Università di Pisa

Dipartimento di Informatica

Corso di Laurea Triennale in Informatica

Modelli di Grafo per la Blockchain Ethereum in Neo4j

Candidato:

Daniele Zeolla

Relatori:

Prof.ssa Laura Emilia Maria Ricci

Dott. Damiano Di Francesco Maesa

Dott. Matteo Loporchio

Anno Accademico 2023-2024

A coloro che hanno creduto in me.

Indice

Capitolo 1

Introduzione

La tecnologia blockchain rappresenta una delle innovazioni più sconvolgenti e innovative degli ultimi anni in ambito finanziario ed economico. Le blockchain [1] permettono la gestione di dati e transazioni in modo sicuro, trasparente e con processi completamente decentralizzati. Una blockchain essenzialmente è un registro di contabilità distribuito e immutabile, composto da una catena di blocchi collegati tra di loro tramite apposite funzioni crittografiche, che facilita la registrazione delle transazioni e la loro tracciabilità in una rete completamente distribuita. Qualsiasi entità materiale o immateriale che abbia un valore può essere scambiata e rintracciata su una rete blockchain. La caratteristica distintiva delle blockchain è la loro natura decentralizzata: non sono controllate in nessun modo da singole autorità centrali, ma sono completamente gestite da reti peer-to-peer con nodi che collaborano tra di loro per aggiungere e verificare nuove transazioni. Questo tipo di architettura elimina completamente la necessità di intermediari e rende il sistema resistente a frodi e manipolazioni. Infatti, tutti i nodi partecipanti della rete sono incentivati economicamente a seguire le regole del sistema piuttosto che comportasi in modo scorretto. Le caratteristiche chiave di ogni blockchain sono quindi: decentralizzazione, immutabilità e meccanismo di consenso.

Ad oggi, la blockchain più largamente conosciuta ed utilizzata è sicuramente quella di **Bitcoin** [1]. Ha avuto un enorme successo fin dalla sua creazione nel 2009, introducendo per la prima volta il concetto di criptovaluta decentralizzata e aprendo la strada ad un'intera industria basata sulla tecnologia blockchain. La sua popolarità ha segnato l'inizio di una rivoluzione finanziaria e tecnologica dando una spinta sostanziale a migliaia di altre criptovalute e aumentando l'interesse mondiale per la tecnologia. Nonostante la sua popolarità, innumerevoli altre blockchain sono nate per far fronte alle limitazioni di Bitcoin, prima tra tutte la sua programmabilità limitata.

Bitcoin nasce come semplice valuta digitale e non offre nessun altro tipo di servizio se non lo scambio della valuta tra utenti della rete. Questi limiti hanno portato allo sviluppo di blockchain sempre più evolute tra cui **Ethereum** [2]. Ethereum è stata progettata per poter essere il più flessibile e programmabile possibile, consentendo agli sviluppatori di creare ed eseguire su blockchain speciali programmi chiamati *smart contract* necessari per la realizzazione di applicazioni decentralizzate (DApp). La capacità di poter eseguire codice arbitrario sulla blockchain permette lo sviluppo di qualsiasi applicazione decentralizzata, inclusi giochi, sistemi di voto, mercati finanziari, nuove criptovalute e molto altro. Ethereum è stata la prima ad aver introdotto il concetto di blockchain di livello superiore, general-porpouse e adatta ad applicazioni che vanno al di là delle semplici transazioni finanziarie.

La nascita e l'utilizzo di blockchain più evolute ha reso sempre più importante la loro analisi al fine di individuare e studiare fenomeni e comportamenti sociali della rete. Questa pratica mira a comprendere e interpretare i dati immagazzinati sulla blockchain al fine di trarre informazioni utili per vari scopi. In primo luogo queste analisi permettono agli sviluppatori e agli utenti finali di comprendere meglio il comportamento degli *smart contract* e il loro impatto sulla rete. È possibile infatti individuare bug, comportamenti scorretti, vulnerabilità o addirittura comprendere l'impatto economico e sociale degli *smart contract* presenti sulla blockchain. Inoltre, l'analisi delle transazioni può essere utilizzata per identificare pagamenti fraudolenti, attacchi perpetrati da malintenzionati o comportamenti sospetti da parte degli utenti.

1.1 Contributo della tesi

L'obiettivo di questa tesi è quello di analizzare e confrontare due modelli di grafo progettati per rappresentare i dati della blockchain Ethereum. I modelli di grafo proposti sono stati creati al fine di semplificare l'analisi dei dati della blockchain e permettere una visualizzazione intuitiva e chiara delle transazioni. Tuttavia, i modelli proposti sono stati progettati per soddisfare esigenze di analisi diverse ma complementari per Ethereum e perciò la scelta del modello dipende dall'obiettivo specifico e delle analisi che si vogliono affrontare. I due modelli rappresentano le stesse informazioni ma con una struttura diversa: il modello **TN** (transactions as nodes) rappresenta le transazioni come nodi (*transaction, block, account, event*), mentre il modello **TE** (transaction as edges) rappresenta le transazioni come archi (con solo nodi *account*). Nel primo caso le informazioni delle transazioni, blocchi ed eventi sono memorizzate direttamente nei nodi, mentre nel secondo modello

le stesse informazioni sono raggruppate all'interno degli archi. I nodi di tipo *Account* sono gli unici nodi comuni dei due modelli.

I modelli sono stati implementati utilizzando **Neo4j** [3], un potente graph database NoSQL che permette di gestire e manipolare grandi quantità di dati su grafi in modo semplice grazie ai tool presenti nella suite software e dispone di un linguaggio di interrogazione avanzato chiamato **Cypher**.

Dopo la prima fase di classificazione e importazione dei dati all'interno dei database, abbiamo avviato la fase di valutazione sperimentale per mettere in evidenza le differenze prestazionali tra i database basati sui modelli proposti. Nella prima fase di analisi abbiamo effettuato un confronto tra i due modelli mediante una serie di query generiche. I risultati hanno dimostrato che nessuno dei due modelli è nettamente superiore all'altro. A seconda del caso d'uso e dei requisiti di tempo/spazio conviene utilizzare un modello piuttosto che un altro. Infine abbiamo approfondito le analisi utilizzando query realistiche su entrambi i modelli, concentrandoci sul fenomeno dell'attacco al contratto della prima organizzazione autonoma decentralizzata di Ethereum, conosciuta come *"The DAO"*. Questo ci ha permesso di dimostrare l'utilità pratica dei due modelli su scenari reali.

1.2 Struttura della tesi

La presente tesi è strutturata in modo da fornire una chiara e completa visione del lavoro svolto. Il Capitolo 2 introduce la tecnologia blockchain, con particolare attenzione al funzionamento di Ethereum, ne espone le potenzialità e ripercorre brevemente la storia della loro evoluzione. Il Capitolo 3 spiega il funzionamento del graph database Neo4j, introduce brevemente il query language Cypher e presenta tutti gli strumenti utilizzati al fine di ottimizzare importazione e gestione di grandi quantità di dati. A partire dal Capitolo 4 vengono introdotti i modelli di grafo proposti ed esposte le loro strutture, con un primo confronto teorico.

Nel Capitolo 5 si introduce il dataset usato e tutti gli script e tecniche utilizzate che hanno permesso di effettuare le analisi dei due modelli. I risultati delle analisi svolte vengono esposti nel Capitolo 6, nel quale sono presenti tutte le osservazioni fatte e le conclusioni tratte sull'analisi dei dati. Il Capitolo 7 introduce un confronto tra vari graph database che ci ha portato alla scelta di Neo4j e infine nel Capitolo 8 vengono presentate le conclusioni e le osservazioni finali.

Capitolo 2

Background

Nel capitolo corrente, spiegheremo i concetti principali della tecnologia blockchain e approfondiremo una delle sue implementazioni più famose: Ethereum. Partiremo con i concetti base della tecnologia blockchain delineando il suo funzionamento e le sue caratteristiche distintive. Dedicheremo particolare attenzione ad Ethereum spiegando dettagliatamente il suo funzionamento interno e le differenze rispetto alle blockchain tradizionali.

2.1 Blockchain

La blockchain è un registro digitale aperto e distribuito composto da elementi collegati tra di loro chiamati *blocchi*. La prima blockchain fu introdotta nel 2008 da **Satoshi Nakamoto** nel famoso articolo *Bitcoin: A Peer-to-Peer Electronic Cash System*[1] e implementata l'anno seguente con l'obiettivo di fungere da "libro mastro" della nuova criptovaluta *Bitcoin* (BTC). Inizialmente, la blockchain fu introdotta per registrare scambi di denaro, detti anche *transazioni*, senza l'ausilio dei tradizionali sistemi centralizzati di pagamento. La blockchain può essere vista come una lista di blocchi in continua crescita collegati tra di loro e resi sicuri mediante l'uso di particolari funzioni hash crittografiche che permettono di ottenere una "firma" univoca del blocco.

Ad ogni blocco possono essere associate più transazioni ed inoltre, ai fini della realizzazione della catena, ogni blocco contiene una marca temporale e un puntatore hash al blocco precedente (da qui il nome blockchain, letteralmente "catena di blocchi"). Per questo motivo, le transazioni inserite nella blockchain non possono essere modificate né cancellate senza alterare tutti i blocchi successivi.

La natura decentralizzata del sistema rende il processo di validazione di nuovi blocchi molto robusto e sicuro, a scapito di tempi di aggiornamento

della rete non trascurabili. Per ottenere il consenso da parte di tutti i nodi della rete si utilizzano speciali meccanismi di consenso [4]. Ad oggi quelli più utilizzati sono due e vengono chiamati rispettivamente **proof-of-work** (POW) e **proof-of-stake** (POS).

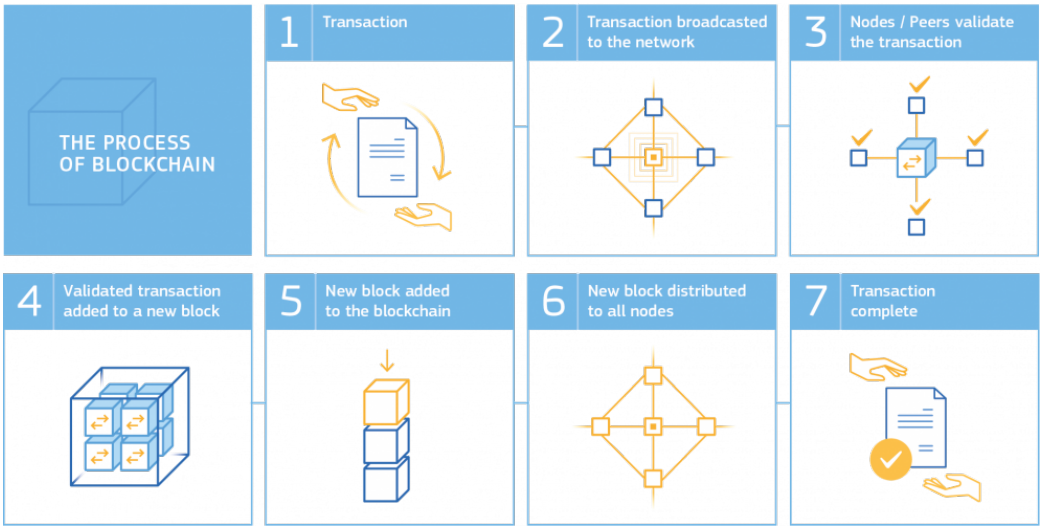

Figura 2.1: Funzionamento della blockchain [5]

La Figura 2.7 illustra le fasi di vita di una transazione inviata sulla blockchain.

Proof of Work (POW)

La *proof-of-work* o protocollo *proof-of-work* è un meccanismo utilizzato per scoraggiare gli attacchi *denial of service*[1] e altri abusi della rete richiedendo un "impegno" in termini di computazione agli utilizzatori della blockchain. Fu inizialmente proposto e successivamente utilizzato per la prima volta sulla blockchain *Bitcoin*.

Il lavoro richiesto dalla rete deve essere moderatamente complesso ma fattibile, in modo da dare la possibilità agli utenti di risolverlo in tempi ragionevoli per poter accedere al servizio. Grazie a questo meccanismo, un eventuale attaccante dovrà essere in possesso di una grande potenza di calcolo per portare a termine un attacco.

[1]Un attacco denial-of-service o attacco DoS si verifica quando un attaccante tenta deliberatamente di esaurire le risorse di un sistema informatico inondandolo di richieste senza attendere risposta. Questo causa il sovraccarico del sistema che non sarà più in grado di soddisfare richieste di client onesti.

Nonostante la sua semplicità, il meccanismo *proof-of-work* ha il grande svantaggio di consumare una grande quantità di energia per il fatto che i lavori richiesti dalla rete richiedono una discreta capacità di potenza di calcolo.

Proof of Stake (POS)

La *Proof-of-stake* è un meccanismo più avanzato della *POW* che permette ai nodi della blockchain di raggiungere un accordo sullo stato della rete in modo più efficiente, sicuro e con meno dispendio di energia. È considerato il successore della *POW* ed è utilizzato su blockchain più moderne come Ethereum. Invece di richiedere di eseguire del lavoro come nel meccanismo del *proof-of-work*, il *proof-of-stake* è un modo per dimostrate che i validatori hanno immesso del valore nella rete che può essere distrutto se si comportano in modo disonesto. Infatti, a un validatore che tenta di imbrogliare o comportarsi in modo disonesto verrà sottratta parte della ricchezza da lui inizialmente depositata. Questo disincentiva enormemente i comportamenti scorretti sulla rete e riduce notevolmente il dispendio di energia necessario per il funzionamento del meccanismo stesso.

2.2 Ethereum

Ethereum è una blockchain nata nel 2015 [2] che segue una serie di regole implementate nel protocollo Ethereum. La rete funge da base per comunità, applicazioni, organizzazioni e risorse digitali che chiunque può creare ed utilizzare [6]. A differenza di *Bitcoin*, Ethereum è una blockchain general-purpose di nuova generazione che permette non solo lo scambio di semplice denaro ma anche l'esecuzione decentralizzata di applicazioni complesse grazie agli *smart contract*.

La criptovaluta in uso su Ethereum è chiamata *Ether* (ETH) e il suo scopo è quello di abilitare un "mercato del calcolo" oltre che a fungere da semplice moneta di scambio. Ogni utente che trasmette una richiesta di transazione deve offrire un certo importo in ETH a titolo di ricompensa che successivamente verrà elargita a chiunque svolga il lavoro effettivo verificando la transazione, eseguendola e trasmettendola alla rete. La quantità di ETH pagato corrisponde alle risorse necessarie ad eseguire il calcolo della transazione. L'*Ether* fornisce sicurezza alla rete nei seguenti modi:

- Usato come ricompensa per i validatori che propongono nuovi blocchi e segnalano comportamenti disonesti;

- "Impegnato" dai validatori a titolo di garanzia contro i comportamenti disonesti;
- Usato per ponderare i "voti" per i nuovi blocchi proposti, aiutando la scelta della diramazione del meccanismo di consenso[2];

Il meccanismo di consenso *proof-of-work*, inizialmente utilizzato su Ethereum, è stato successivamente sostituito con il *proof-of-stake* nel **2022** perché ritenuto più evoluto, sicuro ed efficiente del POW [7].

Su Ethereum, tutte le funzioni utilizzate per calcolare gli hash di transazioni e blocchi fanno parte di una vasta famiglia di funzioni chiamata **Keccak** [8].

2.2.1 EVM

La **EVM** (Ethereum virtual machine) è una macchina virtuale **Turing completa**[3], distribuita gestita da un ingente numero di computer collegati alla rete che eseguono un client Ethereum. Il protocollo Ethereum esiste al solo scopo di mantenere in funzione questa speciale macchina a stati distribuita.

La **EVM** transisce da uno stato all'altro tramite un'apposita **funzione di transizione di stato**:

$$Y(S, T) = S'$$

che, dato uno stato S e un nuovo set di transazioni T, produce un nuovo stato di output S'.

La Figura 2.2 mostra il progredire della blockchain ad ogni nuovo blocco validato e inserito nella catena.

[2] A causa della latenza della rete, è possibile che ci siano versioni diverse della blockchain con blocchi in testa differenti. I client di consenso dispongono di un particolare algoritmo per far fronte a questa problematica.

[3] Nella teoria della calcolabilità un modello si dice Turing completo se può essere usato per simulare una macchina di Turing universale.

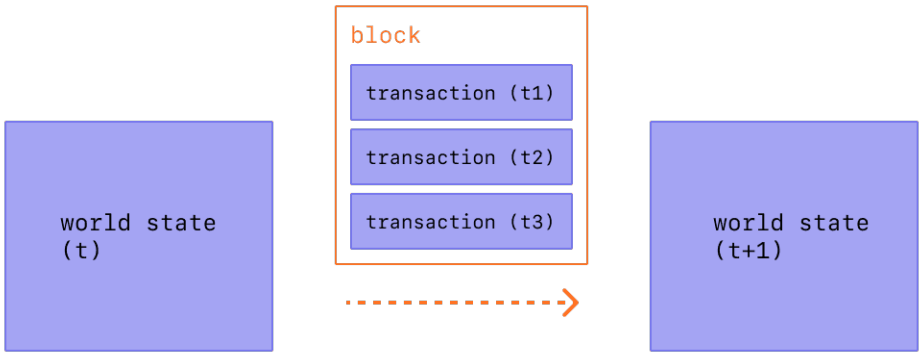

Figura 2.2: Progressione dello stato della rete ad ogni nuovo blocco inserito [9]

2.2.2 Ethereum trie

Lo stato della rete Ethereum (account, saldi e smart contract) è codificato all'interno di una speciale struttura dati chiamata *Modified Merkle Patricia trie* (MPT) [10], una struttura ottenuta dalla combinazione di *Merkle Tree* e *Patricia Trie* con l'aggiunta di alcune ottimizzazioni specifiche per Ethereum. Il *Modified Merkle Patricia Trie* viene utilizzato come struttura dati per il salvataggio degli stati della rete.

Trie

Un *trie* è una struttura dati ad albero utilizzata per archiviare una collezione di dati associati a sequenze di caratteri o chiavi che hanno elementi in comune. In un trie, ogni nodo interno rappresenta un carattere, e ogni percorso dalla radice a una foglia corrisponde a una chiave intera. Questo rende i trie particolarmente efficaci per la ricerca veloce di dati basati su stringhe ed inoltre garantisce un efficiente utilizzo della memoria quante più stringhe hanno prefissi comuni tra di loro. La Figura 2.3 illustra un semplice trie che memorizza parole. I cammini dalla radice ad ogni foglia rappresentano una parola memorizzata nel trie.

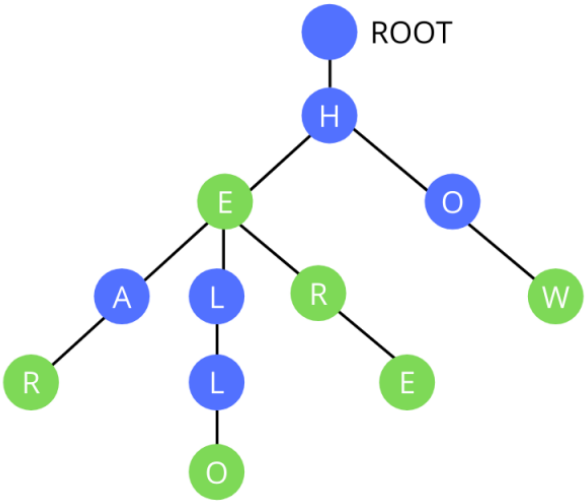

Figura 2.3: Esempio struttura trie [11]

Patricia trie

Un *Patricia trie* o *Prefix tree* è una variante del *trie*, che ottimizza l'utilizzo di memoria rispetto alla struttura tradizionale eliminando i nodi interni con un solo figlio e combinando i nodi con lo stesso prefisso. Questo consente di avere una rappresentazione più compatta (compressa) rispetto ad un *trie* riducendo cosi la memoria necessaria per rappresentare e memorizzare i dati. Come illustrato nella Figura 2.4, alcuni nodi intermedi e le foglie contengono più caratteri. Questa struttura è la più veloce per la ricerca di prefissi comuni e facile da implementare.

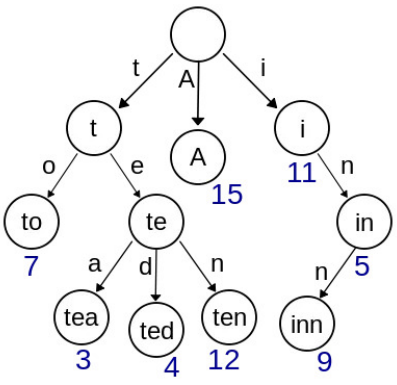

Figura 2.4: Esempio Patricia trie [12]

Merkle Tree

In un *Merkle Tree* o *Hash tree*, ogni foglia è etichettata con un hash ottenuto a partire da un blocco di dati, e ogni nodo che non è una foglia (nodi intermedi) è a sua volta un hash delle etichette dei suoi nodi figli (vedi Figura 2.5). Questo tipo di struttura consente di verificare in modo sicuro ed efficiente il contenuto di grandi strutture dati. La radice dell'albero può essere infatti utilizzata per verificare le informazioni della struttura dati a cui si riferisce.

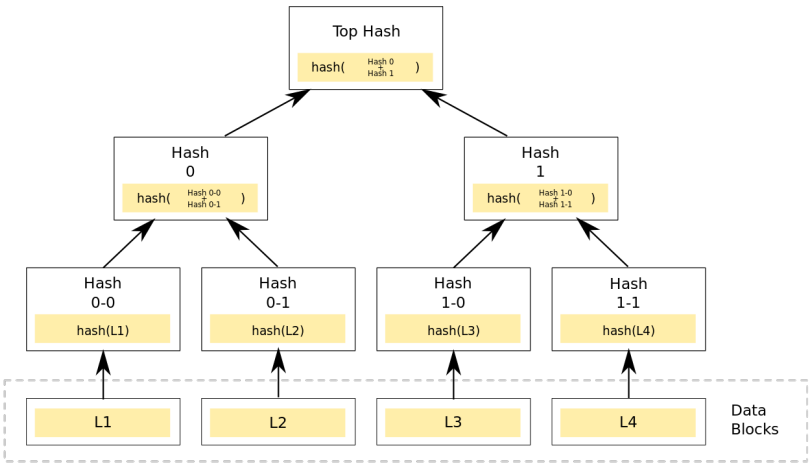

Figura 2.5: Merkle tree [13]

Merkle Patricia Trie

Come accennato ad inizio sezione, i *Merkle Patricia trie* sono la struttura dati utilizzata su Ethereum per la memorizzazione degli stati. La Figura 2.6 mostra la struttura interna di un *Merkle Patricia Trie*.

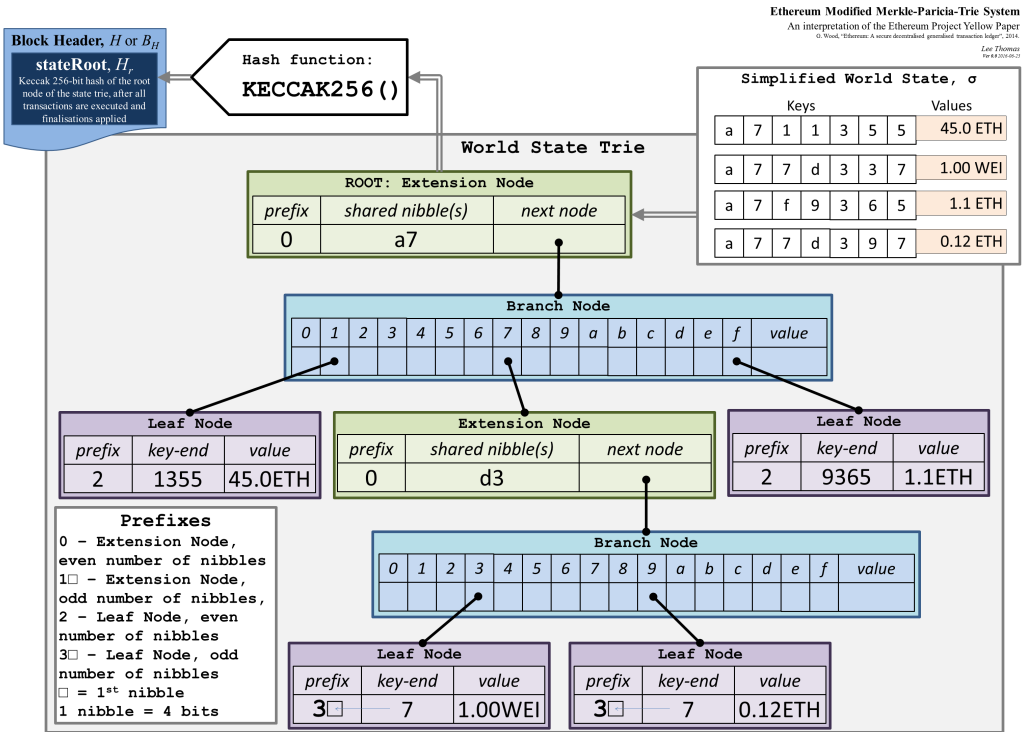

Figura 2.6: Merkle Patricia Trie [14]

Tutte le informazioni di Ethereum sono contenute all'interno di 4 trie.

- **State trie** - Unico trie con scopo globale, mantiene le informazioni degli account;
- **Storage trie** - Uno per account, contiene tutti i dati degli *smart contract*;
- **Transaction trie** - Uno per blocco, contiene tutte le transazioni del blocco;
- **Receipts trie** - Uno per blocco, contiene tutte le ricevute delle transazioni[4] del blocco;

Lo scopo del *Transaction trie* è quello di garantire l'integrità di una transazione e verificare se essa sia stata effettivamente validata dalla rete. Infatti, data la radice di un *Merkle Patricia trie* costruito a partire dagli hash di un

[4]Una ricevuta continente l'esito di una transazione come lo stato post-transazione, il gas utilizzato, eventuali log emessi e uno speciale filtro bloom utile per scopi di information retrival.

insieme di transazioni è possibile verificare con apposite tecniche se, preso uno specifico hash, la corrispondente transazione fa parte del trie o meno. Come vedremo in seguito, le radici di vari *Merkle Patricia Trie* necessari al funzionamento di Ethereum sono memorizzati all'interno degli header dei blocchi (vedi Sezione 2.2.4). Il *receipts trie* ha la stessa funzionalità ma per le ricevute delle transazioni.

2.2.3 Account

Un account Ethereum è un'entità a cui è associato un saldo in ether (ETH) che può inviare transazioni sulla rete. Gli account sono di proprietà degli utenti ma possono essere anche associati a *smart contract*. Ad ogni account è associato un indirizzo esadecimale di 20 bytes[5]. Sulla rete Ethereum possiamo distinguere due tipi di account:

- Externally owned account (*EOA*);
- Account associato ad uno *smart contract*;

Questa distinzione sarà fondamentale per gli scopi di questa tesi in quanto ci consentirà, con tecniche che vedremo nei capitoli successivi, la classificazione delle transazioni Ethereum.

Externally owned account (*EOA*)

Gli *EOA* sono account che permettono agli utenti della rete di inviare transazioni verso qualsiasi altro account esistente su Ethereum. Le transazioni tra *EOA* riguardano unicamente trasferimenti di ether e non causano l'esecuzione del codice di nessun contratto. Gli *EOA* sono composti da una coppia di chiavi crittografiche (pubblica/privata) che consentono di controllare le attività dell'account. Chi possiede la chiave privata di un account ha il pieno controllo su di esso.

L'indirizzo di uno *EOA* è ottenuto considerando gli ultimi bytes del risultato ottenuto della funzione hash **Keccak-256**[6] applicata alla chiave pubblica dell'account. La creazione di un nuovo *EOA* sulla blockchain non ha nessun costo.

[5] 40 caratteri escluso il prefisso 0x, esempio: 0x22466f6c6c6f7720796f757220647265616d7322.
[6] Tipo di famiglia primitiva crittografica.

Account associati a smart contract

L'account di uno *smart contract* è controllato unicamente dal codice del contratto a cui appartiene e non può essere utilizzato per avviare transazioni autonomamente. Un trasferimento di ether (ETH) verso l'indirizzo di uno *smart contract* può causare l'esecuzione del codice del contratto che può eseguire azioni differenti come creare un nuovo contratto, trasferire token o altro ancora.

La creazione di un account associato ad uno *smart contract* è intrinseca nella creazione del contratto stesso. Si rimanda il lettore alla Sezione 2.3.3 per ulteriori dettagli.

2.2.4 Blocchi

I blocchi sono gli elementi fondamentali che compongono la blockchain. Ogni blocco contiene l'hash del blocco precedente e un numero variabile di transazioni (vedi Figura 2.7). Questa configurazione è progettata per evitare le frodi, poiché qualsiasi modifica apportata a un blocco comporterebbe l'invalidazione di tutti i blocchi successivi. Questo avviene perché gli hash dei blocchi verrebbero modificati, rendendo la modifica rilevabile da tutti i client che eseguono la blockchain.

Inoltre, la possibilità di includere più transazioni in un solo blocco velocizza il processo di validazione e permette alla rete di gestire in tempi relativamente brevi una grande quantità di transazioni in modo efficace.

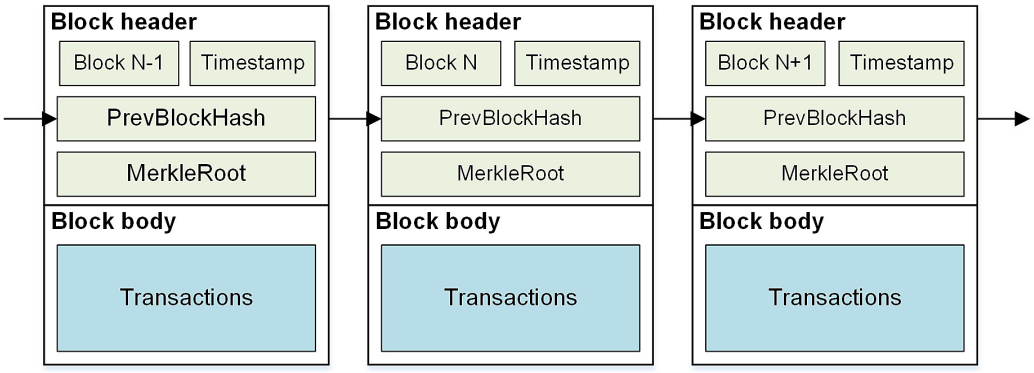

Figura 2.7: Catena di blocchi [15]

I blocchi che compongono la blockchain di Ethereum sono composti da due parti: uno *header* che include metadati relativi al blocco e un *contenuto* che consiste in una lista di transazioni incluse nel blocco stesso. Per garantire l'integrità delle transazioni si costruisce un *Merkle tree* a partire dai loro

hash il cui nodo radice viene memorizzato nello header del blocco. In questo modo, un'eventuale modifica ad una transazione del blocco comporterebbe una modifica del suo header e di conseguenza anche di tutti gli header dei blocchi successivi. Ogni nuovo blocco validato e inserito nella blockchain fa progredire lo stato globale di Ethereum come mostrato in Figura 2.2.

Le più importanti informazioni contenute all'interno dello header di ogni blocco sono:

- **hash** - 32 bytes, hash univoco del blocco;
- **number** - numero del blocco;
- **parent_hash** - hash del blocco precedente;
- **timestamp** - data in formato unix[7] corrispondente al momento della validazione del blocco
- **transactions_roo**t - radice *Merkle tree* transazioni;
- **receipts_root** - radice *Merkle tree* delle ricevute delle transazioni;
- **extra_data** - dati arbitrari aggiuntivi del blocco (raw bytes);
- **miner** - 20 bytes, indirizzo beneficiario ricompensa per blocco;

La Figura 2.8 mostra in modo schematico le strutture dati e le informazioni contentute all'interno di ogni blocco, *header* e *body* inclusi.

[7]Formato utilizzato nei sistemi unix-like, corrisponde al numero di secondi passati dal 1 Gennaio 1970.

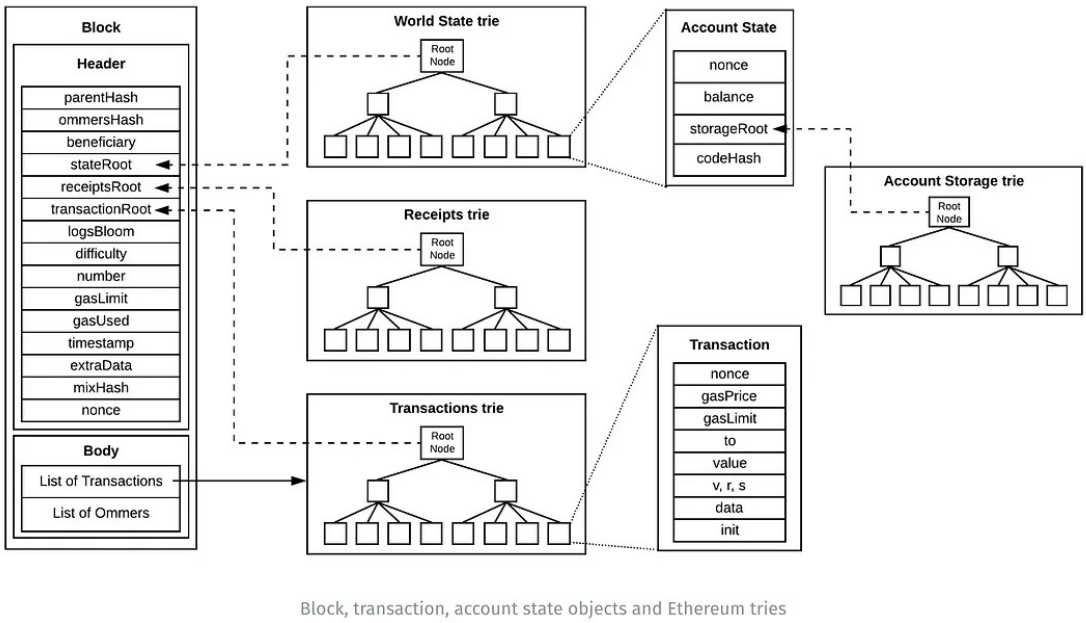

Figura 2.8: Struttura blocco Ethereum [16]

2.2.5 Gas

Come visto nella Sezione 2.2.1, Ethereum basa tutto il suo funzionamento sulla EVM. Il *gas* misura la quantità di *lavoro* necessario per eseguire una transazione sulla EVM. Poiché ogni transazione utilizza delle risorse per poter essere portata a termine, gli utenti devono pagare una commissione atta ad evitare che la rete stessa sia soggetta ad attacchi di denial of service o che non finisca in un ciclo infinito a causa di un errore di programmazione di un qualsiasi contratto.

Ogni utente, prima di inviare una transazione, deve specificare la quantità di gas massima che è disposto a utilizzare con la relativa *base_fee* ed eventuale *priority_fee*. La Figura 2.9 mostra schematicamente le operazioni che "consumano" gas.

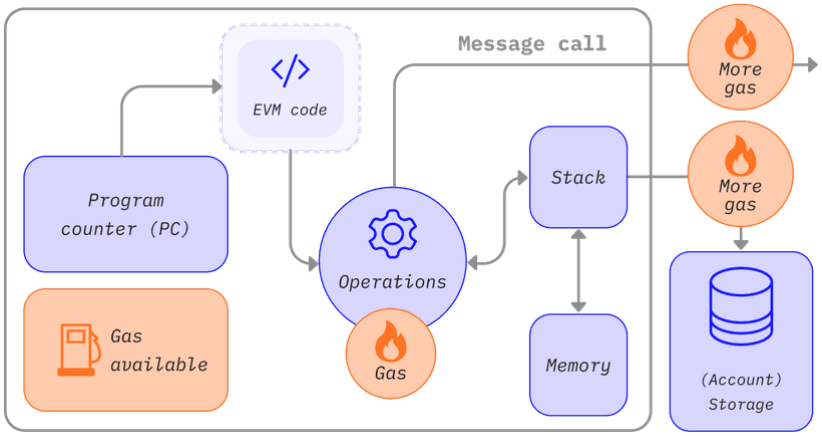

Figura 2.9: Consumo di gas per transazione [17]

Commissioni

La quantità di *gas* necessaria per una transazione è variabile, ma nel caso di un semplice trasferimento di ether da un account ad un altro, la quantità minima necessaria è pari a 21.000 unità, mentre tutte le altre operazioni richiedono una quantità maggiore o uguale.

Nonostante questo, l'effettivo *gas* utilizzato da una transazione può differire da quello specificato dall'utente e, nel caso quello consumato sia inferiore, verrà rimborsata al mittente la parte non consumata altrimenti la transazione non verrà completata e verrà restituito l'errore **OUT OF GAS**. In questo caso, la commissione dovrà essere comunque pagata indipendentemente dal successo o dal fallimento della transazione.

Tipicamente, un utente che paga solo una *base_fee* avrà una probabilità molto bassa che la sua transazione venga inclusa in un blocco (incentivo molto basso per i validatori), mentre un utente che include anche una *priority_fee* per la sua transazione avrà delle possibilità maggiori. Come illustrato nell'equazione seguente, la commissione totale pagata da un utente è pari alla quantità di *gas* effettivamente utilizzato moltiplicato il costo per unità di *gas*.

$$tot\_commission = gas\_used * (base\_fee + priority\_fee)$$

2.2.6 Transazioni

Le transazioni sono il cuore di Ethereum e consentono agli utenti di modificare lo stato della rete (come mostrato in Figura 2.2). La transazione più

semplice che può essere effettuata sulla rete Ethereum è un trasferimento di ETH da un account all'altro.

Uno degli aspetti più importanti da tenere in considerazione è che le transazioni possono essere avviate solamente da account (*EOA*) e mai da *smart contract*.

Ogni transazione richiede una commissione per poter essere inclusa all'interno di un nuovo blocco. Come illustrato nel Codice 1, i campi più importanti in una transazione sono elencati di seguito.

- **from** - indirizzo del mittente;
- **recipient** - indirizzo del destinatario;
- **nonce** - indica il numero della transazione emmessa dall'account del mittente;
- **value** - valore in ETH da trasferire;
- **input** - campo per dati arbitrari;
- **gas_limit** - importo massimo di unità di *gas* che può essere consumato dalla transazione;
- **base_fee** - commissione che il mittente è disposto a pagare per ogni unità di *gas*;
- **priority_fee** - commissione aggiuntiva per ogni unità di gas;

```
{
  "from": "0xEA674fdDe714fd979de3EdF0F56AA9716B898ec8",
  "to": "0xac03bb73b6a9e108530aff4df5077c2b3d481e5a",
  "gas_limit": 21000,
  "max_fee_per_gas": 300,
  "max_priority_fee_per_gas": 10,
  "nonce": 0,
  "value": 10000000000,
}
```

Codice 1: Semplice transazione in formato JSON

Le transazioni vengono sempre firmate con la chiave privata dell'account del mittente per assicurarne l'identità.

Tipi di transazioni

Su Ethereum esistono 3 tipi di transazioni che si distinguono a seconda del loro scopo:

- **Regolari** - Una transazione da un account ad un altro;
- **Creazione contratto** - Una transazione senza indirizzo destinatario, causa la creazione di un contratto con codice fornito dal campo *input*;
- **Invocazione contratto** - Una transazione destinata ad un indirizzo di uno *smart contract* causa l'esecuzione di una funzione del contratto;

Transazioni interne

Sulla blockchain Ethereum esiste un'altra categoria di transazioni chiamate *internal transaction*, che non vengono memorizzate esplicitamente sulla blockchain ma sono il risultato dell'esecuzione di uno *smart contract* e possono essere ricostruite solamente eseguendo la transazione che ne ha causato l'invocazione. Più precisamente, le transazioni interne sono transazioni utilizzate per eseguire operazioni aggiuntive o transazioni all'interno di un contratto durante una transazione principale. Non vengono considerate transazioni regolari perché non vengono iniziate direttamente dall'utente ma sono il risultato della logica dello smart contract eseguito.

Destinatario transazione

Il mittente di ogni transazione deve specificare l'indirizzo destinatario al quale inviare la transazione. Non c'è nessun tipo di controllo da parte della rete sulla validità dell'indirizzo inserito il quale potrebbe essere associato ad un account non esistente. In tal caso, la transazione sarà comunque ritenuta valida dalla rete ma gli ether inviati verranno definitivamente persi.

Sulla rete Ethereum esistono alcuni indirizzi appositamente utilizzati per "bruciare" ether: l'indirizzo nullo e l'indirizzo 0x000...d3ad.

2.3 Smart Contract

Uno **smart contract** o **contratto intelligente** è un programma informatico residente sulla blockchain che, una volta avviato, viene eseguito dalla rete seguendo le logiche definite al suo interno.

Gli **smart contract** sono elementi fondamentali che costituiscono il livello applicazione della rete Ethereum. La loro esecuzione viene scatenata da

una transazione avviata da un utente e diretta verso un indirizzo associato ad uno *smart contract*. È sempre garantito che vengano eseguiti secondo le regole definite nel loro codice, che non può essere modificato una volta creato.

Il termine **smart contract** è stato coniato nel 1994 da Nick Szabo [18] che successivamente definì le loro possibili applicazioni [19]. Un esempio di contratto intelligente potrebbe essere quello di un fornitore di risorse digitali: assegna la proprietà di una risorsa ad un certo utente se egli invia una certa quantità di ether al contratto.

Implementazione

Gli *smart contract* su Ethereum vengono tipicamente creati e implementati utilizzando il linguaggio di programmazione Solidity[8] e successivamente compilati in bytecode[9]. Una volta pubblicati sulla rete il bytecode verrà eseguito dalla blockchain in modo decentralizzato ad ogni invocazione ricevuta. Pubblicare uno *smart contract* sulla rete ha un costo, che dipende dallo spazio occupato dal codice.

2.3.1 Contratti tradizionali

Gli **smart contract** nascono per far fronte a uno dei più grandi problemi dei contratti tradizionali, ovvero la necessità di persone fidate che portino a termine il contratto. Un esempio molto banale per spiegare questo fenomeno è il seguente:

Alice e Bob fanno una scommessa su un qualsiasi gioco, e Alice punta 10 euro sulla sua vittoria. Bob sicuro di vincere accetta la scommessa. Alla fine della sfida, Alice vince e richiede a Bob il pagamento della quota concordata. A questo punto Bob si rifiuta di pagare poiché sostiene che Alice abbia barato.

Questo semplice esempio dimostra che, anche se le condizioni del contratto sono state rispettate (Alice vince la sfida), c'è sempre bisogno di fidarsi che la controparte (Bob) tenga fede all'accordo.

Utilizzando uno *smart contract* la fiducia verso il destinatario è intrinseca nel sistema in quanto l'esecuzione del contratto è decentralizzata e il meccanismo di consenso garantisce che il contratto venga eseguito in modo

[8] Solidity è un linguaggio di programmazione ad alto livello, orientato agli oggetti appositamente ideato per lo sviluppo di *smart contract* su blockchain in particolar modo su Ethereum.

[9] In informatica, il bytecode, è un linguaggio intermedio più astratto del linguaggio macchina usato per descrivere le operazioni di un programma.

corretto e secondo le logiche con il quale è stato implementato. Come indicato nella Sezione 2.1 la rete Ethereum disincentiva i comportamenti disonesti e favorisce i nodi che si comportano in modo "corretto".

2.3.2 Invocazione

L'esecuzione di un **contratto intelligente** avviene solo dopo che un utente invia una transazione sulla blockchain, con l'indirizzo dello smart contract come destinatario. L'esecuzione di un contratto non è scatenata da altri eventi se non dalle transazioni avviate dagli utenti della rete.

Con un un contratto intelligente, input specifici producono output univoci, perciò uno smart contract si comporta come un programma **deterministico**[10].

Quando uno *smart contract* riceve fondi da un utente, il suo codice viene eseguito da tutti i nodi della rete in modo decentralizzato. Questo avviene al fine di raggiungere il consenso in merito all'esito dell'esecuzione e al valore risultante.

2.3.3 Creazione

Come anticipato nella Sezione 2.2.6, i contratti possono essere creati tramite speciali transazioni che hanno come destinatario l'indirizzo nullo. Il bytecode del contratto deve essere incluso nel campo *data* della transazione. L'indirizzo del contratto creato è deterministico, e può essere facilmente ottenuto tramite la seguente formula:

$$contract\_address = keccak(address, nonce)^{11}$$

Viene applicata la funzione hash Keccak [8] all'indirizzo del mittente concatenato con il suo *nonce*. Ricordiamo che il nonce di un account corrisponde al numero di transazioni che ha inviato fino a quel momento.

Autodistruzione

Una volta pubblicati sulla rete, i contratti non possono essere né eliminati né cancellati poiché il loro codice risiede sulla blockchain che è per sua natura

[10] Un programma deterministico è un programma che, dato un particolare input, produce sempre lo stesso output.

[11] L'indirizzo viene calcolato con la formula riportata nel caso in cui il client utilizzi la funzione CREATE. Durante la hard fork *Byzantium* è stata introdotta la funzione CREATE2 che consente di creare uno smart contract ad un indirizzo deterministico che tiene conto anche di un *salt* passato dall'utente. Tutte i dati in nostro possesso sono precedenti all'introduzione della CREATE2.

immutabile (vedi Sezione 2.2.2). Un'eventuale cancellazione è possibile solo nel caso in cui lo sviluppatore includa all'interno del contratto la funzionalità di autodistruzione. Questa funzionalità deve essere necessariamente implementata prima della pubblicazione dello *smart contract*, altrimenti non sarà più possibile eliminarlo.

L'autodistruzione di un contratto comporta la cancellazione del codice del contratto stesso e del suo stato interno, e ogni successiva transazione inviata al contratto non comporterà l'esecuzione di nessun codice.

2.3.4 Eventi e log

Gli *smart contract* possono emettere eventi e scrivere log ogni qual volta che una transazione che li riguarda viene validata e inserita sulla blockchain [20]. Le dapp[12] o qualsiasi altro client collegato ad Ethereum tramite l'interfaccia JSON-RPC può mettersi in ascolto di eventi e agire di conseguenza. Gli eventi sono un ottimo modo per segnalare alla rete che un particolare evento ha avuto luogo e sono utili per comprendere il comportamento di uno *smart contract* e monitorarne il suo utilizzo.

L'evento più comune sulla blockchain Ethereum è il *Transfer* event, emesso dai contratti che rispettano lo standard **ERC20**[13] ogni qual volta che qualcuno trasferisce token. Il Codice 2 mostra la firma dell'evento *Transfer* nel linguaggio Solidity.

```
1 event Transfer(address indexed from, address indexed to, uint256 value);
```

Codice 2: Firma transfer event ERC20

Log

Log ed eventi non sono concetti completamente scollegati, ma sono interconnessi tra di loro. Quando viene emesso un evento, i suoi argomenti vengono memorizzati all'interno dei log della transazione. Il Codice 3 mostra la struttura dati *logs* delle transazioni. I campi di maggior rilevanza sono:

- **topics** - Usati per descrivere l'evento, il primo elemento corrisponde tipicamente alla firma dell'evento stesso[14]. Al massimo può contenere 4 elementi compresa la firma;

[12] Decentralized application

[13] Standard per il trasferimento e lo scambio di token.

[14] La firma dell'evento è calcolata mediante la funzione Keccak-256 applicata al nome del metodo, inclusi i tipi dei suoi parametri (unit254, string, etc.).

- **data** - Dati addizionali, contiene i rimanenti parametri non memorizzati in *topics*;

Come mostrato nel Codice 2 è possibile aggiungere l'attributo *indexed* fino a tre dei parametri dell'evento che causa la loro memorizzazione all'interno dell'array *topics* anziché all'interno del campo *data* dei log. Il Codice 3 mostra l'array *logs* presente all'interno di ogni transazione che ha causato l'emissione di un *transfer* event con firma simile a quella mostrata nel Codice 2. L'array *topics*, presente nell'unico log, contiene 3 elementi: il primo corrisponde alla firma dell'evento *transfer* menzionato in precedenza, mentre gli altri due sono gli argomenti dichiarati come indexed quindi indirizzo mittente e indirizzo destinatario. Il log può quindi essere letto come: *Trasferimento di token da 0xfD3...55a a 0x124...0B4.*

```
{
    "logs": [
        {
            "address": "0x95222290DD7278Aa3Ddd389Cc1E1d165CC4BAfe5",
            "topics": [
                "0xddf252ad1be2c89b69c2b068fc378daa952ba7f163c4a11628f55a",
                "0xfD3dA69c3E6Fc48489169f5F6855a0b8E9AD4A22",
                "0x124E31000D506C1AC68eB64453074cdC4DD460B4"
            ],
            "data": "0x1",
            "blockNumber": "0xcb3d",
            "transactionIndex": "0x0",
            "logIndex": "0x0",
            "@type": "Log"
        }
    ]
}
```

Codice 3: Struttura log contenuta all'interno di un oggetto transazione

Capitolo 3

Neo4j: un graph database

Nel presente capitolo, esploreremo Neo4j, un potente graph database scelto per rappresentare i dati delle transazioni Ethereum. Descriveremo in modo dettagliato il suo funzionamento soffermandoci sul linguaggio di interrogazione e sugli strumenti che offre per l'importazione di grandi quantità di dati.

3.1 Graph database

Un graph database o database a grafo è una tipologia di database NoSQL[1] che utilizza nodi e archi per rappresentare l'informazione e permette di memorizzare ed esplorare grafi in modo efficiente e naturale. In un graph database i nodi rappresentano le entità mentre gli archi rappresentano le relazioni. Come illustrato nella Figura 3.1, un graph database rappresenta le informazioni mediante un grafo orientato. Nella figura sono rappresentati nodi con varie etichette, identificabili dal loro colore e collegati tra di loro tramite frecce orientate che rappresentano relazioni.

I database a grafo permettono una rappresentazione dei dati completamente diversa rispetto a quella dei database relazionali in cui i dati sono organizzati in tabelle con righe e colonne e le relazioni sono definite attraverso relazioni esterne (foreign key). L'elaborazione di grafi è diventata una parte fondamentale per molti settori dell'informatica come quello del machine learning, scienze computazionali, applicazioni mediche, analisi di social network e molto altro [21]. È perciò importante trovare un graph database adatto ad ogni diversa applicazione.

[1]Database di tipo non relazionale.

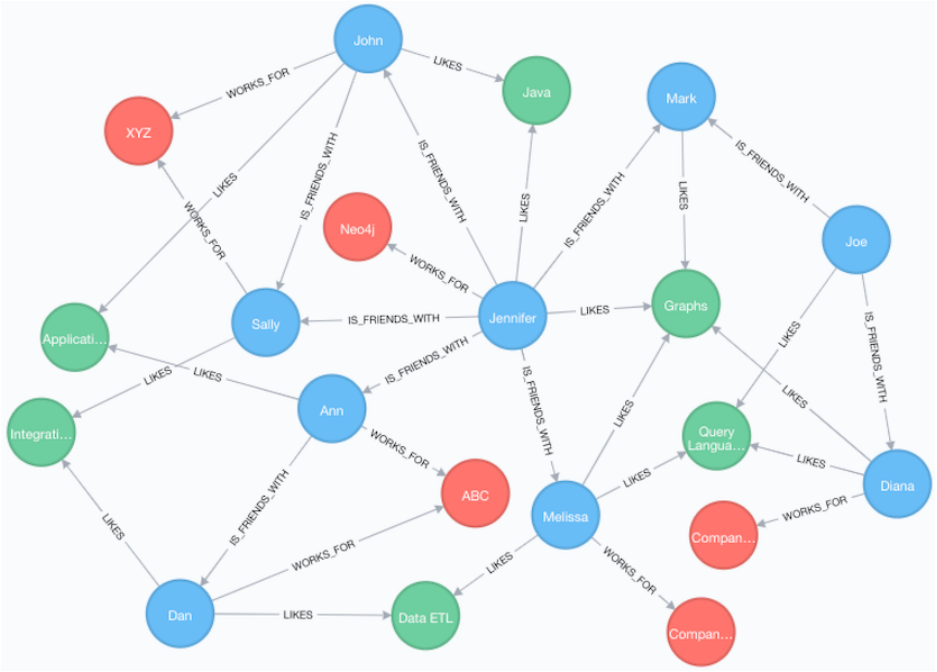

Figura 3.1: Esempio grafo [22]

3.2 Neo4j

Neo4j è il graph database rilasciato nel 2007 leader nel settore che offre una piattaforma potente per la gestione e l'analisi di grafi. È particolarmente adatto per le applicazioni che richiedono l'analisi delle relazioni come reti sociali, transazioni blockchain e molto altro. Una delle caratteristiche distintive di Neo4j è la sua libreria **GDS** (Graph Data Science), che fornisce un insieme di algoritmi avanzati per l'analisi dei grafi tra cui clustering, rilevamento delle comunità, centralità e molto altro.

La suite Neo4j mette inoltre a disposizione software utili alle varie fasi di sviluppo e progettazione di un graph database. Particolarmente utili sono i tool **neo4j-admin** e **neo4j-cli** che offrono la gestione del database e strumenti per importare grandi quantità di dati. Altra peculiarità di Neo4j è la vasta comunità di sviluppatori e utenti che forniscono supporto e contribuiscono alla crescita e all'evoluzione della piattaforma. In rete sono presenti innumerevoli forum dove si trattano gli argomenti e casi d'uso più disparati su Neo4j. Tra le principali caratteristiche di Neo4j troviamo:

- Database scalabile e ottimizzato per memorizzare e interrogare grafi distribuiti su cluster di server;

- Linguaggio di interrogazione molto semplice da comprendere e con curva di apprendimento bassa;
- Molto flessibile, gestisce molto bene dati con formati non strutturati;
- Architettura cluster distribuita ad alte prestazioni;
- Interfaccia grafica intuitiva e semplice da usare (Vedi Figura 3.2);

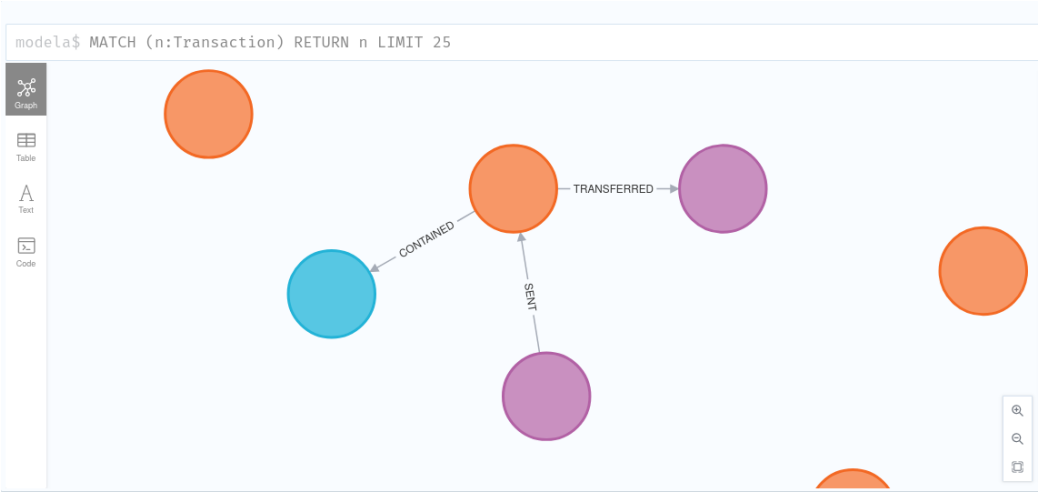

Figura 3.2: Interfaccia grafica Web Neo4j

3.3 Nodi e relazioni

La struttura di un grafo consiste in un insieme di nodi collegati tra di loro mediante relazioni come mostrato in Figura 3.3. L'immagine mostra un grafo con tre nodi (cerchi) e tre relazioni (frecce).

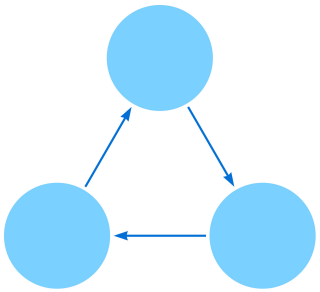

Figura 3.3: Struttura a grafo [23]

Neo4j utilizza un modello di grafo chiamato *property graph database* in cui:

1. I **nodi** rappresentano entità di un dominio, e possono avere zero o più etichette che definiscono di che tipo sono;
2. Le **relazioni** descrivono un collegamento tra un nodo sorgente e un nodo destinatario. Hanno sempre una direzione e devono obbligatoriamente avere un tipo per definire la relazione che rappresentano;
3. **Nodi** e **relazioni** possono entrambi avere proprietà (chiave-valore) che li descrivono ulteriormente;

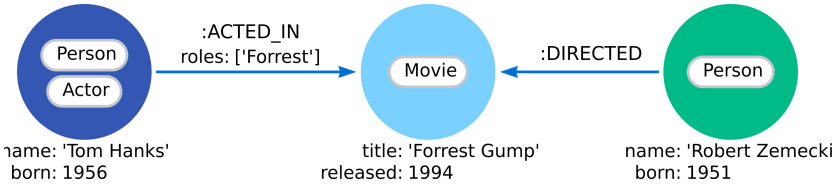

Figura 3.4: Grafo delle proprietà [23]

La Figura 3.4 illustra un semplice grafo in Neo4j con tre nodi (ognuno con etichette differenti) e due tipi di relazione. Si può notare come il nodo di colore blu abbia due etichette "Person" e "Actor" a indicare che rappresenta sia una persona che un attore.

Le relazioni descrivono come le connessioni tra un nodo sorgente e un nodo destinatario sono collegate. È possibile per un nodo avere un relazione verso se stesso. Le relazioni hanno sempre una direzione, tuttavia in Neo4j possono essere ignorate nel caso non siano utili. Questo significa che non è necessario aggiungere relazioni duplicate in direzioni opposte a meno che non siano necessarie per rappresentare il modello dei dati (vedi Figura 3.5).

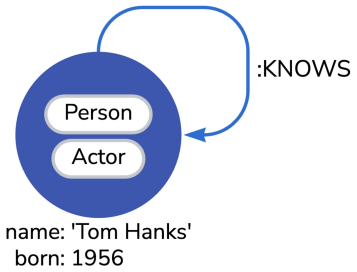

Figura 3.5: Self relation [23]

3.4 Query language

I database Neo4j sono interrogabili mediante **Cypher**, il linguaggio di interrogazione sviluppato specificamente per Neo4j, progettato per facilitare l'interazione con grafi in modo intuitivo ed espressivo.

La sintassi di Cypher è ispirata alla struttura dei grafi stessi, rendendo le query facili da leggere e scrivere anche per gli utenti meno esperti. Utilizzando una sintassi analoga a quella mostrata in Figura 3.6, gli utenti possono specificare le relazioni tra i nodi e le proprietà dei nodi stessi per recuperare informazioni significative dal database. La stessa Figura 3.6 illustra inoltre un esempio di relazione tra nodi con stessa etichetta. Tale relazione denominata *Loves* è definita tra nodi di tipo *Person*. Si può notare inoltre che per ciascun nodo è definita una proprietà *name* che rappresenta il nome della persona e permette di identificarla rispetto a tutte le altre.

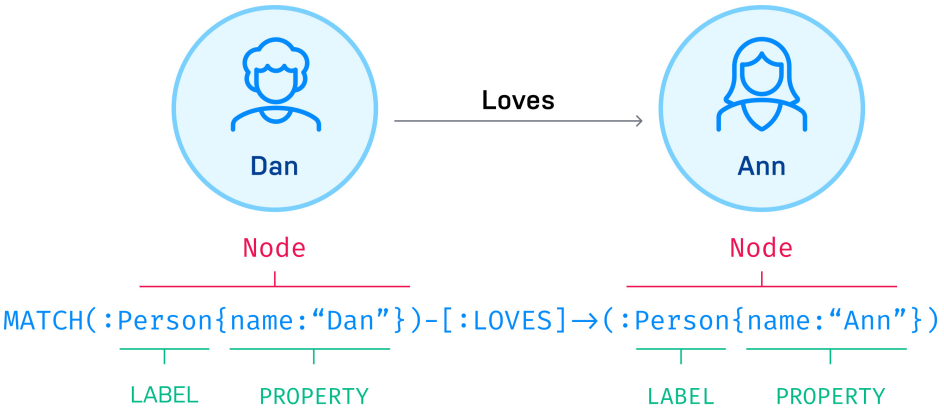

Figura 3.6: Relazioni in Cypher[22]

Nel Codice 4 sono illustrare tre query di esempio utilizzate per mostrare la sintassi del linguaggio di interrogazione Cypher. Supporta una vasta gamma di operazioni di interrogazione, inclusi filtri, corrispondenze di pattern, aggregazioni, ordinamenti e altro ancora.

```
// Query a
MATCH
    (a :Person {name: "Daniel"})-[:KNOWS]->(b :Person)
RETURN
    COUNT(*);

// Query b
MATCH
    (p:Person {name: "Tom Hanks"})-[r:ACTED_IN]->(m:Movie)
RETURN
    m.title,
    r.roles;

// Query c
MATCH
    (person:Person)-[:WORKS_FOR]->(company)
WHERE
    company.name STARTS WITH "Company"
AND EXISTS {
  MATCH
    (person)-[:LIKES]->(t:Technology)
  WHERE
    COUNT { (t)<-[:LIKES]-() } >= 3
}
RETURN
    person.name as person,
    company.name AS company;
```

Codice 4: Semplice query Cypher

Inoltre, offre funzionalità avanzate per la manipolazione dei dati, come l'inserimento, l'aggiornamento e la cancellazione dei nodi e delle relazioni.

Grazie alla sua semplicità d'uso e alla sua potenza, Cypher è diventato lo standard de facto per l'interrogazione dei graph database in Neo4j.

3.5 Data import

Nel nostro scenario di analisi e visualizzazione delle transazioni Ethereum, uno degli aspetti cruciali è stato l'importazione di grandi quantità di dati in modo efficiente e accurato. In questa sezione esploreremo le sfide e le strategie coinvolte per ottimizzare l'importazione di enormi volumi di dati all'interno del nostro graph database Neo4j.

In particolare mostreremo prima la tecnica di importazione dei dati tramite la funzionalità *LOAD CSV* di Cypher e successivamente il più ottimiz-

zato comando *neo4j-admin import* che ci ha consentito di importare grandi quantità di dati in tempi ridotti. Quest'ultimo è stato infine utilizzato per importare la porzione di dataset utilizzata per le analisi sperimentali del Capitolo 6.

3.5.1 Metodo 1: Cypher LOAD CSV

Nel processo di importazione dei dati su Neo4j, abbiamo inizialmente esplorato l'utilizzo del comando *LOAD CSV* messo a disposizione da Cypher. Questo approccio ci ha garantito una notevole flessibilità nell'importazione dei dati, poiché abbiamo potuto sfruttare appieno tutti i comandi e le funzionalità offerte dal linguaggio. Questa caratteristica ci ha inoltre permesso di eseguire operazioni avanzate di manipolazione dei dati durante il processo di importazione. È stato possibile eseguire operazioni di filtraggio, aggregazione e trasformazione dei dati direttamente all'interno del comando *LOAD CSV*, utilizzando le stesse espressioni e funzioni utilizzate in una query Cypher tradizionale.

```
1,ABBA,1992
2,Roxette,1986
3,Europe,1979
4,The Cardigans,1992
```

Codice 5: File artists.csv

```
LOAD CSV FROM  'file:///artists.csv' AS row
MERGE (a:Artist {name: row[1], year: toInteger(row[2])})
RETURN
  a.name,
  a.year
```

Codice 6: Esempio comando Cypher LOAD CSV

Il Codice 6 mostra la sintassi per importare il contenuto del file CSV mostrato nel Codice 5. La flessibilità di questo comando risiede nella sua capacità di sfruttare appieno le potenzialità del linguaggio di interrogazione Cypher. È possibile infatti utilizzare tutti i costrutti del linguaggio per effettuare eventuali operazioni preliminari sui dati.

- **LOAD CSV** - Indica a Neo4j di caricare il file *CSV* denominato "artists.csv" con le intestazioni di colonna;

- **MERGE** - Clausola utilizzata per creare nodi nel grafo. Se il nodo esiste già non viene modificato;

Nonostante la flessibilità del comando, questo comporta notevoli svantaggi in termini prestazionali. Questa inefficienza è dovuta al fatto che il comando processa le righe del file di input una alla volta, aumentando significativamente i tempi di importazione rendendolo poco adatto per dataset di grandi dimensioni. Secondo la documentazione ufficiale di Neo4j [24] il comando *LOAD CSV* presenta la seguente limitazione:

LOAD CSV is great for importing small- or medium-sized datasets (up to 10 million records). For datasets larger than this, you can use the neo4j-admin database import command. This allows you to import CSV data to an unused database by specifying node files and relationship files.

Viste quindi le limitazioni intrinseche del comando, abbiamo dedotto che non sarebbe stato adatto per l'importazione di grandi quantità di dati e questo ci ha spinti ad esplorare altre strategie che potessero garantire prestazioni migliori e una gestione più efficiente delle risorse del sistema.

3.5.2 Metodo 2: neo4j-admin import

Il tool *neo4j-admin*, disponibile in tutte le versioni di **Neo4j** (community/enterprise), offre una serie di comandi utili per la gestione e l'amministrazione di DBMS Neo4j. Tra questi strumenti, vi è anche un'apposita funzionalità dedicata all'importazione di grandi quantità di dati a partire da file CSV che rispettano un preciso formato.

```
bin/neo4j-admin database import full
--delimiter=","
--array-delimiter=";"
--skip-duplicate-nodes
--overwrite-destination=true
--nodes=movie.csv
--nodes=actor.csv
--relationships=acted_relationship.csv
movie_db
```

Codice 7: Esempio comando neo4j-admin import

Il Codice 7 mostra la sintassi del comando *neo4j-admin database import* utilizzato per importare dati all'interno del database. Di seguito sono elencati i significati dei parametri più importati del comando.

- ***delimiter*** - Indica il separatore del file CSV;
- ***array-delimiter*** - Specifica il delimitatore utilizzato per separare gli array di dati all'interno del CSV;
- ***skip-duplicates-nodes*** - Indica di ignorare i nodi duplicati durante l'importazione;
- ***nodes*** - Specifica il file CSV contenente le informazioni dei nodi di tipo (se presenti più tipi di nodi, il parametro può essere ripetuto);
- ***relationships*** - Specifica il file CSV contenente i dati delle relazioni;

Questo comando è utilizzato per importare grandi quantità di dati a partire da file *CSV* opportunamente strutturati di nodi e relazioni. Le relazioni vengono create collegando tra di loro i nodi, che devono possedere un ID univoco per poter essere correttamente referenziati. I campi obbligatori per i file CSV dei nodi sono :ID e :LABEL, mentre per i file delle relazioni è necessario specificare :START_ID, :END_ID e :TYPE.

Il Codice 8 e 9 mostra il contenuto dei due file *CSV* utilizzati per l'import dei nodi mostrato nel Codice 7, mentre il Codice 10 mostra il contenuto del file che rappresenta le relazioni tra di essi.

```
movieId:ID,title,year:int,:LABEL
tt0133093,"The Matrix",1999,Movie
tt0234215,"The Matrix Reloaded",2003,Movie;Sequel
tt0242653,"The Matrix Revolutions",2003,Movie;Sequel
```

Codice 8: movie.csv

```
personId:ID,name,:LABEL
keanu,"Keanu Reeves",Actor
laurence,"Laurence Fishburne",Actor
carrieanne,"Carrie-Anne Moss",Actor
```

Codice 9: actor.csv

```
:START_ID,role,:END_ID,:TYPE
keanu,"Neo",tt0133093,ACTED_IN
laurence,"Morpheus",tt0133093,ACTED_IN
carrieanne,"Trinity",tt0133093,ACTED_IN
```

Codice 10: acted_relationship.csv

Il comando *neo4j-admin import* può operare in modalità **completa** o **incrementale**.

Completa

Questo tipo di importazione è utilizzata per caricare tutti i dati dal/i file di input al graph database. Durante l'importazione tutti i dati nel file di input vengono caricati all'interno del database sovrascrivendo eventuali dati esistenti. Questo metodo è utile in caso di una nuova creazione di un graph database a partire da file CSV di input. È stato il metodo scelto per importare i dati nel nostro database.

Incrementale

L'importazione incrementale viene usata per aggiungere solo i dati nuovi o modificati dal file di input al graph database Neo4j, senza influenzare i dati esistenti. Questo metodo è utile quando si desidera aggiungere nuovi dati senza dove effettuare un'importazione completa.

È importante sottolineare che nonostante questo tipo di importazione risulti molto efficace per integrazioni successive al graph database, questa modalità è generalmente più lenta dell'importazione full (completa).

In breve, l'importazione completa sostituisce tutti i dati nel database con i nuovi dati dal file di input, mentre l'importazione incrementale aggiunge solamente nuovi dati mantenendo i dati importati nelle iterazioni precedenti.

3.6 Grafi proiettati

Neo4j offre anche la possibilità di creare *grafi proiettati*, ovvero grafi che rappresentano solamente un sottoinsieme delle entità e delle relazioni del grafo originale. I *grafi proiettati* consentono di concentrarsi su aspetti specifici del grafo originale semplificando così l'analisi e consentendo uno studio più mirato di relazioni ed entità.

Su **Neo4j** i grafi proiettati sono un requisito obbligatorio per poter utilizzare la libreria **GDS** (graph data science) che permette di applicare una vasta gamma di algorirmi tra cui *clustering*, *page rank*, *centrality* e simili.

La proiezione viene fatta a partire da un grafo più ampio utilizzando appositi criteri di selezione di archi e nodi. Il Codice 11 mostra una semplice query che permette la creazione di un grafo proiettato delle sole relazioni che coinvolgono trasferimenti tra *Account*. Il risultato è un grafo monopartito che ha solamente nodi con etichetta *Account* e la relazione *transfer* che li collega.

```
MATCH
    (from: Account)-[:SENT]-(t:Transaction)-[:TRANSFERRED]->(to: Account)
WITH
    gds.graph.project("transfer",from, to) as g
RETURN
    g.graphName AS graph,
    g.nodeCount AS nodes,
    g.relationshipCount AS rels;
```

Codice 11: Esempio proiezione grafo

Capitolo 4

Modelli di grafo per Ethereum

In questo capitolo, verranno mostrati i modelli di grafo proposti per facilitare la rappresentazione dei dati della blockchain Ethereum e semplificarne l'analisi. I modelli presentati possono essere utilizzati da chiunque voglia effettuare uno studio sulla blockchain Ethereum, mediante grafi con strutture semplici e comprensibili. Una parte fondamentale dello studio della blockchain è infatti quella di scegliere il giusto modello di grafo da utilizzare per rappresentare i dati di blocchi e transazioni. La scelta del modello di grafo è un aspetto cruciale, poiché può avere ripercussioni sulle performance di query e algoritmi.

4.1 Grafo delle transazioni

Con l'obiettivo di studiare e analizzare i comportamenti sulla blockchain, la struttura di Ethereum può essere efficacemente rappresentata come un grafo che, visto che tutta la blockchain Ethereum si basa sulle transazioni, chiameremo *grafo delle transazioni*.

Scopo della nostra tesi è quello di presentare modelli che rappresentino il *grafo delle transazioni* che permettano di identificare e studiare anomalie o comportamenti della blockchain in determinati contesti.

Numerosi studi hanno già esplorato e proposto tecniche per l'analisi e la rappresentazione di questo tipo di grafo. Si rimanda il lettore alla Sezione 7.2 per ulteriori considerazioni sugli articoli esistenti e per un confronto più approfondito con il lavoro svolto in questa tesi.

I modelli proposti nel nostro studio sono completi, racchiudono tutte e tre le attività principali della rete e differiscono solamente dal modo in cui vengono rappresentate. Infatti, ad eccezione di alcuni dati specifici, entrambi

i modelli rappresentano le stesse informazioni fondamentali di Ethereum: trasferimenti di ether, creazione contratti e invocazione contratti.

Nel primo modello che chiameremo **Modello TN** (transaction as nodes) abbiamo considerato le transazioni come nodi, mentre nel secondo modello che chiameremo **Modello TE** (transaction as edges) le transazioni sono state trattate come archi.

4.1.1 Account: nodo unico

Entrambi i modelli TN e TE rappresentano gli account della rete (vedi Sezione 2.2.3) come nodi di etichetta *Account*. I nodi *Account* hanno solamente due attributi:

- $address \in \mathbb{N}$ - Indirizzo univoco per ogni account;
- $accountType \in \{0, 1, 2\}$ - Tipo dell'account con $0 = EOA$, $1 = sc$, $2 = unknown$;

È stata scelta una rappresentazione degli account tramite un unico nodo contrassegnato dall'etichetta *Account* anziché utilizzare etichette separate per i diversi tipi di account (*EOA*, *sc*) a causa della presenza degli account sconosciuti che, se rappresentati con etichetta *unknown*, avrebbero reso ambigua l'identità del nodo.

4.2 Modello TN: transactions as nodes

Il **modello TN** è un multigrafo $G_{TN} = (V, E, l, r, p)$, dove:

- V rappresenta l'insieme dei nodi;
- E rappresenta il multinsieme degli archi;
- $l : V \to Labels$ è una funzione che assegna un'*etichetta* ad ogni nodo, con $Labels = \{Account, Block, Transaction, Event\}$. In particolare:
  - *Account*: rappresenta un generico account Ethereum;
  - *Block*: rappresenta l'entità blocco di Ethereum;
  - *Transaction*: rappresenta l'entità transazione di Ethereum;
  - *Event*: rappresenta l'entità evento, emesso da una transazione. È possibile avere fino a 4 eventi per ogni transazione (vedi Sezione 2.3.4);

- $r : E \to Relations$ è una funzione che assegna una *relazione* ad ogni arco del grafo, con $Relations = \{SENT, CONTAINED, TRANSFERRED, CREATED, INVOKED, EMITTED, TO, CHILD\_OF\}$. In particolare:
  - *SENT* descrive un arco $(u, v) \in E$ con $l(u) = Account$, $l(v) = Transaction$ e rappresenta l'azione di invio di una transazione da parte di un account;
  - *CONTAINED* descrive un arco $(u, v) \in E$ con $l(u) = Transaction$, $l(v) = Block$ e rappresenta l'appartenenza delle transazione al blocco;
  - *TRANSFERRED* descrive un arco $(u, v) \in E$ con $l(u) = Account$, $l(v) = Transaction$ e rappresenta l'azione di invio di una transazione da parte di un account;
  - *CREATED* descrive un arco $(u, v) \in E$ con $l(u) = Transaction$, $l(v) = Account$ e rappresenta l'azione di creazione di un contratto;
  - *INVOKED* descrive un arco $(u, v) \in E$ con $l(u) = Transaction$, $l(v) = Account$ e rappresenta l'azione di invocazione di uno *smart contract*;
  - *EMITTED* descrive un arco $(u, v) \in E$ con $l(u) = Transaction$, $l(v) = Event$ e rappresenta l'emissione di un evento scaturito dall'esecuzione di uno *smart contract*;
  - *TO* descrive un arco $(u, v) \in E$ con $l(u) = Transaction$, $l(v) = Account$ ed è usata nel caso in cui l'account destinatario sia di tipo sconosciuto;
  - *CHILD_OF* descrive un arco $(u, v) \in E$ con $l(u) = l(v) = Block$ e rappresenta la relazione tra blocchi successivi. L'insieme dei blocchi di questa relazione compone la blockchain.
- $p : V \to Properties$ è una funzione che assegna una serie di proprietà, rappresentate come tuple, ad ogni nodo del grafo. In particolare, dato un nodo $u \in V$:
  - $l(u) = Account \implies p(u) = (address, accountType)$;
  - $l(u) = Block \implies p(u) = (hash, cumulativeGasUsed, fromAddress, gas, gasPrice, gasUsed, input, nonce, root, status, toAddress, transactionIndex, value, contractAddress)$;
  - $l(u) = Transaction \implies p(u) = (hash, cumulativeGasUsed, fromAddress, gas, gasPrice, gasUsed, input, nonce, root, status, toAddress, transactionIndex, value, contractAddress)$;

– $l(u) = Event \implies p(u) = (hash, type, address, blockNumber, data, logIndex, topics, transactionIndex)$;

La Figura 4.1 mostra la struttura del modello. Possiamo notare i 4 tipi di nodi e i 7 tipi di relazioni.

Figura 4.1: Schema modello TN (transactions as nodes)

La rappresentazione delle transazioni come nodi ha un impatto significativo su altri nodi che contengono informazioni direttamente collegate alle transazioni come quelle dei blocchi in cui sono contenute o quelle degli eventi emessi. In questo modello, abbiamo identificato quattro tipi di nodi e sette tipi di relazioni.

4.3 Modello TE: transactions as edges

Il **Modello TE** come accennato in precedenza non rappresenta le transazioni come nodi bensì come archi. Questo significa che ogni transazione è rappre-

sentata come una relazione tra due nodi, come ad esempio un mittente e un destinatario di un trasferimento di ether o un utente e un contratto intelligente. Questo implica inoltre che ogni attributo delle relazioni può essere direttamente usato come peso degli archi.

Il **modello TE** è un multigrafo $G_{TE} = (V, E, l, r, p_V, p_E)$ dove:

- V rappresenta l'insieme dei nodi;
- E è il multinsieme degli archi;
- $l : V \to \mathit{Labels}$ è la funzione che assegna un'etichetta ad ogni nodo, con $\mathit{Labels} = \{\mathit{Account}\}$. Tutti i nodi del grafo hanno la medesima etichetta *Account* e rappresentano un generico account Ethereum;
- $r : E \to \mathit{Relations}$ è la funzione che assegna una relazione ad ogni arco (u, v) del grafo, con $\mathit{Relations} = \{\mathit{TRANSFERRED}, \mathit{CREATED}, \mathit{INVOKED}, \mathit{TO}\}$. In particolare:
  - *TRANSFERRED* rappresenta l'azione di trasferimento di ether tra account;
  - *CREATED* rappresenta l'azione di creazione di uno smart contract;
  - *INVOKED* rappresenta l'azione di invocazione di uno smart contract;
  - *TO* viene usata nel caso in cui il nodo destinatario della relazione sia di tipo sconosciuto;
- $p_V : V \to \mathit{Properties}_V$ è una funzione che associa ad ogni nodo una tupla di proprietà. Per ogni nodo $u \in V$ si ha $p_V(u) = (\mathit{address}, \mathit{accountType})$;
- $p_E : E \to \mathit{Properties}_E$ è una funzione che associa ad ogni arco una tupla di proprietà. Per ogni arco $(u, v) \in E$ si ha $p_E(u, v) = (\mathit{hash}, \mathit{fromAddress}, \mathit{contractAddress}, \mathit{blockHash}, \mathit{blockNumber}, \mathit{block\_difficulty}, \mathit{block\_extraData}, \mathit{block\_gasLimit}, \mathit{block\_gasUsed}, \mathit{block\_hash}, \mathit{block\_minerAddress}, \mathit{block\_mixHash}, \mathit{block\_nonce}, \mathit{block\_number}, \mathit{block\_ommerCount}, \mathit{block\_parentHash}, \mathit{block\_receiptsRoot}, \mathit{block\_sha3Uncles}, \mathit{block\_stateRoot}, \mathit{block\_timestamp}, \mathit{block\_totalDifficulty}, \mathit{block\_transactionsRoot}, \mathit{cumulativeGasUsed}, \mathit{gas}, \mathit{gasPrice}, \mathit{gasUsed}, \mathit{logs\_address}, \mathit{logs\_block\_number}, \mathit{logs\_data}, \mathit{logs\_index}, \mathit{logs\_topic}, \mathit{logs\_transaction\_hash}, \mathit{logs\_transaction\_index}, \mathit{logs\_type}, \mathit{nonce}, \mathit{root}, \mathit{status}, \mathit{toAddress}, \mathit{transactionIndex}, \mathit{value})$.

Come si può osservare dalla Figura 4.2, nel modello TE esiste una sola tipologia di nodo, corrispondente al generico account Ethereum. Pertanto, tutte le relazioni, ovvero gli archi, hanno come nodi di partenza e di arrivo elementi di tipo *Account*. Inoltre, tutti gli archi rappresentano singole transazioni e sono caratterizzati dal medesimo insieme di proprietà. Tali proprietà riportano informazioni sul blocco nel quale la transazione è avvenuta, sugli eventuali eventi prodotti come conseguenza dell'esecuzione e sulla quantità di denaro trasferito.

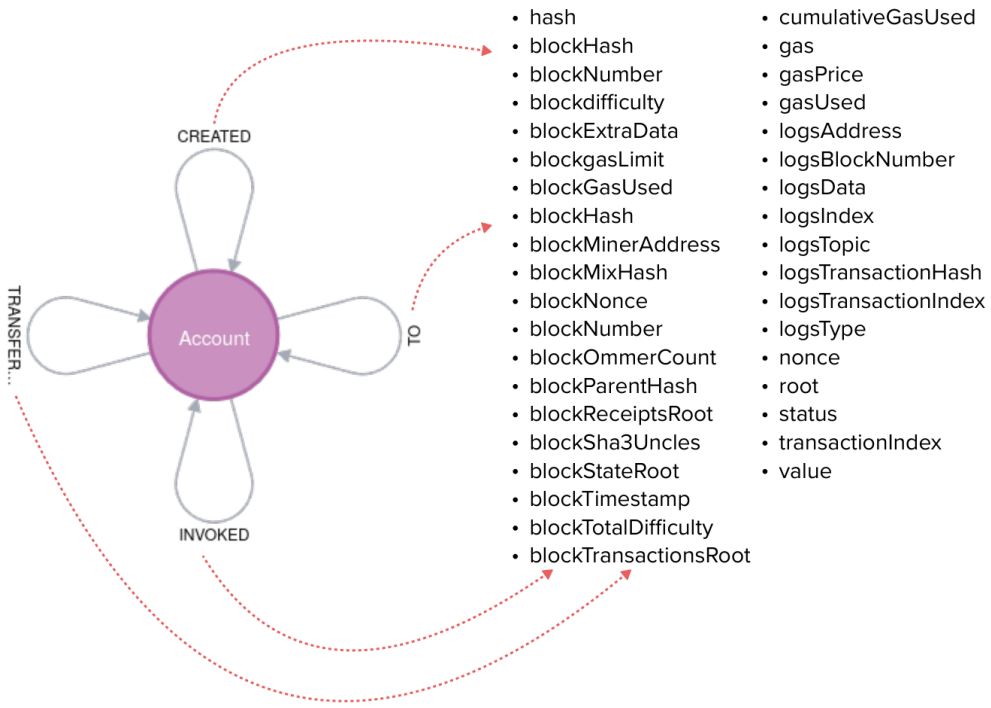

Figura 4.2: Modello TE (transactions as edges)

4.4 Confronto teorico

Nonostante l'obiettivo di entrambi i modelli sia quello di rendere accessibili le informazioni della blockchain in un formato universale e di facile comprensione, le loro strutture sono profondamente differenti. Nel **modello TN** ogni informazione è rappresentata in modo univoco e non viene duplicata all'interno del grafo. Inoltre, avere nodi specifici per ogni entità consente di importare parte distinte dei dati, ad esempio solo le informazioni relative ai blocchi o solo quelle delle transazioni, in modo più flessibile.

Viceversa, nel **modello TE** le informazioni dei blocchi risultano duplicate poiché le singole transazioni vengono rappresentate come un arco tra due nodi e di conseguenza le informazioni dei blocchi non possono essere rappresentate come nodi (non è possibile avere una relazione tra un arco e un nodo) ma devono necessariamente essere all'interno dell'arco. Questo porta inevitabilmente ad un aumento dello spazio di archiviazione necessario per la memorizzazione del grafo (vedi Sezione 6.3). Inoltre l'incapacità del **modello TE** di poter rappresentare i blocchi come nodi, comporta anche l'impossibilità di avere a disposizione nel grafo le informazioni dei blocchi che non hanno nessuna transazione al loro interno. Nonostante questo, il vantaggio del **modello TE** è quello di poter rappresentare le informazioni come un grafo mono-partito, che semplifica notevolmente la sintassi delle query e permette di analizzare la rete in modo più intuitivo e diretto.

Un aspetto fondamentale da tenere in considerazione sono i pesi degli archi: nonostante il **modello TN**, a differenza del **modello TE**, non presenti attributi sulle relazioni è comunque possibile utilizzare gli attributi dei nodi intermedi come pesi.

Capitolo 5

Implementazione

In questo capitolo verrà introdotto il dataset utilizzato e verranno illustrate le tecniche applicate e i tool di supporto creati per facilitare l'elaborazione dei dati a nostra disposizione. Ricordiamo che il dataset è stato utilizzato per la costruzione dei modelli di grafo introdotti nel Capitolo 4, sui quali poi sono state eseguite query e analisi per la valutazione delle loro performance.

5.1 Ethereum dataset

Il dataset utilizzato ai fini del nostro studio è lo stesso utilizzato nelle analisi dell'articolo [25]. È composto da 19 file in formato JSON compresso[1] ognuno dei quali contiene una porzione delle transazioni che vanno dal blocco 0 (**30 Luglio 2015**) al blocco 14 999 999 (**21 Giugno 2022**). Ogni file contiene un array in cui ogni elemento rappresenta un blocco della blockchain. I dati sono stati acquisiti mediante le seguenti chiamate **JSON RPC**[2]ad un full node Ethereum.

- `eth_getBlockByHash` - Permette di ottenere le informazioni di un blocco (compresi gli hash di tutte le transazioni al suo interno) dato in input il block number;
- `eth_getTransactionReceipt` - Ritorna le informazioni della ricevuta di una transazione dato in input il suo hash;

[1]Il file disponibili sono compressi nel formato gzip.
[2]API standard di cui ogni client Ethereum è provvisto. Permette di interagire con la rete al più basso livello.

Blocchi

Il Codice 12 mostra un esempio della struttura JSON di un blocco contenuto nel dataset. Come spiegato dettagliatamente nella Sezione 2.2.4 questa struttura dati contiene tutte le informazioni rilevanti di ogni blocco che compone la blockchain Ethereum.

```
{
  "hash": "0x4e3...bdd",
  "number": 13674,
  "difficulty": "0x153886c1bbd",
  "extraData": "0x657468706f6f6c2e6f7267",
  "gasLimit": 21003,
  "gasUsed": 21000,
  "logsBloom": "0x0",
  "mixHash": "0xb48...cf4",
  "nonce": "0xba4f8ecd18aab215",
  "parentHash": "0x5a4...bfc",
  "receiptsRoot": "0xfe2...8c4",
  "sha3Uncles": "0x1dc...347",
  "stateRoot": "0x0e0...169",
  "timestamp": 1438918233,
  "totalDifficulty": "0x97a50222edba99",
  "transactionsRoot": "0x451...598",
  "@type": "Block",
  "miner": {
    "@type": "Account",
    "address": "0xe6a7a1d47ff21b6321162aea7c6cb457d5476bca"
  },
  "ommerCount": "0x0",
  "transactions": [...]
}
```

Codice 12: Esempio dati blocco (JSON)

Transazioni

Analogamente alla struttura dati mostrata in precedenza, il Codice 13 mostra le informazioni a nostra disposizione per ogni singola transazione contenute nel dataset. Si rimanda il lettore alla Sezione 2.2.6 per maggiori dettagli sui singoli campi.

```json
{
  "hash": "0x218...57e",
  "blockNumber": "0xcb3d",
  "blockHash": "0x3a1...e28",
  "gas": 90000,
  "gasPrice": 57380137580,
  "input": "0x432",
  "nonce": "0x1f",
  "transactionIndex": "0x0",
  "value": 0,
  "logs": [
    {
      "address": "0x5564886ca2c518d1964e5fcea4f423b41db9f561",
      "topics": [
        "0xa66...9bc",
        "0x4d7...000"
      ],
      "data": "0x",
      "blockNumber": "0xcb3d",
      "transactionIndex": "0x0",
      "logIndex": "0x0",
      "@type": "Log"
    }
  ],
  "root": "0x31e...ad5",
  "cumulativeGasUsed": "0xab4d",
  "gasUsed": 43853,
  "status": "0x0",
  "toAddress": "0x5564886ca2c518d1964e5fcea4f423b41db9f561",
  "fromAddress": "0x3d0768da09ce77d25e2d998e6a7b6ed4b9116c2d"
}
```

Codice 13: Esempio dati transaction (JSON)

5.2 Adattamento dei dati al modello di grafo

Durante la fase di progettazione dei modelli (vedi Capitolo 4) è stata tenuta in considerazione la struttura dei dati grezzi del dataset a nostra disposizione. Tuttavia, alcune modifiche si sono rese necessarie per garantire che i dati fossero rappresentati in modo appropriato all'interno dei modelli. Poiché il supporto dei tipi di dato in Neo4j è limitato solo a stringhe, numeri, booleani e liste, si è resa necessaria un'attenta valutazione prima di trasformare i dati grezzi in formati compatibili. Questo è risultato essere di particolare rilevanza durante la fase di import del **modello TE** nel quale è stato necessario trovare non solo un modo per includere le informazioni dei blocchi all'interno

degli archi transazione, ma anche riuscire a codificare in modo coerente le informazioni di array di tipi non nativamente supportati da Neo4j (vedi array *logs* in Codice 13).

Oggetti annidati

Gli oggetti annidati sono stati trattati utilizzando una tecnica chiamata *flattening*[3] che ha consentito di trasformali in una forma più semplice e piatta all'interno dell'oggetto padre. Questo ha permesso di "collassare" le informazioni dei blocchi all'interno delle relazioni tra nodi di tipo *account* del **modello TE**.

Il Codice 14 mostra un esempio di *flattening* applicato alle voci dell'array dei logs relativi ad una transazione.

```
{
    "log": {
        "address": "0x555...561",
        "topics": [
        "0xa66...9bc",
        "0x4d7...000"
        ],
        "data": "0x",
        "blockNumber": "0xcb3d",
        "transactionIndex": "0x0",
        "logIndex": "0x0",
        "@type": "Log"
    }
}
```

```
{

    "log_address": "0x555...561",
    "log_topics": [
    "0xa66...9bc",
    "0x4d7...000"
    ],
    "log_data": "0x",
    "log_blockNumber": "0xcb3d",
    "log_transactionIndex": "0x0",
    "log_logIndex": "0x0",
    "log_@type": "Log"

}
```

Codice 14: A sinistra: oggetto iniziale, a destra: oggetto ottenuto dopo flattening.

Array di oggetti

La presenza dell'array di oggetti *logs* (vedi Codice 13) ha rappresentato un altro elemento di ostacolo per la corretta rappresentazione dei dati in particolar modo nel **modello TE** che, a differenza del **modello TN**, non permette di avere relazioni verso altri nodi se non quelli di tipo *account*. Infatti, nel **modello TN** il problema è stato implicitamente risolto dalla presenza di più

[3]Dall'inglese piatto, appiattimento.

nodi di tipo *Event* per ogni log emesso (relazione 1 a n) rendendo non necessario l'adattamento dei dati. La codifica di array di oggetti nel **modello TE** è stata effettuata utilizzando degli appositi array paralleli come mostrato nel Codice 15.

```
{
    "logs": [
    {
        "address": "0x555...561",
        "topics": [
            "0xa66...9bc"
        ],
        "data": "0x",
        "blockNumber": "0xcb3d",
        "transactionIndex": "0x0",
        "logIndex": "0x0",
        "@type": "Log"
    },
    {
        "address": "0x531...5a1",
        "topics": [],
        "data": "0x",
        "blockNumber": "0xcb3d",
        "transactionIndex": "0x1",
        "logIndex": "0x0",
        "@type": "Log"
    }
    ]
}
```

```
{

    "logs_address": [
        "0x555...561",
        "0x531...5a1" ],
    "logs_topics": [
        "0xa66...9bc|-1|-1|-1",
        "-1|-1|-1|-1" ],
    "logs_data": [
        "0x",
        "0x" ],
    "logs_blockNumber": [
        "0xcb3d",
        "0xcb3d" ],
    "logs_transactionIndex": [
        "0x0",
        "0x0" ],
    "log_logIndex": [
        "0x0",
        "0x01" ],
    "logs_@type": [
        "Log",
        "Log"]
}
```

Codice 15: Memorizzazione di array di oggetti complessi in array paralleli di tipi nativi.

Per ricostruire l'elemento di indice 0 dell'array iniziale è sufficiente prendere tutti gli elementi di indice 0 degli array paralleli. L'elemento *topics* essendo un array è stato ulteriormente codificato come una stringa in cui ogni elemento è separato dal successivo dallo speciale separatore "|". Inoltre, dal momento che ci possono essere al massimo 4 parametri per ogni log (vedi Sezione 2.3.4), gli elementi non valorizzati sono stati codificati con il valore -1.

5.3 Classificazione degli indirizzi

La classificazione degli indirizzi è una fase preliminare che ha consentito di classificare gli indirizzi degli account nei tipi menzionati nella Sezione 4.1.1. Il risultato ottenuto da questo processo è stato utilizzato per arricchire l'input della successiva fase di classificazione delle transazioni come illustrato nella Figura 5.1.

Una corretta distinzione tra *EOA* e *smart contract* permette infatti di stabilire se una transazione diretta verso un account è un semplice trasferimento di ether oppure un'invocazione di uno *smart contract*. Inoltre, la conoscenza delle transazioni effettuate dagli account fino ad un certo momento della blockchain permette di inferire il tipo di indirizzo in blocchi precedenti. Ad esempio una transazione t fatta nel blocco n consente di classificare anche tutte le transazioni fatte nei blocchi precedenti ad n che hanno come destinatario lo stesso indirizzo del mittente della transazione t come dirette verso *EOA* (sappiamo che solo gli utenti possono avviare transazioni).

La classificazione degli indirizzi è stata fatta utilizzando solo ed esclusivamente il dataset a nostra disposizione tramite apposite **euristiche** che verranno illustrate in questo capitolo.

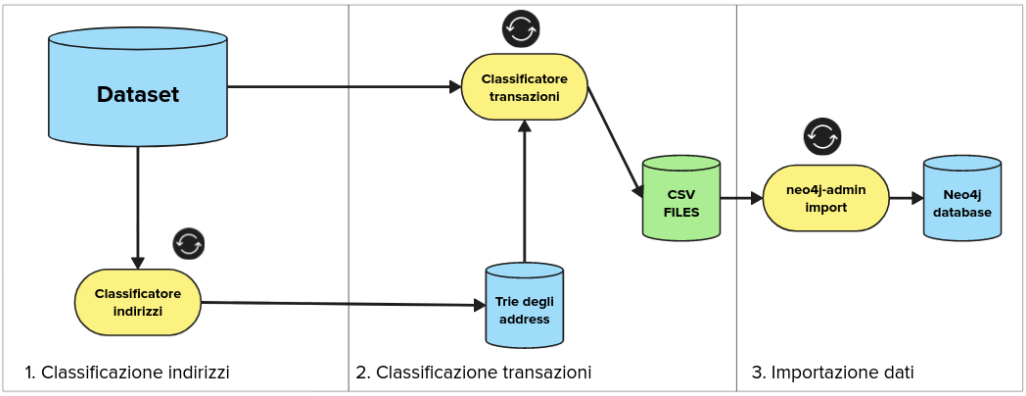

Figura 5.1: Step classificazione ed importazione dati

La Figura 5.1 mostra i 3 step successivi utilizzati per classificare i dati e importarli all'interno di Neo4j.

5.3.1 Trie degli indirizzi

La preliminare classificazione degli indirizzi del dataset è stata effettuata mediante l'ausilio di uno script Python. Sulla falsariga del *MPT* (vedi Sezione 2.2.2) di Ethereum abbiamo scelto di utilizzare un semplice *trie* per

memorizzare gli indirizzi degli account classificati. Il **trie** ha come unica informazione sulle foglie il tipo di account al quale appartiene l'indirizzo (vedi Sezione 2.2.3).

- ***EOA*** - Indirizzo di un *externally owned account*;
- ***sc*** - Indirizzo di uno *smart contract*;
- ***unk*** - Indirizzo sconosciuto, non classificato;

L'utilizzo di un **trie** ci ha consentito di ridurre notevolmente lo spazio necessario per memorizzare gli indirizzi e ha ridotto il tempo di lookup durante la fase di classificazione delle transazioni.

Euristiche

Le euristiche utilizzate per la classificazione degli indirizzi destinatari si basano su alcune proprietà e caratteristiche delle transazioni di Ethereum.

1. **Indirizzo mittente** - L'indirizzo del mittente di una transazione sarà sicuramente un *EOA* poiché come visto nella Sezione 2.2.6 sono gli unici tipi di account che possono avviare una transazione;
2. **Eventi in transazione** - Se all'interno della ricevuta della transazione troviamo degli eventi (vedi Sezione 2.3.4), significa che questa transazione ha scaturito l'invocazione di uno *smart contract* che ha emesso degli eventi e di conseguenza possiamo classificare con sicurezza l'indirizzo destinatario come *smart contract*;
3. **Indirizzo destinatario nullo** - Se l'indirizzo del destinatario risulta essere nullo, significa che l'intento della transazione era quello di creare uno *smart contract* di cui possiamo facilmente calcolarne l'indirizzo (vedi Sezione 2.3.3);

Se nessuna delle sopra citate euristiche viene soddisfatta, l'indirizzo verrà classificato come *sconosciuto*.

Limiti euristiche

Le euristiche sopra menzionate presentano alcune limitazioni che è doveroso spiegare per avere una visione completa e accurata. In primo luogo, l'euristica numero 2 risulta corretta fintanto che lo *smart contract* destinatario non venga cancellato mediante autodistruzione (vedi Sezione 2.3.3). In un

simile caso, l'indirizzo verrebbe comunque classificato come *smart contract*. Inoltre, riguardo l'euristica numero 3 le transazioni dirette verso l'indirizzo nullo non sono il solo modo di creare nuovi *smart contract*, ma è possibile creare un contratto a partire da un altro contratto usando uno dei due *opcode*[4] CREATE o CREATE2. Questo significa che gli *smart contract* creati da altri contratti verranno classificati come *unk* nel caso in cui nessuna delle euristiche precedenti sia stata soddisfatta.

5.4 Preparazione dei dati

Data la dimensione e complessità del dataset, si è reso necessario sviluppare uno script personalizzato[5] in Python per eseguire il parsing[6], il data cleaning e la conversione in *CSV*. Lo script è stato progettato per gestire in modo efficiente la lettura di file JSON di grandi dimensioni senza occupare una grande quantità di memoria.

Lo script principalmente itera sui dati delle singole transazioni e dei blocchi ed esegue le seguenti operazioni:

1. **Data streaming** - Fase in cui i dati vengono elaborati in tempo reale mentre vengono generati o ricevuti;
2. **Data cleaning** - Processo di individuazione, correzione e rimozione di errori all'interno di un dataset;
3. **Pre-processing** - Processo di preparazione preliminare dei dati;
4. **Data split** - Processo di divisione del dataset principale in due o più dataset finali;
5. **Conversione in CSV** - Processo finale di conversione nel formato adatto per Neo4j;

5.4.1 Data streaming

Come specificato all'inizio del capitolo, i file a nostra disposizione sono file **JSON** compressi di dimensione compresa tra 1GB e 10GB. Quando ci troviamo di fronte a grandi file testuali compressi, il caricamento per intero in

[4]Gli opcode sono le istruzioni base eseguite dalla EVM.

[5]Tutto il codice prodotto è disponibile per la consultazione nel seguente repository: https://github.com/daniel-97/computer-science-thesis

[6]Il parsing è un processo che analizza un flusso continuo di dati in ingresso in modo da valutarne la correttezza ed applicare eventuali correzioni.

memoria è da escludere in quanto anche con l'utilizzo di nodi con elevata memoria è possibile che risulti insufficiente. Per ovviare a questo problema è stata utilizzata la libreria Python **ijson** che consente lo streaming e il parsing di file JSON senza richiederne il caricamento in memoria ma leggendo solamente piccole porzioni alla volta.

5.4.2 Data cleaning

Il data cleaning, o pulizia dei dati, è un processo fondamentale nell'analisi dei dati che mira a identificare, correggere errori, anomalie e inconsistenze presenti in un grande campione di dati. Durante il processo di data cleaning diversi passaggi vengono eseguiti per rendere i dati corretti, completi e coerenti con il nostro obiettivo. Un altro vantaggio del data cleaning è la riduzione della dimensione complessiva del dataset. Rimuovendo alcuni campi non necessari per i nostri scopi, si riduce notevolmente la dimensione complessiva dei dati da elaborare. Questo può migliorare in modo considerevole le prestazioni delle query e la dimensione finale del graph database. Principalmente sono stati eliminati i dati che sono stati considerati non necessari agli scopi della nostra tesi, o che in gran parte aveano valori nulli o con poco significato. Alcuni dei campi eliminati sono elencati di seguito.

- **logsBloom** - Rappresentazione esadecimale di un Bloom filter che riassume gli eventi lanciati dal blocco/transazione;
- **@type** - Tipo della transazione [26], nel nostro caso sempre legacy;
- **r, s, v** - Valori necessari per la firma della transazione;

5.4.3 Pre-processing

Per facilitare l'utilizzo e l'interrogazione dei dati sono state applicate alcune conversioni dei campi stringa nei corrispettivi valori numerici. È importante notare che le operazioni di casting e conversione a esadecimale sono computazionalmente onerose e richiedono una quantità significativa del tempo totale necessario al parsing dell'intero dataset. Tuttavia, tali operazioni sono state ritenute necessarie per garantire una migliore fruibilità dei dati. Infatti, senza questa trasformazione la maggior parte delle query finali con operatori di confronto applicate al graph database sarebbe risultata notevolmente limitata.

5.4.4 Data split

Va sottolineato che l'obiettivo principale dello script non è stato semplicemente quello di effettuare data cleaning e preparazione sui dati, ma ha incluso anche una logica per la suddivisione delle transazioni in base al tipo dell'account destinatario. Lo script è stato quindi progettato per suddividere e classificare le transazioni in base al tipo di account destinatario ed estrapolare in modo semplice le informazioni di transazioni, blocchi ed eventi. Questa classificazione è stata resa possibile grazie al trie degli address costruito a partire dal dataset stesso (vedi Sezione 5.3.1).

5.4.5 Conversione in CSV

Questa ultima fase è stata cruciale per garantire una facile ed efficiente importazione dei dati all'interno del graph database. Neo4j dispone infatti del tool *neo4j-admin import* ottimizzato per l'importazione di grandi quantità di dati a partire da file *CSV* (vedi Sezione 3.5.2). I file generati sono stati strutturati in modo da rappresentare le entità e le relazioni del modello di riferimento ed in particolare sono stati generati file per nodi e relazioni. Lo script dispone inoltre di una funzionalità che ha permesso di suddividere i file CSV in file più piccoli di uguale dimensione consentendo un'importazione più agevole e gestibile. Per ulteriori dettagli riguardo il tool *neo4j-admin import* si rimanda il lettore al Capitolo 3.

5.5 Importazione dei dati in Neo4j

L'importazione dei dati in Neo4j può essere effettuata sia con il comando Cypher *LOAD CSV* sia mediante il tool *neo4j-admin import*. Si rimanda il lettore alle Sezione 3.5 per maggiorni dettagli. Il Codice 17 mostra la sintassi del comando LOAD CSV, mentre il Codice 17 quello del comando *neo4j-admin import*. Entrambi ci hanno permesso di caricare gli stessi identici dati ma con tempi nettamente differenti. Il comando LOAD CSV processa le righe del file di input una alla volta, mentre il comando *neo4j-admin import* permette di scrivere in memoria di archiviazione l'intero file senza overhead dovuti al pre processing delle singole righe. I dataset utilizzati nella valutazione sperimentale (vedi Capitolo 6) sono stati importati utilizzando *neo4j-admin import*.

```
LOAD CSV WITH HEADERS FROM "file:///data.csv" AS transaction
CALL {
    WITH transaction

    MERGE (t:Transaction {hash: transaction.hash})
    ON CREATE
        SET t = properties(transaction)

    MERGE (fromNode: Account {address: transaction.fromAddress})
    ON CREATE
        SET fromNode.account_type = 1

    MERGE (toNode: Account {address: transaction.toAddress})
    ON CREATE
        SET toNode.account_type = 1

    MERGE (block: Block {hash: transaction.blockHash})

    CREATE (t)-[:CONTAINED]->(block)
    CREATE (t)<-[:SENT]-(fromNode)
    CREATE (t)-[:TRANSFERRED]->(toNode)

} IN TRANSACTIONS OF 10000 ROWS;
```

Codice 16: Import dati transazioni mediante il comando LOAD CSV di Cypher

```
bin/neo4j-admin database import full
--verbose
--delimiter=","
--array-delimiter=";"
--skip-duplicate-nodes
--skip-bad-relationships
--report-file=report-model1.txt
--nodes=Block=block_node_headers.csv,blocks.csv
--nodes=Account=account_node_headers.csv,account.csv
--nodes=Transaction=txs_node_headers.csv,txs.csv
--nodes=Log=log_node_headers.csv,log.csv
--relationships=SENT=sent_rel_headers.csv,sent.csv
--relationships=CREATED=creation_rel_headers.csv,creation.csv
--relationships=EMITTED=log_rel_headers.csv,emitted.csv
--relationships=INVOKED=invocation_rel_headers.csv,invocation.csv
--relationships=TRANSFERRED=transfer_rel_headers.csv,transfer.csv
--relationships=TO=unk_rel_headers.csv,unk.csv
--relationships=CONTAINED=contain_rel_headers.csv,contained.csv
--relationships=CHILD_OF=block_child_rel_headers.csv,child-of.csv
modelTN
```

Codice 17: Import dati transazioni su modello TN mediante comando neo4j-admin import

5.6 Indici Neo4j

Un indice è una struttura dati che migliora le prestazioni delle query e la velocità di recupero dei dati al costo di alcune scritture addizionali e di spazio aggiuntivo per il mantenimento della struttura dell'indice. Gli indici sono utilizzati per accedere velocemente alle informazioni senza dover controllare tutte le entità (righe per tabelle, nodi per grafi) per ogni query eseguita. Neo4j offre la possibilità di aggiungere indici sia sui nodi che sulle relazioni. Il Codice 18 mostra gli indici utilizzati per il modello TN. La keyword *CONSTRAINT* oltre che aggiungere un indice aggiunge anche un vinicolo di unicità. I vincoli sulle relazioni si sono resi necessari solamente per il modello TE, come mostrato nel Codice 19.

```
CREATE CONSTRAINT IF NOT EXISTS
    FOR (a: Account) REQUIRE a.address IS UNIQUE;

CREATE CONSTRAINT IF NOT EXISTS
    FOR (t:Transaction) REQUIRE t.hash IS UNIQUE;

CREATE CONSTRAINT IF NOT EXISTS
    FOR (block: Block) REQUIRE block.hash IS UNIQUE;

CREATE CONSTRAINT IF NOT EXISTS
    FOR (block: Block) REQUIRE block.number IS UNIQUE;

CREATE CONSTRAINT IF NOT EXISTS
    FOR (log: Log) REQUIRE log.hash IS UNIQUE;
```

Codice 18: Indici modello TN (transaction as nodes)

```
CREATE CONSTRAINT IF NOT EXISTS
    FOR (a: Account) REQUIRE a.address IS UNIQUE;

// Transaction hash index
CREATE INDEX transaction_rel_index IF NOT EXISTS
    FOR ()-[r: TO]-() ON (r.hash);

CREATE INDEX transaction_rel_index2 IF NOT EXISTS
    FOR ()-[r: CREATED]-() ON (r.hash);

CREATE INDEX transaction_rel_index3 IF NOT EXISTS
    FOR ()-[r: INVOKED]-() ON (r.hash);

CREATE INDEX transaction_rel_index4 IF NOT EXISTS
    FOR ()-[r: TRANSFERRED]-() ON (r.hash);

// Transaction block hash index
CREATE INDEX transaction_block_rel_index IF NOT EXISTS
    FOR ()-[r: TO]-() ON (r.block_hash);

CREATE INDEX transaction_block_rel_index IF NOT EXISTS
    FOR ()-[r: CREATED]-() ON (r.block_hash);

CREATE INDEX transaction_block_rel_index IF NOT EXISTS
    FOR ()-[r: INOVKED]-() ON (r.block_hash);

CREATE INDEX transaction_block_rel_index IF NOT EXISTS
    FOR ()-[r: TRANSFERRED]-() ON (r.block_hash);
```

Codice 19: Indici modello TE (transaction as edges)

Capitolo 6

Valutazione sperimentale

In questo capitolo, mostreremo i risultati delle analisi ottenuti nel corso della nostra ricerca. Inizieremo esaminando i risultati ottenuti dalla costruzione del trie degli address (vedi Sezione 5.3.1), quelli della classificazione delle transazioni e successivamente quelli relativi alla costruzione dei grafi proiettati. Infine, verranno mostrate le differenze tra i due modelli proposti (vedi Capitolo 4) in termini di spazio di archiviazione e rispetto all'efficienza di alcune query Cypher appositamente create per valutare le performance dei modelli in condizioni verosimili. Per concludere ci concentreremo su un caso d'uso specifico per valutare il **modello TN** e il **modello TE** su scenari reali.

6.1 Trie degli indirizzi

Come anticipato nella Sezione 5.3.1, la classificazione delle transazioni è stata possibile grazie al trie degli indirizzi costruito preventivamente a partire dal dataset stesso.

La costruzione del trie degli indirizzi è stata realizzata attraverso un processo graduale, che ha permesso di elaborare in modo progressivo la grande quantità di dati a nostra disposizione. In particolare il trie è stato costruito in modo incrementale, partendo dal primo file e aggiungendo successivamente i dati contenuti nel secondo, nel terzo e cosi via. Nella Tabella 6.1 sono indicati per ogni range di blocchi il tempo di elaborazione, la dimensione finale del trie e la percentuale di address classificati come sconosciuti.

Trie	Blocchi	Tempo elab.	Dim.	Ind. sconosciuti
A	0 - 999 999	57s	3.5	67.71 %
B	1 000 000 - 1 999 999	261	24	20.75 %
C	2 000 000 - 2 999 999	575	46	15.63 %
D	3 000 000 - 3 999 999	8 700	204	12.22 %
E	4 000 000 - 4 499 999	86 400	490	10.35 %

Tabella 6.1: Risultati costruzione trie degli indirizzi. Temp elaborazione in secondi, dimensione in Megabytes.

Si può notare che all'aumentare della dimensione del dataset preso in considerazione la percentuale di address sconosciuti cala notevolmente. Questo perché mano a mano che il dataset preso in considerazione aumenta, indirizzi inizialmente sconosciuti sono stati classificati grazie ad un loro maggior conivolgimento sulla rete.

6.2 Classificazione transazioni

In questa sezione, mostriamo i risultati ottenuti nella classificazione delle transazioni sfruttando il trie degli indirizzi.

Trie	Blocchi	Trie lookup	Tempo elab.	Tot. txs	Txs. sconosciute
A	0 - 999 999	2.80	155	1 674 262	6.63 %
B	0 - 999 999	2.84	158	1 674 262	2.25 %
C	0 - 999 999	3.14	164	1 674 262	1.92 %
D	0 - 999 999	3.4	203	1 674 262	1.62 %
A	1 000 000 - 1 999 999	12.83	491	6 383 128	56.21 %
B	1 000 000 - 1 999 999	13.0	501	6 383 128	5.12 %
C	1 000 000 - 1 999 999	14.11	507	6 383 128	3.25 %
D	1 000 000 - 1 999 999	14.69	546	6 383 128	2.05 %
A	2 000 000 - 2 999 999	13	563	7 305 457	67.46 %
B	2 000 000 - 2 999 999	14.63	578	7 305 457	32.48 %
C	2 000 000 - 2 999 999	15	576	7 305 457	4.03 %
D	2 000 000 - 2 999 999	15.12	616	7 305 457	1.79 %

Tabella 6.2: Risultati classificazione transazioni. Tempi di lookup ed elaborazione in secondi.

La Tabella 6.2 è suddivisa in 3 parti, ognuna delle quali contiene la classificazione di uno stesso range di blocchi per ogni trie degli indirizzi (vedi Tabella 6.1). Possiamo notare come per ogni range l'utilizzo di trie degli indirizzi via via più completi (A, B, C, D) permette di ottenere una classificazione sempre più precisa. Il trie degli indirizzi *D* ci ha consentito di classificare i primi 3 000 000 di blocchi con una percentuale di transazioni sconosciute molto bassa. Notiamo inoltre come i tempi di elaborazione aumentano notevolmente tra il range di blocchi [0, 999 999] e il range [1 000 000, 1 999 999], nonostante entrambi abbiano lo stesso numero di blocchi. Questo è dovuto al fatto che, come la colonna **Tot.txs** mostra, il numero di transazioni del secondo e terzo range è nettamente superiore. Il numero di transazioni per blocco è aumentato notevolmente nei primi periodi di popolarità di Ethereum andando poi ad aumentare progressivamente come mostrato in Figura 6.1.

Figura 6.1: Transazioni giornaliere Ethereum

6.3 Confronto modelli

Il confronto tra i modelli **TN** e **TE** introdotti nel Capitolo 4 è stata una parte fondamentale della nostra tesi. Come vedremo in dettaglio successivamente, nessuno dei due modelli si è dimostrato nettamente superiore all'altro in tutti i test eseguiti ma ognuno ha dimostrato delle prestazioni migliori in specifici scenari.

Il confronto tra i due modelli è stato effettuato tenendo in considerazione alcuni punti chiave:

- **Spazio** di archiviazione occupato per memorizzare gli stessi dati;
- **Tempo** esecuzione Cypher query generiche;

6.3.1 Spazio occupato

Abbiamo visto all'inizio del Capitolo 4 che la principale differenza tra il **modello TN** e il **modello TE** è il modo in cui le transazioni vengono rappresentate. Difatti, se nel primo modello le transazioni sono nodi del grafo, nel secondo sono rappresentate come archi.

Dal momento che le informazioni da memorizzare non riguardano solo le transazioni ma anche i blocchi, il **modello TE** presenta il grosso svantaggio di avere questi dati memorizzati negli stessi archi delle transazioni. Ricordiamo inoltre, come specificato nella Sezione 2.2.4, che un blocco è composto da un insieme di transazioni e di conseguenza nel **modello TE** i blocchi che ne contengono più di una sono memorizzati tante volte quante sono le transazioni nel blocco producendo cosi una duplicazione dei dati.

Nella Tabella 6.3, si evidenzia come questo impatti sulle dimensioni finali del database a parità di numero di blocchi memorizzati.

Range blocchi	**modello TN**	**modello TE**
0 - 2 999 999	25 GB	35 GB

Tabella 6.3: Spazio occupato dai due modelli per archiviare le stesse informazioni

Il database basato sul **modello TE** ha una dimensione nettamente superiore rispetto a quello del **modello TN** nonostante il numero "esiguo" di blocchi rispetto alla dimensione totale della blockchain Ethereum. Questo rappresenta un vincolo importante da tenere in considerazione se il sistema che si ha a disposizione dispone di risorse limitate.

6.3.2 Cypher query

In questa sezione verranno illustrati i tempi di elaborazione di alcune query Cypher eseguite su ciascun modello di grafo. Questo tipo di query è stato il principale strumento utilizzato per interrogare e analizzare i dati in Neo4j e ci ha fornito una buona indicazione delle prestazioni complessive dei due modelli. Si ricorda che il codice delle query eseguite per entrambi i modelli è consultabile al seguente link https://github.com/daniel-97/computer-science-thesis.

I test sono stati eseguiti su due dataset distinti:

- **Dataset 1** - Da blocco 0 (30 Luglio 2015) a blocco 180 998 (3 Settembre 2015), con un totale di 100 000 transazioni;
- **Dataset 2** - Da blocco 0 (30 Luglio 2015) a blocco 2 999 999 (15 gennaio 2017), per un totale di 15 362 847 transazioni;

Query generiche

Le query mostrate in Tabella 6.4 sono query generiche appositamente ideate per evidenziare le differenze iniziali tra i due modelli. Mirano principalmente a fornire una prima analisi preliminare e comparativa delle prestazioni dei due modelli di grafo proposti.

		Dataset 1		Dataset 2	
#	**Descrizione**	**modello TN**	**modello TE**	**modello TN**	**modello TE**
Q1	Conta tutte le transazioni	2	3	1	2
Q2	Conta le transazioni che hanno trasferito 0 ether (ETH)	61	108	32 947	952 051
Q3	Trova tutte le transazioni di uno specifico indirizzo	42	60	53	67
Q4	Conta il numero di account che hanno effettuato transazioni verso se stessi	119	50	119 255	52 759
Q5	Trova tutti gli account che non hanno mai effettuato transazioni (solo destinatari)	16	12	-	-
Q6	Media transazioni effettuate dagli account	96	55	56 347	10 095
Q7	Contare il numero di transazioni che hanno emesso uno specifico evento	13	1 306	7 716	80 574
Q8	Trova tutte le transazioni contenute in uno specifico blocco	6	123	20	1 369 829
Q9	Media numero transazioni per blocco	47	153	2 122	215 895
Q10	Conta il numero di transazioni dirette verso l'indirizzo nullo	1	1	55	42
Q11	Numero di transazioni per ogni blocco	1 643	851	-	-

Tabella 6.4: Tempi di esecuzione delle query generiche. Tutti i tempi riportati sono in millisecondi (ms). Le celle marcate con "-" indicano che la query non è terminata in tempi ragionevoli.

Possiamo notare come le query generiche eseguite sul database basato sul **modello TN** sono mediamente più veloci rispetto a quelle eseguite sul **modello TE**, con una differenza più marcata sul **dataset 2**. In particolare si può osservare che:

- **Q1, Q10** - Entrambe query semplici e non impegnative, nel modello TN si contano i nodi *transaction*, nel modello TE si contano le relazioni. Non si apprezza alcuna differenza degna di nota.
- **Q2** - Differenze leggere sul dataset 1, più marcate sul dataset 2. In questo caso il valore di ether trasferito si trova sui nodi per il modello TN, sugli archi per il modello TE. Dal momento che non è possibile in Cypher scrivere una query che parta direttamente da un arco ma è sempre necessario partire da un nodo, questo implica che nel modello TE si accede sempre a un nodo e a un arco mentre nel modello TN si accede direttamente al nodo.
- **Q3, Q5** - Tempi molto simili, nessuna differenza sostanziale.
- **Q4** - Query nettamente più lenta sul modello TN. Come illustra la Figura 6.2 il numero di hop necessario sul modello TN è doppio rispetto a quello del modello TE per poter andare da un nodo account all'altro. Nel caso delle transazioni verso se stessi nel modello TN sono necessari 2 hop mentre nel modello TE ne basta 1. Questo spiega la lentezza del modello TN.
- **Q6** - Database basato su modello TN decisamente più lento di quello basato su modello TE. In questo tipo di query si devono contare le relazioni del tipo

  ```
  (:Account)-[:SENT]-(:Transaction)-[:TRANSFERRED]- (:Account)
  ```

  nel modello TN, mentre nel modello TE si contano quelle

  ```
  (a:Account)-[:TRANSFERRED]- (:Account)
  ```

  È dunque ovvio che il numero di hop sia più elevato nel primo caso.
- **Q7** - In questo caso il modo in cui i log sono stati codificati nel modello TE (vedi Sezione 5.2) ha un impatto notevole sulle prestazioni. La ricerca degli eventi nel modello TN viene fatta su singoli nodi mentre nel modello TE sono tutti codificati all'interno di speciali array negli archi. Questo necessita di una query nettamente più complessa del modello TN.

- **Q8, Q9** - In queste query si nota uno dei più grandi svantaggi del modello TE ovvero la mancanza dei nodi di tipo *Block*. Sul modello TN è sufficiente utilizzare la relazione *CONTAINED* mentre sul modello TE è necessario raggruppare per il campo *blockHash* sulla relazione, operazione nettamente più onerosa.

- **Q11** - Il modello TE risulta più veloce del 50% rispetto al modello TN. Riconducibile al fatto che sul modello TE si contano solamente gli archi raggruppando per *blockHash*, mentre nel modello TN è necessario fare un hop in più per prendere le informazioni dei blocchi.

Query di esplorazione dei vicini

Le query elencate in Tabella 6.5 sono finalizzate al confronto dei due modelli con query che esplorano i nodi adiacenti alle transazioni e agli account. Questo tipo di analisi è molto comune e utile all'interno della rete Ethereum perché tipicamente consente di rilevare i comportamenti degli utenti, identificare le transazioni più influenti o i contratti più utilizzati. Le query che coinvolgono nodi vicini sono generalmente più onerose in termini computazionali rispetto ad altre query. Questo perché richiedono l'esplorazione delle relazioni adiacenti con conseguente analisi di porzioni più ampie del grafo.

		Dataset 1		Dataset 2	
#	**Descrizione**	**modello TN**	**modello TE**	**modello TN**	**modello TE**
Q12	Media vicini *externally owned account*	81	24	58 401	5 396
Q13	Conta i vicini di secondo grado dei primi 100 account con più transazioni effettuate	10 389	2 845	-	-
Q14	Conta i vicini di terzo grado dell'account con più transazioni verso *EOA*	417 310	105 504	-	-
Q15	Conta gli account vicini di terzo grado di uno specifico account	5 516	1 631	-	-
Q16	Conta gli account vicini di quarto grado di uno specifico account	613 127	149 181	-	-
Q17	Ritorna tutte le transazioni vicine di una specifica transazione	67	5836	96	-
Q18	Ritorna tutte le transazioni vicine di tutte le transazioni effettuate da uno specifico account	-	17 475	-	-
Q19	Conta tutte le transazioni vicine dei primi 10 account con più transazioni effettuate	1 189	131	119 603	18 169
Q20	Ritorna tutte le transazioni nello stesso blocco di una transazione specifica	2	242	-	-

Tabella 6.5: Tempi di esecuzione query per analisi nodi vicini. Tutti i tempi riportati sono in millisecondi (ms). Le celle marcate con "-" indicano che la query non è terminata in tempi ragionevoli.

Dalle query elencate nella Tabella 6.5, notiamo come il **modello TE** sia mediamente superiore al **modello TN**. In linea generale per ottenere gli stessi dati, il modello TE richiede di attraversare la metà degli archi rispetto al modello TN come illustrato nella Figura 6.2 e questo è di rilevanza centrale per query che riguardano l'esplorazione di nodi vicini. Notiamo anche come la maggior parte delle query non abbia valori riportati per il dataset 2. Questo è stato causato da problemi di memoria insufficiente o tempi di attesa eccessivamente lunghi per l'esecuzione delle query.

Di seguito riportiamo, come fatto per le query generiche, le osservazioni sui risultati ottenuti.

- **Q12, Q18** - Modello TN nettamente svantaggiato, come visto in casi precedenti simili a questo si deve attraversare il doppio degli archi nel **modello TN** rispetto al **modello TE**. Nel caso della query Q18 questo svantaggio è talmente elevato che la query eseguita sul dataset 1 non è terminata in tempi ragionevoli.

- **Q13, Q14, Q15, Q16, Q19** - Rispecchiano tutti i risultati ottenuti dalle query precedenti. Il modello TE risulta superiore in tutti i test di analisi di nodi vicini grazie alla struttura semplificata e al numero ridotto di hop necessari per raggiungere gli stessi nodi del modello TN. Anche in questo caso la Figura 6.2 illustra adeguatamente questo concetto.

- **Q17** - Modello TN più veloce di due ordini di grandezza. Nel modello TN si parte dal nodo transazione con l'hash specifico, mentre nel modello TE si parte da un generico nodo account e si cerca la relazione che tra tutte ha l'hash indicato poiché non è possibile in Neo4j scrivere una query a partire da una relazione.

- **Q20** - Modello TN più veloce di due ordini di grandezza. Accesso alle informazioni dei blocchi più rapido nel modello TN grazie agli appositi nodi di tipo *block*.

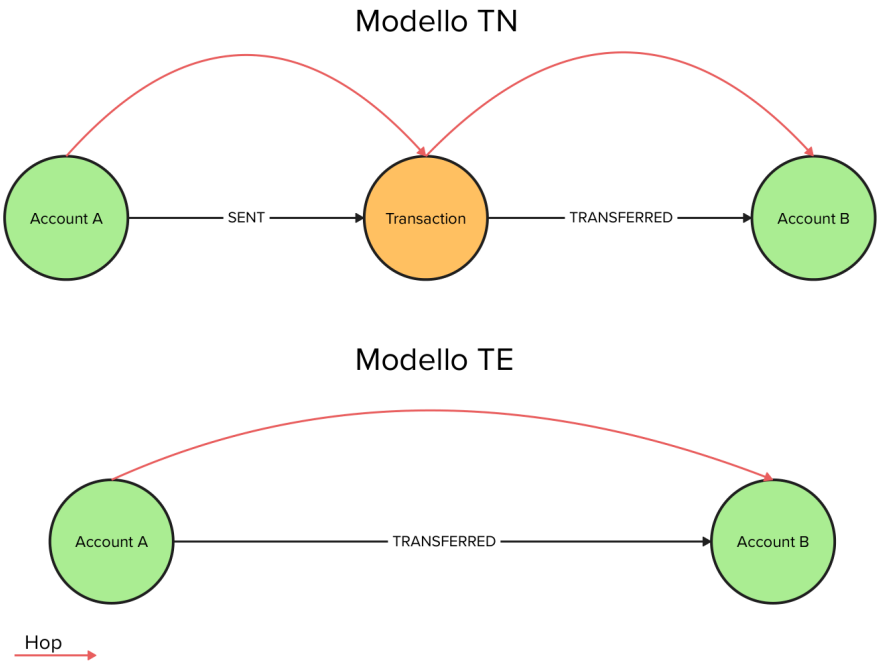

Figura 6.2: Differenza in numero di hop tra modello TN e modello TE

Il Codice 20 mostra la differenza tra la query Q19 per il modello TN e il modello TE. È chiaramente visibile che il numero di hop necessario per ottenere gli stessi dati è superiore nel modello TE (righe 3,14).

```
// MODEL TN
MATCH
    (n:Account)-[r:SENT]-(t:Transaction)-[:TRANSFERRED]->(b:Account)
WHERE n <> b
WITH n, COUNT(r) AS numTransactions
ORDER BY numTransactions DESC
LIMIT 10
MATCH (b)-[]->(t1: Transaction)
RETURN
    COUNT(r1);

// MODEL TE
MATCH
    (n:Account)-[r:TRANSFERRED]->(b:Account)
WHERE
    n <> b
WITH n, COUNT(r) AS numTransactions
ORDER BY numTransactions DESC
LIMIT 10
MATCH (b)-[r2]->(c)
RETURN
    COUNT(r2);
```

Codice 20: Query Q19: in alto modello TN, in basso modello TE

6.3.3 Grafi proiettati

Come anticipato nella Sezione 3.6, Neo4j offre la possibilità di creare grafi proiettati, a partire dal grafo completo, sui quali eseguire algoritmi con la libreria GDS. I grafi proiettati possono essere costruiti in due modi distinti:

- **Native projection**;
- **Cypher projection**;

Ai fini della nostra ricerca è stata utilizzata la *Cypher projection*, più lenta della *Native projection* ma più flessibile. Si rimanda il lettore alla documentazione ufficiale Neo4j per maggiori dettagli sui tipi di proiezione [27]. La Tabella 6.6 evidenzia le differenze nella costruzione dei grafi proiettati a partire dai grafi del **modello TN** e **modello TE**. Tutti i grafi proiettati ottenuti sono mono-partiti, quindi hanno sempre e solo un tipo di nodo e un tipo di arco. I test di costruzione dei grafi proiettati sono stati effettuati sul dal dataset 1.

	Dataset 1	
Descrizione	**modello TN (ms)**	**modello TE (ms)**
Costruisce un grafo proiettato di tutte le relazioni tra account	67 639	11 128
Costruisce un grafo proiettato che rappresenta solo i trasferimenti di ether tra account	73 090	13 428
Costruisce un grafo proiettato che rappresenta solo le invocazioni di smart contract	3 945	1 815
Costruisce un grafo proiettato che rappresenta solo le creazioni di smart contract	371	170

Tabella 6.6: Tempi costruzione grafi proiettati su dataset 1

Come possiamo notare, il **modello TE** risulta nettamente superiore al **modello TN**. Infatti questo è dovuto al numero inferiore di nodi che bisogna attraversare per ottenere lo stesso grafo proiettato.

6.4 Caso d'uso: The DAO

Per il confronto delle potenzialità e dei punti deboli dei due modelli introdotti nel Capitolo 4 è di fondamentale importanza individuare un caso d'uso significativo.
Per motivi pratici e a causa delle limitate risorse disponibili abbiamo focalizzato la nostra analisi sui fenomeni verificatosi nei primi blocchi della storia di Ethereum. Basti pensare che ad oggi (inizio 2024) la dimensione totale della blockchain Ethereum dalla genesi è pari a **970 GB** [28].
I fenomeni più importanti avvenuti nei primi 5 milioni di blocchi sono 2:

- ***The DAO*** - Prima organizzazione autonoma decentralizzata di Etehreum;
- **CriptoKitties** - Primo gioco decentralizzato realizzato su Ethereum, ebbe molta popolarità agli inizi del 2018;

In questa sezione focalizzeremo la nostra attenzione sul fenomeno dell'attacco al contratto *The DAO*, che portò ad uno dei più grandi cambiamenti nella storia di Ethereum: la prima hard fork. Il cambiamento fu proposto e

successivamente approvato per "annullare" l'attacco al contratto perpetrato ai danni della rete ritornando i fondi investiti dagli utenti [29]. Non tutti furono d'accordo con questa decisione, che causò la divisione della blockchain in due: Ethereum ed Ethereum Classic. Si rimanda il lettore alla Sezione 6.4.4 per ulteriori dettagli.

6.4.1 DAO: finanza distribuita

Un'organizzazione autonoma decentralizzata o **DAO** in breve, è una forma di organizzazione basata su blockchain governata da cripto token. Chiunque possieda e detenga token guadagna la possibilità di votare su questioni direttamente collegate alla *DAO* [30]. Su Ethereum queste organizzazioni sono implementate mediante il supporto di *smart contract* (vedi Sezione 2.3).

Le fasi di vita di un'organizzazione decentralizzata [31] sono riassunte nel seguente elenco:

1. Scrittura del codice del contratto necessario per il corretto funzionamento del meccanismo DAO;

2. Inizio campagna di raccolta fondi a tempo limitato durante la quale gli utenti aggiungono fondi alla DAO ricevendo in cambio token (ICO[1]);

3. Una volta terminato il periodo di raccolta fondi, la DAO può iniziare ad operare;

4. Come ultima fase, gli utenti possono fare proposte alla DAO su come spendere il denaro e i membri possono votare le proposte fatte;

6.4.2 "The DAO"

The DAO è il nome di un particolare DAO ideato e implementato dal team della startup Slock.it diventato in breve tempo una delle organizzazioni autonome decentralizzate digitali di maggior successo sulla rete Ethereum. Fu lanciato il **30 Aprile 2016** con una finestra di raccolta fondi di 28 giorni sotto forma di *smart contract* e con lo scopo di fornire un nuovo modello di business decentralizzato. La campagna iniziale prevedeva di raccogliere tra i 5 e i 10 milioni di dollari dagli utenti ma per il **15 Maggio 2016** aveva raccolto più di 100 milioni. Infine la campagna terminò con più di 150 milioni raccolti da circa 11000 membri: molto al di là delle più ottimistiche previsioni dei suoi creatori [31].

[1]Una inital coin offering (ICO) è un operazione finalizzata alla raccolta fondi necessaria per un progetto imprenditoriale.

La Figura 6.3 mostra il numero di transazioni dirette verso *The DAO* prima e dopo il periodo di crowdfunding. Si possono notare 3 picchi spiegati di seguito.

1. **13 Maggio 2016** - Periodo centrale di raccolta fondi del contratto *The DAO*, momento di piena interazione da parte degli utenti;
2. **28 Maggio 2016** - Ultimo giorno di raccolta fondi, possibile corsa dell'ultimo minuto per depositare fondi sul contratto;
3. **18 Giugno 2016** - Data inizio presunto attacco vero il contratto *The DAO* (vedi Sezione 6.4.3);

Il Codice 21 contiene la query eseguita sul database del **modello TN** utilizzata per ottenere i dati del grafico mostrato in Figura 6.3.

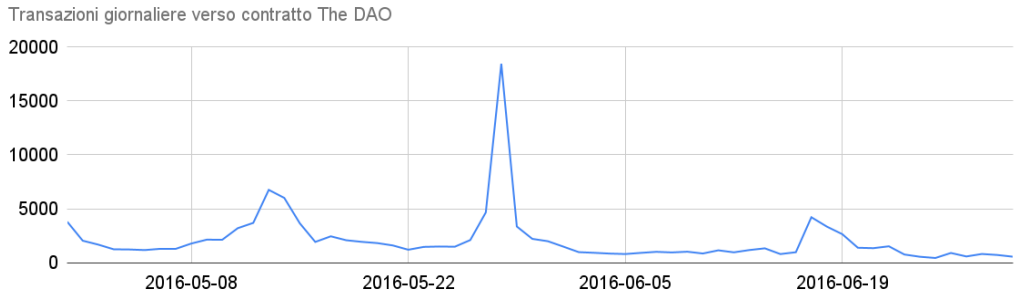

Figura 6.3: Transazioni verso *The DAO*

```
MATCH
    (block: Block)<-[]-(txs: Transaction)-[]->
    (b: Account {address: "0xbb9bc244d798123fde783fcc1c72d3bb8c189413"})
WHERE
    block.number >= 1400000 AND block.number <= 2500000
WITH
    datetime({epochSeconds:block.timestamp}) as datetime,
    count(txs) as tot_txs
RETURN
    date(datetime) as date,
    SUM(tot_txs) as day_txs
ORDER BY date ASC;
```

Codice 21: Query Cypher modello TN per dati Figura 6.3

Durante il periodo di raccolta fondi, in molti espressero la loro preoccupazione in merito alla sicurezza dello *smart contract The DAO*.

Terminata la raccolta fondi, la comunità discusse molto sulle necessità di sistemare le vulnerabilità prima dell'inizio della fase di proposta e il 12 Giugno 2016 fu annunciata da **Stephan Tual**, uno dei creatori del *The DAO*, la scoperta di un "recursive call bug" nel codice dello smart contract che avrebbe permesso ad un eventuale attaccante di prelevare fondi dal contratto in modo illecito (si rimanda il lettore alla Sezione 6.4.3). Fu tuttavia assicurato che nessun fondo era a rischio.

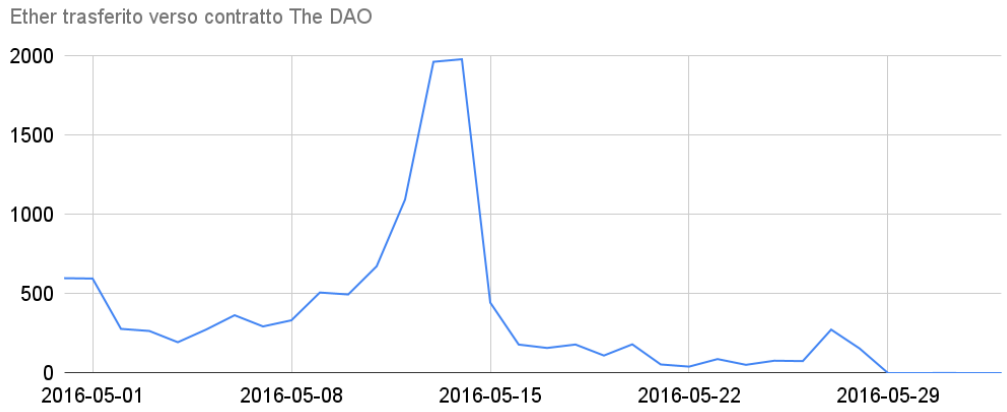

Figura 6.4: Ether trasferito verso *The DAO*

```
MATCH
    (block: Block)<-[]-(txs: Transaction)-[]->
    (b: Account {address: "0xbb9bc244d798123fde783fcc1c72d3bb8c189413"})
WHERE
    block.number >= 1400000 AND block.number <=2000000
WITH
    datetime({epochSeconds:block.timestamp}) as datetime,
    txs.value as eth
RETURN
    date(datetime) as date,
    SUM(eth) as tot_eth
ORDER BY date ASC
```

Codice 22: Query Cypher modello TN per dati Figura 6.4

La Figura 6.4 illustra i trasferimenti di ether nel tempo verso il contratto The DAO e il Codice 22 la relativa query Cypher usata per ottenere i dati.

Il picco avvenuto tra il 13 e il 14 Maggio 2016 chiaramente distinguibile corrisponde al periodo di piena attività di crowdfunding del contratto come precedentemente illustrato nella Figura 6.3.

#	**Nome**	**Indirizzo**	**Eth. trasf.**	**Tot. txs**
A1	Poloniex	0xdf21fa922215b1a56f5a6d6294e6e36c85a0acfb	0.00	25787
A2	Bittrex	0xfbb1b73c4f0bda4f67dca266ce6ef42f520fbb98	206.87	13522
A3	Poloniex	0x32be343b94f860124dc4fee278fdcbd38c102d88	871.51	8107
A4	Kraken 2	0x0a869d79a7052c7f1b55a8ebabbea3420f0d1e13	0.00	1975
A5	Shape shift 2	0x9e6316f44baeeee5d41a1070516cc5fa47baf227	48.22	1401
A6	Possibile attaccante	0xf0e42abda410cefb5b4dc4de92a3de5b309e02f2	0.00	1004
A7	Yunbi 1	0xd94c9ff168dc6aebf9b6cc86deff54f3fb0afc33	0.02	943
A8	Gatecoin	0x40b9b889a21ff1534d018d71dc406122ebcf3f5a	194.19	844
A9	Sconosciuto	0x69ea6b31ef305d6b99bb2d4c9d99456fa108b02a	14.04	828
A10	Sconosciuto	0x946c555081313c5e0986c6cd5f6978257a406237	0.00	658

Tabella 6.7: Classifica degli indirizzi con più transazioni effettuate verso *The DAO*

La Tabella 6.7 illustra i primi dieci indirizzi ordinati per numero di transazioni verso *The DAO* e per ognuno di essi il dettaglio del totale di ether trasferiti. Si può notare come alcuni indirizzi abbiano effettuato un numero elevato di transazioni verso il contratto ma come abbiano trasferito esigue quantità di ether. Per gli indirizzi di famosi exchange come Poloniex (A1), Kraken 2 (A4) e Yunbi 1 (A7) questo può essere giustificato con la chiamata di particolari funzioni del contratto che non implicano necessariamente il trasferimento di ether. Gli indirizzi A6 e A10 mostrano invece un comportamento sospetto essendo account di utenti "normali" e non famosi exchange riconosciuti da tutta la comunità. In particolare, in [32] l'indirizzo A6 viene ricondotto ad uno dei possibili utenti ritenuti responsabili dell'attacco eseguito ai danni del contatto *The DAO* descritto nella Sezione 6.4.3.

6.4.3 Attacco a "The DAO"

Nonostante le rassicurazioni in merito al bug scoperto nel codice dello smart contract, un hacker sconosciuto iniziò ad utilizzare la vulnerabilità per prelevare fondi da *The DAO* dentro un contratto figlio. Il **18 Giugno 2016** l'attaccante riuscì a prelevare più di 3.5 milioni di ether dal contratto.

In risposta all'attacco, molti si adoperarono per limitare il danno: ci si accorse fin da subito che il contratto creato dell'attaccante soffriva dello stesso bug del *The DAO* stesso, per cui squadre di "white hat"[2] si adoperarono per sfruttare la stessa vulnerabilità contro l'attaccante. Nonostante le contromisure adoperate, vennero comunque prelevati in modo illegittimo 1/3 dei fondi totali, fatto che contribuì alla fine di questo ambizioso progetto.

Reentrancy attack

Il tipo di attacco usato verso il contratto *The DAO* sfruttato dall'attaccante è chiamato *reentrancy attack*. Questo tipo di attacco è realizzabile a causa di vulnerabilità del contratto dovute al modo in cui viene strutturato il codice in Solidity (vedi Sezione 2.3). In particolare, il *reentrancy attack* sfrutta il comportamento di particolari funzioni chiamate funzioni di "fallback". Queste speciali funzioni sono "unamed" (senza nome) e vengono eseguite in alcune situazioni specifiche:

- Nessuna delle funzioni del contratto corrisponde a quella specificata dal chiamante del contratto;
- Non sono stati forniti dati con la chiamata della funzione (*calldata*[3]);

6.4.4 Ethereum hard fork

L'attaco a *The DAO* portò ad una delle più controverse decisioni della storia di Ethereum: una hard fork coordinata, chiamata anche *cambiamento di stato irregolare*. La fork permise di annullare l'attacco verso *The DAO* mediante la riscrittura della blockchain Ethereum cosicché i partecipanti poterono riavere indietro i fondi inizialmente depositati.

Sia prima che dopo la hard fork, ci furono degli importanti dibattiti riguardo alla immutabilità delle blockchian. In molti ritenevano che la fork avrebbe creato un precedente nella storia di Ethereum che avrebbe portato ad azioni simili nel futuro, andando contro i principi fondamentali delle blockchain. Fortunatamente nessun episodio simile si è mai ripetuto.

[2]Un "white hat" è un esperto di sicurezza informatica in grado di "bucare" sistemi altrui al fine di aiutare i proprietari a rendersi conto del problema di sicurezza. Si contrappongono ai "black hat" che violano illegalmente sistemi informatici per scopi personali o di lucro.

[3]I *calldata* sono informazioni all'interno di una transazione che contengono gli argomenti della funzione da chiamare. Sulla base di tali informazioni la rete capisce quale funzione dello *smart contract* invocare.

Nonostante le controversie, in un post [33] di **Jeffrey Wilcke** pubblicato sul blog ufficiale della **Ethereum Foundation**[4] il 15 Luglio 2016 gli utenti decisero a favore della hard fork e, il 20 Luglio 2016 fu ufficialmente comunicato in un post [34] di **Vitalik Buterin**[5] il completamento della hard fork al blocco 1 980 000.

La parte di comunità che non ritenne corretta la modifica della blockchain non aderì alla hard fork e continuò a sostenere e sviluppare la blockchain originale. Nacque cosi **Ethereum Classic**.

La Figura 6.5, illustra gli ether trasferiti durante il periodo della fork. Si possono notare due picchi in corrispondenza del 20 Luglio e 26 Luglio 2016, periodo immediatamente successivo alla hard fork. È possibile che l'evento abbia scosso la comunità portando gli utenti ha trasferire ether verso gli exchange al fine di mitigare i rischi e adeguarsi alle nuove dinamiche del mercato.

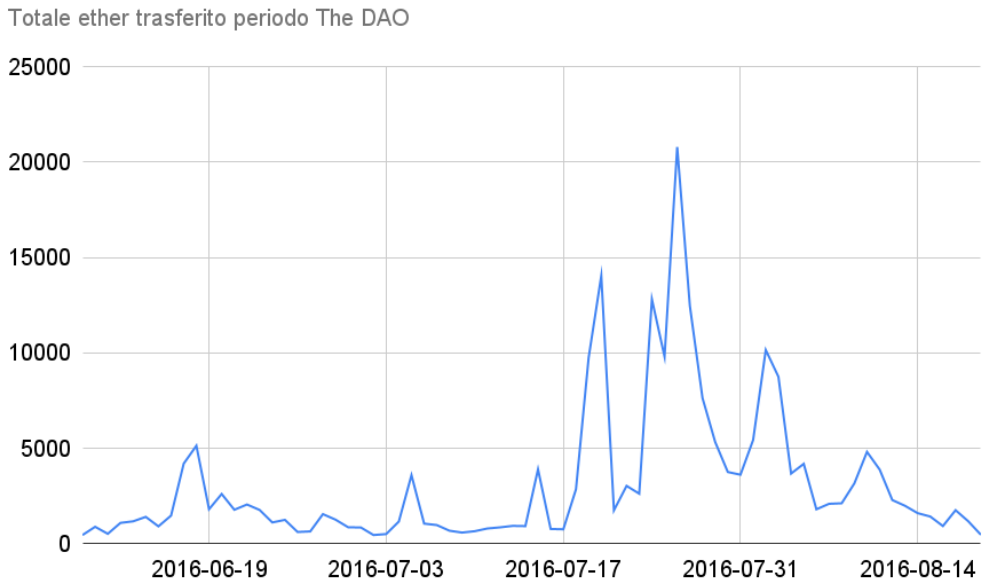

Figura 6.5: Ether trasferito durante periodo del contratto The DAO

[4]La Ethereum Foundation è un organizzazione no-profit dedita al supporto di Ethereum e delle tecnologie correlate.

[5]Vitalik Buterin è un programmatore e fondatore di Ethereum.

6.5 Analisi attacco

Al fine di esaminare l'evento in modo approfondito sono state condotte due macro analisi dell'attacco a *The DAO*: nella prima abbiamo concentrato la nostra attenzione principalmente sulle transazioni direttamente coinvolte con lo *smart contract* del *The DAO* mentre in un secondo momento ci siamo concentrati sull'impatto dell'attacco su tutta la rete Ethereum.

Al fine di dare al lettore una maggiore chiarezza rispetto agli eventi che si sono susseguiti, di seguito riportiamo le date degli avvenimenti più significativi:

- **30 Aprile 2016** - Lancio *The DAO*;
- **28 Maggio 2016** - Fine periodo crowdfunding;
- **12 Giugno 2016** - Scoperta prima vulnerabilità del contratto;
- **18 Giugno 2016** - Reentry attack;
- **20 Luglio 2016** - Hard fork completata;

La Tabella 6.8 mostra i tempi di elaborazione di alcune query appositamente create per studiare il fenomeno *The DAO*. Tutti i test sono stati effettuati sul dataset 2 (vedi Sezione 6.3.2), comprensivo dei blocchi [0, 2 999 999].

#	**Descrizione**	**modello TN**	**modello TE**
Q21	Trova tutti gli account che hanno fatto transazioni verso *The DAO*	13 834	13 468
Q22	Conta tutte le transazioni dirette verso *The DAO*	3	3
Q23	Trova gli indirizzi che hanno effettuato transazioni verso *The DAO* ordinati in modo decrescente per numero di transazioni	21 588	15 285
Q24	Trova gli indirizzi che hanno effettuato transazioni verso *The DAO* ordinati in modo decrescente per ether inviati	21 315	14 652
Q25	Ritorna il numero di transazioni effettuate verso *The DAO* raggruppate per giorno	700	400
Q26	Ritorna il numero di ether inviati a *The DAO* raggruppati per giorno	12 857	5 417
Q27	Ritorna per i primi 100 account che hanno effettuato più transazioni verso *The DAO*, gli indirizzi più comuni tra i mittenti e la percentuale di "comunanza" tra di essi	21 417	14 076
Q28	Ritorna il numero totale di transazioni dirette verso *eoa* ed *sc* effettuate sulla rete nel periodo del *The DAO*	17 608	545 053
Q29	Numero di contratti creati nel periodo del *The DAO*	1 342	80

Tabella 6.8: Tempi di esecuzione delle query per il recupero delle informazioni legate a The DAO. Tutti i tempi riportati sono in millisecondi (ms).

Di seguito le osservazioni sui risultati ottenuti delle query mostra in Tabella 6.8.

- **Q21, Q22, Q25** - Query molto simili, tempi comparabili. Nessuna differenza degna di nota.
- **Q23, Q24, Q27** - Query molto simili tra di loro. Richiedono tutte di effettuare un numero di hop simile a quello rappresentato in Figura 6.2. È immediato capire perché il modello TE prevale: il numero di hop è ridotto rispetto al modello TN.
- **Q26, Q29** - Query simili. Entrambe necessitano delle informazioni dei blocchi per poter accedere al timestamp. Nel modello TE le informa-

zioni del blocco sono direttamente accessibili dalla relazione e non c'è nessun bisogno di saltare verso il nodo blocco corrispondente. Questo spiega il tempo maggiore del modello TN.

- **Q28** - Netta differenza tra modello TN e modello TE: unica differenza tra le due query è la posizione del dato *blockHash*. Sul modello TN, Neo4j filtra preventivamente i blocchi del periodo interessato e successivamente raggruppa, mentre nel modello TE prima raggruppa e successivamente filtra, operazione molto più costosa.

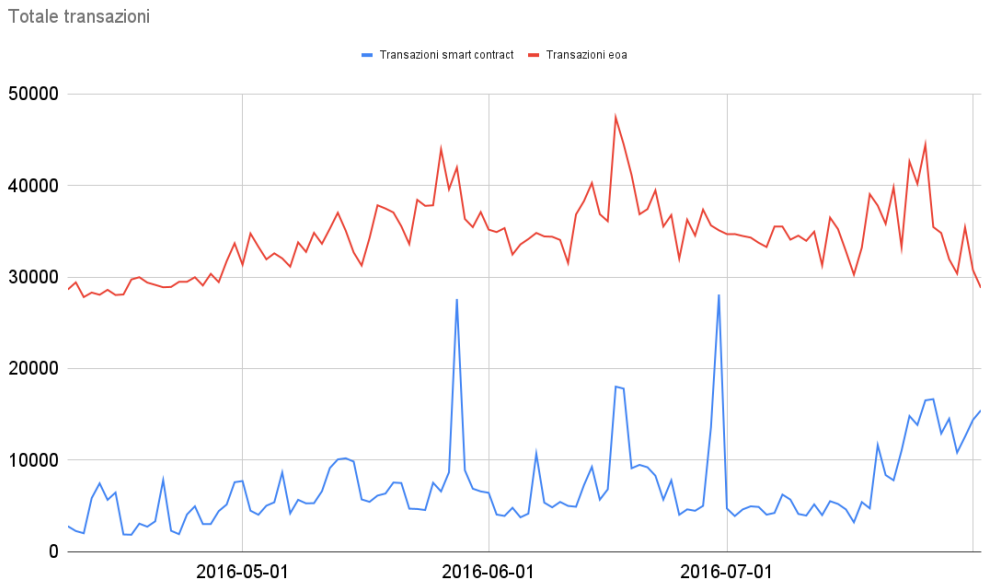

Figura 6.6: Transazioni periodo The DAO

La Figura 6.6 illustra le transazioni avvenute nel periodo del *The DAO*. In particolare la linea blu mostra le transazioni effettuate verso *smart contract*, mentre la rossa rappresenta le transazioni effettuate verso *eoa*. Si notano immediatamente i due picchi nel grafico delle transazioni dirette verso *smart contract*: il primo avvenuto in data 28 Maggio 2016 corrisponde alla fine del periodo di corwdfunding del *The DAO*, mentre il secondo avvenuto in data 30 Giugno 2016 corrisponde ad un picco di chiamate avvenute verso il contratto EtherID, un nuovo servizio che iniziò ad essere popolare in quel periodo.

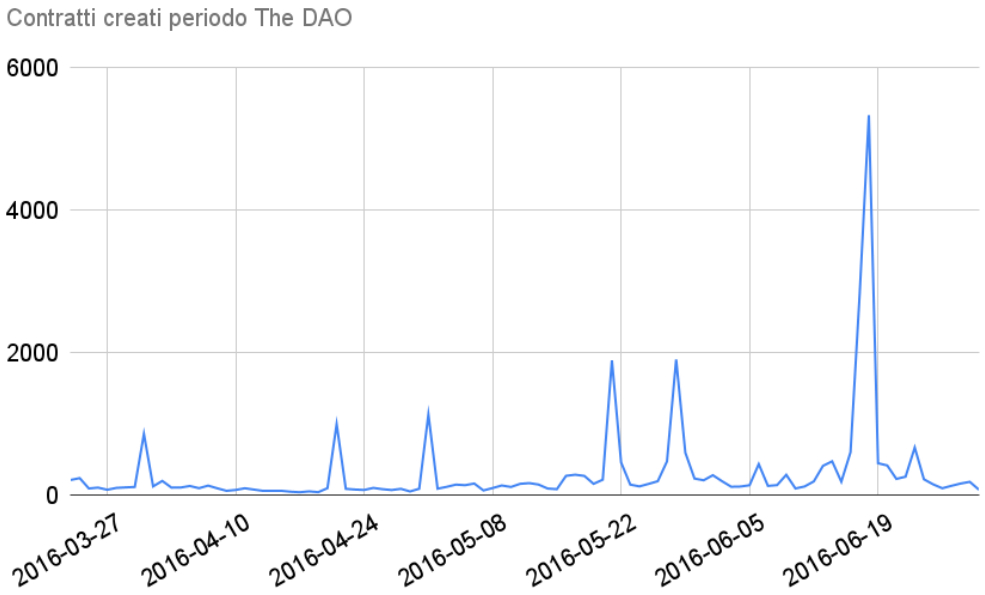

Figura 6.7: Contratti creati durante il periodo del The DAO

Il grafico mostrato in Figura 6.7 illustra le transazioni che hanno causato la creazione *smart contract* durante il periodo del The DAO. I dati sono stati ottenuti mediante l'esecuzione della query Q29. Il picco più alto del giorno 18 Luglio 2016 corrisponde al giorno dell'attacco al contratto *The DAO*.

Capitolo 7

Related Work

Nel presente capitolo descriveremo il lavoro svolto per selezionare il graph database più adatto alle nostre esigenze nell'abito delle analisi delle transazioni e faremo una breve panoramica sugli studi attualmente esistenti che riguardano l'analisi della blockchain Ethereum.

Esamineremo alcuni dei principali graph database disponibili sul mercato, confrontando le loro caratteristiche principali ed eventuali limitazioni. Infine, giustificheremo la nostra scelta finale di adottare il database Neo4j (vedi Capitolo 3).

7.1 Database considerati

La scelta del database adatto agli scopi della nostra tesi è stato uno dei primi aspetti sui quali è stato necessario dedicare particolare attenzione durante le prime fasi del nostro studio. I database presi in considerazione sono:

- **TitanDB**;
- **TigerGraph**;
- **Neo4j**;

Ognuno dei database elencati presenta caratteristiche uniche che lo rendono particolarmente adatto per determinati casi d'uso. Principalmente abbiamo valutato gli aspetti più rilevanti per il nostro caso tra cui le performance, la semplicità dei tool, il supporto della comunità e la disponibilità di strumenti per l'importazione di grandi quantità di dati a partire da file JSON.

7.1.1 TitanDB

TitanDB è uno dei più famosi graph database distribuiti. Completamente open source e disponibile sotto licenza Apache2, è stato progettato per gestire grandi quantità di dati su cluster di server. La sua architettura distribuita consente di scalare orizzontalmente aggiungendo nuovi nodi cluster per gestire eventuali aumenti di carichi senza compromettere le prestazioni del sistema. Consente l'interazione contemporanea di migliaia di utenti sulla stessa base di dati. Supporta diversi backend di archiviazione (Apache Cassandra, Apache HBase o Oracle Barkeley DB) e ha un'integrazione nativa nello stack TinkerPop, un framework per la manipolazione di grafi che fornisce un linguaggio di interrogazione chiamato **Gremlin**. Lo stack Tinkerpop viene utilizzato per fornire librerie e interfacce standard per diversi graph database tra cui Titan e Neo4j. Titan implementa l'intero stack Tinkerpop composto da:

- **Blueprints** - Insieme di interfacce e implementazioni per graph database, simile a **JDBC**;
- **Pipes** - Un framework per l'elaborazione del flusso di dati;
- **Gremlin** - Un linguaggio di attraversamento dei grafi;
- **Frames** - Un mappatore di grafi ad oggetti;
- **Furnace** - Algoritmi di grafi standard per grafi non orientati e diretti;
- **Rexster** - Un server che espone graph database tramite REST API;

Query language

Gremlin è il linguaggio di attraversamento di grafi di Apache TinkerPop. È un linguaggio funzionale che consente agli utenti di esprimere attraversamenti complessi di grafi. Ogni attraversamento di Gremlin è composto da una sequenza di passaggi (operazione atomica sui dati) potenzialmente nidificati. Il Codice 23 illustra una semplice query Gremlin che esegue i seguenti passaggi:

1. Prendi il nodo di nome "gremlin" ed etichettalo "a";
2. Ottieni i progetti creati da "gremlin" e poi chi li ha creati...;
3. ...ad esclusione di "gremlin" stesso;
4. Raggruppa e conta in base al titolo;

```
g.V().has("name","gremlin").as("a").
  out("created").in("created").
    where(neq("a")).
  groupCount().by("title")
```

Codice 23: Esempio query Gremlin

Gli attraversamenti Gremlin possono essere scritti in modo imperativo (procedurale), dichiarativo (descrittivo) oppure in modo ibrido. Un attraversamento imperativo dice come procedere in ogni fase della traversata mentre un attraversamento dichiarativo non dice esplicitamente l'ordine in cui eseguire l'attraversamento ma consente di selezionale un modello di attraversamento in ciascuna fase. L'attraversamento dichiarativo ha il vantaggio aggiuntivo di sfruttare non solo un pianificatore di query in fase di compilazione ma anche di un pianificatore di query a runtime che sceglie quale modello di attraversamento usare in base alla statistiche storiche di ciascun modello.

La Figura 7.1 illustra l'attraversamento effettuato da Gremlin nell'eseguire una query che ritorna i nomi di tutte le persone che "marko" conosce.

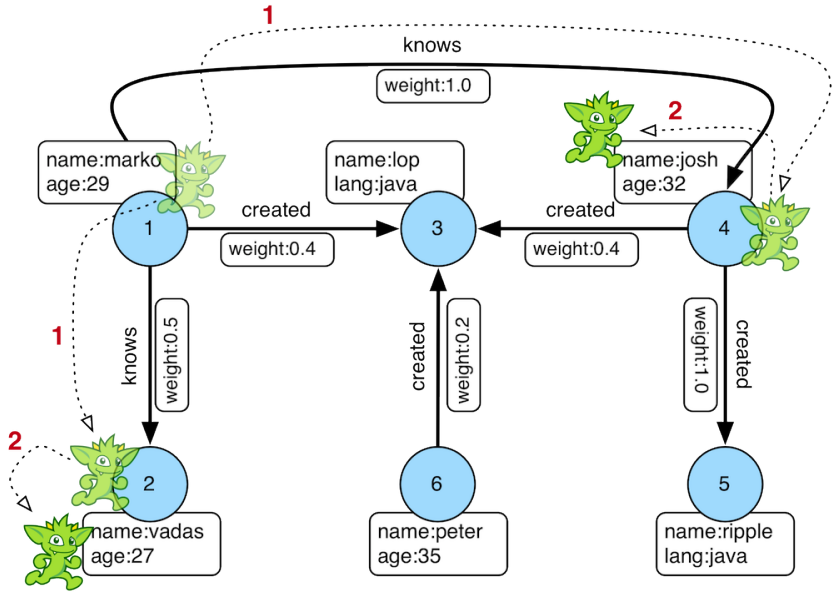

Figura 7.1: Esempio attraversamento nodi di Gremlin

Nella suite di TitanDB sono inoltre disponibili i tool **Fanus** (graph analytics engine) e **Fulgora** (fast graph processor).

7.1.2 TigerGraph

TigerGraph è un graph database open source rilasciato nel 2017 scritto in C++. È un sistema principalmente sviluppato per poter eseguire calcoli simultanei paralleli ed è il primo e unico *Native Parallel Graph* (NPG). Supporta l'esecuzione di carichi di lavoro su grandi grafi con miliardi di nodi e fino a 1 trilione di relazioni. Supporta l'esecuzione di query native parallele, consentendo l'esplorazione rapida e approfondita dei dati archiviati. Tra le principali funzionalità chiave di TigerGraph troviamo:

- Capacità di caricare grandi quantità di dati: permette di importare da 50 a 150 GB di dati all'ora per macchina;
- Rapida esecuzione di algoritmi paralleli su grafo: è in grado di attraversare centinaia di milioni di nodi/archi al secondo per macchina;
- Aggiornamento e inserimento di dati in tempo reale tramite interfaccia REST;
- Capacità di unificare l'analisi in tempo reale con l'elaborazione dei dati offline su larga scala;

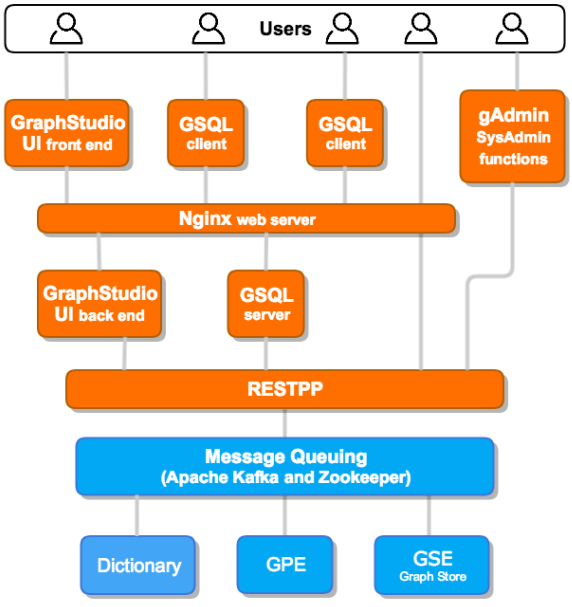

Figura 7.2: Struttura interna TigerGraph [35]

La Figura 7.2 illustra una panoramica della piattaforma TigerGraph con i suoi componenti principali. Si può chiaramente notare come gli utenti possono scegliere vari modi per interagire con il sistema: **GSQL client** permette di collegasi a istanze di Tiger su server remoti, **GraphStudio** è una semplice e intuitiva interfaccia grafica adatta agli utenti inesperti che supporta la maggior parte dei comandi base GSQL, **REST API** permette ad applicazioni aziendali di massimizzare la loro efficienza comunicando direttamente con RESTPP e infine **gAdmin** viene utilizzata per l'amministrazione del sistema.

Query language

Il query language usato da TigerGraph è chiamato **GSQL** che, come suggerisce il nome, è una diretta estensione del linguaggio **SQL** che ha una curva di apprendimento molto bassa per i nuovi sviluppatori. Contrariamente ad altri query language, è un linguaggio Turing-completo il che implica che è possibile esprimere qualsiasi query utilizzando il linguaggio GSQL.

Una query GSQL è una sequenza di istruzioni di recupero e calcolo dei dati eseguite come una singola operazione. Nonostante la somiglianza con SQL, una distinzione importante tra i due linguaggi è che una query GSQL è una sequenza di istruzioni definite come una singola operazione, mentre una query SQL tende ad essere una singola istruzione che viene eseguita direttamente su un database.

```
1  USE GRAPH Social
2
3  SET syntax_version="v2"
4
5  CREATE QUERY hello (VERTEX<Person> p) {
6    start = {p};
7    result = SELECT tgt
8             FROM start:s -(Friendship:e)- Person:tgt;
9    PRINT result;
10 }
```

Codice 24: Esempio query GSQL

Il Codice 24 mostra la sintassi di un semplice comando GSQL che definisce la query *hello*. A partire dal nodo *p* fornito come parametro alla query la clausola *SELECT* descrive un attraversamento di 1 salto secondo il modello descritto nella clausola *FROM*: *start:s -(Friendship:e)- Person:tgt*. Il modello significa che stiamo selezionando tutti i nodi che hanno una relazione *Frindship* con il nodo di partenza.

7.1.3 Neo4j

Dopo un'attenta analisi dei graph database inizialmente proposti tra TitanDB, TigerGraph e Neo4j, abbiamo concluso che quest'ultimo sarebbe stata la scelta migliore per il nostro progetto di analisi e visualizzazione delle transazioni Ethereum. Come ampiamente discusso nel Capitolo 3, Neo4j dispone della maggior parte delle funzionalità da noi richieste ed ha ottenuto i risultati migliori nei primi test eseguiti. Nonostante le funzionalità di scalabilità di TitanDB e quelle di calcolo parallelo di TigerGraph abbiamo ritenuto Neo4j più indicato per i nostri scopi accademici. In aggiunta l'articolo [36] ci ha fornito una dettagliata analisi delle principali differenze tra Neo4j e TitanDB che può essere riassunta come segue:

In our analysis, Neo4j was the graph database with the best performance across all metrics.

La semplicità di utilizzo del linguaggio di interrogazione ha giocato Un altro elemento tenuto in considerazione è stata la semplicità di utilizzo del linguaggio di interrogazione. Tra Cypher, Gremlin e GSQL, il query languge Cypher di Neo4j è stato ritenuto il più semplice ed intuitivo da utilizzare (vedi Sezione 3.4).

Riassumendo, le caratteristiche che hanno giocato un ruolo cruciale nella scelta del database sono state:

- **Facilità d'uso** - Neo4j offre molti strumenti semplici da utilizzare sia da CLI che da interfaccia Web. Inoltre il query language Cypher semplifica lo sviluppo di query strutturate e complesse;
- **Performance** - Dai nostri personali test, Neo4j ha sempre dimostrato delle performance eccezionali su database di grandi dimensioni;
- **Libreria GDS** - La libreria Graph Data Science di Neo4j offre algoritmi avanzati per l'analisi dei grafi, utilizzabili "out of the box";
- **Supporto community e documentazione** - Essendo uno dei più famosi ed utilizzati graph database, Neo4j vanta una vasta community di utenti attivi e di una documentazione completa e ben strutturata;

7.2 Analisi della blockchain Ethereum

Nel Capitolo 4 abbiamo introdotto il *grafo delle transazioni* (vedi Sezione 4.1) sul quale abbiamo basato i nostri due modelli utilizzati per analizzare

la blockchain. Il lavoro svolto in questa tesi va tuttavia ritenuto un'aggiunta ai numerosi studi già esistenti sull'analisi delle transazioni e in generale sulla blockchain Ethereum. In questa sezione esploreremo brevemente la letteratura esistente sull'argomento e faremo un rapido confronto con il lavoro svolto nella nostra tesi.

Come abbiamo menzionato più volte, le attività di maggior rilevanza su Ethereum sono tre: trasferimento di ether, creazione di contratti e invocazione di contratti. In questo contesto, numerosi studi hanno già esplorato e proposto modelli specifici per rappresentare queste attività con grafi ad hoc. Nell'articolo [37] vengono proposti tre modelli di grafo: **MFG** (money flow graph), **CCG** (contract creation graph) e **CIG** (contract invocation graph) usati per calcolare *degree distribution*, *degree correlation*, *clusters* e simili delle attività sopra menzionate. Questi 3 modelli hanno delle strutture fondamentalmente diverse tra di loro e non sono interscambiabili. Ogni tipo di grafo è appositamente pensato per studiare una sola delle attività principali di Ethereum.

L'articolo [38] introduce un modello molto simile al nostro **modello TN** con l'aggiunta delle *transazioni interne* (vedi Sezione 2.2.6). Il modello presentato è stato utilizzato per l'analisi del **Gatecoin hack**, un attacco verificatosi a Maggio 2016 che causò una perdita totale di 185 000 ether.

Un' altro articolo che vale menzionare è [39]: il quale esamina la natura evolutiva di vari reti presenti su Ethereum da un punto di vista temporale. Viene studiato il tasso di crescita e il modello di quattro reti distinte, la loro durata e il tasso di aggiornamento dei nodi. Infine in [40] viene condotta un'indagine approfondita sulle relazioni delle transazioni Ethereum. Questo studio rappresenta infatti il primo tentativo conosciuto di analizzare i dati delle transazioni registrate sulla blockchain e di esaminare le loro leggi statistiche da un punto di vista di scienza delle reti.

I modelli proposti negli articoli precedenti sono stati progettati e ottimizzati per studi specifici di eventi e fenomeni avvenuti su Ethereum. I modelli da noi proposti si distinguono invece per la loro natura generica e la loro adattabilità ad una vasta gamma di analisi e casi d'uso. Il loro scopo, è principalmente quello di mettere a disposizione le informazioni della blockchain Ethereum in un formato il più comune e standard possibile per facilitare l'analisi e la comprensione della rete. I nostri modelli sono stati intenzionalmente creati con un approccio generico, privo di vincoli legati a specifici casi d'uso.

Capitolo 8

Conclusioni

Il lavoro svolto durante questa tesi ha permesso di comprendere le differenze dei due modelli di grafo per la rappresentazione dei dati della blockchain Ethereum. Partendo dai dati a disposizione è stato possibile progettare i modelli e successivamente metterli a confronto su scenari reali utilizzando il graph database Neo4j.

L'importazione dei dati all'interno del database è stata una delle sfide più impegnative di questa tesi. Se inizialmente le soluzioni adottate non hanno presentato grosse limitazioni nei primi test effettuati con piccole porzioni di dati, con l'aumentare dei dati presi in considerazione si è reso necessario ottimizzare i processi di importazione e classificazione degli indirizzi a causa dell'aumento esponenziale dei tempi di attesa.

Durante il processo di importazione è stata dedicata particolare attenzione all'utilizzo degli strumenti adatti per la manipolazione di grandi quantità di dati in Neo4j come il tool *neo4j-admin import* mostrato nel Capitolo 3, che ha permesso di ridurre sensibilmente i tempi di import. Questo ha richiesto la conversione dei dati dal formato JSON al formato CSV, che ha inevitabilmente portato a problemi di compatibilità di tipi composti (vedi Sezione 5.2).

Durante la fase di classificazione degli indirizzi (vedi Sezione 5.3) è stata fondamentale la scelta di un trie come struttura dati di supporto (vedi Sezione 5.3.1) che ha reso possibile in fase di classificazione delle transazioni avere dei tempi di lookup trascurabili.

Concludiamo riassumendo le principali differenze dei due modelli e le raccomandazioni relative al loro utilizzo. Il primo fattore da tenere in considerazione durante la scelta del modello è lo spazio di archiviazione a disposizione. Abbiamo visto come le strutture interne dei due modelli influiscano profondamente sulla dimensione finale del database (vedi Sezione 6.3.1). In particolare se si desidera un modello compatto ed efficiente da un punto di

vista dello spazio conviene utilizzare sempre il **modello TN** a causa delle informazioni ridondate dei blocchi del **modello TE**.

Dai test effettuatati è inoltre emerso che nessuno dei due modelli è nettamente superiore all'altro, ma ognuno ha prestazioni migliori rispetto all'altro su determinati casi d'uso. Abbiamo verificato nella Sezione 6.3.2 come il **modello TN** sia preferibile quando si desidera focalizzarsi maggiormente sulle singole transazioni e sulle loro proprietà senza dover esplorare in dettaglio i collegamenti con altri nodi. D'altra parte il **modello TE** è più adatto nel caso sia necessario analizzare relazioni con nodi vicini come dimostrato dai test effettuati. Infine, è opportuno menzionare che l'ulteriore analisi effettuata sul caso d'uso relativo al contratto *The DAO* ha permesso di evidenziare come i nostri modelli supportino analisi di scenari reali. Entrambi i modelli hanno permesso di accedere a tutte le informazioni necessarie per analizzare il fenomeno fornendo una visione dettagliata sulle transazioni coinvolte e delle interazioni avvenute sulla rete nel periodo dell'attacco.

8.1 Lavori futuri

Nella presente tesi sono stati mostrati e analizzati solamente due modelli di grafo per la rappresentazione dei dati della blockchain Ethereum. È tuttavia possibile continuare con l'esplorazione di altri modelli oltre a quelli proposti. Ulteriori studi potrebbero essere condotti per valutare le prestazioni dei modelli proposti su dataset di dimensioni maggiori o addirittura completi di tutte le transazioni Ethereum e in contesti applicativi diversi. Potrebbe essere utile integrare nuove forme di dati esterne oltre a quelle utilizzate. Abbiamo già discusso nella Sezione 6.2 come l'utilizzo delle sole euristiche per la classificazione non permette di classificare correttamente tutti gli indirizzi del dataset. Uno sviluppo futuro potrebbe introdurre l'utilizzo di un nodo locale Ethereum per una classificazione completa e precisa. Inoltre, una delle più grandi limitazioni del dataset utilizzato è quella della mancanza delle transazioni interne (vedi Sez. 2.2.6) la cui presenza andrebbe indubbiamente a migliorare la qualità dei dati.

Un'area decisamente interessante per i lavori futuri riguarda l'ottimizzazione delle query Cypher necessarie per l'estrazione di informazioni utili dal grafo delle transazioni. Sarebbe estremamente interessante esplorare varie tecniche e query al fine di ottimizzare le prestazioni, specialmente su dataset di grandi dimensioni. Infine, un aspetto non esplorato in questa tesi ma sicuramente degno di nota è l'impiego della libreria GDS di Neo4j per l'analisi del grafo delle transazioni mediante algoritmi specializzati.

Ringraziamenti

Il lavoro svolto in questa tesi non sarebbe stato possibile senza l'aiuto dei miei relatori. Ringrazio la prof.ssa Laura Emilia Maria Ricci, il dott. Damiano Di Francesco Maesa e il dott. Matteo Loporchio che mi hanno seguito passo passo durante la stesura e dato preziosi consigli per migliorare l'elaborato finale. Un ringraziamento speciale va ai miei genitori, che mi sono stati vicini durante tutto il mio percorso formativo, incoraggiandomi nei momenti più difficili e supportandomi in tutte le mie scelte. Desidero ringraziare anche mio fratello Samuele, la cui presenza mi ha aiutato a distrarmi dopo lunghe giornate di studio e intenso lavoro. La tua compagnia è stata molto importante e lo sarà sempre, sono profondamente grato di averti come fratello. Un sincero ringraziamento va a Giada per avermi dato la motivazione per continuare, sei stata fondamentale per il completamento dei miei studi sopratutto nei momenti in cui la mia determinazione è venuta meno, non ci sarei mai riuscito senza di te. Un ringraziamento va inoltre a tutti i miei amici per il loro sostegno ed incoraggiamento, in particolare desidero ringraziare il mio collega Luca, il quale mi ha offerto preziosi consigli durante l'ultimo periodo del mio percorso formativo. Infine vorrei ringraziare coloro che non hanno mai creduto in me. Mi avete dato la forza per dimostrarvi che vi sbagliavate.

Per aspera ad astra!

Bibliografia

[1] Satoshi Nakamoto. «Bitcoin: A Peer-to-Peer Electronic Cash System». In: (2008). URL: https://www.bitcoin.org/bitcoin.pdf.

[2] Vitalik Buterin. «Ethereum: A Next-Generation Smart Contract and Decentralized Application Platform». In: (2014). URL: https://ethereum.org/content/whitepaper/whitepaper-pdf/Ethereum_Whitepaper_-_Buterin_2014.pdf.

[3] *Introduction*. URL: https://neo4j.com/docs/cypher-manual/current/introduction/.

[4] Wenbo Wang et al. «A Survey on Consensus Mechanisms and Mining Strategy Management in Blockchain Networks». In: *IEEE Access* 7 (2019), pp. 22328–22370. DOI: 10.1109/ACCESS.2019.2896108. URL: https://doi.org/10.1109/ACCESS.2019.2896108.

[5] *Blockchain is a promising technology, but bottlenecks and complex challenges lie ahead*. URL: https://joint-research-centre.ec.europa.eu/jrc-mission-statement-work-programme/facts4eufuture/blockchain-now-and-tomorrow-assessing-multidimensional-impacts-distributed-ledger-technologies/blockchain-promising-technology-bottlenecks-and-complex-challenges-lie-ahead_en.

[6] Tharaka Mawanane Hewa, Mika Ylianttila e Madhusanka Liyanage. «Survey on blockchain based smart contracts: Applications, opportunities and challenges». In: *J. Netw. Comput. Appl.* 177 (2021), p. 102857. DOI: 10.1016/J.JNCA.2020.102857. URL: https://doi.org/10.1016/j.jnca.2020.102857.

[7] *Ethereum's energy expenditure*. 2023. URL: https://ethereum.org/en/energy-consumption/.

[8] Guido Bertoni et al. *Keccak and the SHA-3 Standardization*. 2013. URL: https://csrc.nist.gov/csrc/media/projects/hash-functions/documents/keccak-slides-at-nist.pdf.

[9] nhsz. *Blocks*. 2023. URL: https://ethereum.org/en/developers/docs/blocks/.

[10] *Merkle Patricia Trie*. URL: https://ethereum.org/en/developers/docs/data-structures-and-encoding/patricia-merkle-trie/.

[11] *Creating an Efficient Trie Data Structure with Python*. URL: https://www.askpython.com/python/examples/trie-data-structure.

[12] *Trie*. URL: https://en.wikipedia.org/wiki/Trie.

[13] *Merkle tree*. URL: https://it.wikipedia.org/wiki/Albero_di_Merkle.

[14] *Ethereum State Trie Architecture Explained*. URL: https://medium.com/@eiki1212/ethereum-state-trie-architecture-explained-a30237009d4e.

[15] URL: https://static.nicehash.com/marketing%2FJohannes%20Larsson%20-%203.png.

[16] *Ethereum under the hood*. URL: https://medium.com/coinmonks/ethereum-under-the-hood-part-7-blocks-c8a5f57cc356.

[17] *Gas*. URL: https://ethereum.org/en/developers/docs/gas/.

[18] Nick Szabo. *Smart Contracts*. 1994. URL: https://www.fon.hum.uva.nl/rob/Courses/InformationInSpeech/CDROM/Literature/LOTwinterschool2006/szabo.best.vwh.net/smart.contracts.html.

[19] Nick Szabo. «Smart Contracts: Building Blocks for Digital Markets». In: (1996). URL: https://www.fon.hum.uva.nl/rob/Courses/InformationInSpeech/CDROM/Literature/LOTw%20interschool2006/szabo.best.vwh.net/smart_contracts_2.html.

[20] *Logging data from smart contracts with events*. URL: https://ethereum.org/en/developers/tutorials/logging-events-smart-contracts/.

[21] Maciej Besta et al. «Demystifying Graph Databases: Analysis and Taxonomy of Data Organization, System Designs, and Graph Queries». In: *ACM Comput. Surv.* 56.2 (2024), 31:1–31:40.

[22] *What is a graph database?* URL: https://neo4j.com/docs/getting-started/get-started-with-neo4j/graph-database/.

[23] *Graph database concepts*. URL: https://neo4j.com/docs/getting-started/appendix/graphdb-concepts/.

[24] *Importing CSV data into Neo4j*. URL: https://neo4j.com/docs/getting-started/data-import/csv-import/.

[25] Matteo Loporchio et al. «Analysis and Characterization of ERC-20 Token Network Topologies». In: *International Conference on Complex Networks and Their Applications*. Springer. 2023, pp. 344–355.

[26] *Transaction types*. URL: https://docs.infura.io/api/networks/ethereum/concepts/transaction-types.

[27] *Creating graphs*. URL: https://neo4j.com/docs/graph-data-science/current/management-ops/graph-creation/.

[28] *Ethereum blockchain size chart*. URL: https://blockchair.com/ethereum/charts/blockchain-size.

[29] *What Was The DAO?* 2023. URL: https://www.gemini.com/cryptopedia/the-dao-hack-makerdao.

[30] Alyssa Hertig. *What is a DAO?* 2023. URL: https://www.coindesk.com/learn/what-is-a-dao/.

[31] David Siegel. *Understanding The DAO Attack*. 2023. URL: https://www.coindesk.com/learn/understanding-the-dao-attack/.

[32] Ayden Férdeline. *The cryptopians: idealism, greed, lies, and the making of the first big cryptocurrency craze: by Laura Shin, New York City, Public Affairs, 2022, 497 pp., ISBN 9781541763012*. 2022.

[33] Jeffrey Wilcke. «To fork or not to fork». In: (2016). URL: https://blog.ethereum.org/2016/07/15/to-fork-or-not-to-fork.

[34] Vitalik Buterin. *Hard Fork Completed*. 2016. URL: https://blog.ethereum.org/2016/07/20/hard-fork-completed.

[35] *Internal Architecture*. URL: https://docs.tigergraph.com/tigergraph-server/current/intro/internal-architecture.

[36] Jéssica Monteiro, Filipe Sá e Jorge Bernardino. «Experimental evaluation of graph databases: Janusgraph, nebula graph, neo4j, and tigergraph». In: *Applied Sciences* 13.9 (2023), p. 5770.

[37] Ting Chen et al. «Understanding Ethereum via Graph Analysis». In: *ACM Trans. Internet Techn.* 20.2 (2020), 18:1–18:32.

[38] Wren Chan e Aspen Olmsted. «Ethereum transaction graph analysis». In: *12th International Conference for Internet Technology and Secured Transactions, ICITST 2017, Cambridge, United Kingdom, December 11-14, 2017*. IEEE, 2017, pp. 498–500.

[39] Lin Zhao et al. ≪Temporal Analysis of the Entire Ethereum Blockchain Network≫. In: *Proceedings of the Web Conference 2021*. WWW '21. Ljubljana, Slovenia: Association for Computing Machinery, 2021, pp. 2258–2269. ISBN: 9781450383127. URL: https://doi.org/10.1145/3442381.3449916.

[40] Dongchao Guo, Jiaqing Dong e Kai Wang. ≪Graph structure and statistical properties of Ethereum transaction relationships≫. In: *Information Sciences* 492 (2019), pp. 58–71. ISSN: 0020-0255. URL: https://www.sciencedirect.com/science/article/pii/S0020025519303159.