Relazione Basi di dati NoSQL

Fabiana Presti

June 2025

1 Introduzione

In un'epoca in cui il riciclaggio di denaro ha assunto una nota rilevanza, un intricato metodo di money laundering prevede che una persona possa mascherarsi e assumere identità fittizie, evitando ripercussioni personali e massimizzando il suo profitto. Il caso di studio punta all'identificazione di fonti, documenti, transazioni e persone sospette che possano fare parte di un quadro più complesso.

2 Database utilizzati

Sono stati utilizzati i database Neo4J e BaseX, il primo ottimizzato per la gestione di dati complessi e il secondo eccelle nelle ricerche lineari e dirette.

2.1 Neo4J

Neo4J è un graph database. I grafi consentono di poter andare a implementare algoritmi e casi di studio complessi, ma non sono semplici da gestire. Neo4j ci dà la possibilità di gestire un database con molte relazioni (orientato al grafo) e di realizzare delle query complesse direttamente a livello database, senza l'overhead di dover trasferire i dati a livello applicativo per analizzarli. Un vantaggio di Neo4j, essendo un database distribuito e parallelo, offre la possibilità di scalare il sistema, quindi di installare Neo4j in maniera distribuita. Inoltre, Neo4j ha un'interfaccia client sia da console che interattiva che ci consente di visualizzare i dati in varie forme: possiamo rappresentare i nostri dati sia graficamente, come un grafo, che sotto forma di documenti JSON o sotto forma tabellare. Le sue prestazioni migliorano con l'aumentare di dati relazionati. Ogni database a grafo memorizza qualsiasi dato utilizzando questi semplici concetti:

- Nodi: rappresentano i nostri record;
- Relazioni: archi che connettono i vari nodi;
- Proprietà: coppie chiave valore che possono essere associate sia ai nodi che alle relazioni

Il linguaggio di querying di neo4J è Cypher. Cypher utilizza dei patterns per descrivere i dati del grafo, quindi ha una sintassi familiare per certi punti di vista ad SQL ed è un linguaggio dichiarativo che consente di descrivere cosa vogliamo fare e come.

2.2 BaseX

BaseX è un motore di database *nativo XML*, open source, progettato per l'archiviazione, la gestione e l'interrogazione efficiente di dati strutturati nel formato XML. Sviluppato in Java, BaseX fornisce un ambiente completo per lavorare con documenti XML.

BaseX eccelle nelle ricerche lineari con strutture gerarchizzate.

XQuery, il linguaggio di querying supportato da BaseX, si ispira ad SQL per l'interrogazione di dati XML. XQuery si basa su 5 espressioni fondamentali, dette FLWOR: -

- For: per iterare i valori di variabili su sequenze di nodi;
- Let: per legare variabili a intere sequenze di nodi;
- Where: per esprimere condizioni, si applica un filtro;
- Order by: per imporre un ordinamento sulla sequenza risultante

3 Progettazione

3.1 Schema E-R

Per questo esperimento è stata definita la struttura seguente:

- Persone: l'insieme di lavoratori protagonisti del caso di studio. Sono presenti dei duplicati: le possibili identità fittizie.
- Fonti: società o aziende. Hanno un indice di affidabilità che determina la reputazione della fonte.
- Documenti: i dati che definiscono l'identità della persona alla quale sono associati. I documenti condivisi da più persone indicano un sospetto.
- Transazioni: le transazioni che vengono effettuate dalle persone che possono permettere di risalire a riciclaggio di denaro
- Banche: le banche hanno un massimale prestabilito oltre il quale non è possibile effettuare transazioni

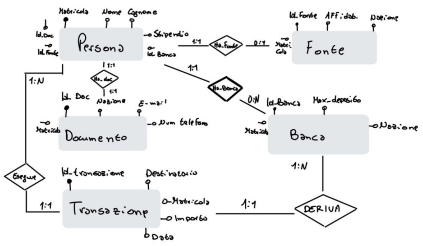

Figure 1: schema E-R

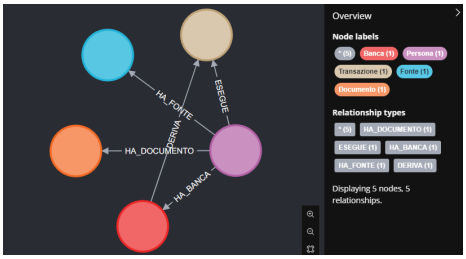

Figure 2: neo4J schema

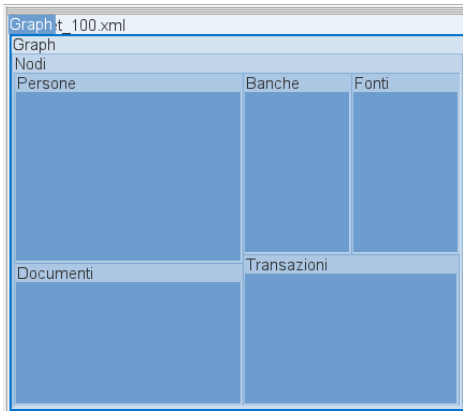

Figure 3: BaseX schema

Sono state stabilite delle relazioni:

- DERIVA DA: Lega Documento e Fonte. Si tratta di una relazione 1:1.

- HA FONTE: Lega Persona e Fonte. Una persona può avere più fonti, ma una fonte deve essere legata a una sola persona.
- HA DOCUMENTO: Lega Persona e Documento. Una persona può avere più documenti, ma un documento deve essere legato a una sola persona.
- HA BANCA: la relazione che collega una persona alla sua banca.
- ESEGUE: La relazione che permette alle persone di eseguire transazioni.

3.2 Query

Riporto le 4 query:

- Trovare banche residenti in germania
- Trovare fonti con poca affidabilità che iniziano con p
- Contare email ripetute nei documenti
- Trovare persone che hanno ricevuto in entrata più di quanto permette la loro banca in un giorno solo

3.3 Struttura dati csv

Si contano 100.000 elementi per ogni entità (500.000). I file csv hanno la struttura seguente:

```
matricola:ID,nome,cognome,stipendio:INT,id_banca,id_documento,id_fonte,:LABEL
id_transazione:ID,matricola,importo:INT,destinatario,data:DATE,id_banca_deriva,:LABEL
...
```

3.4 Generazione dataset

Il dataset è stato generato prima attraverso la libreria faker in formato csv e successivamente convertito in XML.

3.4.1 Generazione automatica delle Fonti con Python

Per la generazione automatica dei dati relativi alle banche, è stato utilizzato uno script Python che sfrutta la libreria `Faker` per creare contenuti fittizi e realistici. Lo snippet seguente mostra la procedura utilizzata:

```
banche = {
    bid: {
        "id_banca": bid,
        "nome": fake.company() + " Bank",
        "nazione": fake.country_code(),
        "max_deposito": random.randint(1500, 10000)
```

```
    }
    for bid in banche_ids
}
```

Questa procedura verrà utilizzata anche per la generazione di Persone, Fonti, Documenti e Transazioni.

Sono successivamente stati inseriti alcuni dati anomali come documenti che registrano lo stesso numero di telefono o la stessa e-mail insieme alle persone che hanno ricevuto denaro che superi il massimale consentito dalla loro banca da persone che condividono i suddetti campi.

```
    #per generare qualche duplicato "realistico"
emails_pool   = [fake.email() for _ in range(2000)]
phones_pool   = [fake.phone_number() for _ in range(2000)]

email = random.choice(emails_pool) if random.random() < 0.1 else fake.email()
    phone = random.choice(phones_pool) if random.random() < 0.1 else fake.phone_number()


#per le transazioni sospette:
for destinatario in persone_dest_sospette:
    if len(transazioni) >= NUM_TRANS:
        break
    ...
    add_tx(m, dest_matr, base, day)

#aggiunge destinatari sospetti:
NUM_PATTERN_SOSPETTI = min(10_000, int(NUM_PERSONE * 0.1))
persone_dest_sospette = random.sample(persone, NUM_PATTERN_SOSPETTI)
```

Pool di email e telefoni con duplicati realistici

- Si costruiscono due pool di valori riutilizzabili:

  $\texttt{emails\_pool} = [\texttt{fake.email()}]_{2000}, \qquad \texttt{phones\_pool} = [\texttt{fake.phone\_number()}]_{2000}.$

- Per ogni documento si decide se creare un duplicato o un valore nuovo:
  - con probabilità 0.1 (10%) si estrae *uniformemente* un valore dal pool (`random.choice(...)`), introducendo così ripetizioni plausibili;
  - con probabilità 0.9 si genera un valore sintetico inedito con `Faker`.
- Risultato: il dataset contiene cluster di email/telefoniche ripetute per test di identity resolution.

Selezione dei destinatari "sospetti"

- Si definisce la quota di persone da rendere anomale:

$$\texttt{NUM\_PATTERN\_SOSPETTI} = \min(10{,}000,\ \lfloor 0{,}1 \cdot \texttt{NUM\_PERSONE} \rfloor).$$

- Si campionano senza rimpiazzo `NUM_PATTERN_SOSPETTI` persone da `persone`:

  `persone_dest_sospette` ← `random.sample(persone, NUM_PATTERN_SOSPETTI)`.

- Queste costituiranno i destinatari per cui verranno create transazioni anomale in un singolo giorno.

Inserimento delle transazioni sospette

- Per ogni `destinatario` in `persone_dest_sospette`:
  1. Si verifica di non aver già raggiunto il budget totale di transazioni (`NUM_TRANS`); in tal caso si interrompe il ciclo.
  2. Si calcolano importi e giorno condiviso in modo che la *somma giornaliera* delle entrate del destinatario superi il `max_deposito` della sua banca.
  3. Si invoca `add_tx(m, dest_matr, base, day)` più volte (da mittenti diversi) per creare le transazioni con stesso giorno e importo sufficiente a far superare la soglia.
- Effetto: per ciascun destinatario sospetto, la somma degli **importi ricevuti nello stesso giorno** eccede il limite della banca, generando un pattern rilevabile.

3.5 Subset

Per la creazione dei dataset a dimensione ridotta (25%, 50% e 75%), è stato utilizzato il seguente script Python, che si occupa di caricare i dati originali completi e salvare versioni parziali mantenendo la coerenza tra le entità.

```
#sample Persone
   #PERSONE
   k_people = int(round(len(dfp) * fraction)) #calcola quante persone mettere nel subset in
   people_ids = set(people_order[:k_people]) #lo converte in un set
   dfp_sub = dfp[dfp["matricola:ID"].isin(people_ids)].copy() #copia gli id delle persone n

   #Documenti delle persone
   dfd_sub = dfd[dfd["matricola"].isin(people_ids)].copy()

   #transazioni
   dft_sub = transazioni_subset(dft, people_ids, mode)
```

```
...

 #BANCHE usate da persone/transazion
bank_ids_people = set(dfp_sub["id_banca"]) #crea set con id delle banche
bank_ids_tx = set(dft_sub["id_banca_deriva"]) if "id_banca_deriva" in dft_sub.columns el
bank_ids_all = bank_ids_people | {x for x in bank_ids_tx if isinstance(x, str) and x !=
dfb_sub = dfb[dfb["id_banca:ID"].isin(bank_ids_all)].copy()
...

# + ORFANE (prefisso dell'ordine orfane ->annidate)
k_banks = int(round(len(orphan_banks_order) * fraction))
if k_banks > 0:
    orphan_bank_ids = set(orphan_banks_order[:k_banks])
    dfb_sub = (pd.concat([dfb_sub, dfb[dfb["id_banca:ID"].isin(orphan_bank_ids)]], ignor
                .drop_duplicates(subset=["id_banca:ID"], keep="first"))
```

Lo script esegue le seguenti operazioni:

- Seleziona una percentuale di persone
- Importa le altre entità ad esse collegate
- riempie i buchi con altre percentuali di banche e fonti rimaste 'orfane'

Ogni subset verrà convertito in un nuovo dataset xml, per esempio questo snippet riguarda le persone:

```
]
write_open(fout, "Persone")
        with open_csv(os.path.join(INPUT_DIR, "persone.csv")) as fin:
            r = csv.DictReader(fin)
            for row in r:
                mid = row.get("matricola:ID")
                if not mid:
                    continue
                fout.write(
                    '  <Persona '
                    f'matricola="{esc(mid)}" '
                    f'stipendio="{esc(row.get("stipendio:INT"))}">\n'
                )
                fout.write(f'    <Nome>{esc(row.get("nome"))}</Nome>\n')
                fout.write(f'    <Cognome>{esc(row.get("cognome"))}</Cognome>\n')
                fout.write(f'    <BancaRef id="{esc(row.get("id_banca"))}"/>\n')
                fout.write(f'    <DocumentoRef id="{esc(row.get("id_documento"))}"/>\n')
                fid = row.get("id_fonte") or row.get("id_fonte:ID")
```

```
            fout.write(f'    <FonteRef id="{esc(fid)}"/>\n')
            fout.write("  </Persona>\n")
    write_close(fout, "Persone")
```

Questo frammento di codice svolge le seguenti operazioni:

- Scrive il tag di apertura `<Persone>` nel file XML di output.
- Apre il file CSV delle persone (`persone.csv`) e prepara un `DictReader` per leggere ogni riga come dizionario.
- Per ogni riga del CSV:
  - Recupera la matricola dalla colonna `matricola:ID`; se manca, salta la riga.
  - Scrive un elemento `<Persona>` con attributi:
    - `matricola` (valore della colonna `matricola:ID`),
    - `stipendio` (valore della colonna `stipendio:INT`).
  - All'interno della persona scrive sottoelementi:
    - `<Nome>` con il nome della persona,
    - `<Cognome>` con il cognome,
    - `<BancaRef>` con attributo `id` che punta alla banca collegata,
    - `<DocumentoRef>` con attributo `id` che punta al documento collegato,
    - `<FonteRef>` con attributo `id` che punta alla fonte collegata.
  - Chiude il tag `</Persona>`.
- Dopo aver elaborato tutte le righe, scrive il tag di chiusura `</Persone>`.

4 Implementazione

4.1 Caricamento dei dati in neo4j

Sono stati creati 4 database: IR 25, 50, 75 e 100. Accedendo alla cartella import del database creato e copiando i file csv è stato possibile caricare i dataset tramite l'interfaccia del browser attraverso questi comandi:

```
CREATE CONSTRAINT persona_pk IF NOT EXISTS
FOR (p:Persona) REQUIRE p.matricola IS UNIQUE;

CREATE CONSTRAINT documento_pk IF NOT EXISTS
FOR (d:Documento) REQUIRE d.id_documento IS UNIQUE;

CREATE CONSTRAINT banca_pk IF NOT EXISTS
FOR (b:Banca) REQUIRE b.id_banca IS UNIQUE;
...
```

Le prime tre istruzioni definiscono dei vincoli di unicità per i primi 3 nodi:

- Ogni nodo `Persona` deve avere un attributo `matricola` univoco;
- Ogni nodo `Documento` deve avere un attributo `id_documento` univoco;
- Ogni nodo `Banca` deve avere un attributo `id_banca` univoco.

```
// PERSONE
LOAD CSV WITH HEADERS FROM 'file:///persone.csv' AS row
MERGE (p:Persona {matricola: row.`matricola:ID`})
SET p.nome = row.nome,
    p.cognome = row.cognome,
    p.stipendio = toInteger(row.`stipendio:INT`),
    p.id_banca = row.id_banca,
    p.id_documento = row.id_documento,
    p.id_fonte = row.id_fonte;

    ...
```

- Legge il file CSV `persone.csv` usando `LOAD CSV WITH HEADERS`, cioè considera la prima riga come intestazione.
- Per ogni riga (`row`):
  - Cerca un nodo con etichetta `Persona` che abbia proprietà `matricola = row.`matricola:ID``.
  - Se non esiste, lo crea (`MERGE`).
  - Aggiorna/imposta le proprietà del nodo `Persona`:
    - `nome` ← valore della colonna `nome`;
    - `cognome` ← valore della colonna `cognome`;
    - `stipendio` ← conversione a intero di `stipendio:INT`;
    - `id_banca` ← valore della colonna `id_banca`;
    - `id_documento` ← valore della colonna `id_documento`;
    - `id_fonte` ← valore della colonna `id_fonte`.
- In questo modo, ogni riga del CSV corrisponde a un nodo `Persona` con tutte le proprietà valorizzate.

Riporto anche le relazioni:

```
// (Persona)-[:HA_DOCUMENTO]->(Documento)
LOAD CSV WITH HEADERS FROM 'file:///persone.csv' AS row
MATCH (p:Persona {matricola: row.`matricola:ID`})
MATCH (d:Documento {id_documento: row.id_documento})
MERGE (p)-[:HA_DOCUMENTO]->(d);
```

```
// (Persona)-[:HA_BANCA]->(Banca)
LOAD CSV WITH HEADERS FROM 'file:///persone.csv' AS row
MATCH (p:Persona {matricola: row.`matricola:ID`})
MATCH (b:Banca {id_banca: row.id_banca})
MERGE (p)-[:HA_BANCA]->(b);

// (Persona)-[:HA_FONTE]->(Fonte)
LOAD CSV WITH HEADERS FROM 'file:///persone.csv' AS row
MATCH (p:Persona {matricola: row.`matricola:ID`})
MATCH (f:Fonte {id_fonte: row.id_fonte})
MERGE (p)-[:HA_FONTE]->(f);

// (Persona)-[:ESEGUE]->(Transazione)  (mittente -> transazione)
LOAD CSV WITH HEADERS FROM 'file:///transazioni.csv' AS row
MATCH (p:Persona {matricola: row.matricola})
MATCH (t:Transazione {id_transazione: row.`id_transazione:ID`})
MERGE (p)-[:ESEGUE]->(t);

// (Banca)-[:DERIVA]->(Transazione)  (NUOVO legame banca di provenienza)
LOAD CSV WITH HEADERS FROM 'file:///transazioni.csv' AS row
MATCH (b:Banca {id_banca: row.id_banca_deriva})
MATCH (t:Transazione {id_transazione: row.`id_transazione:ID`})
MERGE (b)-[:DERIVA]->(t);
```

- **(Persona)–[:HA_DOCUMENTO]→(Documento)**: per ogni riga di `persone.csv` collega la persona (per `matricola`) al suo documento (per `id_documento`) creando la relazione se manca (`MERGE`).
- **(Persona)–[:HA_BANCA]→(Banca)**: dalla stessa riga collega la persona alla banca associata (per `id_banca`) con una relazione `HA_BANCA`.
- **(Persona)–[:HA_FONTE]→(Fonte)**: collega la persona alla fonte di provenienza/registrazione (per `id_fonte`) tramite `HA_FONTE`.
- **(Persona)–[:ESEGUE]→(Transazione)**: da `transazioni.csv` collega il mittente (persona con `matricola`) alla transazione (`id_transazione`) indicando chi l'ha eseguita.
- **(Banca)–[:DERIVA]→(Transazione)**: da `transazioni.csv` collega la banca di provenienza (`id_banca_deriva`) alla transazione, marcando l'origine bancaria del movimento.

4.2 Creazione database in BaseX

BaseX permette di creare un nuovo database selezionando un file XML. Per avviare il server, si digita "basexserver" nel cmd di Windows. Ho così creato 4

database per ciascuna dimensione del dataset originale.

4.3 Interrogazione del Database

Uno script python si è occupato di interrogare, testare e salvare i risultati delle query di entrambi i database attivi contemporaneamente:

```
    QUERIES = [
  {
        "name": "Query 1",
        "cypher": """
match (b:Banca{nazione:'GE'}) return b.nome;


""",
        "xquery": r'''
xquery version "3.1";

for $b in /Graph/Nodi/Banche/Banca[@nazione = 'GE']
return <nome>{ $b/Nome/text() }</nome>
)


'''
    },
```

Il codice python comprende una prima parte in cui vengono definite le query. Questa è la prima query definita attraverso Cypher e XQuery.

Lo script permette anche di calcolare l'intervallo di confidenza al 95Per ogni insieme di misurazioni di tempo (in millisecondi), viene calcolato l'intervallo:

```
def confidence_interval_95(times):
    n = len(times)
    if n < 2:
        return 0.0
    mean = statistics.mean(times)
    stdev = statistics.stdev(times)

    t = 2.045 if n == 30 else 1.96
    ci = t * (stdev / math.sqrt(n))
    return ci
```

La funzione accetta una lista `times` contenente i tempi di esecuzione di una query, e calcola l'intervallo di confidenza al 95% utilizzando la formula:

$$CI = t \cdot \frac{s}{\sqrt{n}}$$

dove:

- n è il numero di osservazioni;
- s è la deviazione standard campionaria;
- t è il valore critico della distribuzione t-Student (approssimato a 2.045 per $n = 30$ o 1.96 per $n > 30$).

Se il numero di osservazioni è minore di 2, la funzione restituisce `0.0`, poiché non è possibile calcolare una deviazione standard significativa.

Misurazione su BaseX

```
def measure_basex(xquery):
    times = []
    try:
        session = BaseXClient.Session(HOST, PORT, USERNAME, PASSWORD)
        session.execute(f"open {DATABASE}")


        for i in range(31):  # 1 warm-up + 30 misure
            start = time.perf_counter()
            session.execute(f"xquery {xquery}")
            end = time.perf_counter()
            times.append((end - start) * 1000.0)
    except Exception as e:
        try:
            session.close()
        except:
            pass
        return float("nan"), float("nan"), float("nan"), f"Errore BaseX: {e}"
    finally:
        try:
            session.close()
        except Exception:
            pass

    first = times[0]
    rest = times[1:] if len(times) > 1 else [times[0]]
    avg = sum(rest) / len(rest)
    ci = confidence_interval_95(rest)
    return first, avg, ci, None
```

- Apre una nuova sessione con il server BaseX (`BaseXClient.Session`) e seleziona il database specificato.
- Avvia un ciclo di 31 iterazioni:
  - Registra il tempo di inizio con `time.perf_counter()`.

  - Esegue l'XQuery tramite `session.execute`.
  - Registra il tempo di fine e calcola la durata in millisecondi.
  - Salva ciascun tempo nella lista `times`.
- Se avviene un'eccezione, chiude la sessione e ritorna valori `NaN` con un messaggio di errore.
- Alla fine chiude comunque la sessione (`finally`).
- Calcola:
  - **first**: tempo della prima esecuzione registrata;
  - **avg**: media delle restanti 30 esecuzioni;
  - **ci**: intervallo di confidenza al 95% sui 30 tempi (funzione `confidence_interval_95`).
- Ritorna una tupla con i tre valori statistici e `None` come segnalatore di assenza errori.

Misurazione su Neo4j

```
def measure_neo4j(cypher):
    times = []
    try:
        driver = GraphDatabase.driver(NEO4J_URI, auth=(NEO4J_USER, NEO4J_PASSWORD))
        with driver.session() as session:
            #Forza handshake/connessione fuori dal cronometro
            driver.verify_connectivity()
            session.run("RETURN 1").consume()

            #1 warm-up + 30 misure reali
            for i in range(31):
                start = time.perf_counter()
                session.run(cypher).data()   #end-to-end con deserializzazione
                end = time.perf_counter()
                times.append((end - start) * 1000.0)
    except Exception as e:
        return float("nan"), float("nan"), float("nan"), f"Errore Neo4j: {e}"
    finally:
        try:
            driver.close()
        except Exception:
            pass

    first = times[0]
    rest = times[1:] if len(times) > 1 else [times[0]]
    avg = sum(rest) / len(rest)
    ci = confidence_interval_95(rest)
    return first, avg, ci, None
```

Funziona in modo analogo a quelle per BaseX, ma eseguono query Cypher su Neo4j.

- Crea un driver Neo4j con le credenziali fornite e apre una sessione.
- Forza la connessione fuori dal cronometro:
  - `driver.verify_connectivity()` assicura che il database sia raggiungibile.
  - `session.run("RETURN 1").consume()` stabilisce la sessione e consuma un risultato minimo.
- Avvia un ciclo di 31 iterazioni (1 warm-up + 30 misure):
  - Registra il tempo iniziale con `time.perf_counter()`.
  - Esegue la query Cypher con `session.run(cypher).data()`, includendo il costo di deserializzazione in Python.
  - Registra il tempo finale e calcola la durata in millisecondi.
  - Salva ciascun tempo nella lista `times`.
- In caso di errore ritorna valori `NaN` e un messaggio `"Errore Neo4j: ..."`.
- Alla fine chiude comunque il driver.
- Calcola:
  - **first**: tempo della prima esecuzione registrata;
  - **avg**: media delle successive 30 esecuzioni;
  - **ci**: intervallo di confidenza al 95% sui 30 tempi calcolato con `confidence_interval_95`.
- Restituisce una tupla con i tre valori statistici e `None` come indicatore di assenza errori.

Formato dei risultati Ogni riga del file Excel contiene:

- Nome della query;
- Motore di database (BaseX o Neo4j);
- Tempo della prima esecuzione;
- Media delle 30 successive esecuzioni;
- Intervallo di confidenza al 95

5 Esperimenti

5.1 Query 1

Trovare banche con sede in Germania

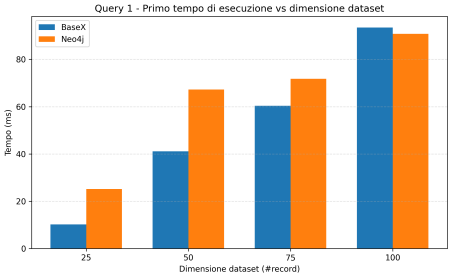

Figure 4: Query 1, prima esec

La prima query favorisce analisi su nodi diretti. In questo caso, BaseX dimostra un vantaggio netto rispetto a neo4j, che fatica a causa della sua struttura a grafo fortemente orientata alle relazioni. Con l'aumentare della grandezza del dataset neo4j diventa leggermente più veloce.

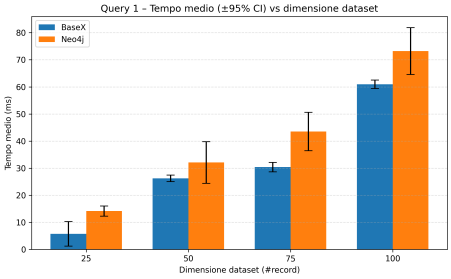

Figure 5: Query 1, media

Nonostante il meccanismo di caching di neo4j, basex resta comunque il vincitore del confronto mostrandosi ancora più performante durante l'analisi su dati lineari.

5.2 Query 2

Trovare fonti con poca affidabilità che iniziano con 'P'.

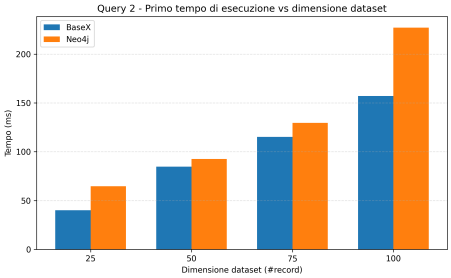

Figure 6: Query 2, prima

Anche in questo caso BaseX dimostra un vantaggio e Neo4J risulta ancora più lento soprattutto con l'aumentare del dataset.

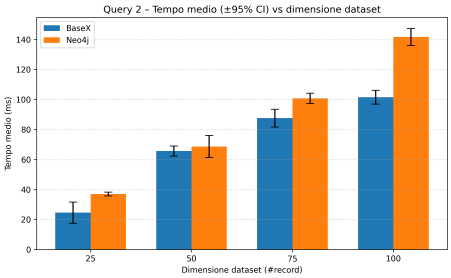

Figure 7: Query 2, media

Il caching di neo4j permette di avvicinarsi alle performance di BaseX, ma con l'aumentare del dataset inizia a faticare un po' di più, con performance stabili per neo4j.

5.3 Query 3

Trovare e contare le email ripetute nei documenti

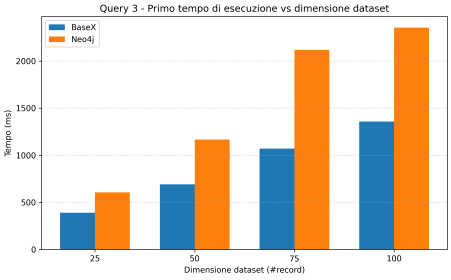

Figure 8: Query 3, media

La query si dimostra estremamente pesante per neo4j con tempi lunghi, mentre baseX la gestisce meglio, pur sforzandosi di più rispetto alle precedenti query.

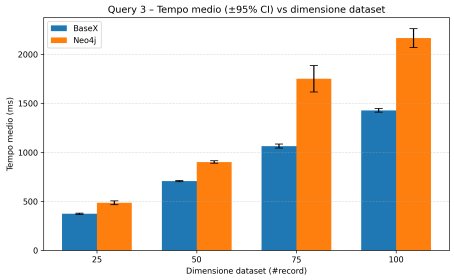

Figure 9: Query 3, media

BaseX impiega comunque meno tempo grazie alla sua capacità di gestione dei documenti XML, favorendo operazioni complesse che non coinvolgono relazioni.

5.4 Query 4

Trovare persone che hanno ricevuto in entrata più di quanto permette la loro banca in un giorno solo

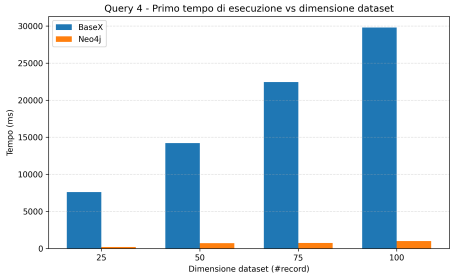

Figure 10: Query 4, prima

La query 4 rappresenta un punto di svolta a favore di neo4j: una query che coinvolge relazioni. Dal confronto emerge la straordinaria abilità dei database a grafo di gestire dati complessi mentre BaseX impiega molto più tempo.

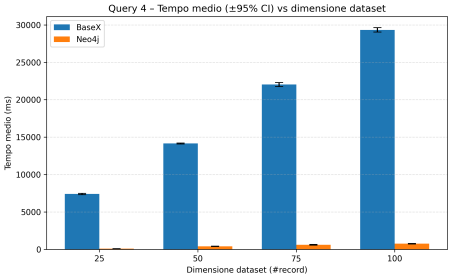

Figure 11: Query 4, media

Per neo4J, la query 4 risulta addirittura più leggera della 3 che non coinvolge relazioni. Le performance si dimostrano stabili per entrambi.

6 Conclusioni

- **BaseX** mostra un comportamento costante, con tempi medi inferiori per le prime tre query, ma rivelando la sua vulnerabilità nell'ultima.
- **Neo4j** presenta invece tempi medi molto più lunghi nelle prime tre, con performance migliori nell'ultima. Tuttavia, nei primi tempi di esecuzione è più sensibile al carico iniziale, mostrando picchi che si stabilizzano solo nelle esecuzioni successive.

6.1 Pro VS Contro

BaseX

- **Pregi:**
  - Buona stabilità tra esecuzioni successive.
  - Prestazioni iniziali competitive
  - Comportamento prevedibile senza caching o ottimizzazioni implicite.
- **Difetti:**
  - Tempi medi significativamente elevati per dataset medio-grandi relazionati.
  - Limitato impatto delle esecuzioni ripetute: assenza o inefficienza del caching.

Neo4j

- **Pregi:**

  - Ottima scalabilità: neo4J gestisce bene anche grandi volumi di dati complessi.
  - Presenza di caching efficace: forti miglioramenti dopo la prima esecuzione.

- **Difetti:**
  - Primi tempi di esecuzione più elevati
  - Minore prevedibilità nelle prime esecuzioni

6.2 Considerazioni

L'analisi mostra che **Neo4j è più efficiente quando lavora con dati complessi e relazionati** grazie a un caching aggressivo e a una migliore scalabilità. Tuttavia, **BaseX può risultare più competitivo in scenari in cui sono richieste ricerche lineari e accessi diretti senza caching**, in cui l'esecuzione iniziale è critica e la prevedibilità dei tempi è più importante dell'efficienza media.

La scelta tra i due sistemi dipende quindi dallo scenario di utilizzo:

- Se l'applicazione esegue query complesse ripetutamente con dati fortmente relazionati, Neo4j risulta vantaggioso.
- Se invece ogni query è lineare, effettua ricerche su nodi e attributi e viene eseguita una sola volta, BaseX può essere preferibile per la sua immediatezza.