UNIVERSITÀ
POLITECNICA
DELLE MARCHE

Facoltà di Ingegneria
Corso di Laurea Magistrale in Ingegneria Informatica e dell'Automazione

---

Progetto e Sviluppo di un sistema per la gestione ed estrazione di Instance Graph Prefix tramite Neo4j

Progetto del corso di New Generation Databases

Studentesse:
Chiara Gobbi
Alice Moretti

Docente:
Prof.ssa Claudia Diamantini

Anno Accademico 2023-2024

UNIVERSITÀ
POLITECNICA
DELLE MARCHE

Facoltà di Ingegneria
Corso di Laurea Magistrale in Ingegneria Informatica e dell'Automazione

---

Progetto e Sviluppo di un sistema per la gestione ed estrazione di Instance Graph Prefix tramite Neo4j

Progetto del corso di New Generation Databases

Studentesse:
Chiara Gobbi
Alice Moretti

Docente:
Prof.ssa Claudia Diamantini

Anno Accademico 2023-2024

Università Politecnica delle Marche
Facoltà di Ingegneria
Corso di Laurea Magistrale in Ingegneria Informatica e dell'Automazione
Via Brecce Bianche – 60131 Ancona (AN), Italy

Indice

Elenco delle figure

Elenco delle tabelle

Capitolo 1

Introduzione

Nell'era pionieristica dell'informatica, le risorse hardware dei calcolatori erano estremamente limitate. La memoria era scarsa, i processori avevano una capacità computazionale ridotta e lo spazio di archiviazione era costoso e limitato. Di conseguenza, gli sviluppatori erano costretti a ottimizzare ogni singolo bit e ciclo di clock per ottenere prestazioni accettabili dalle macchine a loro disposizione. Questa necessità di efficienza ha dato vita a pratiche di programmazione e gestione delle risorse che miravano a massimizzare l'utilizzo dell'hardware disponibile.

Oggi, nonostante il progresso tecnologico abbia portato ad una drastica riduzione dei costi delle risorse hardware e ad un aumento esponenziale della loro capacità, ci si trova nuovamente di fronte a una sfida simile, ma su una scala diversa: l'era dei Big Data.
La quantità di dati generata quotidianamente è immensa e continua a crescere a ritmi vertiginosi. Questo ha creato nuove esigenze di ottimizzazione delle risorse, non solo per gestire efficacemente questi enormi volumi di dati, ma anche per estrarre informazioni significative e utili in tempi ragionevoli.

In questo contesto, l'ottimizzazione delle risorse non riguarda più solo l'hardware fisico, ma si estende anche alle infrastrutture cloud, ai sistemi di archiviazione distribuiti, agli algoritmi di elaborazione dati e alle tecniche di gestione dei flussi di dati. La necessità di efficienza, dunque, rimane una costante nell'evoluzione dell'informatica, richiedendo un continuo adattamento e innovazione per far fronte alle sfide emergenti.

Lo scopo di questo elaborato è dunque quello di comprendere e identificare strategie efficaci che possono essere implementate per gestire al meglio l'enorme quantità di dati che ormai caratterizza il mondo digitale.
In particolare, a partire da un dataset, l'obiettivo è quello di ottimizzare una procedura di gestione ed estrazione di Instance Graph Prefixes utili per il training di una rete neurale deep in ambito del process mining.

1.1 Metodologia

Nel presente elaborato, è stato utilizzato un dataset a grafo estratto tramite l'algoritmo BIG [1] a partire dal log BPI12 [2], ampiamente utilizzato nell'ambito del process mining.
L'obiettivo principale è quello di ottimizzare il processing del dataset con l'intento di migliorare l'analisi e l'estrazione di informazioni rilevanti. Per gestire la complessità del dataset, è stato utilizzato il Graph DBMS Neo4j.

In particolare, la metodologia seguita si è composta dei seguenti step:

1. Modellazione dei grafi su Neo4j: processing di un sottinsieme dei dati di partenza in modo che questi potessero essere compatibili con la struttura del database a grafo. Nel modello, ogni nodo rappresenta un evento, mentre le relazioni tra i nodi indicano la sequenza degli eventi nei diversi casi. Inoltre, si sono utilizzate le proprietà dei nodi per memorizzare attributi.

2. Codifica delle interrogazioni in Neo4j: sviluppo delle query a partire dai grafi importati con lo scopo di comprendere i vantaggi della ricerca di informazioni in Neo4j ed estrarre informazioni utili dai grafi.

3. Ottimizzazione delle interrogazioni: studio e applicazione delle interrogazioni per migliorare le prestazioni del database, specialmente quando si lavora con grandi volumi di dati o con query complesse.

4. Esperimenti e risultati: confronto delle operazioni svolte sui dati in modo da osservare i vantaggi e gli svantaggi dei vari approcci.

Capitolo 2

Interrogazioni su Neo4j

In questo capitolo verrà descritto brevemente il dataset utilizzato.
Successivamente, si presenterà una panoramica di Neo4j e si mostrerà lo sviluppo delle interrogazioni sui grafi.

2.1 Dataset

Si è deciso di lavorare con il dataset BPI12 [2], fornito per la Business Process Intelligence Challenge 2012 e tratto da un istituto finanziario olandese. Questo dataset contiene gli eventi di un processo di richiesta di prestiti.
Il dataset consiste in un event log, cioè una serie di tracce, ciascuna delle quali rappresenta un singolo caso e contiene una sequenza di eventi. Ogni evento rappresenta un'attività eseguita durante il processo di gestione del mutuo ed è accompagnato da informazioni come l'identificatore del caso, il nome dell'attività, la data e l'ora dell'evento e la risorsa che ha eseguito l'attività.
Le caratteristiche generali del dataset sono riportate nella tabella 2.1.

N. di tracce	Eventi totali	N. di tipi di attività
13.087	262.200	23

Tabella 2.1: Caratteristiche dei dataset BPI12

A partire da questo log è stato estratto un dataset a grafo tramite l'algoritmo BIG [1]. In particolare, questo algoritmo ha permesso di generare un grafo per ciascuna traccia, rappresentando la sequenza degli eventi svolti in ogni esecuzione.
Ognuno di questi grafi, prende il nome di Instance Graph [3].
Un esempio di *Instance Graph* relativo alla traccia

$$\sigma = <(1, \mathrm{S}), (2, \mathrm{SENT}), (3, \mathrm{CANCEL}), (4, \mathrm{APPROVE}), (5, \mathrm{DECLINE}), (6, \mathrm{E})>$$

e costruito secondo l'algoritmo BIG è riportato in Figura 2.1.

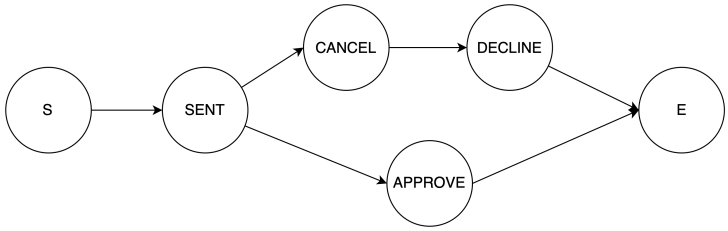


Figura 2.1: Esempio di *Instance Graph*

2.2 Neo4j

Neo4j è un Graph DBMS open source sviluppato dalla Neo4j, Inc.
È progettato per memorizzare dati come nodi, relazioni tra nodi e proprietà che descrivono sia i nodi che le relazioni.

Uno dei principali vantaggi di Neo4j è la sua capacità di modellare i dati in modo naturale, riflettendo direttamente le loro interrelazioni reali. La sua architettura è ottimizzata per eseguire query su relazioni complesse e multi-hop, rendendolo molto più veloce rispetto ai database relazionali per questo tipo di operazioni. Inoltre, offre una grande flessibilità, permettendo di aggiungere o modificare nodi e relazioni senza dover alterare schemi rigidi, facilitando l'adattamento ai cambiamenti nei requisiti dei dati.

2.2.1 Ecosistema Neo4j

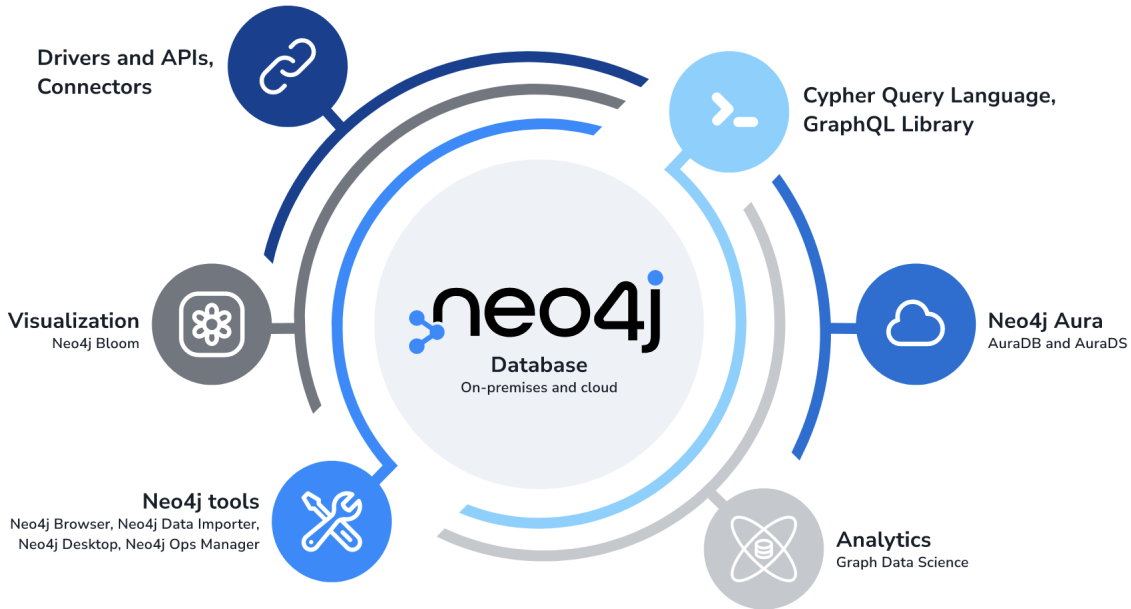


Figura 2.2: Ecosistema di Neo4j

In Figura 2.2 è mostrata una panoramica dell'ecosistema di Neo4j, evidenziando le diverse componenti e strumenti che lavorano insieme per offrire una piattaforma completa per la gestione dei dati.
Le componenti principali sono:

- Neo4j Database On-premises and Cloud: il cuore dell'ecosistema Neo4j è il database stesso, che può essere implementato sia on-premises (sui server locali) che nel cloud. Questo offre flessibilità in base alle esigenze dell'utente.
- Cypher Query Language e GraphQL Library
  - Cypher è il linguaggio di query utilizzato per interagire con il database Neo4j. Cypher rende le query facili da scrivere e comprendere, grazie a una sintassi intuitiva che consente di esprimere in modo semplice e leggibile le operazioni sui dati di grafo. La vasta comunità di sviluppatori e il buon supporto, sia nella documentazione che attraverso forum e altre risorse, rappresentano un ulteriore punto di forza di Neo4j.
  - Neo4j fornisce anche una libreria per integrarsi con GraphQL, un linguaggio di query per API.
- Neo4j Aura: servizio di database a grafo gestito nel cloud. AuraDB è il servizio principale per l'hosting dei database Neo4j, mentre AuraDS è una versione orientata alla data science che offre strumenti avanzati per l'analisi dei dati.
- Graph Data Science: componente che include strumenti e librerie per eseguire analisi avanzate sui dati a grafo. Permette di utilizzare algoritmi di machine learning e analisi dei grafi per estrarre conoscenza dai dati.
- Neo4j Tools:
  - Neo4j Browser: interfaccia grafica per esplorare il grafo, eseguire query Cypher e visualizzare i risultati in modo interattivo.
  - Neo4j Data Importer: strumenti per importare dati da diverse fonti nel database Neo4j.
  - Neo4j Desktop: applicazione per sviluppatori che offre strumenti per la gestione dei database, la creazione di grafi e il monitoraggio delle prestazioni.
  - Neo4j Ops Manager: strumenti per il monitoraggio e la gestione operativa dei database Neo4j.
- Neo4j Bloom: strumento di visualizzazione che consente di esplorare e interagire con i dati a grafo in modo intuitivo, utile per utenti non tecnici e business analysts.
- Drivers and APIs, Connectors

- Drivers per vari linguaggi di programmazione (Java, Python, JavaScript, etc.), che facilitano l'integrazione di Neo4j nelle applicazioni.
- APIs and Connectors per collegare il database ad altri sistemi e applicazioni, supportando una varietà di protocolli e formati di dati.

Questa architettura modulare e integrata permette a Neo4j di essere una soluzione completa per la gestione dei dati a grafo, dalla memorizzazione e query, all'analisi avanzata e visualizzazione, fino all'integrazione con altre applicazioni e servizi.

2.2.2 Distribuzione dei dati

Sebbene Neo4j supporti il clustering e la distribuzione dei dati, gestire grafi molto grandi può diventare complicato. La scalabilità orizzontale, che è una caratteristica fondamentale di molti database NoSQL, è più difficile da ottenere con Neo4j. Questo è dovuto in parte alla complessità intrinseca delle operazioni sui grafi, che spesso richiedono traversate su più nodi e relazioni che non si prestano bene alla partizione dei dati su più nodi fisici.

La distribuzione delle query in Neo4j può essere meno efficiente rispetto ad altri database distribuiti. Le query su un grafo spesso necessitano di accedere a dati che sono strettamente collegati tra loro, e se questi dati sono distribuiti su più nodi, la latenza di rete può rallentare significativamente le prestazioni delle query. A differenza di alcune architetture di database distribuiti, dove le operazioni possono essere eseguite in parallelo su diversi nodi, le query sui grafi richiedono spesso operazioni sequenziali che attraversano molti nodi.
Le funzionalità avanzate di clustering e distribuzione dei dati in Neo4j sono disponibili principalmente nelle versioni enterprise e sono molto costose. A causa di questi limiti di budget, tali funzionalità non sono state applicate per questo lavoro.

Ulteriori peculiarità e caratteristiche di Neo4j utili per l'ottimizzazione verranno analizzate e approfondite direttamente nelle sezioni successive.

2.3 Interrogazioni

Sono state svolte delle interrogazioni per analizzare la struttura del dataset utilizzando Cypher, il linguaggio di interrogazione dei grafi di Neo4j. La sua natura dichiarativa consente agli utenti di concentrarsi sulla descrizione di ciò che vogliono ottenere, piuttosto che su come ottenerlo. Pertanto, è possibile definire delle query in modo intuitivo, concentrandosi sui pattern e sulle relazioni di interesse, senza soffermarsi sui dettagli implementativi.

2.3.1 Import dei dati

Prima di poter eseguire delle interrogazioni è necessario importare i dati nel database Neo4j. Questo processo coinvolge la definizione dei nodi, delle relazioni e delle proprietà che costituiscono il grafo.

Il file CSV ottenuto dopo l'applicazione dell'algoritmo BIG contiene delle informazioni sulle attività. Ogni riga fornisce dettagli sull'identificativo univoco dell'attività, il nome dell'attività, l'identificativo univoco della traccia associata all'attività, la data e l'ora di inizio e di fine dell'attività e la risorsa coinvolta nell'esecuzione dell'attività. Di seguito è riportata la query, integrata con le funzioni dell'estensione APOC (Awesome Procedures On Cypher), per gestire e formattare i dati durante l'importazione.

```
LOAD CSV FROM "file:///prefix_log.csv" AS row
MERGE (:Event{
activity_id:toInteger(row[0]),
event_name:row[1],
track_id:row[2],
start_time:datetime(apoc.text.replace(apoc.text.replace(row[3], "\+\d{2}:\d{2}$", ''), ' ', 'T')),
finish_time:datetime(apoc.text.replace(apoc.text.replace(row[4], "\+\d{2}:\d{2}$", ''), ' ', 'T')),
resource:row[5]})
```

Listing 2.1: Query per l'import dei dati

La query carica il file CSV e legge ogni riga del file. Ogni riga viene quindi assegnata ad una variabile *row* che permette di accedere ai dati di ogni riga e manipolarli.
Il comando *MERGE* viene utilizzato per creare un nodo di tipo *Event* se non esiste già un nodo con le stesse proprietà. Se il nodo esiste, il comando evita di creare un duplicato, assicurando l'unicità dei nodi nel grafo.

Per ogni nodo vengono definite una serie di proprietà:

- *activity_id*: indica l'ID dell'attività all'interno di ogni grafo, ottenuto convertendo il valore della prima colonna della riga in un intero;
- *event_name*: indica il nome dell'attività, ottenuto dalla seconda colonna della riga;
- *track_id*: indica l'ID della traccia, ottenuto dalla terza colonna della riga;
- *start_time* e *finish_time*: indicano rispettivamente le date di inizio e di fine dell'evento. Queste informazioni hanno richiesto una manipolazione per rimuovere il fuso orario dalle stringhe e per sostituire lo spazio tra la data e l'ora con una 'T' rendendo la stringa conforme allo standard ISO 8601 richiesto dal tipo *datetime* di Neo4j;
- *resource*: indica la risorsa che ha svolto l'attività, ottenuta dal sesto valore della riga.

In particolare, la coppia (*activity_id*, *track_id*) costituisce un identificativo univoco del nodo.

Dopo aver importato i nodi è stato necessario definire anche gli archi.

```
MATCH (start:Event), (end:Event)
WHERE
((start.activity_id + 1= end.activity_id AND start.finish_time = end.start_time)
OR
(
  start.finish_time = end.start_time AND
  start.start_time <> start.finish_time
  AND NOT EXISTS {
      MATCH (n:Event)
      WHERE n.activity_id > start.activity_id
        AND n.activity_id < end.activity_id
        AND n.start_time = n.finish_time
        AND n.start_time >= start.finish_time
        AND n.start_time <= end.start_time
  }
))
AND start.track_id = end.track_id
MERGE (start)-[r:NEXT]->(end) SET r.connection=start.activity_id+"_"+end.activity_id
```

Listing 2.2: Query per la creazione degli archi

In particolare, la difficoltà nel gestire gli archi deriva dalla possibilità di avere attività parallele e istantanee all'interno di un grafo.

Per questo motivo, la query è strutturata in due parti. La prima parte collega due nodi sulla base del loro ID (proprietà *activity_id*) secondo la logica che due nodi sono collegati se l'ID del primo è immediatamente precedente all'ID del secondo. Questa condizione non è però sufficiente per determinare le attività sequenziali a causa del problema dei parallelismi. Infatti, le attività parallele hanno ID immediatamente progressivi e, per ovvie, ragioni non devono essere collegate. Per questo motivo, occorre anche specificare che il timestamp di fine del primo nodo collegato deve coincidere con il tempo di inizio del secondo nodo collegato seguendo l'algoritmo utilizzato per generare i timestamp delle attività. In questo modo, le attività parallele non saranno collegate tra loro in quanto i loro timestamp non rispetteranno il vincolo appena citato.

La seconda condizione serve invece per gestire le attività parallele e quelle istantanee. Innanzitutto, due nodi devono essere collegati se il timestamp dell'attività di partenza coincide con il timestamp dell'attività di arrivo. Inoltre, per evitare la presenza di self loops nel caso in cui le attività siano istantanee (per definizione un'attività è istantanea se il suo timestamp di inizio coincide con il suo timestamp di fine), i timestamp di inizio e di fine devono essere differenti.

Questo vincolo è necessario secondo la logica in cui opera Neo4j.
Infatti la query esprime un match di due *Event* a cui saranno assegnati gli alias *start* e *end*. Questo signfica che uno stesso nodo, nell'esecuzione della query sara considerato sia come *start* che come *end*.

La sotto-query *NOT EXISTS* assicura che non esista nessun altro evento n istantaneo che abbia *activity_id* tra quelli di *start* ed *end* e che cada tra *start.finish_time* e *end.start_time*. In questo modo, due nodi che abbiano un'attività istantanea nel

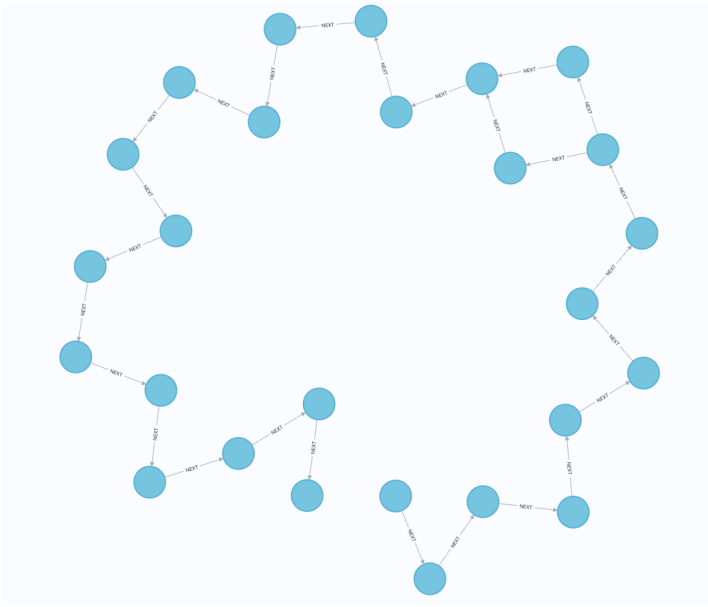


Figura 2.3: Esempio di creazione di nodi e archi in Neo4j

mezzo e che rispettino il vincolo di sequenzialità non verranno collegati tra loro.

L'ultima condizione specificata assicura che questi collegamenti valgano solo per nodi di uno stesso grafo. Infine, viene creata la relazione *NEXT* tra *start* ed *end* e impostata la proprietà *connection* della relazione r concatenando gli *activity_id* dei due nodi.

In Figura 2.3 è riportato un esempio di grafo creato tramite queste due query.

2.3.2 Risorse

Una delle proprietà importanti da considerare nell'esecuzione delle attività riportate nel log riguarda le risorse. È fondamentale tenere presente che all'interno di qualsiasi processo le risorse sono limitate. Per tale ragione, sono state sviluppate alcune utili interrogazioni per acquisire tali informazioni.

In particolare, potrebbe essere interessante capire quali risorse sono libere in un determinato istante temporale.

```
MATCH (e:Event)
WHERE e.start_time<=datetime("2011-10-01T10:11:08Z") AND e.finish_time>=datetime("
    2011-10-01T10:11:08Z") AND e.resource<>"No_resource"
RETURN DISTINCT(e.resource)
```

Listing 2.3: Query per determinare le risorse libere all'istante di tempo i

Oppure, si potrebbero voler identificare le risose più occupate, ovvero quelle che eseguono più attività in un certo istante di tempo.
In questo caso, l'interrogazione è più articolata. In particolare, la query cerca tutti i nodi con l'etichetta *Event* e utilizza il comando *WITH DISTINCT* per ottenere una lista di tutti i tempi di inizio unici delle attività (*activity.start_time*), assegnandoli all'alias *activity_start_time*.
In seguito, vengono raggruppati i risultati per *activity_start_time* e *resource* (la risorsa dell'evento *e*), contando il numero di eventi (*COUNT(*)*) che corrispondono a ciascuna combinazione di *activity_start_time* e *resource*, e assegnando questo conteggio a *resource_count*. Infine, vengono ordinati i risultati per *resource_count*.

```
MATCH (activity:Event)
WITH DISTINCT activity.start_time AS activity_start_time
MATCH (e:Event)
WHERE e.start_time <= activity_start_time AND e.finish_time >= activity_start_time
WITH activity_start_time, e.resource AS resource, COUNT(*) AS resource_count
WITH activity_start_time, resource, resource_count
RETURN activity_start_time, resource, resource_count
ORDER BY resource_count DESC
```

Listing 2.4: Query per determinare le risorse più occupate

Una porzione di dati riportati dalla query è mostrato in Figura 2.4.

	activity_start_time	resource	resource_count
1	"2011-10-01T15:39:44Z"	"No_resource"	7
2	"2011-10-11T10:28:50Z"	"10138"	7
3	"2011-10-11T10:39:30Z"	"10138"	7
4	"2011-10-01T15:43:28Z"	"No_resource"	6
5	"2011-10-05T13:27:38Z"	"No_resource"	6
6	"2011-10-05T13:28:37Z"	"No_resource"	6

Figura 2.4: Occupazione delle risorse

In maniera simile ma indipendentemente dall'istante di tempo, si potrebbe voler conoscere la risorsa più occupata, ovvero la risorsa a cui vengono assegnate più attività.

```
MATCH (e:Event)
WHERE e.resource <> "No_resource"
WITH e.resource AS resource, COUNT(*) AS resource_count
ORDER BY resource_count DESC
LIMIT 1
RETURN resource, resource_count
```

Listing 2.5: Query per determinare la risorsa a cui sono assegnate più attività

Con il sottinsieme di dati scelto per il testing delle query risulta che la risorsa più occupata è la *112* come mostrato in Figura 2.5.

	resource	resource_count
1	"112"	102

Figura 2.5: Risorsa a cui vengono assegnate più attività

Un'altra interrogazione interessante è quella che riguarda le risorse occupate all'avvio di ogni attività.
Ovvero, per ogni tempo di inizio delle attività del log, si vogliono conoscere le risorse che sono impegnate nei vari sotto-processi in modo da ricavare informazioni sulla loro occupazione.

```
MATCH (activity:Event)
WITH DISTINCT activity.start_time AS activity_start_time
MATCH (e:Event)
WHERE e.start_time <= activity_start_time AND e.finish_time >= activity_start_time
WITH activity_start_time, e.resource AS resource
RETURN activity_start_time, collect(resource) as busy
ORDER BY activity_start_time
```

Listing 2.6: Query per determinare l'occupazione delle risorse all'avvio di ogni attività

La query ritorna i risultati nel formato riportato in figura 2.6

	activity_start_time	busy
1	"2011-10-01T08:11:07Z"	["No_resource", "112"]
2	"2011-10-01T08:11:08Z"	["112", "112"]
3	"2011-10-01T08:11:09Z"	["112", "112"]
4	"2011-10-01T08:11:46Z"	["112", "No_resource"]
5	"2011-10-01T08:15:38Z"	["No_resource", "112"]

Figura 2.6: Risorse occupate all'avvio di ogni attività

Infine, in un contesto aziendale, è importante riconoscere che, tipicamente, non tutte le risorse sono specializzate per eseguire tutte le attività richieste dai vari processi. Pertanto, compiti specifici saranno assegnati alle risorse più appropriate, tenendo conto delle loro competenze, esperienze e abilità specifiche, al fine di garantire un'efficace ed efficiente esecuzione delle operazioni aziendali.
Tale informazione può essere ricavata tramite la seguente query.

```
MATCH (n)
WHERE n.resource IS NOT NULL AND n.event_name IS NOT NULL
RETURN n.resource AS Resource, COLLECT(DISTINCT n.event_name) AS Activities
```

Listing 2.7: Query per determinare le attività che le risorse possono eseguire

Come è possibile notare in Figura 2.7, potrebbero anche esserci attività che non hanno bisogno di risorse per essere eseguite o di cui la risorsa non è specificata. In entrambi i casi, al campo relativo relativo alla risorsa è stato assegnato il valore *"No_resource"*.

	Resource	Activities
1	"No_resource"	["START", "END", "W_Completerenaanvraag", "W_Nabellenoffertes", "W_Nabellenincompletedossiers", "W_Validerenaanvraag", "W_Afhandelenleads"]
2	"112"	["A_SUBMITTED", "A_PARTLYSUBMITTED", "A_DECLINED", "A_PREACCEPTED", "W_Completerenaanvraag", "W_Afhandelenleads", "A_CANCELLED"]
3	"10912"	["W_Completerenaanvraag", "A_CANCELLED", "W_Afhandelenleads", "A_PREACCEPTED", "A_ACCEPTED", "O_SELECTED", "A_FINALIZED", "O_CREATED", "O_SEI
4	"11111"	["A_DECLINED", "A_ACCEPTED", "O_SELECTED", "A_FINALIZED", "O_CREATED", "O_SENT"]
5	"10982"	["W_Completerenaanvraag", "W_Nabellenoffertes", "O_SELECTED", "O_CANCELLED", "O_CREATED", "O_SENT"]
6	"11019"	["W_Completerenaanvraag", "W_Nabellenincompletedossiers", "A_ACCEPTED", "A_FINALIZED", "O_SELECTED", "O_CREATED", "O_SENT", "W_Nabellenoffertes"]

Figura 2.7: Attività eseguibili dalle risorse

2.3.3 Active case

Un ulteriore aspetto di particolare interesse riguarda i case attivi ad un determinato istante di tempo, ossia i case che sono attualmente in esecuzione.

A tal proposito, è stata definita una query che cerca i nodi di tipo *Event* e filtra questi eventi per includere solo quelli che sono iniziati prima o allo stesso istante del timestamp specificato e che al tempo stesso sono terminati dopo o allo stesso tempo del timestamp specificato.
La query restituisce solo gli identificatori distinti delle tracce (*track_id*) degli eventi attivi in quel momento.

```
MATCH (e:Event)
WHERE e.start_time<=datetime("2011-10-01T10:11:07Z") AND e.finish_time>=datetime("
    2011-10-01T10:11:07Z")
RETURN DISTINCT e.track_id
```

Listing 2.8: Query per determinare i case attivi all'istante di tempo i

Nella Figura 2.8 sono riportati i case attivi risultanti dall'esecuzione della precedente query.

	e.track_id
1	"173703"
2	"173706"
3	"173709"
4	"173712"
5	"173715"

Figura 2.8: Case attivi all'istante di tempo "2011-10-01T10:11:07Z"

Questa query fornisce una vista puntuale dello stato del sistema in un determinato istante di tempo. Tuttavia, potrebbe essere interessante analizzare l'andamento di tutti gli eventi attivi per avere una panoramica completa dell'attività del sistema. Per questo motivo è stata definita un'ulteriore query che individua tutti i case attivi ad ogni istante di tempo in cui inizia un evento.

La query cerca tutti i nodi di tipo *Event* e raccoglie tutti i tempi di inizio distinti degli eventi (*activity.start_time*). Poi, per ogni tempo di inizio individuato, si cercano tutti gli eventi attivi in quel momento con la medesima logica del caso precedente. Infine, per ogni tempo, si restituisce una lista contenente gli identificatori delle tracce

distinti relativi agli eventi attivi in quel momento. Tali risultati sono ordinati sulla base del tempo di inizio degli eventi.

```
MATCH (activity: Event)
WITH DISTINCT activity.start_time AS activity_start_time
MATCH (e:Event)
WHERE e.start_time <= activity_start_time AND e.finish_time >= activity_start_time
RETURN activity_start_time, COLLECT(DISTINCT e.track_id) as active_case
ORDER BY activity_start_time
```

Listing 2.9: Query per determinare i case attivi ad ogni istante di tempo

Nella Figura 2.9 sono riportati alcuni dei risultati ottenuti a seguito dell'esecuzione della query.

	activity_start_time	active_case
10	"2011-10-01T09:45:25Z"	["173703"]
11	"2011-10-01T09:45:36Z"	["173703", "173706"]
12	"2011-10-01T09:45:37Z"	["173703", "173706"]
13	"2011-10-01T09:46:18Z"	["173703", "173706"]
14	"2011-10-01T09:57:41Z"	["173703", "173706", "173709"]

Figura 2.9: Case attivi

2.3.4 Esecuzione delle attività

Già al momento della definizione degli archi del grafo sono state incontrate delle difficoltà nella gestione delle attività parallele e istantanee. Di conseguenza, si è voluta concentrare l'attenzione proprio su questi eventi.

Attività istantanee

In particolare, la seguente query individua tutte le attività istantanee, ossia tutte quelle attività i cui tempi di inizio e di fine coincidono.
Per far questo, si cercano tutti i nodi di tipo *Event* nel grafo e si filtrano gli eventi in maniera tale da includere solo quelli che hanno lo stesso valore per *start_time* e *finish_time*. Infine, si restituiscono solo i nomi distinti delle attività che soddisfano la precedente condizione.

```
MATCH (e:Event)
WHERE e.start_time=e.finish_time
RETURN DISTINCT e.event_name
```

Listing 2.10: Query per determinare le attività che possono essere istantanee

Inoltre, si è voluta valutare la frequenza con qui queste attività istantanee si verificano rispetto al totale delle occorrenze di ciascuna attività.

```
MATCH (e:Event)
WITH e.event_name AS event_name,
     count(*) AS total_occurrences,
     sum(CASE WHEN e.start_time = e.finish_time THEN 1ELSE 0END) AS instant_count
RETURN event_name, instant_count, total_occurrences
```

Listing 2.11: Query per determinare la frequenza delle attività istantanee

Alcuni dei risultati ottenuti con tale query sono riportati nella Figura 2.10.

	event_name	instant_count	total_occurrences
1	"START"	29	29
2	"A_SUBMITTED"	0	29
3	"A_PARTLYSUBMITTED"	22	29
4	"A_DECLINED"	1	18
5	"END"	0	29
6	"A_PREACCEPTED"	0	20

Figura 2.10: Frequenza delle attività istantanee rispetto al numero di occorrenze totali

Attività parallele

In modo simile al caso dell'attività istantanee, sono state identificate le attività che possono essere eseguite in parallelo, anche se in questo caso la situazione è più complicata.

La query individua coppie di attività che iniziano nello stesso momento e sono eseguite in maniera parallela.
In particolare, dopo aver individuato due eventi $n1$ e $n2$ che sono collegati allo stesso nodo di partenza, si verifica che gli eventi siano distinti, che inizino nello stesso momento, precisamente quando termina l'attività del nodo precedente e che non si tratti di attività istantanee.

Dopodiché, si costruiscono i percorsi che collegano $n1$ e $n2$ ad un nodo finale comune, garantendo che questi siano distinti. Le attività vengono memorizzate su delle liste e la query restituisce tutte le combinazioni distinte di liste di attività parallele.

```
MATCH (s1:Event)−[:NEXT]−>(n1:Event)
MATCH (s1)−[:NEXT]−>(n2:Event)
MATCH p1=(n1)−[*]−>(f1:Event), p2=(n2)−[*]−>(f1)
WHERE n1.activity_id <> n2.activity_id
AND n1.start_time = n2.start_time
AND n1.start_time = s1.finish_time
AND n1.start_time <> n1.finish_time
AND n2.start_time <> n2.finish_time
AND n1.activity_id < n2.activity_id
WITH DISTINCT [n1 IN nodes(p1)[..−1] | n1.event_name] AS parallel1, [n2 IN nodes(p2)[..−1] |
    n2.event_name] AS parallel2
WITH CASE WHEN parallel1 <= parallel2 THEN apoc.convert.toList([parallel1,parallel2]) ELSE
    apoc.convert.toList([parallel2,parallel1]) END AS combined
return DISTINCT combined
```

Listing 2.12: Query per determinare le attività che possono essere parallele

Alcuni dei risultati ottenuti dall'esecuzione di questa query sono mostrati nella Figura 2.11.

	combined
1	[["A_FINALIZED"], ["O_SELECTED"]]
2	[["A_FINALIZED"], ["O_SELECTED", "O_CREATED", "O_SENT", "W_Nabellenoffertes", "W_Completerenaanvraag", "W_Nabellenoffertes", "W_Nabellenoffertes"
3	[["A_APPROVED"], ["O_ACCEPTED"]]
4	[["A_REGISTERED"], ["O_ACCEPTED"]]
5	[["A_ACTIVATED"], ["O_ACCEPTED"]]
6	[["A_APPROVED"], ["A_REGISTERED"]]

Figura 2.11: Attività che possono essere eseguite in parallelo ad altre attività

Oppure, in maniera simile all'interrogazione precedente, è possibile ricavare le attività parallele presenti all'interno di ogni grafo.

```
MATCH (s1:Event)−[:NEXT]−>(n1:Event)
MATCH (s1)−[:NEXT]−>(n2:Event)
MATCH p1=(n1)−[*]−>(f1:Event), p2=(n2)−[*]−>(f1)
WHERE n1.activity_id <> n2.activity_id
AND n1.start_time = n2.start_time
AND n1.start_time = s1.finish_time
AND n1.start_time <> n1.finish_time
AND n2.start_time <> n2.finish_time
AND n1.activity_id < n2.activity_id
```

```
RETURN *;
```

Listing 2.13: Query per individuare i parallelismi all'interno dei grafi

In questo caso, l'output proposto dall'interfaccia Neo4j Bloom è mostrato in Figura 2.12

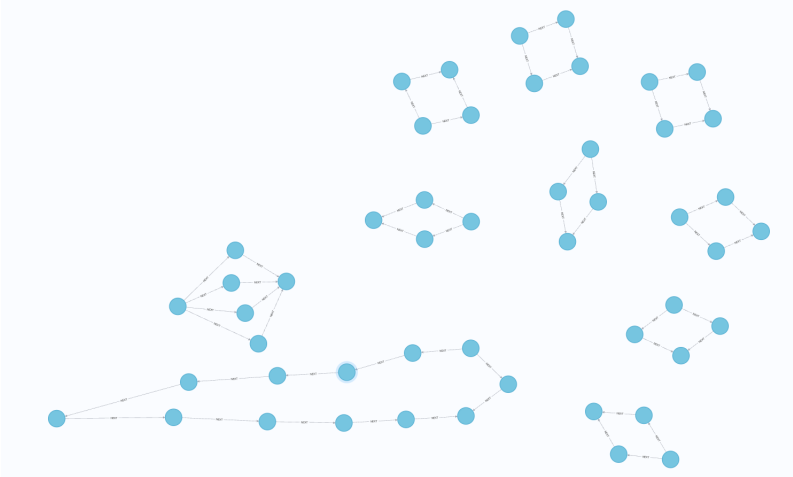


Figura 2.12: Attività eseguite in parallelo

Vista l'importanza della gestione dei parallelismi, è stata definita una query che individua tutti i grafi che contengono attività parallele.
La query cerca un nodo di partenza s che ha due nodi successivi $a1$ e $a2$ e verifica che ciascuno di questi abbia un'attività successiva.
Dopodiché, si impongono delle condizioni per valutare se il grafo ha dei parallelismi. In particolare, si va a verificare che le attività $a1$ e $a2$ siano attività distinte, che non siano attività istantanee e che appartengano alla stessa traccia.
Inoltre, con l'operatore $WITH$, la query raggruppa i risultati sulla base dell'identificativo della traccia e conta il numero di eventi distinti $a2$ che possono essere eseguiti in paralello. Vengono riportate solo le tracce che hanno almeno due attività parallele.

```
MATCH (s:Event)−[:NEXT]−>(a1:Event),
  (s)−[:NEXT]−>(a2:Event),
  (a1)−[:NEXT]−>(e1:Event),
  (a2)−[:NEXT]−>(e2:Event)
WHERE a1 <> a2
  AND a1.finish_time <> a1.start_time
  AND a2.finish_time <> a2.start_time
  AND a1.track_id = a2.track_id
WITH a1.track_id AS track_id, COUNT(DISTINCT a2) AS numParallelActivities
WHERE numParallelActivities >= 2
RETURN track_id;
```

Listing 2.14: Query per individuare i grafi con attività parallele

2.3.5 Calcolo dei prefissi

Un'interrogazione importante da effettuare è quella che consente di calcolare i prefissi. I prefissi sono sottografi ricavabili considerando il nodo con *activity_id* minore del grafo originale e aggiungendo di volta in volta il nodo con *activity_id* progressivo rispetto all'ultimo inserito. In questo modo, seguendo l'ordine dei nodi, a partire da un grafo con n nodi è possibile ricavare n-1 prefissi.
A tali prefissi è poi possibile associare la label del nodo che verrà aggiunto nella successiva iterazione. Tali strutture sono molto rilevanti in quanto possono costituire l'input di una rete neurale a grafo utilizzabile per effettuare task di Next Activity Prediction [3].

A causa dell'overhead introdotto dall'interfaccia di Neo4j e della dinamicità della query, in questo caso, è stato utilizzato il modulo *neo4j* per Python che offre un'interfaccia Python per interagire con un database Neo4j.
In particolare, l'utilizzo di Python si è reso necessario in quanto la query richiede un cambio dinamico di un parametro k che rappresenta la profondità dei nodi che si vuole considerare ad ogni iterazione per il calcolo dei prefissi. Ciò non sarebbe stato possibile in Neo4j.

Il codice utilizzato è riportato nel Listing 2.15.

```
max_len = self.driver.execute_query("MATCH (n:Event) RETURN max(n.
    activity_id) AS max_activity_id")[0][0][0]
        for k in range(1, max_len - 1):
            result = self.driver.execute_query(f"MATCH (e:Event) WHERE
                e.activity_id <= {k} "
            f"WITH collect(e) AS p_nodes, e.track_id as track_id "
            f"WHERE size(p_nodes)={k} "
            f"UNWIND p_nodes AS node1 "
            f"UNWIND p_nodes AS node2 "
            f"OPTIONAL MATCH (node1)-[r]-(node2) "
            f"MATCH (l:Event) WHERE l.activity_id={k + 1} "
            f"AND l.track_id=node1.track_id "
            f"RETURN DISTINCT p_nodes, collect(DISTINCT r) as p_rels, "
            f"track_id, [l] as label",database_="neo4j",
            result_transformer_=neo4j.Result.to_df)
```

Listing 2.15: Query ottimizzata per la creazione dei prefissi

Innanzitutto, è stato necessario definire una prima query per calcolare l'*activity_id* massimo tra tutti i nodi *Event*. Questo permette di trovare sicuramente tutti i possibili prefissi, poichè a partire da tutti i grafi presenti nel dataset, si ritorna il massimo valore di *activity_id*. In questo modo, è garantito che tutti i nodi fino a quel livello di profondità siano considerati.
Successivamente, per ogni valore di k compreso tra 1 e il massimo valore di *activity_id* meno 1, si esegue una query dinamica. Questa query seleziona i nodi *Event* che

hanno un *activity_id* minore o uguale di k e colleziona questi nodi in una lista, raggruppandoli sulla base dell'identificativo della traccia (*track_id*). In seguito, si creano tutte le combinazioni di nodi e si trovano le relazioni tra queste coppie.
In questo caso è stato necessario specificare *OPTIONAL MATCH* in modo da includere anche la casistica in cui il grafo sia composto da un unico nodo, ovvero, per soddisfare la condizione di prefisso di lunghezza 1.
Infine, si selezionano i nodi con *activity_id* pari a $k + 1$ e con lo stesso *track_id* dei nodi selezionati precedentemente, restituendo distintamente i nodi, le loro relazioni, l'identificativo della traccia e la label relativa all'etichetta del nodo successivo.

Nelle sezioni successive verranno esplorati e concepiti nuovi metodi di generazione dei prefissi che sfruttano noti algoritmi di attraversamento dei grafi.

Capitolo 3

Ottimizzazione

In questa sezione verranno riproposte le più rilevanti query descritte nei paragrafi precedenti in chiave ottimizzata.
Nel successivo capitolo, queste saranno applicate al dataset intero o a sue sotto porzioni in modo da testarne l'efficienza.

3.1 Import dei dati

Per quanto riguarda l'ottimizzazione della query di import dei nodi, è stata utilizzata la procedura *apoc.periodic.iterate* offerta dal plugin APOC (Awesome Procedures on Cypher) per Neo4j.
Questa è utile per eseguire iterazioni complesse su grandi quantità di dati.
In particolare, tale procedura permette di selezione dei dati iniziali per poi definire una funzione Cypher che specifica le azioni da eseguire su ciascun elemento dell'insieme selezionato.

Questa funzione deve accettare un parametro di input che rappresenta l'elemento corrente nell'iterazione.
Infine, è possibile configurare dei parametri di iterazione per controllarne il comportamento. In questo caso, i parametri selezionati sono:

- *batchSize* che pecifica il numero di elementi da elaborare in ciascuna iterazione;
- *parallel*, un valore booleano che indica se le iterazioni devono essere eseguite in parallelo.

Durante l'esecuzione, la procedura *apoc.periodic.iterate* esegue la query iniziale per selezionare i dati e quindi applica la funzione di iterazione a ciascun elemento.
L'iterazione continua fino a quando non vengono elaborati tutti gli elementi selezionati dalla query iniziale.

```
CALL apoc.periodic.iterate(
  "LOAD CSV FROM 'file:///prefix_log.csv' AS row
   RETURN row",
  "CREATE (e:Event {
```

```
        activity_id: toInteger(row[0]),
        event_name: row[1],
        track_id: row[2],
        start_time: datetime(apoc.text.replace(apoc.text.replace(row[3], '\\+\\d{2}:\\d{2}$', ''), '
            ', 'T')),
        finish_time: datetime(apoc.text.replace(apoc.text.replace(row[4], '\\+\\d{2}:\\d{2}$', ''),
            ' ', 'T')),
        resource: row[5]
      })",
  {batchSize: 1000, parallel: true}
);
```

Listing 3.1: Query ottimizzata per l'import dei dati

Eseguendo la query direttamente senza utilizzare *apoc.periodic.iterate*, Neo4j carica tutti i dati in memoria prima di elaborarli. Questo può causare problemi di prestazioni soprattutto quando i dati sono tanti, poiché l'intero dataset deve essere elaborato in una sola volta.
Con *apoc.periodic.iterate*, i dati vengono elaborati a blocchi, o "batch", di dimensioni definite (in questo caso, 1000). Questo consente di ridurre l'impatto sulla memoria e di mantenere le prestazioni anche su scale grandi. Inoltre, utilizzando l'opzione *parallel: true*, è possibile sfruttare la parallelizzazione per elaborare più batch contemporaneamente, migliorando ulteriormente le prestazioni.

Anche nel caso di creazione degli archi è stata modificata l'interrogazione in modo da eseguirla tramite il supporto di *apoc.periodic.iterate* in maniera simile al caso precedente.

```
CALL apoc.periodic.iterate(
  "MATCH (start:Event), (end:Event)
   WHERE
     ((start.activity_id + 1 = end.activity_id AND start.finish_time = end.start_time)
     OR
     (
       start.finish_time = end.start_time AND
       start.start_time <> start.finish_time
       AND NOT EXISTS {
           MATCH (n:Event)
           WHERE n.activity_id > start.activity_id
             AND n.activity_id < end.activity_id
             AND n.start_time = n.finish_time
             AND n.start_time >= start.finish_time
             AND n.start_time <= end.start_time
       }
     ))
     AND start.track_id = end.track_id
   RETURN start, end",
  "MERGE (start)-[r:NEXT]->(end) SET r.connection = start.activity_id + '_' + end.
      activity_id",
  {batchSize:1000, iterateList:true}
```

```
) YIELD batches, total
RETURN batches, total
```

Listing 3.2: Query ottimizzata per la definizione degli archi

Però, a differenza di prima, in questo caso, è anche necessario definire degli indici sulle proprietà specificate nella clausola *where* della query.

```
CREATE INDEX FOR (n:Event) ON (n.activity_id);
CREATE INDEX FOR (n:Event) ON (n.start_time);
CREATE INDEX FOR (n:Event) ON (n.finish_time);
CREATE INDEX FOR (n:Event) ON (n.track_id);
```

Listing 3.3: Creazione degli indici

Gli indici così creati sono indici secondari.
Questi indici consentono di accelerare la ricerca di nodi specifici basati su determinate proprietà, in queso caso, *activity_id*, *start_time*, *finish_time* e *track_id*, all'interno del tipo di nodo *Event*.

Gli indici secondari sono particolarmente utili quando si eseguono query che filtrano i nodi in base a queste proprietà migliorando le prestazioni della query.
Quando non vengono utilizzati indici, Neo4j deve eseguire una scansione completa del grafo per trovare i nodi o le relazioni che corrispondono ai criteri della query, il che può diventare inefficiente man mano che la dimensione del grafo aumenta.
Si consideri anche che, anche se gli indici secondari possono migliorare le prestazioni delle query, possono anche avere un impatto sulle prestazioni di scrittura, poiché ogni volta che un nodo con l'indice viene creato, modificato o eliminato, l'indice deve essere aggiornato.
In questo caso, però, le uniche operazioni di scritura che vengono effettuate sono queste due iniziali che consentono di creare il grafo con la struttura desiderata.
Come risulterà evidente nelle prossime sezioni, la sola definizione di indici non è sufficiente per migliorare le prestazioni ma la combinazione dell'utilizzo di *apoc.periodic.iterate* e la definizione di indici permette di raggiungere prestazioni più che accettabili.

3.2 Generazione dei prefissi

Infine, è stata proposta una nuova versione del calcolo dei prefissi sfruttando ancora una volta *apoc* e in particolare *apoc.path.subgraphAll*.

```
MATCH (startNode:Event {activity_id: 1})
CALL apoc.path.subgraphAll(startNode, {
    minLevel: 0,
    bfs: true
})
YIELD nodes AS subgraphNodes, relationships AS subgraphRelationships
UNWIND range(1, size(subgraphNodes)) AS level
```

```
WITH subgraphNodes[..level] AS nodesSlice, subgraphRelationships[..level] AS relsSlice,
    subgraphNodes[level] as target
UNWIND relsSlice AS r
WITH nodesSlice, r, target
MATCH (n)−[r]−(m)
WHERE n IN nodesSlice AND m IN nodesSlice
RETURN nodesSlice, collect(DISTINCT r), target
```

Listing 3.4: Query per la definizione dei prefissi

Si noti innanzitutto come questa query sia molto semplice e snella nella sua definizione. Questo aspetto evidenzia ancora una volta come il linguaggio Cypher sia sintetico e facile nell'uso pur mantenendo una grande espressività.
Nonostante ciò, la query presenta alcune peculiarità importanti da evidenziare.
Innanzitutto, si noti come i prefissi e gli Instance Graph, in realtà, siano degli alberi, cioè dei particolari tipi di grafi che non contengono cicli.
I prefissi possono quindi essere definiti attraversando il grafo e considerando sempre un nodo in più.
Questo tipo di traversal del grafo è però di tipo breadth-first, ovvero, nel caso di figli multipli di un nodo padre (parallelismo), i figli devono essere toccati senza passare dal collegamento diretto tra padre e figlio ma creando un collegamento fittizio tra figlio e figlio come mostra la Figura 3.1.

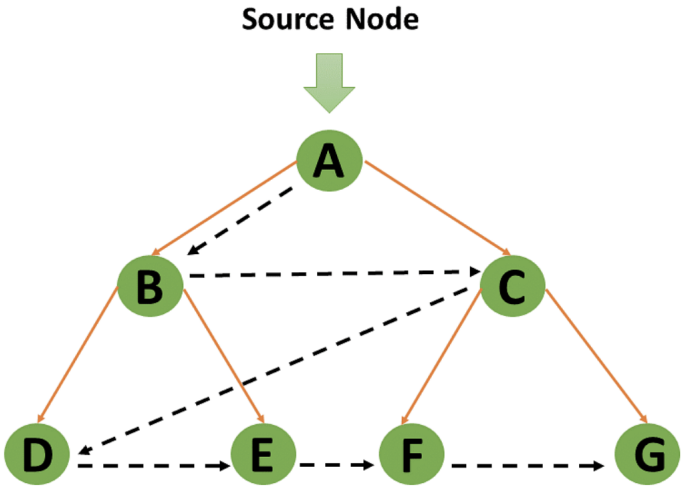


Figura 3.1: Esempio di Breadth-First Search

In definitiva, la breadth-first search analizza i nodi scandendoli per livelli.

D'altro canto, Neo4j utilizza l'algoritmo depth-first per attraversare il grafo.
La traversa depth-first (DFS) è più efficiente in termini di utilizzo della memoria rispetto alla traversa breadth-first (BFS). Questo perché DFS esplora un cammino del grafo fino alla sua profondità massima prima di tornare indietro e provare un

altro cammino. Di conseguenza, l'algoritmo mantiene in memoria solo il cammino corrente e le informazioni necessarie per tornare indietro, riducendo il numero di nodi da tenere contemporaneamente in memoria.
Inoltre, la DFS si sposa molto bene con l'esecuzione di pattern matching su grafi su cui si basa Neo4j, dove si cerca un particolare sotto-grafo o una sequenza di nodi e relazioni. Questo approccio permette di esaminare tutte le possibili vie in profondità per trovare un match, prima di esplorare vie alternative.

Il problema, in questo caso è che utilizzando una ricerca di tipo DFS non è possibile ottenere il risultato desiderato.
A tale scopo, è stata utilizzata la funzione *subgraphAll* della libreria *apoc* che permette di eseguire una BFS su grafi all'interno di Neo4j.
Questa soluzione, così implementata, funziona correttamente ritornando i prefissi di tutti i grafi con associata la loro etichetta.

Capitolo 4

Esperimenti

In questa sezione vengono riportati gli esperimenti effettuati per confrontare le prestazioni nel caso di query ottimizzate e non, in modo da comprendere i trade-off utili a raggirare i colli di bottiglia del sistema.

Nella fase iniziale, le query sono state condotte utilizzando un dataset di dimensioni ridotte. Questo approccio aveva l'obiettivo di valutare la correttezza e ottenere risultati in tempi accettabili. È stato quindi possibile verificare che le operazioni funzionassero correttamente e senza errori significativi.
Una volta stabilita la correttezza delle query introdotte lavorando con un dataset ridotto, l'attenzione si è spostata verso la gestione di dataset reali, di grandi dimensioni.

4.1 Import dei dati

Sono state condotte delle prove per valutare come il sistema scala quando il numero di grafi aumenta per quello che riguarda la query di import dei dati. È importante notare che prima di ogni esecuzione delle query di caricamento dei dati, la cache è stata pulita al fine di avere risultati coerenti.

4.1.1 Query non ottimizzata

Per condurre queste prove, sono stati utilizzati dei file contenenti rispettivamente 10, 100, 200, 500, 1000 e 2000 grafi come campioni rappresentativi della dimensione del dataset. I risultati del test sono riportati in tabella 4.1.

Si può notare che per i primi casi sono state eseguite più prove, mentre per il file contenente 2000 grafi è stata eseguita una sola prova. Questa scelta è determinata dal notevole aumento del tempo di caricamento. Ripetere più prove anche per questo file avrebbe richiesto molto tempo, mentre il trend di crescita era già evidente. Pertanto, è stata presa la decisione di eseguire una sola prova, anche tenendo in considerazione il fatto che i risultati ottenuti facendo due prove distinte nei primi casi sono molto simili tra di loro.

Numero di grafi	Tempo (prova 1)	Tempo (prova 2)	Numero di nodi
10	2.988961 s	2.474231 s	223
100	7.877312 s	7.668197 s	2385
200	21.569412 s	22.842203 s	4859
500	128.600391 s	126.983455 s	12409
1000	498.800449 s	472.750969 s	23902
2000	1555.826346 s	-	46452

Tabella 4.1: Tempi di caricamento dei file

I tempi di caricamento indicano che il tempo necessario per importare i dati aumenta significativamente con l'aumentare del numero di nodi totali come è possibile osservare nella Figura 4.1.

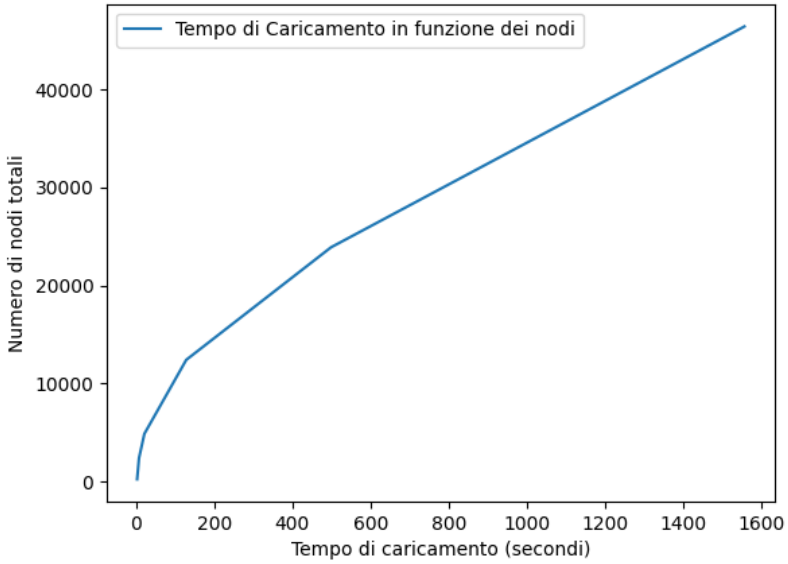


Figura 4.1: Tempo di Caricamento al variare del numero di nodi

Ad esempio, per caricare un file con solo 10 grafi, il tempo richiesto è di circa 3 secondi e per 100 grafi, il tempo è di circa 8 secondi. Tuttavia, quando il numero di grafi aumenta, i tempi iniziano a crescere e il salto più significativo si verifica con il file contenente 2000 grafi.

Questo confronto evidenzia chiaramente come, mentre il caricamento di piccoli dataset è relativamente rapido e conveniente, l'aumento del numero dei grafi considerato comporta un notevole aumento del tempo. Questa crescita non è lineare e ciò suggerisce che il sistema potrebbe avere delle difficoltà a gestire carichi di lavoro più elevati in modo efficiente. Infatti, va notato che il dataset completo di BPI12 contiene 13087 grafi. È evidente che l'approccio attuale potrebbe non essere la scelta migliore per gestire dataset di queste dimensioni.

Per valutare le prestazioni del sistema, si è deciso di non considerare solo l'aspetto temporale ma anche l'utilizzo della CPU e della memoria. Nella Figura 4.2 si riporta l'utilizzo di queste risorse durante la prova di caricamento di 2000 grafi.

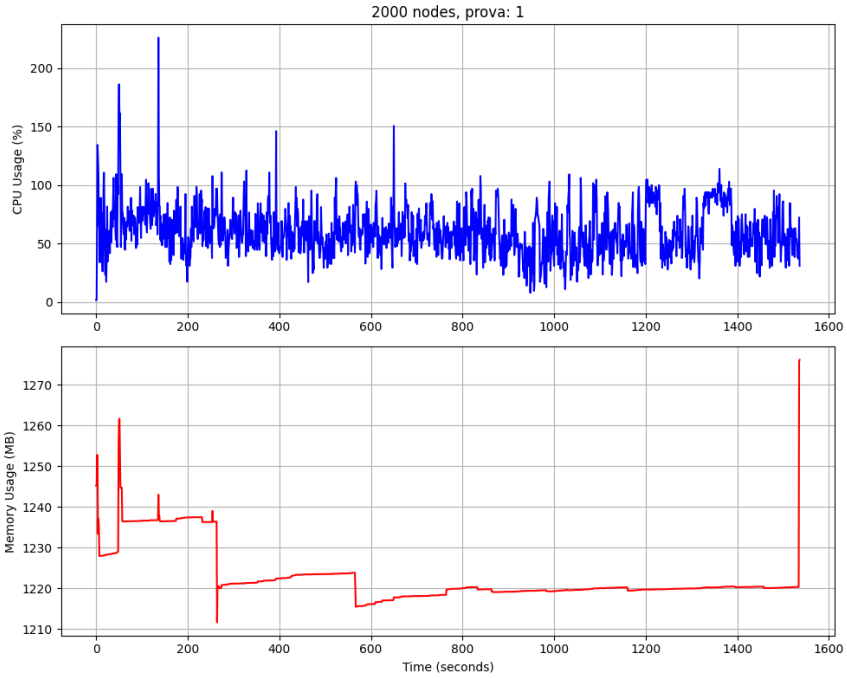


Figura 4.2: Utilizzo di memoria e CPU durante il caricamento di 2000 grafi

Durante l'esecuzione di tutti gli esperimenti finora citati, ed ovviamente anche per il caricamento del file composto da 2000 grafi, si è notato che la memoria massima utilizzata rimaneva più o meno stabile intorno ad 1GB. Questo risultato ha sollevato degli interrogativi sulla configurazione delle impostazioni della memoria di Neo4j. Esaminando le impostazioni di Neo4j, è stato scoperto che ciò è dovuto ad una configurazione che limita l'uso massimo della memoria:

- dbms.memory.heap.initial_size = 512m
- dbms.memory.heap.max_size = 1G
- dbms.memory.pagecache.size = 512m

L'uso della memoria inizia a circa $1220MB$ e si mantiene relativamente stabile per una buona parte del tempo di caricamento.Verso la fine del processo, si osservano alcuni incrementi graduali nell'uso della memoria, ma non sono estremamente pronunciati. Questo può indicare che la configurazione corrente della memoria è sufficiente per gestire il carico di lavoro iniziale, ma può diventare limitante man mano che il processo di caricamento avanza.
Per quanto riguarda la CPU, all'inizio del processo di caricamento, si notano picchi

significativi che raggiungono e superano il 200%. Questo significa che due cores sono completamente sfruttati.
Dopo i picchi iniziali, l'utilizzo della CPU si stabilizza intorno al 100%, ma con notevoli fluttuazioni nel tempo. Queste fluttuazioni possono indicare che la CPU sta lavorando intensamente per gestire operazioni di I/O e calcolo necessarie per il caricamento e la verifica dei dati.

La configurazione corrente sembra essere al limite della capacità di memoria disponibile, causando picchi e fluttuazioni nell'utilizzo della CPU. Questo suggerisce che la CPU sta spendendo un tempo significativo in operazioni di I/O per compensare la mancanza di memoria cache disponibile. Le fluttuazioni e i picchi nell'uso della CPU possono indicare che il sistema non è in grado di mantenere un throughput costante, il che può portare a tempi di caricamento complessivamente più lunghi e meno prevedibili.

Visto che la memoria in tutti questi esperimenti rimane stabile perchè configurata tramite parametri specifici in Neo4j che limitano l'uso massimo della memoria, si è deciso di fare un altro tentativo. Infatti, queste impostazioni limitate potrebbero aver influenzato le prestazioni del sistema durante il caricamento dei dati. Per valutare l'impatto delle impostazioni di memoria sulle prestazioni del sistema, sono state apportate delle modifiche come segue:

- dbms.memory.heap.initial_size = $2G$
- dbms.memory.heap.max_size = $4G$
- dbms.memory.pagecache.size = $2G$

Il test con le impostazioni di memoria ottimizzate ha mostrato un significativo miglioramento delle prestazioni del sistema. In particolare, il tempo di caricamento del file contenente 2000 grafi è stato ridotto da 1555.83 secondi a 795.19 secondi.

Questi risultati indicano chiaramente che la configurazione delle impostazioni di memoria può avere un impatto significativo sulle prestazioni complessive del sistema di gestione dei dati. L'aumento delle dimensioni della memoria heap e della cache delle pagine ha consentito al sistema di elaborare i dati in modo più efficiente e veloce durante il processo di caricamento.

Nella Figura 4.3 è riportato l'utilizzo della memoria e della CPU.

Questa configurazione con maggiore memoria mostra un utilizzo della CPU più efficiente e meno fluttuante nel tempo rispetto alla configurazione precedente (Figura 4.2). Questo indica che una maggiore quantità di memoria disponibile riduce la necessità di operazioni di I/O disco intensive, permettendo alla CPU di lavorare più

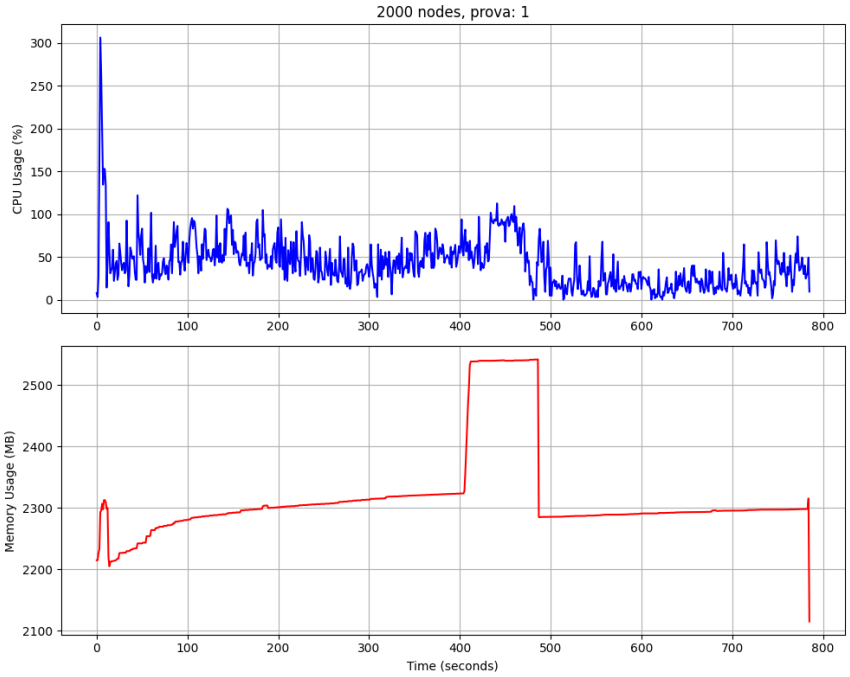


Figura 4.3: Utilizzo di memoria e CPU durante il caricamento di 2000 grafi dopo aver cambiato le impostazioni di memoria

efficacemente.
Inoltre, con questa combinazione di parametri, l'uso della memoria è più elevato e aumenta più significativamente durante il caricamento dei grafi. Questo è atteso, dato che Neo4j può utilizzare la memoria aggiuntiva per mantenere più dati in memoria cache, riducendo le operazioni di I/O e migliorando le prestazioni complessive.
In conclusione, questa prova sembra portare ad un utilizzo della CPU più stabile e ad un caricamento potenzialmente più veloce, dato che le risorse di sistema sono utilizzate in modo più efficiente.

Un'ulteriore analisi che si è voluta svolgere ha mirato a determinare l'efficacia di suddividere i dati in file più piccoli rispetto al caricamento di un unico file contenente tutti i dati. Per condurre l'esperimento, sono stati utilizzati file di dati contenenti un totale di 2000 grafi. Sono stati eseguiti tre diversi test:

1. Caricamento di un singolo file contenente 2000 grafi.
2. Suddivisione dei dati in due file, ciascuno contenente 1000 grafi.
3. Suddivisione dei dati in quattro file, ciascuno contenente 500 grafi.

Ogni test è stato eseguito per valutare il tempo totale di caricamento dei dati e confrontare le prestazioni tra le diverse modalità di suddivisione dei dati. Nella Tabella 4.2 sono riportati i risultati ottenuti.

Tempo di caricamento	Caricamento split 500	Caricamento split 1000
Primo file	102.730455 s	451.573523 s
Secondo file	323.030608 s	1129.107650 s
Terzo file	502.273653 s	-
Quarto file	738.325350 s	-
Totale	1666.361054 s	1580.682174 s

Tabella 4.2: Tempi di caricamento dei file splittati

L'analisi dei risultati suggerisce che suddividere i dati in file più piccoli ha comportato un aumento del tempo totale di caricamento, seppur minimo, rispetto al caricamento di un singolo file con lo stesso numero di grafi. In particolare, suddividere i dati in due file da 1000 grafi ciascuno ha mostrato un tempo totale di caricamento leggermente superiore rispetto al caricamento di un unico file da 2000 grafi e la suddivisione dei dati in quattro file da 500 grafi ciascuno ha mostrato il tempo totale di caricamento più lungo. Pertanto, caricare un singolo file può essere più efficiente perchè Neo4j può gestire le operazioni senza interruzioni, mentre caricare file separati implica overhead aggiuntivi per ogni operazioni di I/O.
Se ciò è evidente già dal caricamento di un sottoinsieme di grafi, a maggior ragione lo sarà per l'intero dataset.

Un'ulteriore osservazione riguarda il fatto che il tempo di caricamento dei file è aumentato con l'incremento del numero di file caricati. Infatti, ci si aspettava che il tempo di caricamento di ogni file fosse sempre lo stesso, invece dai tempi riportati in Tabella 4.2 si può notare che ogni volta che viene inserito un ulteriore file, il tempo di caricamento aumenta notevolmente rispetto al tempo del file precedente.

4.1.2 Query ottimizzata

Visto i notevoli miglioramenti ottenuti dopo aver cambiato le impostazioni predefinite di Neo4j per la memoria, si è deciso di svolgere tutte le seguenti prove con tali configurazioni.
Si è testata l'efficienza della query ottimizzata per l'import dei dati. Caricando il file contenente 13087 grafi sono stati sufficienti $12,76s$. In confronto, dalle prove precedenti, già con file contenenti appena 200 grafi, il tempo di caricamento richiesto era maggiore.
Questo miglioramento drastico nelle performance è stato ottenuto grazie ad un migliore utilizzo delle caratteristiche di Neo4j.

L'utilizzo dell'operatore $MERGE$ è molto più lento dell'operatore $CREATE$, specialmente quando si lavora con grandi volumi di dati perchè deve verificare l'esistenza di ogni nodo prima di crearne uno nuovo evitando in questo modo duplicati. Tuttavia,

se si sa a priori che le proprietà dei nodi sono univoche, si può evitare questo controllo. Inoltre, combinare *CREATE* con la parallelizzazione offerta da *apoc.periodic.iterate* consente di caricare grandi volumi di dati in modo molto più rapido ed efficiente, come dimostrato dal significativo miglioramento ottenuto.

Per comprendere effettivamente i vantaggi offerti dall'utilizzo della query ottimizzata, sono stati messi a confronto i grafici di Figura 4.4 e 4.2 ricavati sull'utilizzo della CPU e della memoria da parte di entrambe le interrogazioni.

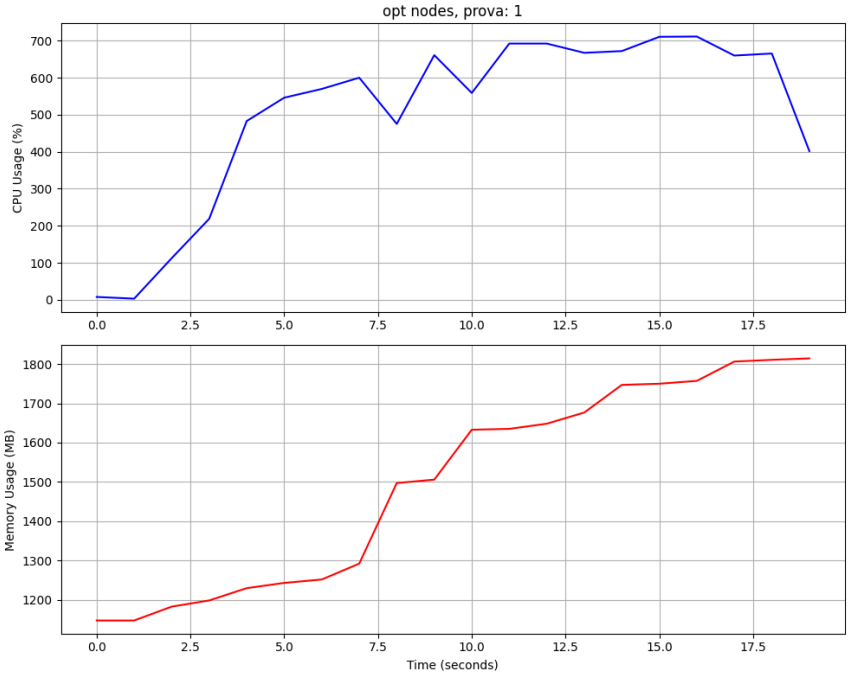


Figura 4.4: Utilizzo di memoria e CPU durante il caricamento di tutti i 13087 grafi tramite la query ottimizzata

Si noti come il grafico di Figura 4.2, all'inizio, mostra un utilizzo della CPU che aumenta significativamente, a volte superando il 200%. Questo probabilmente accade perché Neo4j cerca di elaborare un grande volume di dati senza batching, utilizzando tutte le risorse della CPU disponibili. Dopo i picchi iniziali, l'utilizzo della CPU fluttua ma rimane generalmente alto, indicando un'elaborazione intensiva in corso. Questo è caratteristico di un'operazione di importazione non batched e non parallela dove ogni dato viene elaborato sequenzialmente. Anche per quanto riguarda la memoria sono presenti picchi iniziali che poi si stabilizzano in un modello più coerente. Questo indica che il sistema sta allocando memoria per gestire l'importazione dei dati.
Invece, nel caso della query ottimizzata, il grafico di Figura 4.4 mostra come l'uso della CPU inizia basso e aumenta gradualmente, stabilizzandosi infine intorno al

$600 - 700\%$. Questo indica che la query sta utilizzando efficacemente più core della CPU grazie all'elaborazione parallela abilitata dalla chiamata *apoc.periodic.iterate* con esecuzione parallela. Inoltre, l'utilizzo della CPU rimane alto e costante durante tutto il processo di importazione, cosa attesa con il processamento batch in parallelo. Anche l'utilizzo della memoria inizia basso e aumenta costantemente durante il processo di importazione. Questo è previsto poiché l'elaborazione batch gestisce efficientemente la memoria, riducendo la probabilità di grandi picchi.

In definitiva, la query ottimizzata con *apoc.periodic.iterate* è progettata per migliorare le prestazioni suddividendo l'importazione dei dati in batch ed elaborandoli in parallelo.
Questo si traduce in un uso più consistente ed efficiente sia della CPU che della memoria. L'alto utilizzo della CPU nell'approccio ottimizzato è dovuto alla parallellizzazione efficace, mentre l'aumento costante dell'uso della memoria indica una sua gestione efficiente. La query non ottimizzata, al contrario, mostra inefficienze con picchi iniziali più alti della CPU e modelli di utilizzo della memoria meno prevedibili.

4.2 Creazione degli archi

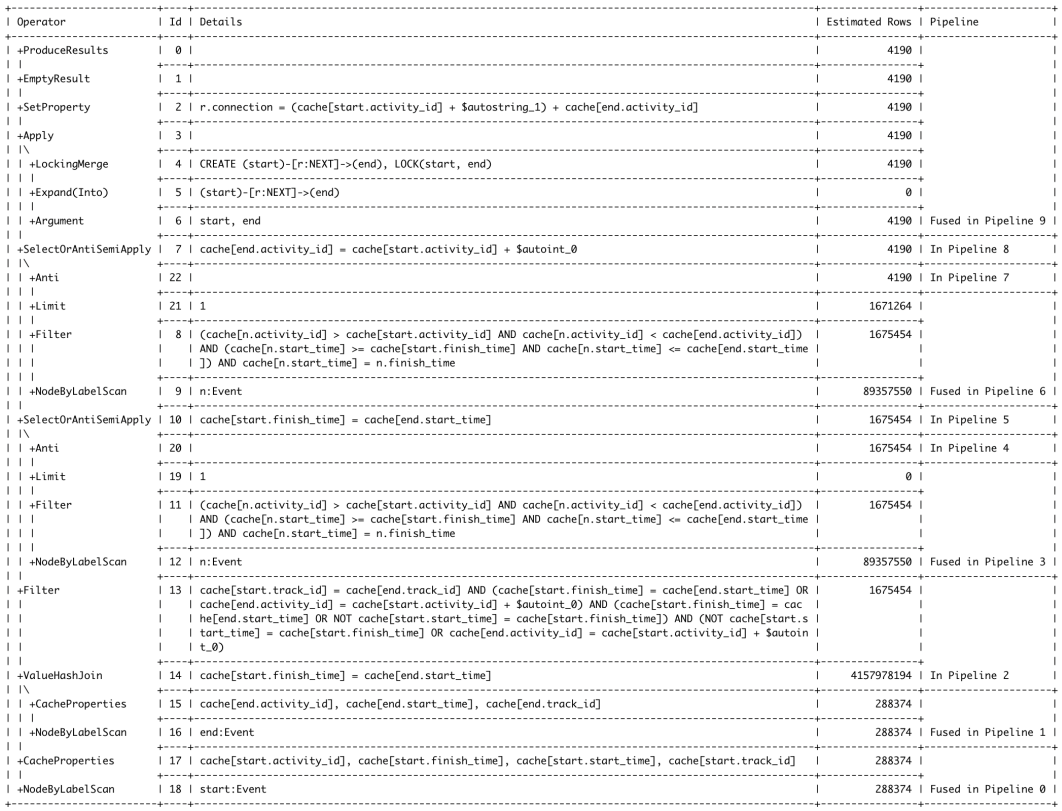
Per quanto riguarda la creazione degli archi, l'approccio ottimizzato ha permesso di ottenere risultati in tempi accettabili.
In particolare, la query ottimizzata crea gli archi (280141 archi) in $448,14$ secondi, ovvero, $7,5$ minuti. Invece, sia utilizzando la query riportata nel capitolo 2, sia utilizzando la query ottimizzata ma senza la definizione di indici, i tempi di caricamento sono risultati non calcolabili in quanto troppo prolungati.

Per analizzare le prestazioni di queste query è stato utilizzato il comando *EXPLAIN* di Neo4j che, se posto all'inizio di una query, permette di fornire informazioni dettagliate su come il database esegue una query, inclusi i nodi e le relazioni esplorati durante l'esecuzione, i tempi di esecuzione delle varie operazioni e altre statistiche importanti.
Tali informazioni sono riportate in Figura 4.5 e 4.6.

```
+-------------------------+----+-----------------------------------------------------------------------------------------------------------+----------------+--------------------+
| Operator                | Id | Details                                                                                                   | Estimated Rows | Pipeline           |
+-------------------------+----+-----------------------------------------------------------------------------------------------------------+----------------+--------------------+
| +ProduceResults         |  0 |                                                                                                           |              0 |                    |
| |                       +----+-----------------------------------------------------------------------------------------------------------+----------------+                    |
| +EmptyResult            |  1 |                                                                                                           |              0 |                    |
| |                       +----+-----------------------------------------------------------------------------------------------------------+----------------+                    |
| +SetProperty            |  2 | r.connection = (cache[start.activity_id] + $autostring_1) + cache[end.activity_id]                        |              0 |                    |
| |                       +----+-----------------------------------------------------------------------------------------------------------+----------------+                    |
| +Apply                  |  3 |                                                                                                           |              0 |                    |
| |\                      +----+-----------------------------------------------------------------------------------------------------------+----------------+                    |
| | +LockingMerge         |  4 | CREATE (start)-[r:NEXT]->(end), LOCK(start, end)                                                          |              0 |                    |
| | |                     +----+-----------------------------------------------------------------------------------------------------------+----------------+                    |
| | +Expand(Into)         |  5 | (start)-[r:NEXT]->(end)                                                                                   |              0 |                    |
| | |                     +----+-----------------------------------------------------------------------------------------------------------+----------------+                    |
| | +Argument             |  6 | start, end                                                                                                |              0 | Fused in Pipeline 8 |
| |                       +----+-----------------------------------------------------------------------------------------------------------+----------------+--------------------+
| +SelectOrAntiSemiApply  |  7 | cache[end.activity_id] = cache[start.activity_id] + $autoint_0                                            |              0 | In Pipeline 7      |
| |\                      +----+-----------------------------------------------------------------------------------------------------------+----------------+--------------------+
| | +Anti                 | 21 |                                                                                                           |              0 | In Pipeline 6      |
| | |                     +----+-----------------------------------------------------------------------------------------------------------+----------------+--------------------+
| | +Limit                | 20 | 1                                                                                                         |              0 |                    |
| | |                     +----+-----------------------------------------------------------------------------------------------------------+----------------+                    |
| | +Filter               |  8 | (cache[n.start_time] >= cache[start.finish_time] AND cache[n.start_time] <= cache[end.start_time]) A      |              0 |                    |
| | |                     |    | ND cache[n.start_time] = n.finish_time                                                                    |                |                    |
| | |                     +----+-----------------------------------------------------------------------------------------------------------+----------------+                    |
| | +NodeIndexSeekByRange |  9 | RANGE INDEX n:Event(activity_id) WHERE activity_id > cache[start.activity_id] AND activity_id < cach      |              0 | Fused in Pipeline 5 |
| |                       |    | e[end.activity_id]                                                                                        |                |                    |
| |                       +----+-----------------------------------------------------------------------------------------------------------+----------------+--------------------+
| +SelectOrAntiSemiApply  | 10 | cache[start.finish_time] = cache[end.start_time]                                                          |              0 | In Pipeline 4      |
| |\                      +----+-----------------------------------------------------------------------------------------------------------+----------------+--------------------+
| | +Anti                 | 19 |                                                                                                           |              0 | In Pipeline 3      |
| | |                     +----+-----------------------------------------------------------------------------------------------------------+----------------+--------------------+
| | +Limit                | 18 | 1                                                                                                         |              0 |                    |
| | |                     +----+-----------------------------------------------------------------------------------------------------------+----------------+                    |
| | +Filter               | 11 | (cache[n.start_time] >= cache[start.finish_time] AND cache[n.start_time] <= cache[end.start_time]) A      |              0 |                    |
| | |                     |    | ND cache[n.start_time] = n.finish_time                                                                    |                |                    |
| | |                     +----+-----------------------------------------------------------------------------------------------------------+----------------+                    |
| | +NodeIndexSeekByRange | 12 | RANGE INDEX n:Event(activity_id) WHERE activity_id > cache[start.activity_id] AND activity_id < cach      |              0 | Fused in Pipeline 2 |
| |                       |    | e[end.activity_id]                                                                                        |                |                    |
| |                       +----+-----------------------------------------------------------------------------------------------------------+----------------+--------------------+
| +Filter                 | 13 | start.track_id = cache[end.track_id] AND (cache[start.finish_time] = cache[end.start_time] OR cache[      |              0 |                    |
| |                       |    | end.activity_id] = cache[start.activity_id] + $autoint_0) AND (cache[start.finish_time] = cache[end.      |                |                    |
| |                       |    | start_time] OR NOT cache[start.start_time] = cache[start.finish_time]) AND (NOT cache[start.start_ti      |                |                    |
| |                       |    | me] = cache[start.finish_time] OR cache[end.activity_id] = cache[start.activity_id] + $autoint_0)         |                |                    |
| |                       +----+-----------------------------------------------------------------------------------------------------------+----------------+                    |
| +Apply                  | 14 |                                                                                                           |         349492 |                    |
| |\                      +----+-----------------------------------------------------------------------------------------------------------+----------------+                    |
| | +NodeIndexSeek        | 15 | RANGE INDEX start:Event(finish_time) WHERE finish_time = cache[end.start_time], cache[start.finish_t      |         349492 | Fused in Pipeline 1 |
| |                       |    | ime]                                                                                                      |                |                    |
| |                       +----+-----------------------------------------------------------------------------------------------------------+----------------+--------------------+
| +CacheProperties        | 16 | cache[end.activity_id], cache[end.track_id]                                                               |         288374 |                    |
| |                       +----+-----------------------------------------------------------------------------------------------------------+----------------+                    |
| +NodeByLabelScan        | 17 | end:Event                                                                                                 |         288374 | Fused in Pipeline 0 |
+-------------------------+----+-----------------------------------------------------------------------------------------------------------+----------------+--------------------+
```

Figura 4.5: Esecuzione della query con la definizione di indici

Per ottenere informazioni di profiling dettagliate, è stato necessario eseguire direttamente la query senza *apoc.periodic.iterate* che maschera l'esecuzione della funzione di iterazione interna.
Questo non signfica che la query verrà eseguita senza l'utilizzo della procedura



```
+-----------------------+----+-----------------------------------------------------------------------------------------------------------------------+----------------+-----------------------+
| Operator              | Id | Details                                                                                                               | Estimated Rows | Pipeline              |
+-----------------------+----+-----------------------------------------------------------------------------------------------------------------------+----------------+-----------------------+
| +ProduceResults       |  0 |                                                                                                                       |           4190 |                       |
| |                     +----+-----------------------------------------------------------------------------------------------------------------------+----------------+                       |
| +EmptyResult          |  1 |                                                                                                                       |           4190 |                       |
| |                     +----+-----------------------------------------------------------------------------------------------------------------------+----------------+                       |
| +SetProperty          |  2 | r.connection = (cache[start.activity_id] + $autostring_1) + cache[end.activity_id]                                    |           4190 |                       |
| |                     +----+-----------------------------------------------------------------------------------------------------------------------+----------------+                       |
| +Apply                |  3 |                                                                                                                       |           4190 |                       |
| |\                    +----+-----------------------------------------------------------------------------------------------------------------------+----------------+                       |
| | +LockingMerge       |  4 | CREATE (start)-[r:NEXT]->(end), LOCK(start, end)                                                                      |           4190 |                       |
| | |                   +----+-----------------------------------------------------------------------------------------------------------------------+----------------+                       |
| | +Expand(Into)       |  5 | (start)-[r:NEXT]->(end)                                                                                               |              0 |                       |
| | |                   +----+-----------------------------------------------------------------------------------------------------------------------+----------------+                       |
| | +Argument           |  6 | start, end                                                                                                            |           4190 | Fused in Pipeline 9   |
| |                     +----+-----------------------------------------------------------------------------------------------------------------------+----------------+-----------------------+
| +SelectOrAntiSemiApply |  7 | cache[end.activity_id] = cache[start.activity_id] + $autoint_0                                                        |           4190 | In Pipeline 8         |
| |\                    +----+-----------------------------------------------------------------------------------------------------------------------+----------------+-----------------------+
| | +Anti               | 22 |                                                                                                                       |           4190 | In Pipeline 7         |
| | |                   +----+-----------------------------------------------------------------------------------------------------------------------+----------------+-----------------------+
| | +Limit              | 21 | 1                                                                                                                     |        1671264 |                       |
| | |                   +----+-----------------------------------------------------------------------------------------------------------------------+----------------+                       |
| | +Filter             |  8 | (cache[n.activity_id] > cache[start.activity_id] AND cache[n.activity_id] < cache[end.activity_id])                   |        1675454 |                       |
| | |                   |    | AND (cache[n.start_time] >= cache[start.finish_time] AND cache[n.start_time] <= cache[end.start_time                  |                |                       |
| | |                   |    | ]) AND cache[n.start_time] = n.finish_time                                                                            |                |                       |
| | |                   +----+-----------------------------------------------------------------------------------------------------------------------+----------------+                       |
| | +NodeByLabelScan    |  9 | n:Event                                                                                                               |       89357550 | Fused in Pipeline 6   |
| |                     +----+-----------------------------------------------------------------------------------------------------------------------+----------------+-----------------------+
| +SelectOrAntiSemiApply | 10 | cache[start.finish_time] = cache[end.start_time]                                                                      |        1675454 | In Pipeline 5         |
| |\                    +----+-----------------------------------------------------------------------------------------------------------------------+----------------+-----------------------+
| | +Anti               | 20 |                                                                                                                       |        1675454 | In Pipeline 4         |
| | |                   +----+-----------------------------------------------------------------------------------------------------------------------+----------------+-----------------------+
| | +Limit              | 19 | 1                                                                                                                     |              0 |                       |
| | |                   +----+-----------------------------------------------------------------------------------------------------------------------+----------------+                       |
| | +Filter             | 11 | (cache[n.activity_id] > cache[start.activity_id] AND cache[n.activity_id] < cache[end.activity_id])                   |        1675454 |                       |
| | |                   |    | AND (cache[n.start_time] >= cache[start.finish_time] AND cache[n.start_time] <= cache[end.start_time                  |                |                       |
| | |                   |    | ]) AND cache[n.start_time] = n.finish_time                                                                            |                |                       |
| | |                   +----+-----------------------------------------------------------------------------------------------------------------------+----------------+                       |
| | +NodeByLabelScan    | 12 | n:Event                                                                                                               |       89357550 | Fused in Pipeline 3   |
| |                     +----+-----------------------------------------------------------------------------------------------------------------------+----------------+-----------------------+
| +Filter               | 13 | cache[start.track_id] = cache[end.track_id] AND (cache[start.finish_time] = cache[end.start_time] OR                  |        1675454 |                       |
| |                     |    | cache[end.activity_id] = cache[start.activity_id] + $autoint_0) AND (cache[start.finish_time] = cac                   |                |                       |
| |                     |    | he[end.start_time] OR NOT cache[start.start_time] = cache[start.finish_time]) AND (NOT cache[start.s                  |                |                       |
| |                     |    | tart_time] = cache[start.finish_time] OR cache[end.activity_id] = cache[start.activity_id] + $autoin                  |                |                       |
| |                     |    | t_0)                                                                                                                  |                |                       |
| |                     +----+-----------------------------------------------------------------------------------------------------------------------+----------------+                       |
| +ValueHashJoin        | 14 | cache[start.finish_time] = cache[end.start_time]                                                                      |     4157978194 | In Pipeline 2         |
| |\                    +----+-----------------------------------------------------------------------------------------------------------------------+----------------+-----------------------+
| | +CacheProperties    | 15 | cache[end.activity_id], cache[end.start_time], cache[end.track_id]                                                    |         288374 |                       |
| | |                   +----+-----------------------------------------------------------------------------------------------------------------------+----------------+                       |
| | +NodeByLabelScan    | 16 | end:Event                                                                                                             |         288374 | Fused in Pipeline 1   |
| |                     +----+-----------------------------------------------------------------------------------------------------------------------+----------------+-----------------------+
| +CacheProperties      | 17 | cache[start.activity_id], cache[start.finish_time], cache[start.start_time], cache[start.track_id]                    |         288374 |                       |
| |                     +----+-----------------------------------------------------------------------------------------------------------------------+----------------+                       |
| +NodeByLabelScan      | 18 | start:Event                                                                                                           |         288374 | Fused in Pipeline 0   |
+-----------------------+----+-----------------------------------------------------------------------------------------------------------------------+----------------+-----------------------+
```

Figura 4.6: Esecuzione della query senza la definizione di indici

apoc.periodic.iterate di cui sono già stati discussi i vantaggi prestazionali.

Osservando le due figure, è lampante che il piano con gli indici mostra operazioni di ricerca significativamente più efficienti utilizzando *NodeIndexSeekByRange*, rispetto alle operazioni di *NodeByLabelScan* nel piano senza indici.
Inoltre, tramite l'utilizzo di indici, il filtraggio è molto più mirato e avviene prima nella pipeline, sfruttando le proprietà indicizzate per ridurre rapidamente il dataset. Senza indici, il filtraggio avviene più tardi e opera su un dataset iniziale molto più grande. Questo è possibile notarlo in particolare osservando il numero di righe calcolato ad ogni passaggio: gli indici sono molto efficaci per restringere rapidamente i dati rilevanti. Il piano senza indici, infatti, mostra costantemente alti numeri di righe fino alle fasi finali, indicando ricerche più ampie e meno efficienti.

In particolare, si osservi che il numero elevato di *ValueHashJoin* nel piano di esecuzione senza indici evidenzia un'inefficienza significativa. Senza indici, il database è costretto a eseguire numerosi join sui valori, risultando in un'enorme espansione del numero di righe da elaborare. Questo non solo rallenta l'esecuzione della query ma consuma anche molte risorse.
L'adozione di indici elimina la necessità di *ValueHashJoin*, permettendo ricerche

più mirate e riducendo drasticamente il numero di righe elaborate a ogni passaggio. Di conseguenza, il piano di esecuzione con indici è molto più efficiente e veloce, dimostrando l'importanza degli indici nella gestione di database complessi come Neo4j.

In definitiva, l'utilizzo degli indici in Neo4j migliora drasticamente le prestazioni delle query, permettendo ricerche di nodi più efficienti e filtraggi anticipati, riducendo il totale dei dati che devono essere elaborati a ogni passaggio. Il piano di esecuzione con gli indici dimostra una pipeline più snella e mirata, risultando in tempi di esecuzione della query significativamente più rapidi.

4.3 Generazione dei prefissi

Anche nel caso della generazione dei prefissi sono state eseguite più prove per valutare le prestazioni delle varie soluzioni individuate.
Oltre alle query precedentemente discusse, è stato anche considerato un terzo approccio che calcola i prefissi direttamente in Python utilizzando la libreria NetworkX senza passare per il tramite di Neo4j.
I risultati ottenuti sono riportati nella Tabella 4.3.

Numero di grafi	Non ottimizzato	BFS	NetworkX
100	16992.956984 s	16.583124 s	7.725920 s
200	-	32.132365 s	24.508113 s
13087	-	2640.634545 s	199.608140 s

Tabella 4.3: Tempi per la generazione dei prefissi

Ciò che si può notare è che l'approccio non ottimizzato ha comportato un tempo di esecuzione notevolmente superiore ($4,72$ ore contro 16 e 7 secondi). Per questo motivo, è stata effettuata una sola prova con questo metodo, mentre negli altri casi sono state svolte più prove, mostrando risultati simili, sebbene la query in Cypher abbia riportato sempre dei tempi leggermente superiori.

Nella Figura 4.7 è riportato un grafico trimensionale dove l'asse X indica il numero di grafi trovati, l'asse Y indica la lunghezza del prefisso e l'asse Z indica il tempo impiegato per calcolare il prefisso misurato in secondi.

Come si evince dal grafico, il tempo di calcolo del prefisso aumenta con la lunghezza del prefisso stesso e tende a diminuire con il numero di grafi rispetto a cui effettuare il calcolo.
In particolare, per comprendere la relazione tra queste tre variabili, si considerino due casi estremi:

1. calcolo dei prefissi di lunghezza 1 (100 prefissi da calcolare);

2. calcolo dell'unico prefisso di lunghezza 117.

Nel primo caso, il tempo di calcolo è ridotto.
D'altro canto, nel secondo caso, poichè in questa porzione di dataset è presente un unico grafo da 117 nodi, i nodi da ritornare sono circa quelli della prima interrogazione. Nonostante ciò, il tempo di calcolo non si riduce così tanto come avviene nel primo caso.
Questo è sicuramente dovuto al fatto che, oltre che il retrieval dei nodi, la query effettua anche la ricostruzione di tutti gli archi che si trovano tra di essi.
Nel primo caso, questa ricostruzione non è da eseguire in quanto ogni grafo contiene un unico nodo. Nel secondo caso, invece, la ricostruzione degli archi è necessaria.

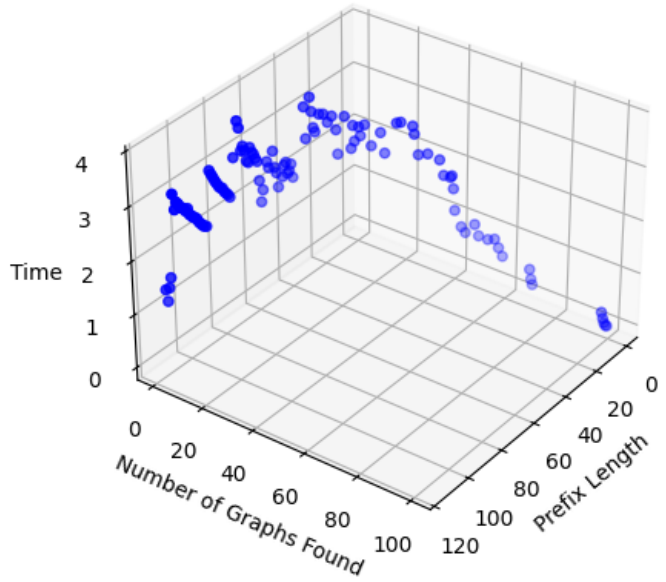


Figura 4.7: Tempo di calcolo dei prefissi per lunghezza del prefisso e numerosità di grafi

In definitiva, la differenza nei tempi di esecuzione delle query può essere attribuita a diversi fattori legati all'ottimizzazione delle query, alla struttura della base di dati e all'efficienza delle operazioni che ciascuna query esegue.
In particolare, la query non ottimizzata, ad ogni ciclo for richiede di eseguire diverse operazioni complesse. L'utilizzo dell'operatore $UNWIND$ può risultare molto inefficiente nel momento in cui non si lavora con dataset di piccole dimensioni.

Invece, la query che si basa sull'utilizzo dell'approccio BFS tramite *apoc.path.subgraphAll* è altamente ottimizzata per esplorare graficamente i nodi e le relazioni. Le procedure $APOC$ sono progettate per essere altamente performanti e sfruttano ottimizzazioni

interne e parallelismo, che potrebbero non essere presenti nell'altro caso. Per questo motivo, utilizzare la procedura di *APOC* può essere molto più efficiente per trovare i nodi e le relazioni in un sottografo rispetto alle rispettive operazioni manuali che sono svolte nella query dinamica.

Infine, il calcolo dei prefissi tramite Python è stato effettuato mediante la libreria *NetworkX* che fornisce una vasta gamma di funzionalità per la manipolazione di grafi. *NetworkX* è potente perché offre un equilibrio tra facilità d'uso, flessibilità e performance e in questo modo è uno strumento ideale per lavorare con grafi in Python.
I tempi in Python risultano migliori quando il dataset aumenta di dimensione.

In particolare, i seguenti grafici mostrano l'utilizzo della CPU e della memoria da parte del calcolo dei prefissi tramite la query che sfrutta l'approccio BFS (Figura 4.8) e tramite il calcolo dei prefissi svolto in Python mediante la libreria NetworkX (Figura 4.9).

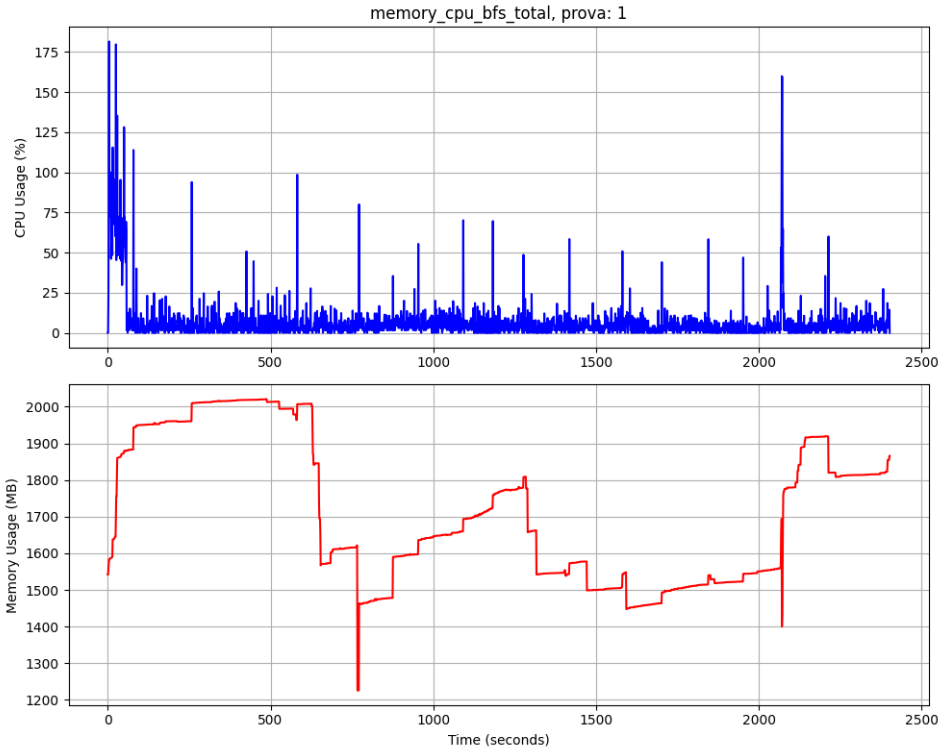


Figura 4.8: Utilizzo di memoria e CPU durante il calcolo dei prefissi con BFS

Nel grafico di figura 4.9, l'andamento del consumo di CPU mostra una crescita costante che può indicare un incremento della complessità computazionale man mano che l'algoritmo elabora i dati. La memoria, similmente, mostra un aumento continuo, il che è indicativo di un accumulo progressivo di dati durante l'esecuzione

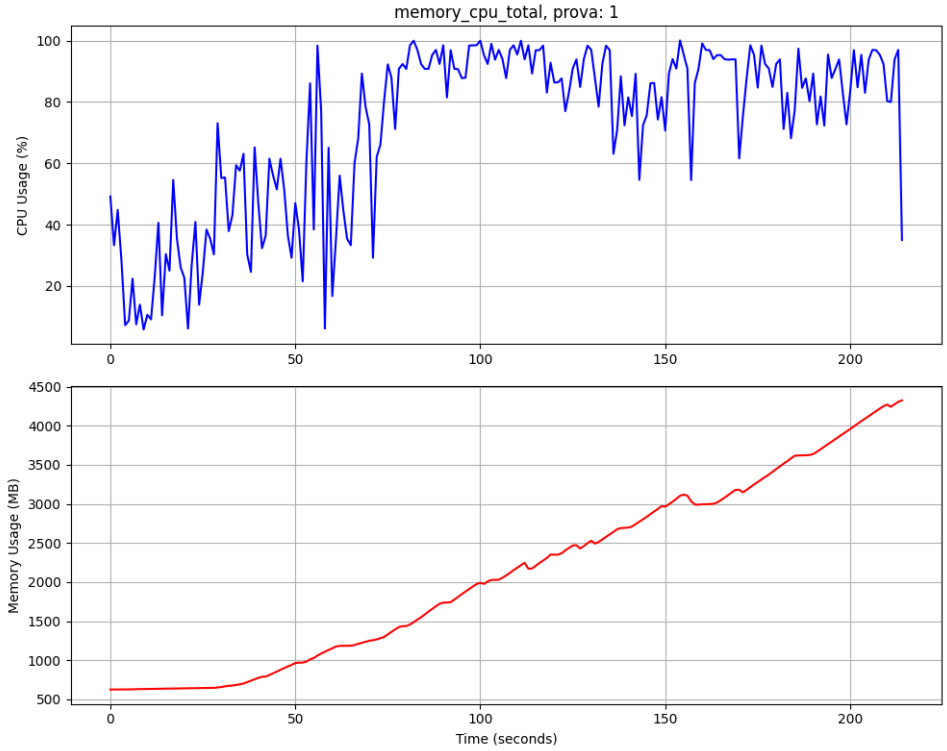


Figura 4.9: Utilizzo di memoria e CPU durante il calcolo dei prefissi con Python

dell'algoritmo.
In particolare, l'approccio Python costruisce i sottografi in modo iterativo e più incrementale rispetto a Neo4j. Questo porta a un costante aumento del carico di lavoro CPU man mano che più nodi e relazioni vengono processati.

Python, infatti, potrebbe eseguire l'algoritmo in modo più sequenziale o con meno operazioni concorrenti rispetto a Neo4j, che potrebbe cercare di ottimizzare l'accesso concorrente ai dati. Questo può ridurre il tempo di elaborazione ma aumentare il consumo di risorse in modo diverso.
Neo4j, d'altro canto, può avere meccanismi di gestione della memoria più avanzati che allocano memoria in grandi blocchi iniziali per poi riutilizzarla, mentre Python tende ad allocare memoria in modo più incrementale.
In definitiva, nonostante Neo4j offra un ambiente robusto con molte funzionalità per la gestione di grafi, l'implementazione in Python con NetworkX può risultare più veloce a causa di una minore complessità di overhead, un algoritmo più diretto e specifico e una gestione della memoria più semplice. Questi fattori combinati possono portare a tempi di calcolo dei sottografi più rapidi in Python.

Capitolo 5

Conclusioni e sviluppi futuri

Dalle analisi svolte, emerge chiaramente l'importanza cruciale delle risorse hardware, in particolare della memoria e della CPU, nella gestione efficiente di grandi volumi di dati. Le configurazioni sperimentate dimostrano come l'allocazione adeguata della memoria possa influenzare significativamente le prestazioni del sistema.
Inoltre, occorre considerare che le configurazioni di memoria scelte per effettuare i vari esperimenti sono molto limitate.
Questo è dovuto principalmente al limite fisico del calcoltore utilizzato per effettuare i test. Di conseguenza, questi limiti hanno influenzato la capacità di sperimentare configurazioni di memoria che si sarebbero potute mostrare più adeguate.

Un'ulteriore opzione potrebbe essere quella di adattare il codice Python per essere eseguito su Spark.
Apache Spark è un motore di elaborazione dati open source che offre capacità di elaborazione dati in-memory ad alta velocità per big data.
Utilizzare Spark potrebbe non solo migliorare le prestazioni del sistema ma anche consentire l'analisi di dataset ancora più grandi e complessi.

In definitiva, risulta comunque evidente l'importanza di monitorare le risorse e di ottimizzarne l'utilizzo al fine di comprendere i colli di bottiglia e agire di conseguenza.

In questo contesto di studio, è emerso che ogni problema di elaborazione dati può avere diverse soluzioni tecnologiche, ciascuna con i propri punti di forza e debolezze. Neo4j è noto per le sue capacità avanzate di gestione di grafi, il che lo rendeva una scelta naturale per il contesto considerato.
In particolare, risulta che Neo4j, pur essendo un potente database a grafo, non si è dimostrato valido quanto Python per l'elaborazione dei dati in questo contesto, specialmente all'aumentare della dimensione del dataset.

Bibliografia

[1] Claudia Diamantini, Laura Genga, Domenico Potena, and Wil van der Aalst. Building instance graphs for highly variable processes. *Expert Systems with Applications*, 59:101–118, 2016.

[2] Boudewijn van Dongen. Bpi challenge 2012. `https://doi.org/10.4121/uuid:3926db30-f712-4394-aebc-75976070e91f`, 2012.

[3] Andrea Chiorrini, Claudia Diamantini, Laura Genga, and Domenico Potena. Multi-perspective enriched instance graphs for next activity prediction through graph neural network. *Journal of Intelligent Information Systems*, pages 1–21, 2023.