



Università degli Studi di Pisa

Dipartimento di Informatica

Corso di Laurea Triennale in Informatica

**Modellazione a Grafo dei Dati
Blockchain: Rappresentazione e
Analisi della Rete Bitcoin nel Graph
Database Neo4j**

Candidato:

Leonardo Arditti

Relatori/Relatrici:

Prof.ssa Laura Emilia Maria Ricci

Prof. Damiano Di Francesco Maesa

Anno Accademico 2023-2024

Indice

1	Introduzione	9
1.1	Lavori correlati	14
1.1.1	Database a grafo	14
1.1.2	Modellazione della blockchain di Bitcoin come grafo	16
2	Background	19
2.1	Blockchain e protocollo Bitcoin	19
2.1.1	DLT e blockchain	23
2.1.2	Il modello UTXO	25
2.1.3	Wallet e chiavi	28
2.1.4	Consenso e mining	32
2.2	Btree	36
3	Neo4j: concetti generali	39
3.1	Graph database nativi	39
3.2	Differenza tra Neo4j e DBMS relazionali	41
3.2.1	Confronto di una query tra Neo4j e un RDBMS	44
3.3	Il modello Labeled Property Graph dei graph database	45
3.4	Query su grafi: introduzione a Cypher	47
3.4.1	Struttura delle query	48
3.4.2	Rappresentazione dei nodi in Cypher	48
3.4.3	Rappresentazione delle relazioni in Cypher	49
3.4.4	Proprietà di nodi o relazioni	50
3.4.5	Clausole	50
3.5	Awesome Procedures On Cypher (APOC)	56
3.6	Indici e ottimizzazione delle query	59
3.6.1	RANGE	60
3.6.2	Composite	61
3.6.3	TEXT	61
3.6.4	POINT	62
3.6.5	Full-text	62

3.7	Memorizzazione dei dati in Neo4j	64
4	Blockchain di Bitcoin come grafo	69
4.1	Labeled Property Graph	70
4.2	Address-transaction graph	71
4.3	Payment graph	73
4.4	Address graph con valori aggregati	75
4.5	Considerazioni	77
5	Implementazioni in Neo4j	79
5.1	Address-transaction graph	80
5.2	Payment graph	82
5.3	Address graph con valori aggregati	84
5.4	Procedura per automatizzare l'importazione ed analisi dei grafi costruiti	87
6	Analisi definite sui grafi	89
6.1	Selezione degli indirizzi per le analisi	89
6.2	Indici definiti	92
6.3	Timestamp della prima transazione con indirizzo α come mittente	93
6.3.1	Address-transaction graph	93
6.3.2	Payment graph	94
6.3.3	Address graph con valori aggregati	95
6.4	Timestamp della prima transazione con indirizzo α come destinatario	96
6.4.1	Address-transaction graph	97
6.4.2	Payment graph	97
6.4.3	Address graph con valori aggregati	98
6.5	Dimensione dell'ego-network di un indirizzo	99
6.5.1	Address-transaction graph	100
6.5.2	Payment graph	101
6.5.3	Address graph con valori aggregati	102
7	Valutazione sperimentale	103
7.1	Evoluzione della dimensione del database e del tempo di importazione	109
7.1.1	Address-transaction graph	109
7.1.2	Payment graph	111
7.1.3	Address graph con valori aggregati	112
7.2	Evoluzione del numero di nodi e archi	114
7.2.1	Address-transaction graph	114

7.2.2	Payment graph	116
7.2.3	Address graph con valori aggregati	117
7.3	Valutazione dei tempi di esecuzione delle query	118
7.3.1	Address-transaction graph	118
7.3.2	Payment graph	119
7.3.3	Address graph con valori aggregati	120
7.4	Confronti complessivi	121
7.4.1	Dimensione del database e tempo di importazione	121
7.4.2	Numero di nodi e archi importati	124
7.4.3	Tempo di esecuzione delle query	126
7.5	Considerazioni	129
8	Caso d'uso: Address Taint Analysis	131
8.1	Implementazione in Neo4j	133
8.1.1	Address-transaction graph	133
8.1.2	Payment graph	142
8.1.3	Address graph con valori aggregati	148
8.2	Conclusioni	154
9	Conclusioni e lavori futuri	157
	Bibliografia	161

Ringraziamenti

Desidero dedicare questo piccolo spazio a chi affronta ogni giorno ostacoli e sfide ma che non per questo smette di perseguire i propri sogni, con la ferma convinzione che l'impegno e la dedizione portino inevitabilmente alla realizzazione di ciò che può sembrare irraggiungibile, sia per noi stessi che agli occhi altrui. Molte cose sembrano impossibili, fino a quando, un giorno, non lo sono più.

Questa tesi rappresenta il culmine di un viaggio impegnativo, tortuoso e segnato da momenti di difficoltà ma non per questo meno significativo e formativo. Riflettendo su questo percorso, ormai concluso, comprendo che senza le rinunce fatte e i sacrifici affrontati non avrei mai potuto valorizzare veramente questa esperienza. Perchè in effetti di questo si tratta: non solo di un traguardo accademico ma anche un importante percorso di crescita personale che mi ha dato modo di riflettere, di mettermi in discussione e di imparare a guardare il mondo con occhi diversi.

Ringrazio infinitamente la mia famiglia, il cui amore e supporto incondizionati sono stati il mio faro soprattutto nei momenti più ardui. Mamma, papà, Laura, Lorenzo, le parole non possono pienamente catturare la profondità della mia gratitudine ma spero che “*grazie di cuore, vi voglio molto bene*” possa avvicinarsi.

Non posso non ringraziare i miei amici, sia livornesi che dell'Università di Pisa, per la loro compagnia e per aver condiviso con me sia studi che momenti di svago. Siete stati voi a mostrarmi che la felicità risiede nelle piccole cose, nell'essere insieme e nel condividere momenti di gioia e di tristezza.

Infine, voglio ricordare Diego, non solo un collega ma un vero amico. Le esperienze condivise e i momenti passati insieme hanno un valore ancora più profondo ora che non ci sei più. Questa tesi vuole anche essere un tributo a te, in segno di ringraziamento per l'influenza positiva che hai avuto sulla mia vita.

Leonardo Arditti, 5 Aprile 2024

Capitolo 1

Introduzione

Il mondo delle criptovalute ha segnato una rivoluzione nel panorama finanziario globale. Bitcoin, la prima criptovaluta basata sulla tecnologia blockchain, ha catturato l'attenzione del pubblico di non specialisti per la sua capacità di garantire scambi di valore sicuri, decentralizzati ed anonimi. Ogni transazione in Bitcoin genera dati che sono pubblicamente accessibili e strettamente interconnessi, intrecciando indirizzi e transazioni in una vasta rete di flussi di valore. La comprensione profonda di tale sistema si fonda sull'analisi di questa intricata rete, necessitando di una rappresentazione dei dati che ne rispecchi l'intrinseca interconnessione.

Tuttavia, la percezione di anonimato e resistenza alla censura del protocollo Bitcoin ha sollevato preoccupazioni riguardo al suo uso in attività illecite come riciclaggio di denaro, finanziamento del terrorismo e altre forme di criminalità finanziaria. In questo contesto, la *address taint analysis* emerge come una tecnica chiave per la comprensione ed analisi della rete Bitcoin in supporto a tentativi di deanonymizzazione degli utenti coinvolti in attività illegali o sospette. Essa si basa sul concetto di “contaminazione” degli indirizzi Bitcoin, dove un indirizzo viene considerato contaminato se ha ricevuto valore direttamente da un indirizzo contaminato o se lo ha ricevuto da un indirizzo destinatario a sua volta di Bitcoin da un indirizzo contaminato. Questo approccio consente di tracciare il flusso di valore attraverso la rete Bitcoin, identificando i percorsi seguiti dalle monete e le relazioni tra gli utenti che le hanno scambiate. Attraverso l'applicazione in questa tesi della address taint analysis e della backward address taint analysis su tre diverse rappresentazioni della blockchain di Bitcoin come grafo si intende esplorare le caratteristiche di ciascun modello nella tracciabilità dei flussi di valore, realizzando una metodologia capace di essere applicata all'intera rete Bitcoin e di identificare eventuali attività sospette o illecite.

In questa tesi viene esplorata la modellazione di dati relativi alla blockchain di Bitcoin utilizzando il database a grafo Neo4j, uno strumento all'avanguardia per la rappresentazione e l'analisi di reti interconnesse. Neo4j è un graph database schema-free che fa utilizzo del modello *Labeled Property Graph*, un modello flessibile e intuitivo che permette di rappresentare le entità come nodi etichettati e le relazioni tra di esse come archi dotati di tipo, con la possibilità di arricchire la rappresentazione della realtà mediante proprietà espresse nel formato chiave-valore. Grazie all'utilizzo di Cypher, un linguaggio di interrogazione dichiarativo, è possibile eseguire interrogazioni su grafi con una sintassi semplice e espressiva, rendendo Neo4j uno strumento ideale per l'analisi di dati complessi e interconnessi come quelli della blockchain di Bitcoin. Infine, Neo4j offre strumenti di visualizzazione dei grafi integrati che consentono di esplorare e interpretare i dati in modo intuitivo, creando grafici interattivi e dinamici che facilitano la comprensione della struttura e delle relazioni all'interno del grafo di Bitcoin. Grazie a queste caratteristiche, Neo4j si è affermato come uno strumento potente e versatile per la modellazione e l'analisi dei dati della blockchain di Bitcoin, consentendo di esplorare in profondità la complessità della rete e di trarre intuizioni significative dalla sua struttura interconnessa.

Nella tesi vengono presentate tre diverse rappresentazioni della blockchain di Bitcoin come grafo, ciascuna con caratteristiche uniche che consentono di analizzare e comprendere diversi aspetti della rete delle transazioni di Bitcoin.

1. La prima rappresentazione è l'address-transaction graph, dove i nodi **Address** rappresentano gli indirizzi Bitcoin e i nodi **Transaction** rappresentano scambi di valore tra indirizzi. I nodi **Address** sono collegati ai nodi **Transaction** tramite archi **INPUT** nel caso in cui sia stato immesso valore nella transazione e tramite archi **OUTPUT** nel caso in cui sia stato ricevuto del valore della transazione. Questa rappresentazione permette di visualizzare la rete Bitcoin come un grafo bipartito, con nodi **Address** e nodi **Transaction** che rappresentano rispettivamente i partecipanti e le transazioni della rete, consentendo una chiara visualizzazione delle relazioni tra gli indirizzi e dei flussi di valore.
2. La seconda rappresentazione è il payment graph, dove i nodi **TXO** rappresentano gli output delle transazioni e le relazioni **CONTRIBUTES** indicano il contributo di un output a un input di una transazione successiva. Questo modello semplifica la struttura della rete eliminando i cicli presenti nell'address-transaction graph e consentendo una visualizzazione più chiara e diretta dei trasferimenti di valore. Inoltre, riflette in modo accurato la struttura temporale della blockchain, con ciascun output che

contribuisce solo alla creazione di output successivi, senza la possibilità di cicli nel grafo risultante.

3. Infine, la terza rappresentazione è l'address graph con valori aggregati, grafo dove gli indirizzi sono rappresentati come nodi **Address** e le proprietà aggregate degli archi **TRANSFERS_TO** riflettono la storia delle transazioni tra gli indirizzi, come il timestamp della prima e dell'ultima transazione e la proporzione di valore scambiato. Questo modello fornisce una visione sintetica e aggregata delle transazioni, consentendo di analizzare i flussi di valore e le relazioni tra gli indirizzi a un livello più alto di astrazione.

Attraverso un approccio sperimentale vengono proposti tre modelli dei dati Bitcoin sotto forma di grafo e si realizza una loro implementazione in Neo4j, valutando le risorse richieste e le prestazioni nella fase di importazione e interrogazione dei dati. Il risultato degli esperimenti mostra come la scelta della rappresentazione influenzi l'evoluzione della dimensione dei database, il tempo necessario per l'importazione dei dati e i tempi di risposta delle query, fornendo spunti critici sull'applicabilità ed efficienza dei modelli proposti. Dai risultati emerge che il modello address graph con valori aggregati risulta essere il più oneroso, in termini di spazio richiesto per la memorizzazione e di tempo necessario per l'importazione dei dati della blockchain di Bitcoin, a causa del recupero e della modifica frequente delle proprietà degli archi (ad esempio la proporzione di valore totale scambiata e il timestamp dell'ultima transazione), proprietà che devono essere aggiornate ad ogni nuova transazione che coinvolge una data coppia di indirizzi che si sono già scambiati valore in passato, offrendo comunque prestazioni accettabili e in linea con gli altri modelli in fase di interrogazione. I modelli address-transaction graph e payment graph, invece, si dimostrano più efficienti in fase di importazione portando a due database di dimensioni simili tra loro ma con tempi per l'importazione leggermente più lunghi per il payment graph a causa della necessità di creare un nuovo nodo **TXO** per ciascun output di transazione e di indicizzare i nodi creati al fine di garantire prestazioni ottimali in fase di interrogazione. Per valutare l'efficienza delle query di analisi su nodi scelti casualmente sono stati eseguiti test utilizzando diverse dimensioni del database e diversi tipi di query. I risultati hanno dimostrato che Neo4j è in grado di gestire efficacemente le query su nodi scelti casualmente, mantenendo tempi di esecuzione bassi (inferiori al secondo) anche con l'aumentare delle dimensioni del database. Questo è dovuto principalmente all'utilizzo di indici definiti su proprietà utilizzate nelle query che consentono un recupero rapido dei dati di interesse. Infine, è stata condotta un'analisi approfondita della address taint analysis su

ciascuna delle tre rappresentazioni come grafo della blockchain. Tutte e tre le rappresentazioni hanno dimostrato di essere in grado di tracciare il flusso di bitcoin provenienti da un indirizzo mittente ma con un maggiore dettaglio nell'address-transaction graph e nel payment graph dove è possibile tracciare i singoli scambi di valore. L'address graph con valori aggregati ha mostrato limitazioni nella capacità di fornire informazioni dettagliate sulle transazioni, poiché si basa su proprietà aggregate anziché sui singoli trasferimenti di valore.

Nel cuore di questa tesi giace l'ambizione di trascendere la mera rappresentazione numerica dei dati della blockchain di Bitcoin formalizzandoli invece in una serie di grafi che ne enfatizzano gli aspetti distintivi. Attraverso l'adozione del graph database Neo4j questo lavoro non solo illustra la potenzialità di una rappresentazione dati alternativa a quella tradizionale, che si basa su tabelle e relazioni, ma apre anche la porta a una gamma di applicazioni pratiche reali, dimostrando la versatilità e l'efficacia delle tecnologie basate su grafi.

L'approccio adottato non si limita a esplorare la blockchain attraverso lenti convenzionali ma si spinge oltre sfruttando le potenzialità dei database a grafo per analizzare e interpretare la rete Bitcoin, consentendo analisi di complessità variabile, partendo dalle più semplici per giungere a quelle più complesse e avanzate, con un'attenzione particolare rivolta alle implicazioni reali e all'applicabilità di tali analisi. In questo contesto, le tecniche di address taint analysis e backward address taint analysis emergono come esemplari applicazioni pratiche, offrendo strumenti potenti per il tracciamento di attività sospette, quali i flussi finanziari legati a ransomware, in modo più intuitivo e accessibile grazie alla natura intrinseca dei grafi che enfatizzano le relazioni e le interconnessioni.

La scelta di Neo4j come strumento principale per la modellazione e l'analisi dei dati della blockchain di Bitcoin si è rivelata vincente grazie alle sue capacità di gestire, analizzare e visualizzare dati complessi e interconnessi. Il suo utilizzo in questo contesto non solo evidenzia le sue prestazioni nella manipolazione di grandi set di dati, ma serve anche a dimostrare come i database a grafo possano essere impiegati in scenari investigativi, facilitando il lavoro degli analisti e potenziando la loro capacità di decifrare le dinamiche sottostanti alla rete Bitcoin.

La struttura della tesi è organizzata come segue:

- Il capitolo 2 fornisce un background essenziale sulla blockchain e sul protocollo Bitcoin, approfondendo aspetti basilari come il modello UTXO e il protocollo di consenso Proof-of-Work. Viene inoltre esaminata, seppur

brevemente, la struttura dati Btree utilizzata da Neo4j per l'indicizzazione, elemento fondamentale per garantire prestazioni ottimali in fase di interrogazione dei dati.

- Nel capitolo 3 viene introdotto il database a grafo Neo4j, focalizzando l'attenzione sui graph database nativi e il modello Labeled Property Graph che sarà poi adottato per le rappresentazioni della blockchain di Bitcoin come grafo nel capitolo successivo. Il capitolo procede poi con una panoramica di Cypher, il linguaggio di interrogazione di Neo4j, per concludere con una presentazione degli indici disponibili in Neo4j e una descrizione di come avvenga la memorizzazione dei dati in memoria.
- Il capitolo 4 presenta le tre rappresentazioni formali della blockchain di Bitcoin come grafo: l'Address-transaction graph, il Payment graph e l'Address graph con valori aggregati, ciascuna con la sua struttura e peculiarità per modellare i flussi di valore della rete Bitcoin.
- Nel capitolo 5 viene descritta la struttura del dataset della blockchain di Bitcoin ed il processo di implementazione in Neo4j delle rappresentazioni definite nel capitolo precedente, sfruttando la potenza espressiva di Cypher al fine di trasformare i dati testuali in grafo.
- Il capitolo 6 delinea una serie di query di analisi che verranno impiegate nei test sui diversi database, query progettate per valutare l'impatto dei diversi modelli dei dati sulle prestazioni di Neo4j.
- Nel capitolo 7 la valutazione sperimentale rivela come la scelta della rappresentazione influenzi l'evoluzione della dimensione dei database, il tempo necessario per l'importazione dei dati e i tempi di risposta delle query. Questa analisi empirica fornisce spunti critici sull'adattabilità e l'efficienza dei modelli proposti.
- Il capitolo 8 presenta un caso d'uso mostrando le tecniche di address taint analysis e backward address taint analysis e come queste possono essere implementate nelle tre rappresentazioni della blockchain di Bitcoin come grafo, evidenziando le differenze e le somiglianze tra i modelli nella tracciabilità dei flussi di valore.
- Infine, il capitolo 9 conclude il lavoro svolto riflettendo sulle evidenze emerse dall'analisi sperimentale e suggerendo possibili sviluppi futuri utilizzando database a grafo come Neo4j per l'analisi della blockchain di Bitcoin.

1.1 Lavori correlati

1.1.1 Database a grafo

Nel contesto dello sviluppo della tesi, che mira a esplorare le potenzialità del database a grafo Neo4j per modellare dati complessi e interconnessi quali quelli derivanti dalla blockchain di Bitcoin, è essenziale inquadrare il ruolo e le caratteristiche dei graph database all'interno del panorama tecnologico attuale. Questi sistemi di gestione dei dati si distinguono per la loro capacità di rappresentare le relazioni in modo esplicito e intuitivo attraverso strutture a grafo che mediante nodi e archi riflettono le interconnessioni tra le entità del dominio di interesse.

L'articolo [1] fornisce un'ampia panoramica sui database a grafo, mettendo in luce come questi sistemi di gestione dati siano particolarmente adatti alla rappresentazione e all'analisi di informazioni fortemente connesse. Tra i vari GDBMS (*Graph Database Management Systems*) disponibili che rappresentano lo stato dell'arte in questo campo, Neo4j emerge come uno dei più popolari e ampiamente utilizzati. La sua conformità alle proprietà ACID e la compatibilità con diversi linguaggi di programmazione ne fanno uno strumento versatile e robusto per l'analisi di dati complessi. Altri database a grafo popolari sono:

- ArangoDB:
 - base di dati orientata ai documenti, ogni nodo nel grafo è un documento JSON;
 - ArangoSearch, motore di indicizzazione, ricerca testuale e classificazione integrato nativamente, ottimizzato per velocità e memoria;
 - modellazione flessibile con la possibilità di modellare dati come coppie chiave-valore, documenti JSON o grafi;
 - schema-free o schema-full, a seconda delle esigenze per lavorare con dati testuali, strutturati, geo-spatiali o temporali;
 - AQL (ArangoDB Query Language) per interrogare il database, linguaggio dichiarativo simile a SQL ma con funzionalità specifiche per i grafi;
 - disponibile in una versione Community Edition gratuita e open-source, nonché in una Enterprise Edition commerciale con funzionalità aggiuntive;
 - scritto in C++ e supporta indici di diverso tipo per ottimizzare le prestazioni delle query.

- OrientDB
 - database multi-modello che supporta il modello a grafo, a documento e chiave-valore;
 - schema-free, schema-full o schema-mixed per adattarsi a diversi tipi di dati e applicazioni;
 - supporta query con Gremlin e SQL esteso per la navigazione dei grafi.
- Amazon Neptune
 - servizio di database a grafo completamente gestito da Amazon Web Services (AWS);
 - web service, proprietario;
 - supporta i modelli Labeled Property Graph e RDF per la modellazione dei dati;
 - compatibile con Apache TinkerPop Gremlin, SPARQL e OpenCypher per interrogare i dati.
- FlockDB
 - orientato principalmente alla gestione di grandi liste di adiacenza, senza supporto nativo per la navigazione o la traversata di grafi multi-hop;
 - non usa un linguaggio di query tradizionale per grafi; si concentra invece su operazioni su set efficienti;
 - altamente ottimizzato per operazioni specifiche, meno flessibile per applicazioni generali su grafi;
 - open source, progettato specificamente per esigenze di Twitter come la gestione di relazioni utente come "follows" o "likes".
- AllegroGraph
 - usa il modello RDF per la gestione di dati semanticamente ricchi e interconnessi;
 - supporta SPARQL, un linguaggio di query standardizzato per database RDF, consentendo interrogazioni complesse sui dati;
 - schema rigido data la conformità al modello RDF e al focus sul Semantic Web e Linked Data;

- proprietario, con un focus sull'integrazione di dati eterogenei e sulla realizzazione di inferenze logiche sui dati.
- GraphDB (Ontotext GraphDB)
 - modello dei dati basato su RDF, progettato per la gestione di Knowledge Graph e dati semantici;
 - utilizza SPARQL per l'esplorazione e l'interrogazione dei dati;
 - orientato agli standard RDF e OWL (Web Ontology Language), supporta modelli di dati complessi e ontologie per l'inferenza semantica;
 - disponibile in versioni con licenze sia open source sia proprietarie, a seconda delle esigenze di scalabilità e supporto.

FlockDB, ad esempio, è specializzato nella gestione di reti ampie ma poco profonde come quelle rappresentate dalle relazioni tra utenti su piattaforme social come Twitter. Si differenzia per il suo focus sulla rapidità delle operazioni su set piuttosto che sulla navigazione multi-livello del grafo, mostrando come i graph database possano essere ottimizzati per specifici casi d'uso. AllegroGraph e Ontotext GraphDB si distinguono invece per il loro orientamento verso la gestione di dati semanticamente ricchi, conformi agli standard del World Wide Web Consortium (W3C) per il Resource Description Framework (RDF), dimostrando la versatilità dei graph database nell'interpretare e analizzare dati strutturati secondo modelli ontologici complessi.

La tesi intende sottolineare l'importanza di questi sistemi nel trattamento di dati interconnessi e complessi, evidenziando come ciascun database a grafo abbia caratteristiche uniche che lo rendono più o meno adatto a specifici contesti e applicazioni. La scelta di Neo4j come piattaforma di riferimento per lo sviluppo dei modelli di rappresentazione della blockchain mira a sfruttare le sue avanzate capacità di gestione dei grafi, l'espressività del linguaggio Cypher e le sue potenzialità di scalabilità con l'obiettivo di valutare in modo approfondito e sperimentale la sua idoneità per esprimere e analizzare i dati della blockchain di Bitcoin a diversi livelli di dettaglio.

1.1.2 Modellazione della blockchain di Bitcoin come grafo

Questa sezione esplora i principali modelli a grafo proposti in letteratura [2] per rappresentare la blockchain di Bitcoin, evidenziando come ciascun modello

enfatizzi aspetti diversi della rete e faciliti l’analisi di specifiche caratteristiche della blockchain.

1. Address-transaction graph [3]: questo modello bipartito distingue chiaramente tra nodi che rappresentano gli indirizzi Bitcoin e nodi che rappresentano le transazioni. Gli archi orientati collegano gli indirizzi ai nodi delle transazioni per indicare l’invio di Bitcoin e le transazioni agli indirizzi per rappresentare la ricezione di Bitcoin. Questa rappresentazione consente di tracciare il flusso di valore tra gli indirizzi e di identificare i percorsi attraverso i quali i Bitcoin si muovono nella rete. La chiara distinzione tra partecipanti e transazioni rende l’Address-transaction graph particolarmente adatto per analisi che richiedono di comprendere le relazioni dirette tra indirizzi e movimenti di fondi;
2. Transaction graph: modello che rappresenta il flusso di Bitcoin tra le transazioni nel tempo. Ogni vertice all’interno del grafo simboleggia una transazione mentre ogni arco diretto tra i vertici rappresenta un output di transazione utilizzato come input in una transazione successiva;
3. Address graph: concentrandosi sul flusso di Bitcoin tra gli indirizzi questo modello visualizza le relazioni dirette tra gli indirizzi all’interno della rete. Ogni vertice rappresenta un indirizzo Bitcoin mentre gli archi diretti simboleggiano i valori delle transazioni che si muovono da un indirizzo sorgente a un indirizzo destinatario. Il punto di forza dell’Address graph risiede nella sua capacità di mappare le interazioni tra gli utenti della rete Bitcoin, offrendo una visione comprensiva delle relazioni di scambio e facilitando l’identificazione di pattern di comportamento, come l’accumulo di fondi o la distribuzione tra molteplici indirizzi;
4. Cluster graphs: una variazione dell’Address graph. I Cluster graphs aggregano indirizzi in cluster sulla base di euristiche specifiche. Questo modello è particolarmente utile per analizzare le strutture e le relazioni a livello di utente, raggruppando insieme gli indirizzi che presumibilmente appartengono alla stessa entità. Ad esempio, lo User graph è un grafo diretto in cui un utente o entità consiste in una collezione di indirizzi utilizzati in diverse transazioni. La creazione di un User graph attraverso la chiusura transitiva dell’insieme degli indirizzi coinvolti in tutte le transazioni, dopo aver collegato insieme le chiavi pubbliche utilizzate in input multipli, permette di visualizzare le reti di relazioni e transazioni effettuate da singoli utenti.

Nell’ambito della tesi verrà esplorato dalla letteratura il modello Address-transaction graph e verranno proposti due nuovi modelli per la rappresentazione della blockchain di Bitcoin come grafo:

1. Payment graph [4]: modello dove i nodi rappresentano output di transazioni e gli archi indicano il contributo di un output a un input di una transazione successiva, creando un grafo che elimina i cicli presenti nell’Address-transaction graph e permette una rappresentazione diretta e granulare del flusso di valore. Questo modello mette in luce la temporalità delle transazioni e il modo in cui i Bitcoin vengono spesi e trasferiti, fornendo una visione dettagliata della sequenza delle transazioni;
2. Address graph con valori aggregati: modello basato sul concetto di Address graph noto in letteratura ma arricchito con proprietà aggregate sugli archi che riflettono la storia delle transazioni tra gli indirizzi. Questa versione aggregata collassa le molteplici transazioni tra una coppia di indirizzi in un singolo arco con proprietà aggregate che riflettono la proporzione di valore scambiata, il timestamp, il blocco e l’identificativo della prima e ultima transazione in cui gli indirizzi sono coinvolti. Questo approccio semplifica l’analisi dei flussi di valore a lungo termine e delle relazioni stabilite tra indirizzi, rendendo il modello particolarmente accessibile per chi si avvicina allo studio della blockchain senza necessità di esaminare ogni singola transazione. Sebbene offra una visione ad alto livello e sintetica, l’Address graph con valori aggregati perde dettagli sulle transazioni individuali in favore di una maggiore leggibilità e semplificazione.

I modelli presentati precedentemente rappresentano diversi livelli di astrazione e focalizzazione all’interno dell’ecosistema Bitcoin, dal dettaglio granulare del flusso di transazioni nell’Address-transaction graph alla rete di scambi tra indirizzi nell’Address graph per poi arrivare alla visione aggregata degli utenti e delle entità nei Cluster graphs. Ciascun modello offre strumenti unici per l’analisi della blockchain di Bitcoin, consentendo agli investigatori, agli analisti e ai ricercatori di adattare il loro approccio in base agli obiettivi specifici di studio e alle domande di ricerca. La selezione del modello appropriato dipende dal tipo di analisi desiderata, dal livello di dettaglio richiesto e dalla natura delle informazioni che si intendono esplorare all’interno della vasta e complessa rete di transazioni Bitcoin.

Capitolo 2

Background

2.1 Blockchain e protocollo Bitcoin

Bitcoin rappresenta il primo e più noto esempio di criptovaluta (o più semplicemente *crypto*), ovvero una valuta digitale che utilizza la crittografia per garantire la sicurezza delle transazioni, la creazione di nuove unità e la verifica della trasferibilità delle stesse [5]. Nato come un esperimento di crittografia elettronica nel 2008 da un anonimo [6], o un gruppo di anonimi, sotto lo pseudonimo di Satoshi Nakamoto [7], Bitcoin è stato creato con l'obiettivo di creare un sistema di pagamento elettronico che non richiedesse la fiducia in un intermediario, come una banca, per lo scambio di valore tra due parti. Il protocollo Bitcoin è stato rilasciato nel 2009 come progetto open-source e da allora si è evoluto grazie al contributo di una comunità di sviluppatori che nel tempo ha portato a una serie di miglioramenti e aggiornamenti del protocollo originale.

Bitcoin è caratterizzato da una serie di proprietà che lo rendono unico rispetto alle valute tradizionali come il dollaro o l'euro (dette *fiat*). Tra queste, le principali sono la sua natura decentralizzata, la pseudoanonimità degli utenti coinvolti nelle transazioni e la sua offerta limitata di unità:

1. le valute tradizionali sono emesse e regolate da un'autorità centrale che funge da intermediario fidato tra le parti coinvolte in una transazione, che non necessariamente si fidano l'una dell'altra. Bitcoin, invece, è decentralizzato, ovvero non è controllato da un'autorità centrale, ma si basa su una rete di nodi che utilizzano un registro pubblico e condiviso, chiamato *blockchain*, per registrare e verificare le transazioni. Questo significa che non esiste un'unica autorità che possa controllare le transazioni effettuate con Bitcoin, rendendo il sistema più resistente a censura e controllo da parte di entità centrali. La natura distribuita di Bitcoin garantisce

inoltre che non esista un singolo punto di fallimento, rendendo il sistema più resistente a attacchi informatici e tentativi di manipolazione del registro delle transazioni grazie a un meccanismo di consenso tra i nodi della rete. In figura 2.1 e in figura 2.2 è possibile vedere un confronto tra un sistema di pagamento centralizzato e uno decentralizzato.

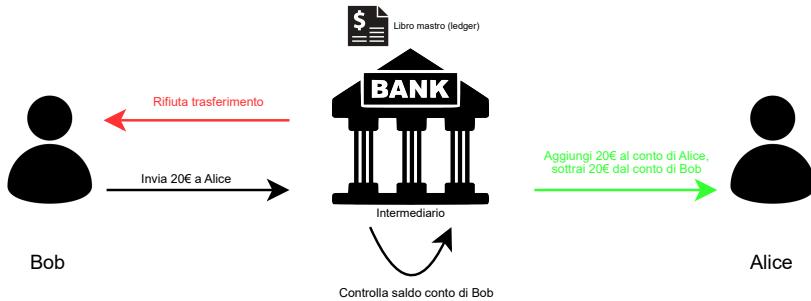


Figura 2.1: Esempio di sistema di pagamento centralizzato (in rosso il caso di credito insufficiente, in verde il caso di credito sufficiente).

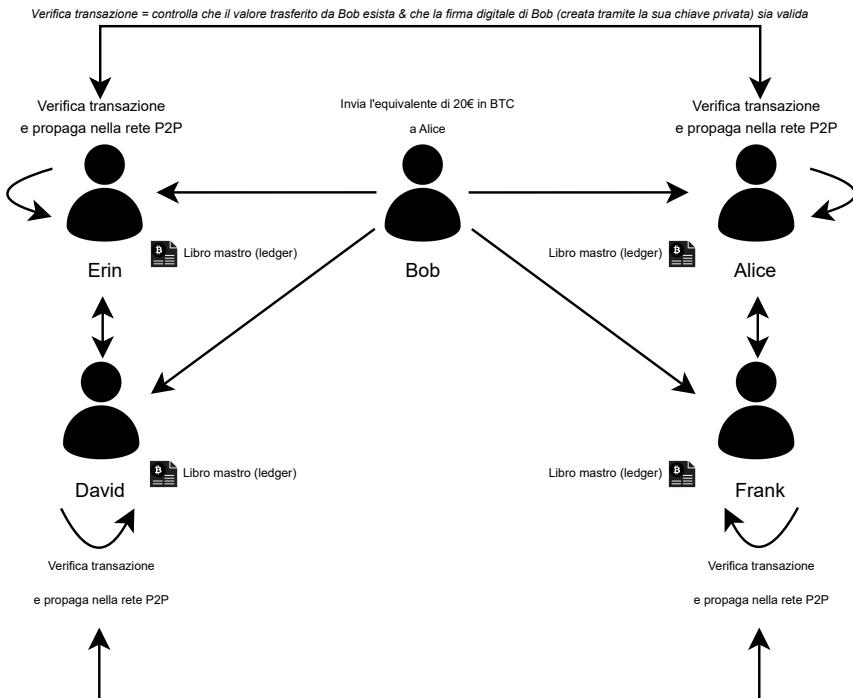


Figura 2.2: Esempio di sistema di pagamento decentralizzato.

2. gli scambi di valore effettuati con Bitcoin sono pseudoanonimi, ovvero non è possibile associare direttamente un utente coinvolto in una transazione (identificato da un hash alfanumerico della sua chiave pubblica) a una entità fisica come una persona o un'azienda. Tuttavia, data la natura pubblica del registro delle transazioni, è possibile tracciare il flusso di denaro scambiato tra gli indirizzi Bitcoin [8] e, utilizzando euristiche - come la *multi-input heuristic* [9] che associa a uno stesso utente tutti gli indirizzi che compaiono come input in una transazione, giustificata dal fatto che per poter spendere un output di una transazione è necessario conoscere la chiave privata associata all'indirizzo Bitcoin - e, combinando queste informazioni con altre fonti di dati (come ad esempio i social network o i forum online [10]) è possibile risalire alla reale identità che si cela dietro un indirizzo o un insieme di indirizzi Bitcoin. In figura 2.3 è mostrato il caso di Ross Ulbricht (a.k.a Dread Pirate Roberts/D-PR, altoid) [11] che nel forum BitcoinTalk ha lasciato tracce che hanno permesso di risalire alla sua identità e al suo coinvolgimento nel sito di e-commerce per acquisto/vendita di prodotti e servizi illegali Silk Road. In ordine dall'alto verso il basso: in figura 2.3a il post dove DPR chiede aiuto per un problema di programmazione PHP, lasciando però il suo indirizzo Bitcoin nel codice, in figura 2.3b il post di Ulbricht (eliminato ma ripreso in risposta da un utente) che invita gli utenti del forum a visitare Silk Road e infine in figura 2.3c il post di Ulbricht volto a reclutare programmatore per la sua start-up basata su Bitcoin (che si scoprirà essere Silk Road), esponendo il suo indirizzo email personale.
3. Bitcoin è caratterizzato da un'offerta limitata di unità, fissata dal suo creatore a un limite superiore di 21 milioni di Bitcoin. Questo significa che, a differenza delle valute tradizionali, non è possibile creare nuove unità di Bitcoin al di fuori di un processo di creazione di moneta chiamato *mining*, che prevede la risoluzione di complessi problemi matematici chiamati Proof-Of-Work (PoW) - che richiedono il calcolo di un hash crittografico che soddisfi determinate condizioni - da parte di nodi della rete, chiamati *miners* (o *minatori*), in cambio di una ricompensa in Bitcoin. Questo processo di creazione di moneta è regolato da un algoritmo che prevede una diminuzione della ricompensa per i minatori che risolvono i problemi matematici con il passare del tempo (un processo chiamato *halving*, che avviene approssimativamente ogni 4 anni). Il limite superiore 21 milioni è previsto per essere raggiunto intorno al 2140, una volta che questo limite sarà raggiunto i miners saranno ricompensati solo con le commissioni delle transazioni.

Author Topic: help with Bitcoin development in php (variable parameters) (Read 5958 times)

altoid (OP) Jr. Member #1

Activity: 48 Merit: 9

Hi all, I have run into some trouble using the bitcoin api with php. When I issue a command like:
`$bitcoin->sendfrom($userid, $receiving_address, $amount);`

I get an error like:
`fopen(http://...@localhost:8332/): failed to open stream: HTTP request failed! HTTP/1.1 500 Internal Server Error`

But when I hard code in the parameters:
`$bitcoin->sendfrom("1", "1LDNLreKJ6GawBHPgB5yfVLBERi8g3SbQS", 10);`

it works fine.

I did notice that I had to put quotes around the variables in my parameters for other functions to work. For example:
`$bitcoin->getnewaddress("$userid");`

But every combination of quotes or no quotes produces the error in the sendfrom function.

Thanks in advance for any help you can give. Let me know if you need more info too.

(a)

ShadowOfHarbrin Re: A Heroin Store #71

January 30, 2011, 08:09:37 PM

Quote from: Nefario on January 30, 2011, 06:30:07 AM
 (probably buried in the desert or in a forest)

Somehow, I'm seeing tremendous increase of popularity of forest sightseeing 😊

Quote from: altoid on January 29, 2011, 07:44:51 PM
 What an awesome thread! You guys have a ton of great ideas. Has anyone seen Silk Road yet? It's kind of like an anonymous amazon.com. I don't think they have heroin on there, but they are selling other stuff. They basically use bitcoin and tor to broker anonymous transactions. It's at <http://tdgdcyixpbu6uz.onion>. Those not familiar with Tor can go to silkroad420.wordpress.com for instructions on how to access the .onion site.

Let me know what you guys think

So here we go, first Bitcoin drug store.
 We're going into deep water faster than i thought then.

I wonder how long will it take for govs to start investigating Bitcoin.

(b)

Author Topic: IT pro needed for venture backed bitcoin startup (Read 38726 times)

altoid (OP) Jr. Member #1

Activity: 48 Merit: 6

IT pro needed for venture backed bitcoin startup
 Merited by taserz (3), Krubster (2)

Hello, sorry if there is another thread for this kind of post, but I couldn't find one. I'm looking for the best and brightest IT pro in the bitcoin community to be the lead developer in a venture backed bitcoin startup company. The ideal candidate would have at least several years of web application development experience, having built applications from the ground up. A solid understanding of oop and software architecture is a must. Experience in a start-up environment is a plus, or just being super hard working, self-motivated, and creative.

Compensation can be in the form of equity or a salary, or somewhere in-between.

If interested, please send your answers to the following questions to rossulbright@gmail.com

1) What are your qualifications for this position?
 2) What interests you about bitcoin?

From there, we can talk about things like compensation and references and I can answer your questions as well. Thanks in advance to any interested parties. If anyone knows another good place to recruit, I am all ears.

(c)

Figura 2.3: Il caso di Ross Ulbricht: deanonimizzare Bitcoin.

2.1.1 DLT e blockchain

Un concetto chiave per comprendere il funzionamento di Bitcoin è la *blockchain*, ovvero un registro pubblico e condiviso che contiene l'intera storia delle transazioni effettuate con Bitcoin. La blockchain è una forma di *distributed ledger technology* (DLT), ovvero una tecnologia che consiste in un database decentralizzato e condiviso tra i nodi di una rete. Ciascun nodo della rete mantiene una copia del registro (*ledger*) delle transazioni e collabora con gli altri nodi per garantire la coerenza del registro attraverso un meccanismo di consenso.

La blockchain Bitcoin, come è possibile intuire dal nome, è organizzata in una sequenza di blocchi, ciascuno dei quali contiene un insieme di informazioni relative alle transazioni contenute nel blocco stesso e altre informazioni necessarie per garantire l'integrità e l'immutabilità della blockchain. In particolare, ciascun blocco contiene [12]:

- una intestazione (*header*) contenente:
 - la versione del protocollo utilizzata per creare il blocco;
 - l'hash del blocco precedente;
 - l'hash del *merkle root* [13] delle transazioni contenute nel blocco;
 - un vincolo temporale (*timelock*) che limita la spendibilità dei Bitcoin contenuti nel blocco fino al raggiungimento di un determinato blocco o timestamp;
 - un timestamp in formato Unix;
 - un *target* che rappresenta la difficoltà del problema matematico da risolvere per creare il blocco;
 - un *nonce* che rappresenta il numero casuale utilizzato per risolvere il problema matematico nell'algoritmo di consenso PoW.
- un insieme di transazioni che definiscono il flusso di denaro tra degli indirizzi Bitcoin, ciascuna delle quali contiene:
 - un numero di versione;
 - un insieme di input, ciascuno dei quali contiene:
 - * l'hash di una transazione precedente;
 - * un indice che identifica l'output della transazione precedente;

- * una sequenza di istruzioni (dette script) che soddisfa le condizioni per spendere l'output della transazione precedente, spesso attraverso la verifica di una firma digitale che attesti la proprietà dell'output della transazione precedente.
- un insieme di output, ciascuno dei quali contiene:
 - * un valore in Bitcoin (espresso in satoshi, ovvero $\frac{1}{100\ 000\ 000}$ di Bitcoin);
 - * una sequenza di istruzioni (dette script) che definisce le condizioni per spendere l'output della transazione.

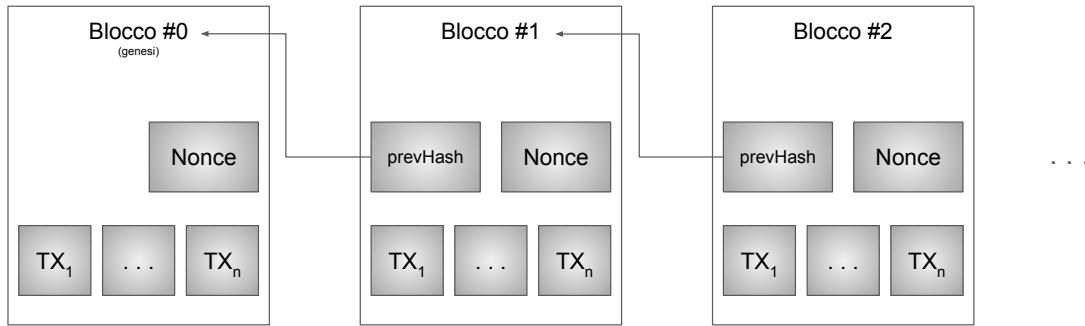


Figura 2.4: Organizzazione in blocchi della blockchain Bitcoin.

La struttura della blockchain di Bitcoin è illustrata in figura 2.4 ed è possibile notare come ciascun blocco sia collegato al blocco che lo precede attraverso l'hash del blocco precedente. Quest'ultimo aspetto è particolarmente importante perché fa sì che qualsiasi modifica apportata a un blocco comporti la modifica di tutti i blocchi successivi, rendendo impossibile alterare il registro delle transazioni senza che ciò sia rilevato dalla rete. Per modificare una transazione l'attaccante dovrebbe infatti disporre di una quantità di potenza di calcolo superiore a quella della rete (con un attacco chiamato *51% attack* [14]), in modo da poter ricalcolare gli hash crittografici dei blocchi successivi

e garantire che la sua versione della blockchain sia quella considerata valida dagli altri nodi.

Bitcoin transaction structure	
Field	Size
Version	4 bytes
Input counter	1-9 bytes (Varint)
Inputs*	Variable
Output counter	1-9 bytes (Varint)
Outputs*	Variable
Locktime	4 bytes

	Field	Size
Inputs*	Transaction hash	32 bytes
	Output index	4 bytes
	Unlocking-script size	1-9 bytes (Varint)
	Unlocking-script	Variable
	Sequence number	4 bytes

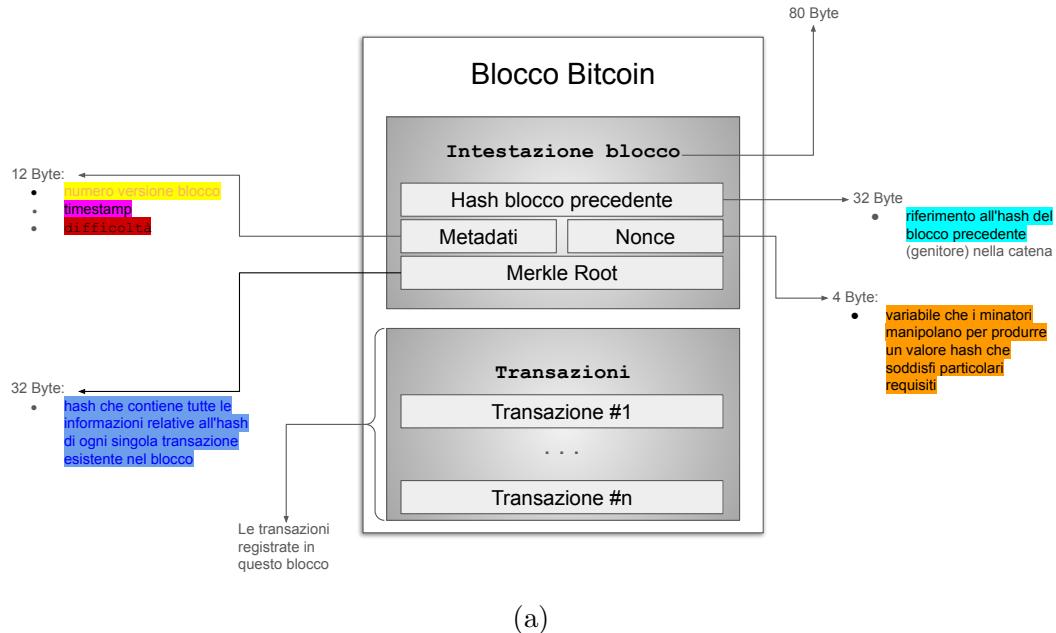
	Field	Size
Outputs*	Amount	8 bytes
	Locking-script size	1-9 bytes (Varint)
	Locking-script	Variable

Figura 2.6: Struttura di una transazione Bitcoin contenuta all'interno del blocco in figura 2.5a [15].

In figura 2.5a è possibile vedere la struttura di un blocco Bitcoin, i colori utilizzati riferiscono quelli in figura 2.5b dove è rappresentata l'intestazione del blocco #645536 minato il 27-08-2020 alle 09:43:07 della blockchain Bitcoin [16]. In figura 2.6 è riportata invece la struttura di una transazione Bitcoin con i campi chiave descritti nella sezione 2.1.1.

2.1.2 Il modello UTXO

La struttura delle transazioni di Bitcoin è basata su un modello chiamato *Unspent Transaction Output* (UTXO) che differisce da quello utilizzato dalle valute tradizionali. Mentre le valute tradizionali utilizzano un modello basato su account, in cui il saldo di un utente è definito come la differenza tra l'ammontare delle transazioni in entrata e in uscita, Bitcoin utilizza un modello che associa i Bitcoin posseduti da un utente a un insieme di output non spesi



```
0000002e5873757623f61cf57d122d3c18a877c8628f3193d2f90600000000000
0000000005c6b6c678a85005e91647f022798a27fd4bf5e07a877115ae7691373
de4f9e912b80475fea0710179052dc97
```

(b)

Figura 2.5: Struttura di un blocco in relazione all'intestazione del blocco 645536.

di transazioni. In particolare, ciascun output di transazione contiene un valore in Bitcoin e una sequenza di istruzioni (dette script) che definisce le condizioni per spendere l'output. Quando un utente intende effettuare una transazione, seleziona un insieme di output non spesi di transazioni precedenti e li utilizza come input per la nuova transazione in modo da definire il flusso di denaro tra gli indirizzi Bitcoin coinvolti nella transazione. L'output della transazione precedente viene quindi segnato come speso e non può essere utilizzato in transazioni future. Se la quantità di Bitcoin utilizzata come input per la transazione è maggiore della quantità richiesta come output allora la differenza tra le due quantità prende il nome di *fee* e rappresenta la ricompensa per il miner che inserisce la transazione nel blocco. Nel modello UTXO quando si spende un output non precedentemente speso è necessario utilizzare l'intero valore dell'output, una differenza rispetto al modello basato su account in cui è possibile spendere una frazione del saldo disponibile.

I vantaggi del modello UTXO rispetto al modello tradizionale basato su account sono diversi, tra cui:

- facilità di verifica delle transazioni [17] — in quanto è sufficiente per i nodi della rete verificare che l'output utilizzato come input per una transazione non sia già stato speso in una transazione precedente, senza dover verificare l'intera storia delle transazioni di un utente per garantire che abbia i fondi necessari per effettuare una transazione;
- scalabilità [18] — in quanto il modello UTXO permette di processare molteplici UTXO in parallelo e quindi di aumentare la velocità di elaborazione delle transazioni;
- tracciabilità [19] — ogni UTXO ha un identificativo univoco che consente di tracciare la storia delle transazioni e di verificare l'intera ciclo di vita di un UTXO.

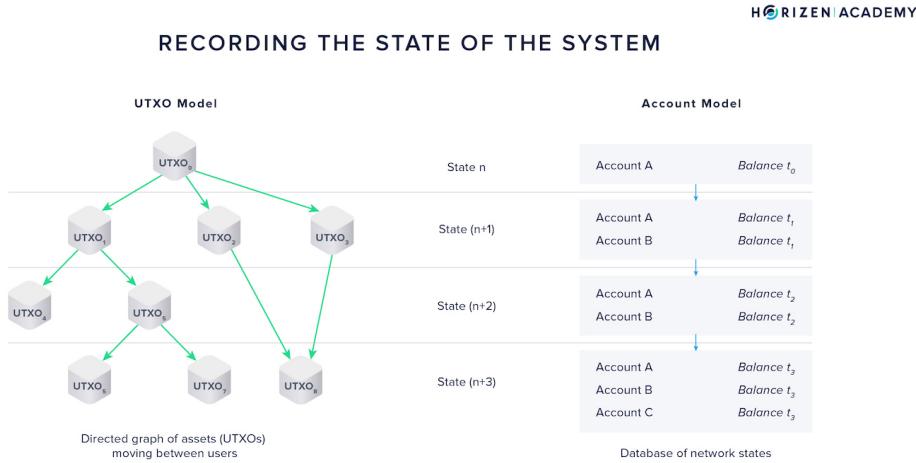


Figura 2.7: Confronto tra il modello UTXO e il modello basato su account [20].

L’immagine 2.7 mette a confronto due modelli per registrare lo stato dei fondi in un sistema di pagamento: il modello UTXO e il modello basato su account. A sinistra è rappresentato il modello UTXO utilizzato da Bitcoin, dove si ha un grafo diretto aciclico di asset (UTXOs) che si spostano tra gli indirizzi degli utenti. Ogni UTXO rappresenta un’uscita di una transazione non spesa che può essere utilizzata come input per una nuova transazione. Questo modello è come un flusso continuo di monete specifiche che si spostano da un proprietario all’altro. A destra, il modello conto corrente (Account Model), mostra una serie di stati della rete che rappresentano i bilanci degli account in momenti diversi, contrassegnati come t_0, t_1, t_2, t_3 . Ogni stato registra il saldo totale degli account, piuttosto che tracciare monete individuali. Questo è simile al modo in cui le banche registrano il saldo dei conti dei loro clienti, aggiornando semplicemente il saldo quando i fondi vengono depositati o prelevati.

2.1.3 Wallet e chiavi

Per utilizzare Bitcoin è necessario disporre di un *wallet*, ovvero un’applicazione o dispositivo che consente di generare, gestire e utilizzare le chiavi necessarie per firmare e verificare le transazioni. A differenza delle carte di credito o dei conti correnti, in cui l’accesso ai fondi è garantito da una password o da un PIN condiviso con l’istituto finanziario, Bitcoin utilizza un sistema a crittografia asimmetrica [21] in cui ciascun utente dispone di una coppia di chiavi, una privata e una pubblica, per firmare e verificare le transazioni associate a

un indirizzo Bitcoin.

È importante notare che a differenza della rete Bitcoin che è pubblica e distribuita, un wallet Bitcoin è privato e controllato solamente dal suo proprietario [22].

Un wallet Bitcoin è composto da [23]:

- un insieme di chiavi private¹, dove ciascuna chiave è una stringa alfanumerica di 256 bit generata casualmente e utilizzata per firmare le transazioni di un determinato indirizzo Bitcoin. Questa è la parte più critica di un wallet Bitcoin. La chiave privata è ciò che un utente usa per accedere e autorizzare le transazioni in uscita dalla wallet ed è essenziale tenere questa chiave in sicurezza poiché chiunque la possiede ha il controllo sui Bitcoin collegati a quella chiave;
- un insieme di chiavi pubbliche², ciascuna corrispondente a una chiave privata e associata a un indirizzo Bitcoin consentendo di verificarne la firma delle transazioni;
- un insieme di indirizzi Bitcoin, ciascuno dei quali è associato a una chiave pubblica e consente di ricevere Bitcoin.

¹continuando l'analogia con le carte di credito gestite da un istituto finanziario: le chiavi private possono essere considerate come le chiavi di accesso (o PIN/password) a un conto corrente in quanto consentono di accedere ai fondi associati a un indirizzo Bitcoin.

²ancora, le chiavi pubbliche possono essere considerate come il numero di conto corrente associato a una carta di credito in quanto consentono di ricevere Bitcoin e di verificare la firma delle transazioni.

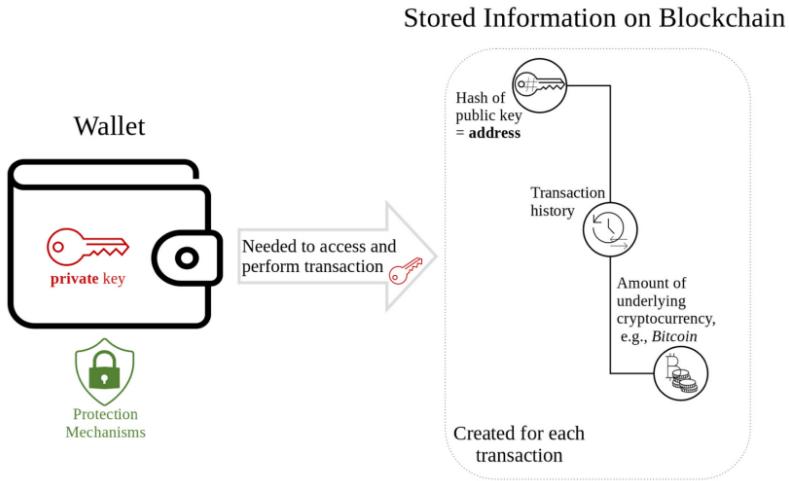


Figura 2.8: Spiegazione semplificata del ruolo dei wallet nelle transazioni Bitcoin [24].

Nell’immagine 2.8 il wallet agisce come un’interfaccia per l’utente nel gestire le chiavi private necessarie per accedere e autorizzare transazioni Bitcoin in uscita, che vengono poi trasmesse alla rete Bitcoin per essere validate e inserite in un blocco della blockchain. Il wallet contiene meccanismi di protezione per mantenere la chiave privata al sicuro. Nella blockchain sono memorizzati: l’hash della chiave pubblica che funge da indirizzo, la cronologia delle transazioni e l’ammontare di criptovaluta posseduto che viene speso, in questo caso Bitcoin.

I crypto wallet sono conosciuti anche come Self-Custody wallet in quanto l’utente è pienamente responsabile della custodia delle chiavi private e quindi dei fondi associati. Al contrario, quando si utilizza un wallet custodito da altri (come ad esempio un Exchange centralizzato - o CEX - come Coinbase o Binance) l’utente affida la custodia delle chiavi private e quindi dei fondi associati a un terzo, che si assume la responsabilità della sicurezza delle chiavi private e della loro gestione. Se da un lato l’utilizzo di un wallet custodito da un terzo consente di semplificare la gestione delle chiavi private e di poter ottenere assistenza in caso di smarrimento delle chiavi, dall’altro comporta il rischio di perdere il controllo dei fondi [25] in caso di furto (ad esempio a seguito di un attacco informatico all’Exchange, come avvenuto in passato con Mt. Gox [26] con un furto di circa 740 000 BTC, all’epoca circa 460 milioni di euro — il 6% del totale di Bitcoin in circolazione [27] — NiceHash [26], YouBit [26] e Coincheck [28]), blocco dell’account o chiusura del servizio [29]. Da qui il detto “Not your keys, not your coins” [30] che sottolinea l’importanza di

conservare le chiavi private in un luogo sicuro e di non affidare la custodia dei fondi a terzi non fidati.

I tipi di wallet si distinguono in base al livello di sicurezza e controllo che l'utente desidera avere sui propri fondi, una distinzione può essere la seguente [31]:

- Wallet *hot* - sono wallet connessi a Internet e quindi esposti a rischi di sicurezza come attacchi informatici o furto. Sono adatti per piccole quantità di Bitcoin e per transazioni frequenti, ma non sono adatti per conservare grandi quantità di Bitcoin a lungo termine
- Wallet *cold* - sono wallet non connessi a Internet e quindi meno esposti a rischi di sicurezza. Sono adatti per conservare grandi quantità di Bitcoin a lungo termine, ma non sono adatti per transazioni frequenti
- Wallet *hardware* - sono wallet fisici che memorizzano le chiavi private in un dispositivo hardware appositamente progettato per garantire la sicurezza delle chiavi private. Sono adatti per conservare grandi quantità di Bitcoin a lungo termine e per transazioni frequenti, ma sono esposti a rischi di danni fisici o smarrimento
- Wallet *software* - sono wallet che memorizzano le chiavi private in un dispositivo software, come ad esempio un'applicazione per smartphone o un'applicazione per desktop. Offre un buon compromesso tra sicurezza e facilità d'uso, ma sono esposti a rischi di sicurezza come attacchi informatici o furto
- Wallet *cartacei* - le chiavi private sono memorizzate su un supporto cartaceo come ad esempio un foglio di carta. Questa opzione è l'unica che consente di conservare le chiavi private in modo completamente offline, ma è anche la più vulnerabile a danni fisici o smarrimento
- Wallet *web* - sono wallet che memorizzano le chiavi private su un server remoto e sono accessibili tramite un browser web

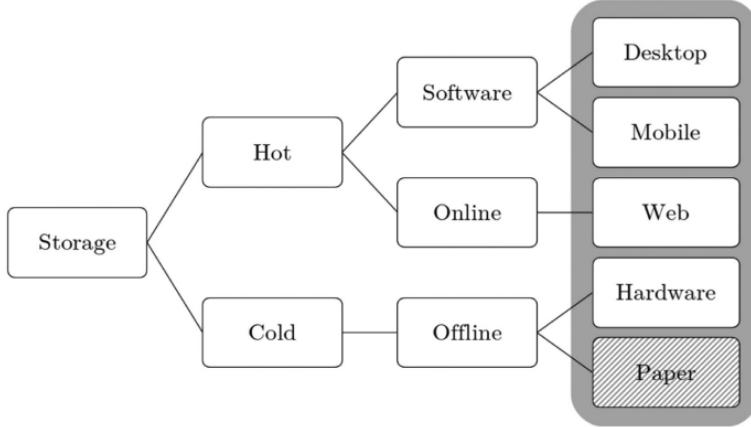


Figura 2.9: Diversi tipi di wallet Bitcoin [31].

2.1.4 Consenso e mining

Bitcoin utilizza un algoritmo di consenso chiamato *Proof-Of-Work* (PoW) per assicurare che tutti i nodi partecipanti concordino sullo stato del registro delle transazioni. L'idea principale che si cela dietro questo algoritmo è di combinare la potenza di calcolo e la crittografia per creare il consenso sull'autenticità dei dati registrati nella blockchain. In particolare, l'algoritmo PoW richiede ai miners di utilizzare la loro potenza di calcolo per risolvere una One-Way Function (OWF) [32], cioè una funzione che prende in input un valore di lunghezza arbitraria e restituisce un valore di lunghezza fissa, che è difficile da calcolare ma facile da verificare. In Bitcoin l'OWF utilizzata è la funzione di hash crittografico SHA-256 [33], che prende in input un blocco di dati e restituisce un hash di 256 bit. Il problema matematico che i miners devono risolvere consiste nel trovare un valore di *nonce* tale che l'hash del blocco, il cui contenuto è concatenato con il nonce, soddisfi determinate condizioni, chiamate *target*, che definiscono la difficoltà del problema matematico. In Bitcoin il target è dato dal numero di zeri iniziali dell'hash del blocco, dove un numero maggiore di zeri indica una difficoltà maggiore del problema da risolvere per i miners. Tra i miners che partecipano alla risoluzione, il primo che riesce a trovare una soluzione valida è ricompensato con una quantità di Bitcoin e con le commissioni delle transazioni contenute nel blocco. Successivamente, il blocco validato viene aggiunto alla blockchain e viene trasmesso alla rete di miners per verificare la validità del blocco e delle transazioni contenute. Il blocco viene considerato valido solo se la maggioranza dei miners concorda sulla sua validità, in caso contrario il blocco viene scartato e le transazioni contenute vengono reinserite

nella *mempool* (una sorta di “coda” di transazioni in attesa di essere inserite in un blocco) per essere processate in un blocco successivo. Nel raro caso in cui due blocchi validi vengano creati contemporaneamente, la rete viene temporaneamente divisa in due diramazioni (o *fork*) e quando uno dei due rami della blockchain diventa più lungo dell’altro la rete converge su quella diramazione facendo sì che il ramo più lungo diventi il principale della blockchain [34].

In figura 2.10 viene illustrato il meccanismo di Proof of Work (PoW) che è essenziale per il funzionamento della rete Bitcoin. Un blocco proposto, che si trova nella parte superiore dell’immagine, deve essere convalidato prima di essere aggiunto alla blockchain. La convalida avviene tramite la creazione di un hash, un codice unico derivante dall’algoritmo SHA-256, che cifra il contenuto del blocco insieme a un numero casuale detto nonce. La parte centrale dell’immagine mostra il processo iterativo di hashing. I minatori modificano ripetutamente il nonce per generare hash diversi finché uno di essi non risulta essere inferiore a un valore target predefinito, il che dimostra che è stato eseguito un lavoro computazionale significativo. Questo valore target rappresenta la difficoltà corrente della rete e serve a mantenere un tasso di aggiunta di nuovi blocchi costante. Quando il minatore trova un hash che soddisfa la condizione del valore target, come illustrato dal segno di spunta verde, il blocco viene approvato e poi viene aggiunto alla blockchain esistente. Infine, il minatore che ha risolto il blocco riceve una ricompensa in Bitcoin, indicata nell’immagine come “Block Reward”. Questa ricompensa serve da incentivo economico per i minatori a partecipare nel processo di mining e sostiene la crescita e la sicurezza della rete Bitcoin.

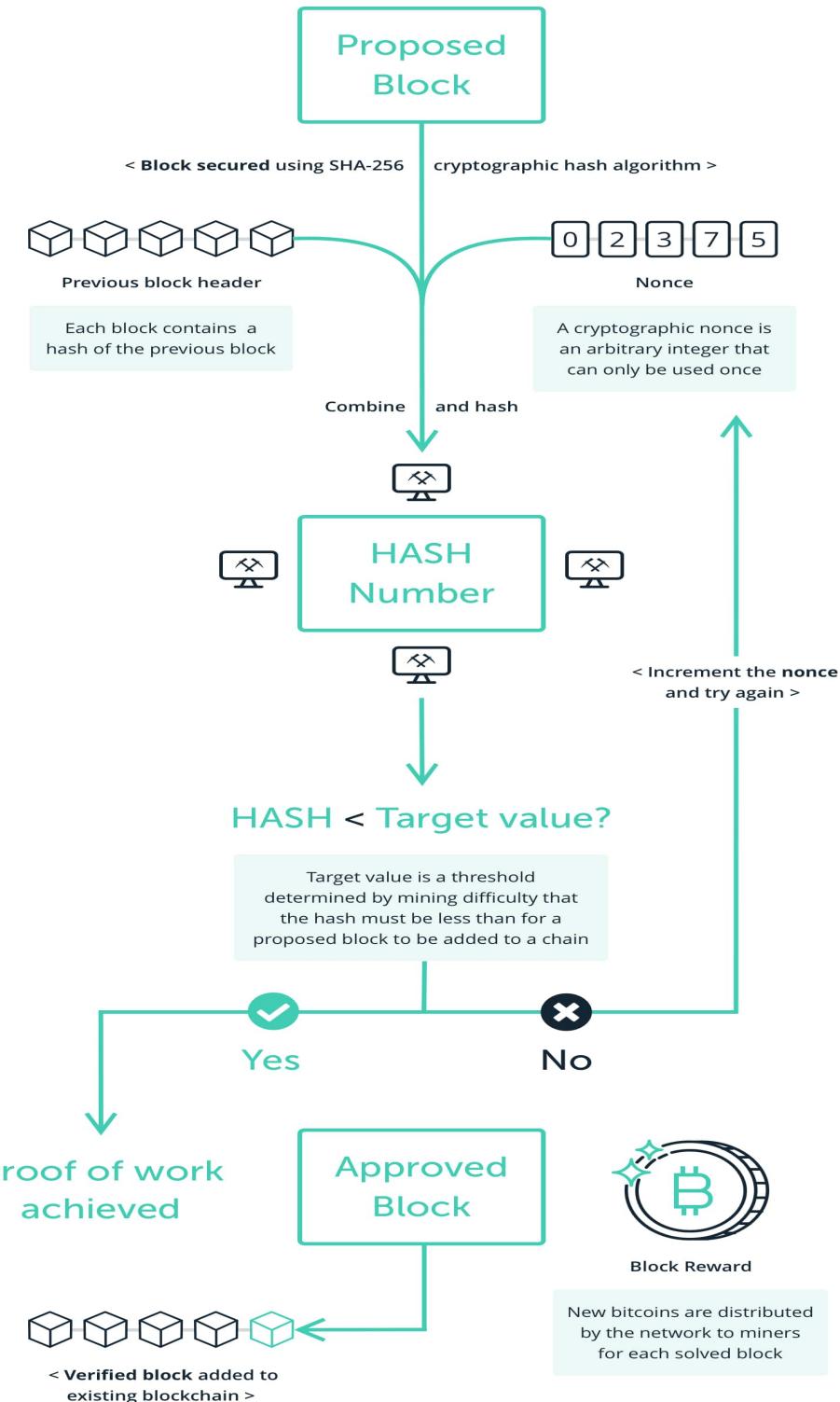


Figura 2.10: Descrizione del meccanismo di consenso PoW [35].

Per i miners il ruolo della potenza di calcolo è cruciale, in quanto maggiore è la potenza di calcolo a disposizione di un miner (misurata in *hashes per secondo* o *hashrate*), maggiore è la probabilità che un miner riesca a risolvere il problema matematico e a validare un blocco [36]. Tuttavia, la difficoltà del problema matematico è regolata automaticamente dal protocollo Bitcoin in modo da garantire che un blocco venga validato approssimativamente ogni 10 minuti. Questo meccanismo di regolazione della difficoltà, chiamato *difficulty adjustment*, prevede che la difficoltà computazionale venga regolata ogni 2016 blocchi (circa ogni 2 settimane) in base al tempo medio impiegato per validare i blocchi precedenti. Se il tempo medio è inferiore a 10 minuti, la difficoltà del problema matematico viene aumentata, se il tempo medio è superiore a 10 minuti, la difficoltà del problema matematico viene diminuita.

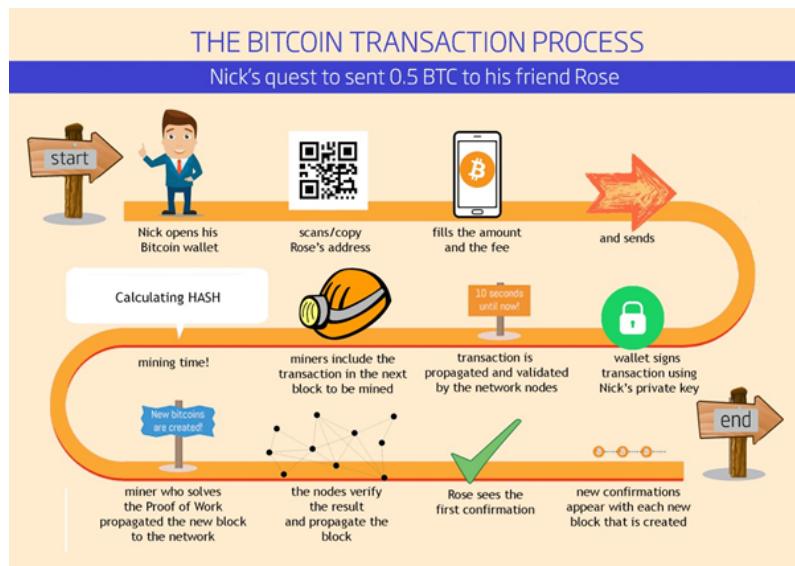


Figura 2.11: Esempio semplificato del ciclo di vita di una transazione Bitcoin da 0.5 BTC fatta da un utente Nick a un utente Rose [37].

In figura 2.11 viene presentato il processo di una transazione Bitcoin, delineando le fasi attraverso cui Nick invia 0.5 BTC all'utente Rose. Il processo inizia con l'apertura del suo portafoglio Bitcoin e la scansione dell'indirizzo di Rose, che fungerà da destinazione per la transazione. Nick inserisce quindi l'importo desiderato e la commissione di transazione, che è essenziale per incentivare i minatori a processare la sua transazione. Dopo aver inviato la transazione, questa viene firmata digitalmente utilizzando la chiave privata di Nick, un passaggio critico che assicura che solo il proprietario del wallet possa autorizzare spostamenti di fondi. La transazione è ora in attesa nella pool

di transazioni non confermate (mempool), dove i minatori, attraverso un processo competitivo noto come mining, tentano di includere la transazione nel prossimo blocco della blockchain.

Il minatore che per primo ha la potenza di calcolo necessaria per risolvere il puzzle crittografico dell'algoritmo PoW annuncia alla rete l'aggiunta di un nuovo blocco. Gli altri nodi verificano l'accuratezza del blocco proposto e, se confermato, lo aggiungono alla loro copia della blockchain, rendendo la transazione di Nick a Rose confermata una volta. Con ogni blocco successivo che viene aggiunto la transazione riceve conferme aggiuntive aumentando la sicurezza e rendendo praticamente impossibile una doppia spesa. Infine, Rose vede la transazione riflessa nel suo wallet, segnando il completamento del processo di trasferimento dei Bitcoin da Nick a lei.

2.2 Btree

All'interno di Neo4j, un database orientato ai grafi che verrà approfondito successivamente nel capitolo 3, la struttura dati B-tree è adoperata per gestire gli indici al fine di migliorare le prestazioni nell'accesso al disco. Un B-tree è una struttura dati auto-bilanciata che mantiene i dati ordinati e consente ricerche, inserimenti e cancellazioni in tempo logaritmico.

Un B-tree di ordine m è un albero che soddisfa le seguenti proprietà [38]:

- ogni nodo interno ha al più m figli;
- ogni nodo, ad eccezione della radice e delle foglie, ha almeno $\lceil m/2 \rceil$ figli;
- la radice ha almeno due figli se non è una foglia;
- tutte le foglie si trovano allo stesso livello;
- un nodo interno con k figli contiene $k - 1$ chiavi.

Un B-tree si distingue per alcune caratteristiche specifiche [39]. L'albero è bilanciato in modo che ogni foglia sia alla stessa distanza dalla radice. Questo bilanciamento è fondamentale per assicurare che le operazioni di ricerca abbiano una complessità logaritmica, ovvero $O(\log n)$, rendendo la struttura particolarmente efficiente per database di grandi dimensioni (sia a grafo come Neo4j, sia relazionali come Oracle Berkeley DB e MySQL [40]) dove n rappresenta il numero totale di chiavi presenti nell'albero. Tutti i dati sono memorizzati nei nodi foglia in modo ordinato, e ogni nodo interno funge da indice che guida il processo di ricerca verso la corretta posizione dei dati nel

nodo foglia. L'algoritmo del B-tree si assicura che i nodi siano riempiti almeno alla metà, eccetto la radice che deve avere almeno due figli se non è un nodo foglia. Ciò evita un uso inefficiente dello spazio e ottimizza il numero di confronti richiesti durante le operazioni di ricerca, inserimento e cancellazione.

Quando un nodo raggiunge la capacità massima e un nuovo elemento deve essere inserito si verifica una divisione del nodo: il nodo pieno si divide in due nodi e promuove il valore mediano al nodo genitore. Questo processo può propagarsi verso l'alto fino alla radice, mantenendo l'albero bilanciato. Se la radice si divide, la sua altezza aumenta di uno garantendo che la profondità dell'albero cresca in maniera logaritmica rispetto al numero di elementi.

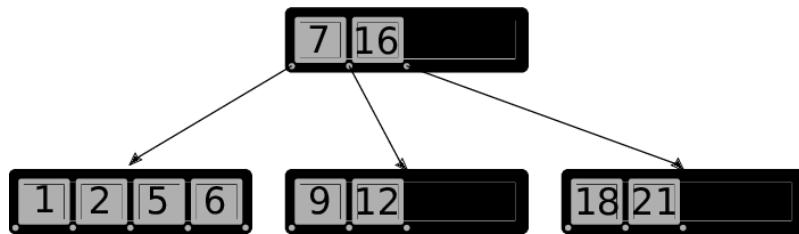


Figura 2.12: Esempio di B-tree con valori interi [39].

Capitolo 3

Neo4j: concetti generali

Neo4j è un *graph database management system* (GDBMS o più semplicemente *graph database*) open-source e NoSQL sviluppato da Neo Technology, Inc. che utilizza il modello a grafo per rappresentare i dati e le relazioni tra essi, supportando transazioni compatibili con il modello ACID [41] e offrendo contemporaneamente un’archiviazione e un’elaborazione nativa delle strutture a grafo. Rilasciato per la prima volta nel 2007, è scritto in Java e Scala e utilizza Cypher, un linguaggio di query dichiarativo basato su pattern, per interrogare i dati. Neo4j supporta inoltre diversi driver per i linguaggi di programmazione più diffusi (e.g. Java, JavaScript, Go, Python, ...) al fine di facilitare l’integrazione con le applicazioni.

3.1 Graph database nativi

I graph database nativi, che aderiscono al principio “Graph First” [42], sono ottimizzati per gestire dati secondo il modello a grafo sia in termini di memorizzazione che di elaborazione. Questa specializzazione consente operazioni di navigazione efficienti attraverso catene di relazioni rappresentando una netta evoluzione rispetto ai database non nativi. I sistemi non nativi, spesso basati su tecnologie preesistenti (anche dette legacy), tendono ad aggiungere una struttura a grafo come strato supplementare senza ottimizzazioni sottostanti (ad esempio utilizzano un backend basato su strutture tabellari come mySQL) andando a inferire relazioni tra entità usando meccanismi come le chiavi esterne e le join, fornendo prestazioni non soddisfacenti durante l’elaborazione di query complesse e la navigazione delle relazioni.

L'elemento distintivo dei graph database nativi è l'adozione della *index-free adjacency*, che consente a ciascun nodo di mantenere un riferimento diretto ai nodi adiacenti, un approccio che consente l'attraversamento¹ di dati correlati senza il costo aggiuntivo delle ricerche con indice per ogni passaggio tra le relazioni. Ciò riduce significativamente la complessità computazionale garantendo una complessità costante di $O(1)$ per l'accesso ai nodi adiacenti, a differenza della complessità logaritmica $O(\log(n))$ associata all'utilizzo degli indici nei database non nativi.

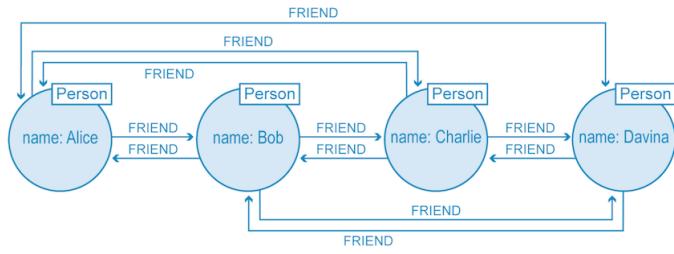


Figura 3.1: Esempio di rappresentazione nativa di un grafo relativo a una rete sociale [43].

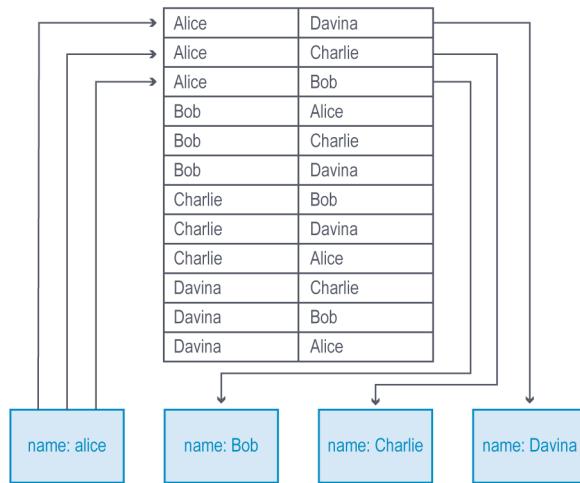


Figura 3.2: Esempio di rappresentazione non nativa di un grafo relativo a una rete sociale [43].

¹con attraversamento (dall'inglese *traversal*) si intende la navigazione di una struttura a grafo a partire da un nodo attraverso le relazioni che lo collegano ad altri nodi secondo un criterio specifico.

L’importanza dell’index-free adjacency nei graph database nativi è ulteriormente enfatizzata dalla diretta contrapposizione con i database non nativi, come mostrato nelle figure 3.1 e 3.2. Mentre i graph database nativi permettono interrogazioni dirette senza la necessità di ulteriori ricerche di indice, i database non nativi richiedono una gestione più complessa dei dati su nodi e relazioni. Come è possibile osservare in figura 3.1, per trovare gli amici di Alice in un graph database nativo basta seguire le sue relazioni di tipo FRIEND in uscita con un costo $O(1)$ per ciascuna. Per trovare invece chi è amico di Alice è sufficiente seguire tutte le relazioni di tipo FRIEND in entrata di Alice fino alla loro origine, sempre al costo $O(1)$ per ciascuna. Al contrario, in un graph database non nativo come quello in figura 3.2, i dati su nodi e relazioni sono organizzati in tabelle o liste. Per identificare gli amici di Alice viene prima effettuata una ricerca per indice con un costo di $O(\log(n))$ mentre la complessità aumenta a $O(m \log(n))$ quando si cerca di scoprire chi è amico di Alice, richiedendo molteplici ricerche per ogni nodo potenzialmente collegato ad Alice.

Le relazioni nei graph database nativi sono considerate entità di prima classe, il che facilita la navigazione in ogni direzione e aumenta l’efficienza degli algoritmi di ricerca. In contrasto, la gestione delle relazioni nei graph database non nativi si rivela più ardua. L’inversione della direzione di un attraversamento, per esempio, può richiedere la creazione di costosi indici di ricerca inversa o l’impiego di ricerche brute-force attraverso gli indici originali, tecniche che, pur raggiungendo l’obiettivo, incidono negativamente sulle prestazioni complessive e sull’efficienza del database.

3.2 Differenza tra Neo4j e DBMS relationali

I DBMS relationali (o *relational database management systems*, RDBMS) si basano sul modello relazionale introdotto da Edgar Frank Codd nel 1970 [44] dove i dati sono organizzati in tuple, insiemi non ordinati di attributi, e relazioni, ovvero insiemi di tuple. In questo caso le relazioni tra le tabelle sono deducibili tramite chiavi esterne e calcolabili al momento delle interrogazioni (o query) tramite operazioni di giunzione (o join).

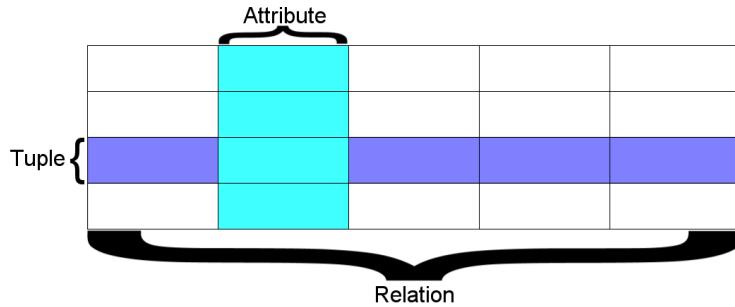


Figura 3.3: Terminologia del modello relazionale [45].

<i>Studenti</i>		
Matricola	Nome	Cognome
1234	Mario	Rossi
5678	Irene	Bianchi
9012	Elisa	Verdi

<i>Professori</i>		
ID_Prof	Nome	Cognome
1	Giulio	Russo
2	Marco	Ferrari
3	Federica	Romano

<i>Insegnamenti</i>	
ID_Prof	ID_Corso
1	244AA
2	729AA
3	730AA

<i>Corsi</i>			
ID_Corso	Nome		
244AA	Ingegneria del software		
730AA	Laboratorio di web scraping		
729AA	Interazione uomo-macchina		

<i>Esami</i>			
ID_Corso	Matricola	Voto	Data
244AA	1234	28	12/12/2023
730AA	9012	32	6/11/2023
729AA	5678	30	5/9/2023
730AA	1234	32	6/7/2023

Figura 3.4: Esempio di rappresentazione del dominio dei dati di una università in un database relazionale. Si noti come le relazioni tra le entità siano deducibili tramite chiavi esterne e calcolabili al momento delle interrogazioni tramite operazioni di join.

Questa struttura funziona efficacemente quando si gestiscono grandi quantità di dati strutturati e il modello dei dati è ben definito e stabile. Tuttavia, le operazioni che implicano la ricerca di pattern complessi, come trovare gli amici di un amico in una rete sociale, possono diventare meno efficienti in un RDBMS poichè le relazioni tra le entità non sono memorizzate direttamente ma devono essere dedotte al momento dell'interrogazione. Questo può comportare un aumento della complessità computazionale e un rallentamento delle prestazioni, specialmente quando si lavora con dati complessi e interconnessi.

Al contrario dei database relazionali i graph database come Neo4j utilizzano nodi e archi per rappresentare e memorizzare i dati, gestendo le relazioni a livello di record singolo e utilizzando offset e puntatori per navigare efficacemente tra i dati da recuperare per le interrogazioni. In questo caso i nodi rappresentano le entità del dominio e gli archi le relazioni tra queste entità. Neo4j non deduce le relazioni tra i nodi ma le memorizza come parte integrante del suo modello dei dati, in modo che connessioni nel dominio corrispondono a connessioni nei dati e consentendo così l'esecuzione di interrogazioni relative alle relazioni in modo molto più efficiente.

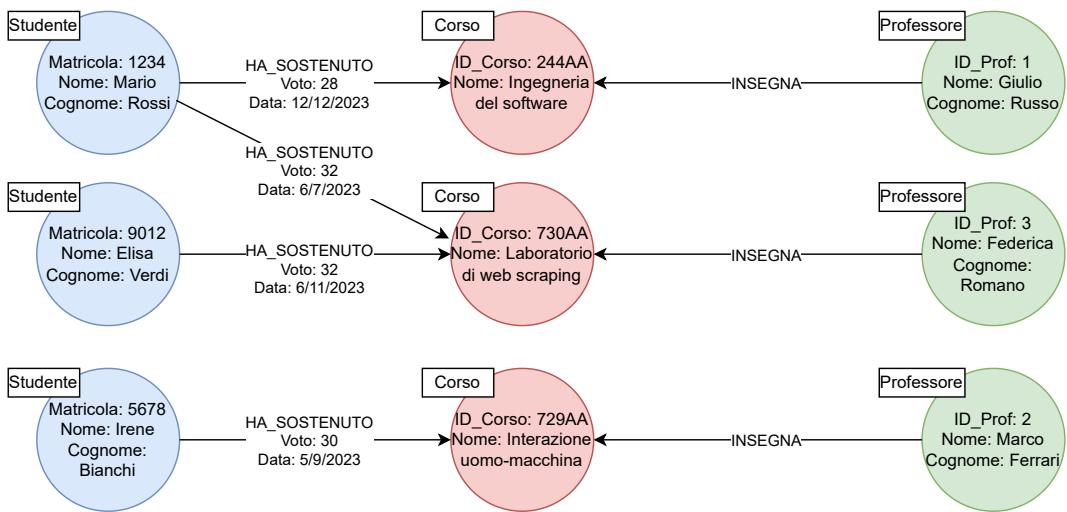


Figura 3.5: Esempio di rappresentazione dello stesso dominio dei dati mostrato in figura 3.4 ma in un graph database, le relazioni tra le entità sono memorizzate come parte integrante del modello dei dati.

Questo approccio rende Neo4j un graph database particolarmente adatto per applicazioni che richiedono analisi di reti complesse e interrogazioni basate su relazioni permettendo di rappresentare in modo più naturale aspetti del mondo reale. Alcuni esempi di applicazioni che possono beneficiare dell'utilizzo di Neo4j sono [46] [47] :

- rilevamento di frodi
- motori di raccomandazione
- basi di conoscenza
- gestione delle reti di telecomunicazioni
- Identity and Access Management (IAM)

3.2.1 Confronto di una query tra Neo4j e un RDBMS

Per illustrare la differenza nella interrogazione dei dati tra Neo4j e un RDBMS si supponga di avere un database relazionale con la struttura riportata in figura 3.4 e un graph database con la struttura riportata in figura 3.5. Viene richiesto ai fini del confronto di “*trovare tutti i corsi che sono stati superati con un voto superiore a 29 da studenti che hanno anche sostenuto un corso tenuto da un particolare professore, in questo caso il professore Marco Ferrari*”.

Query in un RDBMS

In un database relazionale la query proposta nella sezione 3.2.1 potrebbe apparire come segue:

```

1  SELECT DISTINCT c2.Nome
2  FROM Studenti s
3  JOIN Esami e1 ON s.Matricola = e1.Matricola
4  JOIN Corsi c1 ON e1.Id_Corso = c1.Id_Corso
5  JOIN Insegnamenti i ON c1.Id_Corso = i.Id_Corso
6  JOIN Professori p ON i.Id_Prof = p.Id_Prof
7  JOIN Esami e2 ON s.Matricola = e2.Matricola
8  JOIN Corsi c2 ON e2.Id_Corso = c2.Id_Corso
9  WHERE p.Nome = 'Marco' AND p.Cognome = 'Ferrari' AND e2.Voto > 29;
```

Codice 1: Query SQL per trovare il nome dei corsi superati con un voto superiore a 29 da studenti che hanno sostenuto un corso tenuto da un professore specifico.

In questa query l’obiettivo è selezionare i nomi dei corsi basandosi su una serie di join che verificano le relazioni tra studenti, esami e corsi, filtrando per voto e professore specifici. Se il database è di grandi dimensioni le operazioni di join possono diventare complesse e richiedere molto tempo per essere eseguite, specialmente se non sono presenti indici adeguati.

Query in Neo4j

In Neo4j la stessa query nella sezione 3.2.1 può essere espressa in Cypher, il linguaggio di query dichiarativo basato su pattern, come segue:

```

1 MATCH (prof:Professore {Nome: 'Marco', Cognome:
  ↳ 'Ferrari'})-[:INSEGNA]->(cMF:Corso)
2 MATCH (cMF)<-[HA_SOSTENUTO]-(s:Studente)-[h:HA_SOSTENUTO]->(c2:Corso)
3 WHERE h.Voto > 29
4 RETURN DISTINCT c2.Nome;

```

Codice 2: Query Cypher per trovare il nome dei corsi superati con un voto superiore a 29 da studenti che hanno sostenuto un corso tenuto da un professore specifico.

In questa query Cypher, ciò che in SQL richiederebbe join multipli e condizioni di filtro viene espresso in maniera diretta e intuitiva. La performance di tale query è inoltre più prevedibile poiché la latenza è correlata principalmente al numero di percorsi da esplorare piuttosto che alla dimensione complessiva del grafo.

Osservazioni

In entrambe le query la logica applicata è la stessa: non è di interesse il voto nel corso tenuto da “Marco Ferrari” ma solo che lo studente abbia sostenuto un esame di un corso tenuto da lui. Le prestazioni di queste query possono variare notevolmente tra un database relazionale e un database a grafo, specialmente quando il numero di studenti e corsi è molto elevato. Mentre la query relazionale richiede join multiple e può essere onerosa in termini di performance, la query a grafo è naturalmente ottimizzata per questo tipo di interrogazioni che coinvolgono nodi adiacenti, fornendo risposte più veloci e efficienti.

3.3 Il modello Labeled Property Graph dei graph database

Il modello *Labeled Property Graph* (LPG) è il modello a grafo più diffuso nei graph database e rappresenta il modello di riferimento per Neo4j [43]. Questo modello si distingue per alcune caratteristiche peculiari che lo rendono particolarmente adatto a rappresentare dati complessi e interconnessi in modo intuitivo e flessibile. Da un punto di vista teorico un LPG può essere definito come un multigrafo diretto in cui sono consentiti self-loop, dove nodi e relazioni sono etichettati e possono avere proprietà.

Il concetto di Labeled Property Graph è stato introdotto informalmente in [48] come uno dei modelli disponibili per rappresentare grafi, discutendo nell’articolo le caratteristiche di questo modello e come questo possa essere utilizzato per ottenere altri modelli a grafo, come ad esempio il modello a grafo RDF (Resource Description Framework) dove i nodi rappresentano risorse (URI) e le relazioni rappresentano i link tra queste risorse. Una delle caratteristiche del modello LPG è il fatto che sia *schema-free*, ovvero che non sia necessario definire uno schema prima di inserire i dati nel database. Questo significa che i nodi e le relazioni possono avere proprietà diverse e che nuove etichette e nuove proprietà possono essere aggiunte in qualsiasi momento senza dover modificare lo schema del database. Questa flessibilità è particolarmente utile in contesti reali in cui i dati sono eterogenei, in evoluzione e in cui è difficile prevedere a priori quali proprietà saranno necessarie in futuro.

Il problema dell’eccessiva flessibilità di un modello *schema-free*, che renderebbe prono a errori e difficile da gestire un graph database a utenti provenienti da un contesto relazionale, è stato affrontato in [49]. Questo articolo propone un formalismo *schema-driven* per poter consentire di importare dati da altre sorgenti, come ad esempio i database relazionali, in un graph database.

La ricchezza di informazioni che un LPG può rappresentare e la sua flessibilità lo rendono una scelta ideale per rappresentare i dati della Blockchain Bitcoin, consentendo di poter rappresentare in modo naturale lo scambio di valore tra entità e le relazioni tra esse. Mentre una definizione più formale e rigorosa di LPG verrà fornita successivamente nel capitolo 4, riprendendo e estendendo quella proposta in [49] per modellare i dati della blockchain Bitcoin, di seguito sono elencate in maniera più informale le principali caratteristiche di questo modello:

- un LPG è composto da nodi, relazioni, etichette (o label), tipi e proprietà
- i nodi
 - possono essere contrassegnati con una o più etichette che servono non solo a categorizzarli ma anche a indicare i ruoli che assumono all’interno del dataset (e.g. **Studente**, **Professore**). Le etichette facilitano il raggruppamento dei nodi e la loro ricerca all’interno del database
 - possono avere proprietà, ovvero coppie chiave-valore che rappresentano informazioni aggiuntive sul nodo. Queste chiavi sono stringhe e i valori possono essere di tipi primitivi Java, stringhe o array di

questi tipi, consentendo una rappresentazione ricca e dettagliata delle informazioni

- le relazioni

- collegano nodi nel grafo e hanno sempre esattamente un tipo, una direzione, un nodo sorgente e un nodo destinazione
- possono avere proprietà, come i nodi, che rappresentano informazioni aggiuntive sulla relazione

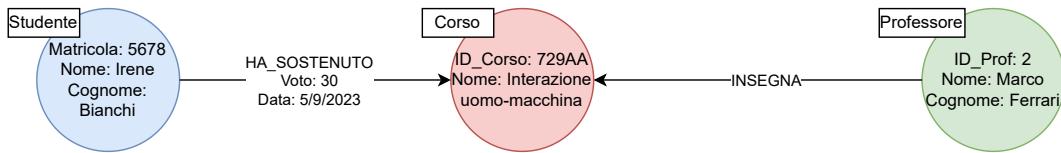


Figura 3.6: Esempio di Labeled Property Graph che illustra le relazioni tra entità in un contesto accademico. Il LPG in figura è un sottografo del grafo in figura 3.5 dove i nodi etichettati come **Studente**, **Corso** e **Professore** rappresentano le entità, ciascuno con proprietà come **Matricola**, **ID_Corso** e **ID_Prof**. Le relazioni con tipo **HA_SOSTENUTO** e **INSEGNA** connettono gli studenti ai corsi e i professori ai corsi rispettivamente.

3.4 Query su grafi: introduzione a Cypher

Cypher è un linguaggio di query dichiarativo basato su pattern per Neo4j. La sua sintassi è stata progettata per essere intuitiva e facile da imparare consentendo di esprimere interrogazioni in modo chiaro e conciso. La descrizione dei pattern in Cypher utilizza ASCII art, ovvero testo, al fine di comporre un disegno per descrivere al database quali dati devono essere recuperati, in maniera simile a come si disegnerebbe un grafo su un foglio di carta.

La rappresentazione in Cypher per il pattern mostrato in figura 3.6 è la seguente:

```

1 bianchi {Matricola: 5678})-[:HA_SOSTENUTO]->(esame {ID_Corso:  
    ↳ "729AA"})<-[INSEGNA]-(ferrari {ID_Prof: 2})

```

Nel pattern sopra **bianchi**, **esame** e **ferrari** sono variabili che rappresentano i nodi del grafo che hanno le proprietà specificate tra {} mentre **HA_SOSTENUTO** e **INSEGNA** sono i tipi delle relazioni tra i nodi. Le variabili in Cypher

sono opzionali e possono essere utilizzate per recuperare i dati dai nodi e dalle relazioni in modo da poterli riferire in futuro nella query.

3.4.1 Struttura delle query

La struttura delle query in Cypher è ereditata da SQL (dove le interrogazioni sono costruite componendo clausole) e da SPARQL (dove le interrogazioni sono costruite componendo pattern). Le clausole sono concatenate in modo che l'output di una clausola venga utilizzato come input per la successiva.

3.4.2 Rappresentazione dei nodi in Cypher

Un nodo in Cypher viene rappresentato usando le parentesi tonde (), in maniera simile a come si rappresenterebbero visivamente i nodi in un grafo disegnato su un foglio di carta. Per riferire un particolare nodo in futuro nella query è possibile assegnargli un nome, e.g. (s) o (studente) per uno studente del grafo mostrato precedentemente in figura 3.5. Se il nodo non è rilevante per la query si può specificare un nodo anonimo usando () ma ciò non permetterà di riferirlo in futuro nella query.

Etichette (o label)

Le etichette (o label) sono utilizzate per categorizzare i nodi e per indicare i ruoli che assumono all'interno del database. Le etichette facilitano il raggruppamento dei nodi e la loro ricerca all'interno del grafo. In Cypher le etichette sono rappresentate come stringhe precedute da : e possono essere assegnate a un nodo usando la sintassi (:<etichetta>). Se un nodo ha più etichette queste possono essere specificate come (:<etichetta1>:<etichetta2>:...).

L'utilizzo delle etichette è opzionale ma aiuta Cypher a distinguere le entità all'interno del grafo e a ottimizzare l'esecuzione delle query. Se non si specifica alcuna etichetta Cypher andrà a considerare tutti i nodi del database, che può risultare un'operazione molto costosa se il grafo è di grandi dimensioni.

```

1 // nodo anonimo senza etichetta, indica un qualsiasi nodo del database
2 ()
```



```

1 // variabile "p" che rappresenta un qualsiasi nodo con etichetta
  ↳ "Professore". L'uso di variabili è opzionale ma consente di riferire
  ↳ il nodo in futuro nella query (non sarebbe possibile riferirsi al
  ↳ nodo usando solo :Professore)
2 (p:Professore)
```



```

1 // nodo con etichetta "Studente" e "Tutor" senza utilizzo di variabili,
  ↳ indica un qualsiasi nodo con etichetta "Studente" e "Tutor"
2 (:Studente:Tutor)
```

Codice 3: Esempio di nodi in Cypher.

3.4.3 Rappresentazione delle relazioni in Cypher

Le relazioni in Cypher sono rappresentate come frecce direzionali --> o <-- tra due nodi. Analogamente ai nodi, la sintassi per esprimere le relazioni è ispirata alla rappresentazione visiva con delle linee che collegano (). Una relazione in Cypher viene rappresentata usando le parentesi quadre [] ed è possibile assegnarle un nome per riferirla in futuro nella query, e.g. [r] o [relazione]. Se la relazione non è rilevante per la query è possibile specificare una relazione anonima usando [] ma ciò non permetterà di riferirla in futuro nella query.

Le relazioni non orientate in Cypher sono rappresentate con trattini privi di freccia --, indicando che la direzione di attraversamento della relazione non è rilevante per la query. Mentre è obbligatorio specificare la direzione della relazione quando si inserisce nel database, è possibile ometterla nelle query per indicare che la direzione non è rilevante ai fini dell'interrogazione.

Tipi delle relazioni

Al contrario dei nodi che possono avere zero o più etichette, le relazioni possono avere un solo tipo. I tipi delle relazioni sono rappresentati come stringhe precedute da : e possono essere assegnati a una relazione usando la sintassi [:<tipo>]. Se si vuole individuare le relazioni che presentano un tipo tra un insieme di tipi è possibile utilizzare la sintassi [:<tipo1>|<tipo2>|...], dove | rappresenta l'operatore logico OR.

```

1 // relazione "r" orientata di tipo "HA_SOSTENUTO" tra coppie di nodi "a"
  ↵ e "b"
2 (a)-[r:HA_SOSTENUTO]->(b)

```

```

1 // relazione "r" orientata di tipo "HA_SOSTENUTO" o "INSEGNA" tra coppie
  ↵ di nodi "a" e "b"
2 (a)-[r:HA_SOSTENUTO|INSEGNA]->(b)

```

```

1 // relazione anonima non orientata tra coppie di nodi "a" e "b"
2 (a)--(b)

```

Codice 4: Esempio di relazioni in Cypher.

3.4.4 Proprietà di nodi o relazioni

Le proprietà sono coppie chiave-valore che rappresentano informazioni aggiuntive sui nodi e sulle relazioni. Per aggiungere proprietà a un nodo o a una relazione è sufficiente specificare la chiave e il valore tra parentesi graffe {}.

```

1 // nodo "s" con proprietà "Matricola" uguale a 5678
2 (s {Matricola: 5678})

```

```

1 // relazione "r" tra i nodi "a" e "b" con proprietà "Voto" uguale a 30
2 (a)-[r {Voto: 30}]->(b)

```

Codice 5: Esempio di proprietà in Cypher.

3.4.5 Clausole

Cypher offre diverse clausole per la costruzione delle query, in questa sezione verranno trattate *“by example”* quelle principali offerte dal linguaggio e che sono state utilizzate nell’ambito della tesi. Per una trattazione più approfondita si rimanda alla documentazione ufficiale di Cypher [50].

MATCH

La clausola **MATCH** è utilizzata per ricercare pattern nel grafo e può essere specificata all'inizio della query o successivamente dopo una clausola **WITH**. **MATCH** è spesso utilizzata in combinazione con **WHERE** per filtrare i risultati della query.

```

1 // trova e restituisce tutti i nodi del grafo
2 MATCH (n)
3 RETURN n

```

Codice 6: Specificando un singolo nodo senza etichetta verranno restituiti tutti i nodi del grafo.

```

1 // trova tutti i nodi del grafo con etichetta "Studente" e restituisce la
  → proprietà "Matricola" di tutti i nodi trovati
2 MATCH (s:Studente)
3 RETURN s.Matricola

```

Codice 7: Specificando un singolo nodo con etichetta verranno restituiti tutti i nodi del grafo con quella etichetta.

RETURN

La clausola **RETURN** è utilizzata per definire le parti del pattern (nodi, relazioni, proprietà) che devono essere restituite come risultato della query.

```

1 MATCH (s:Studente {Matricola: 5678})-[r:HA_SOSTENUTO]->(c:Corso)
2 RETURN s.Matricola, s.Nome, s.Cognome, c.ID_Corso, c.Nome, r.Voto,

```

Codice 8: Restituzione delle proprietà di un nodo e delle relazioni in uscita di tipo **HA_SOSTENUTO**.

WITH

La clausola **WITH** è utilizzata per definire le parti del pattern (nodi, relazioni, proprietà) che devono essere passate alla clausola successiva della query.

L'utilizzo di **WITH** ha effetto sullo scope delle variabili: tutte le variabili che non sono esplicitamente incluse nella clausola **WITH** non saranno disponibili alla clausola successiva della query. Usando la wildcard * è possibile passare tutte le variabili nello scope corrente alla clausola successiva della query. Uno degli utilizzi principali di **WITH** è quello di effettuare operazioni di aggregazione per poter calcolare funzioni come COUNT, SUM, AVG, MIN, MAX e COLLECT. In Cypher, l'operazione di **GROUP BY** tipica dei database relazionali è realizzata in modo implicito da tutte le funzioni di aggregazione. In un'istruzione **WITH** o **RETURN** qualsiasi elemento che non fa parte di una funzione di aggregazione agirà come chiave per l'aggregazione, raggruppando i risultati sulla base di questi elementi non aggregati. Ad esempio, se si considera il codice 9, in **WITH s.Matricola, MIN(r.Voto) as minVoto** la clausola **WITH** raggruppa i risultati in base alla matricola dello studente e calcola il voto minimo per ciascuno studente.

```

1 MATCH (s:Studente)-[r:HA_SOSTENUTO]->(c:Corso)
2 WITH s.Matricola, MIN(r.Voto) as minVoto
3 WHERE minVoto > 25
4 RETURN s.Matricola, s.Nome, s.Cognome

```

Codice 9: Restituzione di tutti gli studenti che hanno avuto nella propria carriera accademica solo voti maggiori di 25.

WHERE

La clausola **WHERE** è utilizzata per aggiungere dei filtri ai pattern specificati e può essere utilizzata in combinazione con una clausola **MATCH**, **OPTIONAL MATCH** o **WITH**. Questa clausola deve essere vista come parte integrante della clausola **MATCH** e non come una clausola separata.

Gli indici potrebbero essere utilizzati per velocizzare le interrogazioni che utilizzano la clausola **WHERE** in alcune circostanze.

```

1 MATCH (s)-->(c)
2 WHERE s:Studente AND c:Corso AND s.Matricola = 5678
3 RETURN c.ID_Corso AS esame_sostenuto

```

Codice 10: Filtraggio dei nodi per etichetta e proprietà.

```

1 MATCH (s:Studente)-[r]->(c)
2 WHERE 25 <= r.voto <= 32
3 RETURN s.Matricola AS matricola, c.ID_Corso AS esame_sostenuto, r.voto AS
   ↵ voto

```

Codice 11: Filtraggio delle relazioni per proprietà.

CREATE

La clausola **CREATE** è utilizzata per creare nodi e relazioni nel grafo.

Ogni nodo creato può avere un numero arbitrario di etichette e proprietà e può essere associato a una o più variabili per riferirlo in futuro nella query.

```

1 CREATE (mr:Studente {Matricola: 1234, Nome: "Mario", Cognome: "Rossi"}),
   ↵ (ib:Studente {Matricola: 5678, Nome: "Irene", Cognome: "Bianchi"}),
   ↵ (ev:Studente {Matricola: 9012, Nome: "Elisa", Cognome: "Verdi"})
2 RETURN mr, ib, ev

```

Codice 12: Creazione di nodi con etichetta e proprietà.

```

1 CREATE (gr:Professore {ID_Prof: 1, Nome: "Giulio", Cognome:
   ↵ "Russo"})-[r:INSEGNA]->(c:Corso {ID_Corso: "244AA", Nome: "Ingegneria
   ↵ del software"})
2 RETURN gr, r, c

```

Codice 13: Creazione di nodi e relazioni con tipo e proprietà.

L'esempio precedente ha creato una nuova relazione di tipo **INSEGNA** tra due nodi specificati **gr** e **c** non precedentemente presenti nel grafo. Per creare invece una relazione tra nodi già esistenti è necessario prima recuperare i nodi di interesse tramite una clausola **MATCH** e poi creare la relazione tra i nodi recuperati.

```

1 MATCH (s:Studente {Matricola: 5678}), (c:Corso {ID_Corso: "244AA"})
2 CREATE (s)-[r:HA_SOSTENUTO {Voto: 28, Data: "12/12/2023"}]->(c)

```

Codice 14: Creazione di una relazione tra nodi esistenti.

SET

La clausola **SET** è utilizzata per aggiornare le etichette dei nodi e le proprietà di nodi e relazioni.

```

1 MATCH (s:Studente)-[r:HA_SOSTENUTO]->(c)
2 WHERE r.Voto = 32
3 SET r.Lode = true

```

Codice 15: Aggiornamento delle proprietà di una relazione.

MERGE

La clausola **MERGE** è utilizzata per trovare pattern nel grafo o, se non presenti, crearli. In questo modo agisce come una combinazione di **MATCH** e **CREATE** e può essere utilizzata in combinazione con una clausola **ON CREATE** e **ON MATCH** per specificare le azioni da eseguire in caso di creazione o di match del pattern.

In maniera simile a **MATCH**, **MERGE** può essere utilizzata per trovare molteplici corrispondenze di un pattern nel grafo. Se ci sono più corrispondenze esse verranno passate alle clausole successive della query.

```

1 // NB: il Mario Rossi già presente nel grafo ha matricola 1234
2 MERGE (s:Studente {Nome: "Mario", Cognome: "Rossi", Matricola: 9999})

```

Codice 16: Fare **MERGE** di un nodo con proprietà che non sono presenti in nessun nodo esistente nel grafo porterà alla creazione di un nuovo nodo.

```

1 MERGE (s:Studente {Nome: "Mario", Cognome: "Rossi", Matricola: 1234})
2 RETURN s

```

Codice 17: Fare **MERGE** di un nodo con proprietà uguali a quelle di un nodo esistente nel grafo non porterà alla creazione di un nuovo nodo.

CALL

La clausola **CALL** è utilizzata per valutare una subquery o richiamare una procedura.

Ogni procedura richiamata necessita di specificare tutti i parametri richiesti. La lista di tutte le procedure offerte da Cypher è disponibile sulla sezione dedicata alle funzioni della documentazione ufficiale di Cypher.

```

1 CALL db.labels()

```

Codice 18: Invocazione della procedura **db.labels()** per recuperare tutte le etichette presenti nel grafo.

UNWIND

La clausola **UNWIND** è utilizzata per iterare su una lista di valori e restituire una riga per ogni valore nella lista. Le clausole successive alla clausola **UNWIND** verranno eseguite per ogni valore nella lista.

```

1 UNWIND [1, 2, 3, 4, 5] AS numero
2 RETURN numero * 2

```

Codice 19: Iterazione su una lista di valori per restituire il doppio di ciascun valore.

LOAD CSV

LOAD CSV è una clausola utilizzata per importare dati da file CSV, in particolare:

- richiede di utilizzare una variabile per riferire ciascuna riga del file CSV utilizzando la parola chiave **AS**
- il separatore di campo predefinito è la virgola ma può essere specificato un separatore diverso utilizzando la parola chiave **FIELDTERMINATOR**
- la posizione dei dati sul filesystem o su un server remoto è specificata utilizzando la parola chiave **FROM**

Supponendo di avere un file CSV `studenti.csv` memorizzato localmente con i dati degli studenti nel formato **Matricola, Nome, Cognome**:

Matricola, Nome, Cognome
 1111, Luca, Rossi
 2222, Giulia, Verdi
 3333, Marco, Bianchi

la query per importare i dati del file CSV nel grafo sarà la seguente:

```

1 LOAD CSV WITH HEADERS FROM "file:///studenti.csv" AS row
2 CREATE (:Studente {Matricola: row.Matricola, Nome: row.Nome, Cognome:
  ↳ row.Cognome})

```

Codice 20: Importazione di dati da un file CSV.

3.5 Awesome Procedures On Cypher (APOC)

APOC [51] è una libreria di procedure e funzioni per Neo4j che permette di estendere le funzionalità di Cypher. In particolare le procedure APOC consentono di importare ed esportare dati in diversi formati (come CSV, JSON, XML) ma anche di eseguire operazioni di manipolazione dei dati, di refactoring del grafo e di esecuzione di algoritmi (come `apoc.algo.aStar` per l'algoritmo di ricerca A* [52] e `apoc.algo.dijkstra` per la ricerca del cammino più breve tra due nodi in un grafo pesato [53]).

Nel contesto specifico di questa tesi sono state utilizzate le seguenti procedure APOC:

- `apoc.periodic.iterate`, procedura che prevede due diverse istruzioni: la prima istruzione (esterna) fornisce uno stream di dati da processare e

la seconda (interna) processa un elemento alla volta di questo stream o l'intero batch di elementi se il flag `batchMode` è impostato a `"BATCH"` (e lo è per default). Questa procedura è stata utilizzata per leggere i dati da un file CSV e creare i nodi e le relazioni corrispondenti nelle varie rappresentazioni della blockchain come grafo, come verrà successivamente mostrato nel capitolo 5

```

1 CALL apoc.periodic.iterate(
2   'LOAD CSV WITH HEADERS FROM "file:///studenti.csv" AS row',
3   'CREATE (:Studente {Matricola: row.Matricola, Nome: row.Nome, Cognome:
4     ↪ row.Cognome})',
5   {batchSize:10, batchMode:"BATCH"}
)

```

Codice 21: Utilizzo di `apoc.periodic.iterate` per importare il file CSV `studenti.csv` descritto precedentemente nella sezione 3.4.5 e creare i corrispondenti nodi `Studente` nel grafo. `batchSize` è il numero di righe del file CSV che vengono fornite dalla query esterna (`LOAD CSV WITH HEADERS FROM 'file:///studenti.csv' AS row`) alla query interna (`CREATE (:Studente Matricola: row.Matricola, Nome: row.Nome, Cognome: row.Cognome)`). `batchMode` è impostato a `"BATCH"` per far processare `batchSize` righe alla query interna in un'unica transazione.

- `apoc.periodic.commit`, procedura che permette di eseguire una query in transazioni diverse fino a quando non viene restituito 0 come risultato. Questa procedura è stata utilizzata per selezionare n indirizzi casuali, dopo l'importazione di ciascun chunk, tra quelli coinvolti come mittenti in almeno una transazione al fine di selezionare un campione rappresentativo di indirizzi da utilizzare successivamente per le query di analisi.

```

1 CALL apoc.periodic.commit("
2 MATCH (s:AttesaImmatricolazione)
3 DETACH DELETE s // elimina i nodi con label AttesaImmatricolazione e le
   ↵ relazioni ad essi collegati
4 RETURN count(s)
5 ", {})

```

Codice 22: Considerando sempre l'esempio in figura 3.5, ipotizzando che sia scaduto il periodo di immatricolazione per studenti universitari e che si voglia eliminare dal database tutti gli studenti che non hanno finalizzato la propria immatricolazione (supponendo che siano identificati dall'etichetta `AttesaImmatricolazione`), è possibile utilizzare `apoc.periodic.commit` per raggiungere questo obiettivo: l'esecuzione della query interna verrà ripetuta finché non ci saranno più nodi `AttesaImmatricolazione` nel database, ovvero `count(s)` restituirà 0.

- `apoc.export.csv.query`, procedura che permette di esportare i risultati di una query in formato CSV. Questa procedura è stata utilizzata per esportare su file il risultato della selezione di n indirizzi casuali, dopo l'importazione di ciascun chunk, tra quelli che sono coinvolti come mittenti in almeno una transazione al fine di selezionare un campione rappresentativo di indirizzi da utilizzare successivamente per le query di analisi.

```

1 CALL apoc.export.csv.query("
2 MATCH (s:Studente)-[:HA_SOSTENUTO]->(e:Esame)
3 WITH s, MIN(e.voto) AS minVoto
4 WHERE minVoto >= 25
5 RETURN s.matricola AS Matricola
6 ", "students.csv", {})

```

Codice 23: Considerando ancora l'esempio in figura 3.5, se si vuole esportare sul file `students.csv` le matricole degli studenti che non hanno mai preso ad esami voti inferiori a 25 è possibile utilizzare `apoc.export.csv.query` per raggiungere questo obiettivo.

- `apoc.path.expandConfig`, procedura che permette di espandere percorsi nel grafo specificando i nodi da cui iniziare la ricerca e in cui interromperla, i tipi delle relazioni da attraversare e le direzioni da seguire. Questa

procedura è stata utilizzata per implementare la address taint analysis e la backward address taint analysis in modo da marcare come contaminati i nodi che sono stati coinvolti in flussi di denaro anch'essi contaminati.

```

1 MATCH (s:Studente {Matricola: 5678})
2 CALL apoc.path.expandConfig(s, {
3   relationshipFilter:'AMICO>',
4   labelFilter:'Studente',
5   minLevel: 1,
6   maxLevel: 3
7 }) YIELD path
8 RETURN path

```

Codice 24: ipotizzando che i nodi `Studente` in figura 3.5 abbiano una relazione di amicizia tra loro di tipo `AMICO` e che si voglia espandere le catene di amicizia a partire da un nodo `Studente` con matricola `5678` fino a una profondità fissata `maxLevel` restituendo i cammini trovati, è possibile utilizzare `apoc.path.expandConfig` per tale scopo.

3.6 Indici e ottimizzazione delle query

Gli indici in Neo4j vengono utilizzati per velocizzare le query Cypher. Se disponibili vengono utilizzati automaticamente e nel caso in cui ne siano presenti molteplici sarà compito del planner Cypher (cioè la componente di Neo4j che si occupa di tradurre le query in piani di esecuzione) scegliere quale indice utilizzare per ottimizzare l'esecuzione della query. Viene offerta la possibilità di forzare l'utilizzo di un indice specifico tramite la clausola `USING` piuttosto che lasciare che il planner scelga l'indice da utilizzare. I vincoli in Neo4j sono implementati internamente come indici e sono utilizzati per 1. garantire l'unicità delle proprietà di un nodo o di una relazione e 2. garantire l'esistenza di una o più proprietà di un nodo o di una relazione.

Un indice in Neo4j è una struttura dati che permette al Graph Engine di recuperare rapidamente i dati al costo di una maggiore occupazione di memoria del grafo. Una buona pratica è quella di creare gli indici una volta che i dati sono caricati. L'utilizzo degli indici renderà più veloci le query ma rallenterà le operazioni di scrittura quali l'inserimento, l'aggiornamento e la cancellazione dei dati indicizzati essendo in questi casi richiesto un aggiornamento della struttura che realizza gli indici ogni volta che i dati cambiano.

Il comportamento predefinito in Neo4j consiste nell'utilizzare un solo indice per una query. Una sottoquery può utilizzare un indice aggiuntivo.

I tipi di indici disponibili in Neo4j sono:

- RANGE
- Composite
- POINT
- Full-text

3.6.1 RANGE

Un indice RANGE rappresenta l'indice predefinito utilizzato in Neo4j ed è una implementazione proprietaria di un indice B-tree. Un indice di questo tipo può essere definito su una proprietà di un nodo o di una relazione² al fine di velocizzare le query, consentendo così una complessità logaritmica per la ricerca all'interno del grafo [43].

Un indice RANGE può essere utilizzato per velocizzare il codice Cypher nei seguenti casi:

- controlli di uguaglianza con =
 - nota: per le proprietà di tipo stringa un indice TEXT potrebbe avere performance migliori
- confronti tra intervalli con <, <=, >, >=
- confronti tra stringhe con **STARTS WITH**
 - nota: gli indici TEXT possono essere usati per confronti usando **STARTS WITH** ma potrebbero non essere efficienti come gli indici RANGE
- controlli di esistenza con **IS NOT NULL**

²il nome dell'indice deve essere univoco tra i nomi degli indici e dei vincoli.

```

1 CREATE INDEX matricola_stud_index IF NOT EXISTS
2 FOR (s:Studente)
3 ON (s.Matricola)

```

Codice 25: Esempio di definizione di un indice RANGE sulla proprietà **Matricola** dei nodi con etichetta **Studente** del grafo in figura 3.5 al fine di velocizzare le query che utilizzano questa proprietà.

3.6.2 Composite

Un indice composito combina valori di più proprietà per un nodo o una relazione. Questo tipo di indice viene creato quando proprietà multiple sono sempre testate insieme in una query.

```

1 CREATE INDEX nome_cognome_stud_index IF NOT EXISTS
2 FOR (s:Studente)
3 ON (s.Nome, s.Cognome)

```

Codice 26: Esempio di definizione di un indice composito sulle due proprietà **Nome** e **Cognome** dei nodi con etichetta **Studente** del grafo in figura 3.5 al fine di velocizzare le query che utilizzano queste proprietà.

3.6.3 TEXT

Un indice TEXT supporta proprietà di nodi o relazioni che sono stringhe. Un indice di questo tipo può essere utilizzato per velocizzare il codice Cypher nei seguenti casi:

- controlli di uguaglianza con =
- confronti tra stringhe con **ENDS WITH**, **CONTAINS**
- appartenenza a una lista di stringhe con **IN**
 - e.g. **WHERE** a.hash **IN** ["1dice...", "1DpsR..."]

```

1 CREATE TEXT INDEX nome_stud_index IF NOT EXISTS
2 FOR (s:Studente)
3 ON (s.Nome)

```

Codice 27: Esempio di definizione di un indice TEXT sulla proprietà di tipo stringa **Nome** dei nodi con etichetta **Studente** al fine di velocizzare le query che utilizzano questa proprietà.

3.6.4 POINT

Gli indici di tipo POINT sono utilizzati per proprietà di nodi o relazioni che rappresentano dati di tipo spaziale. Un indice di questo tipo può essere utilizzato per velocizzare le query che coinvolgono il calcolo di distanze tra punti nello spazio.

```

1 CREATE POINT INDEX posizione_aula_index IF NOT EXISTS
2 FOR (a:Aula)
3 ON (a.posizione)

```

Codice 28: Ipotizzando rispetto al grafo in figura 3.5 che ciascun corso abbia delle relazioni di tipo **TENUTO_IN** con dei nodi **Aula** e che ciascun nodo **Aula** abbia una proprietà di tipo POINT **posizione** con **latitudine** e **longitudine**, si potrebbe velocizzare le query che coinvolgono il calcolo di distanze tra le aule creando un indice di tipo POINT sulla proprietà **posizione** di ciascun nodo **Aula**.

3.6.5 Full-text

Un indice Full-text si basa solo su valori stringa ma fornisce funzionalità di ricerca aggiuntive che non si ottengono dagli indici RANGE o TEXT. Un indice full-text può essere utilizzato per:

- proprietà del nodo o della relazione
- proprietà singole o multiple
- tipi di nodi (labels/etichette)
- tipi di relazioni

Una differenza degli indici full-text in Neo4j rispetto agli altri tipi di indici descritti precedentemente è che è necessario specificarne esplicitamente l'uso mentre gli altri indici disponibili vengono utilizzati automaticamente da Neo4j se viene verificato che possono essere utili per ottimizzare l'esecuzione della query.

Gli indici full-text sono basati sulla libreria di indicizzazione e ricerca Apache Lucene. Ciò significa che possiamo utilizzare il linguaggio di query full-text di Lucene per esprimere ciò che desideriamo cercare.

```

1 CREATE FULLTEXT INDEX questionario_val_rel_index IF NOT EXISTS
2 FOR ()-[r:HA_SOSTENUTO]-()
3 ON EACH [r.questionario_valutazione]
```

Codice 29: Ipotizzando rispetto al grafo in figura 3.5 che ciascuna relazione di tipo HA_SOSTENUTO abbia una proprietà di tipo stringa `questionario_valutazione` contenente il testo inserito da ciascuno studente nel questionario di valutazione al momento dell'iscrizione a un esame, si potrebbero velocizzare le query che coinvolgono la ricerca di testo all'interno di questa proprietà (e.g. “prove in itinere” o “ricevimenti serali per studenti lavoratori”) creando un indice full-text sulla proprietà `questionario_valutazione` di ciascuna relazione HA_SOSTENUTO.

Una volta definito l'indice full-text è possibile utilizzare la funzione `db.index.fulltext.queryRelationships` per interrogare le proprietà indicizzate con un indice full-text. In questo caso si cerca la parola chiave “prove in itinere” all'interno della proprietà `questionario_valutazione` delle relazioni HA_SOSTENUTO:

```

1 CALL db.index.fulltext.queryRelationships('questionario_val_rel_index',
2   ↳ 'prove in itinere') YIELD relationship, score
2 RETURN relationship, score
```

Codice 30: Interrogazione che utilizza un indice full-text per cercare parole chiave all'interno di una proprietà di tipo stringa.

La descrizione completa della sintassi delle query Lucene può essere trovata nella documentazione di Lucene.

3.7 Memorizzazione dei dati in Neo4j

Nel design di un graph database è fondamentale la modalità con cui i dati del grafo sono fisicamente memorizzati dato che ciò determina l'efficienza delle operazioni di attraversamento, interrogazione e scrittura.

Neo4j utilizza diversi file di memorizzazione per archiviare parti specifiche del grafo [54] [43]:

- `neostore.nodestore.db` per i nodi
- `neostore.relationshipstore.db` per le relazioni
- per le etichette, ovvero dei nomi che possono essere assegnati ai nodi per raggrupparli in categorie
 - `neostore.labelsstore.db` mantiene la lista delle etichette disponibili nel database
 - `neostore.labeltokenstore.db` mantiene l'associazione tra le etichette e i nodi del grafo
- `neostore.propertystore.db` per le proprietà dei nodi e delle relazioni

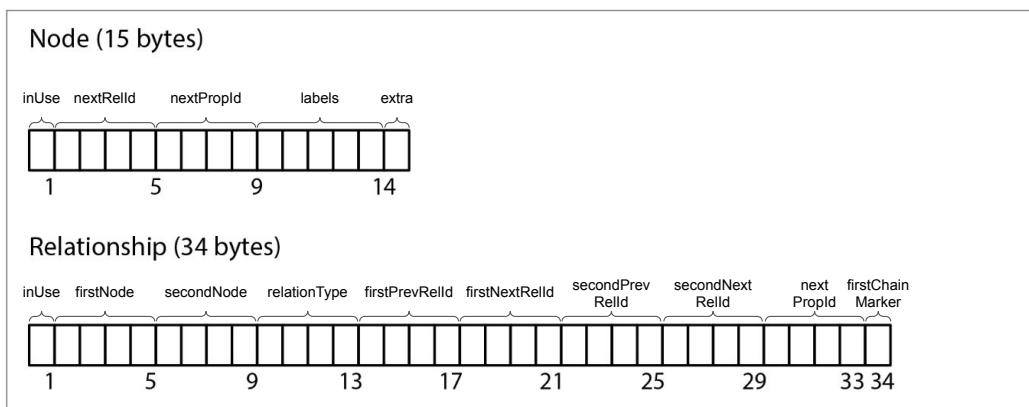


Figura 3.7: Struttura del record di un nodo e del record di una relazione in Neo4j 2.2, versioni successive potrebbero avere strutture diverse [43].

Il file `neostore.nodestore.db` memorizza i nodi del grafo come record di dimensione fissa dove ogni record è di 15 bytes, come è possibile vedere in figura 3.7. L'utilizzo di record di dimensione fissa consente di effettuare ricerche veloci nel file e di accedere direttamente alla posizione di un record senza la necessità di indici aggiuntivi. Se ad esempio si vuole accedere al nodo con `id` 100 è sufficiente calcolare l'offset nel file come $15 \times 100 = 1500$ bytes e leggere i 15 bytes successivi, richiedendo così un costo di $O(1)$ invece di $O(\log(n))$ per la ricerca tramite indice. La struttura del record di un nodo contiene un insieme di campi e puntatori³, in particolare:

- `in-use`: flag che indica se il nodo è attualmente in uso o meno, affinchè il record possa essere riutilizzato per nuovi nodi
- `nextRelId`: puntatore alla prima relazione connessa al nodo
- `nextPropId`: puntatore alla prima proprietà del nodo
- `labels`: puntatore alle etichette del nodo
- `extra`: byte riservato per usi futuri

Analogamente, le relazioni sono memorizzate nel file `neostore.relationshipstore.db` come record di dimensione fissa di 34 bytes dove ogni record contiene un insieme di flag e puntatori, in particolare:

- `firstNode`: puntatore al nodo sorgente della relazione
- `secondNode`: puntatore al nodo destinazione della relazione
- `relationshipType`: un puntatore al tipo della relazione per poter classificare le relazioni in categorie
- `firstPrevRelId, firstNextRelId e secondPrevRelId, secondNextRelId`: puntatori al record della prossima relazione e al record della relazione precedente sia per il nodo sorgente che per il nodo destinazione
- `nextPropId`: puntatore alla prima proprietà della relazione
- `firstChainMarker`: flag per indicare se la relazione è la prima nella catena di relazioni

³i puntatori, sia per i nodi che per le relazioni, riferiscono l'`id` dell'elemento a cui puntano.

La struttura del file di memorizzazione delle relazioni è progettata per essere efficiente e non ridondante. Grazie ai record a dimensione fissa e agli **id** dei record che funzionano come puntatori gli attraversamenti sono implementati semplicemente seguendo questi puntatori all'interno di una struttura dati. Questo processo può essere eseguito ad alta velocità e permette una navigazione efficace attraverso il grafo. Per seguire una particolare relazione da un nodo all'altro si compiono calcoli semplici moltiplicando **id** per la dimensione del record e si prosegue nell'esplorazione dei dati.

File di memorizzazione	Dimensione del record	Contenuto
<code>neostore.nodestore.db</code>	15 B	Nodi
<code>neostore.relationshipstore.db</code>	34 B	Relazioni
<code>neostore.propertystore.db</code>	41 B	Proprietà per nodi e relazioni
<code>neostore.propertystore.db.strings</code>	128 B	Valori delle proprietà di tipo stringa
<code>neostore.propertystore.db.arrays</code>	128 B	Valori delle proprietà di tipo array
Proprietà indicizzate	$1/3 \times AVG(X)$	Ogni elemento indicizzato è circa $\frac{1}{3}$ della dimensione media del valore della proprietà

Tabella 3.1: Gestione dei dati in Neo4j, ogni file memorizza elementi specifici del grafo e sono strutturati per ottimizzare le operazioni di lettura e scrittura [54] [55].

Oltre ai file di memorizzazione relativi a nodi e relazioni, già descritti in precedenza in figura 3.7, si può notare che per la memorizzazione delle proprietà che sono di tipo stringa e array sono utilizzati due file distinti: `neostore.propertystore.db.strings` e `neostore.propertystore.db.arrays`. Questi file specializzati facilitano l'organizzazione e il recupero efficiente dei dati che hanno strutture variabili e potenzialmente più grandi, come stringhe di testo o collezioni di elementi.

Le memorizzazioni di elementi indicizzati è anch'essa ottimizzata per garantire prestazioni elevate, richiedendo circa un terzo della dimensione media delle pro-

prietà indicizzate, riducendo la quantità di spazio necessario per memorizzare riferimenti che potrebbero non essere utilizzati frequentemente. I dati archiviati all'interno dei record sono rappresentati come oggetti Java, il linguaggio di programmazione in cui Neo4j è sviluppato.

Capitolo 4

Rappresentazioni della blockchain di Bitcoin come grafo

Nel contesto della blockchain del protocollo Bitcoin la rappresentazione dei dati assume un ruolo cruciale nel comprendere l'interconnessione e il flusso di valore tra indirizzi e transazioni. In questo capitolo sono esplorate tre diverse rappresentazioni della blockchain come grafo, introducendo - in maniera più rigorosa rispetto alla definizione informale fornita nella sezione 3.3 - un concetto fondamentale per la modellazione come grafo della blockchain: il *Labeled Property Graph* [43]. Questa formalizzazione offre flessibilità e potenza espressiva per modellare la blockchain di Bitcoin come un grafo, consentendo:

- di rappresentare nodi e archi con proprietà, etichette e tipi specifici
- di definire relazioni tra nodi e archi in modo preciso

questi aspetti hanno portato all'utilizzo del Labeled Property Graph come formalismo per definire le tre rappresentazioni della blockchain come grafo oggetto di questa tesi:

1. l'*Address-transaction graph*
2. il *Payment graph*
3. l'*Address graph con valori aggregati*

L'address-transaction graph rappresenta il punto di partenza per la rappresentazione della blockchain di Bitcoin come grafo. Questo modello utilizza nodi **Address** per rappresentare gli indirizzi Bitcoin e nodi **Transaction** per le transazioni, descrivendo il flusso di valore immesso in una transazione attraverso archi **INPUT** tra nodi **Address** e nodi **Transaction** e archi **OUTPUT**

tra nodi **Transaction** e nodi **Address** per modellare il flusso di valore inviato a indirizzi destinatari.

Sulla base dell'address-transaction graph sono poi introdotti altri due modelli: il payment graph e l'address graph con valori aggregati. Il payment graph si concentra sui singoli output delle transazioni, rappresentati come nodi TXO (Transaction Output), e su come ciascun output contribuisca (attraverso relazioni **CONTRIBUTES**) al valore degli output successivi, fornendo una visione dettagliata e sequenziale del flusso di valore. Al contrario, l'address graph con valori aggregati, caratterizzato da nodi **Address** e archi **TRANSFERS_TO**, sposta l'attenzione su proprietà aggregate e di alto livello che riassumono le transazioni che avvengono tra coppie di indirizzi, come ad esempio la proporzione totale di valore trasferito tra due indirizzi o il timestamp della prima e ultima transazione in cui si sono scambiati valuta, offrendo una visione sintetica degli scambi tra indirizzi nella rete Bitcoin.

Definizione 1 (Multigrafo). Un multigrafo orientato [56] G è una quadrupla ordinata $G = (V, A, s, t)$ dove:

- V è un insieme di nodi o vertici
- A è un insieme di archi
- $s : A \rightarrow V$ è una funzione che associa ad ogni arco $e \in A$ un nodo $s(e) \in V$ detto source node
- $t : A \rightarrow V$ è una funzione che associa ad ogni arco $e \in A$ un nodo $t(e) \in V$ detto target node

Per abuso di notazione, quando non è ambiguo, per indicare un arco e si scriverà semplicemente $e = (o, d)$ invece di $e = (s(e), t(e))$.

4.1 Labeled Property Graph

Definizione 2 (Labeled Property Graph). Un Labeled Property Graph [43] è un multi-grafo orientato $G = (V, E)$ dotato di:

- una assegnazione di label e proprietà a ciascun nodo mediante le funzioni $label : V \rightarrow \mathcal{P}(L)$ e $node_properties : V \rightarrow P_V$
- una assegnazione di tipi e proprietà agli archi mediante le funzioni $type : E \rightarrow T$ e $edge_properties : E \rightarrow P_E$

dove:

- V è l'insieme finito di nodi o vertici
- L è l'insieme finito di label o etichette dei nodi
- P_V è un insieme di valori che possono essere assunti dalle proprietà dei nodi
- E è l'insieme finito di archi o relazioni
- T è l'insieme finito dei tipi degli archi
- P_E è un insieme di valori che possono essere assunti dalle proprietà degli archi

4.2 Address-transaction graph

Definizione 3 (Address-transaction graph). Un address-transaction graph [3] è un Labeled Property Graph $G_{AT} = (V, E)$ tale che:

- l'insieme dei label è $L = \{ADDRESS, TRANSACTION\}$
- l'insieme dei valori delle proprietà dei nodi è $P_V = P_{V_A} \cup P_{V_T}$, con:
 - $P_{V_A} = \{(addressId, hash) : addressId \in \mathbb{N}, hash \in H^1\}$, che rappresenta l'insieme delle proprietà dei nodi con label *ADDRESS*
 - $P_{V_T} = \{(txId, timestamp, isCoinbase, fee, blockId) : txId \in \mathbb{N}, timestamp \in \mathbb{N}, isCoinbase \in \{true, false\}, fee \in \mathbb{N}, blockId \in \mathbb{N}\}$, che rappresenta l'insieme delle proprietà dei nodi con label *TRANSACTION*
- l'insieme dei tipi è $T = \{INPUT, OUTPUT\}$
- l'insieme dei valori delle proprietà degli archi è $P_E = P_{E_{IN}} \cup P_{E_{OUT}}$, dove:
 - $P_{E_{IN}} = \{(prevTxId, prevTxPos, amount) : prevTxId \in \mathbb{N}, prevTxPos \in \mathbb{N}, amount \in \mathbb{N}\}$ rappresenta l'insieme delle proprietà degli archi di tipo *INPUT*
 - $P_{E_{OUT}} = \{(amount, position) : amount \in \mathbb{N}, position \in \mathbb{N}\}$ rappresenta l'insieme delle proprietà degli archi di tipo *OUTPUT*

¹ H è l'insieme degli hash degli indirizzi Bitcoin.

- le relazioni di tipo *INPUT* sono coppie di nodi (a, t) dove $\text{label}(a) = \{\text{ADDRESS}\}$ e $\text{label}(t) = \{\text{TRANSACTION}\}$
- le relazioni di tipo *OUTPUT* sono coppie di nodi (t, a) dove $\text{label}(t) = \{\text{TRANSACTION}\}$ e $\text{label}(a) = \{\text{ADDRESS}\}$

In un address-transaction graph i nodi di tipo *ADDRESS* rappresentano indirizzi Bitcoin, ciascuno caratterizzato da un *hash* e un *addressId* dove il primo rappresenta l'hash dell'indirizzo contenuto nella blockchain di Bitcoin mentre il secondo l'identificatore unico di ogni indirizzo contenuto in almeno un output delle transazioni. I nodi di tipo *TRANSACTION* rappresentano transazioni Bitcoin multi-input multi-output caratterizzate da un identificativo unico *txId*, un *timestamp* che indica il momento in cui il blocco che contiene la transazione è stato minato (tempo UNIX del miner), un flag *isCoinbase* che specifica se la transazione è di tipo coinbase (ovvero una ricompensa per il miner che ha risolto la proof-of-work), una *fee* della transazione che funge da incentivo per i miner a includere la transazione nel prossimo blocco della blockchain e un *blockId* che rappresenta la distanza del blocco che contiene la transazione dal blocco genesis di Bitcoin. Gli archi di tipo *INPUT* collegano un indirizzo $a \in V_A$ a una transazione $t \in V_T$ e rappresentano la quantità di Bitcoin contribuita dall'indirizzo sorgente alla transazione. Questi archi sono caratterizzati dall'importo del trasferimento, *amount*, nonché dall'identificativo della transazione precedente da cui l'output è stato speso *prevTxId* e la posizione di quell'output nella transazione precedente *prevTxPos*. Al contrario, gli archi di tipo *OUTPUT* collegano una transazione $t \in V_T$ a un indirizzo $a \in V_A$ e rappresentano la quantità di Bitcoin assegnati all'indirizzo destinatario all'interno di quella transazione. Oltre all'importo del trasferimento, *amount*, questi archi includono anche una proprietà *position* che indica la posizione dell'output all'interno della transazione.

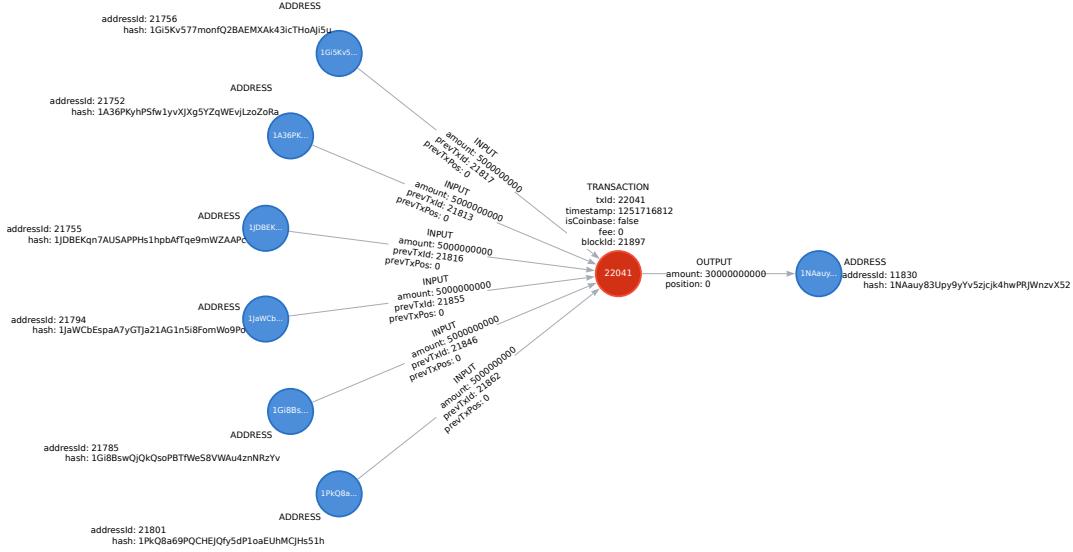


Figura 4.1: Esempio di address-transaction graph che mostra una transazione (txId 22041) con 6 indirizzi sorgente e un indirizzo destinatario, con i relativi valori trasferiti espressi in Satoshi.

4.3 Payment graph

Definizione 4 (Payment graph). Dato un address-transaction graph $G_{AT} = (V_{AT}, E_{AT})$, il payment graph $G_P = (V, E)$ associato è un Labeled Property Graph dove:

- l'insieme dei label è $L = \{TXO\}$
- l'insieme dei tipi è $T = \{CONTRIBUTES\}$
- l'insieme dei valori delle proprietà dei nodi è $P_V = \{(\text{addressId}, \text{hash}, \text{amount}, \text{position}, \text{txId}, \text{timestamp}, \text{isCoinbase}) : \text{addressId} \in \mathbb{N}, \text{hash} \in H, \text{amount} \in \mathbb{N}, \text{position} \in \mathbb{N}, \text{txId} \in \mathbb{N}, \text{timestamp} \in \mathbb{N}, \text{isCoinbase} \in \{\text{true}, \text{false}\}\}$
- l'insieme dei valori delle proprietà degli archi è $P_E = \{\}$
- per ogni arco $o = (t, d) \in E_{AT}$ di tipo *OUTPUT* che collega un nodo $t \in V_{AT}$ di label *TRANSACTION* a un nodo $d \in V_{AT}$ di label *ADDRESS* esiste un nodo $p \in V$ di label *TXO* le cui proprietà sono calcolate come $\text{node_properties}(p) = (\text{addressId}, \text{hash}, \text{amount}, \text{position}, \text{txId}, \text{timestamp}, \text{isCoinbase})$, dove:

$$(addressId, hash) = node_properties_{AT}(d)$$

$$(amount, position) = edge_properties_{AT}(o)$$

$$(txId, timestamp, isCoinbase) = node_properties_{AT}(t)$$

- ad ogni nodo $t \in V_{AT}$ e per ogni coppia di archi $i = (s, t) \in E_{AT}$ di tipo $INPUT$ e $o = (t, d) \in E_{AT}$ di tipo $OUTPUT$, dove s e d sono nodi di label $ADDRESS$ e t è un nodo di label $TRANSACTION$, corrisponde un arco $c = (p_1, p_2) \in E$ di tipo $CONTRIBUTES$ con $label(p_1) = label(p_2) = \{TXO\}$, dove:
 - Il nodo p_1 rappresenta un transaction output di una transazione precedente t_{prev} usato come input in t . Questo TXO è univocamente identificato dall'arco $o_{prev} = (t_{prev}, s) \in E_{AT}$ di tipo $OUTPUT$, dove $t_{prev}.txId = i.prevTxId \wedge o_{prev}.position = i.prevTxPos$
 - il nodo p_2 rappresenta un transaction output della transazione corrente t ed è univocamente identificato dall'arco $o = (t, d) \in E_{AT}$ di tipo $OUTPUT$

In un payment graph i nodi di label TXO rappresentano gli output delle transazioni Bitcoin e sono caratterizzati da $hash$ e $addressId$ come definito per i nodi di tipo $ADDRESS$ nell'Address-transaction graph, da $txId$, $timestamp$ e $isCoinbase$ come definito per i nodi di tipo $TRANSACTION$ nell'address-transaction graph e da $amount$ e $position$ come definito per gli archi di tipo $OUTPUT$ nell'address-transaction graph. Gli archi rappresentano il contributo di valore tra due pagamenti, ovvero se esiste un arco $c = (p_1, p_2) \in E$ di tipo $CONTRIBUTES$ allora p_1 e p_2 compaiono nella stessa transazione come input e come output rispettivamente, indicando che il valore creato dall'output p_1 viene speso per contribuire a creare il valore dell'output p_2 .

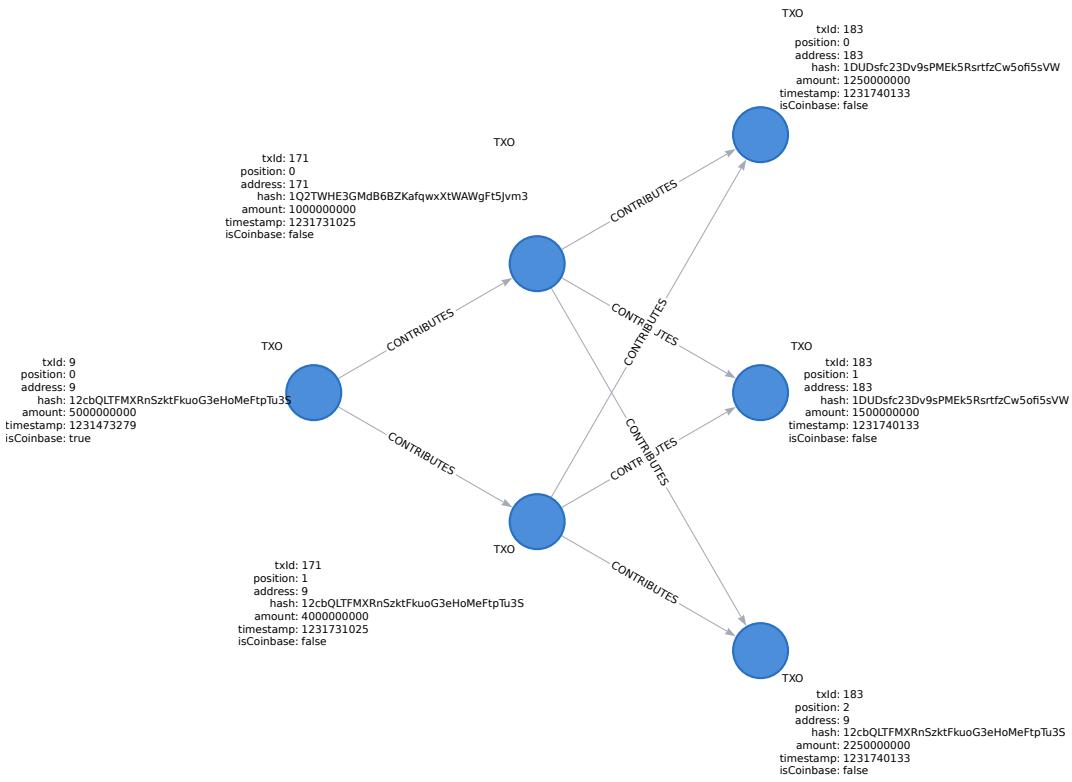


Figura 4.2: Esempio di payment graph che mostra come il valore di un transaction output (con txId 9) sia speso per creare il valore di altri transaction output, con i relativi valori trasferiti espressi in Satoshi. La catena dei pagamenti viene interrotta per brevità agli output con txId 183.

4.4 Address graph con valori aggregati

Definizione 5 (Address graph con valori aggregati). Dato un Address-transaction graph $G_{AT} = (V_{AT}, E_{AT})$, l'*Address graph con valori aggregati* $G_{AG} = (V, E)$ associato è un Labeled Property Graph dove:

- l'insieme dei nodi è $V = a \in V_{AT} : \text{label}_{AT}(a) \supset \text{ADDRESS}$
- l'insieme dei label è $L = \{\text{ADDRESS}\}$
- l'insieme dei valori delle proprietà dei nodi è $P_V = P_A$ dove P_A è l'insieme dei valori delle proprietà dei nodi *ADDRESS* del Address-transaction graph
- l'insieme dei tipi è $T = \{\text{TRANSFERS_TO}\}$

- l'insieme dei valori delle proprietà delle relazioni è $P_E = \{(sum, firstTimestamp, lastTimestamp) : sum \in \mathbb{R}_0^+, firstTimestamp \in \mathbb{N}, lastTimestamp \in \mathbb{N}\}$
- per ogni coppia di nodi s e d di label *ADDRESS* e per ogni nodo t di label *TRANSACTION* nel Address-transaction graph le proprietà della relazione $c = (s, d) \in E$ sono calcolate come $edge_properties = (sum, firstTimestamp, lastTimestamp, firstBlockId, lastBlockId, firstTxId, lastTxId)$, dove:

$$sum_{s,d} = \sum_{\substack{t \in V_{AT} | label(t)=TRANSACTION \\ i=(s,t) \in E_{AT_{IN}} \\ o=(t,d) \in E_{AT_{OUT}}}} \left(\frac{edge_properties_{AT}(i).amount}{\sum_{k=(s',t) \in E_{AT_{IN}}} edge_properties_{AT}(k).amount} \right) * edge_properties_{AT}(o).amount$$

$$firstTimestamp_{s,d} = \min_{\substack{t \in V_{AT} | label(t)=TRANSACTION \\ i=(s,t) \in E_{AT_{IN}} \\ o=(t,d) \in E_{AT_{OUT}}}} node_properties_{AT}(t).timestamp$$

$$lastTimestamp_{s,d} = \max_{\substack{t \in V_{AT} | label(t)=TRANSACTION \\ i=(s,t) \in E_{AT_{IN}} \\ o=(t,d) \in E_{AT_{OUT}}}} node_properties_{AT}(t).timestamp$$

$$firstBlockId_{s,d} = \min_{\substack{t \in V_{AT} | label(t)=TRANSACTION \\ i=(s,t) \in E_{AT_{IN}} \\ o=(t,d) \in E_{AT_{OUT}}}} node_properties_{AT}(t).blockId$$

$$lastBlockId_{s,d} = \max_{\substack{t \in V_{AT} | label(t)=TRANSACTION \\ i=(s,t) \in E_{AT_{IN}} \\ o=(t,d) \in E_{AT_{OUT}}}} node_properties_{AT}(t).blockId$$

$$firstTxId_{s,d} = \min_{\substack{t \in V_{AT} | label(t)=TRANSACTION \\ i=(s,t) \in E_{AT_{IN}} \\ o=(t,d) \in E_{AT_{OUT}}}} node_properties_{AT}(t).txId$$

$$lastTxId_{s,d} = \max_{\substack{t \in V_{AT} | label(t)=TRANSACTION \\ i=(s,t) \in E_{AT_{IN}} \\ o=(t,d) \in E_{AT_{OUT}}}} node_properties_{AT}(t).txId$$

In un address graph con valori aggregati i nodi rappresentano indirizzi Bitcoin e sono caratterizzati da *hash* e *addressId* come definito per i nodi di label *ADDRESS* nel Address-transaction graph. Gli archi di tipo *TRANSFERS_TO* rappresentano le relazioni di trasferimento di valore aggregato tra gli indirizzi: ogni arco tra due nodi *s* (sorgente) e *d* (destinatario) rappresenta un insieme aggregato di tutte le transazioni avvenute tra questi due indirizzi nel corso del tempo. Le proprietà che caratterizzano gli archi sono *sum*, che rappresenta la somma totale dei valori trasferiti tra *s* e *d* calcolata considerando proporzionalmente il contributo di ciascun input delle transazioni e la parte di valore che viene effettivamente trasferita all'indirizzo destinatario in ogni transazione, *firstTimestamp*, *lastTimestamp*, *firstBlockId*, *lastBlockId*, *firstTxId* e *lastTxId* che rappresentano rispettivamente il timestamp, il blockId e il txId della prima e dell'ultima transazione tra *s* e *d*.

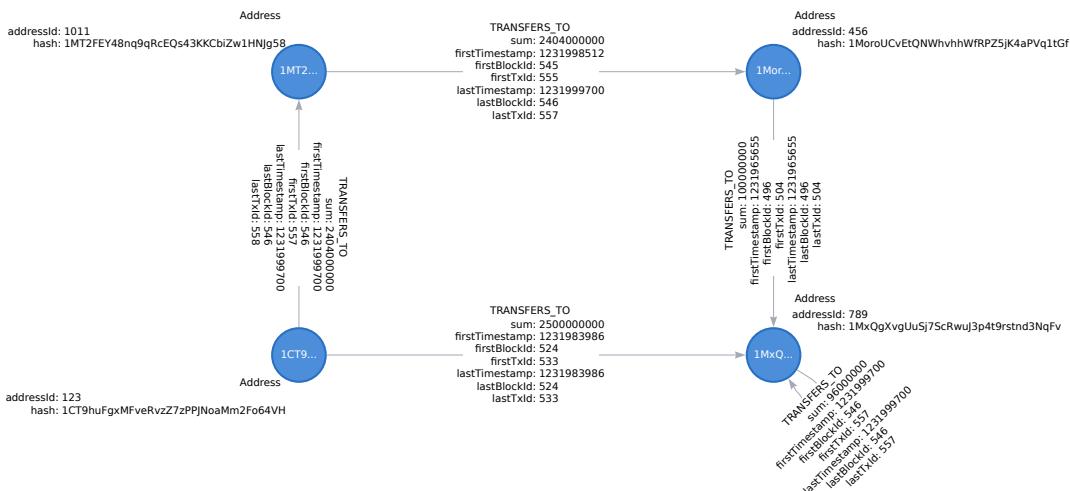


Figura 4.3: Esempio di address graph con valori aggregati, con i relativi valori trasferiti espressi in Satoshi.

4.5 Considerazioni

Ognuno dei tre modelli di grafo sopra descritti offre una prospettiva unica sulla grande e complessa rete di transazioni che costituisce la blockchain di Bitcoin e la scelta di uno rispetto agli altri dipende dal caso d'uso considerato, ossia dalle specifiche necessità analitiche e dal tipo di informazioni che si desidera ottenere:

- l'Address-transaction graph si distingue per la sua fedeltà e naturalezza nella rappresentazione delle transazioni Bitcoin, mantenendo una distinzione chiara tra indirizzi e transazioni e mappando in modo preciso il flusso di Bitcoin tra di essi, consentendo, così, una comprensione profonda degli scambi di valuta all'interno della blockchain;
- il Payment graph, al contrario, semplifica la modellazione concentrando l'attenzione sui flussi di valore piuttosto che sulle transazioni individuali. Attraverso la focalizzazione sui TXO e le relazioni di contributo, questo modello offre una rappresentazione più astratta ma anche più chiara dei movimenti di valore all'interno della blockchain. Tale semplificazione permette di ottenere grafi aciclici che possono essere analizzati più facilmente, pur aumentando le informazioni da memorizzare per ogni singolo nodo del grafo rispetto all'Address-transaction graph;
- l'Address graph con valori aggregati propone un ulteriore livello di astrazione, aggregando le transazioni per evidenziare in maniera sintetica le relazioni di valore tra coppie di indirizzi. Questo approccio riduce drasticamente la complessità visiva del grafo e facilita l'interpretazione umana delle relazioni finanziarie chiave, rendendolo ideale per analisi ad alto livello sulle dinamiche di scambio tra indirizzi. Ciononostante, l'aggregazione dei dati comporta una perdita di informazioni granulari, limitando la portata delle analisi che è possibile condurre.

In conclusione, la comprensione dei vantaggi e delle limitazioni di ciascun modello è cruciale per navigare efficacemente la complessità della blockchain Bitcoin e ottenere informazioni significative. In questo scenario, l'analisi della blockchain non è solo una questione di dati ma (*soprattutto*) di interpretazione e contestualizzazione di tali dati, e la scelta strategica del modello di grafo più adatto è un passo fondamentale per garantire che le informazioni estratte siano rilevanti, pertinenti e utili per gli obiettivi di analisi di questo vasto e intricato ecosistema finanziario.

Capitolo 5

Implementazioni in Neo4j

In questa sezione sono riportate le implementazioni in Neo4j delle rappresentazioni della blockchain di Bitcoin come grafo descritte nel capitolo 4.

Si riporta di seguito la struttura del dataset memorizzato in un file CSV contenente una transazione per riga nel formato:

`infos': 'inputs': 'outputs` dove:

- **infos** := `'timestamp', 'blockId', 'txId', 'isCoinbase', 'fee', 'approxSize'`
- **inputs** := 0 o più input separati da punto e virgola `','` dove `input := addressId', 'amount', 'prevTxId', ' prevTxPos`
- **outputs** := 1 o più output separati da punto e virgola `','` dove `output := addressId', 'amount', ' scriptType1`

Per esempio:

<code>1281006505, 72494, 99971,</code>	<code>0 , 0 , 438</code>
<i>timestamp</i>	<i>isCoinbase</i>
<code>: 93903 , 49500000000, 94374 , 0 ; 94235 , 152000000000, 97225 , 0</code>	<code>blockId</code>
<i>addressId</i>	<i>fee</i>
<i>amount</i>	<i>approxSize</i>
<i>prevTxId</i>	<i>prevTxPos</i>
<i>prevTxPos</i>	

¹Lo `scriptType` può essere dei seguenti 8 tipi: UNKNOWN=0; P2PK=1; P2PKH=2; P2SH=3; RETURN=4; EMPTY=5; P2WPKH=6; P2WSH=7

```

#1 output          #2 output
: {97138, 15000000000, 2}; {97129, 200000000000, 2}
addressId      amount    scriptType   addressId      amount    scriptType

```

Ovvero: la transazione con id 99971 è avvenuta nel blocco 72494 con timestamp 1281006505 (GMT giovedì 5 agosto 2010 11:08:25) pagando una commissione di 0 Satoshi. Ha due ingressi, uno di importo 49500000000 dall'indirizzo 93903 e uno di importo 152000000000 dall'indirizzo 94235, e due uscite che accreditano 1500000000 all'indirizzo 97138 e 200000000000 a 97129. Tutti gli importi sono in Satoshi, la più piccola unità di Bitcoin equivalente a centomilionesimi di un singolo Bitcoin (0.00000001 BTC).

Si noti che gli output non contengono esplicitamente la proprietà `position`, questa dovrà essere calcolata e aggiunta in fase di importazione nel database Neo4j in base alla loro posizione relativa.

5.1 Address-transaction graph

Prima di eseguire la query di importazione e creazione dell'address-transaction graph è utile creare degli indici per le proprietà su cui verranno eseguite frequentemente operazioni di **MERGE**. In particolare ciò verrà fatto definendo un indice per la proprietà `addressId` sui nodi `Address`:

```
1 CREATE INDEX addressId_index FOR (a:Address) ON (a.addressId);
```

Query Cypher per la realizzazione dell'address-transaction graph:

```

1 CALL apoc.periodic.iterate(
2   "LOAD CSV FROM '{{CSV_PATH}}' AS line FIELDTERMINATOR ':' RETURN line",
3   "WITH line,
4     split(line[0], ',') AS infos,
5     (CASE WHEN line[1] IS NOT NULL THEN split(line[1], ';') ELSE []
6      → END) AS inputs, // per gestire le transazioni coinbase
7     split(line[2], ',') AS outputs
8   CREATE (t:Transaction {txId: toInteger(infos[2])})
9   SET t.timestamp = toInteger(infos[0]),
10    t.blockId = toInteger(infos[1]),
11    t.isCoinbase = infos[3] = '1',
12    t.fee = toInteger(infos[4])
13   WITH t, inputs, outputs
14   // Gestione degli output

```

```

14   CALL {
15     WITH t, outputs
16     UNWIND range(0, size(outputs) - 1) AS outputIndex
17     WITH t, split(outputs[outputIndex], ',') AS output, outputIndex
18     MERGE (a:Address {addressId: toInteger(output[0])})
19     CREATE (t)-[:OUTPUT {amount: toInteger(output[1]), position:
20       ↳ outputIndex}]->(a)
21   }
22   // Gestione degli input
23   CALL {
24     WITH t, inputs
25     UNWIND inputs AS inputString
26     WITH t, split(inputString, ',') AS input
27     MERGE (a:Address {addressId: toInteger(input[0])})
28     CREATE (a)-[:INPUT {amount: toInteger(input[1]), prevTxId:
29       ↳ toInteger(input[2]), prevTxPos: toInteger(input[3])}]->(t)
30   }",
31   {batchSize: 10000, batchMode: "BATCH"}
32 )

```

La query Cypher fornita utilizza la funzione APOC `apoc.periodic.iterate` per gestire l'importazione e la creazione dell'Address-transaction graph da un dataset in formato CSV, ottimizzando il processo in batch. Questa approccio permette di dividere il lavoro in sottoinsiemi più piccoli e gestibili migliorando le prestazioni e riducendo il carico sulla memoria. Una descrizione dettagliata dei passaggi chiave della query è la seguente:

1. Caricamento dei dati

- **LOAD CSV FROM ' {{CSV_PATH}} ' AS line FIELDTERMINATOR ' : '**
RETURN line carica ogni riga del file CSV (che corrisponde a una transazione Bitcoin) come un array di stringhe, dove ogni elemento dell'array rappresenta una parte della riga di dati divisa in base al carattere ' : '. Il parametro `CSV_PATH` è un placeholder che verrà sostituito con il percorso effettivo del file CSV corrispondente al chunk di dati da importare

2. Separazione delle informazioni relative alla transazione, agli input e agli output

- la query divide ulteriormente ogni riga in tre parti principali: `infos` (informazioni generali della transazione), `inputs` (informazioni sugli input della transazione) e `outputs` (informazioni sugli output della transazione). Questo viene fatto tramite la funzione `split`, separando le stringhe basandosi sui separatori specifici (qui ',')

3. Creazione del nodo Transaction

- per ogni riga viene creato un nodo `Transaction` con le proprietà estratte da `infos`, utilizzando l'operatore `CREATE` e l'operatore `SET` per assegnare le proprietà al nodo

4. Gestione degli outputs

- per ogni output di una transazione la query itera sugli elementi dell'array `outputs` usando `UNWIND`, crea o trova un nodo `Address` corrispondente a un indirizzo² destinatario del valore della transazione usando `MERGE` e crea una relazione `OUTPUT` dal nodo `Transaction` al nodo `Address`, includendo sull'arco le proprietà di importo e posizione dell'output

5. Gestione degli inputs

- per ogni input di una transazione la query itera sugli elementi dell'array `inputs`, crea o trova un nodo `Address` corrispondente a un indirizzo mittente del valore della transazione usando `MERGE` e crea una relazione `INPUT` dal nodo `Address` al nodo `Transaction`, includendo le proprietà relative all'input sull'arco (identificativo della transazione precedente da cui proviene l'input, posizione dell'output nella transazione precedente e importo dell'input)

L'utilizzo della clausola `MERGE` consente di assicurarsi che i nodi `Address` siano creati solo se non esistono già, evitando duplicati e mantenendo così l'integrità del grafo. La scelta di impostare `batchSize: 10000` e `batchMode: "BATCH"` come parametri di configurazione per `apoc.periodic.iterate` permette di gestire il processo in batch, migliorando le prestazioni e riducendo il carico sulla memoria durante l'importazione di grandi set di dati.

5.2 Payment graph

Per rendere le operazioni di `MATCH` più efficienti su un dataset di grandi dimensioni è possibile creare un indice sulle proprietà che sono frequentemente utilizzate in queste operazioni. In questo caso è utile creare un indice Composite sulle proprietà `txId` e `position` dei nodi `TXO`:

²l'`addressId` è un identificativo intero univoco che rappresenta un indirizzo Bitcoin, ciò consente di astrarre dall'indirizzo reale (ottenuto tramite l'hashing della chiave pubblica) risparmiando spazio nel database (in quanto l'indirizzo reale è una stringa di lunghezza variabile).

```
1 CREATE INDEX txId_position_index FOR (n:TXO) ON (n.txId, n.position);
```

Query Cypher per la realizzazione del payment graph:

```
1 CALL apoc.periodic.iterate(
2   "LOAD CSV FROM '{{CSV_PATH}}' AS line FIELDTERMINATOR ':' RETURN line",
3   "WITH line,
4     split(line[0], ',') AS infos,
5     [i IN (CASE WHEN line[1] IS NOT NULL THEN split(line[1], ';') ELSE
6       [] END) | split(i, ',')] AS inputs,
7     split(line[2], ';') AS outputs
8   UNWIND range(0, size(outputs) - 1) AS outputIndex
9   WITH outputs[outputIndex] AS outputString, outputIndex, inputs, infos
10  WITH split(outputString, ',') AS output, outputIndex, inputs, infos
11  CREATE (txo:TXO {
12    txId: toInteger(infos[2]),
13    timestamp: toInteger(infos[0]),
14    isCoinbase: infos[3] = '1',
15    addressId: toInteger(output[0]),
16    amount: toInteger(output[1]),
17    position: outputIndex,
18    blockId: toInteger(infos[1])
19  })
20  WITH txo, inputs, infos
21  UNWIND inputs AS input
22  MATCH (txo_in:TXO {txId: toInteger(input[2]), position:
23    toInteger(input[3])})
24  CREATE (txo_in)-[:CONTRIBUTES]->(txo)",
25  {batchSize: 10000, batchMode: "BATCH"}
```

Il codice Cypher sopra si occupa di caricare le transazioni da un file CSV e di trasformarle nel formato del Payment graph. Questo processo si basa sul concetto, già definito formalmente nella sezione 4.3, di nodi TXO (Transaction Output) e sulla relazione CONTRIBUTES tra questi nodi, rappresentando il contributo di un output di una transazione all'input di un'altra transazione. Di seguito, una spiegazione dettagliata dei passaggi chiave della query:

1. Caricamento dei dati

- in maniera analoga alla query per la creazione dell'Address transaction graph viene caricato il file CSV e le righe vengono divise in tre parti principali: **infos** (informazioni generali della transazione), **inputs** (informazioni sugli input della transazione), e **outputs** (informazioni sugli output della transazione)

2. Creazione dei nodi TXO

- per ogni output della transazione viene creato un nodo TXO nel grafo. Ogni TXO contiene i dati dell'output (identificativo della transazione, timestamp, indirizzo destinatario, importo e posizione dell'output, identificativo del blocco) estratti dalle informazioni della transazione e dell'output

3. Gestione degli input e creazione delle relazioni CONTRIBUTES

- dopo aver creato i nodi TXO per gli output il codice procede a gestire gli input (**inputs**). Per ciascun input viene cercato nel grafo un nodo TXO corrispondente all'output di una transazione precedente (identificato da **txId** e **position**). Una volta trovato, viene creata una relazione di tipo **CONTRIBUTES** da questo nodo TXO (input della transazione corrente) al nodo TXO appena creato (output della transazione corrente)

L'utilizzo della clausola **UNWIND** permette di iterare su una lista di valori, in questo caso gli output di una transazione e per ogni output i relativi input. Come per l'Address-transaction graph, la query utilizza **apoc.periodic.iterate** per ottimizzare le prestazioni durante l'importazione di grandi set di dati.

5.3 Address graph con valori aggregati

Anche in questo caso per velocizzare le operazioni **MERGE** più efficienti, che si occupano di creare i nodi **Address** con un certo **addressId** solo se non esistono già, è stato definito un indice sulla proprietà **addressId** dei nodi **Address**:

```
1 CREATE INDEX addressId_index FOR (a:Address) ON (a.addressId);
```

Query Cypher per la realizzazione dell'address graph con valori aggregati:

```
1 CALL apoc.periodic.iterate(
2   "LOAD CSV FROM '{{CSV_PATH}}' AS line FIELDTERMINATOR ':' RETURN line",
3   "WITH line,
4     split(line[0], ',') AS infos,
5     (CASE WHEN line[1] IS NOT NULL THEN split(line[1], ';') ELSE []
6       END) AS inputs,
7     split(line[2], ';') AS outputs
```

```

7   WITH infos, inputs, outputs, toInteger(infos[2]) AS txId,
8     ↵ toInteger(infos[0]) AS currentTimestamp, toInteger(infos[1]) AS
8     ↵ blockId,
8   // Calcolo la somma totale degli importi degli input una volta per
8     ↵ transazione
9   REDUCE(s = 0.0, i IN inputs | s + toInteger(split(i, ',')[1])) AS
8     ↵ totalInputAmount
10  CALL {
11    WITH inputs, outputs, totalInputAmount
12    UNWIND outputs AS output
13    UNWIND inputs AS input
14    WITH input, output, totalInputAmount,
15      split(input, ',')[0] AS srcAddrId, toInteger(split(input,
16        ↵ ',')[1]) AS inputAmount,
16      split(output, ',')[0] AS dstAddrId, toInteger(split(output,
17        ↵ ',')[1]) AS outputAmount
17    MERGE (src:Address {addressId: toInteger(srcAddrId)})
18    MERGE (dst:Address {addressId: toInteger(dstAddrId)})
19    CREATE (src)-[tempTransfers:TEMP_TRANSFERS {amount: (outputAmount
20      ↵ * (inputAmount / totalInputAmount))}]->(dst)
20  RETURN src, dst, tempTransfers
21  // restituisce ogni possibile coppia di indirizzi sorgente e
21    ↵ destinazione insieme a tutti gli archi TEMP_TRANSFERS che li
21    ↵ collegano
22  }
23  WITH src, dst, COLLECT(tempTransfers) AS tempTransfers, txId,
24    ↵ currentTimestamp, blockId
24  MERGE (src)-[transfers:TRANSFERS_TO]->(dst)
25  ON CREATE SET transfers.sum = REDUCE(s = 0.0, t IN tempTransfers | s +
25    ↵ t.amount),
26  transfers.firstTxId = txId,
27  transfers.firstTimestamp = currentTimestamp,
28  transfers.firstBlockId = blockId,
29  transfers.lastTxId = txId,
30  transfers.lastTimestamp = currentTimestamp,
31  transfers.lastBlockId = blockId
32  ON MATCH SET transfers.sum = transfers.sum + REDUCE(s = 0.0, t IN
32    ↵ tempTransfers | s + t.amount),
33  // basta un assegnamento per lastTimestamp perchè le transazioni sono
33    ↵ ordinate temporalmente in modo crescente
34    transfers.lastTimestamp = currentTimestamp,
35          transfers.lastTxId = txId,
36          transfers.lastBlockId = blockId
37  WITH tempTransfers
38  UNWIND tempTransfers AS tempTransfer
39  DELETE tempTransfer",
40  {batchSize: 10000, batchMode: "BATCH"}}

```

41

)

Questo codice Cypher si occupa di costruire un Address graph con valori aggregati a partire da un dataset in formato CSV contenente dati relativi a transazioni Bitcoin. Il grafo risultante presenta nodi **Address** e relazioni **TRANSFERS_TO** tra questi nodi, rappresentando in maniera aggregata i trasferimenti di valore tra indirizzi. I passaggi chiave della query sono:

1. Caricamento dei dati

- come nel caso dell'Address-transaction graph e del Payment graph viene effettuata come prima operazione la lettura del file CSV e per ciascuna riga avviene la divisione in tre parti principali: **infos** (informazioni generali della transazione), **inputs** (informazioni sugli input della transazione), e **outputs** (informazioni sugli output della transazione)

2. Calcolo dell'importo totale degli input per transazione

- per ogni transazione viene calcolata la somma totale degli importi degli input utilizzando la funzione **REDUCE**, questa somma verrà utilizzata per calcolare la proporzione di valore che ogni mittente contribuisce a ciascun destinatario

3. Creazione delle relazioni **TEMP_TRANSFERS**

- per ogni coppia di indirizzi sorgente e destinazione la query crea una relazione **TEMP_TRANSFERS** con un attributo **amount** che rappresenta l'importo trasferito proporzionalmente alla somma totale degli input

4. Creazione o aggiornamento delle relazioni **TRANSFERS_TO**

- per ogni coppia di indirizzi sorgente e destinazione coinvolta nella transazione corrente si crea o si aggiorna una relazione **TRANSFERS_TO** tra i corrispondenti nodi **Address**. Se la relazione non esiste, viene creata e vengono impostate le proprietà relative al primo e all'ultimo trasferimento (identificativo della transazione, timestamp, blocco) oltre che la somma aggregata degli importi trasferiti. Se la relazione esiste già, vengono aggiornate solo le proprietà relative all'ultimo trasferimento e la somma aggregata degli importi trasferiti

5. Eliminazione delle relazioni temporanee **TEMP_TRANSFERS**

5.4. PROCEDURA PER AUTOMATIZZARE L'IMPORTAZIONE ED ANALISI DEI GRAFI

- una volta completata la creazione o l'aggiornamento delle relazioni TRANSFERS_TO le corrispondenti relazioni temporanee TEMP_TRANSFERS tra gli indirizzi coinvolti nella transazione corrente vengono eliminate

5.4 Procedura per automatizzare l'importazione ed analisi dei grafi costruiti

È stato realizzato uno script Bash al fine di automatizzare per ciascuna delle tre rappresentazioni della blockchain di Bitcoin come grafo i processi di:

1. importazione di chunk della blockchain di Bitcoin corrispondenti a sei mesi di transazioni
2. salvataggio dei risultati relativi all'importazione
3. memorizzazione della nuova dimensione totale del database
4. esecuzione delle query di analisi sul database appena aggiornato all'ultimo chunk e salvataggio dei risultati

Tale script si occupa, attraverso l'utilizzo del comando `cypher-shell` per inviare query Cypher al database Neo4j e altri comandi Bash, di eseguire tutte le operazioni necessarie per automatizzare questo processo. Lo script può essere configurato per considerare solo un sottoinsieme dei chunk della blockchain di Bitcoin, in modo da poter eseguire l'importazione e l'analisi su un periodo di tempo specifico se desiderato. Il codice dello script è stato omesso per brevità ma può essere visualizzato nella repository GitHub³ contenente il codice utilizzato per lo svolgimento della tesi.

³<https://github.com/leonardo-arditti/codice-tesi>

Capitolo 6

Analisi definite sui grafi

In questo capitolo è presentata l'implementazione di tre query di analisi applicate alle diverse rappresentazioni della blockchain di Bitcoin come grafo. Ciascuna delle rappresentazioni considerate, ovvero Address-transaction graph, Payment graph e Address graph con valori aggregati, presenta delle caratteristiche specifiche che influenzano la formulazione delle query e la loro implementazione. In particolare, le query di analisi considerate sono le seguenti:

1. timestamp della prima transazione in cui è coinvolto un indirizzo α come
 - (a) mittente
 - (b) destinatario
2. dimensione dell'ego-network (diretta) di un indirizzo α , cioè il numero di nodi che hanno ricevuto valore direttamente da α

Sono state scelte queste query non solo per la loro capacità di offrire intuizioni significative sulle dinamiche della rete Bitcoin. Ad esempio, conoscere il momento in cui un indirizzo ha iniziato ad essere attivo può aiutare a comprendere la sua storia e il suo ruolo all'interno della rete. Analogamente, analizzare l'ego-network di un indirizzo può rivelare il suo grado di centralità e influenza. Ma anche perché mostrano due tipi diversi di analisi, la prima basata sul recupero di informazioni dipendenti sia da indirizzi che transazioni, e la seconda basata sulla topologia delle relazioni tra indirizzi attraverso transazioni.

6.1 Selezione degli indirizzi per le analisi

Per garantire un'analisi accurata e dettagliata è stata adottata una strategia di campionamento che coinvolge la selezione casuale di $n = 1000$ indirizzi,

coinvolti come mittenti in almeno una transazione, per ogni fase di importazione di chunk della blockchain. Questi indirizzi sono stati estratti utilizzando l'Address-transaction graph e salvati su file con la nomenclatura `randomAddresses_up_to_chunk_x_blockId_y.txt`. La scelta dell'Address-transaction graph per questa operazione è arbitraria e non influisce sulla natura casuale della selezione degli indirizzi. Si ricorda, in accordo con le definizioni formali definite nel capitolo 4, che gli indirizzi sono trattati come identificatori numerici univoci (`addressId`) per facilitare l'astrazione dalle stringhe alfanumeriche che rappresentano gli indirizzi Bitcoin reali.

La query Cypher che consente di individuare e salvare su file gli $n = 1000$ indirizzi casuali è la seguente:

```

1  // 1.
2  MATCH (a:Sampled)
3  REMOVE a:Sampled;
4
5  // 2.
6  CALL apoc.periodic.iterate(
7      "MATCH (a:Address)-[:INPUT]->(:Transaction)
8      WHERE NOT a:Input
9      RETURN a",
10     "SET a:Input",
11     {batchSize: 10000}
12 );
13
14 // 3.
15 CALL {
16     MATCH (i:Input)
17     WITH count(i) AS num_input
18     CALL apoc.periodic.commit(
19         "MATCH (i:Input)
20         WITH i
21         SKIP toInteger(floor($num_input*rand()))
22         LIMIT 1
23         SET i:Sampled
24         WITH 1 AS dummy
25         MATCH (n:Sampled)
26         RETURN 1000 - count(n)",
27         {num_input: num_input})
28     YIELD updates, executions
29     RETURN updates, executions
30 }
31
32 // 4.

```

```

33 CALL apoc.export.csv.query(
34   "MATCH (a:Sampled)
35     WITH collect(a.addressId) as randomAddresses
36     UNWIND randomAddresses AS randomAddress
37     RETURN randomAddress",
38   "randomAddresses_chunk_x.txt",
39   {quotes: "none"}
40 ) YIELD file
41 RETURN "Generazione indirizzi completata su file"

```

La query precedente non è altro che la combinazione di 4 query che svolgono ciascuna un compito specifico. In particolare:

1. Rimozione dell'etichetta Sampled

- Il codice inizia rimuovendo l'etichetta `Sampled` da tutti i nodi che la posseggono. I nodi `Sampled` sono i nodi `Address` coinvolti come mittenti in almeno una transazione e selezionati casualmente in iterazioni precedenti. Questo passaggio è necessario per garantire che la selezione degli indirizzi casuali avvenga su un sottoinsieme di nodi non influenzato da selezioni precedenti

2. Etichettatura dei nuovi nodi `Address` con l'etichetta `Input`

- Viene aggiunta l'etichetta `Input` ai nodi `Address` coinvolti come mittenti in almeno una transazione che non hanno già l'etichetta. Questa operazione viene effettuata utilizzando la procedura `apoc.periodic.iterate` per ottimizzare le prestazioni

3. Selezione casuale di $n = 1000$ indirizzi

- `apoc.periodic.commit` consente di selezionare casualmente $n = 1000$ nodi `Address` etichettati come `Input`. In ogni esecuzione un nodo `Input` è selezionato casualmente tra tutti i nodi `Input` presenti nel grafo con (`SKIP toInteger(floor($num_input*rand()))` `LIMIT 1`) e successivamente viene etichettato come `Sampled`. Questo processo continua finché $1000 - \text{count}(n)$ non restituisce 0, ovvero finché non sono stati selezionati $n = 1000$ nodi etichettati come `Sampled`. Una volta restituito 0, `apoc.periodic.commit` termina

4. Salvataggio degli indirizzi selezionati su file

- Infine, gli indirizzi selezionati vengono salvati in un file CSV tramite `apoc.export.csv.query`. La query estrae l'`addressId` di tutti i nodi con l'etichetta `Sampled` e li salva nel file specificato per utilizzarli nelle analisi successive

Il placeholder del file `randomAddresses_chunk_x.txt` nel codice sarà sostituito durante l'esecuzione dello script Bash di automazione del processo di importazione e analisi con il nome reale del file che riflette il numero del chunk e l'ultimo `blockId` processato, seguendo il formato `randomAddresses_up-to_chunk_x_blockId_y.txt`. Questo meccanismo di sostituzione permette di tenere traccia degli indirizzi casuali selezionati in relazione all'ultimo chunk importato e al `blockId` finale del chunk.

L'esecuzione della query soprastante avviene in maniera iterativa ad ogni importazione di un nuovo chunk della blockchain di Bitcoin. In questo modo, per ogni nuovo chunk importato, vengono selezionati $n = 1000$ nuovi indirizzi casuali coinvolti come mittenti in almeno una transazione e salvati su file per essere utilizzati nelle successive analisi.

6.2 Indici definiti

Gli indici attualmente presenti per ciascuna delle rappresentazioni della blockchain di Bitcoin come grafo sono i seguenti:

1. Address-transaction graph
 - `addressId` per i nodi `Address`
2. Payment graph
 - la coppia `(txId, position)` per i nodi `TXO`
3. Address graph con valori aggregati
 - `addressId` per i nodi `Address`

Per poter velocizzare le query di analisi, che richiedono l'accesso puntuale a nodi con specifici `addressId`, nel caso del Payment graph è stato creato un ulteriore indice per la proprietà `addressId` dei nodi `TXO`. Questo indice permette di accedere direttamente ai nodi `TXO` in base all' `addressId` senza dover eseguire una scansione completa del grafo per trovare i nodi desiderati.

6.3 Timestamp della prima transazione con indirizzo α come mittente

Per rispondere a questa richiesta¹ è necessario un approccio differente a seconda della rappresentazione della blockchain considerata. In particolare:

- per l'Address-transaction graph è sufficiente seguire le relazioni INPUT che collegano ciascun nodo Address scelto casualmente ad una transazione e poi utilizzare la proprietà timestamp della transazione per trovare il timestamp più piccolo;
- per il Payment graph è necessario individuare tutti i nodi TXO che corrispondono all'indirizzo casuale letto da file. Una volta individuati, basta seguire le relazioni CONTRIBUTES che collegano un output (TXO) utilizzato come input in una transazione con gli output della transazione stessa. Infine, per ciascun nodo TXO a cui è collegato l'output dell'indirizzo casuale, si può trovare il timestamp accedendo alla proprietà timestamp e selezionare il valore minimo usando la funzione MIN;
- per l'Address graph con valori aggregati è necessario seguire le relazioni TRANSFERS_TO che collegano ciascun nodo Address scelto casualmente ai suoi vicini e poi ricavare il timestamp più piccolo tra tutte le proprietà firstTimestamp che caratterizzano gli archi che collegano i nodi.

6.3.1 Address-transaction graph

```

1 LOAD CSV WITH HEADERS FROM 'file:///randomAddresses_chunk_x.txt' AS line
2 MATCH (a:Address {addressId:
  → toInteger(line.randomAddress)})-[:INPUT]->(t:Transaction)
3 RETURN a.addressId AS scrAddressId, MIN(t.timestamp) AS
  → FirstTransactionTimestamp

```

Codice 31: Query 1a per l'address-transaction graph.

¹ricordare in questa analisi e nella successiva in sezione 6.4 che la proprietà timestamp per tutte le rappresentazioni della blockchain Bitcoin come grafo corrisponde al timestamp del blocco in cui si trova la transazione.

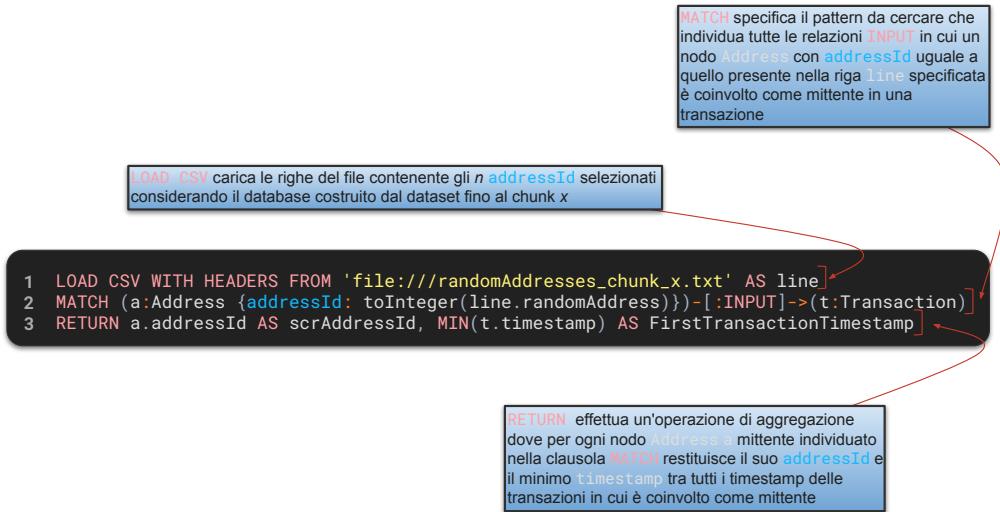


Figura 6.1: Spiegazione della query 1a per l'address-transaction graph.

6.3.2 Payment graph

```

1 LOAD CSV WITH HEADERS FROM 'file:///randomAddresses_chunk_x.txt' AS line
2 MATCH (startTX0:TX0 {addressId:
  ↵ toInteger(line.randomAddress)})-[:CONTRIBUTES]->(endTX0:TX0)
3 RETURN startTX0.addressId AS srcAddressId, MIN(endTX0.timestamp) AS
  ↵ FirstTransactionTimestamp
  
```

Codice 32: Query 1a per il payment graph.

6.3. TIMESTAMP DELLA PRIMA TRANSAZIONE CON INDIRIZZO α COME MITTENTE

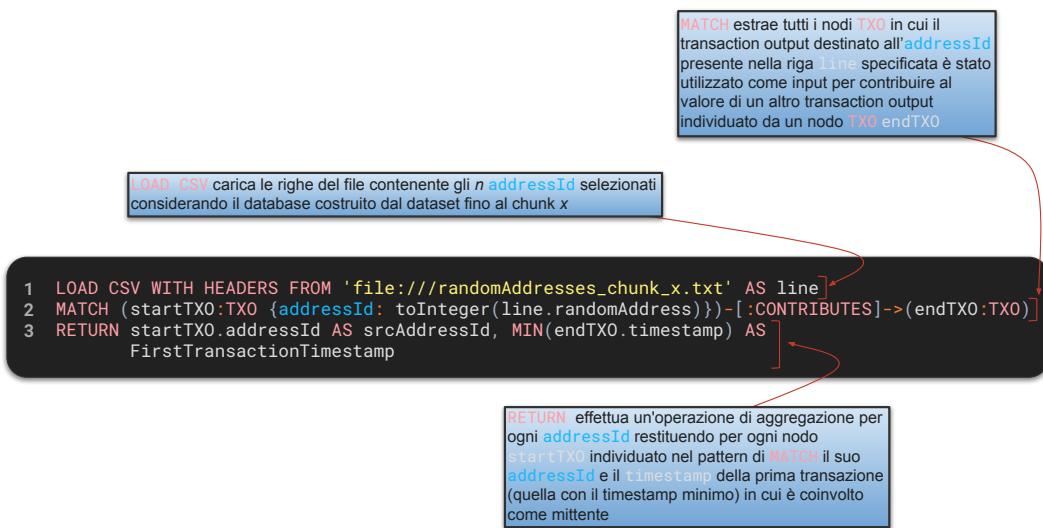


Figura 6.2: Spiegazione della query 1a per il payment graph.

6.3.3 Address graph con valori aggregati

```

1 LOAD CSV WITH HEADERS FROM 'file:///randomAddresses_chunk_x.txt' AS line
2 MATCH (src:Address {addressId:
  ↳ toInteger(line.randomAddress)})-[r:TRANSFERS_TO]->(:Address)
3 RETURN src.addressId AS srcAddressId, MIN(r.firstTimestamp) AS
  ↳ FirstTransactionTimestamp

```

Codice 33: Query 1a per l'address graph con valori aggregati.

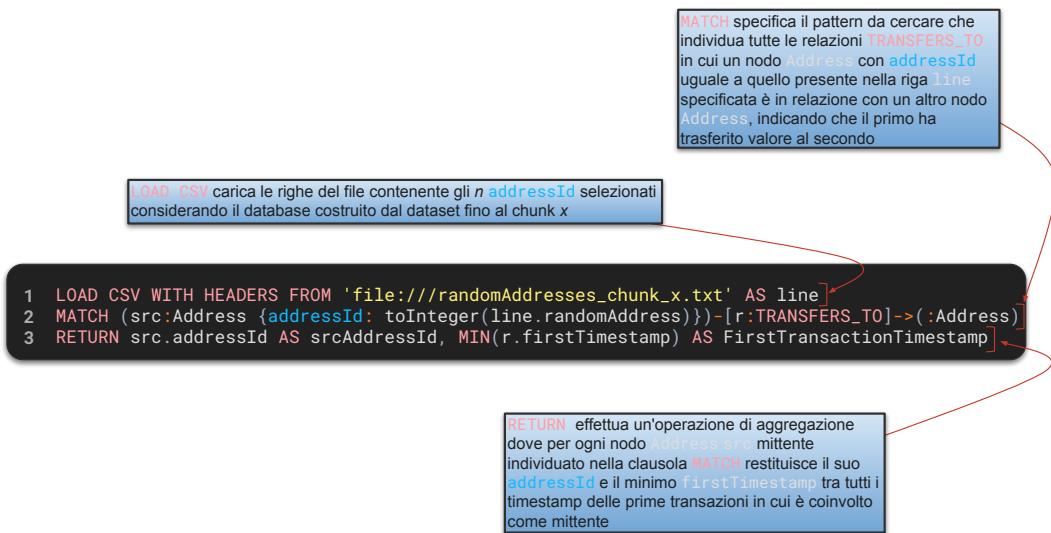


Figura 6.3: Spiegazione della query 1a per l'address graph con valori aggregati.

6.4 Timestamp della prima transazione con indirizzo α come destinatario

Rispetto alla query precedente sono limitate le modifiche da apportare a ciascuna delle query per le 3 rappresentazioni, in particolare:

- per l'Address-transaction graph è sufficiente seguire le relazioni `OUTPUT` che collegano ciascun nodo `Transaction` ad ogni nodo `Address` scelto casualmente, poi utilizzare la proprietà `timestamp` della transazione per trovare il timestamp più piccolo;
- per il Payment graph dato che ciascun nodo rappresenta un output di una transazione destinato ad un indirizzo, è sufficiente leggere la proprietà `timestamp` dei nodi `TXO` di proprietà dell'indirizzo casuale e selezionare il valore minimo usando la funzione `MIN`;
- per l'Address graph con valori aggregati è necessario seguire le relazioni `TRANSFERS_TO` entranti in ciascun nodo `Address` corrispondente all'indirizzo casuale e, successivamente, ricavare il timestamp più piccolo tra tutte le proprietà `firstTimestamp` che caratterizzano gli archi entranti.

6.4. TIMESTAMP DELLA PRIMA TRANSAZIONE CON INDIRIZZO α COME DESTINATA

6.4.1 Address-transaction graph

```

1 LOAD CSV WITH HEADERS FROM 'file:///randomAddresses_chunk_x.txt' AS line
2 MATCH (t:Transaction)-[:OUTPUT]->(a:Address {addressId:
   ↳ toInteger(line.randomAddress)})
3 RETURN a.addressId AS dstAddressId, MIN(t.timestamp) AS
   ↳ FirstTransactionTimestamp

```

Codice 34: Query 1b per l'address-transaction graph.



Figura 6.4: Spiegazione della query 1b per l'address-transaction graph.

6.4.2 Payment graph

```

1 LOAD CSV WITH HEADERS FROM 'file:///randomAddresses_chunk_x.txt' AS line
2 MATCH (endTX0:TXO {addressId: toInteger(line.randomAddress)})
3 RETURN endTX0.addressId AS dstAddressId, MIN(endTX0.timestamp) AS
   ↳ FirstTransactionTimestamp

```

Codice 35: Query 1b per il payment graph.

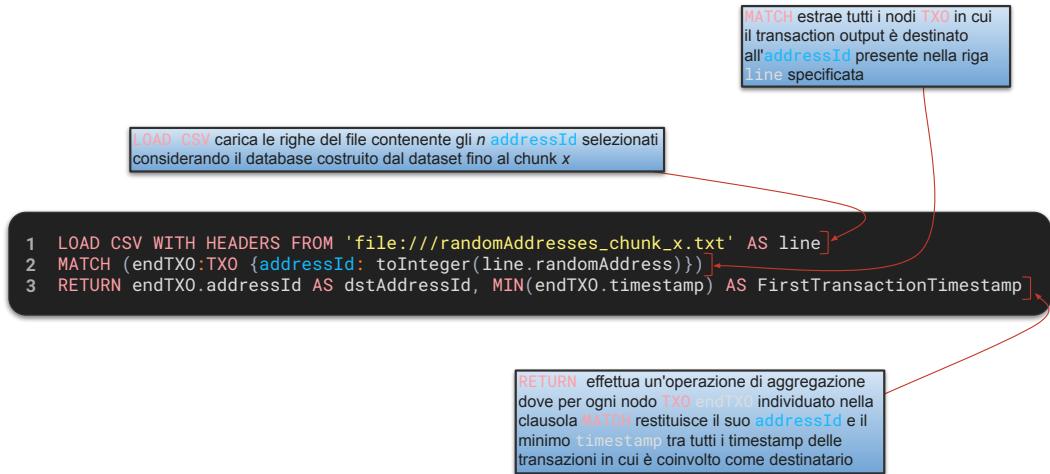


Figura 6.5: Spiegazione della query 1b per il payment graph.

6.4.3 Address graph con valori aggregati

```

1 LOAD CSV WITH HEADERS FROM 'file:///randomAddresses_chunk_x.txt' AS line
2 MATCH (:Address)-[r:TRANSFERS_TO]->(dst:Address {addressId:
   ↵ toInteger(line.randomAddress)})
3 RETURN dst.addressId AS dstAddressId, MIN(r.firstTimestamp) AS
   ↵ FirstTransactionTimestamp
  
```

Codice 36: Query 1b per l'address graph con valori aggregati.

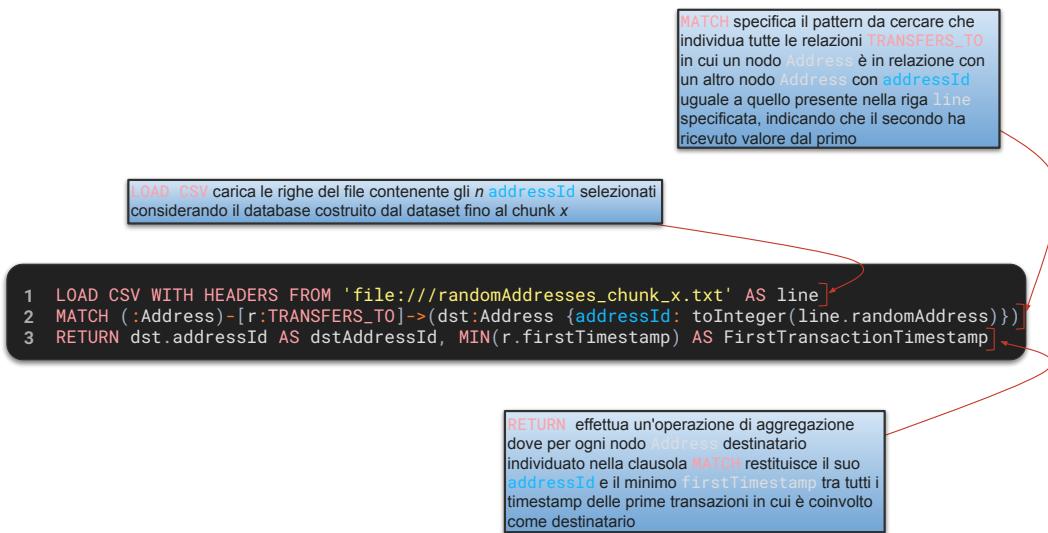


Figura 6.6: Spiegazione della query 1b per l'address graph con valori aggregati.

6.5 Dimensione dell'ego-network di un indirizzo

Per il calcolo della dimensione dell'ego-network, ovvero il numero di indirizzi unici che hanno ricevuto valore da un determinato indirizzo, è necessario:

- per l'Address-transaction graph individuare tutti i nodi **Address** che sono destinatari di valore in transazioni in cui l'indirizzo casuale è coinvolto come mittente. Successivamente, è necessario escludere le transazioni in cui l'indirizzo casuale è coinvolto come destinatario e, infine, contare i nodi **Address** distinti che hanno ricevuto valore dall'indirizzo casuale;
- per il Payment graph vanno individuati tutti i transaction output (nodi **TXO**) che appartengono all'indirizzo casuale letto da file. Una volta individuati, è necessario seguire le relazioni **CONTRIBUTES** che collegano un output (**TXO**) utilizzato come input in una transazione con gli output della transazione stessa. Infine, una volta individuati tutti i nodi a cui è collegato l'output dell'indirizzo casuale, è sufficiente contare i nodi **TXO** con **addressId** distinti e diversi da quello dell'indirizzo del mittente;
- per l'Address graph con valori aggregati basta seguire le relazioni **TRANSFERS_TO** uscenti dal nodo **Address** corrispondente all'indirizzo casuale e, successivamente, contare i nodi **Address** a cui l'indirizzo casuale

è collegato mediante un arco. Anche in questo caso è necessario escludere il nodo **Address** corrispondente all'indirizzo casuale mittente tra i destinatari del valore.

6.5.1 Address-transaction graph

```

1 LOAD CSV WITH HEADERS FROM 'file:///randomAddresses_chunk_x.txt' AS line
2 MATCH (a:Address {addressId: toInteger(line.randomAddress)})
3 MATCH (a)-[:INPUT]->(:Transaction)-[:OUTPUT]->(dst:Address)
4 WHERE a.addressId <> dst.addressId
5 RETURN a.addressId AS srcAddressId, COUNT(DISTINCT dst) AS EgoNetworkSize

```

Codice 37: Query 2 per l'address-transaction graph.

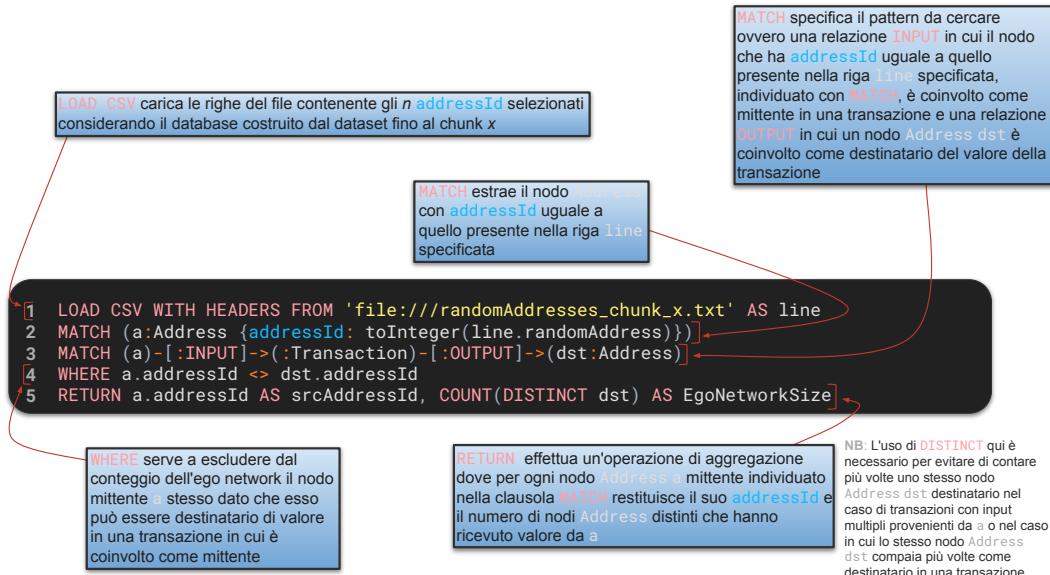


Figura 6.7: Spiegazione della query 2 per l'address-transaction graph.

6.5.2 Payment graph

```

1 LOAD CSV WITH HEADERS FROM 'file:///randomAddresses_chunk_x.txt' AS line
2 MATCH (startTX0:TX0 {addressId: toInteger(line.randomAddress)})
3 MATCH (startTX0)-[:CONTRIBUTES]->(endTX0:TX0)
4 WHERE startTX0.addressId <> endTX0.addressId
5 RETURN startTX0.addressId AS srcAddressId, COUNT(DISTINCT
    ↵ endTX0.addressId) AS EgoNetworkSize

```

Codice 38: Query 2 per il payment graph.

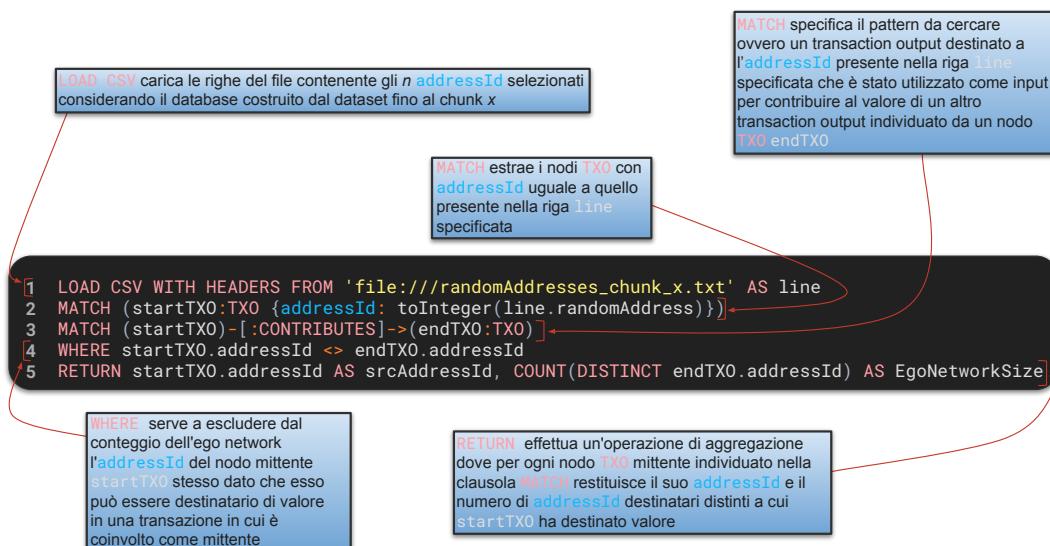


Figura 6.8: Spiegazione della query 2 per il payment graph.

6.5.3 Address graph con valori aggregati

```

1 LOAD CSV WITH HEADERS FROM 'file:///randomAddresses_chunk_x.txt' AS line
2 MATCH (src:Address {addressId: toInteger(line.randomAddress)})
3 MATCH (src)-[:TRANSFERS_TO]->(dst:Address)
4 WHERE src.addressId <> dst.addressId
5 RETURN src.addressId AS srcAddressId, COUNT(dst) AS EgoNetworkSize

```

Codice 39: Query 2 per l'address graph con valori aggregati.

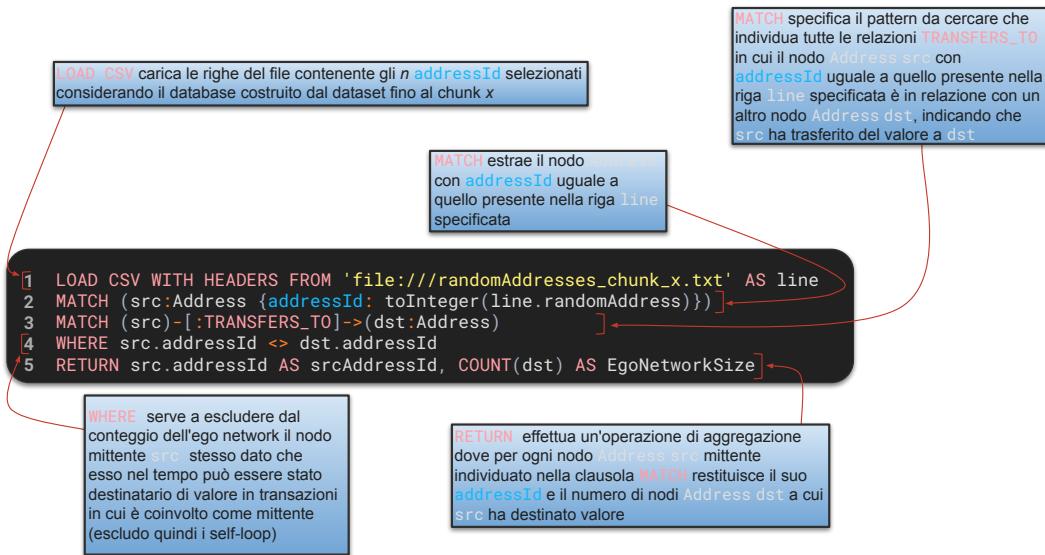


Figura 6.9: Spiegazione della query 2 per l'address graph con valori aggregati.

Capitolo 7

Valutazione sperimentale

In questo capitolo viene presentata una valutazione sperimentale delle tre rappresentazioni della blockchain di Bitcoin come grafo, già descritte formalmente nel capitolo 4 e implementate in Neo4j con Cypher come delineato nel capitolo 5, ovvero:

1. Address-transaction graph, dove si hanno nodi **Address** (che rappresentano indirizzi Bitcoin) collegati a nodi **Transaction** mediante archi **INPUT** e/o **OUTPUT** a seconda del ruolo dell’indirizzo nella transazione;
2. Payment graph, dove si hanno nodi **TXO** (che rappresentano output di transazioni) collegati tra loro mediante archi **CONTRIBUTES** per indicare che il valore di un output di una transazione è stato utilizzato come input per contribuire a creare valore di un insieme di altri output;
3. Address graph con valori aggregati, dove si hanno nodi **Address** (la cui definizione è identica a quella data dall’Address-transaction graph) collegati tra loro mediante archi **TRANSFERS_TO** caratterizzati da proprietà aggregate come definito nella sezione 4.4.

Il periodo considerato della blockchain di Bitcoin è quello che va da gennaio 2009 ad agosto 2017. Durante questo arco temporale Bitcoin è passato da essere un semplice esperimento a una realtà consolidata nell’ambito delle criptovalute, con una crescita esponenziale sia in termini di transazioni che di partecipanti alla rete. Per gestire e analizzare efficacemente il vasto volume di dati, che rende Bitcoin un caso concreto di “Big Data”, è stato diviso l’intero dataset testuale originale di circa 40GB (la cui struttura è già stata riportata in dettaglio nel capitolo 5), in 19 chunk contenenti ciascuno 25920 blocchi (corrispondenti a circa sei mesi di attività, ipotizzando un tempo medio per la creazione di un blocco di 10 minuti e mesi di 30 giorni). Questo approccio

ha permesso di osservare come le dinamiche della rete e le prestazioni delle diverse rappresentazioni della blockchain di Bitcoin come grafo evolvono nel tempo, riflettendo la transizione di Bitcoin da un interesse di nicchia a un sistema globale di pagamento digitale.

Nell'ambito di questa tesi sono presentati una serie di esperimenti condotti per valutare l'impatto della scelta della rappresentazione sulla dimensione del database, sulla velocità di importazione e sui tempi di risposta delle query. In particolare sono considerate:

1. dimensione del database e tempo di importazione: viene esaminata l'evoluzione della dimensione complessiva del database e il tempo necessario per l'importazione di ciascun chunk di dati nelle tre rappresentazioni. L'analisi mette in luce l'impatto delle diverse rappresentazioni dei grafici sulla dimensione dei dati, fornendo informazioni utili su efficienza e scalabilità;
2. evoluzione del numero di nodi e archi: viene approfondita l'evoluzione temporale del numero di nodi e archi importati in ciascun chunk, mettendo in luce le variazioni nelle dimensioni del grafo e le differenze tra le rappresentazioni;
3. tempi di esecuzione delle query: valutazione dei tempi di esecuzione delle specifiche query 1a, 1b e 2 introdotte e motivate nel capitolo 5, su ogni rappresentazione della blockchain come grafo. Per ogni esperimento sono stati di volta in volta selezionati casualmente $n = 1000$ indirizzi tra quelli presenti nel database fino all'ultimo chunk importato e sono stati registrati i tempi di esecuzione delle query, permettendo di fare un benchmark delle prestazioni durante l'evoluzione del grafo.

Tutti gli esperimenti sono stati condotti utilizzando Neo4j Enterprise Edition 5.18.0 su un server con le seguenti specifiche:

- Sistema operativo: Ubuntu 22.04.02 LTS
- Processore: Intel(R) Xeon(R) Gold 5218 CPU @ 2.30GHz, 8 core
- RAM: 256 GB
- Storage disponibile ai fini degli esperimenti: 750 GB

Chunk	Intervallo di blocchi	Numero di transazioni
1	0 - 25 919	26 085
2	25 920 - 51 839	29 954
3	51 840 - 77 759	54 543
4	77 760 - 103 679	127 943
5	103 680 - 129 599	432 335
6	129 600 - 155 519	1 278 770
7	155 520 - 181 439	1 475 950
8	181 440 - 207 359	5 333 215
9	207 360 - 233 279	8 042 066
10	233 280 - 259 199	7 373 953
11	259 200 - 285 119	8 464 961
12	285 120 - 311 039	10 020 372
13	311 040 - 336 959	12 851 681
14	336 960 - 362 879	18 127 755
15	362 880 - 388 799	24 602 092
16	388 800 - 414 719	35 245 375
17	414 720 - 440 639	40 433 329
18	440 640 - 466 559	48 744 575
19	466 560 - 479 969	22 745 129

Tabella 7.1: Associazione tra chunk, intervallo di blocchi considerato e numero di transazioni presenti in ciascun chunk.

Rappresentazione	Numero totale di nodi	Numero totale di archi
Address-transaction graph	539 208 251	1 283 445 382
Payment graph	668 261 953	2 162 523 341
Address graph con valori aggregati	293 725 439	1 537 624 607

Tabella 7.2: Numero totale di nodi e archi presenti in ciascuna rappresentazione della blockchain di Bitcoin come grafo.

Chunk	Address-transaction graph	Payment graph	Address graph con valori aggregati
1	15 MB	3.2 MB	1.1 MB
2	19 MB	12 MB	4.2 MB
3	49 MB	26 MB	17 MB
4	100 MB	55 MB	36 MB
5	311 MB	186 MB	243 MB
6	930 MB	738 MB	866 MB
7	1.7 GB	1.3 GB	1.5 GB
8	4.2 GB	3.3 GB	3.1 GB
9	7.9 GB	6.7 GB	5.6 GB
10	11 GB	9.7 GB	8.3 GB
11	16 GB	14 GB	14 GB
12	22 GB	22 GB	28 GB
13	29 GB	32 GB	44 GB
14	40 GB	45 GB	65 GB
15	55 GB	62 GB	89 GB
16	74 GB	81 GB	114 GB
17	93 GB	98 GB	136 GB
18	117 GB	119 GB	165 GB
19	130 GB	131 GB	184 GB

Tabella 7.3: Evoluzione della dimensione totale del database per ciascuna rappresentazione della blockchain di Bitcoin come grafo al variare del numero di chunk importati.

Chunk	Address-transaction graph	Payment graph	Address graph con valori aggregati
1	1	<1	<1
2	1	1	1
3	3	2	4
4	9	6	12
5	48	41	75
6	176	159	317
7	196	185	592
8	667	659	1880
9	1067	1131	2457
10	1044	1076	2275
11	1390	1641	1641
12	1953	3093	8181
13	2535	3828	9611
14	3618	5268	13 518
15	5073	7091	18 267
16	6715	8239	30 542
17	7318	7961	33 091
18	8608	9442	31 768
19	4336	5462	22 260

Tabella 7.4: Tempo di importazione di ciascun chunk (in secondi) per le tre rappresentazioni della blockchain di Bitcoin come grafo.

I risultati sperimentali presentati nelle tabelle 7.1, 7.3, 7.2 e 7.4 precedenti forniscono una visione dettagliata e quantitativa dello sviluppo della rete Bitcoin come grafo, evidenziando le differenze tra le tre rappresentazioni e le relative prestazioni.

La tabella 7.1 mostra il numero di transazioni presenti in ciascun chunk per l'intero periodo considerato ed è evidente come Bitcoin abbia avuto un'adozione sempre maggiore nel tempo. Si inizia con un numero relativamente basso di transazioni per i primi 6 mesi (circa 26 mila transazioni) per poi arrivare, nonostante si tratti di un periodo incompleto di due mesi conclusivo del dataset anziché sei, a un numero di transazioni di circa 22.7 milioni nell'ultimo chunk, segnalando così un'intensa attività di scambio e una trasformazione di Bitcoin da semplice curiosità a solido sistema di pagamento digitale.

Analizzando la tabella 7.2 possiamo osservare le differenze strutturali tra le tre rappresentazioni della blockchain. L’Address-transaction graph, rappresentazione dove ciascun indirizzo coinvolto in una transazione è modellato come nodo **Address** e le transazioni sono modellate come nodi **Transaction**, arriva a totalizzare 539 milioni di nodi e 1.3 miliardi di archi. Il Payment graph, dove ciascun output di una transazione è invece modellato come nodo **TXO**, registra un numero di nodi e archi ancora maggiore, rispettivamente 668 milioni e 2.1 miliardi. Questo aumento è influenzato dal fatto che mentre nel caso dell’Address-transaction graph il numero di archi è dato dalla somma del numero di input (relazioni **INPUT** tra nodi **Address** e **Transaction**) e output (relazioni **OUTPUT** tra nodi **Transaction** e **Address**) di una transazione, nel payment graph il numero di archi è dato dal prodotto tra il numero di input e output, proprio perchè non è presente un nodo **Transaction** che funge da intermediario tra le entrate e le uscite di una transazione. Infine, l’Address graph con valori aggregati, sebbene presenti un numero di nodi inferiore rispetto alle altre due rappresentazioni (solamente 293 milioni), registra un numero di archi simile all’address-transaction graph con 1.5 miliardi di archi. Tutte e tre le rappresentazioni evidenziano la complessità della rete Bitcoin e la sfida di gestire in maniera efficiente questa vasta quantità di “Big Data” in un database a grafo.

La tabella 7.3 mette in luce come l’aumento del volume di dati considerati della blockchain Bitcoin rifletta un aumento delle dimensioni del database per tutte e tre le rappresentazioni. Risulta interessante notare come l’Address-transaction graph e il Payment graph abbiano dimensioni del database simili (130 GB e 131 GB rispettivamente) nonostante il numero di nodi e archi sia significativamente maggiore nel payment graph. L’Address graph con valori aggregati, anche se presenta un numero di nodi inferiore, registra una dimensione del database maggiore rispetto alle altre due rappresentazioni (184 GB) a causa della presenza di numerose proprietà aggregate associate agli archi **TRANSFERS_TO** (7 in tutto) che aumentano il volume di dati da memorizzare per ciascun arco.

Infine, la Tabella 7.4 evidenzia le differenze nei tempi di importazione dei chunk per le tre rappresentazioni. Tutte mostrano un incremento del tempo di importazione all’aumentare dei chunk, coerentemente con l’aumento del volume di dati da elaborare. Tuttavia, è rilevante notare come il Payment graph e l’Address graph con valori aggregati richiedano, in alcuni casi, tempi più lunghi rispetto all’Address-transaction graph, suggerendo una maggiore complessità nel trattamento dei dati. È interessante osservare come l’ultimo chunk, nono-

7.1. EVOLUZIONE DELLA DIMENSIONE DEL DATABASE E DEL TEMPO DI IMPORTAZIONE

stante il trend generale di crescita, mostri un tempo di importazione inferiore rispetto al penultimo, ad esempio per l'Address graph con poco più di 1 ora e 10 minuti necessaria per il suo completamento rispetto alle oltre 2 ore e 20 minuti richieste dal penultimo. Questa riduzione è spiegata dal fatto che il numero di transazioni è diminuito rispetto al penultimo chunk da 48.7 milioni a 22.7 milioni (a causa della conclusione del periodo considerato dal dataset, troncando l'ultimo chunk a due mesi anziché sei), come riportato in tabella 7.1, e quindi il carico di lavoro è minore.

7.1 Evoluzione della dimensione del database e del tempo di importazione

7.1.1 Address-transaction graph

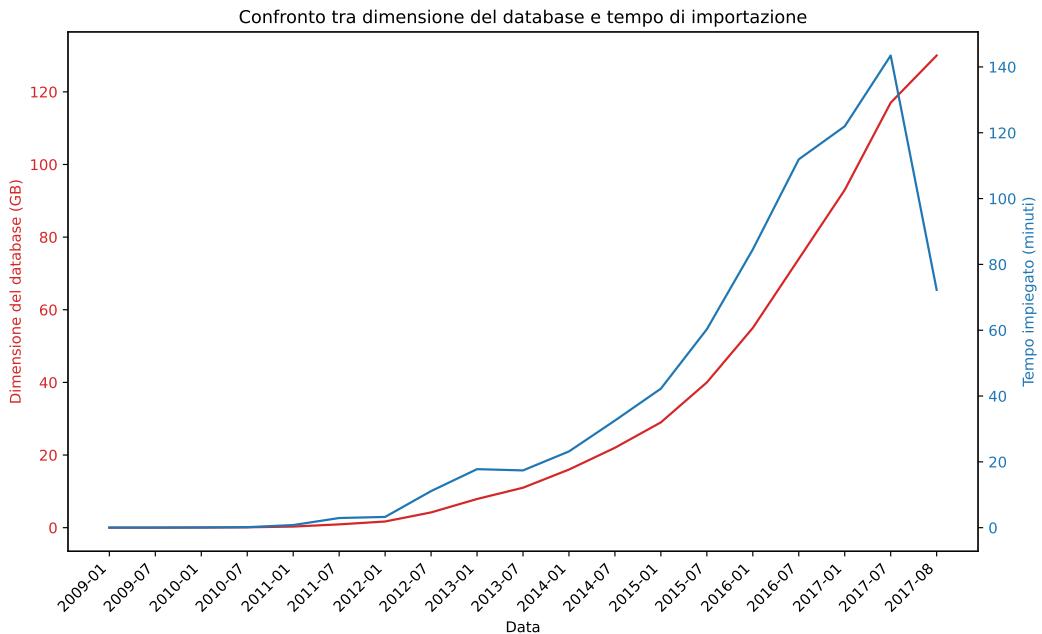


Figura 7.1: Evoluzione della dimensione del database e del tempo di importazione per l'Address-transaction graph.

In figura 7.1 è mostrato un grafico che mette a confronto la dimensione del database e il tempo di importazione all'aumentare del numero di chunk importati nel caso dell'Address-transaction graph. La tendenza generale mostra che, man mano che si procede con l'importazione di chunk successivi corrispondenti a intervalli di tempo più recenti, sia il tempo di importazione

che la dimensione del database aumentano. Questo fenomeno riflette l'intensificarsi dell'attività sulla rete Bitcoin, come confermato dal crescente numero di transazioni riportato in tabella 7.1. Questo aumento di attività è indice del passaggio di Bitcoin da semplice interesse di nicchia a sistema di pagamento digitale globale, con un numero sempre maggiore di partecipanti e transazioni.

Analizzando i dati della tabella 7.1 si può notare come il numero di transazioni per ciascun chunk cresce in maniera repentina nel tempo, il che suggerisce che più utenti stiano facendo uso della rete e/o che gli utenti esistenti stiano eseguendo transazioni più frequentemente. I primi 7 chunk sono stati importati molto velocemente (ognuno in meno di 4 minuti), riflettendo un numero relativamente basso di transazioni e un limitato numero di nodi e archi da gestire. Tuttavia, col progredire dei chunk, è possibile notare un aumento del carico di lavoro per il database: dall'ottavo chunk che richiede più di 11 minuti per poi arrivare al penultimo chunk che ha richiesto oltre 2 ore e 20 minuti per l'importazione, un tempo significativamente maggiore a causa dell'aumento del volume dei dati relativi alle transazioni che devono essere letti e processati.

Il processo di importazione è piuttosto intensivo: per ogni transazione presente nei file CSV viene creato un nodo **Transaction** con 5 proprietà e degli archi **INPUT** e **OUTPUT**, con 3 e 2 proprietà rispettivamente, per connettere i nodi **Address** ai nodi **Transaction** a seconda del ruolo dell'indirizzo nell'input e/o nell'output della transazione. Nonostante l'efficienza potenziale dell'operatore **MERGE** di Neo4j, che verifica l'esistenza di un nodo sulla base di una proprietà e lo crea solo nel caso in cui non esista già, il costante aumento del numero di transazioni nel tempo porta a un maggior numero di ricerche e potenziali creazioni di nuovi nodi **Address**. Va ricordato, in riferimento al codice in sezione 5.1 utilizzato per effettuare l'importazione, che i nodi **Address** vengono creati durante il processo di elaborazione di una transazione solo nel caso in cui non siano stati già coinvolti in transazioni precedenti, richiedendo però del tempo aggiuntivo per controllare l'esistenza dell'indirizzo nel database. Inoltre, ogni nuovo nodo **Address** comporta l'indicizzazione della sua proprietà **addressId** per ottimizzare le future query di analisi, un'operazione che richiede tempo e spazio aggiuntivo.

7.1. EVOLUZIONE DELLA DIMENSIONE DEL DATABASE E DEL TEMPO DI IMPORTAZIONE

7.1.2 Payment graph

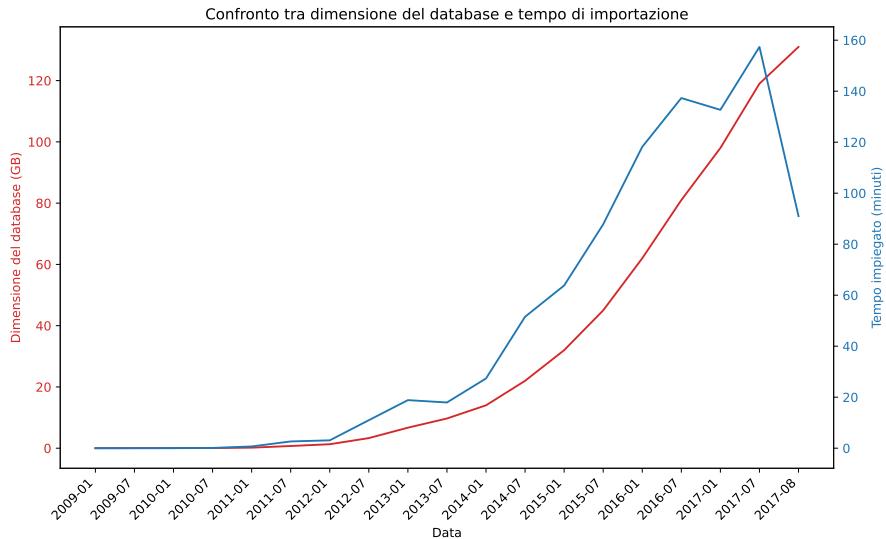


Figura 7.2: Evoluzione della dimensione del database e del tempo di importazione per il Payment graph.

Il grafico in figura 7.2 mostra l’evoluzione della dimensione del database e del tempo di importazione al variare del numero di chunk importati nel caso del Payment graph. Anche in questo caso si osserva un aumento della dimensione del database e del tempo di importazione all’aumentare del numero di chunk importati. Questo trend di crescita raggiunge il suo picco nel penultimo chunk, dove il tempo di importazione raggiunge le 2 ore e 30 minuti e la dimensione del database i 131 GB, risultati in linea con quelli ottenuti per l’Address-transaction graph (2 ore e 20 minuti e 130 GB rispettivamente).

Anche in questo caso il tempo di importazione è influenzato dalla quantità di dati da elaborare e dalla complessità delle operazioni richieste. Nel caso del Payment graph, la cui implementazione nel codice in sezione 5.2 evidenzia il carico di lavoro che deve essere sostenuto da Neo4j per ciascuna riga del dataset, è necessario creare un nodo TX0 per ciascun output di una transazione e poi andare a recuperare i nodi TX0 utilizzati come input, già esistenti, per creare una relazione CONTRIBUTES tra ciascun nodo TX0 di input e di output. La complessità di questa fase di creazione di nodi e archi aumenta ulteriormente se si considera che per velocizzare le operazioni di importazione è costantemente aggiornato un indice composto per la coppia di proprietà txId,position dei nodi TX0, questo al fine di garantire un accesso rapido ai nodi di output coinvolti

come input in una transazione ma aumentando l'onere computazionale e di memoria richiesto da Neo4j.

7.1.3 Address graph con valori aggregati

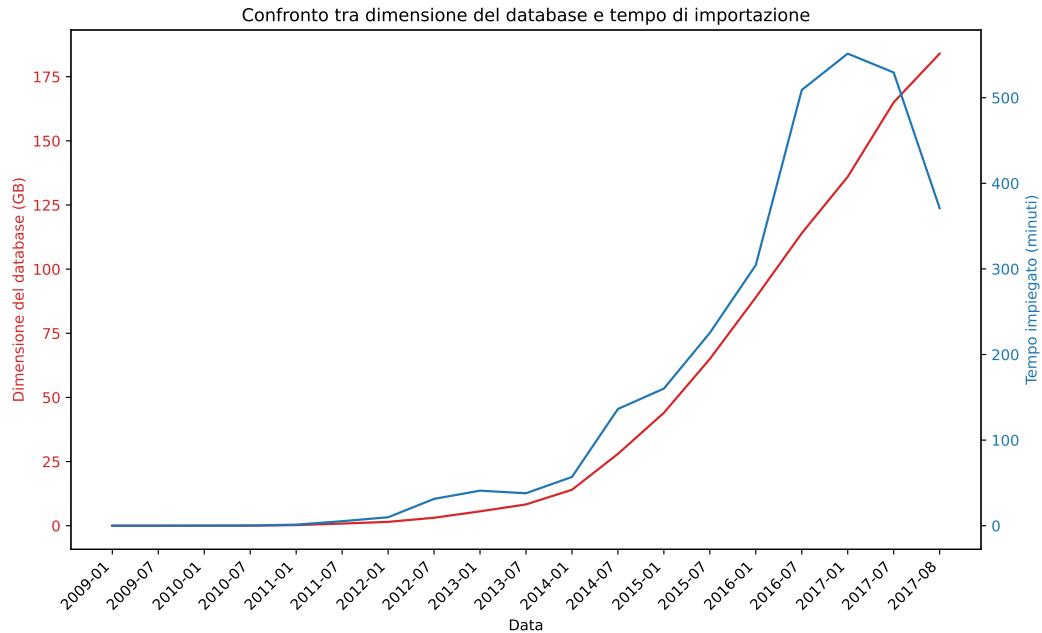


Figura 7.3: Evoluzione della dimensione del database e del tempo di importazione per l'Address graph con valori aggregati.

Come mostrato in figura 7.3, l'evoluzione della dimensione del database e del tempo di importazione per l'Address graph con valori aggregati segue una tendenza di crescita ma con alcune differenze rispetto alle due rappresentazioni precedenti riprese nei grafici di figura 7.1 e 7.2. La principale differenza che si ha in questo caso è data dalle operazioni complesse richieste dall'aggregazione di valori, come il calcolo della proporzione di valore che ogni input contribuisce ad ogni output e la gestione degli archi TEMP_TRANSFERS e TRANSFERS_TO. Il calcolo delle proporzioni di valore e l'aggiornamento o la creazione degli archi TRANSFERS_TO con le loro relative proprietà incrementali e, infine, la rimozione degli archi TEMP_TRANSFERS richiedono più elaborazione e quindi più tempo per l'importazione.

Le operazioni che vengono eseguite per ogni transazione durante l'importazione, descritte dettagliatamente nel codice in sezione 5.3, includono:

7.1. EVOLUZIONE DELLA DIMENSIONE DEL DATABASE E DEL TEMPO DI IMPORTAZIONE

- il calcolo della somma totale degli importi degli input per ogni transazione;
- la creazione degli archi TEMP_TRANSFERS tra ogni coppia di indirizzo mittente e destinatario per determinare la proporzione di valore trasferito da ogni input ad ogni output, corrispondente alla proprietà **amount** dell'arco;
- l'aggiornamento o la creazione degli archi TRANSFERS_T0, che aggregano le somme degli importi e registrano gli identificativi della prima e ultima transazione e blocco in cui è avvenuto un trasferimento per ogni coppia di nodi che hanno scambiato valore. Questo comporta la verifica della presenza di archi TRANSFERS_T0 esistenti e l'aggiornamento delle loro proprietà oppure la creazione di nuovi archi con queste proprietà se non sono già presenti;
- la rimozione degli archi TEMP_TRANSFERS dopo l'aggiornamento degli archi TRANSFERS_T0.

7.2 Evoluzione del numero di nodi e archi

7.2.1 Address-transaction graph

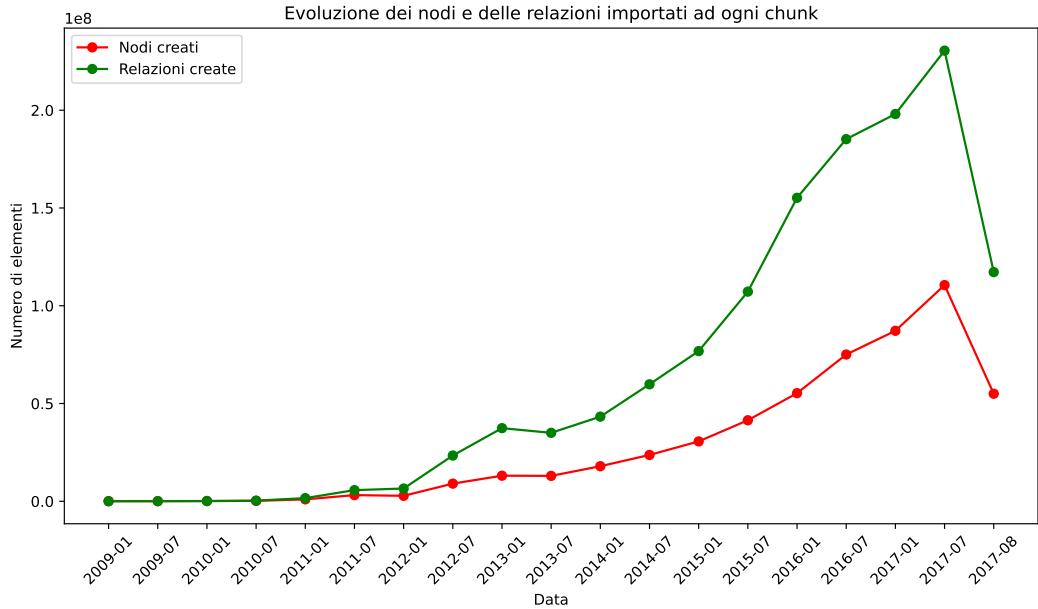


Figura 7.4: Evoluzione del numero di nodi e archi importati per l’Address-transaction graph al variare del numero di chunk.

Il grafico in figura 7.4 illustra l’evoluzione del numero di nodi e archi creati nel tempo, considerando chunk corrispondenti a intervalli di sei mesi, nella blockchain di Bitcoin rappresentata come Address-transaction graph. Nelle fasi iniziali, corrispondenti ai primi chunk, il numero di nodi e archi è piuttosto contenuto e cresce lentamente. In Neo4j, la creazione di nodi e archi in questa fase avviene in modo efficiente, grazie al minor volume di dati e alla semplicità della rete.

Progressivamente, si osserva un’accelerazione significativa nella crescita dei nodi e archi. Per esempio, analizzando l’intervallo di tempo tra il 2012-01 e il 2013-01 e similmente tra il 2014-01 e il 2016-01, il numero di nodi e archi aumenta notevolmente. L’aumento del numero di nodi può corrispondere alla generazione di nuovi indirizzi Bitcoin mentre l’aumento degli archi è diretto indicatore della maggiore frequenza di transazioni e del coinvolgimento di più indirizzi in ciascuna transazione. Questa crescita anomala potrebbe essere dovuta all’introduzione di nuovi servizi o piattaforme che hanno portato a un

aumento dell'attività sulla rete Bitcoin come ad esempio satoshiDice [57], un popolare sito di gioco d'azzardo online che ha iniziato ad accettare Bitcoin come metodo di pagamento nel 2012, o Silk Road [58], un noto mercato nero lanciato nel 2011 sul dark web che accettava Bitcoin come unica forma di pagamento per beni e servizi illegali e che ha consentito vendite stimate per 9 519 664 Bitcoin. Il numero totale di nodi totali che si ha per rappresentare la blockchain di Bitcoin come un Address-transaction graph è di 539 208 251 (quasi 540 milioni) e il numero totale di archi è 1 283 445 382 (più di 1 miliardo). Questi numeri riflettono la complessità della rete Bitcoin e la sfida di gestire in maniera efficiente questa vasta quantità di dati in un database a grafo.

Verso la fine del periodo considerato, in particolare nel terzultimo e penultimo chunk, c'è una notevole espansione del numero di archi rispetto ai nodi. Questo potrebbe essere dovuto a un aumento delle transazioni che coinvolgono indirizzi esistenti, piuttosto che alla creazione di nuovi indirizzi. Il fatto che il numero di archi cresca più rapidamente dei nodi potrebbe indicare anche una maggiore interconnettività tra gli indirizzi e la comparsa di transazioni che coinvolgono più indirizzi contemporaneamente. Questo aspetto può rappresentare una sfida per Neo4j poiché gestire un numero crescente di relazioni può aumentare la complessità delle query e l'overhead della gestione del database.

Anche in questo caso la fluttuazione del numero di nodi e archi che si ha nell'ultimo chunk è spiegata da una diminuzione del numero di transazioni importate, come riportato in tabella 7.1.

7.2.2 Payment graph

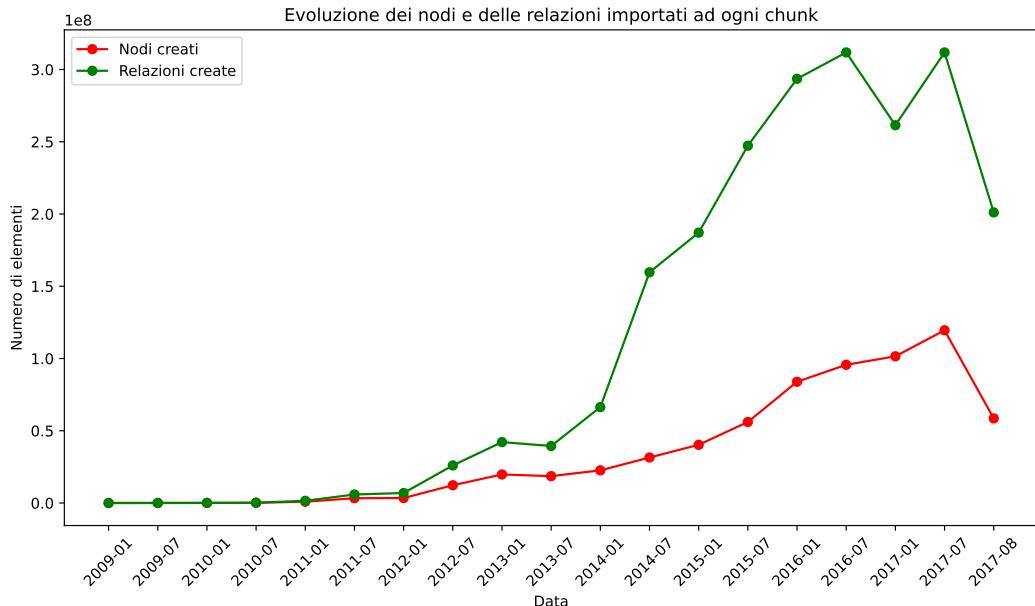


Figura 7.5: Evoluzione del numero di nodi e archi importati per il Payment graph al variare del numero di chunk.

Nella figura 7.5 relativa al Payment graph si nota che il numero di relazioni cresce a un ritmo molto più accelerato rispetto al numero di nodi. Questo è conforme alla natura del Payment graph dove per ogni transazione un numero di archi maggiore è creato poiché ogni input contribuisce alla creazione di ogni output. La curva delle relazioni mostra una pendenza molto più ripida a partire da luglio 2013, periodo in cui l'exchanger Mt. Gox [59] ha gestito oltre il 70% di tutte le transazioni Bitcoin. Si può inoltre notare una diminuzione del numero di archi creati nel terzultimo chunk - relativo al periodo da 2016-07 a 2017-01 - rispetto al chunk precedente e al successivo. Questo fenomeno potrebbe essere dovuto a un aumento nelle transazioni che coinvolgono un numero minore di input, che si traduce nel Payment graph in un minor numero di archi creati dato che il numero di archi per transazione è dato dal prodotto tra il numero di input e output.

7.2.3 Address graph con valori aggregati

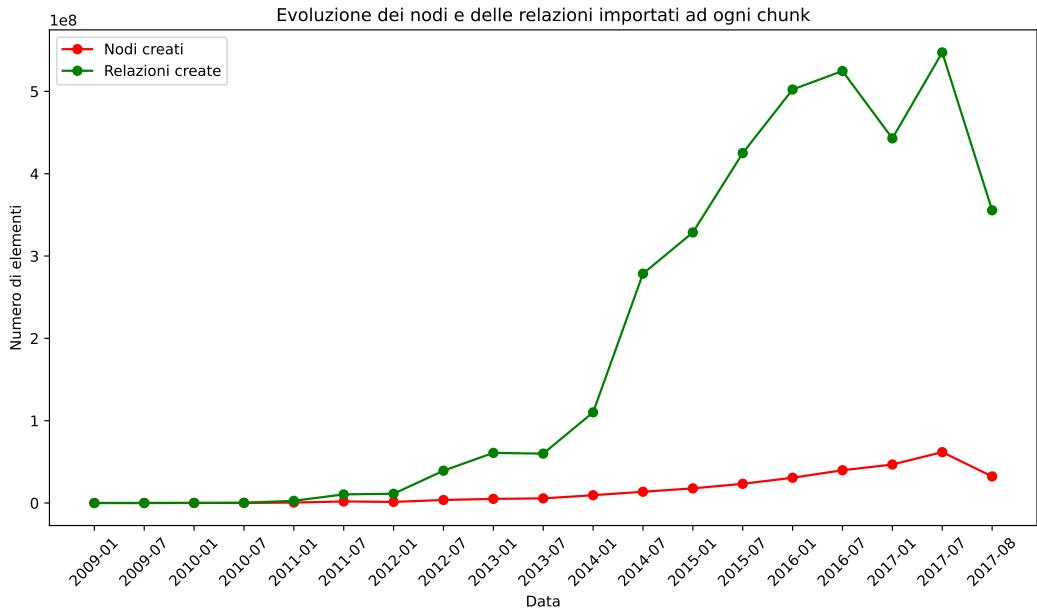


Figura 7.6: Evoluzione del numero di nodi e archi importati per l’Address graph con valori aggregati al variare del numero di chunk.

L’analisi dell’evoluzione dei nodi e delle relazioni che sono importati ad ogni chunk per l’Address graph con valori aggregati (figura 7.6) rispetto all’Address-transaction graph e al Payment graph (figura 7.4 e 7.5) rivela le differenze nella modellazione delle transazioni e l’impatto di tali modellazioni sulla crescita del database.

Nell’Address graph con valori aggregati ogni transazione tra due indirizzi richiede la creazione di archi temporanei (`TEMP_TRANSFERS`) per calcolare le proporzioni di valore trasferito. Questi archi temporanei, creati per ogni coppia di indirizzi mittente e destinatario di una transazione e eliminati dopo l’aggiornamento degli archi `TRANSFERS_TO`, sono utilizzati per aggiornare o creare nuovi archi `TRANSFERS_TO` che mantengono le proprietà aggregate delle transazioni avvenute tra gli indirizzi nel tempo. Questo processo, nonostante sia necessario per fornire una rappresentazione accurata e dettagliata della storia delle transazioni, aggiunge una complessità significativa. Ogni chunk di dati importati richiede la creazione e poi l’eliminazione di una grande quantità di archi temporanei, un’operazione che non solo aumenta il tempo di elaborazione ma anche la dimensione complessiva del database per gestire

questi elementi temporanei.

7.3 Valutazione dei tempi di esecuzione delle query

7.3.1 Address-transaction graph

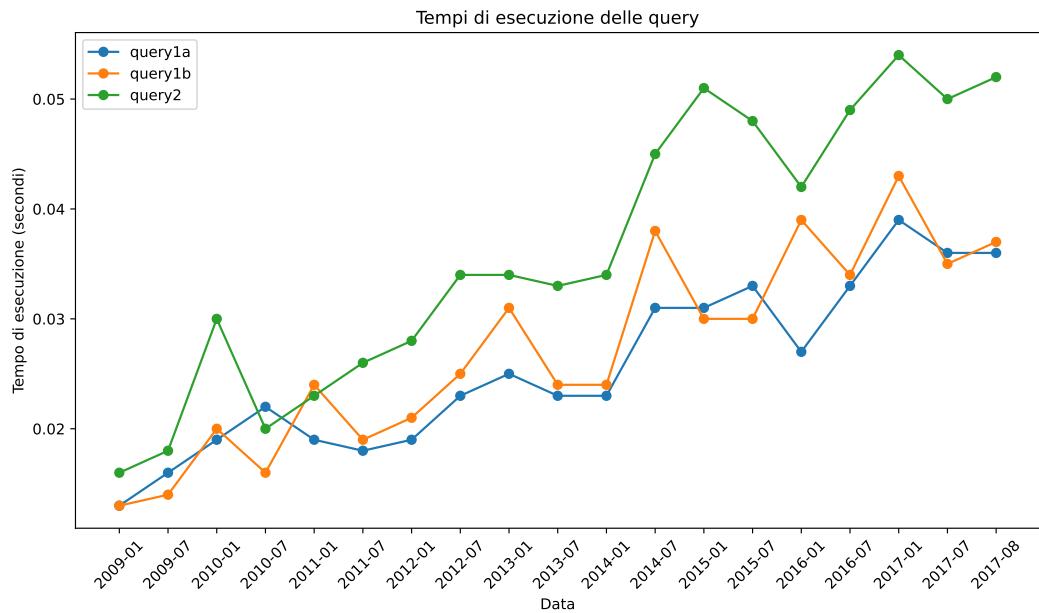


Figura 7.7: Evoluzione dei tempi di esecuzione delle query per l’Address-transaction graph usando $n = 1000$ indirizzi scelti casualmente nell’intero graph database ad ogni chunk importato.

Analizzando i tempi di esecuzione delle tre diverse query effettuate sull’Address-transaction graph si può osservare che i tempi di esecuzione delle query rimangono bassi e pressoché istantanei per tutto il periodo considerato, nonostante l’incremento significativo delle dimensioni del grafo riportate in figura 7.1 e 7.4. Questo sottolinea l’efficienza degli indici di Neo4j, i quali permettono di mantenere tempi di risposta rapidi anche quando si lavora con grandi quantità di dati. Le query specifiche, che riguardano il timestamp della prima transazione di un indirizzo come mittente (query1a) e come destinatario (query1b), e la dimensione dell’ego-network di un indirizzo (query2), beneficiano particolarmente dall’indicizzazione delle proprietà

`addressId` per poter individuare rapidamente nodi specifici all'interno del grafo.

La scelta di pre-selezionare gli n indirizzi casuali prima dell'esecuzione delle query, salvarli su un file e poi eseguire le query utilizzando il file creato precedentemente come input consente di eliminare la variabilità dei tempi di esecuzione legata alla selezione casuale degli indirizzi. Questo approccio si rivela molto efficace, come evidenziato dal grafico in figura 7.7, poiché isola il tempo di esecuzione della query dalle operazioni preliminari di selezione degli indirizzi, garantendo che i tempi riflettano unicamente la performance della query di analisi stessa su Neo4j. Questi benchmark confermano l'importanza di strategie di ottimizzazione come l'uso di indici per poter accedere direttamente ai nodi di interesse e ridurre i tempi di esecuzione delle query in grafi di grandi dimensioni.

7.3.2 Payment graph

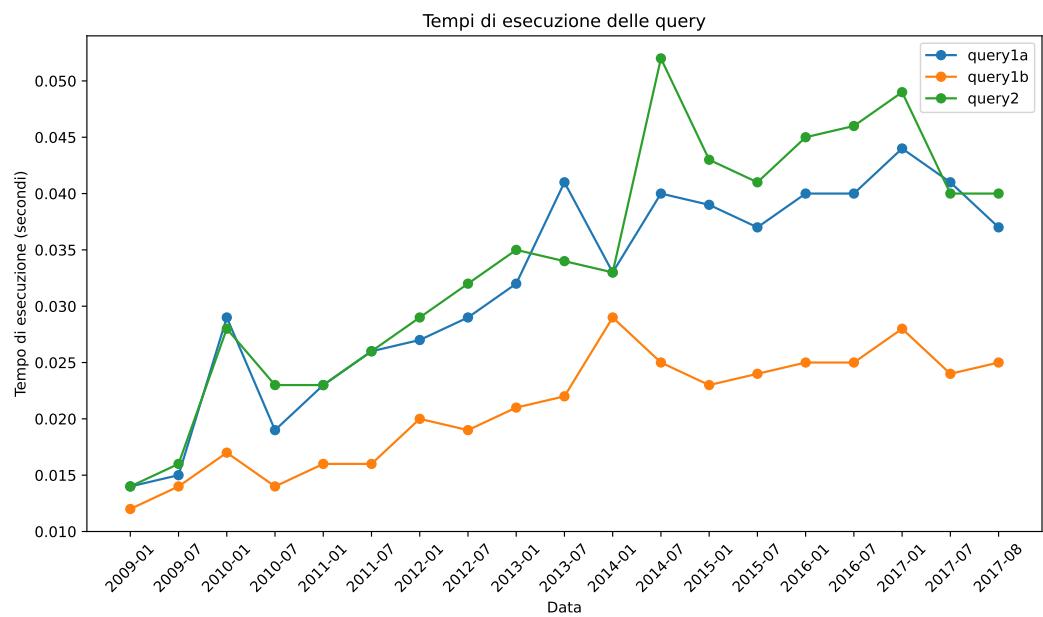


Figura 7.8: Evoluzione dei tempi di esecuzione delle query per il Payment graph usando $n = 1000$ indirizzi scelti casualmente nell'intero graph database ad ogni chunk importato.

In figura 7.8 si osserva che i tempi di esecuzione delle query variano nel tempo, ma generalmente mantengono una tendenza crescente con alcuni picchi. Per

esempio si può osservare che le query 1a e 1b, che si occupano di calcolare per un insieme di indirizzi il timestamp della prima transazione che coinvolge un dato indirizzo come mittente e come destinatario, mostrano un andamento crescente nel tempo. Questo potrebbe essere dovuto all'aumento del numero di output che devono essere analizzati per ciascun indirizzo, in quanto ogni output è rappresentato da un nodo TXO nel Payment graph. La query 2 invece, che misura la dimensione dell'ego-network di un indirizzo, mostra variazioni più marcate con un picco evidente verso luglio 2014, suggerendo che in quel periodo la rete Bitcoin ha subito un cambiamento sufficientemente significativo da influenzare la struttura del grafo e quindi, seppur leggermente, i tempi di esecuzione della query.

7.3.3 Address graph con valori aggregati

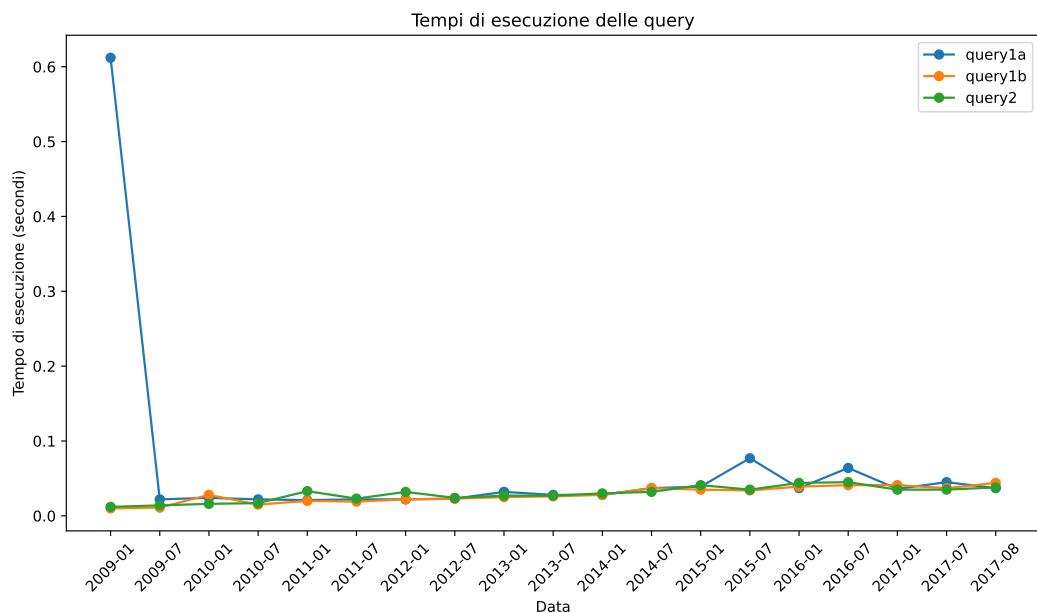


Figura 7.9: Evoluzione dei tempi di esecuzione delle query per l'Address graph con valori aggregati usando $n = 1000$ indirizzi scelti casualmente nell'intero graph database ad ogni chunk importato.

I tempi di esecuzione delle query per l'Address graph con valori aggregati, come mostrato in figura 7.9, sono caratterizzati da un picco iniziale insolito che potrebbe essere attribuito a una variazione casuale dato che il numero di nodi e archi è ancora relativamente basso nel primo chunk. Questo picco non sembra influire sulla tendenza complessiva che mostra tempi di esecuzione

delle query generalmente bassi e costanti nel tempo, suggerendo che nonostante l’elaborazione aggiuntiva richiesta per calcolare e mantenere gli archi con valori aggregati, gli indici definiti sulla proprietà `addressId` per i nodi `Address` sono sufficienti per garantire una rapida esecuzione delle query di analisi.

7.4 Confronti complessivi

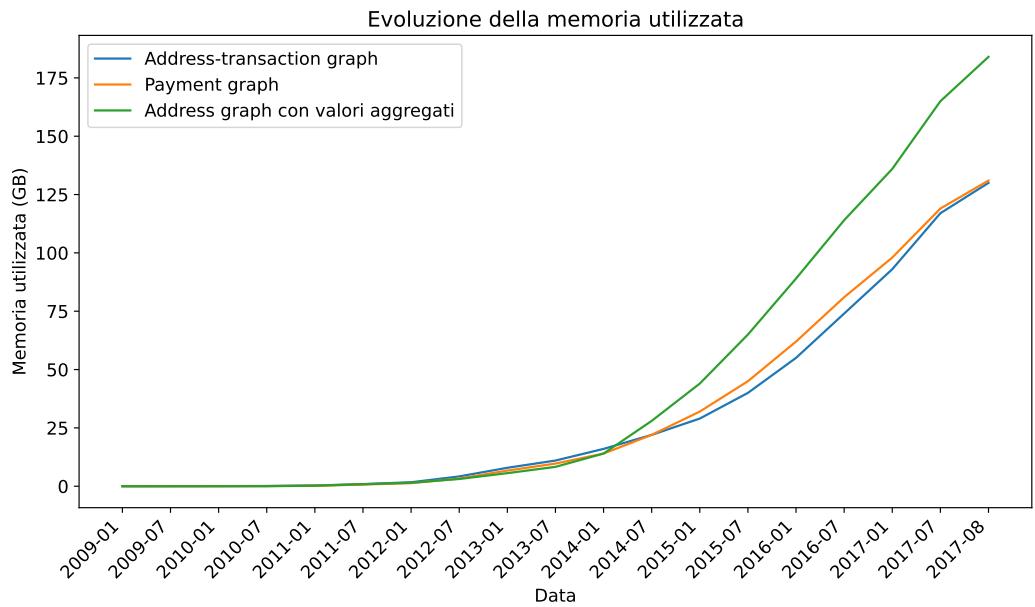
In questa sezione l’obiettivo si sposta sull’analisi comparativa dei modelli considerati singolarmente nelle sezioni 7.1, 7.2 e 7.3 precedenti, con l’intento di mettere in luce i relativi vantaggi e svantaggi derivanti dall’applicazione di ciascuna rappresentazione in contesti d’uso specifici, come ad esempio applicazioni real-time o strumenti che richiedono analisi dettagliate dei flussi di valore (magari per scopi investigativi o forensi) della rete Bitcoin. In particolare, sarà sottolineato come le diverse scelte di modellazione si traducano in un impatto concreto sulle risorse computazionali richieste (tempo e spazio di memoria) e sul livello di dettaglio delle informazioni estratte dalla blockchain.

Durante il confronto emergeranno le sfide e le opportunità associate all’uso di un modello rispetto agli altri, rivelando come non esista una soluzione universalmente ottimale ma, piuttosto, compromessi tra il grado di dettaglio desiderato nella descrizione degli scambi all’interno della rete Bitcoin e l’efficienza in termini di risorse necessarie per mantenere e interrogare il database a grafo.

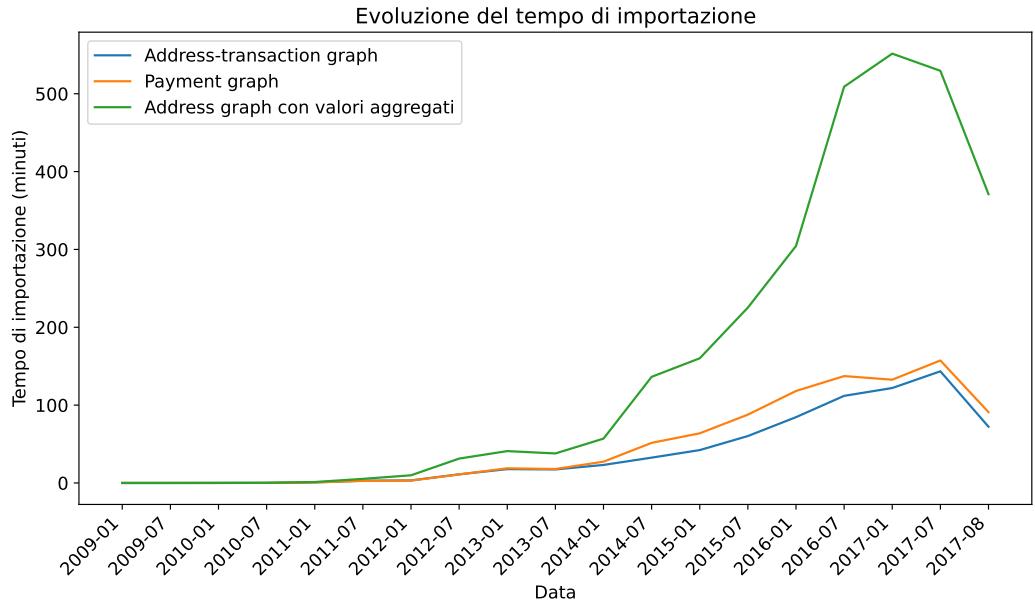
Attraverso un’analisi dettagliata e comparativa si cercherà di delineare scenari applicativi ideali dei modelli evidenziando come la scelta del modello più appropriato possa variare a seconda delle esigenze specifiche di analisi, del contesto d’uso e delle priorità in termini di prestazioni e dettaglio delle informazioni.

7.4.1 Dimensione del database e tempo di importazione

Confrontando mediante i grafici 7.10a e 7.10b l’Address-transaction graph e il Payment graph si può osservare che le due rappresentazioni condividono una tendenza simile nell’aumento della dimensione del database e del tempo di importazione, ma con delle lievi differenze. Sebbene le due rappresentazioni, basate sugli stessi dati, finiscano per convergere verso dimensioni del database simili, il Payment graph tende ad avere dei tempi di importazione leggermente più lunghi rispetto all’Address-transaction graph soprattutto nei chunk più recenti. Un’analisi più approfondita del codice in sezione 5.2 rivela che nel Payment graph la rappresentazione di ciascun output



(a) Evoluzione della dimensione di ciascun database al variare del numero di chunk importati.



(b) Evoluzione del tempo di importazione di ciascun database al variare del numero di chunk importati.

Figura 7.10

di una transazione come un nuovo nodo TXO comporta inevitabilmente la creazione di un numero maggiore di nodi rispetto all'Address-transaction graph, rappresentazione dove si può avere un risparmio nella creazione di nodi **Address** grazie all'operatore **MERGE** di Cypher, operatore utilizzato in questo caso per recuperare indirizzi già esistenti qualora essi siano stati coinvolti in transazioni precedenti. Questa scelta implementativa si riflette direttamente nel tempo di importazione più elevato per il Payment graph. Inoltre, la relazione **CONTRIBUTES** nel Payment graph crea una rete di connessioni più complessa rispetto all'Address-transaction poiché ogni input (che è a sua volta un transaction output, cioè un nodo TXO) è collegato ad ogni output della transazione corrente, con un numero totale di relazioni **CONTRIBUTES** per transazione che è dato dal prodotto tra il numero di input e il numero di output, anziché la somma del numero di input e output come nel caso dell'Address-transaction graph. Questo comporta un aumento del numero di archi da gestire e quindi un aumento del tempo necessario per l'importazione.

In termini di indici, sia Address-transaction graph che il Payment graph beneficiano dell'indicizzazione sulla proprietà **addressId** per velocizzare le query di importazione e di analisi. Tuttavia, l'impatto di questa scelta può variare a seconda della struttura del grafo e della natura delle relazioni. Nel Payment graph la presenza di più nodi TXO associati allo stesso **addressId** ha un chiaro impatto sulla creazione e gestione degli indici: mentre nel caso dell'Address-transaction graph un dato **addressId** è associato a un solo nodo **Address**, nel Payment graph più transaction output (ognuno identificato da un nodo TXO) possono essere associati allo stesso **addressId**, proprio perché nel tempo un indirizzo può ricevere valore da più transazioni. Questo comporta un aumento del numero di nodi da indicizzare e quindi un aumento, seppur lieve, del tempo necessario per l'importazione.

Nell'Address-transaction graph e nel Payment graph le operazioni di importazione risultano essere meno onerose in termini di calcolo rispetto al calcolo delle proporzioni di valore e alla gestione degli archi **TRANSFERS_TO** nell'Address graph con valori aggregati, il che si riflette in tempi di importazione più brevi e dimensioni del database più contenute. I tempi di importazione per l'Address graph con valori aggregati sono molto più lunghi, raggiungendo 500 minuti (circa 8 ore e 20 minuti) nell'ultimo chunk importato rispetto alle circa 2 ore e 30 minuti per il Payment graph e l'Address-transaction graph. Questo risultato è coerente con l'intensa attività di calcolo richiesta per ogni transazione processata. Anche nel caso dell'Address graph con valori aggregati è stata effettuata, come nel caso delle altre due rappresentazioni, un'indicizzazione

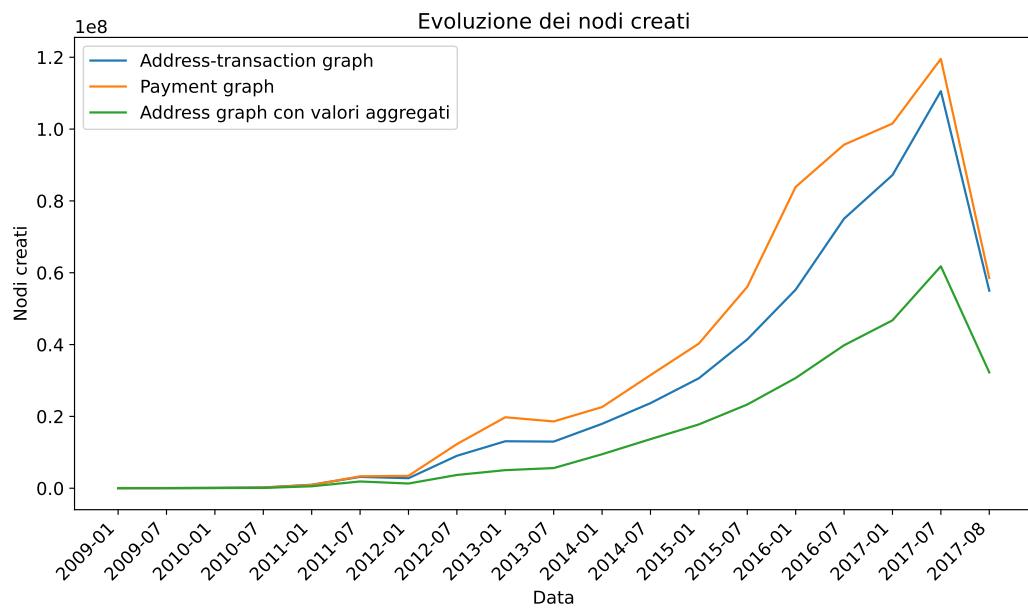
della proprietà `addressId` dei nodi `Address` al fine di ottimizzare le query di analisi e importazione, comportando però un ulteriore costo computazionale e di memorizzazione durante l'importazione. Questo si riflette in un database più grande, come indicato dalla dimensione massima di 184 GB per l'Address graph con valori aggregati, confrontata con i 130 GB e 131 GB rispettivamente per l'Address-transaction graph e il Payment graph.

7.4.2 Numero di nodi e archi importati

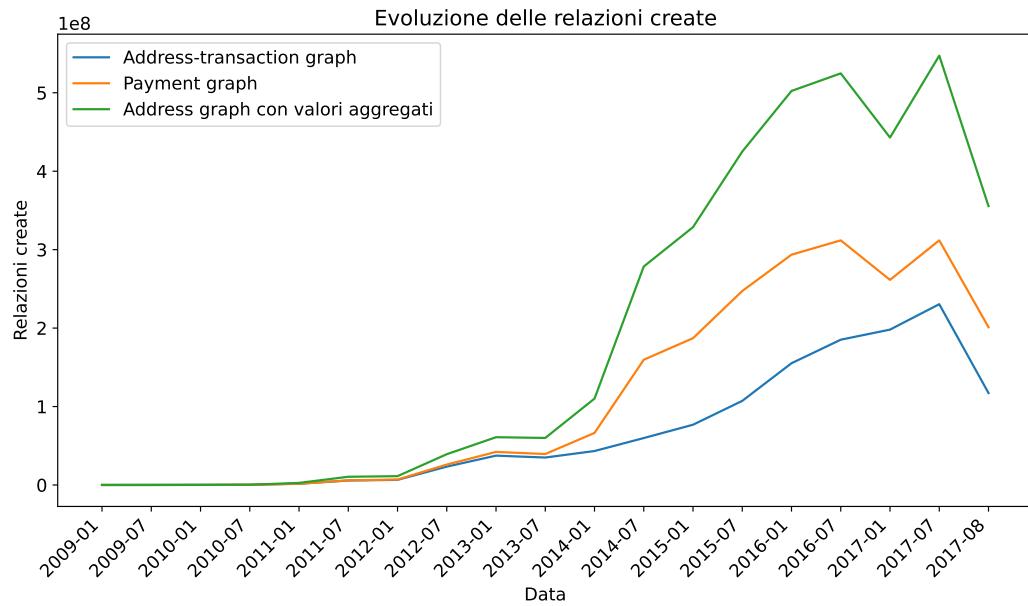
Nella figura 7.11b si osserva come la crescita del numero di relazioni nell'Address-transaction graph non sia altrettanto marcata come nel Payment graph. Questo è giustificato dal fatto che nell'Address-transaction si ha un numero di archi creati per transazione che è dato dalla somma del numero di input e output, piuttosto che il prodotto tra i due come nel Payment. L'andamento del numero di nodi creati in ogni chunk, presente in figura 7.11a, mostra invece che l'aumento dell'attività della rete Bitcoin influisce in modo simile su entrambe le rappresentazioni.

È evidente che l'intensificarsi delle transazioni si traduce in un aumento del numero di nodi e archi per entrambi i grafici. Questa correlazione è particolarmente notevole nei periodi più recenti, dove l'uso e la popolarità di Bitcoin hanno raggiunto il loro apice. L'aumento del numero di transazioni si traduce in un maggiore numero di nodi `Transaction` e `Address` nel caso dell'Address-transaction graph e nodi `TXO` nel Payment graph, nonché un incremento nelle relazioni create in entrambi i casi. Mentre il Payment graph fornisce una rappresentazione più granulare e dettagliata del valore generato dalle transazioni andando a considerare ogni output come un nodo separato, l'Address-transaction graph offre una visione più ad alto livello della rete Bitcoin, concentrandosi sulle transazioni e sugli indirizzi in esse coinvolti, e gestibile soprattutto per quanto riguarda il numero di relazioni create. Il numero totale di nodi e archi per il Payment graph è rispettivamente di 668 261 953 (circa 668 milioni) e 2 162 523 341 (più di 2 miliardi), confermando la maggiore complessità di questa rappresentazione per quanto riguarda la modellazione del flusso di valore delle transazioni rispetto all'Address-transaction graph.

Confrontando l'Address graph con valori aggregati con l'Address-transaction graph e il Payment graph, la crescita del numero di nodi risulta diretta e meno onerosa in termini di operazioni di database, cosa non altrettanto vera per gli archi. Nell'Address-transaction graph ogni transazione comporta la creazione di un nodo `Transaction` e la creazione di relazioni `INPUT` e `OUTPUT` tra i nodi `Transaction` e `Address`, senza la necessità di archi temporanei o il calcolo di



(a) Evoluzione del numero di nodi creati in ciascun database al variare del numero di chunk importati.



(b) Evoluzione del numero di relazioni create in ciascun database al variare del numero di chunk importati.

Figura 7.11

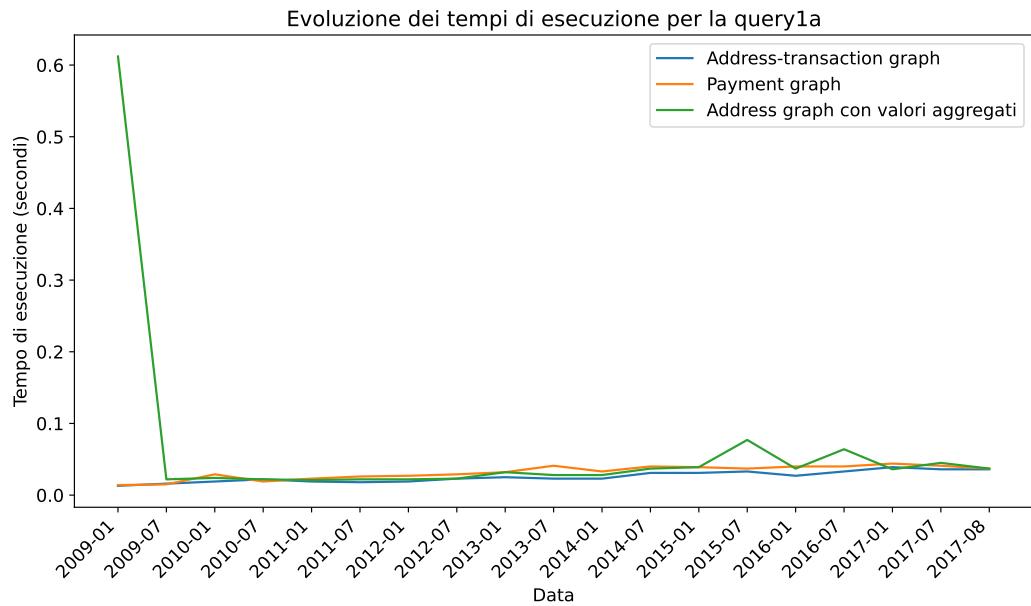
valori aggregati. Nel Payment graph invece ogni output di una transazione è rappresentato come un nodo TXO e ogni input come un contributo a quegli output attraverso relazioni **CONTRIBUTES**, approccio che aumenta il numero di relazioni ma mantiene un modello più diretto senza necessità di passaggi aggiuntivi di aggregazione.

Questo confronto tra i 3 database nelle figure 7.11a e 7.11b mostra come la rappresentazione scelta per la modellazione della blockchain influenzi non solo la struttura del database ma anche la sua gestibilità. L'Address graph con valori aggregati, nonostante offra una rappresentazione più ricca delle transazioni, richiede notevoli risorse computazionali e di storage per gestire la complessità aggiunta dagli archi temporanei e dalle proprietà aggregate, che si riflette nell'aumento esponenziale del numero relazioni nel tempo come si può vedere dai picchi presenti nel grafico intorno al periodo conclusivo considerato dal dataset. In contrasto, l'Address-transaction graph e il Payment graph, pur aumentando nel tempo, mostrano un tasso di crescita più gestibile, riflettendo una modellazione diretta e meno elaborata delle transazioni. Il numero totale di nodi e archi per l'Address graph con valori aggregati è rispettivamente di 293 725 439 (circa 294 milioni) e 1 537 624 607 (1 miliardo e mezzo), dove nel caso degli archi è stato conteggiato solo il numero di archi **TRANSFERS_TO** che si hanno alla fine del processo di importazione e non si è considerato gli archi **TEMP_TRANSFERS** creati e poi eliminati con lo scopo di calcolare le proprietà aggregate.

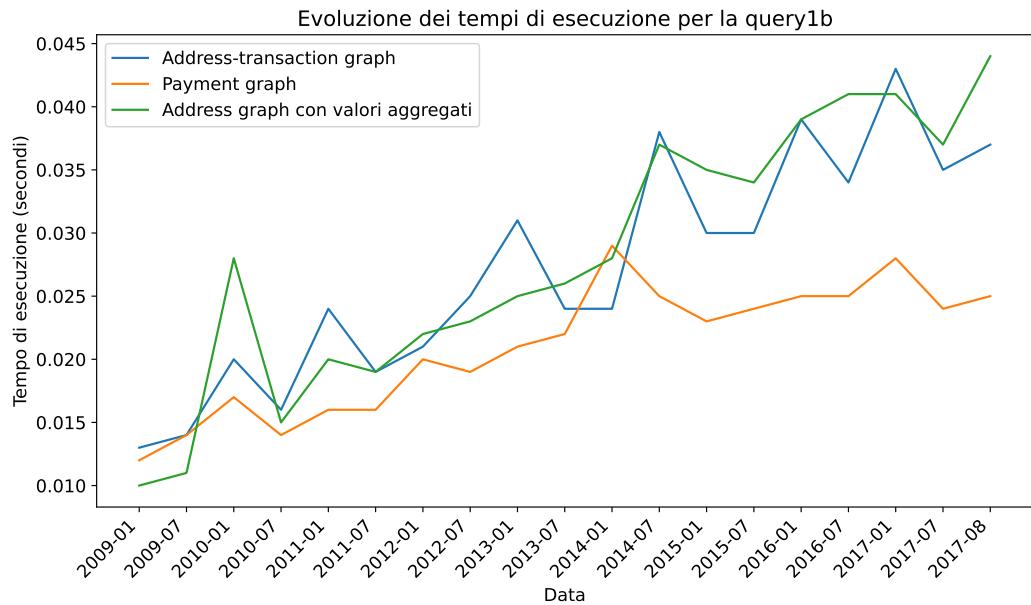
7.4.3 Tempo di esecuzione delle query

Confrontando i tempi di esecuzione delle query sul Payment graph e sull'Address-transaction graph, riportati nelle figure 7.12a, 7.12b e 7.12c, si può esservare una tendenza generale all'aumento col progredire dei chunk importati. È interessante notare che nonostante il Payment graph abbia un numero maggiore di nodi con la proprietà **addressId** indicizzata, i tempi di esecuzione delle query non sono significativamente più lunghi rispetto all'Address-transaction graph.

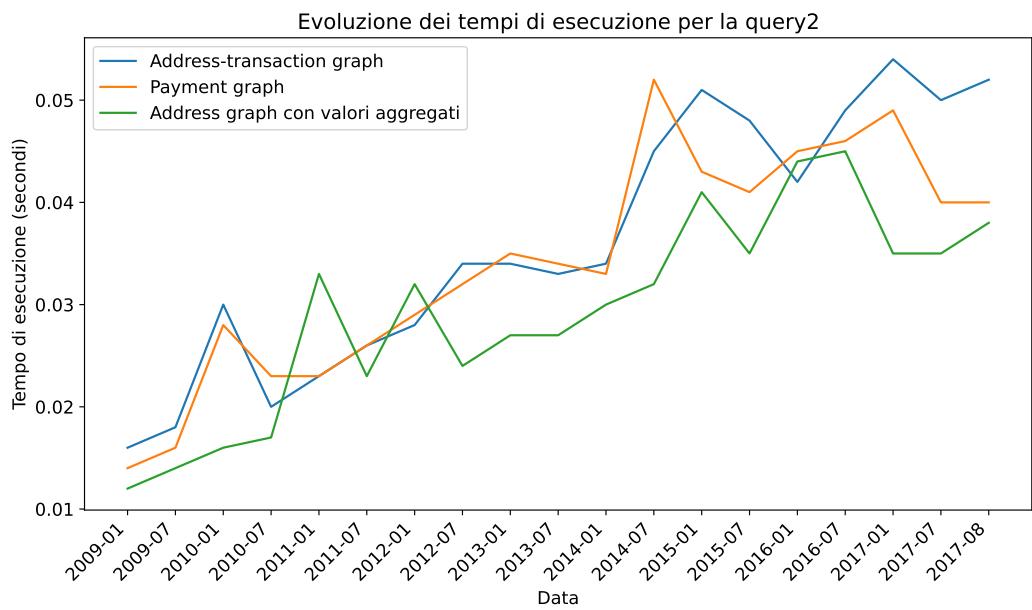
Considerando anche l'Address graph con valori aggregati in questi confronti sui tempi di esecuzione delle query possiamo concludere che quest'ultima rappresentazione, nonostante la sua complessità intrinseca e il maggiore one-re computazionale, mantiene prestazioni in linea con gli altri modelli grazie all'efficace strategia di indicizzazione. Questo potrebbe indicare che gli indici di Neo4j sono particolarmente efficienti e che il costo aggiuntivo dell'indicizzazione di più nodi è compensato dalla capacità di Neo4j di gestire in modo



(a) Evoluzione dei tempi di esecuzione della query 1a per ciascun database al variare del numero di chunk importati.



(b) Evoluzione dei tempi di esecuzione della query 1b per ciascun database al variare del numero di chunk importati.



(c) Evoluzione dei tempi di esecuzione della query 2 per ciascun database al variare del numero di chunk importati.

Figura 7.12

efficiente le query su grafi di grandi dimensioni, non influenzando in modo significativo i tempi di esecuzione. Tutti i grafici mostrano l'importanza di avere indici ben progettati, soprattutto quando le query richiedono l'accesso rapido a informazioni specifiche all'interno di un ampio insieme di dati.

7.5 Considerazioni

La sezione sperimentale della tesi ha fornito intuizioni fondamentali sulle prestazioni e le caratteristiche di tre diverse rappresentazioni della blockchain di Bitcoin come grafo: l'Address-transaction graph, il Payment graph e l'Address graph con valori aggregati. Gli esperimenti condotti hanno messo in luce come ciascun modello si adatti a specifici scenari d'uso, evidenziando pregi e limiti in termini di tempi di importazione dei dati, gestione della memoria e tempi di esecuzione delle query.

Dai risultati sperimentali emerge che l'Address-transaction graph offre una rappresentazione equilibrata tra dettaglio e performance, rendendolo particolarmente adatto per applicazioni che necessitano di un aggiornamento costante o quasi in tempo reale della blockchain. Ovviamente, tutti i modelli descritti devono importare i dati della blockchain inizialmente e quindi sostenere in maniera obbligata costi computazionali e di memoria significativi, ma una volta completata questa fase iniziale è possibile procedere con un caricamento incrementale dei dati che richiede meno risorse e che permette di mantenere aggiornato il database con le informazioni dei blocchi più recenti. Con tempi di importazione relativamente contenuti, evidenziati in figura 7.10b, e una struttura che riflette la natura bipartita della blockchain, questo modello facilita l'analisi delle relazioni tra indirizzi e transazioni, pur mantenendo un numero gestibile di nodi e relazioni.

Il Payment graph, nonostante presenti tempi di importazione leggermente superiori rispetto all'Address-transaction a causa della maggiore quantità di nodi TXO creati e proprietà da indicizzare, come illustrato nelle figure 7.10b e 7.11a, si distingue per la sua capacità di offrire una visione granulare dei flussi di valore all'interno della rete. Questo modello si rivela quindi prezioso per analisi dettagliate delle transazioni, pur richiedendo risorse computazionali leggermente superiori rispetto all'Address-transaction graph, aspetto interessante specialmente in contesti dove possono essere richiesti frequenti aggiornamenti dei dati.

D'altro canto, l'Address graph con valori aggregati, sebbene offra informazioni

sintetiche e di facile interpretazione a chi non è esperto di blockchain, presenta significativi tempi di importazione e un elevato utilizzo della memoria, come dimostrato nelle figure 7.10b e 7.10a. Queste caratteristiche ne limitano l'impiego in applicazioni che richiedono aggiornamenti in tempo reale o frequenti, rendendolo più idoneo per analisi retrospettive o per utenti che richiedono una visione aggregata e semplificata delle transazioni.

La scelta del modello da utilizzare dipende fortemente dalle specifiche esigenze applicative. Per applicazioni che richiedono un aggiornamento costante dei dati, l'Address-transaction graph rappresenta una soluzione bilanciata che consente un'importazione efficiente dei dati senza sacrificare la ricchezza informativa. Al contrario, il Payment graph è più indicato per analisi dettagliate del flusso di valore, a costo di tempi di importazione e di occupazione della memoria leggermente maggiori. Infine, l'Address graph con valori aggregati si propone come soluzione per analisi ad alto livello, ideale per utenti finali o analisti che desiderano una visione comprensiva e semplificata della rete Bitcoin senza necessità di aggiornamenti tempestivi.

Questi risultati sottolineano l'importanza di considerare non solo le prestazioni tecniche di ciascuna rappresentazione ma anche le implicazioni pratiche derivanti dalla loro applicazione in contesti reali. La scelta tra i modelli non dovrebbe quindi basarsi esclusivamente su metriche come il numero di nodi o le dimensioni del database ma dovrebbe tenere conto degli obiettivi dell'analisi, del contesto applicativo e delle esigenze degli utenti finali. In sintesi, questa sezione sperimentale non solo illustra le capacità e i limiti di ciascuna rappresentazione della blockchain Bitcoin come grafo ma fornisce anche linee guida per orientare la scelta del modello più adeguato a seconda delle specifiche esigenze analitiche e applicative.

Capitolo 8

Caso d’uso: Address Taint Analysis

In questo capitolo conclusivo vengono esplorate usando Neo4j le tecniche di “Address taint analysis” e “Backward address taint analysis”, descritte in [60] dal punto di vista teorico ossia (senza fornirne una valutazione sperimentale), che sono applicabili al tracciamento dei flussi di valore che sono stati sottoposti a servizi di mixing, ovvero servizi nati con lo scopo di offuscare la provenienza dei fondi.

L’utilizzo di Neo4j per implementare queste analisi si presta perfettamente alle peculiarità dei grafi usati per rappresentare la blockchain di Bitcoin, consentendo interrogazioni complesse e analisi approfondite grazie alla flessibilità nell’esplorare le relazioni tra nodi altamente interconnessi. Attraverso esempi concreti viene mostrato nel corso di questo capitolo come sia possibile realizzare sia la *address taint analysis* che la *backward address taint analysis* usando Cypher ed applicando queste tecniche alle tre rappresentazioni della blockchain di Bitcoin come grafo discusse in questa tesi: l’Address-transaction Graph, il Payment graph e l’Address graph con valori aggregati.

La taint analysis tradizionale (detta anche *transaction taint analysis* [60]) si concentra sul tracciamento di specifici amount Bitcoin contaminati attraverso le transazioni. Questo è realizzato isolando i movimenti di valuta sospetta ma non considerando l’impatto su altri Bitcoin posseduti dallo stesso indirizzo a meno che non siano coinvolti nella stessa transazione. Al contrario, la *address taint analysis*, proposta in [60], sposta l’attenzione dal livello transazionale al livello di connessione tra indirizzi, operando sotto l’assunzione che qualsiasi indirizzo che riceva Bitcoin da indirizzi contaminati diventi a sua volta contaminato, influenzando così ogni Bitcoin posseduto da tale indirizzo nel tempo.

Un’interessante estensione di questa metodologia è la *backward address taint analysis*, tecnica che opera andando a marcare come contaminati tutti gli indirizzi che hanno inviato Bitcoin a indirizzi contaminati, considerando quindi contaminati anche tutti i Bitcoin posseduti da tali indirizzi nel corso della loro esistenza. Questa tecnica è particolarmente utile per identificare i flussi di valore che hanno contribuito a contaminare un indirizzo specifico, consentendo di tracciare all’indietro il valore contaminato fino a raggiungere la sua origine in una transazione coinbase.

Gli esempi presentati in questo capitolo illustrano come sia possibile usando Neo4j:

1. identificare i nodi contaminati dall’*address taint analysis* e dalla *backward address taint analysis*;
2. fornire i cammini che collegano un nodo contaminato mittente con un destinatario scelto.

Questo capitolo non solo mette in luce la potenzialità di Neo4j nel tracciare e analizzare movimenti di valore contaminato all’interno delle blockchain ma apre anche la strada a future ricerche e applicazioni di queste metodologie su casi reali, fornendo potenzialmente nuovi strumenti per la lotta contro attività illecite che sfruttano le criptovalute.

8.1 Implementazione in Neo4j

8.1.1 Address-transaction graph

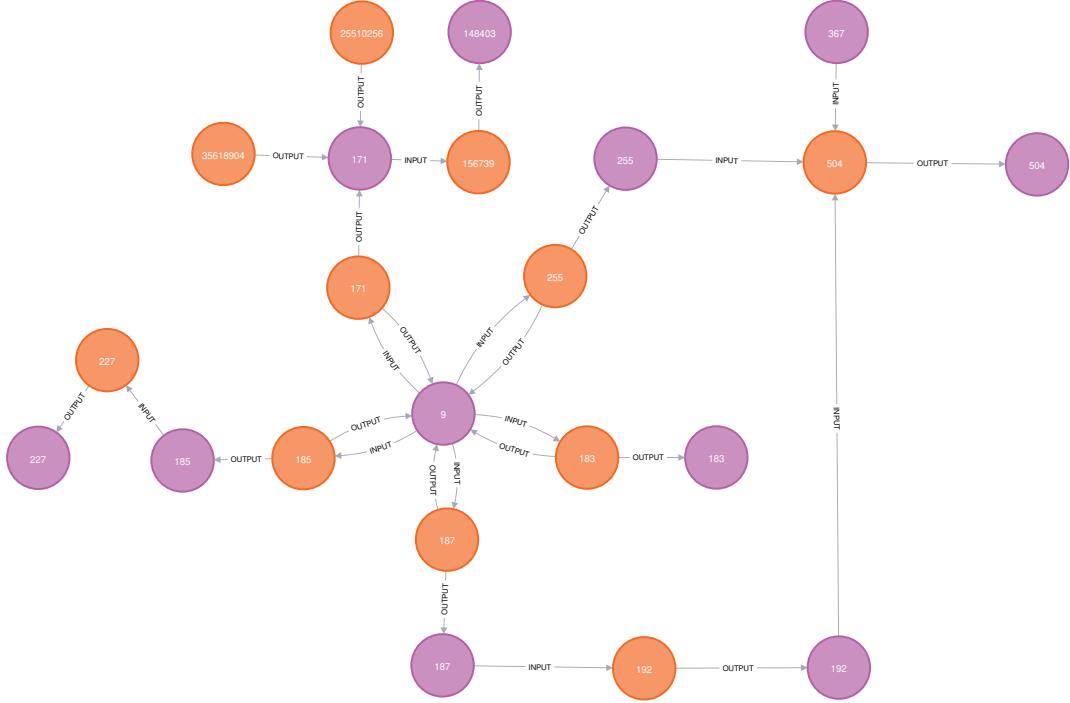


Figura 8.1: Porzione semplificata di un Address-transaction graph, i nodi arancioni rappresentano transazioni (**Transaction**) e i nodi lilla indirizzi (**Address**). Le analisi che seguono considerano l'intero grafo costruito a partire dai dati a disposizione della blockchain Bitcoin, non solo la porzione mostrata in figura che è a solo scopo illustrativo.

Si può osservare dall'esempio in figura 8.1 come intorno al nodo con etichetta 9 si dispiega una rete di nodi arancioni e lilla che rappresentano rispettivamente transazioni (nodi **Transaction**) e altri indirizzi (nodi **Address**) con i quali il nodo ha avuto interazioni dirette o indirette (a seconda della distanza dal nodo 9). I nodi **Transaction** sono intermediari che riflettono le operazioni di scambio di valore tra indirizzi: ogni nodo transazione (etichettato con un identificatore numerico della transazione) connette uno o più input (indirizzi mittenti) a uno o più output (indirizzi destinatari). Queste connessioni sono raffigurate dalle etichette di relazione INPUT o OUTPUT a seconda del flusso di valore. Per esempio, il nodo **Transaction** 504 riceve tre input da tre indirizzi

diversi (i nodi 255, 367 e 192) e invia un output all'indirizzo 504.

Sempre riferendosi alla figura 8.1, si ipotizzi per gli esempi che seguono che il nodo con etichetta 9 (cioè `addressId = 9`), posizionato centralmente nella figura e contraddistinto da un colore lilla, sia sospetto e si voglia tracciare il flusso di valore contaminato fino ad una determinata profondità di connessioni. Si ricorda che per essere considerato contaminato un indirizzo deve:

1. essere direttamente coinvolto come destinatario in una transazione con il nodo contaminato originale come mittente (in questo caso 9), come nel caso dei nodi `Address 183, 185, 187, 171 e 255`;

oppure

2. aver ricevuto valore da un indirizzo che a sua volta è stato contaminato, come ad esempio nel caso degli indirizzi 227 (contaminato da 185) e 504 (contaminato da 255).

Il codice Cypher qui di seguito sfrutta la procedura `expandConfig` della libreria APOC di Neo4j per estendere le funzionalità di base del database a grafo in modo da eseguire una *address taint analysis* partendo dal nodo 9 e tracciando il flusso di valore contaminato fino al raggiungimento di una massima profondità specificata, indicata con il parametro `maxLevel`. Questa procedura è particolarmente adatta per attraversare dinamicamente il grafo in maniera flessibile e configurabile, consentendo di specificare criteri di ricerca personalizzati durante l'espansione dei cammini.

```

1   WITH 9 AS startAddressId, 504 AS endAddressId
2   MATCH (start:Tainted {addressId: startAddressId})
3   CALL apoc.path.expandConfig(start, {
4       labelFilter: '+Address|-Tainted,+Transaction',
5       relationshipFilter: 'INPUT>,OUTPUT>',
6       uniqueness: 'RELATIONSHIP_PATH',
7       minLevel: 1,
8       maxLevel: 6
9   }) YIELD path
10  WITH [node in nodes(path) WHERE "Address" in labels(node) |
11      → node.addressId] AS taintedAddressIds
12  WITH COLLECT(taintedAddressIds) AS taintedAddressIds
13  WITH REDUCE(s = [], x IN taintedAddressIds | s + x) AS taintedAddressIds
14  WITH apoc.coll.toSet(taintedAddressIds) AS taintedAddressIds
15  UNWIND taintedAddressIds AS addressId
16  MATCH (toTaint:Address {addressId: addressId})
17  SET toTaint:Tainted
18  RETURN toTaint

```

Una spiegazione dettagliata del codice e delle sue componenti principali è la seguente:

- **WITH 9 AS startAddressId** definisce il valore numerico dell'identificatore dell'indirizzo di partenza della contaminazione (in questo caso 9)
- **MATCH (start:Tainted {addressId: startAddressId})** seleziona il nodo Tainted di partenza della contaminazione (cioè il solo nodo 9)
- **CALL apoc.path.expandConfig(start, {...}) YIELD path** esegue l'espansione del grafo a partire dal nodo di partenza **start** seguendo le relazioni INPUT che collegano i nodi Address a Transaction e le relazioni OUTPUT che collegano i nodi Transaction a Address. L'espansione del grafo è configurata per proseguire fino al raggiungimento del livello massimo di profondità specificato
 - **labelFilter: '+Address|-Tainted,+Transaction'** specifica un filtro di etichette che impone che i cammini includano prima un nodo con l'etichetta Address e poi un nodo con etichetta Transaction ma escludendo i nodi con etichetta Tainted (cioè i nodi già contaminati) lungo il percorso, ripetendo questo schema fino al raggiungimento del livello massimo di profondità specificato. Il filtro esclude i nodi Tainted per evitare cammini che tornano indietro verso nodi già contaminati, mantenendo l'espansione del grafo in avanti (si veda

ad esempio le relazioni **INPUT** che ha il nodo 9 con transazioni il cui output è destinato a 9 stesso)

- **relationshipFilter:** 'INPUT>,OUTPUT>' filtra le relazioni per espandere solo quelle che escono da un nodo **Address** (relazioni **INPUT**) e quelle che escono da un nodo **Transaction** (relazioni **OUTPUT**), mantenendo l'ordine del flusso del valore nella blockchain. Le frecce > indicano la direzione di movimento attraverso le relazioni in modo da seguire il flusso del valore che si muove attraverso la rete
 - **uniqueness:** 'RELATIONSHIP_PATH' impone che per ogni cammino venga considerata solo una volta ciascuna relazione, cammini diversi possono comunque condividere le stesse relazioni
 - **minLevel:** 1, **maxLevel:** 6 specificano il livello minimo e massimo di profondità della ricerca dal nodo di partenza (in questo caso il nodo **Address** 9)
- Il frammento di codice che segue **YIELD path** (righe 10-17) estrae gli **addressId** dei nodi **Address** dai cammini trovati, unisce questi identificatori numerici in un'unica lista rimuovendo i duplicati (perchè un indirizzo che viene contaminato può essere raggiunto da più cammini, si veda ad esempio il nodo **504** raggiungibile in 2 modi diversi dall'indirizzo **9**) e infine marca come contaminati (aggiungendo l'etichetta **Tainted**) tutti i nodi **Address** identificati

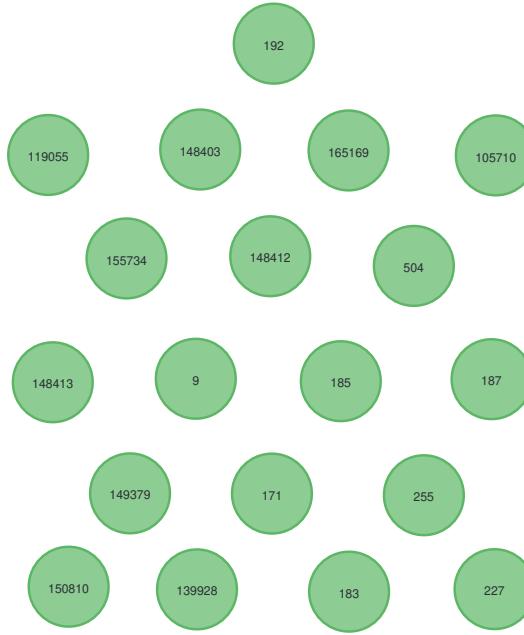


Figura 8.2: I 19 nodi **Address** contaminati individuati con l'address taint analysis effettuata sull'Address-transaction graph, partendo dal nodo 9 mostrato nello snapshot del grafo in figura 8.1

In figura 8.2 è possibile osservare i nodi **Tainted** individuati con l'esecuzione del codice Cypher sopra riportato, si noti ad esempio che il nodo **Address** 504 è stato contaminato. Per ottenere i collegamenti tra il nodo 9 e il nodo 504 attraverso i nodi **Address** contaminati individuati con l'*address taint analysis*, si possono sostituire le righe 10-17 del codice precedente con il seguente frammento di codice (avendo cura di avere solo il nodo 9 marcato come **Tainted** per poter ripetere l'analisi):

```

1 WHERE ANY(node IN nodes(path) WHERE node:Address AND node.addressId =
    ↳ 504)
2 RETURN path

```

Ottenendo così i 2 cammini visualizzati in figura 8.3, ovvero la sequenza di nodi **Address** (in lilla) 9 → 255 → 504 e 9 → 187 → 192 → 504.

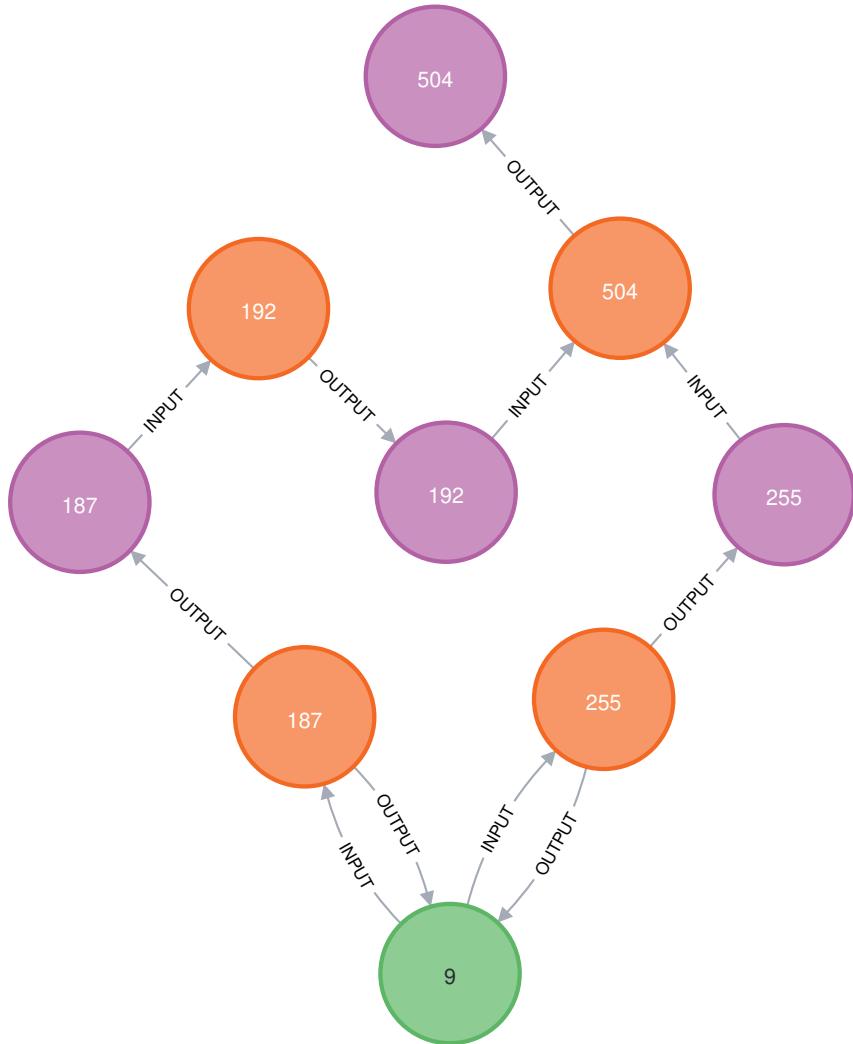


Figura 8.3: Cammini con profondità massima 6 che collegano il nodo 9 in figura 8.1 al nodo 504 attraverso i nodi **Address** contaminati individuati con l'address taint analysis nel grafo Address-Transaction.

In maniera analoga è possibile effettuare la *backward address taint analysis* partendo da un nodo **Address** contaminato e tracciando a ritroso il flusso di valore fino a una determinata profondità di connessioni.

Ipotizzando ora che il nodo 504 sia sospetto e si voglia tracciare l'origine del flusso di valore contaminato fino a una determinata profondità, si può utilizzare il seguente codice Cypher:

```

1 WITH 504 AS startAddressId
2 MATCH (start:Tainted {addressId: startAddressId})
3 CALL apoc.path.expandConfig(start, {
4     labelFilter: '+Address|-Tainted,+Transaction',
5     relationshipFilter: '<OUTPUT,<INPUT',
6     uniqueness: 'RELATIONSHIP_PATH',
7     minLevel: 1,
8     maxLevel: 6
9 }) YIELD path
10 WITH [node IN nodes(path) WHERE "Address" IN labels(node) |
11   → node.addressId] AS taintedAddressIds
12 WITH COLLECT(taintedAddressIds) AS taintedAddressIds
13 WITH REDUCE(s = [], x IN taintedAddressIds | s + x) AS taintedAddressIds
14 WITH apoc.coll.toSet(taintedAddressIds) AS taintedAddressIds
15 UNWIND taintedAddressIds AS addressId
16 MATCH (toTaint:Address {addressId: addressId})
17 SET toTaint:Tainted
18 RETURN toTaint

```

Il codice è molto simile a quello usato per l'*address taint analysis* ma con la differenza che rispetto alle righe 4-5 del codice in sezione 8.1.1 le relazioni INPUT e OUTPUT sono invertite in modo che l'espansione del grafo avvenga a ritroso rispetto al flusso di valore. Ciò consente di seguire al contrario, da un indirizzo, le transazioni che hanno inviato valore a tale indirizzo e poi i nodi Address che hanno immesso valore in queste transazioni.

Anche in questo caso, sostituendo alle righe 10-17 del codice in sezione 8.1.1 il seguente frammento di codice è possibile ottenere i cammini che collegano il nodo 504 a uno specifico nodo Tainted individuato con la *backward address taint analysis*, come ad esempio il nodo 9:

```

1 WHERE ANY(node IN nodes(path) WHERE node:Address AND node.addressId = 9)
2 RETURN path

```

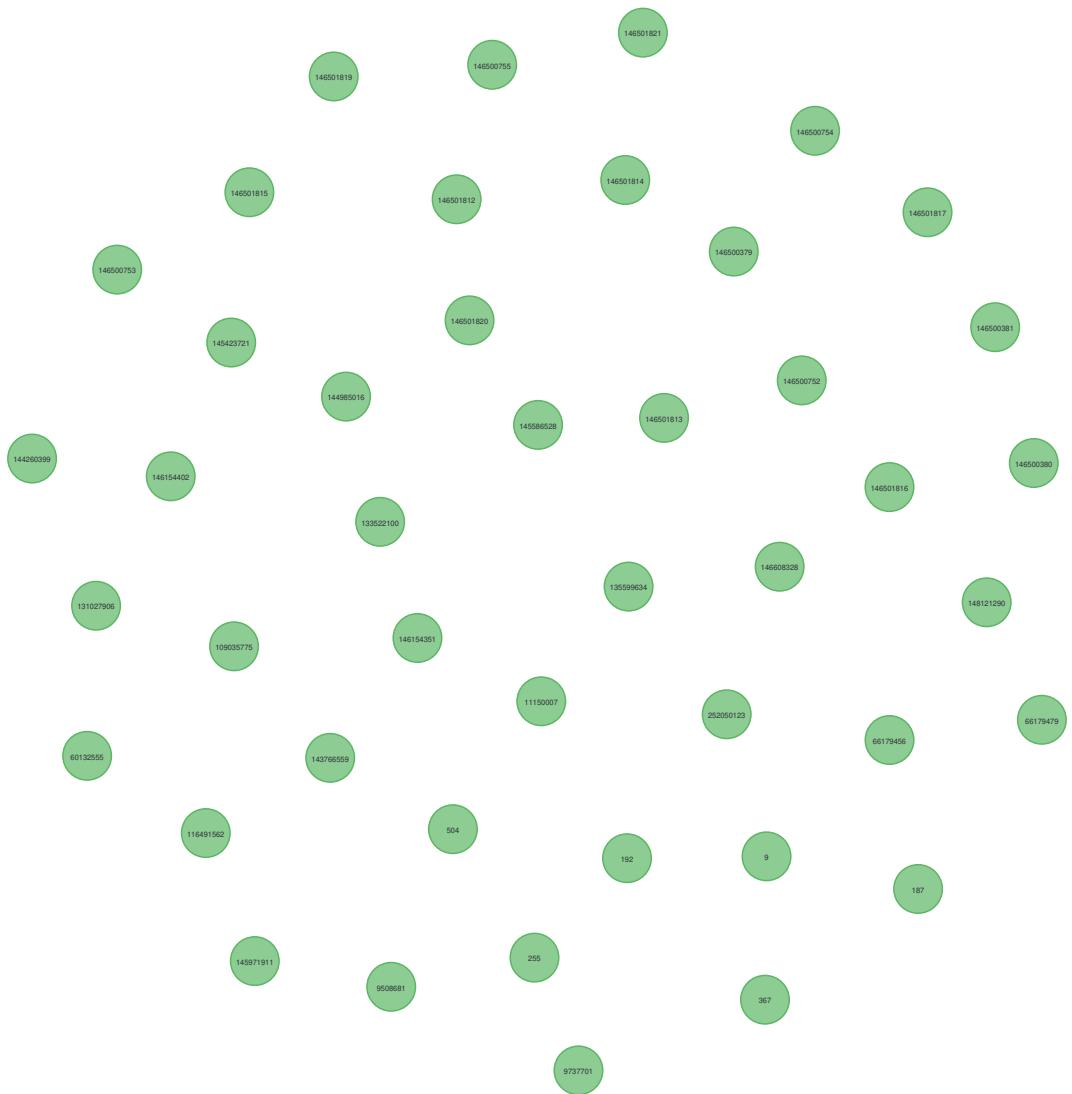


Figura 8.4: I 44 nodi Address contaminati individuati con la backward address taint analysis effettuata sull'Address-transaction graph, partendo dal nodo 504 mostrato nello snapshot del grafo in figura 8.1.

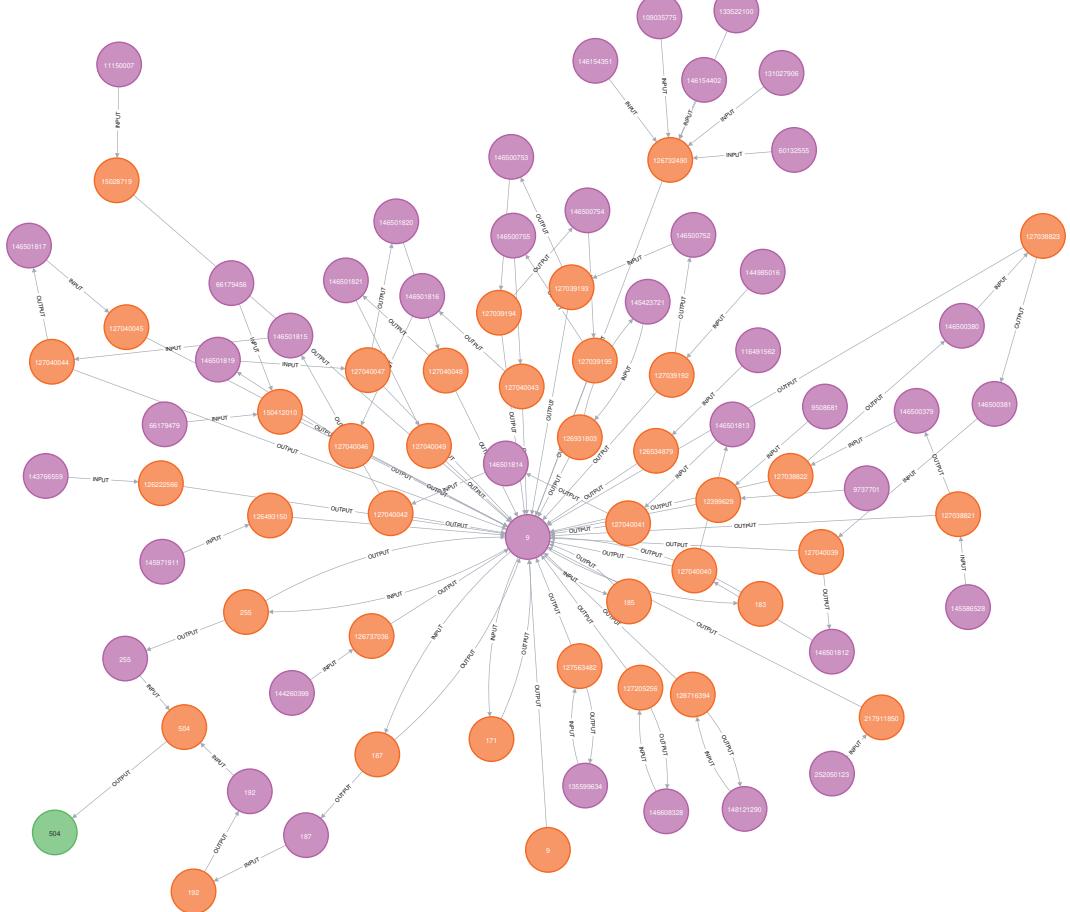


Figura 8.5: Cammini con profondità massima 6 che collegano il nodo 504 in figura 8.1 al nodo 9 attraverso i nodi **Address** contaminati individuati con la backward address taint analysis nel grafo Address-Transaction.

Come si può osservare dalle figure 8.4 e 8.5, la *backward address taint analysis* individua, in questo caso, un numero maggiore di nodi Address contaminati (44) rispetto all'*address taint analysis* (19). Nella *backward address taint analysis* si traccia a ritroso il flusso di valore seguendo le relazioni OUTPUT in entrata ai nodi Address contaminati e risalendo potenzialmente (a seconda della profondità specificata) fino all'origine della valuta in una transazione coinbase. Come è possibile osservare dalla figura 8.5 gli unici due cammini che collegano il nodo 504 al nodo 9 attraverso i nodi Address contaminati (mediante le transazioni 255 e 187 in basso a sinistra, in prossimità del nodo verde corrispondente al nodo Address 504) corrispondono agli stessi cammini individuati con l'*address taint analysis*.

8.1.2 Payment graph

In questa sezione viene esplorato come applicare l'*address taint analysis* e la *backward address taint analysis* al Payment graph, la rappresentazione della blockchain di Bitcoin come grafo in cui i nodi TXO rappresentano output di transazioni e dove le relazioni **CONTRIBUTES** tra nodi TXO indicano che un output di una transazione è stato usato come input in un'altra transazione.

Un esempio di Payment graph corrispondente a parte del grafo Address-transaction in figura 8.1 è mostrato in figura 8.6, il numero presente su ciascun nodo rappresenta l'identificatore numerico (**addressId**) del destinatario di un output di una transazione.

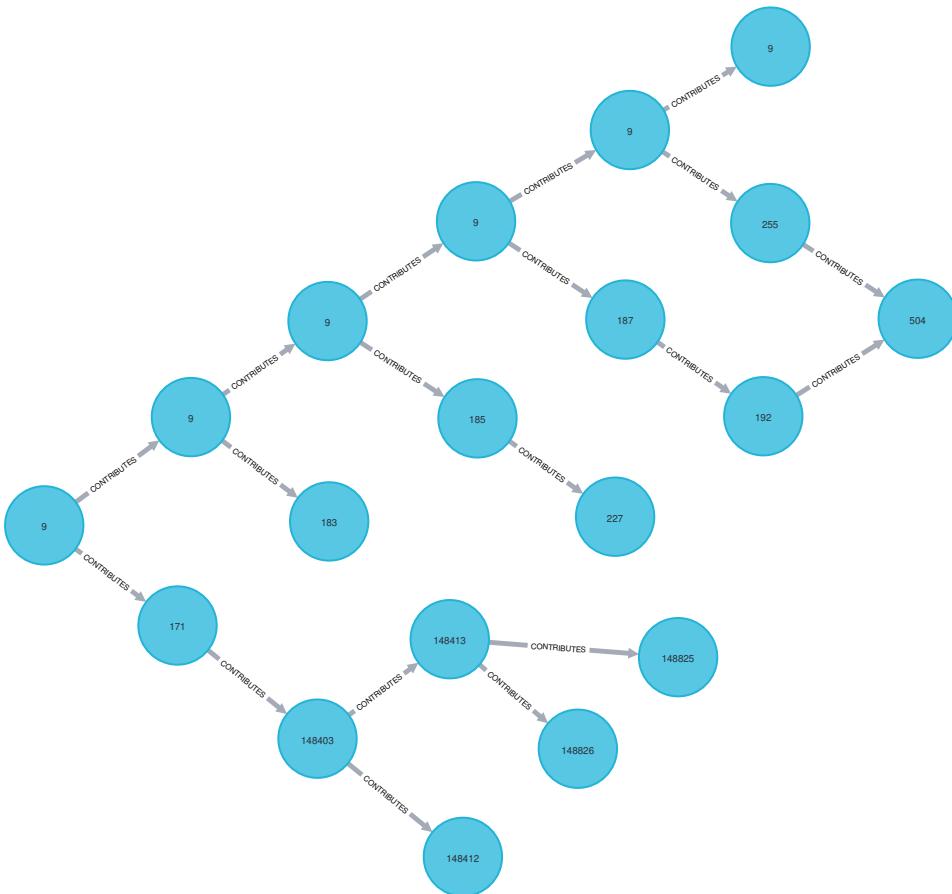


Figura 8.6: Porzione di un Payment graph, i nodi blu rappresentano output di transazioni (TXO) e la presenza di una relazione **CONTRIBUTES** tra due nodi indica che compaiono nella stessa transazione, uno come input e l'altro come output rispettivamente.

L'applicazione dell'*address taint analysis* e della *backward address taint analysis* al Payment graph richiede una modifica del codice Cypher rispetto a quanto mostrato per l'Address-transaction graph in quanto le relazioni di trasferimento di valore tra indirizzi, seppur descrivano lo stesso flusso di valuta, sono rappresentate in modo diverso. Nel caso dell'*address taint analysis* per il Payment graph è necessario per prima cosa identificare i nodi TXO destinati a uno specifico indirizzo (che si ipotizza contaminato) e poi tracciare, per ciascun nodo TXO di proprietà dell'indirizzo contaminato, la catena di contribuzione di tale output alla creazione di altri output in transazioni successive. Una volta che un nodo TXO contaminato è stato identificato, tutti gli output di proprietà dello stesso indirizzo vengono considerati contaminati.

Il codice Cypher che segue mostra come realizzare la procedura appena descritta, partendo dai nodi TXO di proprietà dell'indirizzo 9 e andando a marcare come contaminati tutti i nodi TXO di indirizzi che hanno ricevuto Bitcoin da indirizzi contaminati, fino al raggiungimento di una massima profondità specificata.

```

1  WITH 9 AS startAddressId
2  MATCH (startNode:TXO {addressId: startAddressId})
3  CALL apoc.path.expandConfig(startNode, {
4      labelFilter: '-Tainted',
5      relationshipFilter: "CONTRIBUTES>",
6      minLevel: 1,
7      maxLevel: 6
8  }) YIELD path
9  WITH [node in nodes(path) | node.addressId] AS addressIds
10 WITH COLLECT(addressIds) AS addressIds
11 WITH REDUCE(s = [], x IN addressIds | s + x) AS addressIds
12 WITH apoc.coll.toSet(addressIds) AS addressIds
13 UNWIND addressIds AS addressId
14 MATCH (txo:TXO) WHERE txo.addressId = addressId
15 SET txo:Tainted
16 RETURN txo

```

Il codice sopra riportato è simile a quello usato per l'Address-transaction graph nella sezione 8.1.1 ma con la differenza che

- **MATCH** (startNode:TXO {addressId: startAddressId}) non seleziona più un singolo nodo **Address** come nel caso dell'Address-transaction graph ma potenzialmente più nodi TXO di proprietà dell'indirizzo contaminato (perchè un indirizzo può essere coinvolto come destinatario in più transazioni, ciascuna delle quali crea un nodo TXO)

- **labelFilter:** '-Tainted' specifica che i cammini non devono includere nodi TXO già contaminati, questo è fondamentale dato che per ogni nodo TXO appartenente all'indirizzo contaminato `startAddressId` si traccia la contribuzione di tale output alla creazione di nuovi output, andando a marcare questi ultimi come contaminati (aggiungendo cioè l'etichetta `Tainted`). Dato che l'operazione di espansione del grafo è ripetuta per ciascun nodo TXO di proprietà dell'indirizzo contaminato, è possibile che vengano incontrati più volte lungo i cammini esplorati dei nodi che sono già stati marcati come contaminati in precedenza, motivo per cui, al fine di evitare esplorazioni ridondanti, è necessario interrompere l'espansione del grafo quando si incontra un nodo `Tainted` ovvero un nodo TXO già contaminato
- **relationshipFilter:** "CONTRIBUTES>" specifica che l'espansione del grafo deve avvenire seguendo le relazioni `CONTRIBUTES` in uscita dai nodi TXO, così da tracciare la catena di contribuzione di ciascun TXO alla creazione di nuovi output

L'esecuzione del codice sopra riportato porta a marcare come contaminati 585 nodi TXO appartenenti a 21 indirizzi diversi. Una visualizzazione dei primi 300 nodi TXO contaminati è mostrata in figura 8.7.

Anche in questo caso, seppur difficili da visualizzare, sono stati contaminati nodi TXO appartenenti all'indirizzo 504 (come quello nella catena di nodi collegati tra loro visibile in basso a destra). Per ottenere i cammini che collegano i nodi TXO di proprietà dell'indirizzo 9 contaminato ai nodi TXO di proprietà di un indirizzo specifico, come ad esempio l'indirizzo 504, si può modificare, come già fatto per l'Address-transaction graph, il codice sopra riportato e sostituire le righe 9-16 con il seguente frammento di codice:

```

1 WHERE ANY(node IN nodes(path) WHERE node.addressId = 504)
2 RETURN path

```

Ottenendo così i cammini visualizzati in figura 8.8 che collegano i nodi TXO di proprietà dell'indirizzo 9 ai nodi TXO di proprietà dell'indirizzo 504.

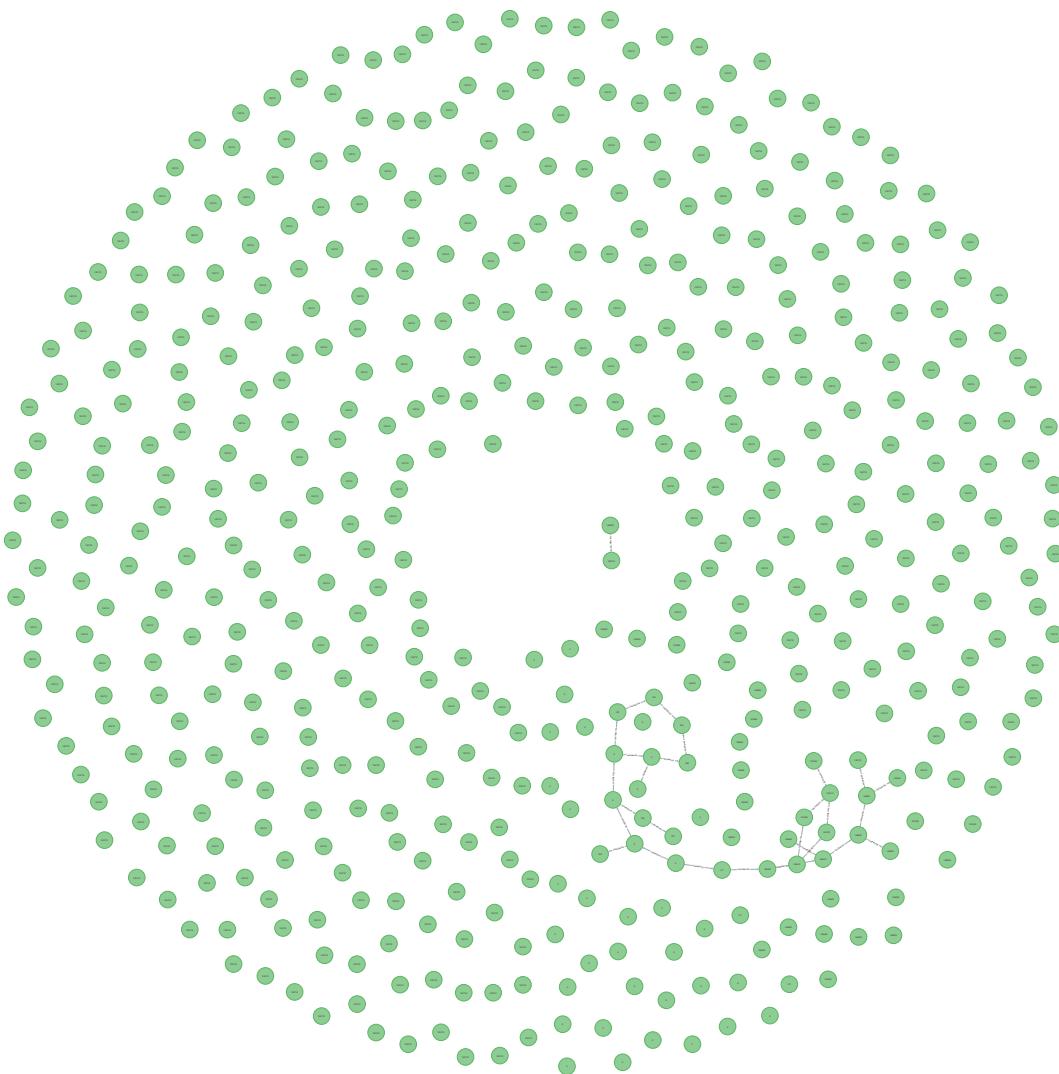


Figura 8.7: I primi 300 nodi TX0 contaminati dei 585 individuati con l'address taint analysis effettuata sul Payment graph, partendo dal nodo 9 mostrato in figura 8.6.

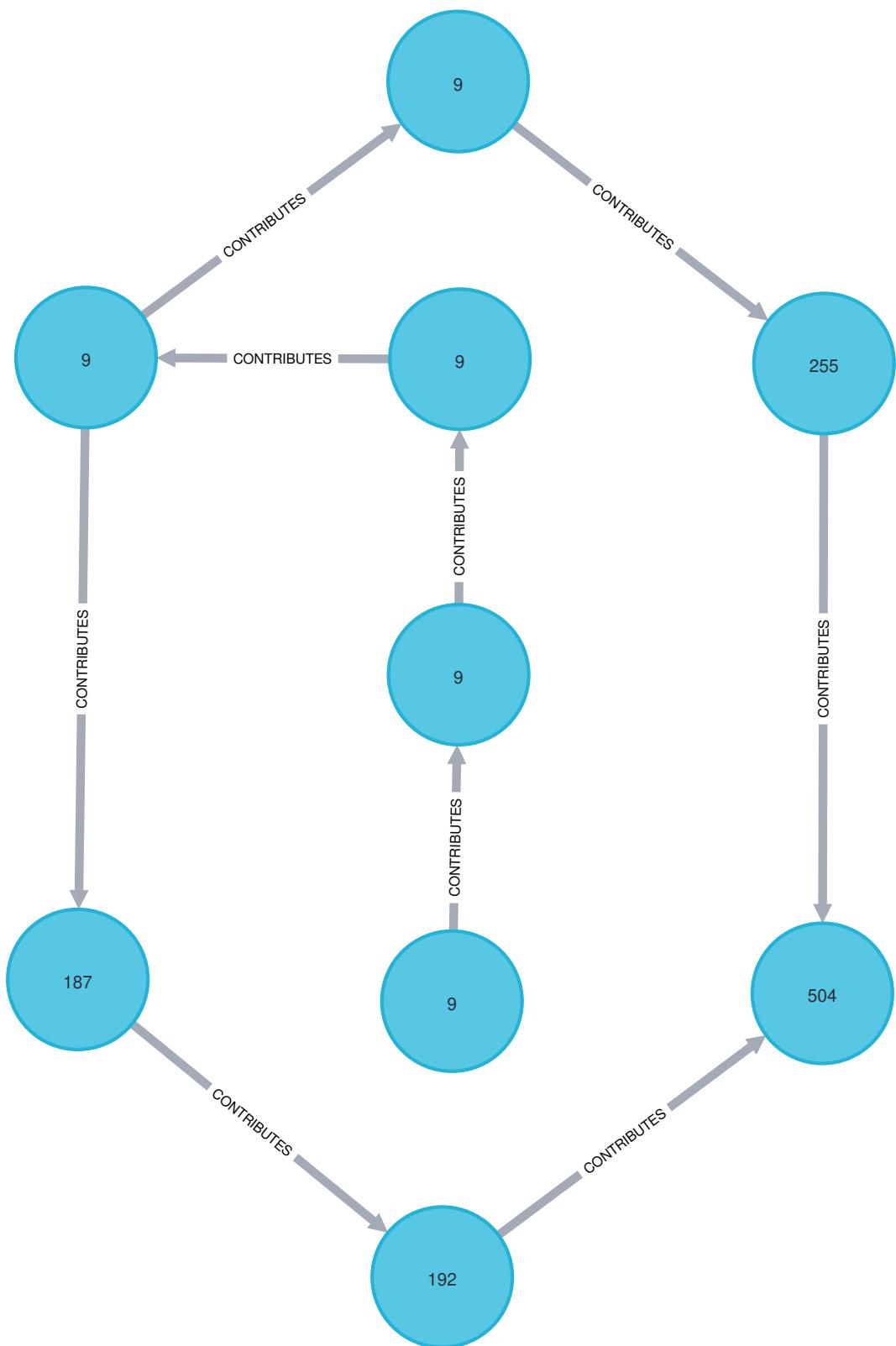


Figura 8.8: Cammini con profondità massima 6 che collegano i nodi TX0 di proprietà dell'indirizzo 9 in figura 8.6 al nodo TX0 di proprietà dell'indirizzo 504 attraverso i nodi TX0 contaminati individuati con l'address taint analysis nel Payment graph.

Per quanto riguarda la *backward address taint analysis* nel Payment graph, il codice Cypher è simile a quello usato in precedenza per l'*address taint analysis* ma con la differenza che le relazioni `CONTRIBUTES` non sono più percorse in avanti ma a ritroso, seguendo i contributi che un nodo `TXO` ha ricevuto da altri output in transazioni precedenti, risalendo (se la profondità lo permette) fino all'origine della valuta in una transazione coinbase.

Il codice Cypher che consente di realizzare la *backward address taint analysis* nel Payment graph considerando i nodi `TXO` di proprietà dell'indirizzo `504` come contaminati è il seguente:

```

1 WITH 504 AS startAddressId
2 MATCH (startNode:TXO {addressId: startAddressId})
3 CALL apoc.path.expandConfig(startNode, {
4     labelFilter: '-Tainted',
5     relationshipFilter: "<CONTRIBUTES>",
6     minLevel: 1,
7     maxLevel: 6
8 }) YIELD path
9 WITH [node in nodes(path) | node.addressId] AS addressIds
10 WITH COLLECT(addressIds) AS addressIds
11 WITH REDUCE(s = [], x IN addressIds | s + x) AS addressIds
12 WITH apoc.coll.toSet(addressIds) AS addressIds
13 UNWIND addressIds AS addressId
14 MATCH (txo:TXO) WHERE txo.addressId = addressId
15 SET txo:Tainted
16 RETURN txo

```

I nodi `TXO` contaminati individuati con la *backward address taint analysis* sono 42 e sono mostrati in figura 8.9. Si può osservare che la maggior parte dei nodi contaminati sono riconducibili all'indirizzo 9 e che risultano presenti (in basso a destra) i due cammini che collegano il nodo `TXO` di proprietà dell'indirizzo `504` ai nodi `TXO` di proprietà dell'indirizzo 9, già individuati con l'*address taint analysis* nel Payment graph in figura 8.8 e nell'Address-transaction graph in figura 8.5.

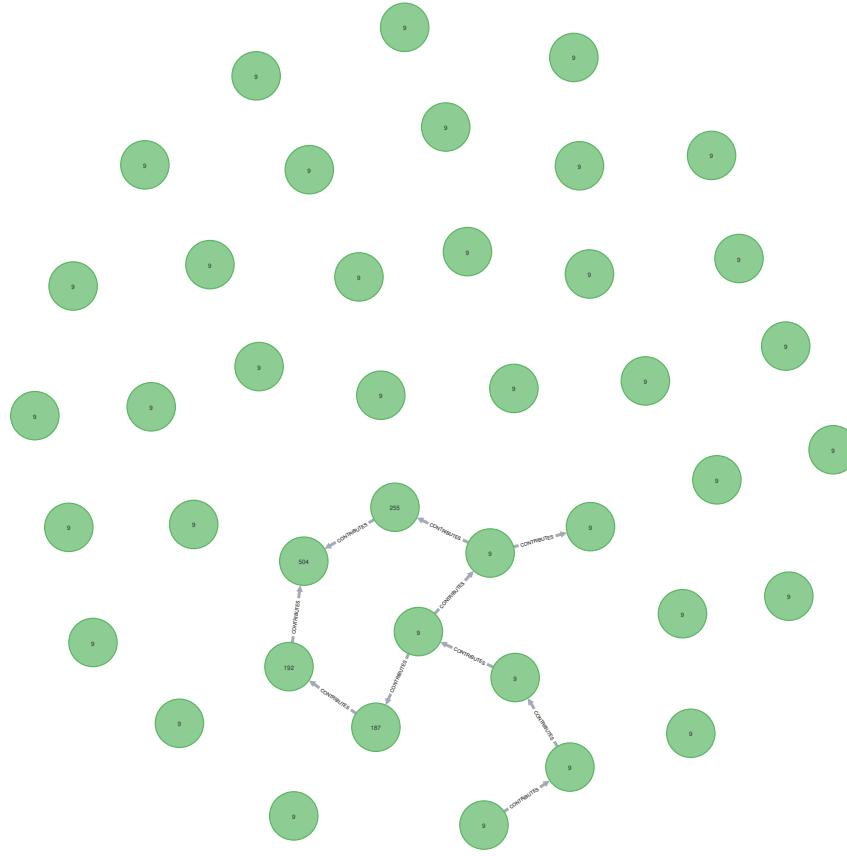


Figura 8.9: I 42 nodi TXO contaminati individuati con la backward address taint analysis effettuata sul Payment graph, partendo dal nodo 504 mostrato in figura 8.6.

8.1.3 Address graph con valori aggregati

L'ultima rappresentazione della blockchain di Bitcoin come grafo dove verrà implementata l'*address taint analysis* e la *backward address taint analysis* è l'Address graph con valori aggregati. In questa rappresentazione i nodi **Address** rappresentano indirizzi Bitcoin e le relazioni **TRANSFERS_TO** tra nodi **Address** indicano che un indirizzo ha nel tempo trasferito valore a un altro indirizzo. Una descrizione più dettagliata di questa rappresentazione e delle proprietà aggregate presenti sulle relazioni è fornita in sezione 4.4. Un esempio di Address graph con valori aggregati corrispondente agli esempi precedenti di Address-transaction graph e Payment graph è mostrato in figura 8.10, anche in questo caso i numeri presenti nei nodi lilla rappresentano l'identificatore numerico (**addressId**) associato a ciascun indirizzo.

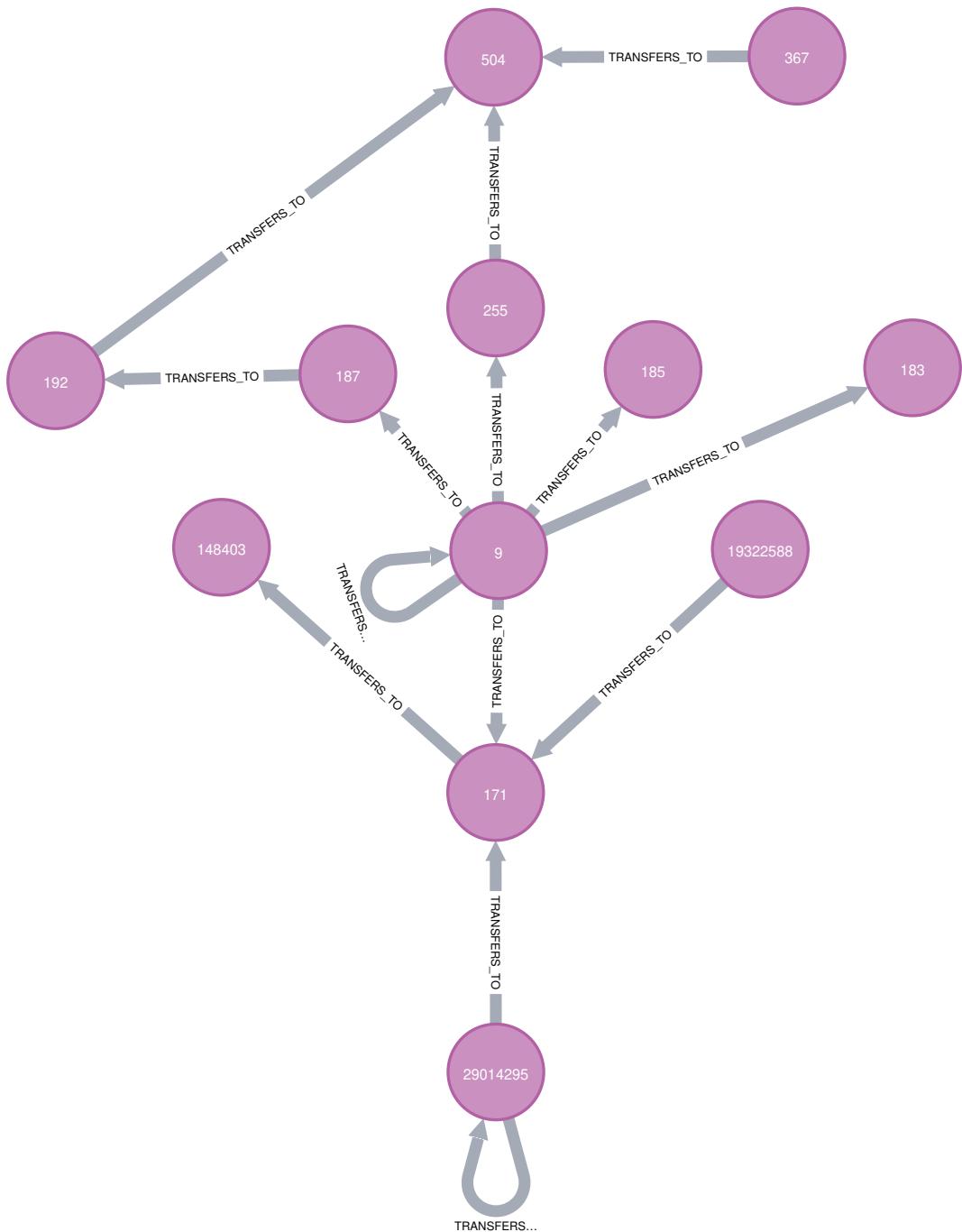


Figura 8.10: Porzione di un Address graph con valori aggregati, i nodi lilla rappresentano indirizzi (Address) e la presenza di una relazione `TRANSFERS_TO` tra due nodi indica che il primo indirizzo ha nel tempo trasferito valore al secondo.

L'implementazione dell'*address taint analysis* in questo caso risulta essere più semplice rispetto al Payment graph in quanto a ciascun `addressId` è associato un unico nodo `Address` e non più nodi TXO come nel caso precedente. Sarà pertanto sufficiente marcare come contaminati i nodi `Address` che hanno ricevuto Bitcoin nel tempo da indirizzi contaminati, seguendo le relazioni `TRANSFERS_TO` in avanti fino al raggiungimento di una massima profondità specificata.

Un esempio di codice Cypher che consente di implementare l'*address taint analysis* nel Address graph con valori aggregati partendo dal nodo 9, che si ipotizza contaminato, è il seguente:

```

1 WITH 9 AS startAddressId
2 MATCH (start:Address {addressId: startAddressId})
3 CALL apoc.path.expandConfig(start, {
4     relationshipFilter: "TRANSFERS_TO",
5     minLevel: 1,
6     maxLevel: 4,
7     uniqueness: 'RELATIONSHIP_GLOBAL'
8 }) YIELD path
9 WITH [node IN nodes(path) | node.addressId] AS addressIds
10 WITH COLLECT(addressIds) AS addressIds
11 WITH REDUCE(s = [], x IN addressIds | s + x) AS addressIds
12 WITH apoc.coll.toSet(addressIds) AS addressIds
13 UNWIND addressIds AS addressId
14 MATCH (toTaint:Address {addressId: addressId})
15 SET toTaint:Tainted
16 RETURN toTaint

```

Il risultato dell'esecuzione del codice sopra riportato porta a marcare come contaminati 476 nodi `Address`. Una visualizzazione di questi nodi e delle relazioni `TRANSFERS_TO` che li collegano è mostrata in figura 8.11.

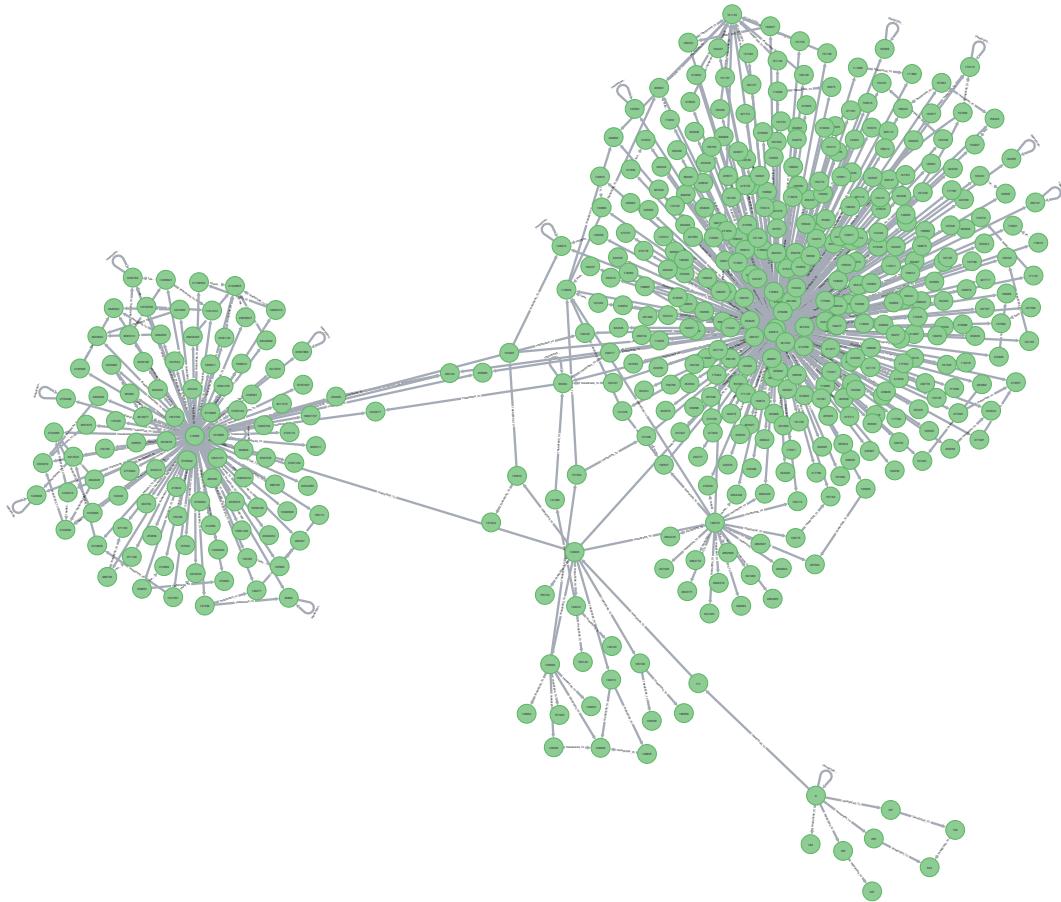


Figura 8.11: I 476 nodi **Address** contaminati individuati con l'address taint analysis sull'Address graph con valori aggregati, partendo dal nodo 9 mostrato in figura 8.10.

Come nel caso degli esempi precedenti, è possibile ottenere i cammini che collegano il nodo 9 a un nodo **Address** specifico, come ad esempio il nodo **Address 504**, attraverso i nodi **Address** contaminati individuati con l'*address taint analysis*. Per ottenere i cammini che collegano il nodo 9 al nodo 504 si può modificare il codice sopra riportato e sostituire le righe 9-16 con il seguente frammento di codice:

```

1 WHERE ANY(node IN nodes(path) WHERE node.addressId = 504)
2 RETURN path

```

Ottenendo così i cammini visualizzati in figura 8.12 che collegano il nodo 9 al nodo 504.

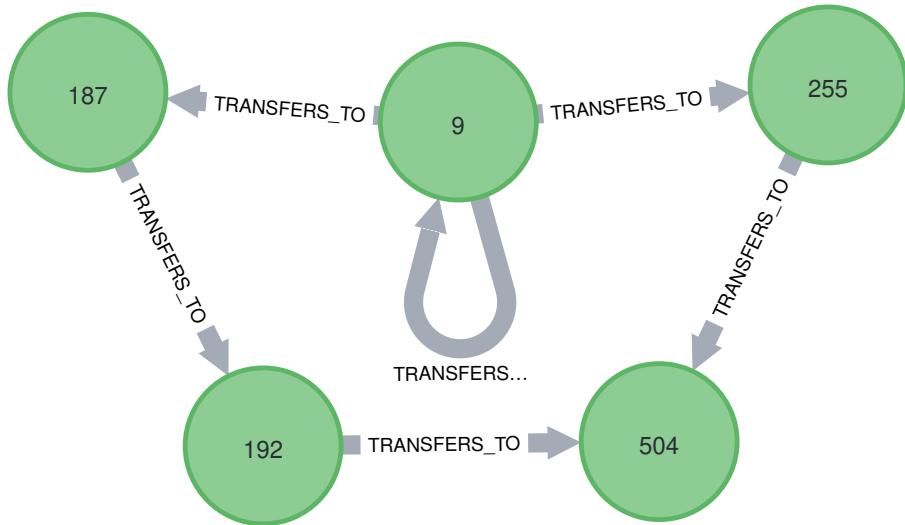


Figura 8.12: Cammini con profondità massima 4 che collegano il nodo 9 in figura 8.10 al nodo 504 attraverso i nodi **Address** contaminati individuati con l'address taint analysis nel Address graph con valori aggregati.

Per concludere, l'implementazione della *backward address taint analysis* nel Address graph con valori aggregati richiede lievi modifiche al codice Cypher usato per l'*address taint analysis* in quanto le relazioni TRANSFERS_TO devono essere seguite a ritroso (con <TRANSFERS_TO>) e non in avanti (TRANSFERS_TO). Il codice Cypher che realizza la *backward address taint analysis* nel Address graph con valori aggregati partendo dal nodo 504, che si ipotizza contaminato, è il seguente:

```

1 WITH 504 AS startAddressId
2 MATCH (start:Address {addressId: startAddressId})
3 CALL apoc.path.expandConfig(start, {
4     relationshipFilter: "<TRANSFERS_TO",
5     minLevel: 1,
6     maxLevel: 3,
7     uniqueness: 'RELATIONSHIP_GLOBAL'
8 }) YIELD path
9 WITH [node IN nodes(path) | node.addressId] AS addressIds
10 WITH COLLECT(addressIds) AS addressIds
11 WITH REDUCE(s = [], x IN addressIds | s + x) AS addressIds
12 WITH apoc.coll.toSet(addressIds) AS addressIds
13 UNWIND addressIds AS addressId
14 MATCH (toTaint:Address {addressId: addressId})
15 SET toTaint:Tainted
16 RETURN toTaint

```

la cui esecuzione porta a marcare come contaminati 44 nodi **Address**. Una visualizzazione di questi nodi e delle relazioni **TRANSFERS_TO** che li collegano è mostrata in figura 8.13. Come è possibile osservare, i cammini che collegano il nodo **504** al nodo **9** attraverso i nodi **Address** individuati con la *backward address taint analysis* sono gli stessi individuati con l'*address taint analysis*, ovvero **9** → **255** → **504** e **9** → **187** → **192** → **504**.

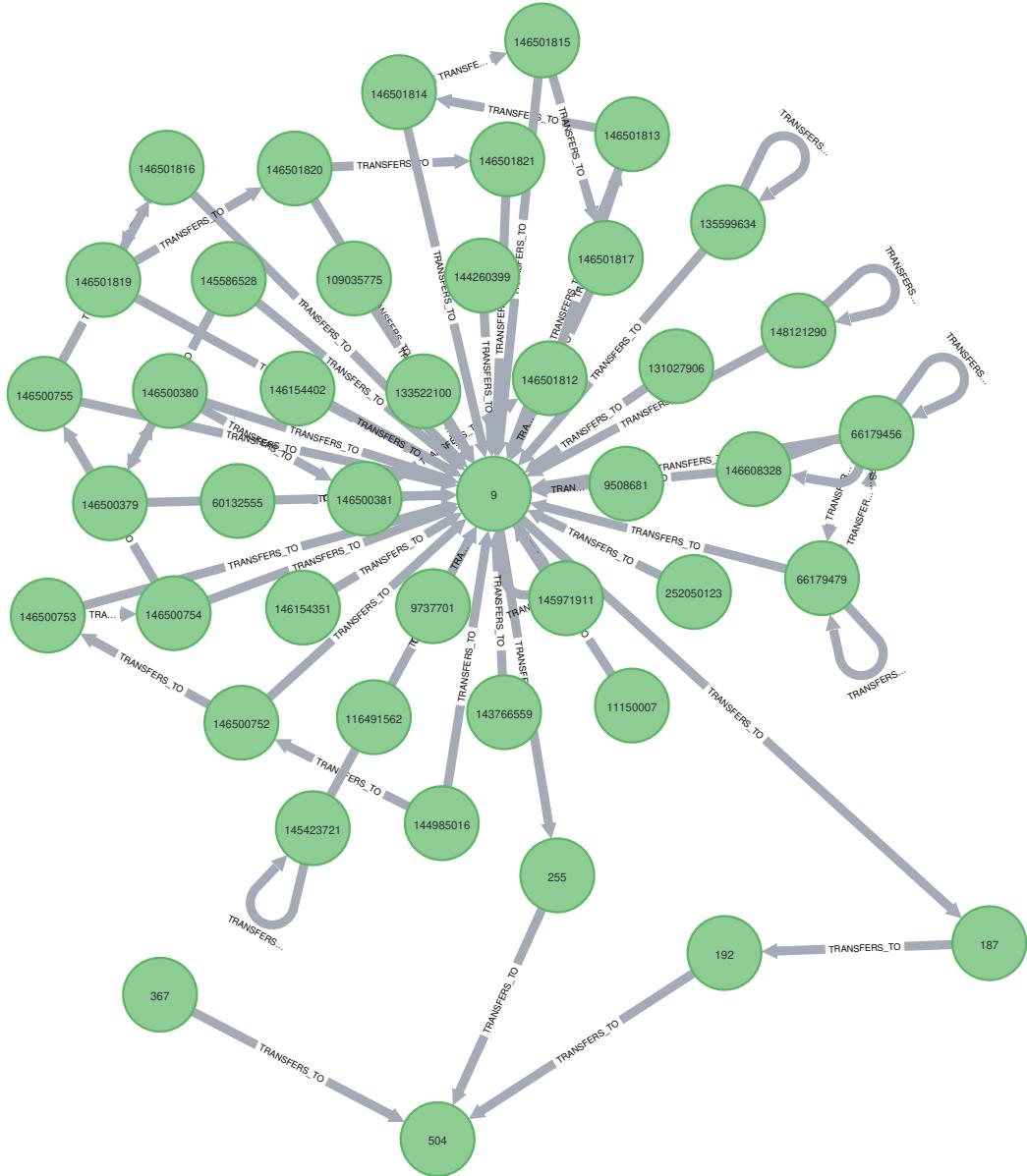


Figura 8.13: I 44 nodi Address contaminati individuati con la backward address taint analysis effettuata sull'Address graph con valori aggregati, partendo dal nodo 504 mostrato in figura 8.10.

8.2 Conclusioni

L'esplorazione delle tecniche di address taint analysis e backward address taint analysis attraverso le tre rappresentazioni della blockchain di Bitcoin

come grafo in Neo4j ha dimostrato l'efficacia di ciascun modello nel tracciare i flussi di valore contaminato. Tuttavia, l'impatto sul processo e sui risultati varia in modo significativo a seconda del modello considerato, evidenziando i diversi punti di forza e le limitazioni di ciascuno.

Nell'Address-transaction graph, la stretta correlazione tra indirizzi e transazioni facilita il tracciamento diretto dei flussi di valore, permettendo un'analisi dettagliata e granulare. Questo modello si è dimostrato particolarmente efficace nell'isolare i percorsi specifici attraverso i quali il valore contaminato si muove, grazie alla sua capacità di rappresentare esplicitamente le relazioni **INPUT** e **OUTPUT** tra transazioni e indirizzi. Tuttavia, la complessità intrinseca del grafo risultante dal dataset del periodo considerato, con un elevato numero di nodi e relazioni, può rendere l'analisi computazionalmente onerosa, specialmente per le reti di grande dimensione.

Il Payment graph, introducendo i nodi **TXO** e le relazioni **CONTRIBUTES**, offre una visione semplificata e diretta dei flussi di valore, trasformando la rete in un grafo aciclico che facilita l'identificazione dei percorsi di contaminazione. Questa rappresentazione semplifica notevolmente l'analisi, riducendo l'ambiguità nella determinazione dell'origine e della destinazione dei fondi. Tuttavia, l'aumento del numero di nodi necessari per rappresentare ogni output di transazione come un nodo separato può portare a una maggiore complessità computazionale, pur offrendo un elevato livello di dettaglio.

Infine, l'Address graph con valori aggregati, pur presentando una visione più astratta e sintetica dei flussi di valore, consente di effettuare analisi ad alto livello con relativa facilità. La riduzione della complessità del grafo, aggregando le transazioni multiple in un'unica relazione **TRANSFERS_TO** tra indirizzi, rende questo modello particolarmente accessibile per analisi rapide e panoramiche delle relazioni di valore. Tuttavia, questa astrazione comporta una perdita di dettaglio che può essere critica per alcune forme di analisi forense come la address taint analysis che richiedono una tracciabilità dettagliata dei flussi di valore, limitando così la capacità di discernere le dinamiche specifiche delle transazioni individuali.

Va sottolineato che, sebbene nel presente capitolo sia stata applicata la metodologia di taint analysis considerando solamente un piccolo sottoinsieme della blockchain Bitcoin (ovvero il sottografo in figura 8.1), quest'approccio è teoricamente estendibile all'intera blockchain. Gli esempi qui presentati hanno lo scopo di offrire una spiegazione semplice e intuitiva, ma è chiaro

che applicando questi metodi a porzioni più ampie della blockchain i risultati ottenuti sarebbero notevolmente più estesi e la complessità nell'interpretazione di questi ultimi aumenterebbe di conseguenza. Questo sottolinea l'importanza di una fase preliminare di scelta consapevole della porzione di blockchain da analizzare, che può concentrarsi su specifiche transazioni, casi sospetti di riciclaggio, o altri aspetti di particolare interesse. Questa considerazione enfatizza ulteriormente la versatilità e l'adattabilità di Neo4j come strumento analitico, pur ricordando la necessità di un approccio metodico e mirato nell'indagine delle reti di transazioni di criptovalute.

Complessivamente, tutti e tre i modelli dimostrano di poter essere utilizzati con successo sia per la taint analysis che per la backward address taint analysis, offrendo strumenti preziosi per il tracciamento dei flussi di valore contaminato all'interno della blockchain di Bitcoin. La scelta del modello più appropriato dipenderà dagli obiettivi specifici dell'analisi, dalla necessità di dettaglio analitico e dalle risorse computazionali disponibili. La comprensione delle peculiarità di ciascuna rappresentazione è fondamentale per sfruttare al meglio le potenzialità di Neo4j nella lotta contro le attività illecite che sfruttano le criptovalute, aprendo la strada a future ricerche e applicazioni pratiche di queste tecniche in contesti reali.

Capitolo 9

Conclusioni e lavori futuri

La tesi ha esplorato la modellazione della blockchain di Bitcoin attraverso tre diverse rappresentazioni come grafo: l'Address-transaction graph, il Payment graph e l'Address graph con valori aggregati. L'implementazione di queste rappresentazioni in Neo4j ha permesso un'analisi approfondita dell'impatto delle scelte di modellazione sulla dimensione del database, sulla velocità di importazione dei dati e sui tempi di risposta delle query. Il dataset utilizzato ha coperto un periodo di studio ampio, da gennaio 2009 ad agosto 2017, coprendo la fase iniziale di sperimentazione della criptovaluta fino alla sua affermazione come realtà consolidata nel panorama delle valute digitali.

Attraverso gli esperimenti condotti è stato possibile evidenziare come le diverse rappresentazioni influenzino significativamente le prestazioni del sistema. In particolare, l'Address graph con valori aggregati, nonostante la sua utilità nel fornire una visione complessiva e dettagliata delle transazioni tra indirizzi, ha mostrato un maggiore onere computazionale e di storage risultando la più onerosa tra le rappresentazioni descritte. D'altra parte, l'Address-transaction graph e il Payment graph hanno presentato una migliore efficienza in termini di tempi di importazione e dimensione del database, pur mantenendo una rappresentazione accurata e dettagliata degli scambi di valore effettuati sulla rete Bitcoin. Tutte e tre le rappresentazioni hanno beneficiato dell'indicizzazione della proprietà `addressId` per accelerare le query di analisi, portando a tempi di esecuzione delle query bassi e costanti nel tempo indipendentemente dalle dimensioni del database, enfatizzando l'importanza di una progettazione attenta degli indici per garantire prestazioni ottimali. Inoltre, per quanto riguarda la address taint analysis e la backward address taint analysis, i tre database hanno permesso di ottenere risultati coerenti e confrontabili riguardo la tracciabilità del flusso di valore tra indirizzi, dimostrando la validità delle rappresentazioni proposte per analisi di questo tipo.

Per quanto riguarda i lavori futuri, ci sono diverse direzioni promettenti che possono essere esplorate per estendere e approfondire l'analisi della blockchain di Bitcoin utilizzando un database a grafo come Neo4j e le rappresentazioni proposte in questa tesi. Alcune possibili aree di sviluppo includono:

- estensione del dataset per includere dati più recenti sulla blockchain di Bitcoin;
- analisi avanzate della rete usando procedure APOC e della suite Graph Data Science (GDS) di Neo4j al fine di identificare pattern e strutture significative all'interno della rete Bitcoin;
- approfondimento della taint analysis e analisi del flusso di valore di transazioni ad-hoc per rilevare attività illecite e frodi.

Estensione del dataset

Un'opportunità interessante per futuri sviluppi è l'estensione del dataset utilizzato in questa tesi per includere dati più recenti sulla blockchain di Bitcoin. L'aggiunta di nuovi blocchi e transazioni potrebbe fornire una visione più aggiornata e completa della rete Bitcoin, consentendo di esplorare le tendenze e i pattern emergenti nel tempo che non sono stati coperti dal dataset attuale. Ciò permetterebbe di rivelare informazioni preziose riguardo le tendenze recenti e l'evoluzione di Bitcoin come asset finanziario, osservando l'effetto di eventi significativi come variazioni normative e fluttuazioni di mercato.

Analisi avanzate della rete usando procedure APOC e GDS

Un'altra direzione interessante è l'utilizzo di procedure APOC [51] e della suite Graph Data Science (GDS) di Neo4j [61] per eseguire analisi avanzate della rete Bitcoin. Le procedure della libreria APOC offrono una vasta gamma di funzionalità tra cui algoritmi per la ricerca di cammini minimi tra coppie di nodi mentre la GDS fornisce algoritmi di analisi su grafi più sofisticati come PageRank [62], community detection e clustering. Questi strumenti possono essere utilizzati per identificare pattern e strutture significative all'interno della rete Bitcoin aiutando a individuare frodi, attività illegali o anomalie. Ad esempio, l'analisi della centralità dei nodi potrebbe rivelare indirizzi particolarmente importanti all'interno della rete mentre l'analisi delle comunità potrebbe identificare gruppi di indirizzi che interagiscono più frequentemente tra loro.

Taint analysis e analisi del flusso di valore

Le tecniche di address taint analysis e backward address taint analysis [60] esplorate in questa tesi, utilizzando Neo4j e considerando un campione ridotto di indirizzi, potrebbero essere utilizzate in un contesto più ampio della blockchain per rilevare attività illecite e frodi all'interno della rete Bitcoin. In particolare, questo tipo di analisi trova applicazione nel tracciare valuta sottoposta a riciclaggio di denaro [63], il finanziamento del terrorismo [64] [65] o il pagamento di riscatti (ad esempio in caso di ransomware, come WannaCry [66] o CryptoLocker [67]). Una possibile estensione di questo lavoro potrebbe essere l'implementazione di una taint analysis più sofisticata che consideri una soglia di taint, cioè una percentuale di valore “contaminato” rispetto al valore totale ricevuto o inviato da un indirizzo. Questo approccio potrebbe fornire una maggiore precisione nell'identificare indirizzi sospetti e transazioni illecite, consentendo di individuare più facilmente attività fraudolente e comportamenti criminali all'interno della rete Bitcoin. Le informazioni derivanti da un'analisi di questo tipo potrebbero fornire prove utili alle autorità competenti per perseguire attività illegali e contrastare l'uso di Bitcoin per scopi malevoli.

Bibliografia

- [1] Jaroslav Pokorný. «Graph Databases: Their Power and Limitations». In: *Computer Information Systems and Industrial Management*. A cura di Khalid Saeed e Wladyslaw Homenda. Cham: Springer International Publishing, 2015, pp. 58–69. ISBN: 978-3-319-24369-6.
- [2] Aman Sharma et al. «Bitcoin’s Blockchain Data Analytics: A Graph Theoretic Perspective». In: *Advanced Information Networking and Applications*. A cura di Leonard Barolli, Farookh Hussain e Tomoya Enokido. Cham: Springer International Publishing, 2022, pp. 459–470. ISBN: 978-3-030-99584-3.
- [3] Francesco Zola et al. «Bitcoin and Cybersecurity: Temporal Dissection of Blockchain Data to Unveil Changes in Entity Behavioral Patterns». In: *Applied Sciences* 9.23 (2019). ISSN: 2076-3417. DOI: 10.3390/app9235003. URL: <https://www.mdpi.com/2076-3417/9/23/5003>.
- [4] Damiano Di Francesco Maesa, Andrea Marino e Laura Ricci. «Introducing the Bitcoin Payments Graph». In: *Preprint* ().
- [5] Martin Vejačka. «Basic aspects of cryptocurrencies». In: *Journal of Economy, Business and Financing* 2.2 (2014), pp. 75–83.
- [6] Jens Ducrée. *Satoshi Nakamoto and the Origins of Bitcoin – The Profile of a 1-in-a-Billion Genius*. 2022. arXiv: 2206.10257 [cs.GL].
- [7] Satoshi Nakamoto. *Bitcoin: A peer-to-peer electronic cash system*. 2008. URL: <http://www.bitcoin.org/bitcoin.pdf>.
- [8] Damiano Di Francesco Maesa, Andrea Marino e Laura Ricci. «Uncovering the Bitcoin Blockchain: An Analysis of the Full Users Graph». In: *2016 IEEE International Conference on Data Science and Advanced Analytics, DSAA 2016, Montreal, QC, Canada, October 17-19, 2016*. IEEE, 2016, pp. 537–546. DOI: 10.1109/DSAA.2016.52. URL: <https://doi.org/10.1109/DSAA.2016.52>.

- [9] Sarah Meiklejohn et al. «A fistful of bitcoins: characterizing payments among men with no names». In: *Proceedings of the 2013 Internet Measurement Conference, IMC 2013, Barcelona, Spain, October 23-25, 2013*. A cura di Konstantina Papagiannaki, P. Krishna Gummadi e Craig Partridge. ACM, 2013, pp. 127–140. DOI: 10.1145/2504730.2504747. URL: <https://doi.org/10.1145/2504730.2504747>.
- [10] Shuo Chen e Shaikh Muhammad Uzair Norman. «Social Networks are Divulging Your Identity behind Crypto Addresses». In: *CoRR* abs/2211.09656 (2022). DOI: 10.48550/ARXIV.2211.09656. arXiv: 2211.09656. URL: <https://doi.org/10.48550/arXiv.2211.09656>.
- [11] Joe Mullin. *The incredibly simple story of how the govt Googled Ross Ulbricht — arstechnica.com*. <https://arstechnica.com/tech-policy/2015/01/the-incredibly-simple-story-of-how-the-govt-googled-ross-ulbricht/>. [Accessed 17-03-2024]. Gen. 2015.
- [12] Andreas M Antonopoulos e David A Harding. *Mastering bitcoin*. ” O'Reilly Media, Inc.”, 2023.
- [13] Ralph C. Merkle. «A Digital Signature Based on a Conventional Encryption Function». In: *Advances in Cryptology - CRYPTO '87, A Conference on the Theory and Applications of Cryptographic Techniques, Santa Barbara, California, USA, August 16-20, 1987, Proceedings*. A cura di Carl Pomerance. Vol. 293. Lecture Notes in Computer Science. Springer, 1987, pp. 369–378. DOI: 10.1007/3-540-48184-2\32. URL: https://doi.org/10.1007/3-540-48184-2%5C_32.
- [14] Fredy Andres Aponte-Novoa et al. «The 51% Attack on Blockchains: A Mining Behavior Study». In: *IEEE Access* 9 (2021), pp. 140549–140564. DOI: 10.1109/ACCESS.2021.3119291. URL: <https://doi.org/10.1109/ACCESS.2021.3119291>.
- [15] *Are multi-signature transaction fees really that expensive? — fortris.com*. <https://fortris.com/blog/multi-signature-transaction-fees>. [Accessed 04-04-2024].
- [16] Rachel Rybczyk. *Understanding the Bitcoin Blockchain Header — medium.com*. <https://medium.com/fcats-blockchain-incubator/understanding-the-bitcoin-blockchain-header-a2b0db06b515>. [Accessed 16-03-2024]. Dic. 2020.
- [17] Zhang YaNing. *My Comparison between the UTXO and Account Model — medium.com*. <https://medium.com/nervosnetwork/my-comparison-between-the-utxo-and-account-model-821eb46691b2>. [Accessed 16-03-2024]. Ott. 2018.

- [18] Flora Sun. *UTXO vs Account/Balance Model* — sunflora98. <https://medium.com/@sunflora98/utxo-vs-account-balance-model-5e6470f4e0cf>. [Accessed 16-03-2024]. Apr. 2018.
- [19] *UTXO: Analyzing Outputs for Efficient Bitcoin Transactions - Faster-Capital* — fastercapital.com. <https://fastercapital.com/content/UTXO--Analyzing-Outputs-for-Efficient-Bitcoin-Transactions.html#Benefits-of-UTXO-Model.html>. [Accessed 16-03-2024]. Feb. 2024.
- [20] *The UTXO vs Account Model — Horizen Academy* — horizen.io. <https://www.horizen.io/academy/utxo-vs-account-model/>. [Accessed 18-03-2024].
- [21] W. Diffie e M. Hellman. «New directions in cryptography». In: *IEEE Transactions on Information Theory* 22.6 (1976), pp. 644–654. DOI: 10.1109/TIT.1976.1055638.
- [22] *What is a Bitcoin wallet? — Coinbase Help* — help.coinbase.com. <https://help.coinbase.com/en/coinbase/getting-started/crypto-education/what-is-a-bitcoin-wallet>. [Accessed 17-03-2024].
- [23] *What is a Bitcoin wallet? — Get Started with Bitcoin.com* — bitcoin.com. <https://www.bitcoin.com/get-started/what-is-a-bitcoin-wallet/>. [Accessed 17-03-2024].
- [24] Sabine Houy, Philipp Schmid e Alexandre Bartel. «Security Aspects of Cryptocurrency Wallets - A Systematic Literature Review». In: *ACM Comput. Surv.* 56.1 (2024), 4:1–4:31. DOI: 10.1145/3596906. URL: <https://doi.org/10.1145/3596906>.
- [25] *Guida agli Hardware Wallet — Il blog di Binance* — binance.com. <https://www.binance.com/it/blog/community/guida-agli-hardware-wallet-5835526156524040496>. [Accessed 17-03-2024].
- [26] Patrick McCorry, Malte Möser e Syed Taha Ali. «Why Preventing a Cryptocurrency Exchange Heist Isn't Good Enough». In: *Security Protocols XXVI*. A cura di Vashek Matyáš et al. Cham: Springer International Publishing, 2018, pp. 225–233. ISBN: 978-3-030-03251-7.
- [27] Andrew Norry. *The History of the Mt Gox Hack: Bitcoin's Biggest Heist*. <https://blockonomi.com/mt-gox-hack/>. [Accessed 18-03-2024]. Nov. 2023.

- [28] John “Jack” Castonguay e Sean Stein Smith. «Digital Assets and Blockchain: Hackable, Fraudulent, or Just Misunderstood?*». In: *Accounting Perspectives* 19.4 (2020), pp. 363–387. DOI: <https://doi.org/10.1111/1911-3838.12242>. eprint: <https://onlinelibrary.wiley.com/doi/pdf/10.1111/1911-3838.12242>. URL: <https://onlinelibrary.wiley.com/doi/abs/10.1111/1911-3838.12242>.
- [29] Shange Fu et al. «FTX Collapse: A Ponzi Story». In: *CoRR* abs/2212.09436 (2022). DOI: 10.48550/ARXIV.2212.09436. arXiv: 2212.09436. URL: <https://doi.org/10.48550/arXiv.2212.09436>.
- [30] Andreas Antonopoulos. *Bitcoin Q&A: How Do I Secure My Bitcoin?* — youtube.com. <https://www.youtube.com/watch?v=vt-zXEsJ61U&t=0s>. [Accessed 18-03-2024]. Lug. 2017.
- [31] Saurabh Suratkar, Mahesh Shirole e Sunil Bhirud. «Cryptocurrency Wallet: A Review». In: *2020 4th International Conference on Computer, Communication and Signal Processing (ICCCSP)*. 2020, pp. 1–7. DOI: 10.1109/ICCCSP49186.2020.9315193.
- [32] Matthew J. B. Robshaw. «One-Way Function». In: *Encyclopedia of Cryptography and Security*. A cura di Henk C. A. van Tilborg. Springer, 2005. DOI: 10.1007/0-387-23483-7_287. URL: https://doi.org/10.1007/0-387-23483-7%5C_287.
- [33] Quynh H. Dang. *Secure Hash Standard*. 2015. DOI: 10.6028/nist.fips.180-4. URL: <http://dx.doi.org/10.6028/NIST.FIPS.180-4>.
- [34] Shi Yan. «Analysis on Blockchain Consensus Mechanism Based on Proof of Work and Proof of Stake». In: *2022 International Conference on Data Analytics, Computing and Artificial Intelligence (ICDACAII)*. 2022, pp. 464–467. DOI: 10.1109/ICDACAII57211.2022.00098.
- [35] Kirsty Moreland. *What is Proof-of-Work (PoW)? — Ledger — ledger.com*. <https://www.ledger.com/academy/blockchain/what-is-proof-of-work>. [Accessed 18-03-2024]. Ott. 2019.
- [36] Sothearith Seang e Dominique Torre. *Proof of Work and Proof of Stake Consensus Protocols: A Blockchain Application for Local Complementary Currencies*. GREDEG Working Papers 2019-24. Groupe de REcherche en Droit, Economie, Gestion (GREDEG CNRS), Université Côte d’Azur, France, set. 2019. URL: <https://ideas.repec.org/p/gre/wpaper/2019-24.html>.
- [37] Gayan Samarakoon. *Bitcoin Fundamentals* — samarakoon-gayan.medium.com. <https://samarakoon-gayan.medium.com/bitcoin-fundamentals-a5d62fe98bac>. [Accessed 17-03-2024]. Dic. 2018.

- [38] Donald Ervin Knuth. *The art of computer programming, , Volume III, 2nd Edition.* Addison-Wesley, 1998. ISBN: 0201896850. URL: <https://www.worldcat.org/oclc/312994415>.
- [39] J RODRIGUEZ, D ESCUDERO e M TORO. «Data Structure for efficient indexing of files and directories». In: *Research Gate* (2018).
- [40] Cathrin Weiss e Abraham Bernstein. «On-disk storage techniques for Semantic Web data-Are B-Trees always the optimal solution ?» In: 2009. URL: <https://api.semanticscholar.org/CorpusID:15019067>.
- [41] Theo Härdter e Andreas Reuter. «Principles of Transaction-Oriented Database Recovery». In: *ACM Comput. Surv.* 15.4 (1983), pp. 287–317. DOI: [10.1145/289.291](https://doi.org/10.1145/289.291). URL: <https://doi.org/10.1145/289.291>.
- [42] Bryce Merkl Sasaki. *Native vs. Non-Native Graph Database.* Giu. 2023. URL: <https://neo4j.com/blog/native-vs-non-native-graph-technology/>.
- [43] Ian Robinson, Jim Webber e Emil Eifrem. *Graph databases: new opportunities for connected data.* ” O'Reilly Media, Inc.”, 2015.
- [44] E. F. Codd. «A Relational Model of Data for Large Shared Data Banks». In: *Commun. ACM* 13.6 (giu. 1970), pp. 377–387. ISSN: 0001-0782. DOI: [10.1145/362384.362685](https://doi.org/10.1145/362384.362685). URL: <https://doi.org/10.1145/362384.362685>.
- [45] Wikipedia. *Relational database — Wikipedia, The Free Encyclopedia.* <http://en.wikipedia.org/w/index.php?title=Relational%20database&oldid=1190039934>. [Online; accessed 02-January-2024]. 2024.
- [46] URL: https://go.neo4j.com/rs/710-RRC-335/images/Neo4j_Top5-UseCases_Graph%20Databases.pdf.
- [47] *Neo4j in Production: A look at Neo4j in the Real World.* Mag. 2017. URL: <https://www.slideshare.net/neo4j/neo4j-in-production-a-look-at-neo4j-in-the-real-world>.
- [48] Marko A. Rodriguez e Peter Neubauer. «Constructions from Dots and Lines». In: *CoRR* abs/1006.2361 (2010). arXiv: [1006.2361](https://arxiv.org/abs/1006.2361). URL: [http://arxiv.org/abs/1006.2361](https://arxiv.org/abs/1006.2361).

- [49] Chandan Sharma e Roopak Sinha. «A Schema-First Formalism for Labeled Property Graph Databases: Enabling Structured Data Loading and Analytics». In: *Proceedings of the 6th IEEE/ACM International Conference on Big Data Computing, Applications and Technologies, BDCAT 2019, Auckland, New Zealand, December 2-5, 2019*. A cura di Kenneth Johnson et al. ACM, 2019, pp. 71–80. DOI: 10.1145/3365109.3368782. URL: <https://doi.org/10.1145/3365109.3368782>.
- [50] URL: <https://neo4j.com/docs/getting-started/cypher-intro/>.
- [51] *Awesome Procedures On Cypher (APOC) - Neo4j Labs — neo4j.com*. <https://neo4j.com/labs/apoc/>. [Accessed 24-03-2024].
- [52] Peter E. Hart, Nils J. Nilsson e Bertram Raphael. «A Formal Basis for the Heuristic Determination of Minimum Cost Paths». In: *IEEE Trans. Syst. Sci. Cybern.* 4.2 (1968), pp. 100–107. DOI: 10.1109/TSSC.1968.300136. URL: <https://doi.org/10.1109/TSSC.1968.300136>.
- [53] Edsger W. Dijkstra. «A note on two problems in connexion with graphs». In: *Numerische Mathematik* 1 (1959), pp. 269–271. DOI: 10.1007/BF01386390. URL: <https://doi.org/10.1007/BF01386390>.
- [54] José Rocha. *Understanding neo4j's data on disk - knowledge base*. URL: <https://neo4j.com/developer/kb/understanding-data-on-disk/>.
- [55] Stelios Gerogiannakis. Lug. 2015. URL: <https://sgerogia.github.io/Disk-Capacity-Planning-for-Neo4J/>.
- [56] Wikipedia contributors. *Multigraph — Wikipedia, The Free Encyclopedia*. [Online; accessed 10-November-2023]. 2023. URL: <https://en.wikipedia.org/w/index.php?title=Multigraph&oldid=1158564961>.
- [57] Tian Min et al. «Blockchain Games: A Survey». In: *IEEE Conference on Games, CoG 2019, London, United Kingdom, August 20-23, 2019*. IEEE, 2019, pp. 1–8. DOI: 10.1109/CIG.2019.8848111. URL: <https://doi.org/10.1109/CIG.2019.8848111>.
- [58] Alice Huang. «Reaching Within Silk Road: The Need for a New Subpoena Power That Targets Illegal Bitcoin Transactions». In: *Boston College Law Review* 56 (2015), p. 2093. URL: <https://api.semanticscholar.org/CorpusID:14431191>.
- [59] Mt. Gox - Wikipedia — en.wikipedia.org. https://en.wikipedia.org/wiki/Mt._Gox. [Accessed 04-04-2024].

- [60] Tin Tironakkul et al. «Tracking Mixed Bitcoins». In: *Data Privacy Management, Cryptocurrencies and Blockchain Technology - ESORICS 2020 International Workshops, DPM 2020 and CBT 2020, Guildford, UK, September 17-18, 2020, Revised Selected Papers*. A cura di Joaquín García-Alfaro, Guillermo Navarro-Arribas e Jordi Herrera-Joancomartí. Vol. 12484. Lecture Notes in Computer Science. Springer, 2020, pp. 447–457. DOI: [10.1007/978-3-030-66172-4_29](https://doi.org/10.1007/978-3-030-66172-4_29). URL: https://doi.org/10.1007/978-3-030-66172-4%5C_29.
- [61] *The Neo4j Graph Data Science Library Manual v2.6 - Neo4j Graph Data Science* — neo4j.com. <https://neo4j.com/docs/graph-data-science/current/>. [Accessed 24-03-2024].
- [62] Lawrence Page et al. *The PageRank Citation Ranking: Bringing Order to the Web*. Rapp. tecn. Stanford Digital Library Technologies Project, 1998. URL: <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.31.1768>.
- [63] Junwoo Seo et al. «Money Laundering in the Bitcoin Network: Perspective of Mixing Services». In: *International Conference on Information and Communication Technology Convergence, ICTC 2018, Jeju Island, Korea (South), October 17-19, 2018*. IEEE, 2018, pp. 1403–1405. DOI: [10.1109/ICTC.2018.8539548](https://doi.org/10.1109/ICTC.2018.8539548). URL: <https://doi.org/10.1109/ICTC.2018.8539548>.
- [64] Amita Majumder, Megnath Routh e Dipayan Singha. «A Conceptual Study on the Emergence of Cryptocurrency Economy and Its Nexus with Terrorism Financing». In: *The Impact of Global Terrorism on Economic and Political Development* (2019). URL: <https://api.semanticscholar.org/CorpusID:159088600>.
- [65] Shacheng Wang e Xixi Zhu. «Evaluation of potential cryptocurrency development ability in terrorist financing». In: *Policing: A Journal of Policy and Practice* 15.4 (2021), pp. 2329–2340.
- [66] M. Satheesh Kumar, Jalel Ben-Othman e K. G. Srinivasagan. «An Investigation on WannaCry Ransomware and its Detection». In: *2018 IEEE Symposium on Computers and Communications, ISCC 2018, Natal, Brazil, June 25-28, 2018*. IEEE, 2018, pp. 1–6. DOI: [10.1109/ISCC.2018.8538354](https://doi.org/10.1109/ISCC.2018.8538354). URL: <https://doi.org/10.1109/ISCC.2018.8538354>.
- [67] Kevin Liao et al. «Behind closed doors: measurement and analysis of CryptoLocker ransoms in Bitcoin». In: *2016 APWG Symposium on Electronic Crime Research, eCrime 2016, Toronto, ON, Canada, June 1-3, 2016*. IEEE, 2016, pp. 1–13. DOI: [10.1109/ECRIME.2016.7487938](https://doi.org/10.1109/ECRIME.2016.7487938). URL: <https://doi.org/10.1109/ECRIME.2016.7487938>.

Glossario

ACID Atomicity, Consistency, Isolation, Durability. 39

APOC Awesome Procedures On Cypher. 56

AQL ArangoDB Query Language. 14

AWS Amazon Web Services. 15

CEX Centralized Exchange. 30

DBMS Database Management System. 41

DLT Distributed Ledger Technology. 23

GDBMS Graph Database Management Systems. 14, 39

GDS Graph Data Science. 158

IAM Identity and Access Management. 43

LPG Labeled Property Graph. 45

NoSQL Not Only SQL. 39

OWF One-Way Function. 32

OWL Web Ontology Language. 16

PoW Proof-Of-Work. 21, 32

RDBMS Relational Database Management System. 41

RDF Resource Description Framework. 46

SPARQL SPARQL Protocol and RDF Query Language. 48

SQL Structured Query Language. 48

URI Uniform Resource Identifier. 46

UTXO Unspent Transaction Output. 25

W3C World Wide Web Consortium. 16

Liste dei codici

1	Query SQL per trovare il nome dei corsi superati con un voto superiore a 29 da studenti che hanno sostenuto un corso tenuto da un professore specifico.	44
2	Query Cypher per trovare il nome dei corsi superati con un voto superiore a 29 da studenti che hanno sostenuto un corso tenuto da un professore specifico.	45
3	Esempio di nodi in Cypher.	49
4	Esempio di relazioni in Cypher.	50
5	Esempio di proprietà in Cypher.	50
6	Specificando un singolo nodo senza etichetta verranno restituiti tutti i nodi del grafo.	51
7	Specificando un singolo nodo con etichetta verranno restituiti tutti i nodi del grafo con quella etichetta.	51
8	Restituzione delle proprietà di un nodo e delle relazioni in uscita di tipo HA_SOSTENUTO.	51
9	Restituzione di tutti gli studenti che hanno avuto nella propria carriera accademica solo voti maggiori di 25.	52
10	Filtraggio dei nodi per etichetta e proprietà.	52
11	Filtraggio delle relazioni per proprietà.	53
12	Creazione di nodi con etichetta e proprietà.	53
13	Creazione di nodi e relazioni con tipo e proprietà.	53
14	Creazione di una relazione tra nodi esistenti.	54
15	Aggiornamento delle proprietà di una relazione.	54
16	Fare MERGE di un nodo con proprietà che non sono presenti in nessun nodo esistente nel grafo porterà alla creazione di un nuovo nodo.	54
17	Fare MERGE di un nodo con proprietà uguali a quelle di un nodo esistente nel grafo non porterà alla creazione di un nuovo nodo.	55
18	Invocazione della procedura db.labels() per recuperare tutte le etichette presenti nel grafo.	55

19	Iterazione su una lista di valori per restituire il doppio di ciascun valore.	55
20	Importazione di dati da un file CSV.	56
21	Utilizzo di <code>apoc.periodic.iterate</code> per importare il file CSV <code>studenti.csv</code> descritto precedentemente nella sezione 3.4.5 e creare i corrispondenti nodi <code>Studente</code> nel grafo. <code>batchSize</code> è il numero di righe del file CSV che vengono fornite dalla query esterna (<code>LOAD CSV WITH HEADERS FROM 'file:///studenti.csv' AS row</code>) alla query interna (<code>CREATE (:Studente Matricola: row.Matricola, Nome: row.Nome, Cognome: row.Cognome)</code>). <code>batchMode</code> è impostato a "BATCH" per far processare <code>batchSize</code> righe alla query interna in un'unica transazione.	57
22	Considerando sempre l'esempio in figura 3.5, ipotizzando che sia scaduto il periodo di immatricolazione per studenti universitari e che si voglia eliminare dal database tutti gli studenti che non hanno finalizzato la propria immatricolazione (supponendo che siano identificati dall'etichetta <code>AttesaImmatricolazione</code>), è possibile utilizzare <code>apoc.periodic.commit</code> per raggiungere questo obiettivo: l'esecuzione della query interna verrà ripetuta finché non ci saranno più nodi <code>AttesaImmatricolazione</code> nel database, ovvero <code>count(s)</code> restituirà 0.	58
23	Considerando ancora l'esempio in figura 3.5, se si vuole esportare sul file <code>students.csv</code> le matricole degli studenti che non hanno mai preso ad esami voti inferiori a 25 è possibile utilizzare <code>apoc.export.csv.query</code> per raggiungere questo obiettivo.	58
24	ipotizzando che i nodi <code>Studente</code> in figura 3.5 abbiano una relazione di amicizia tra loro di tipo <code>AMICO</code> e che si voglia espandere le catene di amicizia a partire da un nodo <code>Studente</code> con matrictola <code>5678</code> fino a una profondità fissata <code>maxLevel</code> restituendo i cammini trovati, è possibile utilizzare <code>apoc.path.expandConfig</code> per tale scopo.	59
25	Esempio di definizione di un indice RANGE sulla proprietà <code>Matricola</code> dei nodi con etichetta <code>Studente</code> del grafo in figura 3.5 al fine di velocizzare le query che utilizzano questa proprietà.	61
26	Esempio di definizione di un indice composito sulle due proprietà <code>Nome</code> e <code>Cognome</code> dei nodi con etichetta <code>Studente</code> del grafo in figura 3.5 al fine di velocizzare le query che utilizzano queste proprietà.	61

27	Esempio di definizione di un indice TEXT sulla proprietà di tipo stringa Nome dei nodi con etichetta Studente al fine di velocizzare le query che utilizzano questa proprietà.	62
28	Ipotizzando rispetto al grafo in figura 3.5 che ciascun corso abbia delle relazioni di tipo TENUTO_IN con dei nodi Aula e che ciascun nodo Aula abbia una proprietà di tipo POINT posizione con latitudine e longitudine , si potrebbe velocizzare le query che coinvolgono il calcolo di distanze tra le aule creando un indice di tipo POINT sulla proprietà posizione di ciascun nodo Aula	62
29	Ipotizzando rispetto al grafo in figura 3.5 che ciascuna relazione di tipo HA_SOSTENUTO abbia una proprietà di tipo stringa questionario_valutazione contenente il testo inserito da ciascuno studente nel questionario di valutazione al momento dell’iscrizione a un esame, si potrebbero velocizzare le query che coinvolgono la ricerca di testo all’interno di questa proprietà (e.g “prove in itinere” o “ricevimenti serali per studenti lavoratori”) creando un indice full-text sulla proprietà questionario_valutazione di ciascuna relazione HA_SOSTENUTO	63
30	Interrogazione che utilizza un indice full-text per cercare parole chiave all’interno di una proprietà di tipo stringa.	63
31	Query 1a per l’address-transaction graph.	93
32	Query 1a per il payment graph.	94
33	Query 1a per l’address graph con valori aggregati.	95
34	Query 1b per l’address-transaction graph.	97
35	Query 1b per il payment graph.	97
36	Query 1b per l’address graph con valori aggregati.	98
37	Query 2 per l’address-transaction graph.	100
38	Query 2 per il payment graph.	101
39	Query 2 per l’address graph con valori aggregati.	102