UNIVERSITÀ POLITECNICA DELLE MARCHE
FACOLTÀ DI INGEGNERIA
Dipartimento di Ingegneria dell'Informazione
Corso di Laurea Magistrale in Ingegneria Informatica e dell'Automazione

TESI DI LAUREA

Progetto e sviluppo di un sistema per l'estrazione e la gestione di Prefix Instance Graph

Design and development of a system for the extraction and management of Prefix Instance Graph

Relatore

Prof. Domenico Potena

Candidata

Alice Moretti

ANNO ACCADEMICO 2023-2024

Alla mia famiglia,
a chi c'è sempre stato,
a chi ha sempre creduto in me,
fin dal primo giorno, forse anche più di me.

Sommario

Nell'era pionieristica dell'informatica, la limitatezza delle risorse hardware richiedeva un'ottimizzazione estrema di ogni bit e ciclo di clock. Oggi, nonostante l'aumento esponenziale delle capacità hardware, la gestione efficiente delle risorse è cruciale a seguito dell'avvento dell'era dei Big Data. Le aziende raccolgono dati significativi dai processi aziendali, ma l'interpretazione efficace di questi dati è complessa. Il *Process Mining* emerge come disciplina innovativa che combina tecniche di data mining e analisi dei processi aziendali per scoprire, monitorare e migliorare i processi attraverso i log di eventi. In particolare, il *Predictive Process Monitoring* utilizza modelli predittivi per anticipare eventi futuri, migliorando l'allocazione delle risorse e l'efficienza operativa. Questo studio si propone di ottimizzare la gestione e l'estrazione di *Prefix Instance Graph*, i quali possono essere utilizzati per prevedere la prossima attività di un processo in corso. L'obiettivo dell'elaborato è quello di investigare i vantaggi dell'utilizzo di un database a grafo per una gestione e un processamento efficaci dei dati.

Il presente elaborato è articolato in sei capitoli, suddivisi come segue:

- Nel Capitolo 1 viene introdotto il contesto in cui si colloca il lavoro e l'obiettivo dello studio.
- Nel Capitolo 2 viene descritto il background, la terminologia usata e l'algoritmo utilizzato per la generazione di Instance Graph.
- Nel Capitolo 3 vengono introdotti i database a grafo, la loro architettura, i loro vantaggi e svantaggi e lo stato dell'arte. L'attenzione si sofferma su Neo4j, il database scelto per la gestione dei dati.
- Nel Capitolo 4 viene presentato il dataset utilizzato per svolgere gli esperimenti e si utilizza Cypher, il linguaggio messo a disposizione da Neo4j, per definire delle query efficienti per importare i dati e, successivamente, per analizzare la struttura del dataset.
- Nel Capitolo 5 vengono analizzate differenti tecniche per la generazione di Prefix Instance Graph e per il calcolo dei case attivi parallelamente a ciascun prefisso. Le varie soluzioni individuate vengono messe a confronto per valutarne le prestazioni.
- Nel Capitolo 6, infine, vengono tratte delle conclusioni a seguito delle analisi svolte e vengono specificati dei possibili sviluppi futuri.

Indice

Elenco delle figure

Elenco delle tabelle

Listings

CAPITOLO 1

Introduzione

Nell'era pioneristica dell'informatica, le risorse hardware dei calcolatori erano estramente 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 delle loro capacità, ci si trova di fronte ad una sfida simile, ma su una scala differente: 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 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 dei 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 una continua innovazione.

Nell'odierno panorama aziendale, è sempre più comune raccogliere dati relativi all'esecuzione dei processi aziendali. Questi dati si sono dimostrati essere di estrema importanza, poichè consentono di migliorare la gestione dei processi stessi. Tuttavia, la sfida principale risiede nell'interpretazione e nell'utilizzo di questi dati in modo efficace. Attualmente, è possibile derivare i dati dell'esecuzione dei processi da sistemi ad hoc per la gestione dei processi o dai log dei diversi sistemi informatici utilizzati nelle aziende. Tali esecuzioni sono registrate in un log di eventi. Una singola esecuzione può includere l'esecuzione parallela di attività aziendali, ma i log memorizzano solo la traccia di un'istanza di processo, ossia la sequenza dell'attività sulla base dell'ordine temporale di accadimento. È possibile eseguire analisi direttamente sulle tracce sequenziali, ma si ottengono informazioni più approfondite quando si conosce la struttura dell'istanza, ovvero il modello che rappresenta esplicitamente le dipendenze casuali tra le attività, evidenziando sequenze, parallelismi e cicli tra le attività, nascosti nella traccia sequenziale. Gli *Instance Graph* sono delle rappresentazioni a grafo che permettono di comprendere la struttura sottostante delle esecuzioni dei processi.

In questo ambito, il *Process Mining* emerge come disciplina innovativa che combina tecniche di data mining e di analisi dei processi aziendali per ottenere una comprensione dettagliata di come funzionano i processi operativi di un'organizzazione. L'obiettivo principale del Process Mining è quello di estrarre informazioni dai log di eventi generati dai sistemi informativi per scoprire, monitorare e migliorare i processi aziendali. Attraverso l'analisi di questi log, è possibile ricostruire il flusso effettivo delle attività, identificare colli di bottiglia, individuare deviazioni rispetto ai processi previsti e fornire suggerimenti per l'ottimizzazione. Il punto centrale del Process Mining è rappresentato dalla fase di Process Discovery, durante la quale i modelli di processo vengono generati automaticamente dai log di eventi senza utilizzare alcuna informazione a priori, tramite algoritmi come l'algoritmo Alpha. Successivamente, la fase di Conformance Checking verifica la conformità tra i modelli di processo scoperti e quelli prescritti per identificare deviazioni e non conformità. Infine, la fase di Process Enhancement utilizza i risultati delle fasi precedenti per migliorare i processi esistenti, ottimizzando le scelte e fornendo supporto decisionale.

Il *Predictive Process Monitoring* è una branca del Process Mining che ha come obiettivo quello di prevedere il futuro dell'esecuzione di un processo in corso, non ancora completato. Questa disciplina, utilizzando tecniche di machine learning e data mining, analizza i dati storici dei processi per costruire modelli predittivi che possono anticipare eventi futuri, tempi di completamento, possibili colli di bottiglia, e altri aspetti rilevanti dei processi operativi. Questa capacità predittiva è fondamentale perchè consente alle organizzazioni di pianificare in modo efficiente l'allocazione delle risorse. Essere in grado di prevedere in anticipo l'esito dell'esecuzione di un processo, il tempo necessario per completare un'istanza di processo o le attività che verranno eseguite successivamente, permette alle organizzazioni di prevenire esiti indesiderati, problemi e ritardi e di ottimizzare l'uso delle risorse disponibili. Questo non solo riduce il rischio di sovraccarico o sottoutilizzo delle risorse, ma permette anche di massimizzare l'efficienza operativa. A differenza del monitoraggio reattivo dei processi aziendali che identifica eventuali problemi solo dopo che sono accaduti, il Predictive Process Monitoring consente alle aziende di adottare misure preventive tempestive. Questo approccio proattivo non solo migliora la capacità di risposta alle emergenze, ma contribuisce anche a ridurre i tempi di ciclo complessivi dei processi e a migliorare la gestione delle risorse nel lungo periodo. Supportato dai progressi tecnici nel campo della Data Science, della Predictive Analytics e dell'Intelligenza Artificiale data-driven, lo sviluppo di tecniche predittive adattate al campo del Process Mining si è rapidamente affermato.

Il presente elaborato si propone di esplorare ed identificare strategie efficaci ed efficienti per la gestione ottimale della vasta quantità di dati che caratterizza il mondo odierno. Partendo da un dataset esistente, l'obiettivo è quello di ottimizzare una procedura di gestione ed estrazione di *Prefix Instance Graph*. Questi dati sono utili per il training di una rete neurale deep nel contesto del Process Mining per eseguire un task di Next Activity Prediction: dato lo stato attuale di esecuzione di un processo, si vuole prevedere quale attività verrà eseguita successivamente.
Poiché gli Instance Graph sono dei grafi, questo studio si concentrerà sull'utilizzo di database a grafo per gestire e processare in maniera efficace questi dati.

CAPITOLO 2

Metodologia

2.1 Background

I sistemi informativi sono ampiamente utilizzati dalle organizzazioni per supportare l'esecuzione dei processi aziendali. Generalmente, questi sistemi registrano le esecuzioni passate dei processi in un log di eventi. Un *event log* è un insieme di tracce, ognuna delle quali è costituita da eventi, cioè attività eseguite da una certa risorsa in un determinato istante di tempo. In altre parole, un evento è un oggetto con delle proprietà.

Una singola esecuzione di un processo chiamata *istanza di processo* o *case*, può includere l'esecuzione parallela di attività aziendali. Tuttavia, gli event log memorizzano solo la traccia di un'istanza di processo, ovvero la sequenza delle attività ordinate temporalmente. Solitamente, vengono registrati anche l'orario di inizio delle attività e l'identificativo dell'istanza. A seconda del sistema, si possono trovare anche altre informazioni come l'esecutore, i dati di input/output e la durata totale dell'attività.
Sfruttando le tracce sequenziali è possibile eseguire delle semplici analisi che permettono di ottenere informazioni come il tempo medio di esecuzione delle istanze, l'utilizzo delle risorse e l'analisi degli intervalli degli eventi. Tuttavia, è possibile ottenere maggiori informazioni solo quando si conosce il modello dell'istanza, il quale rappresenta esplicitamente i parallelismi tra le attività, che sono invece nascosti nella traccia sequenziale.

Per costruire modelli di istanza, le relazioni causali tra le attività devono essere note, oppure possono essere dedotte dagli event log. Infatti, le *causal relations* sono le relazioni che descrivono come un evento è legato ad un altro all'interno del processo. Il Process Discovery è la disciplina che si occupa di inferire le relazioni da un log e costruire modelli. Dunque, il suo obiettivo è quello di elaborare un event log per ricavare un modello di processo che descriva le relazioni d'ordine esistenti tra le attività del processo.

Un *Instance Graph* (IG) è una rappresentazione a grafo di un'istanza di processo costruita a partire da un insieme di tracce in un event log e dal modello di processo corrispondente. Ogni nodo nell'IG rappresenta un evento e include informazioni sull'attività associata e la posizione dell'evento nella traccia. Gli archi del grafo rappresentano le relazioni causali tra eventi che riflettono il flusso del processo. In altre parole, le relazioni causali sono rappresentate dagli archi diretti che collegano i nodi. Queste relazioni indicano che un evento

deve precedere un altro affinché quest'ultimo possa verificarsi, riflettendo così la dipendenza temporale e logica tra gli eventi. Gli Instance Graph sono delle strutture che permettono di visualizzare parallelismi e variabilità nel processo, che non sono invece evidenti nelle sole tracce sequenziali.
Nella Figura 2.1 sono mostrate tre differenti tracce ed i rispettivi Instance Graph.

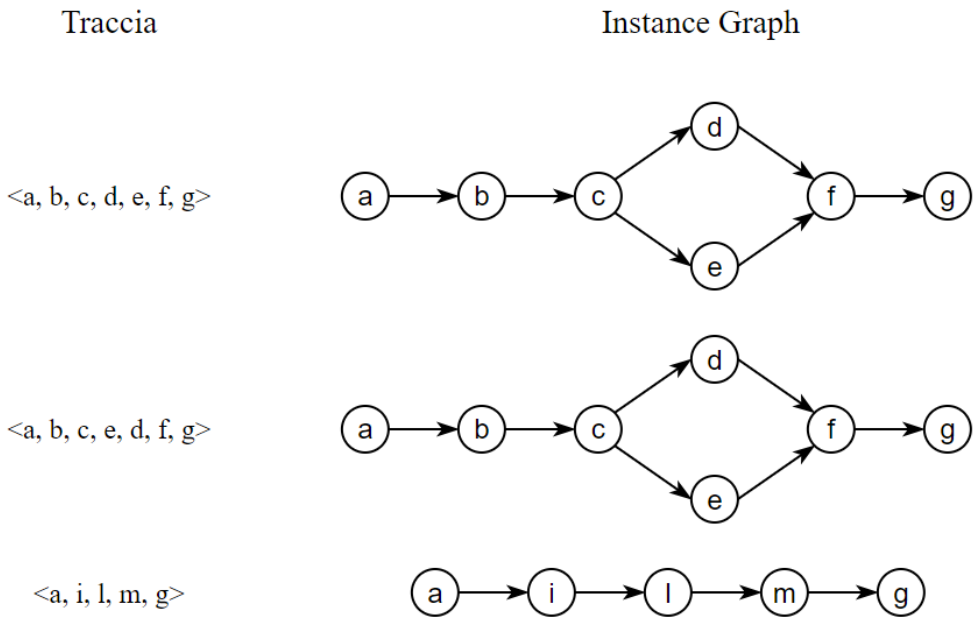


Figura 2.1: Esempio di tracce e corrispondenti Instance Graph

Ogni traccia rappresenta un percorso specifico attraverso un grafo. Tutte le tracce partono dal nodo a e terminano nel nodo g e nonostante le variazioni nell'ordine degli eventi, le prime due tracce corrispondono allo stesso Instance Graph. Pertanto, diverse sequenze possono corrispondere allo stesso IG. In particolare, i nodi c, d, e, f si combinano per formare dei percorsi alternativi che convergono nello stesso stato finale. La terza traccia, invece, rappresenta un percorso completamente distinto, partendo sempre dal nodo a e terminando nel nodo g, ma passando per una sequenza diversa di nodi intermedi. Questa traccia è indipendente dalle prime due, mostrando la possibilità di avere dei segmenti indipendenti che possono essere percorsi senza interferire con gli altri.
Tale esempio mostra chiaramente che tracce differenti possono corrispondere allo stesso Instance Graph, indicando che ci sono molteplici modi per attraversare il grafo pur mantenendo una struttura comune. Inoltre, la terza traccia evidenzia la possibilità di avere un percorso distinto dai precedenti che però ha lo stesso nodo di partenza e lo stesso nodo di arrivo.

A partire dalla definizione di Instance Graph è possibile introdurre la nozione di *Prefix Instance Graph* (prefix-IG). Esso è una versione parziale dell'IG che rappresenta solo un prefisso di un'istanza di processo. In particolare, mentre un IG mostra l'intero flusso di un'istanza di processo fino al suo completamento, un prefix-IG ne mostra solo un parte iniziale, ossia il flusso dell'istanza fino ad un determinato punto. Questa struttura è particolarmente utile quando viene combinata con tecniche di machine learning per fare previsioni sui prossimi eventi di un processo. In particolare, le Graph Neural Network (GNN) possono essere utilizzare per analizzare i prefix-IG e predire le attività future.

La costruzione di prefix-IG a partire dai grafi ottenuti, è realizzata tenendo in considerazione l'ordine degli eventi in ogni traccia. A tale scopo, ogni nodo di un Instance Graph è etichettato con un indice progressivo che fa riferimento alla posizione del corrispondente evento in una traccia. Tale indice serve per indicare l'ordine dei nodi e viene utilizzato per costruire i prefissi relativi al grafo. In particolare, dato un Instance Graph g, il prefisso $p_i(g)$ è ottenuto selezionando i primi i nodi e gli archi tra di essi compresi e può essere etichettato con l'attività dell'evento in posizione $i+1$ per svolgere task di Next Activity Prediction.
Seguendo questa procedura per un grafo g con N nodi, è possibile generare $N-1$ prefissi, quindi, la procedura è ripetuta fino a che l'ultimo nodo dell'Instance Graph è selezionato come label.

Si consideri la rete di Petri riportata in Figura 2.2.

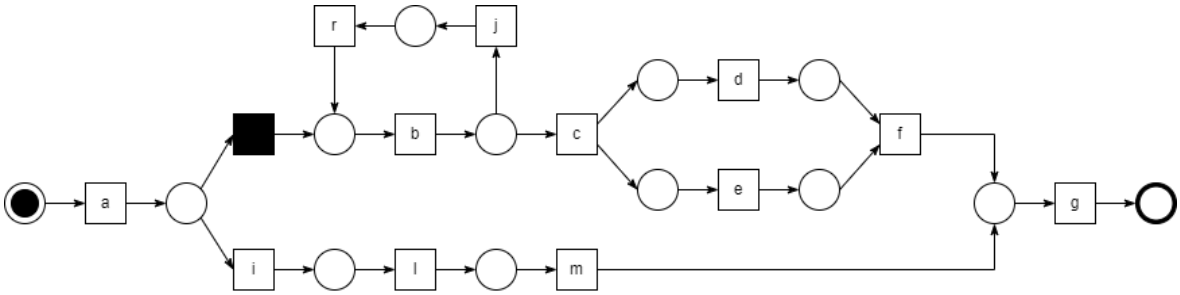


Figura 2.2: Rete di Petri

Un esempio di Instance Graph relativo alla traccia

$$\sigma = <(1,\mathrm{a}),(2,\mathrm{b}),(3,\mathrm{c}),(4,\mathrm{d}),(5,\mathrm{e}),(6,\mathrm{f}),(7,\mathrm{g})>$$

e alla rete di Petri di Figura 2.2 è riportato nella Figura 2.3.

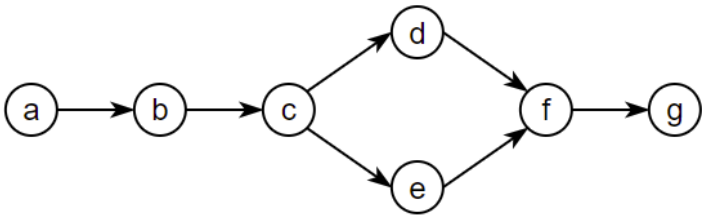


Figura 2.3: Esempio di Instance Graph

In riferimento all'Instance Graph in Figura 2.3, i prefissi di lunghezza 4 e 5 sono riportati in Tabella 2.1.

Prefissi	Nodi	Archi	Label
$p_4(g)$	{(1, a),(2, b),(3, c),(4, d)}	{((1, a),(2, b)), ((2, b),(3, c)), ((3, c),(4, d))}	e
$p_5(g)$	{(1, a),(2, b),(3, c),(4, d),(5, e)}	{((1, a),(2, b)), ((2, b),(3, c)), ((3, c),(4, d), ((3, c),(5, e))}	f

Tabella 2.1: Prefissi di lunghezza 4 e 5 relativi all'IG di Figura 2.3

2.2 Building Instance Graph

L'algoritmo Building Instance Graph (BIG) descritto nell'articolo Diamantini *et al.* [2016] definisce una metodologia per la definizione degli Instance Graph.
Tale algoritmo ha lo scopo di costruire un grafo diretto per ogni traccia dell'event log e si sviluppa in due passi:

1. costruzione di un grafo per ogni traccia tenendo conto delle causal relations estratte a partire dalla rete di Petri che modella il processo;

2. riparazione delle tracce non conformi al modello tramite l'attività di Conformance Checking in modo da evitare overfitting.

In presenza di eventi non conformi, vengono generati degli IG anomali e di bassa qualità. Pertanto, è importante notare come la procedura di riparazione delle tracce anomale sia essenziale, altrimenti, la presenza di eventi non conformi al modello in una traccia porterebbe a grafi sconessi o con un alto livello di parallelismo, confondendo le relazioni temporali tra le varie attività. La procedura di riparazione degli IG corrispondenti a tracce anomale, trasforma gli IG in grafi in grado di rappresentare anche le tracce anomale senza generalizzare eccessivamente. In primo luogo, le tracce anomale e, di conseguenza gli IG, vengono riconosciuti nell'event log mediante una tecnica di Conformance Checking (Adriansyah *et al.* [2011]). Dopodiché, l'algoritmo ripara prima le attività cancellate (*Deletion Repairing Algorithm*) e poi le attività inserite (*Insertion Repairing Algorithm*). In particolare, per gli eventi cancellati, la riparazione consiste nell'identificare i nodi che avrebbero dovuto essere collegati all'attività cancellata, collegandoli correttamente. Per la riparazione dell'inserimento, si modificano gli archi che collegano i nodi corrispondenti agli eventi precedenti e successivi dell'evento inserito. Per collegare tali nodi con il nodo corrispondente all'evento inserito nel grafo, si tiene conto delle relazioni causali tra i suoi predecessori e successori nella traccia.

A partire dall'output ottenuto tramite BIG, è possibile estrapolare delle features che tengano in considerazione ulteriori prospettive temporali, come specificato nell'articolo Chiorrini *et al.* [2023]. Le tre features che vengono considerate sono di due tipologie:

- dirette: corrispondenti ad attributi già memorizzati nell'event log;

- indirette: derivate a partire dalle informazioni disponibili nell'event log.

In particolare, occorre notare come il calcolo delle features indirette venga eseguito dopo la costruzione degli Instance Graph e non direttamente a partire dall'event log. Questo assicura che gli intervalli temporali relativi ad ogni attività siano calcolati rispetto all'attività precedente nel grafo e non rispetto all'attività precedente nella sequenza.

Questo algoritmo permette di generare un grafo per ciascuna traccia, rappresentando la sequenza degli eventi svolti in ogni esecuzione. A partire dagli Instance Graph ottenuti con questa procedura, è interessante generare tutti i possibili prefissi di ciascun IG.

CAPITOLO 3

Database a grafo

Gli Instance Graph sono delle rappresentazioni di dati strutturate come grafi dove i nodi rappresentano gli eventi e gli archi rappresentano le relazioni tra di essi, quindi, si è evidenziata la necessità di un sistema di gestione dei dati che possa sfruttare al meglio questa struttura. A tal proposito, si è deciso di utilizzare i database a grafo in modo da poter sfruttare la loro capacità di rappresentare e navigare le relazioni tra nodi in modo naturale ed efficiente.

Un database a grafo è un tipo di databse NoSQL che utilizza strutture a grafo per rappresentare e memorizzare le informazioni, differenziandosi dai database relazionali.
I database relazionali, introdotti negli anni '70, si basano sul modello relazionale e utilizzano tabelle per rappresentare i dati. Ogni tabella è costituita da righe e colonne. Le righe rappresentano le istanze dei dati, mentre le colonne rappresentano gli attributi di questi dati. La struttura relazionale permette di definire chiavi primarie e chiavi esterne per mantenere l'integrità referenziale tra le tabelle, facilitando la normalizzazione dei dati. Il linguaggio standard per interagire con i database relazionali è SQL (Structured Query Language). Le operazioni di CRUD (Create, Read, Update, Delete) sono ben supportate e le query complesse possono essere eseguite attraverso join tra tabelle multiple.
I database a grafo, invece, sono progettati per gestire dati altamente interconnessi. Utilizzano nodi, archi e proprietà per rappresentare e memorizzare le informazioni. I nodi rappresentano le entità, gli archi rappresentano le relazioni tra le entità e le proprietà memorizzano i dati associati a nodi e archi. Questo modello è particolarmente potente per rappresentare strutture di dati complesse come reti sociali, grafi di conoscenza, raccomandazioni e altre applicazioni in cui le relazioni tra i dati sono fondamentali.
A differenza dei database relazionali, i database a grafo eccellono nelle operazioni di traversing, cioè nell'esplorazione delle relazioni tra i nodi. Questo rende le query sui grafi molto efficienti per analizzare connessioni e pattern complessi che sarebbero difficili da modellare e interrogare con SQL.

3.1 Architettura dei database a grafo

L'architettura di un database a grafo è basata su tre elementi fondamentali:

- **Nodi**: rappresentano le entità o i vertici del grafo. Ogni nodo può contenere un insieme di proprietà che memorizzano informazioni specifiche sull'entità.

- **Archi**: rappresentano le connessioni tra i nodi. Ogni arco connette due nodi e definisce il tipo di relazione tra di essi. Gli archi possono avere una direzione (da un nodo sorgente ad un nodo di destinazione) oppure possono essere bidirezionali e possono avere delle proprietà.
- **Proprietà**: sono informazioni aggiuntive che possono essere associate sia ai nodi che agli archi. Le proprietà arricchiscono i nodi e gli archi con dettagli specifici, rendendo il grafo più informativo e utile per le query.

Il modello di dati dei database a grafo si distingue per la sua flessibilità e semplicità nel rappresentare relazioni complesse. A differenza del modello relazionale, dove le relazioni tra entità richiedono molti join, operazioni spesso costose, nei database a grafo è sufficiente seguire direttamente le relazioni attraverso gli archi del grafo.

L'avvento del Web 2.0 ha favorito lo sviluppo dei database a grafo grazie alla crescente complessità delle interazioni online e alla necessità di gestire dati con molteplici connessioni. Questo modello di dati si adatta perfettamente a situazioni in cui le relazioni sono altrettanto importanti quanto le entità stesse, come nel caso delle reti sociali, delle reti di trasporto e dei sistemi di raccomandazione.
I database a grafo offrono numerosi vantaggi, soprattutto in termini di performance, nelle query complesse. La gestione delle relazioni in questi database è estremamente efficiente: operazioni che nei database relazionali richiederebbero numerosi join possono essere eseguite in modo molto più rapido e intuitivo nei database a grafo. Questo è particolarmente vero per le query che richiedono di seguire catene di relazioni, come la ricerca di tutti i nodi connessi a un nodo specifico. Un esempio pratico dell'utilità dei database a grafo riguarda l'analisi delle reti sociali. Invece di rappresentare ogni utente come un record separato con collegamenti a tabelle diverse, un database a grafo può rappresentare ogni utente come un nodo e ogni connessione tra gli utenti come un arco tra i rispettivi nodi. Ciò consente di identificare rapidamente strutture complesse come comunità di utenti, influenze sociali e percorsi di informazioni virali. In maniera simile, nei sistemi di e-commerce, i database a grafo aiutano a tracciare le relazioni tra prodotti, utenti e acquisti, migliorando le capacità di raccomandazione basate sulle relazioni tra gli elementi. Anche nella logistica e nella gestione delle supply chain, i grafi sono utili in quanto consentono di rappresentare flussi di prodotti e materiali tra diverse entità, permettendo di ottimizzare percorsi e identificare colli di bottiglia.
Un altro vantaggio significativo dei database a grafo è la flessibilità del loro modello di dati. La loro architettura consente una rappresentazione più naturale delle entità e delle loro relazioni, rendendoli particolarmente utili per applicazioni come i social network, i sistemi di raccomandazione e la gestione delle reti. Inoltre, i database a grafo sono altamente adattabili e possono facilmente rispondere a cambiamenti nei requisiti dei dati e delle relazioni senza necessità di ristrutturazioni complesse. La loro capacità di gestire facilmente dati semistrutturati e variabili, che possono essere problematici da modellare in modelli di database più rigidi, è un punto fondamentale. Un nodo può avere attributi che variano in base al contesto o che sono specifici solo a determinate relazioni, consentendo una rappresentazione più accurata e dettagliata della realtà.
La scalabilità è un altro aspetto in cui i database a grafo eccellono. Molti di questi sistemi sono progettati per funzionare in ambienti distribuiti, permettendo di scalare orizzontalmente e gestire grandi volumi di dati. Inoltre, le operazioni sui grafi possono essere parallelizzate, migliorando ulteriormente le performance in sistemi distribuiti. I database a grafo sono spesso più efficienti nel trovare percorsi e connessioni tra dati rispetto ai database relazionali tradizionali. Gli algoritmi specializzati come la ricerca dei cammini più brevi possono essere eseguiti in modo più diretto e efficiente su strutture dati a grafo, rendendoli ideali per applicazioni che richiedono query complesse come "Trova la connessione più breve tra due entità".

Infine, i database a grafo offrono potenti capacità di scoprire pattern e relazioni nascoste. Sono particolarmente adatti per l'analisi di grafi, la rilevazione di comunità e l'identificazione di anomalie nelle relazioni, attività che risultano più complesse nei database tradizionali.

Tuttavia, l'adozione dei database a grafo presenta anche alcuni svantaggi. La progettazione del modello a grafo richiede una comprensione approfondita delle relazioni tra le entità e delle query, al fine di ottimizzare le prestazioni e garantire la scalabilità del sistema. Inoltre, l'integrazione con strumenti e piattaforme esistenti può essere più complessa rispetto ai sistemi più consolidati come i database relazionali. Gli sviluppatori abituati ai database relazionali possono trovare inizialmente difficile adattarsi ai nuovi paradigmi di modellazione dei dati richiesti dai database a grafo. Anche i linguaggi di query specifici, come Cypher o Gremlin, possono richiedere un apprendimento approfondito per essere utilizzati efficacemente.
Inoltre, esistono molteplici implementazioni di database a grafo, ciascuna con le proprie caratteristiche, linguaggi di query e metodologie. Questa varietà può rendere difficile la standardizzazione e la portabilità delle applicazioni tra diverse piattaforme. L'interoperabilità tra diversi database a grafo e con altri tipi di database può essere limitata, complicando l'integrazione con sistemi esistenti.

In sintesi, i database a grafo offrono vantaggi significativi nella gestione e nell'analisi di dati fortemente connessi, rendendoli ideali per applicazioni che richiedono una navigazione efficiente delle relazioni complesse. La scelta di adottare una database a grafo piuttosto che un database tradizionale dipende fortemente dal contesto specifico dell'applicazione. Ad esempio, per applicazioni di tipo transazionale dove la qualità dei dati è essenziale, le relazioni tra le entità sono ben definite, l'accesso tramite join è ragionevole e le proprietà ACID (Atomicity, Consistency, Isolation, Durability) devono essere sempre garantite, i database relazionali rappresentano la scelta più adatta. D'altro canto, in scenari caratterizzati da dati fortemente connessi e relazioni complesse, come le reti sociali, i database a grafo risultano essere una scelta preferibile. Essi consentono una rappresentazione naturale delle entità e delle loro relazioni, permettendo query più efficienti e una navigazione più agevole del grafo. Pertanto, la decisione verso l'una o l'altra soluzione deve essere guidata dalle specifiche esigenze di prestazioni, scalabilità e dalla natura delle relazioni dei dati.

Alcuni esempi di database a grafo sono i seguenti:

- **Neo4j**: è uno dei database a grafo più diffusi e popolari. È un database a grafo nativo che permette di memorizzare e gestire dati attraverso nodi e relazioni. Neo4j utilizza il linguaggio di query Cypher, che è stato progettato specificamente per le query su grafi. È particolarmente apprezzato per la sua facilità d'uso, la capacità di scalare e l'efficienza nelle operazioni di navigazione del grafo. Viene utilizzato in vari settori, come l'analisi delle reti sociali, la gestione delle identità, le raccomandazioni personalizzate e la rilevazione delle frodi.

- **Amazon Neptune**: è un servizio di database a grafo completamente gestito, offerto da Amazon Web Services (AWS). Supporta sia il modello di grafo a proprietà (Property Graph) che il modello RDF (Resource Description Framework), offrendo così flessibilità nella gestione di dati semantici e di rete. Neptune è progettato per applicazioni che richiedono la navigazione efficiente di dati connessi, come i motori di raccomandazione, la gestione delle conoscenze e l'analisi delle relazioni. La sua integrazione con altri servizi AWS lo rende una scelta popolare per le aziende che utilizzano l'infrastruttura cloud di Amazon.

- **OrientDB**: è un database multimodello che supporta grafi, documenti, modelli key-value e modelli orientati agli oggetti. Questo lo rende molto flessibile e adatto a vari tipi di applicazioni. OrientDB può gestire grandi quantità di dati con elevata velocità e offre funzionalità avanzate come la sicurezza a livello di campo, la gestione delle transazioni ACID e l'indicizzazione avanzata. È spesso utilizzato per applicazioni di social networking, gestione dei contenuti e analisi di grafi.

- **ArangoDB**: è un database multimodello che supporta grafi, documenti e modelli key-value. Utilizza il linguaggio di query AQL (ArangoDB Query Language) per permettere agli sviluppatori di eseguire query complesse su vari modelli di dati. ArangoDB è noto per la sua scalabilità e performance, ed è utilizzato in applicazioni come il machine learning, l'analisi di grafi e i motori di raccomandazione.

- **JanusGraph**: è un database a grafo distribuito e scalabile, derivato dal progetto Titan. È progettato per la gestione di grafi di grandi dimensioni che devono essere distribuiti su cluster di macchine. JanusGraph si integra con vari backend di memorizzazione (come Apache Cassandra, HBase e BerkeleyDB) e motori di ricerca (come Elasticsearch e Solr). È utilizzato in scenari che richiedono la gestione di grafi molto grandi, come le reti di telecomunicazioni, i sistemi di raccomandazione e l'analisi dei dati di sensori.

- **DGraph**: è un database a grafo distribuito e nativo, progettato per la scalabilità e le performance elevate. Utilizza un linguaggio di query chiamato DQL (DGraph Query Language) che rende facile la definizione e l'esecuzione di query complesse. DGraph è ottimizzato per la gestione di grafi molto grandi e viene utilizzato in applicazioni come l'analisi delle reti sociali, il rilevamento delle frodi e la gestione delle identità.

3.2 Neo4j

Nel presente elaborato si è deciso di utilizzare Neo4j come database a grafo per la gestione e l'analisi dei dati. Neo4j è uno dei database a grafo open source sviluppato dalla Neo4j, Inc.
La sua architettura è stata progettata per ottimizzare le operazioni su grafi, rendendo particolarmente efficiente la gestione e la navigazione delle relazioni tra i dati. Neo4j memorizza i dati come nodi, relazioni e proprietà, consentendo di rappresentare in modo naturale e intuitivo le strutture complesse tipiche dei grafi.
Neo4j è un database a grafo nativo, il che significa che è stato costruito da zero per gestire grafi. Utilizza una struttura adiacente per memorizzare i dati, dove ogni nodo mantiene direttamente i puntatori ai nodi connessi. Questo approccio garantisce un'accesso rapido e efficiente alle relazioni, una caratteristica fondamentale per molte applicazioni di grafi.
La piattaforma supporta il linguaggio di query Cypher, progettato per essere intuitivo e facile da apprendere. 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.
Neo4j è anche altamente scalabile. Supporta la distribuzione dei dati su cluster di macchine, consentendo di gestire grandi volumi di dati senza sacrificare le performance. La scalabilità orizzontale è un aspetto cruciale per molte applicazioni moderne che devono gestire enormi quantità di dati e richieste di accesso simultanee.

3.2.1 Ecosistema Neo4j

Nella Figura 3.1 è mostrata una panoramica dell'ecosistema Neo4j, evidenziando i diversi componenti e strumenti che lavorano insieme per offrire una piattaforma completa per la gestione dei dati.

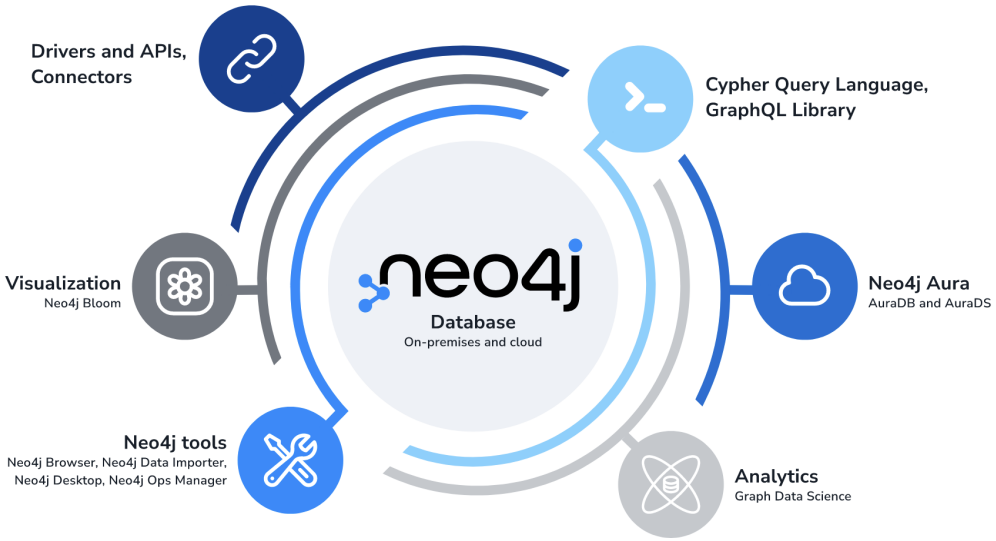


Figura 3.1: Ecosistema di Neo4j

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.

3.2.2 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.

CAPITOLO 4

Creazione del database a grafo

4.1 Dataset

Si è deciso di lavorare con il dataset BPI12 (van Dongen [2012]), 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. Le caratteristiche generali del dataset sono riportate nella tabella 4.1.

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

Tabella 4.1: Caratteristiche del dataset BPI12

Ogni evento rappresenta un'attività eseguita durante il processo di gestione del mutuo ed è accompagnato da informazioni aggiuntive che forniscono dei dettagli sulla particolare attività eseguita. Un esempio di traccia presente all'interno di questo event log è riportato nel Listing 4.1. Va notato che per semplicità è stato riportato un solo evento nella traccia ma generalmente una traccia è composta da più eventi.

```
<trace>
    <date key="REG_DATE" value="2011-10-01T00:38:44.546+02:00"/>
    <string key="concept:name" value="173688"/>
    <string key="AMOUNT_REQ" value="20000"/>
    <event>
        <string key="org:resource" value="112"/>
        <string key="lifecycle:transition" value="COMPLETE"/>
        <string key="concept:name" value="A_SUBMITTED"/>
        <date key="time:timestamp" value="2011-10-01T00:38:44.546+02:00"/>
    </event>
</trace>
```

Listing 4.1: Esempio event log

A partire da questo log è stato estratto un dataset a grafo tramite l'algoritmo BIG (Diamantini *et al.* [2016]). In particolare, questo algoritmo ha permesso di generare un grafo per ciascuna traccia, rappresentando la sequenza degli eventi svolti in ogni esecuzione.

4.2 Pre-processing

Il processo di estrazione degli eventi, realizzato tramite BIG, consente di ottenere informazioni dettagliate su ciascun evento. In particolare, a partire dai dati presenti nell'event log, per ogni evento è possibile ricavare informazioni quali l'identificativo e il nome dell'attività eseguita, l'identificativo della traccia, il timestamp e la risorsa associata.

Tipicamente, ogni evento ha associati due timestamp che indicano l'inizio e la fine dell'attività. Tuttavia, nella maggior parte dei casi, si dispone di un solo timestamp che rappresenta l'inizio o la fine dell'attività. Partendo da questo unico timestamp, è possibile ricavare l'altro, assumendo che la fine di un'attività coincida con l'inizio della successiva.
Riuscire a determinare entrambi i timestamp è fondamentale quando si analizzano degli eventi e si vogliono identificare i parallelismi tra i processi in esecuzione. Questo significa che è possibile che un'attività sia in esecuzione mentre un'altra è già in corso e ciò implica che i loro tempi di esecuzione si sovrappongano. Di conseguenza, è necessario disporre di un tempo di inizio e di fine per ciascun evento al fine di poter gestire tali intersezioni.
Il dataset utilizzato per svolgere gli esperimenti presenta un solo timestamp, che si è supposto essere il tempo di fine dell'attività. Facendo questa scelta, è possibile definire anche il tempo di fine dell'ultima attività, cosa che non sarebbe possibile se il timestamp fosse interpretato come tempo di inizio dell'attività.
Il tempo di inizio viene calcolato supponendo che nel momento in cui finisce un'attività inizia la successiva. Pertanto, il tempo di inizio dell'attività $i+1$ corrisponde al tempo di fine dell'attività i per tutte le attività della traccia, ad eccezione della prima. In tal caso, non essendoci attività precedenti, si è supposto che il tempo di inizio coincida con il tempo di fine. Questa ipotesi è ragionevole e non altera le informazioni della traccia, considerando che ad ogni traccia sono stati aggiunti due nodi fittizi, $START$ all'inizio e END alla fine.
Tuttavia, quando nelle tracce sono presenti delle attività parallele, attuando questa procedura si verificano delle situazioni particolari. Infatti, due attività parallele nell'event log sono caratterizzate dallo stesso timestamp. Si consideri la traccia riportata in tabella 4.2.

Attività	Tempo di fine	Tempo di inizio
a	t_1	t_1
b	t_2	t_1
c	t_3	t_2
d	t_4	t_3
e	t_4	t_4
f	t_5	t_4

Tabella 4.2: Gestione tempi attività parallele

In questo esempio, l'attività d viene eseguita in maniera parallela all'attività e, infatti, entrambe hanno lo stesso tempo di fine. Seguendo le assunzioni precedenti, il tempo di inizio dell'attività e corrisponde al tempo di fine dell'attività d e questo vuol dire che la seconda delle attività parallele risulta essere sempre un'attività istantanea. Per risolvere questa situazione, i tempi delle attività parallele sono stati modificati nel seguente modo:

- il tempo di inizio delle attività parallele viene impostato al valore del tempo di fine del nodo precedente a cui le attività sono collegate;

- il tempo di fine delle attività parallele viene modificato individuando tutti gli archi uscenti dal nodo e assegnando il minimo tra tutti i tempi di inizio dei nodi collegati tramite tali archi.

4.3 Import dei nodi

Al fine 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.
L'output di BIG è costituito da un insieme di Instance Graph, uno per ogni traccia presente nell'event log. Pertanto, esso comprende sia i nodi che gli archi di ciascun grafo. Nonostante ciò, si è deciso di importare solo i nodi nel database e poi definire una procedura per la creazione degli archi. Il file CSV ottenuto dopo l'applicazione dell'algoritmo BIG e utilizzato per l'import dei nodi 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.
Nel Listing 4.2 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:///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 4.2: 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.

Va notato che *activity_id* e *track_id* presi singolarmente non definiscono un identificativo del nodo, ma è la coppia (*activity_id*, *track_id*) che definisce un identificativo univoco del nodo.

4.3.1 Esperimenti

Nella fase iniziale, la query di import dei nodi è stata eseguita con un dataset di dimensioni ridotte. Questo approccio aveva l'obiettivo di valutare la correttezza e mirava ad ottenere risultati in tempi accettabili. È stato quindi possibile verificare che le operazioni funzionassero correttamente. Una volta stabilita la correttezza della query lavorando con un dataset ridotto, l'attenzione si è spostata verso la gestione di dataset reali, di grandi dimensioni, come BPI12. Innanzitutto, sono state condotte delle prove per valutare come il sistema scala quando il numero di grafi aumenta. 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.
Per eseguire 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 4.3.

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 4.3: Tempi di caricamento dei file

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.
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.

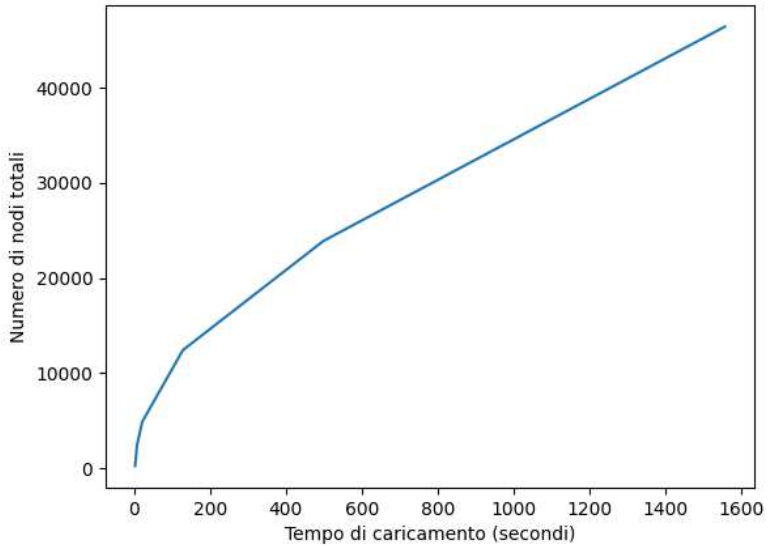


Figura 4.1: Tempo di caricamento al variare del numero di nodi

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 di grafi 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.

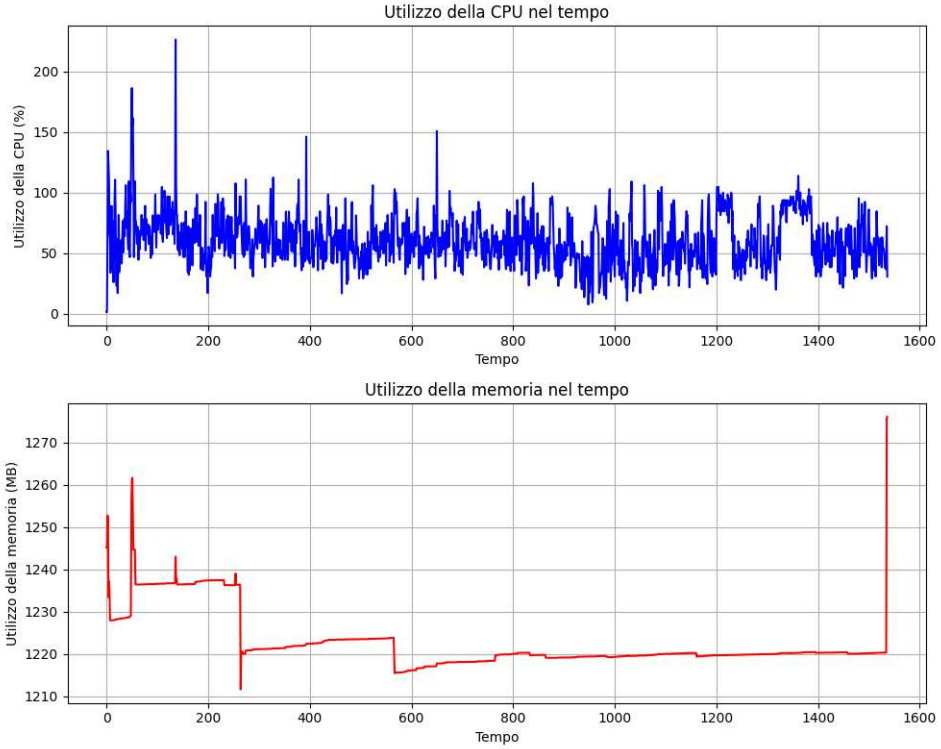


Figura 4.2: Utilizzo di memoria e CPU durante il caricamento di 2000 grafi

Durante l'esecuzione di tutti gli esperimenti finora citati, si è notato che la memoria massima utilizzata è rimasta sempre più o meno stabile intorno ad un 1GB. Questo risultato ha sollevato degli interrogativi sulla configurazione della memoria in 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 nella prova in Figura 4.2 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'ulteriore prova. Infatti, queste impostazioni 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 miglioramento significativo delle prestazioni del sistema. In particolare, il tempo di caricamento del file contenente 2000 grafi si è ridotto da 1555.83 secondi a 795.19 secondi.
Nella Figura 4.3 è riportato l'utilizzo della memoria e della CPU di questa prova.

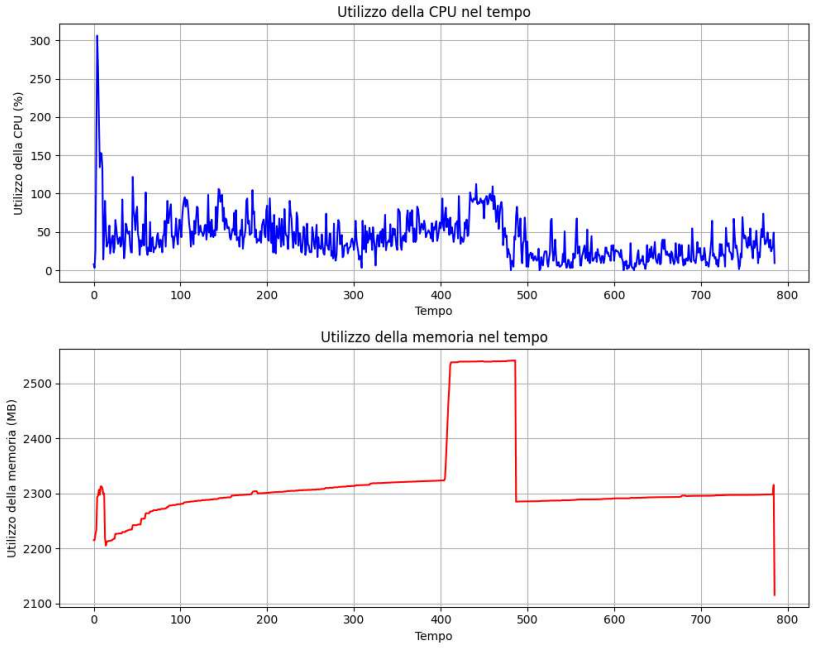


Figura 4.3: Utilizzo di memoria e CPU durante il caricamento di 2000 grafi dopo aver cambiato le impostazioni di memoria

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. 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 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 4.4 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 4.4: 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.
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 operazione 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 pressoché sempre lo stesso, invece dai tempi riportati in Tabella 4.4 si può notare che ogni volta che viene inserito un ulteriore file, il tempo di caricamento aumenta notevolmente rispetto al tempo del file precedente.

4.3.2 Ottimizzazione

Nonostante l'utilizzo di file di dimensioni ridotte, i tempi di caricamento già con 2000 file sono molto elevati. Per questo motivo, si è cercato di definire una query che fosse più efficiente in maniera tale da poter lavorare anche con dataset di dimensioni reali.
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 selezionare 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 specifica 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. Il Listing 4.3 riporta la query di import dei nodi che fa utilizzo di questa procedura.

```
CALL apoc.periodic.iterate(
  "LOAD CSV FROM 'file:///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 4.3: 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 la procedura *apoc.periodic.iterate*, i dati vengono elaborati a blocchi, o "batch", di dimensioni definite, in questo caso pari a 1000. Questo consente di ridurre l'impatto sulla memoria e permette 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.

Visto i notevoli miglioramenti otttenuti dopo aver cambiato le impostazioni predefinite di Neo4j per la memoria, si è deciso di svolgere tutte le successive prove con tali configurazioni.

Si è testata l'efficacia della query ottimizzata per l'import dei nodi. 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 delle interrogazioni rispettivamente ottimizzata e non ottimizzata.

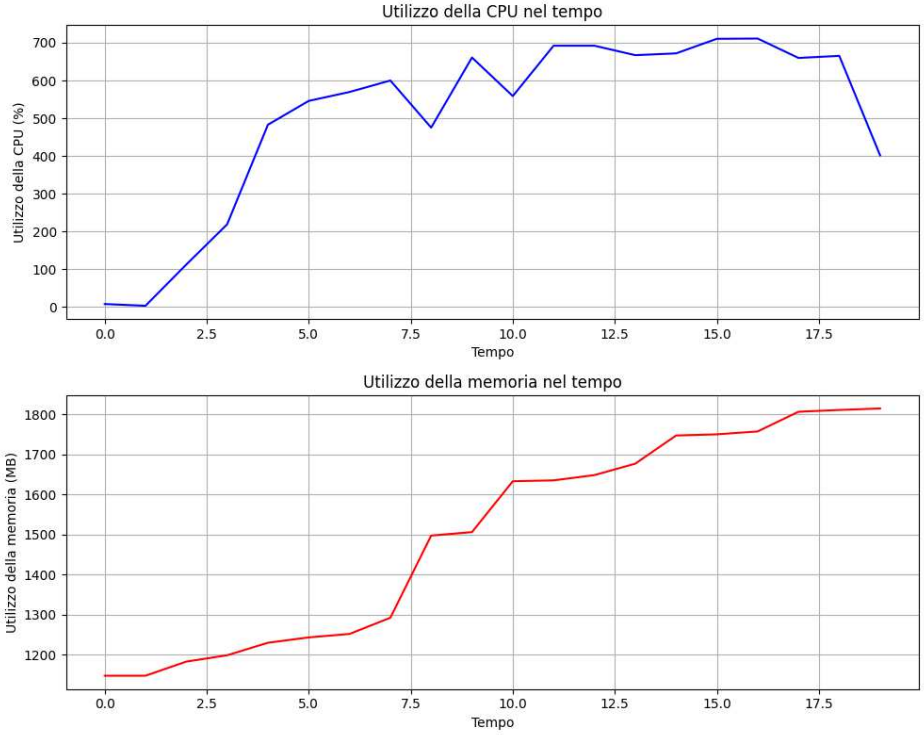


Figura 4.4: Utilizzo di memoria e CPU durante il caricamento di tutti i 13087 grafi tramite la query ottimizzata

Si noti come il grafico di Figura 4.2, inizialmente mostri 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. 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.

4.4 Creazione degli archi

Dopo aver importato i nodi è stato necessario definire gli archi. Nel Listing 4.4 è riportata la query utilizzata per la definizione degli 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 4.4: 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 tempo 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, ossia attività tali per cui il timestamp di inizio coincide con il 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 sarà 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 mezzo e che rispettino il vincolo di sequenzialità non verranno collegati tra di loro.
In Figura 4.5 è riportato un esempio di grafo generato dopo aver eseguito la query di import dei nodi e la query di creazione degli archi.

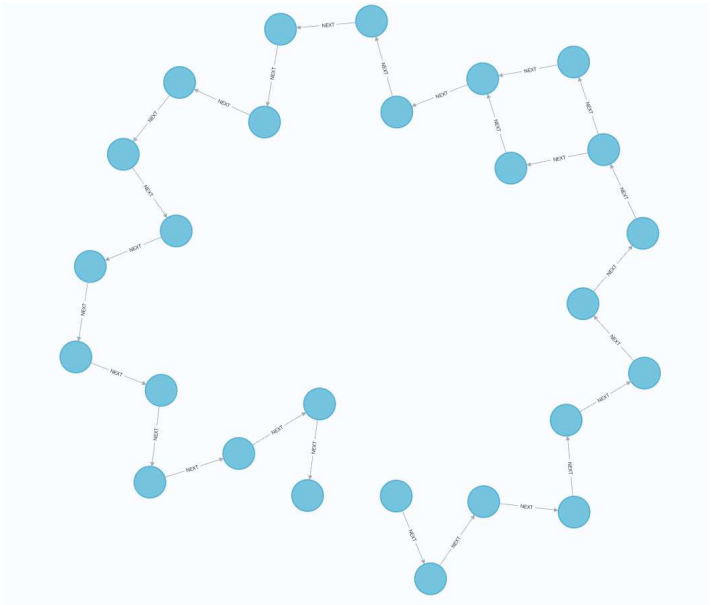


Figura 4.5: Esempio di creazione di nodi e archi in Neo4j

4.4.1 Esperimenti e ottimizzazione

Utilizzando la query definita nel Listing 4.4 dopo aver caricato il dataset completo, i tempi di creazione degli archi sono risultati non calcolabili in quanto troppo prolungati. Pertanto, anche nel caso di creazione degli archi, è stata modificata la query iniziale, sfruttando la procedura *apoc.periodic.iterate* come nel caso dell'import dei nodi. La query definita è riportata nel Listing 4.5.

```
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 4.5: Query ottimizzata per la definizione degli archi

Tuttavia, anche dopo aver definito questa query, non si è riusciti ad ottenere dei risultati in un tempo calcolabile. Per ovviare a questa situazione, è stato necessario definire degli indici sulle proprietà specificate nella clausola *where* della query, come riportato nel Listing 4.6.

```
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 4.6: Creazione degli indici

Gli indici così creati sono degli indici secondari. Questi indici consentono di accelerare la ricerca dei nodi specifici basati su determinate proprietà, in questo caso, *activity_id*, *start_time*, *end_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 delle 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, sebbene gli indici secondari possano 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 scrittura che vengono effettuate sono le due operazioni iniziali che consentono di creare il grafo con la struttura desiderata.
Dopo aver definito la query utilizzando la procedura *apoc.periodic.iterate* e dopo aver definito gli indici, i 280141 archi sono stati creati in 448.14 secondi, ovvero 7.5 minuti.

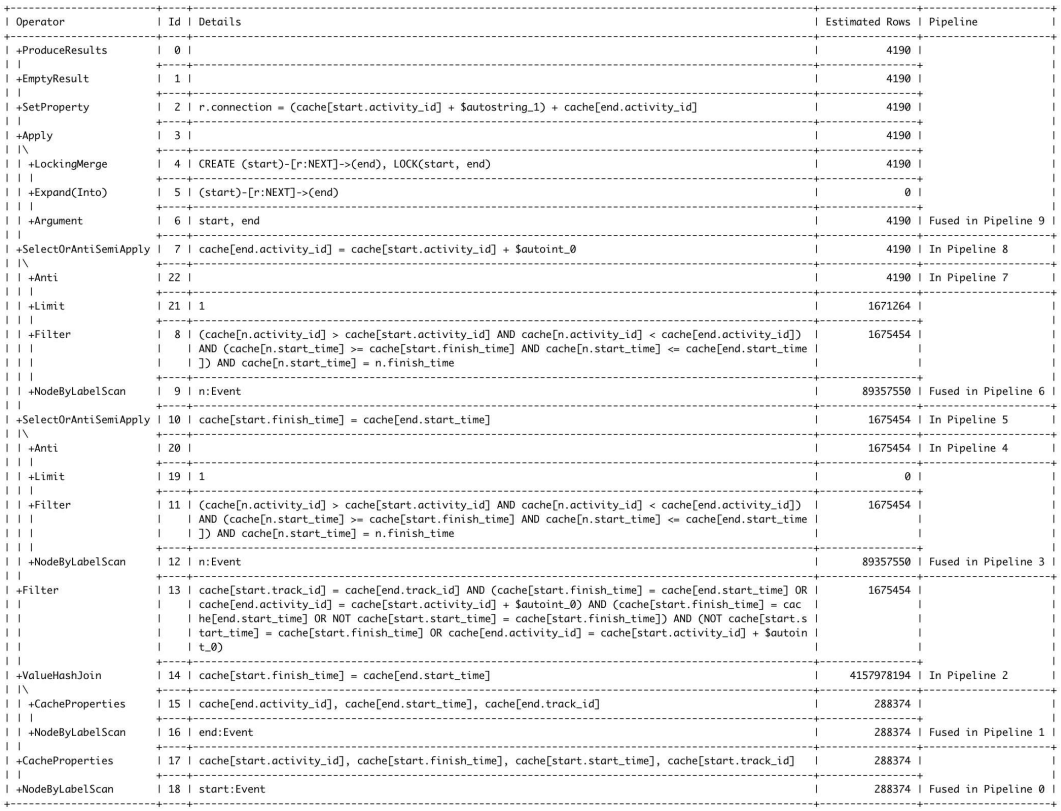
Per analizzare le prestazioni di queste query è stato utilizzato il comando $EXPLAIN$ di Neo4j che, se posto all'inizio di una query, permette di fornire informazioni dettagliate su come il database esegue una query, inclusi i nodi e le relazioni esplorati durante l'esecuzione, i tempi di esecuzione delle varie operazioni e altre statistiche.
Tali informazioni sono riportate in Figura 4.6, relativamente all'esecuzione della query dopo aver definito gli indici e in Figura 4.7, relativamente all'esecuzione della query senza aver definito gli indici.

Per ottenere delle informazioni di profiling dettagliate è stato necessario eseguire le query con il comando $EXPLAIN$ senza *apoc.periodic.iterate* che maschera l'esecuzione della funzione di iterazione interna. Questo non vuol dire che la query verrà eseguita senza l'utilizzo di tale procedura, ma semplicemente che si è voluto mostrare la differenza tra l'esecuzione della medesima query con e senza aver definito degli indici.
Osservando le Figure 4.6 e 4.7, è chiaro 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. In particolare, è possibile notare questo fatto 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 (Figura 4.7) 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.

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 ND cache[n.start_time] = n.finish_time	0	
| +NodeIndexSeekByRange	9	RANGE INDEX n:Event(activity_id) WHERE activity_id > cache[start.activity_id] AND activity_id < cach e[end.activity_id]	0	Fused in Pipeline 5
+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 ND cache[n.start_time] = n.finish_time	0	
| +NodeIndexSeekByRange	12	RANGE INDEX n:Event(activity_id) WHERE activity_id > cache[start.activity_id] AND activity_id < cach e[end.activity_id]	0	Fused in Pipeline 2
+Filter	13	start.track_id = cache[end.track_id] AND (cache[start.finish_time] = cache[end.start_time] OR cache[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)	0	
+Apply	14		349492	
| +NodeIndexSeek	15	RANGE INDEX start:Event(finish_time) WHERE finish_time = cache[end.start_time], cache[start.finish_t ime]	349492	Fused in Pipeline 1
+CacheProperties	16	cache[end.activity_id], cache[end.track_id]	288374	
+NodeByLabelScan	17	end:Event	288374	Fused in Pipeline 0

Figura 4.6: Esecuzione della query con la definizione di indici



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]) 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	1675454	
+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]) 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	1675454	
+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 cache[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_time] = cache[start.finish_time] OR cache[end.activity_id] = cache[start.activity_id] + $autoint_0)	1675454	
+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 4.7: Esecuzione della query senza la definizione di indici

4.5 Analisi descrittiva dei dati

Prima di soffermarsi sulla definizione di una query per la generazione dei prefix-IG, sono state svolte delle interrogazioni per analizzare la struttura e le caratteristiche del dataset, utilizzando una porzione ridotta del dataset BPI12.

4.5.1 Risorse

Un aspetto interessante quando si lavora con degli event log che contengono informazioni relative a dei processi riguarda le risorse. Infatti, è fondamentale tenere in considerazione che all'interno di qualsiasi processo le risorse sono limitate e può essere interessante individuare se sono presenti delle risorse sovraccaricate rispetto ad altre. Per tale ragione sono state sviluppate alcune interrogazioni per acquisire informazioni a riguardo.
In particolare, è interessante capire quali risorse sono occuppate in un certo intervallo di tempo. Un esempio di query che individua tali risorse su un intervallo di tempo specificato è riportata nel Listing 4.7.

```
MATCH (e:Event)
WHERE e.start_time>=datetime("2011-10-01T10:15:00Z") AND e.finish_time<=datetime("2011-10-01T10
    :30:00Z") AND e.resource<>"No_resource"
RETURN DISTINCT(e.resource), e.start_time, e.finish_time
```

Listing 4.7: Query per determinare le risorse occupate in un intervallo di tempo

Il risultato dell'esecuzione di questa query è riportato nella Figura 4.8.

	(e.resource)	e.start_time	e.finish_time
1	"10912"	"2011-10-01T10:15:43Z"	"2011-10-01T10:16:48Z"
2	"10912"	"2011-10-01T10:16:48Z"	"2011-10-01T10:16:48Z"
3	"10912"	"2011-10-01T10:16:48Z"	"2011-10-01T10:16:49Z"
4	"10912"	"2011-10-01T10:26:42Z"	"2011-10-01T10:27:07Z"

Figura 4.8: Risorse occupate in un intervallo di tempo specifico

Inoltre, si potrebbero voler identificare le risorse più occupate, ovvero quelle che eseguono più attività, in un certo istante di tempo. In questo caso, però, l'interrogazione è più articolata della precedente ed è riportata nel Listing 4.8.

```
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 4.8: Query per determinare le risorse più occupate

La query ricerca 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*, ossia la risorsa che esegue l'attività, contando il numero di eventi che corrispondono a ciascuna combinazione di *activity_start_time* e *resource* e assegnando questo conteggio a *resource_count*. Infine, i risultati vengono ordinati in maniera descrente.
Una porzione del risultato ottenuto dall'esecuzione della query è mostrato in Figura 4.9.

	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

Figura 4.9: Risorse più occupate per ogni istante di tempo

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. La query definita per individuare tale risorsa è riportata nel Listing 4.9.

```
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 4.9: Query per determinare la risorsa a cui sono assegnate più attività

Con il sottoinsieme di dati scelto per il testing delle query, risulta che la risorsa più occupata è la 112, come mostrato in Figura 4.10.

	resource	resource_count
1	"112"	102

Figura 4.10: 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. Nel Listing 4.10 è riportata la query definita.

```
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 4.10: Query per determinare l'occupazione delle risorse all'avvio di ogni attività

La query ritorna i risultati nel formato riportato in Figura 4.11.

	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 4.11: Risorse più occupate per ogni istante di tempo

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 query riportata nel Listing 4.11.

```
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 4.11: Query per determinare le attività che le risorse possono eseguire

In Figura 4.12 sono riportati alcuni dei risultati ottenuti.

	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_SE
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 4.12: Attività che possono essere eseguite da ciascuna risorsa

4.5.2 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, nel Listing 4.12 è riportata una query che cerca i nodi di tipo *Event* e filtra questi eventi per includere solo quelli che appartengono all'intervallo di tempo specificato. La query restituisce solo gli identificatori distinti delle tracce (*track_id*) degli eventi attivi in quell'intervallo.

```
MATCH (e:Event)
WHERE e.start_time>=datetime("2011-10-01T10:15:00Z") AND e.finish_time>=datetime("2011-10-01T10
    :30:00Z")
RETURN DISTINCT e.track_id
```

Listing 4.12: Query per determinare i case attivi nell'intervallo di tempo specificato

Per l'intervallo di tempo considerato, il risultato ottenuto è mostrato in Figura 5.7.

	e.track_id
1	"173706"
2	"173709"

Figura 4.13: Case attivi durante l'intervallo di tempo specificato nella query del Listing 4.12

Questa query fornisce una visione dello stato del sistema in uno specifico intervallo di tempo. Tuttavia, potrebbe essere interessante analizzare l'andamento generale 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 riportata nel Listing 4.13 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, ossia quelli già iniziati ma non ancora finiti. 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 4.13: Query per determinare i case attivi ad ogni istante di tempo

Nella Figura 4.14 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"	["173706", "173703"]
12	"2011-10-01T09:45:37Z"	["173706", "173703"]
13	"2011-10-01T09:46:18Z"	["173703", "173706"]
14	"2011-10-01T09:57:41Z"	["173709", "173706", "173703"]

Figura 4.14: Case attivi per ogni istante di tempo

4.5.3 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

La query riportata nel Listing 4.14 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 4.14: Query per determinare le attività che possono essere istantanee

Nella Figura 4.15 sono riportate le attività che possono essere istantanee.

	e.event_name
1	"START"
2	"A_PARTLYSUBMITTED"
3	"W_Completerenaanvraag"
4	"O_SENT"
5	"W_Nabellenoffertes"
6	"O_CANCELLED"

Figura 4.15: Attività che possono essere istantanee

Inoltre, si è voluta valutare la frequenza con cui ciascuna attività è istantanea rispetto al numero totale delle sue occorrenze tramite la query riportata nel Listing 4.15.

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

Listing 4.15: Query per determinare la frequenza delle attività istantanee

Alcuni dei risultati ottenuti con tale query sono riportati nella Figura 4.16.

	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 4.16: Frequenza delle attività istantanee rispetto al numero totale di occorrenze

Attività parallele

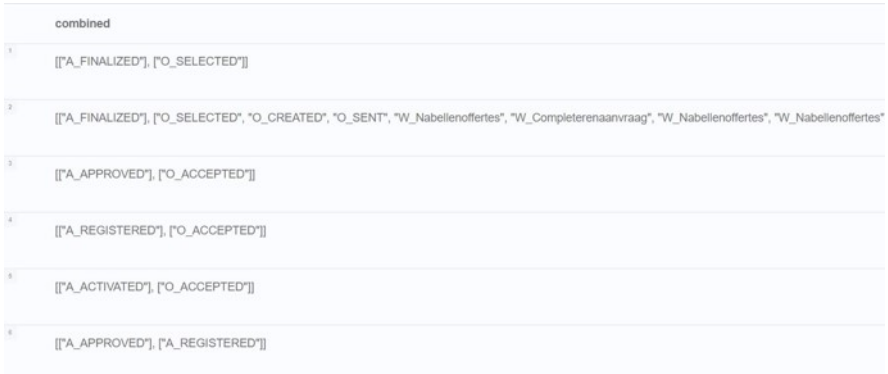
In modo simile al caso dell'attività istantanee, sono state identificate le attività che possono essere eseguite in parallelo, anche se in questo caso la situazione è più complicata.
La query riportata nel Listing 4.16 individua coppie di attività che iniziano nello stesso momento e sono eseguite in 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 4.16: Query per determinare le attività che possono essere parallele

Alcuni dei risultati ottenuti dall'esecuzione di questa query sono mostrati nella Figura 4.17. Si può notare che non necessariamente i parallelismi sono tra due sole attività.



	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 4.17: Attività che possono essere eseguite in parallelo ad un'altra attività

Oppure, in maniera simile all'interrogazione precedente, è possibile ricavare le attività parallele presenti all'interno di ogni grafo e visualizzare graficamente (Listing 4.17).

```
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 4.17: Query per individuare i parallelismi all'interno dei grafi

I risultati ottenuti sono mostrati in Figura 4.18.

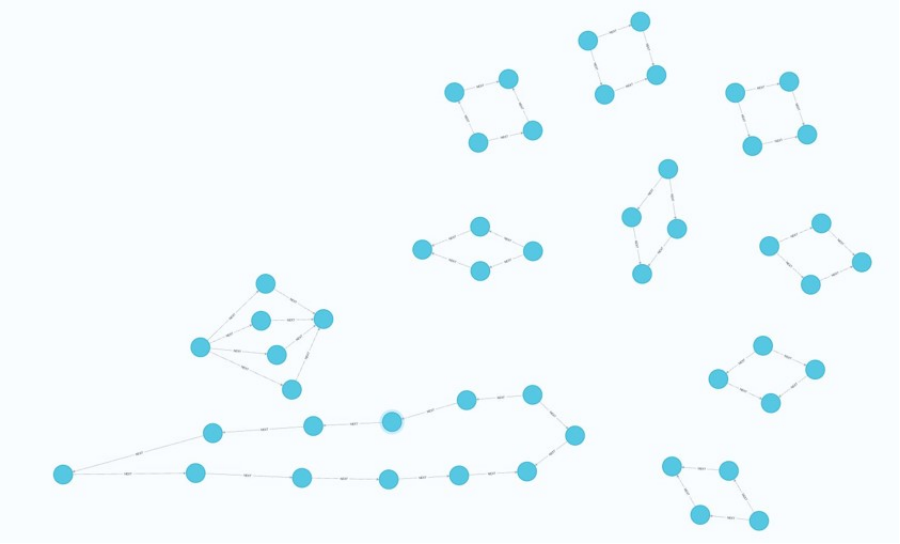


Figura 4.18: Attività eseguite in parallelo

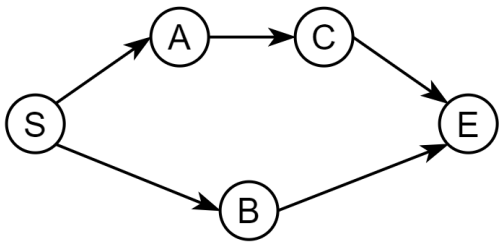
CAPITOLO 5

Prefix Instance Graph

5.1 Estrazione dei prefissi

Come discusso in precedenza, 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 (Chiorrini *et al.* [2023]).

Si consideri l'Instance Graph riportato in Figura 5.1.



$\sigma = ((1,S), (2,A), (3,B), (4,C), (5,E))$

Figura 5.1: Esempio Instance Graph

Dato che gli Instance Graph sono, in realtà, degli alberi, cioè dei particolari tipi di grafi che non contengono cicli, i prefissi possono essere definiti attraversando il grafo e considerando sempre un nodo in più. Considerando l'IG di Figura 5.1, si può utilizzare la query riportata nel Listing 5.1 per definire i suoi prefissi.

```
MATCH p = (start:Event)–[:NEXT*]–>(end:Event)
WHERE start.activity_id=1 AND end.activity_id=5
RETURN p
```

Listing 5.1: Query per la generazione dei prefissi

Questa query permette di individuare i cammini che partono da un nodo *start* e terminano in un nodo *end* e sono collegati tra loro da una o più relazioni di tipo $NEXT$, senza limitazioni sulla lunghezza, permettendo quindi un numero arbitrario di salti tra i nodi intermedi. La condizione $WHERE$ restringe la ricerca ai soli cammini che partono da un nodo con *activity_id* pari a 1 e terminano in un nodo con *activity_id* pari a 5. Va notato che, per come sono stati costruiti i grafi, il nodo con *activity_id* pari a 1 è sempre il primo nodo del grafo, mentre la scelta di *activity_id* pari a 5 come nodo finale è legata all'esempio di Figura 5.1, ma in generale, il numero varia in base alla lunghezza del grafo considerato. Infine, la clausola *RETURN p* specifica che il risultato della query deve essere l'intero cammino p, includendo tutti i nodi e le relazioni tra di essi.
Tuttavia, il risultato che si ottiene utilizzando questo approccio risulta essere corretto solo nel caso in cui le tracce siano completamente sequenziali. Infatti, se si considera l'IG di Figura 5.1, l'esecuzione della query restituisce due risultati distinti, come riportato in Figura 5.2.

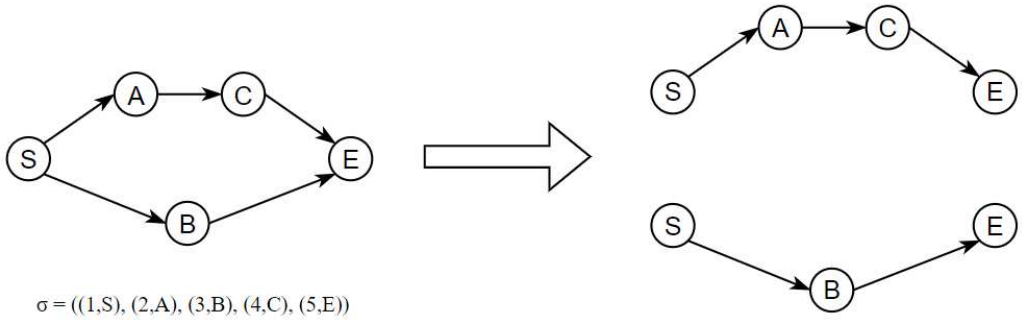


Figura 5.2: Risultato della query 5.1

La scelta del tipo di traversal del grafo per generare i prefissi è fondamentale per ottenere il risultato desiderato. Per generare i prefissi degli Instance Graph, è necessario utilizzare un traversal di tipo Breadth-First (BFS). Questo approccio è adatto quando si devono gestire nodi con figli multipli, ovvero in presenza di parallelismi. In questi casi, la BFS esplora i nodi a livello di profondità, garantendo che tutti i nodi di un livello siano visitati prima di passare al livello successivo. Se ci sono più figli di un nodo padre, la BFS permette di creare un collegamento fittizio tra i figli, senza dover necessariamente attraversare il collegamento diretto tra il nodo padre e ciascuno dei suoi figli. Ciò vuol dire che la ricerca Breadth-First analizza i nodi scandendoli per livelli e solo dopo aver visitato tutti i nodi di un livello, passa a quelli del livello successivo.
D'altro canto, Neo4j di default utilizza l'algoritmo Depth-First (DFS) per attraversare il grafo. La traversa Depth-First esplora un cammino del grafo fino alla sua profondità massima prima di tornare indietro e provare un altro cammino. Questo tipo di traversal è più efficiente in termini di utilizzo della memoria rispetto alla BFS. Durante l'esecuzione, la DFS mantiene in memoria solo il cammino corrente e le informazioni necessarie per tornare indietro, riducendo significativamente il numero di nodi da tenere contemporaneamente in memoria. Questo vantaggio è particolarmente rilevante in grafi molto profondi o con un elevato numero di nodi. 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.
Nella Figura 5.3 i due approcci di navigazione, DFS e BFS, sono messi a confronto per illustrare come ciascuno esplora i nodi del grafo.

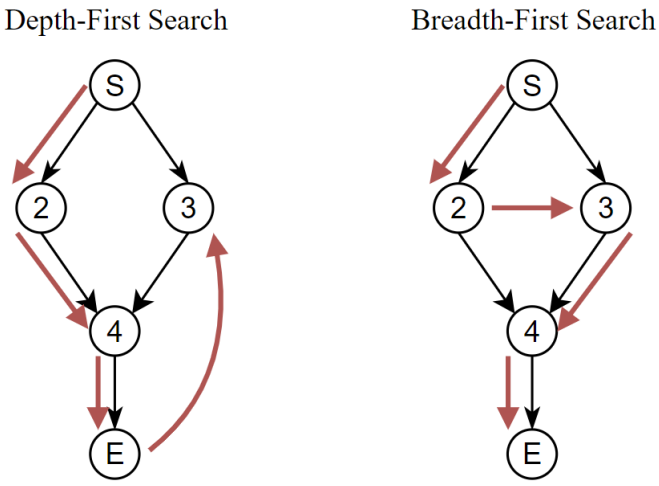


Figura 5.3: DFS vs BFS

Partendo dallo stesso grafo, i due approcci esplorano l'albero in modo differente. Le frecce rosse indicano l'ordine in cui i nodi vengono visitati. Utilizzando una ricerca di tipo DFS, non è possibile ottenere il risultato desiderato per la generazione dei prefissi. Infatti, quando si incontra un parallelismo, la DFS continua l'esplorazione partendo da uno dei due figli e proseguendo fino alla profondità massima, per poi tornare al livello del parallelismo e proseguire con l'altro figlio. Questo genera due sottografi distinti e non permette di definire i prefix-IG secondo la loro definizione.

Di conseguenza, è stata sviluppata una soluzione alternativa. 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 e ciò non sarebbe stato possibile in Neo4j.
Il codice utilizzato è riportato nel Listing 5.2.

```
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 5.2: Query per la generazione dei prefissi in Python

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.

Un'ulteriore soluzione ha previsto di implementare direttamente in Neo4j una ricerca di tipo Breadth-First. Per far questo, è stata utilizzata la funzione *subgraphAll* della libreria *apoc* (Listing 5.3).

```
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) as relsSlice,  target
```

Listing 5.3: Query per la generazione dei prefissi con l'approccio BFS

Inizialmente, viene individuato il nodo di partenza che, per ogni grafo è sempre quello con *activity_id* pari a 1. Successivamente, la procedura *apoc.path.subgraphAll* viene chiamata per generare l'intero sottografo che ha origine dal nodo di partenza, seguendo una traversa di tipo BFS. La procedura restituisce due insiemi: *subgraphNodes* che contiene i nodi del sottografo e *subgraphRelationships* che contiene le relazioni tra questi nodi. La query continua creando sottosequenze incrementali di nodi e archi a partire dal sottografo generato. Per ogni livello, si estrae una porzione (*slice*) dei nodi e delle relazioni che include tutti gli elementi fino a quel livello specifico. Inoltre, il nodo al livello corrente viene identificato come *target*. Una volta che le relazioni nella porzione corrente sono state scomposte, la query verifica quali nodi sono collegati da queste relazioni e se tali nodi appartengono alla porzione corrente di nodi. In questo modo, la query assicura che solo le relazioni pertinenti all'interno della sottosequenza di nodi vengano considerate. Infine, la query restituisce l'insieme dei nodi correnti, le relazioni distinte che collegano questi nodi e il target relativo al prossimo nodo. Ad esempio, se la sottosequenza di nodi *nodesSlice* contiene i nodi con *activity_id* pari a 1 e 2 e la sottosequenza di relazioni *relsSlice* include la relazione che collega questi due nodi, il nodo *target* è quello con *activity_id* pari a 3.

5.1.1 Esperimenti

Anche nel caso della generazione dei prefissi sono state eseguite più prove per valutare le prestazioni delle varie soluzioni individuate.
In realtà, oltre alle query precedentemente discusse, è stato anche considerato un terzo approccio che calcola i prefissi direttamente in Python utilizzando alcune librerie come NetworkX, ma senza passare per il tramite di Neo4j.
In particolare, questo codice si sviluppa come segue. Il file contenente i dati dei grafi viene importato come un DataFrame utilizzando Pandas. Ogni grafo separato da un XP viene importato come grafo di NetworkX. Successivamente, viene verificata la correttezza del grafo e, se risulta corretto, viene aggiunto alla lista dei grafi corretti. Per ogni grafo nella lista dei grafi corretti, viene creato un nuovo sottografo. Ogni nodo del grafo viene aggiunto al sottografo insieme agli archi che lo coinvolgono. Successivamente, viene definito un target per il prefisso ed, infine, il sottografo viene aggiunto alla lista dei sottografi.

In tabella 5.1 sono riportati i tempi delle prove eseguite per la generazione dei prefissi nei vari casi, a partire dal dataset con una dimensione ridotta fino ad arrivare al dataset completo.

Numero di grafi	Non ottimizzato	BFS	Python
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 5.1: Confronto dei tempi per la generazione dei prefissi

Ciò che si può notare è che l'approccio non ottimizzato, ossia quello che ha richiesto di integrare Python con Neo4j per gestire una query di tipo dinamico, ha comportato un tempo di esecuzione notevolmente superiore, quasi 5 ore contro 16 secondi con l'approccio BFS e 7 secondi nell'approccio in Python.

Nella Figura 5.4 è riportato un grafico tridimensionale, relativo alla prova sulla query dinamica, dove l'asse X indica il numero di prefissi da calcolare nell'iterazione corrente, l'asse Y indica la lunghezza del prefisso e l'asse Z indica il tempo impiegato per calcolare il prefisso, misurato in secondi.

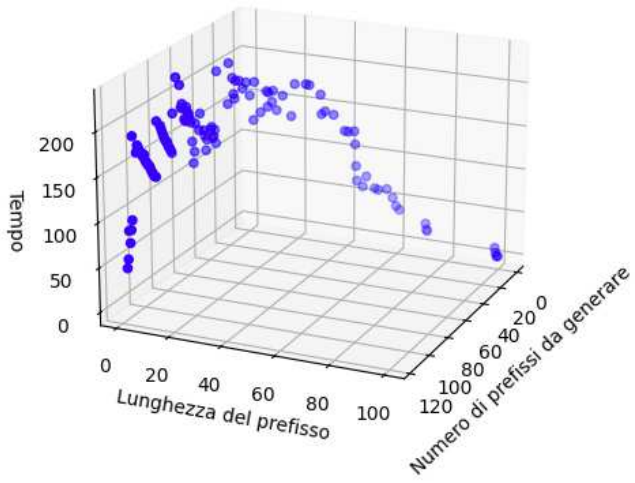


Figura 5.4: Tempo di calcolo dei prefissi per lunghezza del prefisso e numerosità di grafi

Come si può osservare dal grafico, il tempo necessario per il calcolo del prefisso aumenta all'aumentare della lunghezza del prefisso stesso e aumenta anche all'incrementare del numero di prefissi da generare.
Il dataset composto da 100 grafi, richiede di calcolare prefissi di lunghezza minima pari a 1 e di lunghezza massima pari a 117. In particolare, per comprendere la relazione tra queste tre variabili, si considerino due casi estremi:

- calcolo di 100 prefissi di lunghezza 1;
- calcolo di 1 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.

Volendo mettere a confronto la query dinamica e la query che sfrutta una ricerca di tipo BFS, 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. Infatti, tale operatore trasforma una lista di nodi in righe singole su cui poi si eseguono operazioni di $MATCH$ e *OPTIONAL MATCH* e questo può introdurre inefficienza, soprattutto con dataset di grandi dimensioni, poichè ogni elemento della lista viene scomposto in una nuova riga e, di conseguenza, processato individualmente.
Invece, la query che si basa sull'utilizzo dell'approccio BFS tramite *apoc.path.subgraphAll* è altamente ottimizzata per esplorare i nodi e le relazioni a livello di grafo. 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.

Visto il tempo estremamente elevato necessario per il calcolo dei prefissi tramite l'approccio dinamico su una porzione di dataset contenente solo 100 grafi, si è deciso di non proseguire con ulteriori prove. Al contrario, sono stati condotti dei test sia con la query che utilizza BFS sia con l'approccio che non coinvolge Neo4j. Dalla Tabella 5.1 è evidente che, mentre i dati sono limitati, ci sono poche differenze tra la query sviluppata in Cypher e il codice Python. Tuttavia, queste differenze diventano più significative man mano che le dimensioni del dataset aumentano.

I seguenti grafici mostrano l'utilizzo della CPU e della memoria durante il calcolo dei prefissi in Python (Figura 5.5) e tramite la query che sfrutta la procedura *apoc.path.subgraphAll* per navigare i grafi in modalità BFS (Figura 5.6).

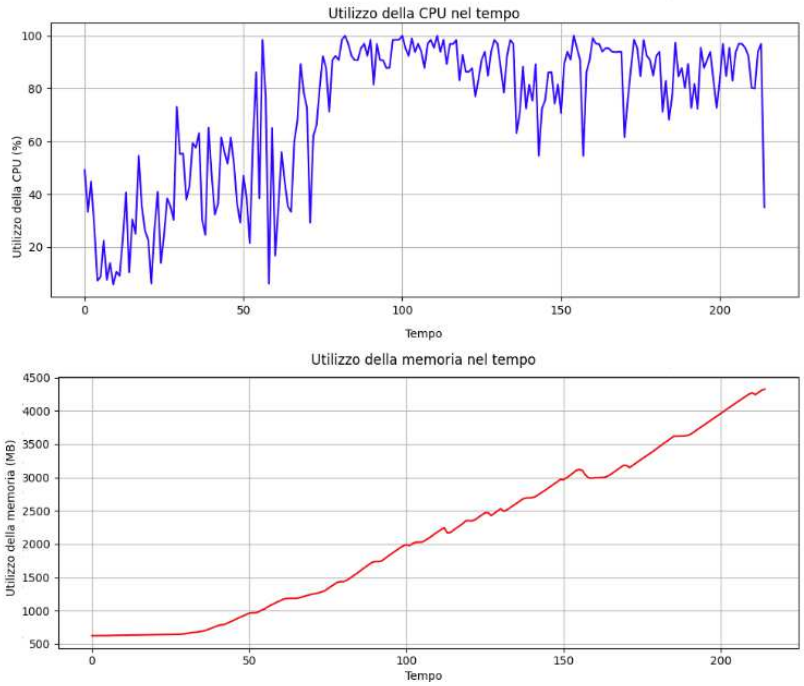


Figura 5.5: Utilizzo di memoria e CPU durante il calcolo dei prefissi con Python

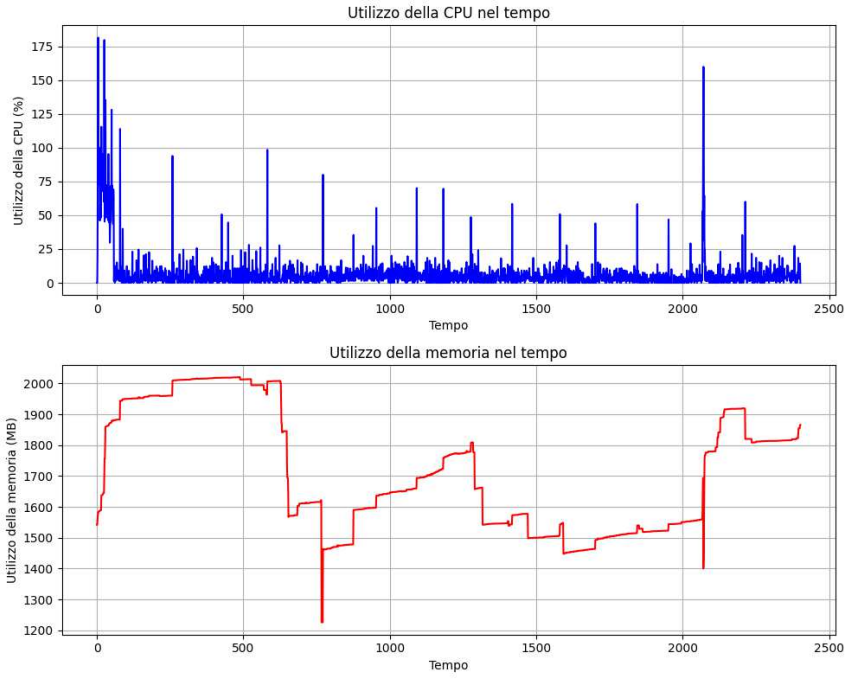


Figura 5.6: Utilizzo di memoria e CPU durante il calcolo dei prefissi con BFS

Nel grafico di Figura 5.5, l'andamento del consumo di CPU mostra una crescita costante, indicando un incremento della complessità computazionale man mano che l'algoritmo elabora i dati. La memoria, similmente, mostra un aumento continuo, indicativo di un accumulo progressivo di dati durante l'esecuzione dell'algoritmo. L'andamento relativamente lineare della memoria suggerisce che non ci sono rilasci significativi di memoria durante l'esecuzione.
L'approccio Python costruisce i sottografi in modo iterativo e incrementale rispetto a Neo4j. Questo porta a un costante aumento del carico di lavoro della 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 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.
Tuttavia, la crescita costante della memoria che si vede in Figura 5.5 suggerisce che Python carica e mantiene tutti i dati in memoria durante l'esecuzione. Questo significa che per dataset di dimensioni maggiori, l'uso della memoria continuerà a crescere, potenzialmente saturando tutta la memoria disponibile del dispositivo. Se il dataset è molto grande, Python potrebbe esaurire la memoria disponibile, causando errori di memoria e interrompendo il processo. Questo rende l'approccio meno scalabile per dataset di grandi dimensioni.
L'approccio con Neo4j mostra un utilizzo della memoria che cresce a gradini e include periodi di stabilità e rilasci di memoria. Questo indica una gestione più dinamica e controllata della memoria. Nonostante l'utilizzo della CPU e della memoria possa essere più variabile e potenzialmente maggiore in alcuni momenti, Neo4j sembra in grado di gestire dataset più grandi senza esaurire completamente la memoria disponibile. Questo rende Neo4j più robusto e scalabile per grandi dataset, anche se a costo di tempi di esecuzione potenzialmente più lunghi. Pertanto, se si prevede di lavorare con dataset di dimensioni considerevoli, Neo4j sarebbe probabilmente la scelta migliore per garantire che il processo possa completarsi senza esaurire le risorse di sistema, anche se questo potrebbe significare tempi di esecuzione più lunghi.

Vista la possibilità di modificare i parametri di configurazione relativi alla memoria, è stata effettuata un'ulteriore prova per vedere come si comporta Neo4j avendo la possibilità di usare 4 GB di memoria, in modo da avere una situazione paragonabile a quella in Python. Pertanto, è stata eseguita una prova con i seguenti parametri:

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

Il tempo necessario per il calcolo dei prefissi è risultato essere superiore a quello ottenuto con la configurazione precedente. È possibile che aumentando la memoria disponibile per Neo4j da 2 GB a 4 GB, il tempo di esecuzione delle operazioni possa effettivamente aumentare invece di diminuire. Questo potrebbe sembrare controintuitivo, ma ci sono diverse ragioni per cui ciò potrebbe accadere.
Innanzitutto, bisogna considerare il ruolo del Garbage Collection (GC). Quando si aumenta la memoria disponibile per Neo4j, il Garbage Collector di Java, che gestisce la memoria per Neo4j, deve lavorare con una quantità maggiore di memoria. Questo può incrementare il tempo necessario per eseguire il GC, causando pause più lunghe o più frequenti durante l'esecuzione delle operazioni. In particolare, se il heap size è molto grande, un ciclo completo di Garbage Collection (Full GC) può richiedere molto tempo, influenzando negativamente le

prestazioni complessive dell'applicazione.
Inoltre, aumentare la memoria non garantisce automaticamente un uso efficiente di questa risorsa. Potrebbe accadere che il sistema non sfrutti appieno la cache disponibile, oppure che ci sia un overhead maggiore nella gestione di una memoria più grande. Questo può portare a un'inefficienza nell'uso della memoria. Una quantità maggiore di memoria può anche portare a una maggiore frammentazione della memoria, che può ridurre le prestazioni complessive.
Un altro aspetto da considerare è che l'aumento della memoria del heap potrebbe non essere bilanciato con altri parametri di configurazione, come la dimensione della cache delle pagine (dbms.memory.pagecache.size). Questo sbilanciamento potrebbe portare a un uso inefficiente della memoria complessiva del sistema. Potrebbe essere necessario ottimizzare ulteriormente i parametri del Garbage Collector per gestire la nuova configurazione di memoria. Senza un tuning adeguato del GC, le prestazioni potrebbero non migliorare come previsto.
Infine, bisogna considerare i limiti di scala. A un certo punto, aumentare semplicemente la memoria potrebbe non apportare miglioramenti significativi se altri fattori, come l'architettura dell'applicazione o i limiti dell'hardware, diventano il collo di bottiglia. In tal caso, l'aumento della memoria potrebbe non avere l'effetto desiderato sulle prestazioni.
Per diagnosticare e ottimizzare le prestazioni dopo aver aumentato la memoria, è importante monitorare attentamente il comportamento del Garbage Collector. Utilizzare strumenti di monitoraggio del GC può aiutare a verificare se il GC sta causando pause lunghe o frequenti. Analizzare i log del GC può aiutare a identificare eventuali problemi di configurazione. Inoltre, potrebbe essere utile ottimizzare altri parametri di configurazione, come la dimensione della cache delle pagine, e considerare l'uso di parametri avanzati del GC per migliorare le prestazioni.

5.2 Active case

Per ogni prefisso calcolato, è interessante individuare tutti gli altri case attivi nel medesimo intervallo temporale. Fissato un intervallo di tempo che inizia con l'inizio del prefisso e termina con la fine dell'ultima attività del prefisso considerato, l'obiettivo è determinare quali altri case sono attivi, ossia quali altri case, relativi ad una traccia differente, sono in esecuzione nello stesso intervallo temporale.
Nella Figura 5.7 sono illustrati degli esempi di come l'esecuzione di altri case può situarsi rispetto a un dato intervallo di tempo $[start, end]$.
I case (1) e (6) non sono case attivi, poiché la loro esecuzione avviene completamente al di fuori dell'intervallo di tempo considerato. In particolare, il case (1) è eseguito interamente prima dell'inizio dell'intervallo di tempo, mentre il case (6) è eseguito interamente dopo la fine dell'intervallo di tempo. Tutti gli altri case sono, invece, case attivi. Tuttavia, essi non devono essere trattati allo stesso modo. In particolare:

- Case (2) e case (3): questi case terminano prima della fine dell'intervallo di tempo considerato. In questo caso, viene riportato il prefisso di lunghezza massima, cioè l'intero case fino alla sua conclusione.

- Case (4) e case (5): questi case terminano dopo la fine dell'intervallo di tempo considerato. In tal caso, è necessario identificare l'ultima attività iniziata prima del termine dell'intervallo e considerare solo il prefisso fino a tale attività.

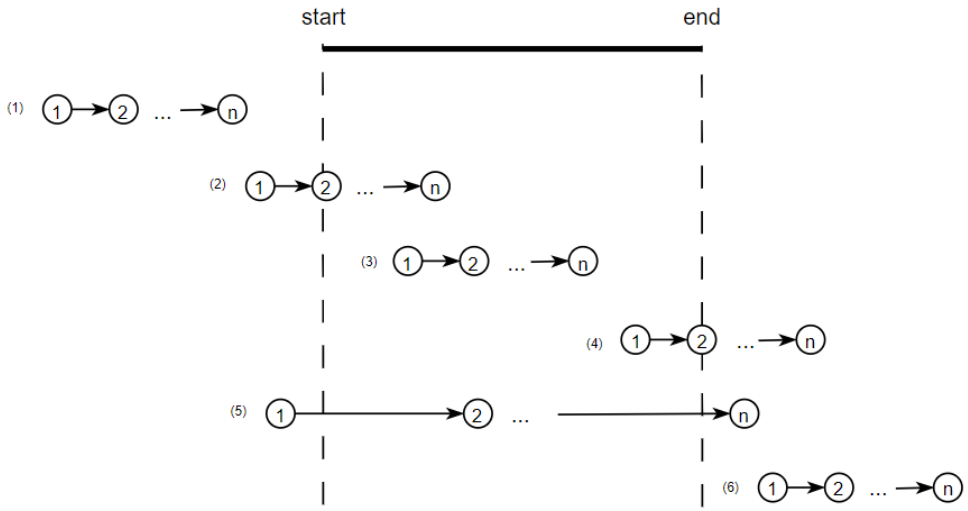


Figura 5.7: Case attivi in un intervallo di tempo

Nel Listing 5.4 è riportata la query definita per individuare i case attivi per ogni prefisso.

```
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
WITH nodesSlice, collect(DISTINCT r) AS relationships, target
WITH nodesSlice, relationships, target,
   head(nodesSlice) AS firstNode,
   last(nodesSlice) AS lastNode
WITH nodesSlice, relationships, target, firstNode.start_time AS start_time, lastNode.finish_time AS end_time,
    firstNode.track_id AS track_id_prefix
MATCH (other:Event)
WHERE other.track_id <> track_id_prefix AND
    ((other.finish_time >= start_time AND other.start_time <= end_time) OR
    (start_time >= other.start_time AND start_time <= other.finish_time) OR
    (end_time >= other.start_time AND end_time <= other.finish_time))
RETURN nodesSlice, relationships, target, start_time, end_time, collect(other.track_id + '_' + other.activity_id)
    AS otherEvents
```

Listing 5.4: Query per individuare i case attivi

In particolare, questa query riprende la query definita precedentemente per individuare tutti i prefissi utilizzando la BFS. In questo caso però, per ogni livello della ricerca, si seleziona una porzione dei nodi (*nodesSlice*) e delle relazioni (*relsSlice*) fino a quel livello e si identifica il nodo target corrispondente al livello corrente. Si associa quindi ogni relazione r ai nodi n e m presenti in nodesSlice e si raccolgono tutte le relazioni distinte in una collezione relationships. Inoltre, si identificano il primo nodo ($firstNode$) e l'ultimo nodo ($lastNode$) della sequenza di nodi considerati. Da questi nodi si estraggono i tempi di inizio ($start\_time$) e di fine (end_time) dell'intervallo di tempo considerato e si memorizza l'identificativo della traccia considerata ($track\_id\_prefix$).
La fase successiva della query consiste nel cercare altri eventi (*other*) che appartengono a

tracce differenti rispetto a quello di riferimento e che risultano attivi all'interno dell'intervallo di tempo definito. Un evento viene considerato attivo se soddisfa le condizioni citate in precedenza. Infine, la query restituisce i nodi del sottografo fino al livello corrente (*nodesSlice*), le relazioni distintive (*relationships*), il nodo target corrente (*target*), i tempi di inizio e di fine dell'intervallo di tempo considerato (*start_time* e *end_time*), e una lista di stringhe composte da due campi: l'identificativo della traccia e l'identificativo dell'ultima attività eseguita nel case attivo.

Al fine di valutare la correttezza della query appena definita, è stata aggiunta la clausola *WHERE* nel primo *MATCH* per specificare un particolare *track_id*. Ad esempio, aggiungendo la clausola "WHERE startNode.track_id = 173715" una parte dei risultati che si ottengono sono riportati nella Figura 5.8. In particolare, i nodi e le relazioni del prefisso che viene mostrato in parte è composto da due nodi aventi *activity_id* rispettivamente pari a 1 e 2 e dalla relazione che li collega tra di loro.
Il risultato che si ottiene (*otherEvents*) è una lista che contiene delle stringhe composte da due valori separate da un underscore, del tipo *trackId_activityId* dove:

- *trackId* rappresenta l'identificativo della traccia corrispondente ad un case attivo;
- *activityId* rappresenta l'identificativo dell'ultima attività iniziata prima del termine dell'intervallo di tempo considerato.

A partire da questa lista è possibile ricavare quali sono i case attivi dato che si hanno a disposizione sia l'identificativo della traccia che l'identificativo dell'ultima attività da considerare.

Figura 5.8: Risultato dell'esecuzione della query nel Listing 5.4 per $track\_id = 173715$

La query riportata nel Listing 5.4 causa un errore di memoria (out of memory) già con un carico di soli 100 grafi. Questo problema è dovuto alla complessità computazionale e alla quantità di memoria necessaria per gestire il sottografo e le operazioni successive su grandi insiemi di dati.

Per affrontare il problema di scalabilità, è stato sviluppato un nuovo approccio che mira a ridurre l'uso della memoria e a migliorare l'efficienza computazionale, consentendo di gestire un numero significativamente maggiore di grafi senza incorrere in problemi di memoria.
Si è voluta massimizzare l'efficacia di Neo4j nell'analisi dei grafi, integrandola con i benefici derivanti dall'utilizzo di database relazionali. Un approccio chiave è stato la materializzazione dei dati. In particolare, le informazioni relative ai prefissi dei grafi sono state salvate utilizzando la query definita in Neo4j. D'altra parte, per identificare i case attivi e condurre operazioni di filtraggio basate su dati temporali, si è optato per l'uso di database relazionali, impiegando librerie come Pandas per la manipolazione dei dati. A tal proposito sono stati definiti due ulteriori database. Il primo di questi è composto da quattro campi:

- *track_id*: rappresenta l'identificativo univoco della traccia;
- *index*: rappresenta l'identificativo univoco dell'attività;
- *start_time_prefix*: rappresenta il tempo di inizio del prefisso;
- *finish_time_last_activity*: rappresenta il tempo di fine dell'ultima attività considerata per il prefisso corrente.

Un esempio della struttura di questo database è riportato nel Listing 5.5.

```
track_id,index,start_time_prefix,finish_time_last_activity
173688,1,2011-10-01 00:38:43+00:00,2011-10-01 00:38:43+00:00
173688,2,2011-10-01 00:38:43+00:00,2011-10-01 00:38:44+00:00
173688,3,2011-10-01 00:38:43+00:00,2011-10-01 00:38:44+00:00
173688,4,2011-10-01 00:38:43+00:00,2011-10-01 00:39:37+00:00
173688,5,2011-10-01 00:38:43+00:00,2011-10-01 00:39:38+00:00
```

Listing 5.5: Esempio di dati salvati nel database

Per ogni traccia presente nel dataset, è presente un numero di righe pari al numero di nodi che compongono il rispettivo grafo. Tutte le righe relative alla stessa traccia, sono caratterizzate dallo stesso tempo di inizio del prefisso, ma da un differente tempo di fine che corrisponde al tempo di fine dell'ultima attività del prefisso. Questo permette di definire l'intervallo che si vuole considerare per il calcolo dei casi attivi rispetto al prefisso scelto. Queste informazioni non sono però sufficienti per la corretta gestione dei case attivi, quindi, è stato definito un altro database caratterizzato da cinque campi:

- *track_id*: rappresenta l'identificativo univoco della traccia;
- *index*: rappresenta l'identificativo univoco dell'attività;
- *event_name*: rappresenta il nome dell'attività;
- *finish*: rappresenta il tempo di fine dell'attività;
- *start*: rappresenta il tempo di inizio dell'attività.

Un esempio della struttura di questo database è riportato nel Listing 5.6.

```
track_id,index,event_name,finish,start
173688,1,START,2011-10-01 00:38:43+00:00,2011-10-01 00:38:43+00:00
173688,2,A_SUBMITTED,2011-10-01 00:38:44+00:00,2011-10-01 00:38:43+00:00
173688,3,A_PARTLYSUBMITTED,2011-10-01 00:38:44+00:00,2011-10-01 00:38:44+00:00
173688,4,A_PREACCEPTED,2011-10-01 00:39:37+00:00,2011-10-01 00:38:44+00:00
173688,5,W_Completerenaanvraag,2011-10-01 00:39:38+00:00,2011-10-01
    00:39:37+00:00
```

Listing 5.6: Esempio di dati salvati nel database con le informazioni temporali sulle attività finali

Anche in questo caso, per ogni traccia presente nel dataset, il numero di righe corrisponde al numero di nodi del rispettivo grafo. Tuttavia, le informazioni temporali sono differenti da quelle riportate nel caso precedente: è possibile determinare il tempo di inizio e di fine dell'attività i-esima, mentre nel caso precedente il tempo di inizio riportato era sempre quello relativo all'inizio del prefisso.

L'algoritmo sviluppato per la definizione dei case attivi si basa sull'utilizzo di tutte queste informazioni. Si considerano tutti i prefissi e per ciascuno si individuano i case attivi contemporaneamente ad esso. Di fatto, ogni riga del primo database permette di individuare gli intervalli di tempo in cui viene eseguito ogni prefisso. Poiché i prefissi sono definiti in modo tale che ogni prefisso considera di volta in volta il nodo con l'id immediatamente successivo rispetto al precedente, ogni riga permette di individuare il tempo di inizio e il tempo di fine dei prefissi che iniziano dal nodo con *index* pari a 1 e che terminano nel nodo con *index* via via successivo. Pertanto, fissato prefisso e, di conseguenza, l'intervallo di tempo in cui viene eseguito, si vanno a ricercare i case che hanno un *track_id* differente da quello corrente e che soddisfano due condizioni temporali:

- il tempo di inizio del case è inferiore del tempo di fine del prefisso corrente;
- il tempo di fine del case è superiore del tempo di inizio del prefisso corrente.

Questo permette di fare un primo filtraggio dei dati che individua quali sono tutti i case attivi, ma ciò non è sufficiente. Se il tempo di fine del case è contenuto nell'intervallo, ossia termina prima dell'istante temporale considerato come fine dell'intervallo, per ogni traccia, vengono individuati molteplici prefissi ed è sufficiente riportare il più grande. Invece, per i case attivi il cui tempo di fine è superiore al tempo di fine del prefisso considerato, bisogna individuare l'ultima attività iniziata prima del termine. Per far questo, si utilizza il db contenente le informazioni temporali relative alle attività finali. In questo caso, dopo aver individuato la stessa traccia, si deve individuare l'ultima attività il cui tempo di inizio è minore del tempo di fine dell'intervallo. Ciò permette di fare un ulteriore filtraggio dei case attivi e, quindi, permette di ridurre ulteriormente la quantità di dati con cui lavorare.
Una volta verificate queste condizioni, per ogni prefisso, vengono riportati i case attivi, individuati dall'identificativo della traccia e dall'identificativo dell'ultima attività eseguita. Ciò permette di ricavare tutte le informazioni sui rispettivi prefissi, a partire dai dati memorizzati a seguito dell'esecuzione della query che individua tutti i prefissi.
Questa procedura viene ripetuta per ogni prefisso presente nel dataset.

5.2.1 Esperimenti

Il codice implementato è stato testato su un dataset di dimensioni ridotte ed i tempi per eseguire tali prove sono riportati in tabella 5.2.

Numero di grafi	Numero di prefissi	Tempo
100	2385	138.86 s
200	4859	547.76 s
500	12409	4680.12 s

Tabella 5.2: Confronto dei tempi per il calcolo dei case attivi

Tutte queste prove hanno mostrato delle somiglianze nell'utilizzo di memoria e CPU. In particolare, l'uso della CPU oscilla significativamente, senza mai raggiungere valori al di sopra del 100%, con dei picchi in vari punti, che indicano periodi di alta domanda di elaborazione.

L'uso della memoria, invece, mostra una tendenza costante ad aumentare nel tempo. Questo aumento nell'uso della memoria suggerisce l'accumulo di processi o la gestione dei dati nel tempo. I dati suggeriscono possibili colli di bottiglia nelle prestazioni durante i periodi di picco dell'uso della CPU. Questi periodi potrebbero indicare quando il sistema è sotto carico elevato, portando potenzialmente a prestazioni più lente o ritardi.
Nella Figura 5.9 è riportato l'utilizzo della memoria e della CPU durante processo di calcolo dei case attivi con il file contenente 500 grafi.

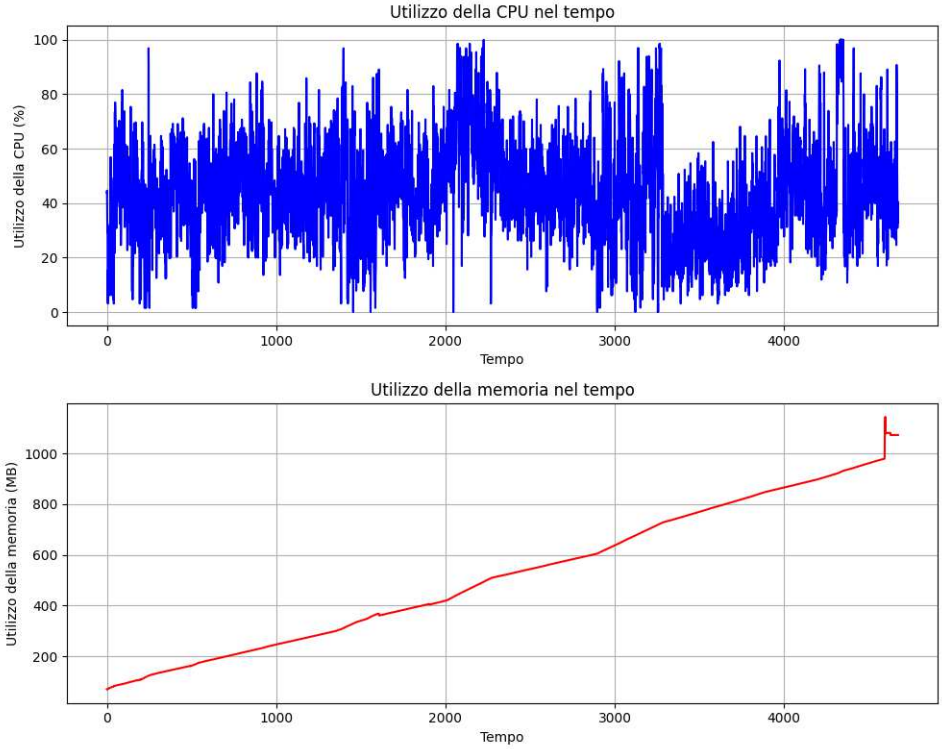


Figura 5.9: Utilizzo di memoria e CPU durante il calcolo dei case attivi

La scelta di non testare il codice su dataset di dimensioni superiori o, addirittura, sul dataset completo è legato principalmente al tempo necessario per il calcolo dei case attivi già con un numero di grafi e, di conseguenza, con un numero di prefissi, ridotto. Inoltre, va notato che passando da 100 grafi a 200 grafi, ossia da 2385 a 4859 prefissi, il tempo necessario per individuare tutti i case attivi non raddoppia. Questo è ragionevole considerato che aumentando il numero di prefissi, aumenta anche la probabilità di avere un numero maggiore di case attivi in parallelo.
Inoltre, va notato che il tempo di calcolo dei casi attivi è fortemente legato alla struttura del log considerato, non solo dalla sua dimensione ma anche dai tempi di esecuzione dei suoi processi. Infatti, a prescindere dalla dimensione del log, più esso contiene processi che vengono eseguiti in maniera parallela, maggiore sarà il tempo richiesto per il calcolo dei case attivi per ogni prefisso.

Queste osservazioni mettono in evidenza la complessità computazionale associata all'aumento del numero di grafi e prefissi. In particolare, l'incremento del tempo di calcolo non è lineare, suggerendo che la gestione dei dati diventa progressivamente più onerosa man mano che il dataset cresce. Questo comportamento potrebbe essere dovuto a vari fattori, tra cui l'aumento della complessità delle interazioni tra i grafi e il numero crescente di case attivi da elaborare simultaneamente. L'aumento non lineare del tempo di calcolo indica che il sistema

potrebbe non essere scalabile in modo efficiente per dataset molto grandi. La presenza di picchi nell'utilizzo della CPU suggerisce che ci sono fasi del calcolo particolarmente intensive, probabilmente legate a operazioni di ricerca o confronto tra i prefissi.
Per affrontare questi problemi e migliorare l'efficienza del calcolo, si potrebbero esplorare diverse strategie. Implementare tecniche di parallelizzazione per distribuire il carico di lavoro su più core della CPU o, se possibile, su più macchine potrebbe ridurre significativamente i tempi di calcolo. Migliorare la gestione della memoria per evitare l'accumulo non necessario di dati potrebbe includere l'uso di tecniche di garbage collection più aggressive o l'ottimizzazione dell'allocazione della memoria. Implementare meccanismi di caching e buffering per ridurre l'accesso ripetuto ai dati potrebbe migliorare l'efficienza complessiva del sistema.
Inoltre, è importante sottolineare che le prestazioni osservate dipendono fortemente dalla macchina su cui sono state eseguite le prove. Le specifiche hardware, come la velocità del processore, la quantità di RAM disponibile e la configurazione del disco, influenzano significativamente i tempi di calcolo e l'uso delle risorse. Le osservazioni e i dati raccolti potrebbero variare se le prove venissero eseguite su una macchina con specifiche diverse. Di conseguenza, per ottenere una valutazione più completa e accurata delle prestazioni del sistema, sarebbe opportuno condurre test su diverse configurazioni hardware, in modo da identificare e mitigare eventuali limitazioni legate all'infrastruttura utilizzata.

CAPITOLO 6

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 un'adeguata allocazione della memoria possa influenzare significativamente le prestazioni del sistema, migliorando i tempi di risposta, la gestione dei processi e dei dati, riducendo il rischio di colli di bottiglia e migliorando la scalabilità complessiva.
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 calcolatore 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. L'uso di una macchina con maggiori risorse avrebbe permesso di esplorare una gamma più ampia di configurazioni, fornendo un quadro più completo delle potenzialità e delle ottimizzazioni possibili.
Oltre ai limiti fisici della memoria, la capacità della CPU di gestire processi paralleli e la sua velocità di clock giocano un ruolo fondamentale. La sinergia tra CPU e memoria è essenziale per garantire prestazioni ottimali, specialmente in applicazioni che richiedono elaborazioni intensive e rapide risposte. Un'analisi approfondita delle interazioni tra questi componenti potrebbe offrire ulteriori spunti per ottimizzare le prestazioni del sistema in scenari reali.

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, noto per le sue capacità avanzate di gestione di grafi, si presentava come una scelta naturale per la manipolazione dei dati nel contesto considerato. Già dalla prima fase di import dei dati, è emersa chiaramente l'importanza di sfruttare a pieno le capacità di Cypher per definire query in grado di gestire efficacemente anche dataset di dimensioni reali.
Le diverse soluzioni implementate per il calcolo dei prefissi e dei case attivi per ciascun prefisso, hanno permesso di mettere a confronto la gestione di questi scenari direttamente in Neo4j piuttosto che tramite Python.
Nei test svolti, Python si è dimostrato più adatto per alcuni compiti specifici, come il calcolo dei prefissi su dataset di dimensioni reali. In queste situazioni, dato che Neo4j ha limitazioni imposte sull'utilizzo delle memoria, riesce a gestire i dati solo con tempi elevati, mentre Python sfrutta a pieno le risorse del calcolatore utilizzato. Tuttavia, è importante notare che in presenza di dataset di dimensioni molto grandi, Python potrebbe saturare l'utilizzo della

memoria, generando un'interruzione del processo.
Per quanto riguarda il calcolo dei case attivi per ogni prefisso, Python ha dimostrato di essere più performante, permettendo una corretta elaborazione, mentre Neo4j ha evidenziato limitazioni nella gestione di grandi quantità di dati, andando in out of memory.
Va sottolineato che il calcolo dei case attivi dipende fortemente dal dataset considerato. La dimensione di un dataset esercita un impatto diretto sulla precisione e sulla velocità dell'analisi: dataset più ampi richiedono risorse computazionali più robuste per gestire e analizzare i dati in modo efficiente. Tuttavia, non è solo la quantità di dati ad influenzare i risultati del calcolo dei case attivi, ma è anche la struttura e la complessità dei processi rappresentati. Un dataset con numerosi parallelismi, dove molteplici tracce possono essere eseguite simultaneamente, avrà un profilo di case attivi molto diverso rispetto ad uno con una struttura più sequenziale e lineare.

Un'ulteriore opzione potrebbe essere quella di adattare il codice Python per essere eseguito su Apache Spark. Spark è un framework progettato per il calcolo distribuito su cluster di computer, il che potrebbe superare i limiti di memoria e CPU incontrati nei test attuali. Utilizzando Spark, sarebbe possibile distribuire i carichi di lavoro su più nodi e sfruttare efficacemente le risorse disponibili, migliorando ulteriormente le prestazioni e consentendo l'elaborazione di volumi di dati ancora più grandi con efficienza.

Infine, un ulteriore sviluppo futuro riguarda l'utilizzo dei prefix-IG nel deep learning per il task di Next Activity Prediction. È interessante considerare le informazioni relative ai case attivi durante l'esecuzione di un prefisso. In particolare, invece di fornire come input alla rete solo il singolo prefisso, si possono includere anche dati sugli altri case attivi. Una possibilità è quella di fornire informazioni statistiche sui case attivi, oppure includere come input sia il prefisso che gli altri case attivi.

Bibliografia

ADRIANSYAH, A., VAN DONGEN, B. F. e VAN DER AALST, W. M. (2011), «Conformance checking using cost-based fitness analysis», in «2011 ieee 15th international enterprise distributed object computing conference», p. 55–64, IEEE. (Cited at page 6)

CHIORRINI, A., DIAMANTINI, C., GENGA, L. e POTENA, D. (2023), «Multi-perspective enriched instance graphs for next activity prediction through graph neural network», *Journal of Intelligent Information Systems*, vol. 61 (1), p. 5–25. (Cited at pages 6 e 34)

DIAMANTINI, C., GENGA, L., POTENA, D. e VAN DER AALST, W. (2016), «Building instance graphs for highly variable processes», *Expert Systems with Applications*, vol. 59, p. 101–118, URL `https://www.sciencedirect.com/science/article/pii/S0957417416301877`. (Cited at pages 6 e 13)

MÁRQUEZ-CHAMORRO, A. E., RESINAS, M. e RUIZ-CORTÉS, A. (2017), «Predictive monitoring of business processes: a survey», *IEEE Transactions on Services Computing*, vol. 11 (6), p. 962–977.

VAN DER AALST, W., ADRIANSYAH, A., DE MEDEIROS, A. K. A., ARCIERI, F., BAIER, T., BLICKLE, T., BOSE, J. C., VAN DEN BRAND, P., BRANDTJEN, R., BUIJS, J. e OTHERS (2012), «Process mining manifesto», in «Business Process Management Workshops: BPM 2011 International Workshops, Clermont-Ferrand, France, August 29, 2011, Revised Selected Papers, Part I 9», Springer.

VAN DONGEN, B. (2012), «BPI Challenge 2012», `https://doi.org/10.4121/uuid:3926db30-f712-4394-aebc-75976070e91f`. (Cited at page 13)

Ringraziamenti

Giunta al termine del mio percorso universitario, desidero esprimere la mia profonda gratitudine a tutte le persone che hanno reso possibile che ciò si avverasse.

Al mio relatore, prof. Domenico Potena, per il costante supporto, la fiducia accordatami e le opportunità offerte. La ringrazio per avermi guidato con pazienza durante questi mesi.

Ai miei genitori, il mio punto di riferimento. Fin da piccola mi avete insegnato a seguire il cuore e mi avete sempre permesso di inseguire i miei sogni. Un grazie a voi, per avermi permesso di arrivare fino a qui.
A mia madre, per la sua infinita pazienza e il suo costante sostegno. A te che sei sempre stata dalla mia parte e non hai mai smesso di fare il tifo per me affinchè riuscissi a raggiungere tutti i traguardi prefissati.
A mio padre, per avermi mostrato il valore del duro lavoro. A te che adori stuzzicarmi e infastidarmi ma che non smetti mai di trattarmi come una principessa.

A nonna Anna, che dopo ogni esame, ha aspettato una chiamata per sapere come era andato. A te che ami avere sempre l'ultima parola e che adori sapere tutto di tutti. Un grazie a te per l'affetto incondizionato che ogni giorno mostri verso noi nipoti.

A nonna Isolina, nonno Lamberto e nonno Nazzareno, i miei angeli custodi. Anche se oggi non potete essere qui con me, spero che mi guardiate e che siate orgogliosi di me e della persona che sono diventata.

Alle mie cugine, per essere state come delle sorelle maggiori, per avermi sempre aiutato e per essere rimaste anche ora che ognuna si sta costruendo il proprio futuro.
A Irene, l'anima creativa della famiglia, per aver sempre dimostrato di poter portare avanti le sue passioni, anche quando gli altri non erano d'accordo.
A Sofia, per la determinazione e l'impegno che mette nel realizzare i mille progetti a cui prende parte.
A Ester, per essere così grande ma così simile a me perchè dopo tutte le volte che mi hanno detto che siamo simili e che abbiamo la stessa mente da ingegnere, fermandomi a pensare, forse mi rivedo in te più di quanto avessi mai potuto immaginare.
A Lorenzo, per avermi fatto ricordare la bellezza delle piccole cose con la sua continua voglia di esplorare il mondo con i suoi occhi meravigliati.

A Giovanni, per la pazienza avuta in questi ultimi mesi dove non c'è stato tempo nemmeno per vedersi. A te che nonostante tutto sei rimasto al mio fianco, ricordandomi che non bisogna mai smettere di provare ad inseguire i propri sogni.

A Susanna, il mio porto sicuro, la persona su cui posso sempre fare affidamento perchè so che riuscirà a trovare un po' di tempo per me anche se è sommersa dalle cose da fare e vive così lontana. Alla nostra amicizia che ci lega ormai da anni e che è rimasta la stessa anche ora che non ci vediamo più tutti i giorni a scuola. Un grazie alla persona che so che farebbe qualsiasi cosa pur di difendermi e pur di strapparmi un sorriso nei giorni più tristi.

A Sara, per essere rimasta al mio fianco e alla nostra amicizia che rimane quella di un tempo anche se il tempo per vedersi non c'è mai. A noi, che passo dopo passo stiamo costruendo il nostro futuro e continuiamo ad essere orgogliose l'una dell'altra per come stanno procedendo le nostre vite.

Ai miei colleghi, per avermi accompagnato in questi anni di vita universitaria, sia nei momenti più felici che in quelli più tristi e stressanti.

Ad Arianna, per la sua capacità di portare avanti le sue idee con determinazione, lavorando senza sosta per raggiungere i suoi obiettivi.

A Federica, per la sua gentilezza, per il suo animo altruista e per la precisione e la cura con cui affronta ogni compito.

A Chiara, per aver messo me prima di sé stessa ed avermi permesso di essere qui oggi. Un grazie a te per aver scelto di lavorare con me, perchè entrambe sappiamo il supporto che ci siamo date a vicenda e sappiamo che se avessimo dovuto fare tutto quello che abbiamo fatto da sole, saremmo impazzite. Tutto questo mi ha fatto capire l'importanza di avere qualcuno vicino in grado di supportarti, quando tutto sembra impossibile. A tutte le volte che ci siamo dette che non saremmo mai riuscite a fare tutto, ma che alla fine, in qualche modo, siamo sempre riuscite a portare a termine. Non posso dimenticare le innumerevoli cene all'università, parte della nostra vita poco salutare dell'ultimo mese, che hanno reso questo percorso meno pesante e ricco di gossip. Spero di essere riuscita a farti capire quanto sono grata del tuo aiuto e spero molto di più, un giorno, di poterti ripagare per tutto questo.

Ad Alessandro, per aver preso a cuore il nostro lavoro e per aver scelto di sopportare due tirocinanti spesso in preda al panico. A tutte le volte in cui ci hai aspettato per pranzo nonostante la fame, a tutte le volte in cui ti abbiamo rubato del tempo quando avresti avuto sicuramente molto altro da fare. Un grazie a te perchè siamo piombate nella tua vita da dottorando, hai appeso un post-it sulla tua scrivania e hai deciso di affiancarci e supportarci.

Ai Gino, i miei amici, per avermi permesso di essere me stessa, con i miei pregi e i miei difetti. Per tutte le volte in cui, nonostante ci siamo organizzati all'ultimo minuto, il divertimento non è mai mancato. Grazie a voi che avete reso questo percorso meno pesante, permettendomi di staccare un po' dalle ansie e dalle preoccupazioni quotidiane.

Al nuoto, per avermi trasmesso l'importanza della costanza, della perseveranza e della dedizione nel perseguire un obiettivo. Ma molto di più per avermi insegnato che anche dopo le peggiori sconfitte, non bisogna darsi per vinti perché ci sarà presto un'altra occasione in cui dimostrare tutto quello di cui si è capaci, così come dopo ogni gara ci sarà una nuova

possibilità di salire sul blocco e mostrare agli altri le proprie capacità.

Ai miei compagni di squadra, per aver reso questi anni pieni di bei ricordi.
Ai miei colleghi istruttori e bagnini, che con il briciolo di pazzia che ci contraddistingue, hanno riempito le ore di lavoro di gioia e risate.
Ai miei allenatori, perché mi avete visto crescere ed è anche grazie a voi, se oggi sono diventata la persona che sono.

A Davide, per avermi trasmesso il suo amore per il nuoto. A tutte le volte che mi hai ricordato di credere nelle miei potenzialità e mi hai spinto ad alzare un po' di più l'asticella. Un grazie alla persona che è rimasta al mio fianco anche quando non era più il mio allenatore in prima linea e che, in ogni gara, è sempre stata con me dalla prima all'ultima vasca.