modello dati utilizzato in Neo4j è un tipo di modello di dati a grafo.
Il modello a grafo è formato da:

- Nodi: Rappresentano le entità.
- Relazioni: Rappresentano le connessioni dirette tra i nodi (es. “Avviene_in”).
- Proprietà: Sia i nodi che le relazioni possono avere attributi sotto forma di coppie chiave-valore.
- Etichette: Categorizzano i nodi e le relazioni.

Per lo studio condotto sono stati utilizzati 4 tipi di entità e 4 tipi di relazione.
Entità:

1) Persona: Entità che rappresenta una persona. La persona è identificata tramite un codice fiscale ed è categorizzata come “criminale” o “vittima”.
2) Evento: Entità che rappresenta l’accaduto o l’evento (es. rapina, omicidio..). Ogni evento è caratterizzato da un “id” che ne identifica l’unicità.
3) Oggetto: Entità che rappresenta l’oggetto utilizzato dal criminale. È stato definito come entità piuttosto che come attributo per riconoscere se l’oggetto è stato utilizzato in altri eventi o da altre persone. Ogni oggetto è identificato da un numero di serie.
4) Luogo: Entità che si riferisce al luogo dell’avvenimento. È identificato tramite un CAP e dà la possibilità di riconoscere quanti eventi sono accaduti in un determinato luogo. Utile anche per smascherare individui che compiono atti criminali nello stesso luogo e con lo stesso modus operandi.

Persona

Attributi	Valori
Codice fiscale	4FII19PN07XS2T19
Città	Napoli
Nome	Trixi
Cognome	Denekamp
Età	74
Ruolo	Vittima
Precedenti	False
Sesso	Femminile

Evento

Attributi	Valori
ID	20201
Data	2025-05-24
Nome	Omicidio

Oggetto

Attributi	Valori
Numero serie	6012
Nome	Pistola

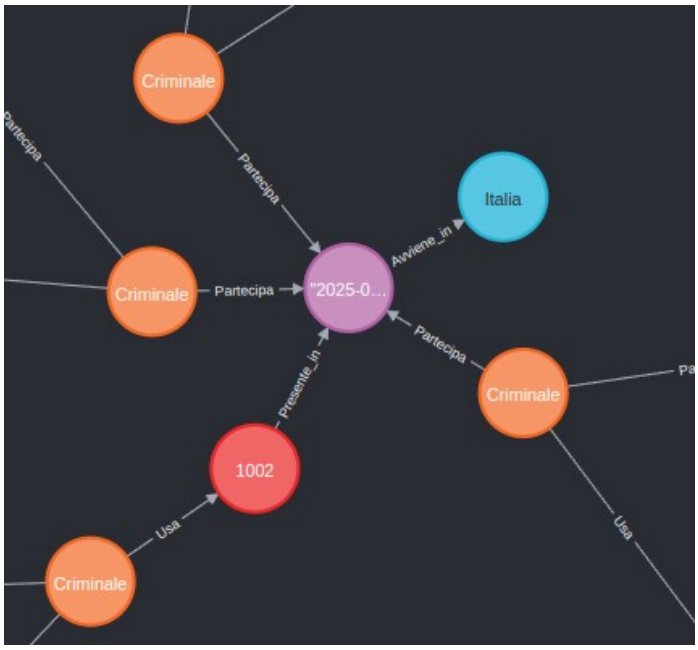
Luogo

Attributi	Valori
CAP	11015
Città	Palermo
Nome	McDonald
Paese	Italia

Relazioni:

1) Partecipa: Relazione che associa una persona ad un evento.
2) Avviene_in: Relazione che associa un evento ad un luogo.
3) Usa: Relazione che associa un oggetto ad una persona.
4) Presente_in: Relazione che associa un oggetto ad un evento.

Nello studio condotto le relazioni sono privi di attributi, hanno solo il compito di associare le entità.



Nella figura è possibile notare le entità con le corrispettive relazioni.
È possibile notare come 3 criminali (nodo arancione) che hanno partecipato ad eventi (nodo viola) diversi abbiano partecipato anche allo stesso evento, ed è possibile notare come un oggetto (nodo rosso) presente all'evento venga associato ad un altro criminale che non è direttamente connesso all'evento. Ciò porterebbe alla conclusione che esiste la probabilità che il criminale, proprietario dell'oggetto coinvolto, abbia partecipato all'evento.
In questo caso nel luogo (nodo azzurro) in cui è avvenuto l'evento è avvenuto solo e soltanto quell'evento.

Oracle

Oracle è un database che utilizza un modello relazionale basato su tabelle, righe e colonne.
Le tabelle sono collegate tramite attributi chiave e le entità utilizzate in Oracle sono le medesime di Neo4j. Le relazioni invece sono state realizzate tramite l'utilizzo di tabelle relazionali che consentono di associare elementi presenti in una tabella con altri elementi di altre tabelle.

Tabelle entità:

1) PERSONA: Codice fiscale (primary key)
2) EVENTO: ID (primary key)
3) OGGETTO: Numero serie (primary key)
4) LUOGO: CAP (primary key)

Tabelle relazionali:

1) EVENTO_LUOGO: Associa gli eventi ai luoghi
2) OGGETTO_EVENTO: Associa gli oggetti agli eventi

3) PERSONA_EVENTO: Associa le persone agli eventi
4) PERSONA OGGETTO: Associa le persone agli oggetti

4. Implementazione (contenente il codice utilizzato per l'inserimento dei dati e per l'implementazione di ciascuna interrogazione)

Per la creazione dei dati da inserire nei database è stato utilizzato Mockaroo, un sito web che facilita l'assegnazione dei valori (casuali) a determinati attributi scelti.
Sono stati usati 25 file .csv per entità, ogni file contiene 1000 elementi quindi un numero totale di 100.000 righe/nodi da inserire per ogni database.

Neo4j

Per l'inserimento dei dati in Neo4j i file .csv sono stati inseriti all'interno della cartella "import".
Mediante Python è stato fatto un inserimento automatico utilizzando la libreria "neo4j".
Per gestire il database è stato creato un file Neo4jQuery.py con le funzioni necessarie per il progetto.

1) check_connection: Funzione che permette di verificare se la connessione al database è andata a buon fine. Viene fatto eseguire l'istruzione "RETURN 1" per verificare se il database manda come risposta "1". In mancanza di tale risposta significa che non è stata effettuata la connessione al database.

```
# Metodo per verificare la connessione al database.
def check_connection(driver):  1 usage
    with driver.session() as session:
        try:
            result = session.run("RETURN 1")
            if result.single()[0] == 1:
                print("Connessione effettuata (Neo4j).")
            else:
                print("Connessione fallita (Neo4j).")
        except Exception as e:
            print("Errore durante la verifica della connessione:", e)
```

2) deleteNeo4j: Funzione che permette di cancellare tutti i nodi presenti nel database.

```
# Svuota database Neo4j
def deleteNeo4j(driver):  2 usages
    with driver.session() as session:
        try:
            session.run("MATCH (n) DETACH DELETE n")
            print("Neo4j svuotato.")
        except Exception as e:
            print("Errore svuotamento Neo4j:", e)
```

3) datiNeo4j: Funzione che permette di inserire nel database i dati contenuti nei file .csv. In base al numero inserito nell'argomento è possibile modificare la quantità di dati 25%, 50%, 75% e 100%. Nel main è stato implementato l'inserimento del numero automaticamente in base all'opzione scelta nel menù.

```
# Metodo per caricare i dati
def datiNeo4j(driver, numero):  4 usages
    with driver.session() as session:
        prima_parte_luogo = "LOAD CSV WITH HEADERS FROM 'file:///nodo_luogo"
        seconda_parte_luogo = (".csv' AS row MERGE (l:Luogo {cap: toInteger(row.cap)}) "
                               "SET l.citta = row.citta, l.nome = row.nome, "
                               "l.paese = row.paese;")
        prima_parte_persona = "LOAD CSV WITH HEADERS FROM 'file:///nodo_persona"
        seconda_parte_persona = (
            ".csv' AS row MERGE (p:Persona {codiceFiscale: row.codiceFiscale}) "
            "SET p.nome = row.nome, "
            "p.cognome = row.cognome, "
            "p.eta = toInteger(row.eta), "
            "p.precedenti = row.precedenti, "
            "p.citta = row.citta, "
            "p.ruolo = row.ruolo, "
            "p.sesso = row.sesso;"
        )
        prima_parte_oggetto = "LOAD CSV WITH HEADERS FROM 'file:///nodo_oggetto"
        seconda_parte_oggetto = (
            ".csv' AS row MERGE (o:Oggetto {numeroSerie: toInteger(row.numeroSerie)}) "
            "SET o.nome = row.nome;"
        )
        prima_parte_evento = "LOAD CSV WITH HEADERS FROM 'file:///nodo_evento"
        seconda_parte_evento = (
            ".csv' AS row MERGE (e:Evento {id: toInteger(row.id)}) "
            "SET e.data = date(row.data), "
            "e.nome = row.nome;"
        )
        for i in range(numero):
            if i == 0 :
                percorso_luogo = prima_parte_luogo + seconda_parte_luogo
                percorso_persona = prima_parte_persona + seconda_parte_persona
                percorso_oggetto = prima_parte_oggetto + seconda_parte_oggetto
                percorso_evento = prima_parte_evento + seconda_parte_evento
            else:
```

```
            carattere = str(i+1)
            percorso_luogo = prima_parte_luogo + carattere + seconda_parte_luogo
            percorso_persona = prima_parte_persona + carattere + seconda_parte_persona
            percorso_oggetto = prima_parte_oggetto + carattere + seconda_parte_oggetto
            percorso_evento = prima_parte_evento + carattere + seconda_parte_evento
        session.run(percorso_luogo)
        session.run(percorso_persona)
        session.run(percorso_oggetto)
        session.run(percorso_evento)
    result1 = session.run("MATCH (l:Luogo) RETURN l LIMIT 10")
    result2 = session.run("MATCH (l:Persona) RETURN l LIMIT 10")
    result3 = session.run("MATCH (l:Oggetto) RETURN l LIMIT 10")
    result4 = session.run("MATCH (l:Evento) RETURN l LIMIT 10")
    if not result1.peek() or not result2.peek() or not result3.peek() or not result4.pee
        print("Errore caricamento. (Neo4j)")
    else:
        print("Caricamento dati completato. (Neo4j)")
```

4) creaRelazione: Funzione che permette di creare relazioni tra i nodi in modo casuale e coerente con il problema affrontato.

```
def creaRelazione(driver):  4 usages
    with driver.session() as session:
        # Relazione univoca Persona->Evento
        session.run("""MATCH (p:Persona) WHERE NOT (p)--() WITH collect(p) AS persone
                    MATCH (e:Evento) WHERE NOT (e)<-[:Partecipa]-() WITH persone, collect(e) AS eventi
                    UNWIND range(0, size(persone) - 1) AS index WITH persone[index]
                    AS persona, eventi[index % size(eventi)]
                    AS evento CREATE (persona)-[:Partecipa]->(evento)""")
        # Relazione 70% Persona->Evento
        session.run("""MATCH (p:Persona) WITH collect(p) AS persone WITH size(persone)
                    AS total, persone, toInteger(0.7 * size(persone)) AS limite
                    WITH persone[0..limite] AS persone_selezionate
                    MATCH (e:Evento) WITH persone_selezionate, collect(e) AS eventi
                    UNWIND persone_selezionate AS persona WITH persona, eventi
                    WITH persona, eventi, toInteger(rand() * size(eventi)) AS randomIndex
                    WITH persona, eventi[randomIndex] AS evento WHERE NOT (persona)-[:Partecipa]->(evento)
                    MERGE (persona)-[:Partecipa]->(evento) // Crea la relazione se non esiste già""")
        # Relazione Evento->Luogo
        session.run("""MATCH (e:Evento) WITH collect(e) AS eventi MATCH (l:Luogo)
                    WHERE NOT (l)<-[:Avviene_in]-() WITH eventi, collect(l) AS luoghi
                    UNWIND range(0, size(eventi) - 1) AS index WITH eventi[index]
                    AS evento, luoghi[index % size(luoghi)] AS luogo
                    MERGE (evento)-[:Avviene_in]->(luogo)""")
        # Relazione Persona->Oggetto
        session.run("""MATCH (p:Persona{ruolo:'Criminale'}) WITH collect(p) AS persone
                    MATCH (o:Oggetto) WHERE NOT (o)<-[:Usa]-() WITH persone, collect(o) AS oggetti
                    UNWIND range(0, size(persone) - 1) AS index WITH persone[index]
                    AS persona, oggetti[index % size(oggetti)] AS oggetto
                    CREATE (persona)-[:Usa]->(oggetto) // Crea la relazione""")
        # Relazione Oggetto->Evento
        session.run("""MATCH (o:Oggetto) WITH collect(o) AS oggetti MATCH (e:Evento)
                    WHERE NOT (e)<-[:Presente_in]-() WITH oggetti, collect(e) AS eventi
                    UNWIND range(0, size(oggetti) - 1) AS index WITH oggetti[index]
                    AS oggetto, eventi[index % size(eventi)] AS evento
                    CREATE (oggetto)-[:Presente_in]->(evento) // Crea la relazione""")
```

```
    # Query per il controllo delle relazioni
    relazioni = {
        "Partecipa": "MATCH (:Persona)-[:Partecipa]->(:Evento) RETURN 1 LIMIT 1",
        "Avviene_in": "MATCH (:Evento)-[:Avviene_in]->(:Luogo) RETURN 1 LIMIT 1",
        "Usa": "MATCH (:Persona)-[:Usa]->(:Oggetto) RETURN 1 LIMIT 1",
        "Presente_in": "MATCH (:Oggetto)-[:Presente_in]->(:Evento) RETURN 1 LIMIT 1"
    }
    for nome, query in relazioni.items():
        result = session.run(query).single()
        if result:
            continue
        else:
            print(f"Errore: relazione '{nome}' non creata.")
            break
    print("Caricamento relazioni completato. (Neo4j)")
```

5) neo4jComplessità1: Funzione che esegue 31 volte la query di complessità 1. La query dà come risultato tutti i nodi di un entità (per analizzare il caso è stato scelto automaticamente l'entità persona). Per poter calcolare il tempo impiegato è stato utilizzato la libreria "time", quindi viene registrato il tempo di ogni esecuzione per poter ricavare il tempo della prima istruzione, il tempo delle successive 30 istruzioni (per il calcolo della confidenza al 95%) e il tempo medio delle ultime 30 istruzioni. I valori registrati verranno poi utilizzati come valori di ritorno della funzione. La variabile "last_return" ritorna il risultato dell'ultima query per poterlo stampare.

```
# Complessità 1
def neo4jComplessita1(driver, label):  1 usage
    lista_tempo = []
    somma_tempo = 0
    tempo_iniziale = 0
    iterations = 31
    last_result = None
    for i in range(iterations):
        query = f"MATCH (n:{label}) RETURN n"
        with driver.session() as session:
            start_time = time.time()
            result = session.run(query)
            end_time = time.time()
            timediff = (end_time - start_time) * 1000
            if tempo_iniziale == 0:
                tempo_iniziale = timediff
            else:
                lista_tempo.append(timediff)
                somma_tempo += timediff
            last_result = [record["n"] for record in result]
    media_tempo = somma_tempo/30
    return last_result, tempo_iniziale, media_tempo, lista_tempo
```

6) neo4jComplessità2: Funzione che fa esattamente la stessa cosa della funzione precedente ma utilizzando una query di complessità 2. La query restituisce un tipo di entita (per analizzare il caso è stato scelto automaticamente l'entità persona), le sue corrispettive relazioni e i nodi relazionati ad esso.

```python
# Complessità 2
def neo4jComplessita2(driver, label):  1 usage
    lista_tempo = []
    tempo_iniziale = 0
    somma_tempo = 0
    iterations = 31
    last_result = None
    for i in range(iterations):
        query = f"MATCH (p:{label})-[r]->(n) RETURN p,r,n"
        with driver.session() as session:
            start_time = time.time()
            result = session.run(query)
            end_time = time.time()
            timediff = (end_time - start_time) * 1000
            if tempo_iniziale == 0:
                tempo_iniziale = timediff
            else:
                lista_tempo.append(timediff)
                somma_tempo += timediff
            last_result = [(record["p"], record["r"], record["n"]) for record in result]
    media_tempo = somma_tempo / 30
    return last_result, tempo_iniziale, media_tempo, lista_tempo
```

7) neo4jComplessità3: Funzione che fa esattamente la stessa cosa della funzione precedente ma utilizzando una query di complessità 3. La query restituisce i nodi di un certo tipo (per analizzare il caso è stato scelto automaticamente l'entità persona) che sono collegati tramite il maggior numero di relazioni ad altri nodi. Restituisce quindi le entità, le relazioni legate alle entità, i nodi connessi dalle relazioni e il numero di connessioni appartenenti alle entità, tutto in ordine decrescente in base al numero di relazioni. Questa query è utile per identificare i nodi con il maggior numero di connessioni in un grafo.

```python
def neo4jComplessita3(driver, label):  1 usage
    lista_tempo = []
    tempo_iniziale = 0
    somma_tempo = 0
    iterations = 31
    persone_con_relazioni = None
    for i in range(iterations):
        query = (f"MATCH (p:{label})-[r]->(n) "+
        "WITH p, collect(r) AS relazioni, collect(n) AS nodi_connessi, count(r) AS num_relazioni "+
        "RETURN p, relazioni, nodi_connessi, num_relazioni "+
        "ORDER BY num_relazioni DESC ")
```

```
        with driver.session() as session:
            start_time = time.time()
            result = session.run(query)
            end_time = time.time()
            timediff = (end_time - start_time) * 1000
            if tempo_iniziale == 0:
                tempo_iniziale = timediff
            else:
                lista_tempo.append(timediff)
                somma_tempo += timediff
            persone_con_relazioni = [
                (record["p"], record["relazioni"], record["nodi_connessi"], record["num_relazioni"])
                for record in result
            ]
    for persona, relazioni, nodi_connessi, num_relazioni in persone_con_relazioni:
        props_persona = persona._properties
        print(f"Persona ID: {persona.id}")
        for chiave, valore in props_persona.items():
            print(f"  {chiave}: {valore}")
        print(f"  Numero di connessioni: {num_relazioni}")
        print("  Relazioni:")
        for relazione in relazioni:
            print(f"    Tipo: {relazione.type}")
        print("  Nodi connessi:")
        for nodo in nodi_connessi:
            props_nodo = nodo._properties
            print(f"    Nodo ID: {nodo.id}")
            for chiave, valore in props_nodo.items():
                print(f"      {chiave}: {valore}")

        print("-" * 40)
    media_tempo = somma_tempo / 30
    return tempo_iniziale, media_tempo, lista_tempo
```

8) neo4jComplessità4: Funzione che fa esattamente la stessa cosa della funzione precedente ma utilizzando una query di complessità 4. La query restituisce p1 (persona1) e p2 (persona2) connessi tramite il nodo intemedio “n”, i tipi univoci di relazione che li collegano, il numero totale di connessioni tra i nodi p1 e p2 tramite il nodo intermedio, il numero totale di connessioni uscenti dal nodo p1, tutto in ordine decrescente.

```
query = f"""
    MATCH (p1:{label})-[r:Avviene_in|Usa|Partecipa|Presente_in]->(n)<-[r2]-(p2:{label})
    WHERE p1.codiceFiscale <> p2.codiceFiscale
    WITH p1, p2, collect(DISTINCT type(r)) AS tipi_relazioni,
        count(r) AS num_connessioni
    OPTIONAL MATCH (p1)-[]->()
    WITH p1, p2, tipi_relazioni, num_connessioni, count(*) AS grado_medio
    RETURN p1, p2, tipi_relazioni, num_connessioni, grado_medio
    ORDER BY num_connessioni DESC """
```

Oracle
Per l'inserimento dei dati in Oracle i file .csv sono stati inseriti all'interno della cartella del progetto.
Mediante Python è stato fatto un inserimento automatico dei dati a Oracle utilizzando la libreria "csv".
Per gestire il database è stato creato un file OracleQuery.py con le funzioni necessarie per il progetto.
Per la creazione delle tabelle è stato utilizzato il linguaggio PL/SQL in Dbeaver.

1) deleteOracle: Funzione che chiama la procedura "drop_tables" contenuta all'interno del package "manageBase". La procedura consente di eliminare tutte le tabelle presenti nel database.

```
def deleteOracle(cursor):  2 usages
    cursor.callproc("manageBase.drop_tables")
    print("Oracle svuotato.")
```

2) creaRelazioniOracle: Funzione che chiama le procedure all'interno del package "relazioni" per l'inserimento casuale e coerente dei dati nelle tabelle relazionali.

```
def creaRelazioniOracle(cursor):  4 usages
        cursor.callproc("relazioni.crea_relazioni_persona_evento")
        cursor.callproc("relazioni.crea_relazioni_persona_oggetto")
        cursor.callproc("relazioni.crea_relazioni_oggetto_evento")
        cursor.callproc("relazioni.crea_relazioni_evento_luogo")
        print("Caricamento relazioni completato. (Oracle)")
```

3) datiOracle: Funzione che permette di caricare i file .csv all'interno delle tabelle. La funzione permette il caricamento parziale dei file (25%, 50%, 75%, 100%) tramite l'argomento "numero".

```
def datiOracle(cursor, conn, numero):  4 usages
    prima_parte_persona = "nodi/nodo_persona"
    prima_parte_evento = "nodi/nodo_evento"
    prima_parte_luogo = "nodi/nodo_luogo"
    prima_parte_oggetto = "nodi/nodo_oggetto"
    seconda_parte = ".csv"
    for i in range(numero):
        if i == 0 :
            percorso_persona = prima_parte_persona + seconda_parte
            percorso_evento = prima_parte_evento + seconda_parte
            percorso_luogo = prima_parte_luogo + seconda_parte
            percorso_oggetto = prima_parte_oggetto + seconda_parte
        else :
            carattere = str(i+1)
            percorso_persona = prima_parte_persona + carattere + seconda_parte
            percorso_evento = prima_parte_evento + carattere + seconda_parte
            percorso_luogo = prima_parte_luogo + carattere + seconda_parte
            percorso_oggetto = prima_parte_oggetto + carattere + seconda_parte
        try:
            with open(percorso_persona, "r") as file:
                reader = csv.reader(file)
                next(reader)
```

```
            for row in reader:
                try:
                    cursor.execute("""
                        INSERT INTO persona (citta, codiceFiscale, cognome, eta, nome, precedenti, ruolo, sesso)
                        VALUES (:1, :2, :3, :4, :5, :6, :7, :8)
                    """, row)
                except Exception as e:
                    print(f"Errore nell'inserimento della riga {row}: {e}")
        with open(percorso_evento, "r") as file:
            reader = csv.reader(file)
            next(reader)
            for row in reader:
                try:
                    cursor.execute("""
                        INSERT INTO evento (id, data, nome)
                        VALUES (:1, TO_DATE(:2, 'YYYY-MM-DD'), :3)
                    """, row)
                except Exception as e:
                    print(f"Errore nell'inserimento della riga {row}: {e}")
        with open(percorso_luogo, "r") as file:
            reader = csv.reader(file)
            next(reader)
            for row in reader:
                try:
                    cursor.execute("""
                        INSERT INTO luogo (cap, citta, nome, paese)
                        VALUES (:1, :2, :3, :4)
                    """, row)
                except Exception as e:
                    print(f"Errore nell'inserimento della riga {row}: {e}")
        with open(percorso_oggetto, "r") as file:
            reader = csv.reader(file)
            next(reader)
            for row in reader:
                try:
                    cursor.execute("""
                        INSERT INTO oggetti (nome, numeroSerie)
                        VALUES (:1, :2)
                    """, row)
                except Exception as e:
                    print(f"Errore nell'inserimento della riga {row}: {e}")
        conn.commit()
    except Exception as e:
        print(f"Errore durante il caricamento: {e}")
        conn.rollback()
print("Caricamento dati completato. (Oracle)")
```

4) oracleComplessita1: Funzione che esegue la query di complessità 1. Il risultato è il medesimo della query di Neo4j.

```
def oracleComplessita1(cursor, nome_tabella):  1 usage
    lista_tempo = []
    tempo_iniziale = 0
    somma_tempo = 0
    iterazioni = 31
    query = f"SELECT * FROM {nome_tabella}"
    for i in range(iterazioni):
        start_time = time.time()
        cursor.execute(query)
        result = cursor.fetchall()
        end_time = time.time()
        time_difference = (end_time - start_time) * 1000
        if tempo_iniziale == 0:
            tempo_iniziale = time_difference
        else:
            lista_tempo.append(time_difference)
            somma_tempo += time_difference
    media_tempo = somma_tempo / 30
    return result, tempo_iniziale, media_tempo, lista_tempo
```

5) oracleComplessita2: Funzione che esegue la query di complessità 2. Il risultato è il medesimo della query di Neo4j.

```
query = f"""
        SELECT p.*, r.*, n.*
        FROM {nome_tabella1} p
        JOIN {nome_relazione} r ON p.codiceFiscale = r.id_persona
        JOIN {nome_tabella2} n ON r.id_evento = n.id
    """
```

6) oracleComplessita3: Funzione che esegue la query di complessità 3. Il risultato è il medesimo della query di Neo4j.

```
query = f"""
            SELECT *
            FROM (
                SELECT
                    p.codiceFiscale,
                    p.nome,
                    COLLECT(r.id_evento) AS relazioni,
                    COLLECT(n.id) AS nodi_connessi,
                    COUNT(r.id_evento) AS num_relazioni
                FROM
                    persona p
                JOIN
                    persona_evento r ON p.codiceFiscale = r.id_persona
                JOIN
                    evento n ON r.id_evento = n.id
                GROUP BY
                    p.codiceFiscale, p.nome
                ORDER BY
                    COUNT(r.id_evento) DESC
            )"""
```

7) oracleComplessita4: Funzione che esegue la query di complessità 4. Il risultato è il medesimo della query di Neo4j.

```
query = f"""
                SELECT *
                FROM (
                    SELECT
                        p1.codiceFiscale AS codiceFiscale_p1,
                        p2.codiceFiscale AS codiceFiscale_p2,
                        COLLECT(DISTINCT r.tipo_relazione) AS tipi_relazioni,
                        COUNT(r.id_relazione) AS num_connessioni,
                        (
                            SELECT COUNT(*)
                            FROM persona_evento r_sub
                            WHERE r_sub.id_persona = p1.codiceFiscale
                        ) AS grado_medio
                    FROM
                        persona p1
                    JOIN
                        persona_evento r ON p1.codiceFiscale = r.id_persona
                    JOIN
                        evento n ON r.id_evento = n.id
                    JOIN
                        persona_evento r2 ON n.id = r2.id_evento
                    JOIN
                        persona p2 ON r2.id_persona = p2.codiceFiscale
                    WHERE
                        p1.codiceFiscale <> p2.codiceFiscale
                    GROUP BY
                        p1.codiceFiscale, p2.codiceFiscale
                    ORDER BY
                        COUNT(r.id_relazione) DESC
                )"""
```

Codice PL/SQL:

```
CREATE OR REPLACE PACKAGE DATABASE2.manageBase is

    PROCEDURE create_tables;

    PROCEDURE drop_tables;

END manageBase;
```

```
CREATE OR REPLACE PACKAGE BODY DATABASE2.manageBase IS

    PROCEDURE create_tables IS
    BEGIN
        -- Tabella persona
        EXECUTE IMMEDIATE 'CREATE TABLE persona (
        citta VARCHAR2(50) NOT NULL,
        codiceFiscale VARCHAR2(16) NOT NULL,
        cognome VARCHAR2(50) NOT NULL,
        eta NUMBER NOT NULL,
        nome VARCHAR2(50) NOT NULL,
        precedenti VARCHAR2(5) NOT NULL,
        ruolo VARCHAR2(10) NOT NULL,
        sesso VARCHAR2(10) NOT NULL,
        CONSTRAINT pk_persona PRIMARY KEY (codiceFiscale)
    )';

        -- Tabella evento
        EXECUTE IMMEDIATE 'CREATE TABLE evento (
        id NUMBER NOT NULL,
        data DATE NOT NULL,
        nome VARCHAR2(50) NOT NULL,
        CONSTRAINT pk_evento PRIMARY KEY (id)
    )';

    -- Tabella luogo
        EXECUTE IMMEDIATE 'CREATE TABLE luogo (
        cap NUMBER NOT NULL,
        citta VARCHAR2(50) NOT NULL,
        nome VARCHAR2(50) NOT NULL,
        paese VARCHAR2(50) NOT NULL,
        CONSTRAINT pk_luogo PRIMARY KEY (cap)
    )';

    -- Tabella oggetto
        EXECUTE IMMEDIATE 'CREATE TABLE oggetti (
        nome VARCHAR2(50) NOT NULL,
        numeroSerie NUMBER NOT NULL,
        CONSTRAINT pk_oggetto PRIMARY KEY (numeroSerie)
    )';
```

```
    -- Tabella persona_evento
        EXECUTE IMMEDIATE 'CREATE TABLE persona_evento (
            id_persona VARCHAR2(16),
            id_evento NUMBER,
            CONSTRAINT fk_persona FOREIGN KEY (id_persona) REFERENCES PERSONA(codiceFiscale),
            CONSTRAINT fk_evento FOREIGN KEY (id_evento) REFERENCES EVENTO(id)
        )';

    -- Tabella oggetto_evento
        EXECUTE IMMEDIATE 'CREATE TABLE oggetto_evento (
            id_oggetto NUMBER,
            id_evento NUMBER,
            CONSTRAINT fk_oggetto FOREIGN KEY (id_oggetto) REFERENCES OGGETTI(numeroSerie),
            CONSTRAINT fk_evento_oggetto FOREIGN KEY (id_evento) REFERENCES EVENTO(id)
        )';

            -- Tabella persona_oggetto
        EXECUTE IMMEDIATE 'CREATE TABLE persona_oggetto (
            id_persona VARCHAR2(16),
            id_oggetto NUMBER,
            CONSTRAINT fk_persona_oggetto FOREIGN KEY (id_persona) REFERENCES PERSONA(codiceFiscale),
            CONSTRAINT fk_oggetto_persona FOREIGN KEY (id_oggetto) REFERENCES OGGETTI(numeroSerie)
        )';
```

```
        -- Tabella evento_luogo
        EXECUTE IMMEDIATE 'CREATE TABLE evento_luogo (
            id_evento NUMBER,
            id_luogo NUMBER,
            CONSTRAINT fk_evento_luogo FOREIGN KEY (id_evento) REFERENCES EVENTO(id),
            CONSTRAINT fk_luogo FOREIGN KEY (id_luogo) REFERENCES LUOGO(cap)
        )';

    END create_tables;

    PROCEDURE drop_tables IS
    BEGIN
        -- Drop delle tabelle in ordine di dipendenza
        EXECUTE IMMEDIATE 'DROP TABLE evento_luogo CASCADE CONSTRAINTS';
        EXECUTE IMMEDIATE 'DROP TABLE persona_oggetto CASCADE CONSTRAINTS';
        EXECUTE IMMEDIATE 'DROP TABLE oggetto_evento CASCADE CONSTRAINTS';
        EXECUTE IMMEDIATE 'DROP TABLE persona_evento CASCADE CONSTRAINTS';
        EXECUTE IMMEDIATE 'DROP TABLE oggetti CASCADE CONSTRAINTS';
        EXECUTE IMMEDIATE 'DROP TABLE luogo CASCADE CONSTRAINTS';
        EXECUTE IMMEDIATE 'DROP TABLE evento CASCADE CONSTRAINTS';
        EXECUTE IMMEDIATE 'DROP TABLE persona CASCADE CONSTRAINTS';
    END drop_tables;


END manageBase;
```

```
CREATE OR REPLACE PACKAGE DATABASE2.relazioni is

    PROCEDURE crea_relazioni_persona_evento;

    PROCEDURE crea_relazioni_persona_oggetto;

    PROCEDURE crea_relazioni_oggetto_evento;

    PROCEDURE crea_relazioni_evento_luogo;

END relazioni;
```

```
CREATE OR REPLACE PACKAGE BODY DATABASE2.relazioni IS

    PROCEDURE crea_relazioni_persona_evento IS
    -- Cursor per tutte le persone
    CURSOR cur_persona IS
        SELECT codiceFiscale FROM persona;
    -- Cursor per tutti gli eventi
    CURSOR cur_evento IS
        SELECT id FROM evento;

    -- Variabili per il loop
    v_codiceFiscale persona.codiceFiscale%TYPE;
    v_evento_id evento.id%TYPE;

    -- Contatore delle righe
    v_row_count_persona INTEGER := 0;
    v_row_count_evento INTEGER := 0;
    v_percentuale FLOAT := 0.7;
    v_max_righe INTEGER;
BEGIN
    -- 1. Associa tutte le righe riga per riga
    OPEN cur_persona;
    OPEN cur_evento;

    LOOP
        FETCH cur_persona INTO v_codiceFiscale;
        FETCH cur_evento INTO v_evento_id;
```

```
        -- Se uno dei due cursori è terminato, esci dal loop
        EXIT WHEN cur_persona%NOTFOUND OR cur_evento%NOTFOUND;

        -- Inserisci l'associazione
        INSERT INTO persona_evento (id_persona, id_evento)
        VALUES (v_codiceFiscale, v_evento_id);
    END LOOP;

    CLOSE cur_persona;
    CLOSE cur_evento;

    -- 2. Associa il 70% delle persone casualmente
    -- Calcola il numero massimo di righe da associare (70%)
    SELECT COUNT(*) INTO v_row_count_persona FROM persona;
    v_max_righe := TRUNC(v_row_count_persona * v_percentuale);

    OPEN cur_persona;
    OPEN cur_evento;

    -- Loop per il 70% delle righe
    FOR i IN 1..v_max_righe LOOP
        FETCH cur_persona INTO v_codiceFiscale;
        FETCH cur_evento INTO v_evento_id;

        -- Se uno dei due cursori è terminato, esci dal loop
        EXIT WHEN cur_persona%NOTFOUND OR cur_evento%NOTFOUND;

        -- Inserisci l'associazione
        INSERT INTO persona_evento (id_persona, id_evento)
        VALUES (v_codiceFiscale, v_evento_id);
    END LOOP;

    CLOSE cur_persona;
    CLOSE cur_evento;
```

```
-- Conferma delle modifiche
COMMIT;

END crea_relazioni_persona_evento;


PROCEDURE crea_relazioni_persona_oggetto IS
-- Cursor per tutte le persone
CURSOR cur_persona IS
    SELECT codiceFiscale
    FROM persona
    WHERE ruolo = 'Criminale';
-- Cursor per tutti gli eventi
CURSOR cur_oggetto IS
    SELECT numeroSerie FROM oggetti;

-- Variabili per il loop
v_codiceFiscale persona.codiceFiscale%TYPE;
v_oggetto_id oggetti.numeroSerie%TYPE;
```

```
BEGIN
    -- 1. Associa tutte le righe riga per riga
    OPEN cur_persona;
    OPEN cur_oggetto;

    LOOP
        FETCH cur_persona INTO v_codiceFiscale;
        FETCH cur_oggetto INTO v_oggetto_id;

        -- Se uno dei due cursori è terminato, esci dal loop
        EXIT WHEN cur_persona%NOTFOUND OR cur_oggetto%NOTFOUND;

        -- Inserisci l'associazione
        INSERT INTO persona_oggetto (id_persona, id_oggetto)
        VALUES (v_codiceFiscale, v_oggetto_id);
    END LOOP;

    CLOSE cur_persona;
    CLOSE cur_oggetto;

    -- Conferma delle modifiche
    COMMIT;

    END crea_relazioni_persona_oggetto;
```

```
BEGIN
    -- 1. Associa tutte le righe riga per riga
    OPEN cur_persona;
    OPEN cur_oggetto;

    LOOP
        FETCH cur_persona INTO v_codiceFiscale;
        FETCH cur_oggetto INTO v_oggetto_id;

        -- Se uno dei due cursori è terminato, esci dal loop
        EXIT WHEN cur_persona%NOTFOUND OR cur_oggetto%NOTFOUND;

        -- Inserisci l'associazione
        INSERT INTO persona_oggetto (id_persona, id_oggetto)
        VALUES (v_codiceFiscale, v_oggetto_id);
    END LOOP;

    CLOSE cur_persona;
    CLOSE cur_oggetto;

    -- Conferma delle modifiche
    COMMIT;

    END crea_relazioni_persona_oggetto;
```

```
BEGIN
    -- 1. Associa tutte le righe riga per riga
    OPEN cur_evento;
    OPEN cur_oggetto;

    LOOP
        FETCH cur_evento INTO v_id;
        FETCH cur_oggetto INTO v_oggetto_id;

        -- Se uno dei due cursori è terminato, esci dal loop
        EXIT WHEN cur_evento%NOTFOUND OR cur_oggetto%NOTFOUND;

        -- Inserisci l'associazione
        INSERT INTO oggetto_evento (id_oggetto, id_evento)
        VALUES (v_oggetto_id, v_id);
    END LOOP;

    CLOSE cur_evento;
    CLOSE cur_oggetto;

    -- Conferma delle modifiche
    COMMIT;

    END crea_relazioni_oggetto_evento;
```

```
    PROCEDURE crea_relazioni_evento_luogo IS
    -- Cursor per tutte le persone
    CURSOR cur_evento IS
        SELECT id FROM evento;
    -- Cursor per tutti gli eventi
    CURSOR cur_luogo IS
        SELECT cap FROM luogo;

    -- Variabili per il loop
    v_id evento.id%TYPE;
    v_luogo_id luogo.cap%TYPE;

    -- Contatore delle righe
    v_row_count_evento INTEGER := 0;
    v_row_count_luogo INTEGER := 0;
    v_percentuale FLOAT := 0.7;
    v_max_righe INTEGER;
BEGIN
    -- 1. Associa tutte le righe riga per riga
    OPEN cur_evento;
    OPEN cur_luogo;

    LOOP
        FETCH cur_evento INTO v_id;
        FETCH cur_luogo INTO v_luogo_id;

        -- Se uno dei due cursori è terminato, esci dal loop
        EXIT WHEN cur_evento%NOTFOUND OR cur_luogo%NOTFOUND;

        -- Inserisci l'associazione
        INSERT INTO evento_luogo (id_evento, id_luogo)
        VALUES (v_id, v_luogo_id);
    END LOOP;

    CLOSE cur_evento;
    CLOSE cur_luogo;
```

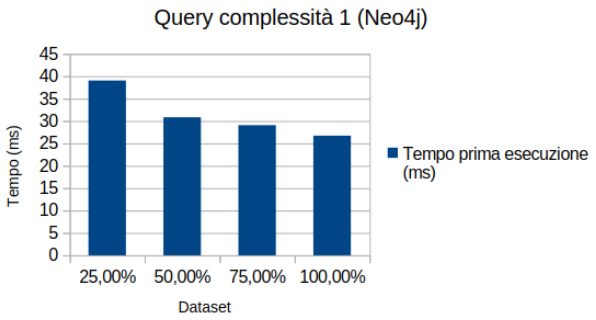
```
    -- 2. Associa il 70% delle persone casualmente
    -- Calcola il numero massimo di righe da associare (70%)
    SELECT COUNT(*) INTO v_row_count_evento FROM evento;
    v_max_righe := TRUNC(v_row_count_evento * v_percentuale);

    OPEN cur_evento;
    OPEN cur_luogo;

    -- Loop per il 70% delle righe
    FOR i IN 1..v_max_righe LOOP
        FETCH cur_evento INTO v_id;
        FETCH cur_luogo INTO v_luogo_id;

        -- Se uno dei due cursori è terminato, esci dal loop
        EXIT WHEN cur_evento%NOTFOUND OR cur_luogo%NOTFOUND;

        -- Inserisci l'associazione
        INSERT INTO evento_luogo (id_evento, id_luogo)
        VALUES (v_id, v_luogo_id);
    END LOOP;

    CLOSE cur_evento;
    CLOSE cur_luogo;

    -- Conferma delle modifiche
    COMMIT;

    END crea_relazioni_evento_luogo;


END relazioni;
```

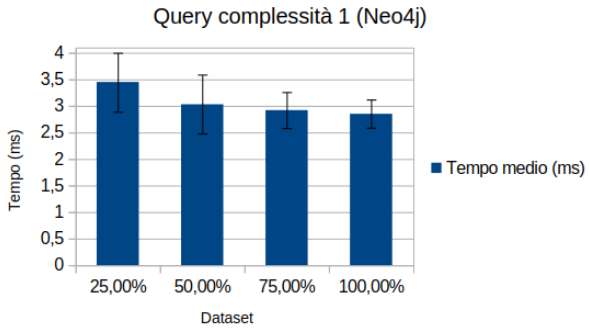
5. Esperimenti (contenenti tabelle con i tempi di risposta ottenuti e relativi istogrammi)

Neo4j

Complessità 1

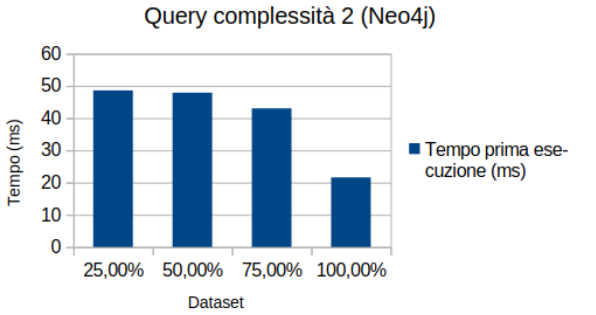
Dataset (Neo4j)	Tempo prima esecuzione (ms)	Tempo medio (ms)	Intervallo di confidenza (ms)	Errore superiore (ms)	Errore inferiore (ms)
25,00%	39,04	3,45	2,89 – 4,00	0,55	0,56
50,00%	30,84	3,03	2,48 – 3,59	0,56	0,55
75,00%	29,07	2,92	2,58 – 3,26	0,34	0,34
100,00%	26,72	2,85	2,59 – 3,12	0,27	0,26

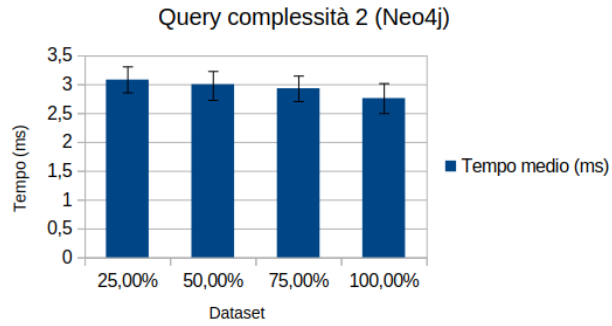




Complessità 2

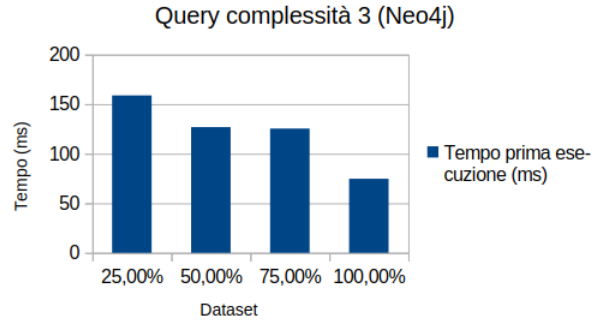
Dataset (Neo4j)	Tempo prima esecuzione (ms)	Tempo medio (ms)	Intervallo di confidenza (ms)	Errore superiore (ms)	Errore inferiore (ms)
25,00%	48,6	3,08	2,86 – 3,31	0,23	0,22
50,00%	47,9	3	2,73 – 3,26	0,23	0,27
75,00%	43,03	2,93	2,71 – 3,15	0,22	0,22
100,00%	21,6	2,76	2,50 – 3,02	0,26	0,26

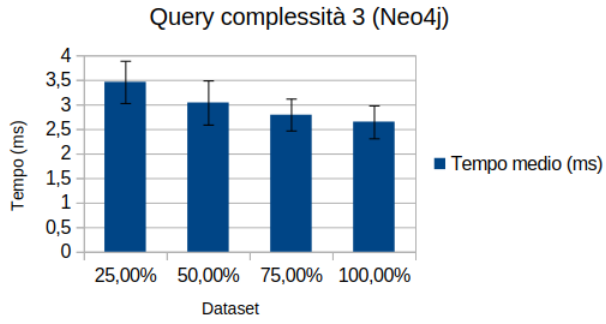




Complessità 3

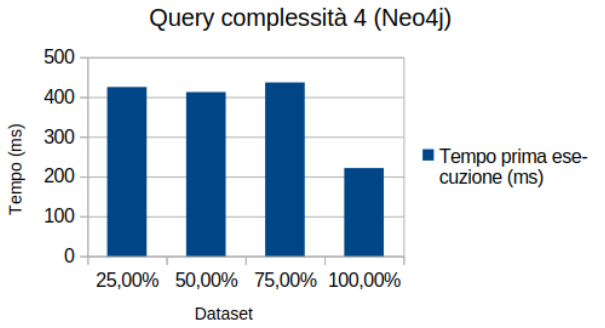
Dataset (Neo4j)	Tempo prima esecuzione (ms)	Tempo medio (ms)	Intervallo di confidenza (ms)	Errore superiore (ms)	Errore inferiore (ms)
25,00%	158,78	3,46	3,03 – 3,89	0,43	0,43
50,00%	126,78	3,04	2,59 – 3,49	0,45	0,45
75,00%	125,33	2,79	2,47 – 3,12	0,33	0,32
100,00%	74,68	2,65	2,31 – 2,98	0,33	0,34

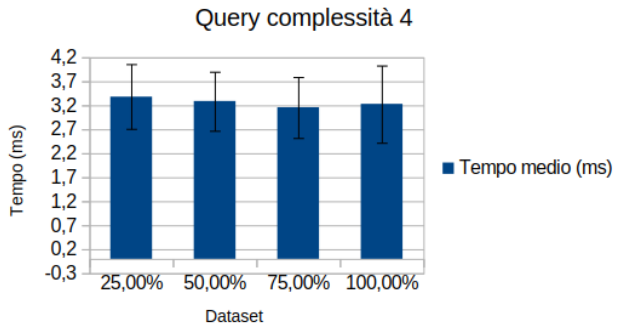




Complessità 4

Dataset (Neo4j)	Tempo prima esecuzione (ms)	Tempo medio (ms)	Intervallo di confidenza (ms)	Errore superiore (ms)	Errore inferiore (ms)
25,00%	424,94	3,38	2,71 – 4,06	0,68	0,67
50,00%	412,19	3,29	2,67 – 3,90	0,61	0,62
75,00%	436,37	3,16	2,52 – 3,79	0,63	0,64
100,00%	221,22	3,23	2,42 – 4,03	0,8	0,81

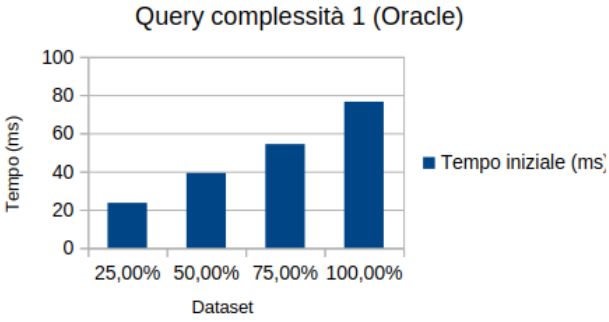


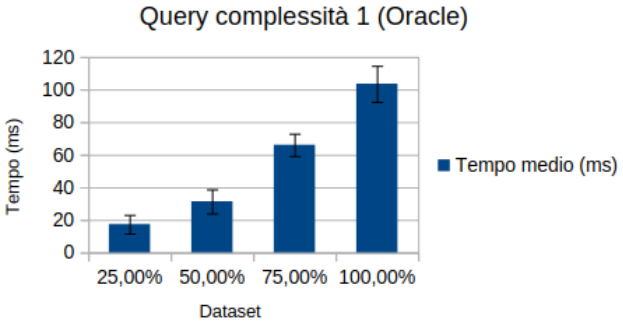


Oracle

Complessità 1

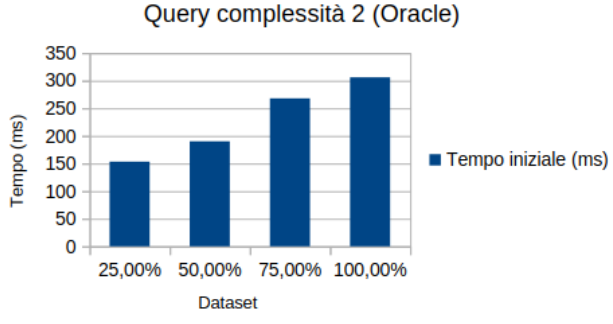
Dataset (Oracle)	Tempo iniziale (ms)	Tempo medio (ms)	Intervallo di confidenza (ms)	Errore superiore (ms)	Errore inferiore (ms)
25,00%	23,56	17,37	11,66 – 23,09	5,72	5,71
50,00%	39,21	31,37	24,01 – 38,73	7,36	7,36
75,00%	54,28	66,06	59,19 – 72,92	6,86	6,87
100,00%	76,46	103,52	92,46 – 114,58	11,06	11,06

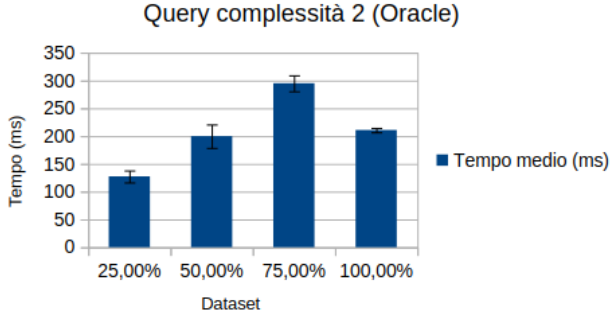




Complessità 2

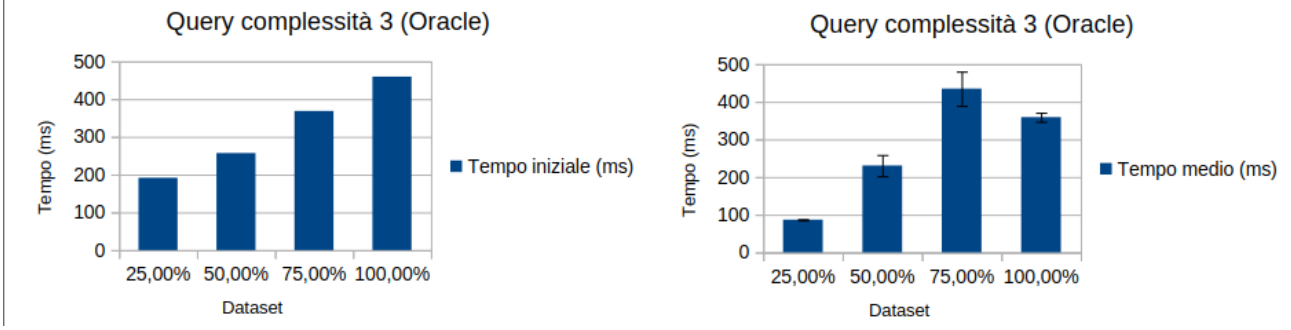
Dataset (Oracle)	Tempo iniziale (ms)	Tempo medio (ms)	Intervallo di confidenza (ms)	Errore superiore (ms)	Errore inferiore (ms)
25,00%	153,35	127,44	116,60 – 138,28	10,84	10,84
50,00%	190,2	199,9	178,87 – 220,93	21,03	21,03
75,00%	267,92	295,09	280,80 – 309,39	14,3	14,29
100,00%	306,05	210,86	206,85 – 214,88	4,02	4,01





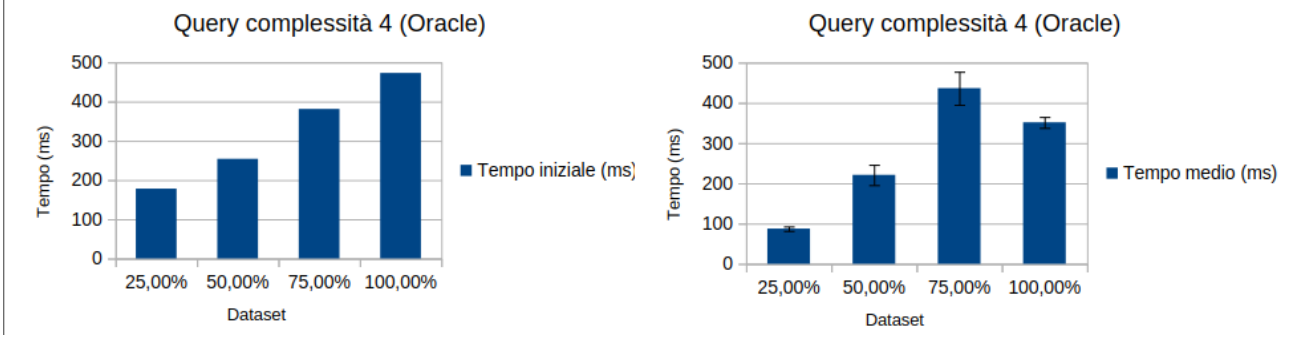
Complessità 3

Dataset (Oracle)	Tempo iniziale (ms)	Tempo medio (ms)	Intervallo di confidenza (ms)	Errore superiore (ms)	Errore inferiore (ms)
25,00%	191,36	86,82	85,27 – 88,36	1,54	1,55
50,00%	256,95	230,57	202,31 – 258,84	28,27	28,26
75,00%	368,47	434,77	389,37 – 480,18	45,41	45,4
100,00%	459,32	358,98	347,05 – 370,91	11,93	11,93



Complessità 4

Dataset (Oracle)	Tempo iniziale (ms)	Tempo medio (ms)	Intervallo di confidenza (ms)	Errore superiore (ms)	Errore inferiore (ms)
25,00%	177,88	87,4	81,96 – 92,84	5,44	5,44
50,00%	253,92	220,97	195,72 – 246,22	25,25	25,25
75,00%	381,06	436,42	395,39 – 477,44	41,02	41,03
100,00%	472,8	351,49	337,99 – 364,98	13,49	13,5



6. Conclusioni

Dall'analisi dei risultati emerge che Neo4j e Oracle presentano comportamenti diversi rispetto alla crescita del dataset. Neo4j mantiene tempi di risposta costanti grazie alla sua struttura a grafo, che ottimizza l'elaborazione delle relazioni senza la necessità di costosi join. Al contrario, Oracle evidenzia un incremento dei tempi di esecuzione al crescere del dataset, riflettendo le limitazioni del modello relazionale per query focalizzate su relazioni complesse.

Tuttavia, è stato osservato che Neo4j impiega più tempo rispetto a Oracle nella prima esecuzione delle query. Questo comportamento è dovuto al meccanismo di caching di Neo4j: durante la prima esecuzione, il sistema carica in memoria i dati necessari, costruendo strutture ottimizzate per le successive interrogazioni. In Oracle, invece, la prima esecuzione risulta più veloce poiché il sistema accede direttamente ai dati senza un processo di caching altrettanto elaborato. Con l'aumentare del dataset, però, Oracle subisce un rallentamento progressivo, poiché l'elaborazione di join complessi e scansioni su tabelle di grandi dimensioni diventa sempre più onerosa.

L'analisi degli intervalli di confidenza riportati negli istogrammi evidenzia inoltre un'importante distinzione tra i due DBMS. Nel caso di Neo4j, l'intervallo di confidenza per i tempi di esecuzione delle query successive alla prima è significativamente ristretto, indicando una maggiore stabilità e prevedibilità delle prestazioni. Oracle, invece, presenta un intervallo di confidenza più ampio, in

particolare per dataset di grandi dimensioni, riflettendo una maggiore variabilità nei tempi di risposta. Questa variabilità può essere attribuita a fattori come la complessità dei join, l'ottimizzazione del piano di query, e il carico di lavoro del sistema al momento dell'esecuzione.

In termini di risorse hardware, Neo4j beneficia di un utilizzo ottimizzato della memoria, mantenendo in cache le pagine di dati più frequenti per ridurre il carico su disco e migliorare le prestazioni delle esecuzioni successive. Questo lo rende particolarmente efficiente per dataset moderatamente grandi e query locali su grafi complessi. Tuttavia, per dataset enormi o ambienti distribuiti, potrebbe essere necessario scalare orizzontalmente, aumentando il numero di nodi del cluster. Oracle, d'altra parte, richiede infrastrutture più potenti per gestire dataset in crescita, specialmente quando si eseguono query che coinvolgono molteplici tabelle e join complessi. La scalabilità verticale, tipica dei sistemi relazionali, implica costi più elevati in termini di risorse hardware.

Neo4j si dimostra quindi ideale per scenari in cui le relazioni sono il focus principale, come analisi di reti sociali o criminali, e dove le prestazioni dopo la prima esecuzione delle query sono critiche. Oracle, invece, rimane una scelta robusta per gestire grandi volumi di dati transazionali e contesti aziendali con requisiti diversificati. La valutazione degli intervalli di confidenza sottolinea che Neo4j offre prestazioni più prevedibili, mentre Oracle necessita di ulteriori ottimizzazioni per garantire una maggiore stabilità, soprattutto in ambienti ad alto carico.