Università degli Studi di Verona

FACOLTÀ DI SCIENZE E INGEGNERIA
Corso di Laurea in Informatica

Tesi di laurea

EtherContracts2DB: Un framework per la memorizzazione e l'analisi degli smart contract di Ethereum tramite un database relazionale

Candidato:
Alberto baroni
Matricola VR421809

Relatore:
DR. Sara Milgiorini

Anno Accademico 2019-2020

Sommario

La tesi ha come obiettivo quello di creare un framework che permetta di memorizzare su un database relazionale parte della blockchain della piattaforma Ethereum. La trasposizione su database permetterà di fare interrogazioni a scopo statistico in maniera più semplice e rapida rispetto alla blockchain integrale, struttura di memorizzazione progettata apposta per essere molto compatta e quindi complessa da interrogare. Come primo passo verrà fatta un'introduzione alla tecnologia blockchain implementata su Ethereum, elencandone le proprietà principali. Un' analsi più approfondita della blockchain e dell'andamento della criptovaluta utilizzata su Ethereum è stata già svolta nella tesi di un collega dell'Università di Verona. Si parlerà poi degli Smart Contracts, programmi salvati sulla blockchain che possono essere eseguiti da utenti o da altri contratti. Gli Smart Contracts sono la caratterstica principale che contraddistingue Ethereum da Bitcoin, inoltre grazie a questi contratti digitali è possibile utilizzare delle applicazioni direttamente sulla blockchain: le DAPPs. Il database verrà introdotto da un modello entità relazione che permetterà una comprensione semplice della sua struttura, poi saranno elencati i singoli campi di ogni tabella nel dettaglio. Il framework è stato scritto completamente in linguaggio Python e, per lo sviluppo, sono stati utilizzati programmi e librerie presenti sul web che permettessero l'interazione con la blockchain di Ethereum. Infine sono state svolte delle analisi di carattere statistico sugli Smart Contracts e presentati dei grafici, raccogliendo le informazioni dal database e da fonti esterne.

Indice

1 Ethereum **1**
- 1.1 Introduzione . . . 1
- 1.2 Blockchain . . . 3
- 1.3 Gas . . . 6
- 1.4 Token . . . 7

2 Smart Contracts **9**
- 2.1 Cosa Sono . . . 9
- 2.2 Oracoli . . . 11

3 Database Relazionale **13**
- 3.1 Modello ER . . . 13
- 3.2 Campi database . . . 14
- 3.3 Schema Relazionale . . . 17

4 Framework **18**
- 4.1 Librerie software utilizzato per il progetto . . . 18
  - 4.1.1 scartate . . . 18
  - 4.1.2 utlizzate . . . 19
- 4.2 Funzionamento del programma . . . 19
  - 4.2.1 database.py . . . 19
  - 4.2.2 organize.py . . . 24
  - 4.2.3 sql_helper.py . . . 24

5 Analisi Contratti **28**
- 5.1 Statistiche generali . . . 28
- 5.2 Query sul database . . . 30

6 Conclusioni **34**

Capitolo 1

Ethereum

1.1 Introduzione

Ethereum è una piattaforma open-source che permette di gestire lo scambio diretto o indiretto di una criptovaluta chiamata **ether**. Ethereum implementa una struttura dati molto particolare detta **blockchain**. Questa struttura come suggerisce la parola è composta da una serie di blocchi collegati tra loro, ogni blocco contiene dei dati che possono riguardare diversi campi, nel caso di Ethereum ogni blocco contiene informazioni che riguardano le transazioni tra utenti, ovvero lo scambio di Ether o token. Altra caratteristica della teconlogia blockchain è l'immutabilità, una volta che un blocco viene salvato questo non potrà subire modifiche da nessuno.

Una delle caratteristiche più importanti di Ethereum è quella di essere una piattaforma completamente decentralizzata, non vi è un unico grande server che contiene i dati ma ogni nodo della rete contiene una copia della blockchain, questo implica che tutti i nodi debbano essere d'accordo e possedere una blockchain uguale. Nel caso in cui un nodo proponga una copia diversa (magari con scopo malevolo) questo verrà automaticamente escluso dal sistema perchè desincronizzato rispetto agli altri. Cercare di hackerare Ethereum è quasi impossibile. Per come sono organizzati i server l'attacco dovrebbe infettare il 51% dei nodi della rete, nodi che sono distribuiti su tutto il globo. Un sistema decentralizzato non è soggetto alla censura delle varie nazioni visto che non c'è, come invece succede per Instagram, Facebook, Google etc. , un server centralizzato che gestisce il servizio. Anche la potenza di calcolo non è una caratteristica che passa in secondo piano, visto che la rete di Ethereum è composta da migliaia di computer garentendo una potenza computazionale elevatissima della rete. Gli utenti possono essere persone vere con un proprio account, che hanno convertito dei soldi in criptovaluta da usare sulla piattaforma, oppure degli **Smart Contracts**. Ogni utente è identificato da un indirizzo univoco. Gli Smart Contracts sono dei programmi eseguiti tramite Ethereum Virtual Machine(EVM), questi

programmi vengono scritti da software developers e pubblicati sulla blockchain. Gli Smart Contracts possono ricevere e scambiare denaro a seconda di come sono scritti, questo permette lo scambio di valuta tra utenti che non si conoscono e quindi che chiaramente "non si fidano" (sistema trustless) l'uno dell'altro, lo Smart Contract fa quindi da intermediario per rendere uno scambio di denaro sicuro. Un esempio di Smart Contract può essere quello di una scommessa su una corsa di cavalli, ogni utente fa la sua puntata e la invia al contratto, poi a corsa finita lo Smart Contract va a consegnare i soldi al vincitore della scommessa.

La differenza principale tra Ethereum e Bitcoin è lo scopo del sistema: Bitcoin viene utilizzato principalmente per scambiare valuta digitale e la sua blockchain ha lo scopo di fare da registro per questi pagamenti, mentre Ethereum permette di creare software decentralizzato che verrà eseguito sulla blockchain stessa, il programma viene pubblicato e tutti lo eseguono e ne sono a conoscenza, non ci sono clausole nascoste o poco chiare che potrebbero ingannare gli utilizzatori. In più siccome le applicazioni sono eseguite su una rete di computer la possibilità che queste vadano offline o vengono hackerate è pressoché nulla. Questa possibilità può rappresentare un'innovazione che, in futuro, potrà cambiare il mondo della finanza. Per questo molte aziende nel campo come Microsoft, Intel, BP e UBS hanno iniziato ad investire milioni sullo sviluppo di applicazioni per Ethereum. Un'altra differenza importante tra Bitcoin ed Ethereum è il tempo che intercorre tra la scoperta di nuovi blocchi. Su Ethereum tra un blocco e l'altro passano poche decine di secondi mentre su Bitcoin circa 10 minuti.

Per diventare degli acquirenti su Ethereum è necessario aprire un portafoglio. Un portafoglio funziona un po' come un conto corrente online, l'utente ha una sua password per accedere ai suoi fondi e delle frasi di recupero in caso di smarrimento della password.
Ogni portafoglio è caratterizzato da:

- Indirizzo Ethereum: indirizzo hash che identifica il nostro portafoglio, gli altri utenti lo useranno per mandarci ether. Questo indirizzo è pubblico, tutti quelli che fanno parte della rete, e non, saranno a conoscenza del saldo e delle transazioni avvenute su tale portafoglio

- Chiave privata: password utilizzata per garantire di essere i proprietari di un portafoglio e trasferire ether

- Client software: è necessario un software per accedere al proprio portafoglio e comunicare con Ethereum. Questo software può essere un'app (come il famoso metamask) o un programma per computer

Come sistema di sicurezza informatica per gli utenti Ethereum utilizza la crittografia a chiave pubblica e privata per generare e gestire un indirizzo. Gli indirizzi vengono generati a partire dalla chiave pubblica, ma non sono

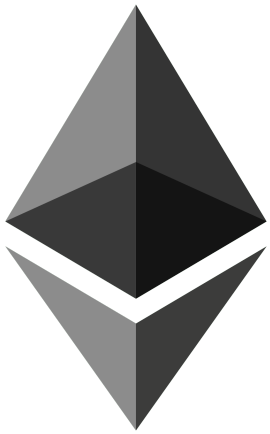

Figura 1.1: Il Simbolo di Ethereum

esattamente la chiave pubblica. In questo modo è possibile generare più indirizzi a partire dalla stessa coppia chiave pubblica/privata. La possibilità di generare ed utilizzare più indirizzi, permette sia di ottenere una maggiore sicurezza da eventuali attacchi, che di offuscare in un qualche modo le transazioni, cioè di rendere meno trasparente la corrispondenza tra gli indirizzi ed il soggetto dietro alla coppia chiave pubblica/privata. In altre parole il ritracciamento di tutti i movimenti eseguiti da uno stesso utente risulta più difficile. Tuttavia è possibile risalire alla chiave pubblica di un utente tramite un calcolo sulla "firma" che l'utente mette sulla transazione.

1.2 Blockchain

La blockchain di Ethereum è ad oggi veramente estesa (circa 300GB) ed è il cuore pulsante del sistema. Ogni nodo che partecipa alla rete contiene una copia della blockchain che aggiorna ogni volta che nuovi blocchi vengono minati (aggiunti alla blockchain).

Ogni blocco viene identificato da un hash unico e contiene delle informazioni che riguardano le transazioni tra utenti, hash del blocco precedente, informazioni di stato e informazioni relative al mining del blocco stesso. Siccome il blocco contiene lo stato di ogni blocco precedente, è stato deciso di

adottare per la memorizzazione una struttura ad albero chiamata Merkle-Patricia-Tree, dove non viene salvato tutto lo stato ma solo le differenze rispetto al blocco precedente. Questo risulta fattibile visto che il cambiamento dell' albero tra due blocchi vicini è sempre di piccole dimensioni e, per vedere l'albero completo, basterà andare indietro con il puntatore all'hash del blocco precedente. Lo stato non è contenuto nella blockchain perchè sarebbe uno spreco di memoria, visto che lo si può ottenere analizzando i blocchi uno per uno. Ogni nodo crea la sua copia di stato da tenere in locale. Lo scopo del progetto che verrà presentato è anche quello di facilitare l'analisi della blockchain di Ethereum per riuscire ad eseguire facilmente delle query (che possono riguardare statistiche d'uso, dati dei contratti, quantità di denaro trasferito e altro) che impiegherebbero invece moltissimo tempo se lanciate sulla blockchain integrale, visto che la struttura della blockchain è stata ideata per ottimizzare il più possibile lo spazio in memoria.

Per decidere quali transazioni aggiungere ad un blocco il minatore va a sceglierne un gruppo dalla transaction pool, che è una pool dove vengono mandate tutte le transazioni ancora in pending, ovvero che sono state richieste da un utente della rete ma non sono state ancora incluse in un blocco.

Ogni blocco può contenere una quantità limitata di transazioni ed esiste una politica di precedenza per l'inclusione delle transazioni in un blocco. Ciascun nodo sceglie autonomamente quali transazioni includere nel blocco da minare, al fine di rispettare i requisiti e allo stesso tempo massimizzare il suo profitto. Uno dei criteri usati per dare priorità alle transazioni è rappresentato dalle fee pagate per processare la transazione.

Ma come viene deciso quale blocco verrà aggiunto alla struttura? Ethereum prevede un sistema chiamato **Proof of Work**, questo sistema garantisce un modo complicato per minare un nuovo blocco da aggiungere alla catena ma, allo stesso tempo, rende facile per gli altri nodi verificare che il blocco aggiunto sia corretto. I blocchi vengono aggiunti da utenti detti miners che vengono ricompensati con degli ether per ogni blocco minato correttamente. La ricompensa è costituita dal reward base di mining più le fee pagate dalle transazioni.

Per l'utilizzo dell'algoritmo di mining viene salvato sulla blockchain un dataset pseudo-casuale che viene poi rigenerato ogni 30 mila blocchi chiamato *DAG*(Directed Acyclic Graph), questo è distribuito a tutti nodi e se si vuole minare un blocco è necessario salvare all'interno della propria GPU l'intero *DAG*.

Per l'algoritmo crittografico vero e proprio viene usato l'header preprocessato preso dal blocco precedente ed un *nonce*, ovvero un numero casuale di 32 bit, questi vengono combinati e usati come input di una funzione SHA-3(Secure Hash Algorithm 3) che fa parte della famiglia di funzioni crittografiche Keccak. Il risultato di questo operazione prende il nome di mix0 ed è una stringa di 128 Byte, tramite una funzione di fetch (basata su mix0)

viene deciso quali dati prendere dalla *DAG*. I dati letti verranno poi mixati a mix0 per ottenere mix1, una stringa lunga sempre 128 byte. Viene poi deciso di nuovo quali dati prendere dalla dag tramite una funzione di fetch e mixando mix1 a questi dati si ottiene mix2. Il processo viene ripetuto fino all'ottenimento della stringa mix64, che viene processata per ottenere mixDigest, lunga 32 Byte. La mixDigest viene confrontata con una sequenza target relativa al blocco che si sta minando, se mixDigest è minore del target allora il nonce viene considerato valido e inviato in broadcast a tutti gli altri nodi della rete per eseguire la verifica del nuovo blocco e confermarlo. In caso sia maggiore l'intero processo viene ripetuto dal minatore. Pertanto l'enigma crittografico da risolvere consiste nell'individuare un valore di nonce che soddisfa il requisito indicato dalla sequenza target.

Un altro importante valore è la difficoltà, che determina quanto sia difficile il mining di quel blocco. La difficoltà viene adattata per fare in modo che minare un blocco impieghi sempre lo stesso tempo e gli altri nodi abbiamo il tempo di verificarlo. La difficoltà è cresciuta nel tempo visto che sempre più miners hanno cominciato a lavorare."Il tempo medio necessario per aggiungere un nuovo blocco alla catena è 12-19 secondi" [3]. Chiaramente il costo computazionale per il mining è altissimo e vi è una corsa per garantirsi il mining dei nuovi blocchi visto che solo il primo miner che indovina il numero vince la ricompensa. Per questo nel corso degli ultimi anni molte aziende e privati hanno creato delle vere e proprie mining farm composte da diversi calcolatori dalla potenza di calcolo enorme. Una volta che un blocco viene minato tutti gli altri nodi devono verificarne la correttezza tramite un calcolo matematico piuttosto rapido, è possibile che due miner indovinino il numero quasi nello stesso momento, in questo viene aggiunto alla chain solo il primo che raggiunge il 51% dei nodi, l'altro nodo diventerà un uncle block che non verrà considerato parte della chain principale(un piccolo reward viene comunque dato ai miner di uncle blocks).

Il mining time di bitcoin è molto più alto (circa 10 minuti), mentre su Ethereum è così breve perché viene applicato il protocollo Greedy Heaviest Observed Subtree (GHOST), che permette anche ai blocchi orfani di ottenere una parte del reward visto che contribuiscono alla costruzione della blockchain principale. In questo modo si riduce di molto lo spreco di potere computazionale che viene fatto nella corsa al mining di nuovi blocchi.

Un'alternativa alla proof of work è la **Proof of Stake**, ideata nel 2011 per risolvere alcuni problemi dell'attuale proof of work. Un algoritmo pseudocasuale decide quale nodo sarà il validatore del prossimo blocco basandosi su alcuni fattori come periodo di staking (da quanto tempo il nodo sta cercando di validare un blocco) e fondi di proprietà del nodo. I blocchi non vengono più minati tramite un grosso lavoro computazionale come succedeva per la PoW ma "forgiati". Generalmente i forgiatori di blocchi non ottengono il reward per il blocco minato ma soltanto le fee pagate dalle transazioni. Per partecipare al processo di forging un nodo deve mettere in gioco una quan-

tità di denaro (detta stake) proveniente dal suo fondo che verrà congelata. Più questa stake sarà alta maggiori saranno le probabilità di aggiudicarsi il prossimo blocco. Tuttavia l'algoritmo non si basa solo sul valore della stake, altrimenti i nodi più ricchi vincerebbero sempre. Un altro metodo di scelta adottato dalla PoS è il **Coin Age**, dove il sistema sceglie il nodo che ha lasciato i propri soldi congelati per più tempo, poi una volta scelto un nodo resetterà il suo valore di coin age e dovrà aspettare un po' di tempo prima di partecipare ad un'altra forgiatura. Come metodo è utilizzato anche il Randomised Block Selection che consiste nel scegliere il nodo con la combinazione di hash più piccolo e stake più grande.

Il nodo forgiatore farà un check di validità per le transazioni e firmerà crittograficamente il blocco da pubblicare. Quando un nodo decide di smettere di forgiare dopo una certa quantità di tempo potrà accedere ai fondi congelati e alle ricompense ottenute se il sistema conferma che il nodo non ha tentato di aggiungere blocchi fraudolenti alla blockchain. Se un forgiatore ha tentato di imbrogliare il sistema questo perderà la stake congelata, grazie a questo meccanismo i nodi sono incentivati a non avere comportamenti dannosi visto che altrimenti perderebbero i loro soldi e la possibilità di forgiare blocchi in futuro.

Per controllare l'intero network e approvare transazioni fraudolente un nodo dovrebbe giocare una stake che valga almeno il 51% dell'unità monetaria in circolazione, che è quasi impossibile se parliamo di grandi network come Ethereum.

La proof of Stake ha come vantaggi quelli di non impiegare molta energia computazionale per la generazione di nuovi blocchi e,rispetto alla PoW, rendere il sistema più decentralizzato visto che non servono più gigantesche mining pool per aggiudicarsi i blocchi ma basta essere degli utenti con una buona quantità di fondi da investire.

1.3 Gas

Eseguire delle operazioni come delle semplici transazioni o degli Smart Contracts all'interno dell'ecosistema Ethereum ha un costo che viene chiamato **gas** (misurato in gwei ,1 gwei vale 0.000000001 ether) e prende questo nome dato che il suo scopo è quello di fare da carburante per le operazioni sulla blockchain. Le operazioni hanno un costo di gas diversificato, chiaramente un contratto che viene eseguito per più tempo sulla blockchain avrà un costo più alto di una normale transazione visto che il contratto sarà incluso in molti più blocchi. I costi in gas delle varie operazioni sono definiti nello yellow paper pubblicato da Ethereum, ad esempio una transazione normale costa 21000 gwei. Al momento della pubblicazione lo sviluppatore definisce anche i costi di esecuzione del contratto, in modo che gli utenti sappiano quanto gas fornire ad esso.

Questo meccanismo è stato introdotto per impedire a uno o più contratti di essere eseguiti in un loop infinito dentro i vari nodi della blockchain causando una saturazione dei nodi stessi. Se non ho abbastanza gas per far eseguire il contratto questo si bloccherà.

Il prezzo del gas subisce fluttuazioni in base soprattutto ai miner, ad esempio se vi sono tante transazioni in pending nella pool il costo medio del gas aumenterà. Quando accedo a un contratto o mando una transazione, imposto anche un limite di gas per evitare che il mio portafoglio venga svuotato nel caso ,ad esempio, che un contratto problematico continui ad andare in loop sulla blockchain.

1.4 Token

I token di Ethereum funzionano come dei gettoni all'interno della piattaforma, si acquistano con gli ether e sono scambiabili tramite transazioni esattamente come gli ether. Questi hanno diversi scopi e funzionalità all'interno della piattaforma, sono utili agli sviluppatori di Smart Contract nel caso questi necessitino di una valuta particolare all'interno della loro applicazione. Lo sviluppatore potrà poi programmare quanti token vengono emessi o ricevuti dal contratto esattamente come con gli ether, il motivo per il quale si decide di scambiare token e non ether riguarda principalmente gli Smart Contract e le Dapp(applicazioni decentralizzate usate su Ethereum). Sviluppare delle funzioni per contratti che fanno uso di token specifici risulta molto più semplice e meno macchinoso. Tuttavia i token non esistono solo come moneta sostitutiva ma possono valere come "azioni" di alcune Dapp, i possessori di questi token potranno votare e decidere il futuro dell'applicazione. Un utente può anche acquistare un token di una futura applicazione per poi usarli come moneta all'interno della Dapp stessa. Questo tipo di crowdfunding all'interno di Ethereum prende il nome di ICO(initial coin offering) ed è un metodo che permette agli sviluppatori di essere finanziati dai futuri utenti. I token seguono gli standard ERC20 ed ERC721.

ECR sta per Ethereum Request for Comments e il numero che segue identifica la proposta. ERC 20 è uno standard utilizzato solo su Ethereum che presenta una serie di regole che riguardano lo scambio, il deposito e la creazione dei token. Per essere valido un token deve rispettare dei set di funzioni che vanno poi a semplificare il compito degli Smart Contract. I contratti che implementano questo standard possono essere usati tramite una singola interfaccia. Un altro vantaggio di questo standard **ERC 20** sta nel poter implementare API per i token all'interno dei contratti, ogni contratto può interagire con un token senza conoscerne i dettagli come se questi token fossero tutti uguali. Vi sono 6 funzioni obbligatorie che il codice dei token deve avere per configurarsi come ERC20:

- *totalsupply*: numero totale di token

- *balanceOf*: token che un indirizzo possiede
- *transfer*: traferimento di n token dalla fornitura ad un utente
- *transferFrom*: trasferimento di n token da un utente ad un'altro
- *approve*: verifica una transazione rispetto al totalsupply, per evitare creazione di nuovi token
- *allowance*: si assicura che un indirizzo abbia abbastanza fondi per inviare token ad un altro

I token **ERC721** sono creati con lo scopo di emulare oggetti rari/collezionabili. Questi token al contrario degli ERC20 non sono intercambiabili, il valore è dato dalla risultante dei loro attributi. Siccome salvare tutte le caratteristiche di questi token risulterebbe oneroso per la blockchain vengono salvati invece solo dei riferimenti agli attributi dei token. I token **ERC 1155** sono stati ideati per ottimizzare la memorizzazione di token. Con gli ERC721 era necessario implementare uno Smart Contract sulla blockchain per ogni singolo token, mentre i nuovi ERC 1155 memorizzano tutti gli articoli in un unico Smart Contract che contengono l'Id dei token e le loro funzioni. Questo risulta in un numero minore di transazioni sulla blockchain e un'importante ottimizzazione di memoria.

Capitolo 2

Smart Contracts

2.1 Cosa Sono

Gli Smart Contracts funzionano come normali contratti tra uno o più utenti, la differenza sta nel fatto che il contratto, come il resto della piattaforma, risulta decentralizzato. Questo significa che non vi è alcun ente terzo a fare da intermediario per gli utenti, il contratto è pubblicato sulla blockchain ed è accessibile ad ogni nodo della catena, viene quindi garantita una forte trasparenza e affidabilità del servizio; mentre al contrario un servizio centralizzato salva solitamente i dati su uno o più server presenti nella stessa locazione geografica. Questo rende il servizio centralizzato soggetto ad attacchi hacking, avarie dei server, frode ed accessi non autorizzati.

I tempi burocratici per uno Smart Contract sono praticamente inesistenti visto che, al contrario dei contratti reali, non necessitano l'approvazione di un tribunale, un notaio o una banca per essere applicati. Una volta pubblicato il contratto è accessibile e usufruibile dagli utenti del network in maniera rapida.

I contratti sono pubblici, siccome ogni nodo della blockchain è a conoscenza di tutto il sistema chiunque può vedere il bytecode del contratto che viene salvato in un blocco. Una volta che un contratto viene inviato sulla blockchain questo risulta immutabile, nemmeno lo sviluppatore originale potrà modificarlo,quindi è necessario prestare molta attenzione quando vogliamo pubblicarne uno visto che non c'è modo di cambiare il codice una volta inviato. Tuttavia è possibile includere una funzione di self destruction all'interno del contratto per fare in modo che questo si auto-cancelli dalla blockchain per essere sostituito da uno nuovo. L'unico modo di aggiornare il proprio codice sulla blockchain è suddividerlo in più contratti, alcuni contratti saranno cancellabili e sostituibili da nuovi contratti mentre altri no(ad esempio la backbone del programma) . Per gli utenti l'immutabilità è un fattore di sicurezza fondamentale, nessuno manderebbe dei soldi ad un contratto se sapesse che da un momento all'altro questo potrebbe subire delle

modifiche.

Un'altra caratteristica del contratto è quella auto-esecuzione, un contratto viene pubblicato e rimane in stato "dormiente" sulla blockchain finché non viene attivato da un particolare evento comunemente detto trigger.

Esistono tool per gli sviluppatori che permettono di lavorare agli Smart Contract su una blockchain di prova senza spendere ether. **Ganache** è il programma open-source più famoso che fornisce agli sviluppatori una blockchain personale e **Truffle** è il framework più usato per sviluppare e testare codice Solidity, ovvero il linguaggio ad alto livello più usato su Ethereum per lo sviluppo di codice.

Nell'effettivo uno Smart Contract è semplicemente del codice di tipo deterministico scritto in uno dei linguaggi accettati dalla Ethereum Virtual Machine ed eseguito sulla blockchain. Nonostante il nome, il contratto non ha nulla di intelligente, si limita ad eseguire il suo codice indipendentemente da quale utente sta interagendo con esso. I linguaggi utilizzati su Ethereum sono: Mutan, LLL, Serpent, Bamboo, Vyper e solidity. **Solidity** è attualmente il più popolare e risulta simile a javascript. Si tratta anche del linguaggio che presenta le features più complete per lo sviluppo di Smart Contract per uso comune. **Vyper** è un linguaggio ispirato a python che si concentra sulla sicurezza e sulla semplicità del contratto. Gli altri linguaggi sopra citati risultano deprecati e non più in uso per i contratti moderni.

Ethereum non è tuttavia una piattaforma perfetta e presenta ancora dei limiti, il codice viene scritto da esseri umani e, di conseguenza, c'è la possibilità che vi siano bug e imperfezioni all'interno di esso. L'immutabilità è un grande vantaggio ma ha anche alcuni svantaggi, ad esempio quando, nel 2016, la DAO (organizzazione che permetteva il crowdfunding di nuove applicazioni per Ethereum) fu vittima di un attacco hacker, milioni di ether sono stati rubati a causa di difetti all'interno dei contratti, questi non potevano essere modificati proprio per l'immutabilità degli stessi. Questo non sarebbe successo in un normale sistema centralizzato gestito da una sola azienda. Comunque per risolvere questo problema è stato necessario eseguire un **hard fork**, una nuova blockchain è stata creata sulla base della precedente, solo che i soldi "rubati" sono stati restituiti ai proprietari. La vecchia catena è tuttora utilizzata e il sistema prende il nome di **Ethereum Classic**.

Un problema (che riguarda più l'etica che l'informatica) è quello giuridico, l'uso di Ethereum non viene normato da leggi specifiche quindi ne consegue che sia gli utenti che il sistema siano più liberi ma meno tutelati. In caso, per esempio, un utente sia vittima di frode e perda i suoi soldi inviandoli all'indirizzo sbalgiato non c'è modo per lui di ottenere indietro questi soldi, visto che non può fare appello a nessun tribunale o a entità terze come invece succede nel mondo odierno. Se una transazione viene eseguita non c'è modo di tornare sui propri passi perchè questa è immutabile. L'assenza di intermediari è anche un punto di forza del sistema, le persone possono stipulare senza problemi contratti con sconosciuti anche senza fiducia tra gli

interlocutori perché è il sistema stesso a garantire la sicurezza tramite la sua tecnologia. Il contratto viene eseguito se e solo se gli utenti lo rispettano e nessuno può fare modifiche in corso d'opera o nascondere informazioni.

2.2 Oracoli

Talvolta gli Smart Contracts necessitano di venire a conoscenza di fatti avvenuti al di fuori della blockchain, come ad esempio la quotazione di una criptovaluta, l'esito di una gara sportiva, il risultato di un'elezione etc. Per uno Smart Contract è impossibile usare informazioni esterne alla blockchain per ragioni di sicurezza, l'esecuzione del sistema deve essere deterministica. L'informazione ottenuta dovrà quindi essere immutabile visto che ogni miner che andrà ad eseguire il contratto dovrà ottenere sempre lo stesso risultato anche in momenti diversi, per una normale API esterna alla blockchain questo aspetto non è chiaramente garantito.

Quindi è necessario ottenere queste informazioni in un altro modo, tramite gli oracoli. Un oracolo è un servizio che prende i dati al di fuori della blockchain e ne crea una copia immutabile che invia ad un blocco tramite una transazione, questo permette agli Smart Contracts di ottenere le informazioni di cui necessitano senza perdere le loro proprietà di affidabilità e immutabilità. Un difetto che hanno gli oracoli è quello di non essere decentralizzati come il resto della piattaforma e questo porta dei rischi:

1. L'informazione ricevuta dall'API potrebbe essere sbagliata, per un errore oppure per una manomissione esterna

2. L'oracolo può essere ingannato da uno Smart Contract per ricevere delle risorse a cui non dovrebbe aver accesso

3. I contratti spesso necessitano di informazioni accurate e con una certa frequenza, anche ad ogni blocco, tuttavia come abbiamo detto nell'introduzione non c'è la certezza che una transazione entri nel prossimo blocco minato, anche se l'offerta di fee è alta.

Una soluzione potrebbe essere quella di affidarsi al proprietario del dataset stesso per ricevere informazioni, questo ente deve essere pubblicamente riconosciuto e la sua affidabilità e sicurezza garantita, questo rimanendo però in un tipo di tecnologia centralizzata. Oppure è possibile rendere l'oracolo decentralizzato, ma questo campo è ancora in via di sviluppo. Il servizio funzionerebbe con lo stesso ragionamento della blockchain, un insieme di nodi che raggiungono il consenso su un'informazione senza nessun meccanismo di fiducia o accordo tra le parti.

Nell'immagine 2.1 vediamo un esempio del ciclo di vita di un contratto tipo che riguarda una possibile assicurazione per un volo aereo.

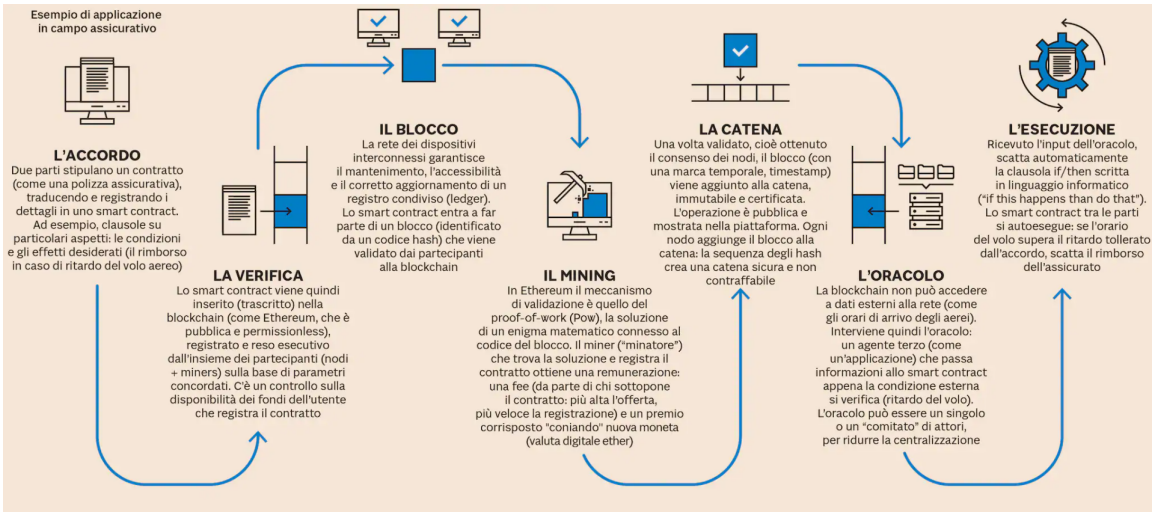

Figura 2.1: esempio di contratto in campo assicurativo,Il Sole 24 ore

Capitolo 3

Database Relazionale

3.1 Modello ER

Per analizzare la blockchain di Ethereum abbiamo trasferito i dati dalla blockchain originale in una banca dati salvata su un server remoto tramite l'utilizzo di un software (cap4) sviluppato durante il periodo di tirocinio presso l'università. Non è stato possibile fare una trascrizione 1:1 di tutti dati della rete Ethereum visto che alcuni campi possono essere ottenuti solo da nodi completi della rete che eseguono gli Smart Contracts tramite l'EVM. Abbiamo dunque salvato sul database una parte dei dati della blockchain che possiamo andare ad interrogare tramite delle query che necessiterebbero invece di una quantità decisamente maggiore di tempo se eseguite direttamente sulla blockchain integrale di Ethereum, visto che questa è ottimizzata per memorizzare una quantità minima di informazioni e non risulta adatta per interrogazioni di tipo statistico.

In questo capitolo viene analizzata la struttura del database, per far questo abbiamo ideato un modello entità-relazione(figura 3.1) per riuscire a spiegare in maniera astratta e ad alto livello il modello relazionale della nostra base di dati. Nel modello in figura sono state rappresentate tutte le entità ma solo gli attributi che servono per definire gli identificatori.
Le entità sono:

- **Block**: blocco della catena identificato qui dal suo numero. Un blocco contiene n transazioni, anche se non esistono blocchi senza transazioni la cardinalità è (0,n) perchè sul database viene salvato per prima cosa il blocco, poi le transazioni che sono al suo interno. Ogni blocco viene minato da uno ed un solo miner, questo miner è un utente di Ethereum che possiede un normale account. Anche qui salviamo prima il blocco dell'account del minatore quindi la cardinalità della relazione è (0,1).
- **TxSummary**: l'entità TxSummary rappresenta una semplice transazione e viene identificata da un codice hash univoco. La differenza da

tx sta nel numero degli attributi, txSummary contiene solo quelli fondamentali mentre tx presenta quelli più specifici, questa scelta è stata presa per rendere l'accesso e l'analisi alla tabella delle transazioni più rapido. La transazione è contenuta in un blocco e ha un mittente e un destinatario, il valore minimo di entrambi è 0 visto che salviamo sulla base prima la transazione e poi l'account. La transazione potrebbe anche avere destinatario nullo in caso sia una transazione per la creazione di un contratto.

- **Tx**: entità che serve per rappresentare alcuni attributi aggiuntivi di una transazione (come ad esempio le informazioni riguardo al gas). Per ogni istanza di txSummary ce n'è una di tx, utilizzano anche lo stesso identificatore visto che rappresentano le stesse transazioni.

- **Address**: address è l'entità che rappresenta un portafoglio di un utente umano o uno Smart Contract su Ethereum. Sono stati raggruppati in una sola entità per via di alcune caratteristiche in comune, come il fatto di essere l'input o l'output di transazioni e l'essere identificati da un indirizzo hash univoco. la generalizzazione è totale perché ogni address sarà un account di un utente o uno Smart Contract, poi è esclusiva siccome non esistono address che siano sia Smart Contract che utenti.

- **Account**: entità figlia di address, rappresenta l'account di un normale utente. Un utente può anche essere un minatore ed aver minato un certo numero di blocchi durante la sua esistenza.

- **Contract**: entità figlia di address, serve per rappresentare un Smart Contract

3.2 Campi database

Ora vado invece a elencare i campi di ogni tabella del database

Block

number: numero scalare progressivo che identifica il blocco
size: dimensione del blocco in byte
timestamp: timestamp nel formato generale unix, contiene l'istante in cui viene generato
miner: indirizzo di 160-bit che indica il miner del blocco
gasLimit: massimo gas utlizzabile da un blocco
gasUsed: gas utilizzato da tutte le transazioni del blocco
transactions: lista contenente tutti gli hash delle transazioni del bloccov

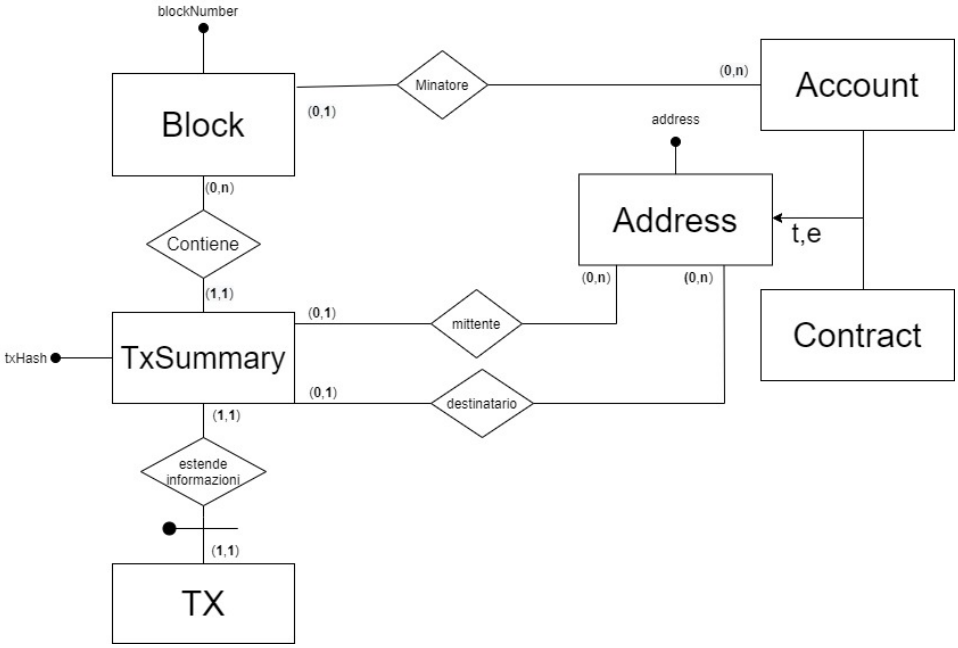

Figura 3.1: modello ER

parentHash: hash ottenuto dal risultato della funzione crittografica keccak-256. Come input questa funzione prende l'header del blocco precedente
hash: hash del blocco corrente
extraData: array di byte contenente eventuali dati aggiuntivi
difficulty: valore scalare che corrisponde alla difficoltà del blocco corrente.
totalDifficulty: somma totale dei valori della difficoltà escluso questo blocco
nonce: hash generato dalla proofOfWork
mixHash: hash di 256 bit che viene combinato al nonce per provare che è stato svolto un sufficiente sforzo computazionale per questo blocco.
receiptsRoot: hash del nodo radice relativo alla struttura trie che contiene le receipt delle transazioni del blocco corrente
stateRoot: hash del nodo radice relativo alla struttura trie che contiene lo stato di tutte le transazioni
transactionRoot: hash del nodo radice relativo alla struttura trie che contiene tutte le transazioni del blocco corrente
logsBloom: informazioni di log salvate in una struttura dati detta bloom filter, è una struttura che permette in modo rapido di conoscere se un elemento fa parte di un set o no. All'interno di Ethereum è utile per vedere se una transazione è presente in un blocco.
uncles: lista degli uncle blocks
sha3Uncles: hash di tipo sha-3 che contiene di dati degli uncle del blocco

dollarquote: valore in dollari dell'Ether nel momento in cui il blocco è stato minato
totalFee: totale delle fee pagate al miner

TxSummary

txHash: hash della transazione
balanceFrom: Ether sul conto del mittente
balanceTo: Ether sul conto del destinatario
blockNumber: numero del blocco che contiene la transazione
sender: indirizzo hash del mittente
nonce: sequenza casuale generata durante la Proof of Work
receiver: indirizzo hash del destinatario
value: quantità di wei trasmesso nella transazione

Tx

txHash: identico a quello del campo tx
blockNumber: identico a quello del campo tx
contractAddress: in caso di transazione per la creazione di un contratto questo campo contiene l'indirizzo del contratto, null altrimenti
cumulativeGasUsed: somma del gas usato da questa transazione più le precedenti del blocco
gas: gas usato dalla transazione
gasPrice: numero di wei pagati per ogni unità di gas spesa durante la transazione
gasUsed: Quantità totale di gas utilizzato dopo che questa transazione è stata eseguita
input: eventuali dati inseriti all'interno della transazione
logs: array di log generati dalla transazione
logsBloom: dati riguardanti il bloom filter
v/r/s: parametri generati per la firma del mittente, volendo tramite questi ultimi e l'address è possibile ottenere la chiave pubblica del mittente.
status: booleano per verificare se l'esecuzione della transazione ha avuto o meno successo
transactionIndex : indice della transazione all'interno del blocco
fee: fee(wei) pagate dal sender, per calcolarle viene moltiplicato il prezzo del gas per il gas usato nella transazione

Account

address: indirizzo hash dell'account
balance: saldo dellaccount(in wei)

txCount: numero di transazioni inviate dall'account

Contract

address: indirizzo hash dell'account
codeSize: numero di caratteri presenti nel codice del contratto
functionNumber: numero di funzioni presenti nel contratto
tokenTotalSupply: fornitura totale di token, solo nel caso il contratto sia un token contract

3.3 Schema Relazionale

Per lo schema relazionale andiamo semplicemente ad indicare chiave primaria e chiave esportata di ogni tabella, riportare tuti gli attributi risulterebbe solo poco chiaro per la comprensione delle relazioni interne al database.

Tabella block
Chiave Primaria: *blockNumber*
Chiave esportata: *miner*, riferito ad *address* della tabella account

Tabella TxSummary
Chiave Primaria: *txHash*
Chiave esportata: *blockNumber* , riferita a *blockNumber* della tabella *block*

Tabella Tx
Chiave Primaria: *txHash*
Chiave esportata: *blockNumber* , riferita a *blockNumber* della tabella *block*
txHash, riferita a *txHash* della tabella txSummary

Tabella account
Chiave Primaria: *address*

Tabella contract
Chiave Primaria: *address*

Capitolo 4

Framework

4.1 Librerie software utilizzato per il progetto

In questo capitolo si presenta la libreria sviluppata durante il progetto. Tale libreria ha lo scopo di salvare il contenuto della blockchain di Ethereum in un database relazionale e di estrarre delle informazioni o eseguire analisi sui corpo dei contratti presenti nella stessa. Per lo sviluppo di questa libreria ho collaborato con un collega dell'università di Verona durante il periodo di stage, la versione finale del software è stata usata da entrambi per fare delle analisi sul database relazionale generato. Al fine di creare tale libreria sono state da prima analizzate alcune librerie esistenti (cap 4.1.1), una volta selezionata la libreria più adatta allo scopo (cap 4.1.2)sono state aggiunte delle funzionalità alla stessa (cap 4.2), infine sono state eseguite delle analisi sugli Smart Contract (cap 5).

4.1.1 scartate

Web3j :serie di librerie java che permettono l'interazione coi nodi del network di Ethereum, queste librerie risultano utili per chi vuole programmare Smart Contracts e scripts per la piattaforma etherum ma sono risultate poco efficaci per i nostri scopi[38]
Web3js: librerie javaScript dove però la scarsa conoscenza di JavaScript ci ha portato ad abbandonare l'idea[37]
Presto: engine che permette di usare delle query sql sulla blockchain. Il programma è stato abbandonato visti i vari errori durante la fase di installazione e le istruzioni d'uso poco chiare[28]
Quick Blocks: collezione di librerie, applicazioni e strumenti(in linguaggio c++) che permettono di estrarre dati dalla blockchain di Ethereum.Gli strumenti offerti Quick Blocks sono risultati efficaci per ottenere i dati necessari tuttavia la transizione su un database sql sarebbe risultata abbastanza macchinosa e complessa vista anche la nostra competenza del linguaggio c++.[27]

4.1.2 utlizzate

web3.py: libreria di Python con API simili a quelle di web3.js. Questa libreria è quella che alla fine abbiamo scelto visto che tramite essa siamo riusciti a trasferire parte della blockchain su un database sqlite3(integrato già in python). Per il nostro scopo abbiamo poi deciso di migrare il database in un file di tipo sql eseguibile da postgreSql in locale. Le tabelle che abbiamo riportato nel database sono: txSummary, tx e block. TxSummary contiene le informazioni fondamentali delle transazioni, tx quelle complete mentre block quelle dei singoli blocchi, ma questo è stato spiegato in maniera più specifica nel cap 3 [36]
psycopg2: libreria python che permette di inserire le istruzioni per database PostgreSql direttamente all'interno di un programma scritto in python, implementa completamente la Python DB API 2.0.[29]

4.2 Funzionamento del programma

Come detto in precedenza lo scopo del programma è il trasferimento della blockchain di Ethereum su un database PostgreSQL. Per fare questo il programma scrive tutte le istruzioni per la creazione del database su un file di estensione *.sql* e poi esegue queste istruzioni sul server vega fornitoci dall'università. Si è deciso di utilizzare un file di estensione *.sql* intermedio per caricare i blocchi invece di eseguire direttamente le singole istruzioni di caricamento sul database, questo per evitare di generare un numero troppo elevato di connessioni al database che avrebbe probabilmente causato uno spreco risorse. Il programma carica sul database le informazioni solo dopo aver analizzato 100 blocchi, questo ci permette, in caso di errore, di poter far ripartire il programma in seguito sapendo dove l'errore è avvenuto e,di conseguenza, da quale blocco partire una volta che il problema viene risolto. Nella Figura 3.1 vediamo l'organizzazione gerarchica dei file che riguardano il nostro progetto. Per utilizzare il software è necessario aver installato sulla propria macchina le librerie web3.py e psycopg, poi per avviare il programma basta eseguire database.py con una versione di python superiore o uguale alla 3.6. Il programma è stato scritto prendendo spunto da quello della dottoressa Aleksandra Sokolowska che ha ideato un programma per interagire con la blockchain di Ethereum tramite la libreria web3.py e trasferire i dati su un database sqlite3. [8]

4.2.1 database.py

Main del progetto, questo sarà il programma che lanceremo per avviare il nostro framework. Qui impostiamo alcuni valori relativi al trasferimento

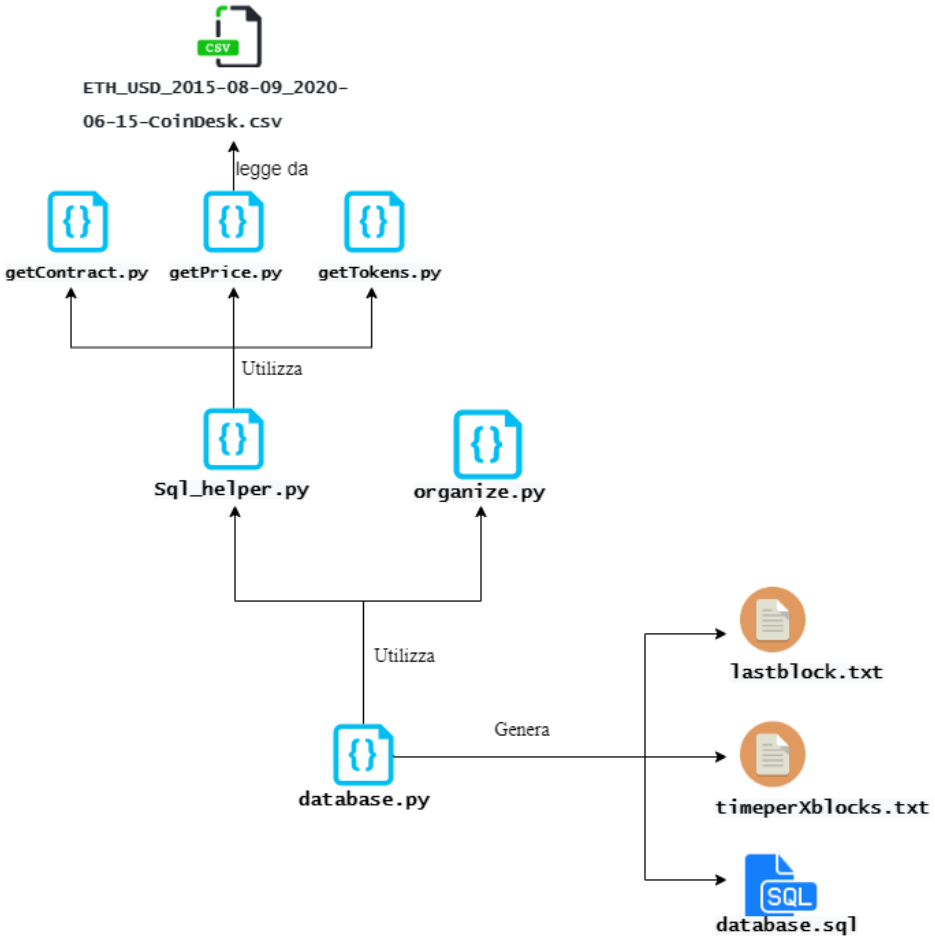

Figura 4.1: schema gerarchizzazione file

della blockchain sul database. Analizzo di seguito le parti più importanti del codice.

```
from web3 import Web3
from organize import *
import time
from sql_helper import *
import os
import psycopg2
```

Iniziamo importando tutte le librerie e i file esterni. **web3** è l'object della libreria web3.py che ci permetterà di collegarci tramite un nodo alla blockchain. **time** è la libreria di python che utilizzeremo per contare quanti secondi sono passati tra la scrittura di un gruppo di blocchi su database.sql. **os** è un modulo python per interagire col proprio sistema operativo

```
from web3 import Web3
# 1. connection via Infura
web3 = Web3(Web3.HTTPProvider("https://mainnet.infura.io/v3
                                /...))
# 2. or connection via local node
web3 = Web3(Web3.IPCProvider('/your-path-to/geth.ipc'))
```

Qui decidiamo come connetterci al nodo di Ethereum per interagire con la blockchain. Ci sono due possibilità: utilizzare un nodo locale oppure uno hostato. Nel caso di un nodo locale è necessario scaricare sulla propria macchina l'intera blockchain tramite un Ethereum client (ad esempio geth) per sincronizzare la propria blockchain con quella della rete. Nel caso di un nodo hostato andremo semplicemente a connetterci tramite connessione HTTP ad un nodo gestito da qualcun altro, nel nostro caso dal servizio **Infura**. Per il nostro progetto l'alternativa migliore è il nodo locale visto che dovremo eseguire letture continue sulla blockchain. Dalla stessa Dr. Aleksandra Sokolowska è stato affermato che la scrittura su database è 100 volte più veloce se si usa un nodo locale rispetto ad uno hostato, questo perché nel caso del nodo locale prendiamo l'informazione dalla nostra macchina mentre in quello hostato eseguiamo una richiesta HTTP ad un server.

```
Nblocks = 100000
try:
    with open('lastblock.txt', 'r') as f:
      start = int(f.read())+1
except FileNotFoundError:
    start = 0
```

Decidiamo quanti blocchi trasferire sul database, il file lastblock.txt contiene il numero dell'ultimo blocco trasferito, se questo file non è presente viene creato e il valore da cui iniziare viene imposto nel programma. Se il file da-

tabase.sql non esiste viene generato dal programma. Poi iniziamo ad iterare su ogni blocco.

```
for block in range(start, start+Nblocks):
```

Salviamo il blocco numero block nel database.

```
  block_table, block_data = order_table_block(block,web3)
  file1 = open("database.sql","a")
  file1.write(replace_wordb(block_table))
  file1.write(replace_worda(block_table,web3,"miner"))
```

order_table_block è una funzione di *organize.py* e si occupa di andare ad ottenere le informazioni che riguardano il blocco, come l'hash, lo state root, il *nonce*...
replace_wordb è la funzione di *sql_helper.py* che permette di scrivere su *database.sql* il comando per l'inserimento del blocco numero *block* mentre replace_worda andrà a scrivere il codice sql per l'inserimento del miner del blocco nella tabella account.

```
    totFee = 0
  if block < 4_370_000 :
    totFee = 5
  elif block < 7_280_000 :
    totFee = 3
  else :
    totFee = 2
```

Selezioniamo il reward statico che il minatore ha ottenuto per il mining del blocco n. *block*, notiamo subito che con l'avanzare del numero dei blocchi Ethereum ha deciso di diminuire il mining reward del blocco. Questo è stato fatto siccome Ethereum non vuole che si generi una quantità troppo grande di ether, altrimenti l'inflazione crescerebbe e il potere d'acquisto calerebbe. Al momento della stesura della tesi Ethereum sta pensando di ridurre ulteriormente il reward statico da 2 a 0.5 ether.

```
  for hashh in block_data['transactions']:
  txSummary_table, tx_data = order_table_txSummary(hashh,block,
                                web3)
  TX_table = order_table_tx(tx_data,hashh, web3)
```

Ora andiamo ad analizzare le singole transazioni presenti nel blocco, per come è strutturato Ethereum ci sono delle transazioni che in gergo vengono chiamate interne, queste non sono ottenibili dalla blockchain ma sono delle message calls (solitamente fatte tra gli Smart Contracts) e sono visibili solo simulando l'esecuzione del contratto tramite un'istanza della EVM. *Order_table_txSummary* è una funzione di organize.py che ottiene

le informazioni base che riguardano una transazione, questi elementi sono: from,to,value,hash,block number e *nonce*. Queste serviranno per popolare la tabella txSummary. *order_table_tx* ha lo stesso scopo di *order_table_txSummary*, la differenza sta nel fatto che i dati presi da order table sono tutti quelli che riguardano la transazione e verranno salvati sulla tabella tx.

```
file1.write(replace_wordt(TX_table))
file1.write(replace_wordq(txSummary_table))
```

questi due *replace_word* sono funzioni di *sql_helper.py* e hanno come compito quello di scrivere su database.sql il codice per l'inserimento delle transazioni rispettivamente nelle tabelle tx e txSummary.

```
totFee = totFee + web3.fromWei((TX_table.get("gasUsed") *
TX_table.get ("gasPrice")) ,'ether')
```

Calcolo il reward totale pagato al miner del blocco, questo si ottiene sommando il reward statico alle fee pagate dalle singole transizioni. Per calcolare le fee pagate dalle transazioni moltiplichiamo il prezzo del gas per la quantità di gas utilizzato.

```
file1.write(replace_worda(txSummary_table,web3,"from"))
file1.write(replace_worda(txSummary_table,web3,"to"))
```

Ora prendiamo il mittente e il destinatario della transazione e aggiungiamo anch'essi al database, verranno salvati nella tabella account gli address degli utenti mentre nella tabella contract gli address dei contratti.

```
file1.write(replace_wordFeeBlock(totFee,block))
```

Infine le fee calcolate in precedenza vengono aggiunte al blocco sempre tramite una funzione di *sql_helper* Ogni 10 blocchi il tempo di esecuzione viene salvato su timeperXblocks.txt, questo ci aiuta a capire se è stato necessario più o meno tempo per elaborare determinati blocchi.

```
connection = psycopg2.connect(user = "deboni", password = "
                              eth2004", host = "localhost" ,
                              port = "5437 ", database = "
                              ethdb")
with connection:
    with connection.cursor() as cursore:
       cursore.execute(open("database.sql", "r").read())
connection.close()
```

Poi tramite la libreria pscopg2 ci colleghiamo al database universitario **"ethdb"** ed eseguiamo tutti i comandi presenti in database.sql.

4.2.2 organize.py

Serie di librerie che permettono di estrarre le informazioni dalla blockchain attraverso web3.py. Le informazioni estratte dalle API di web3 vengono salvate dal programma in una struttura dati di tipo dictionary, queste informazioni vengono poi convertite per essere compatibili col tipo di variabili di *PostgreSQL*. Il risultato restituito dalle varie funzioni order sarà dunque un dictionary contenente per ogni campo del blocco o della transazione il valore corrispondente.

4.2.3 sql_helper.py

sql_helper.py contiene tutte le funzioni relative ai comandi sql che verranno usati per generare ed in seguito popolare il nostro database.

create_sql permette di generare le tabelle del database, queste tabelle sono: block, txSummary, tx, account e contract. Per generare una tabella standard il comando da usare sarà simile a questo:

```
CREATE TABLE IF NOT EXISTS(key1 TIPOkey1 PRIMARY KEY ,
attributo2 TIPOattributo2,attributo3 TIPOattributo3,...)
```

IF NOT EXISTS è un flag che permette di generare la tabella solo se questa non è già presente all'interno del database, questo perché si presuppone che il programma verrà lanciato più volte in istanti di tempo diversi vista l'enorme quantità di dati da caricare dalla blockchain.

CREATE TABLE va anche ad impostare la chiave primaria, come ad esempio l'hash per un blocco o per una transazione. Create_sql crea anche gli indici che verranno usati per ottimizzare le future query di ricerca che verranno eseguite, questo tramite il comando

```
CREATE INDEX IF NOT EXISTS
        nome_indice(attributo 1,attributo 2,...)
```

La funzione *replace_wordb* tramite il file dictionary passato come parametro va a generare il codice sql che riguarda l'inserimento dei record all'interno della tabella block.

```
s = """\nINSERT INTO block VALUES ( blockNumber , blockGasUsed
                                ,\' blockHash \',\'
                                blockLogsBloom \',\' blockNonce
                                 \',\' difficulty \',\'
                                extraData \', gasLimit ,\'
                                miner \',\' mixHash \',\'
                                parentHash \',\' receiptsRoot
                                \',\' sha3Uncles \', size ,\'
                                stateRoot \', timestamp ,\'
                                totalDifficulty \',\'
                                transactionsRoot \',\' uncles
                                \',"""
```

Questa è la stringa di partenza che useremo per generare il codice di ogni blocco, la funzione *replace_ wordb* si occupa di sostituire i vari campi della tabella con gli effettivi valori presenti nella variabile dictionary. Questa funzione verrà lanciata una volta per ogni blocco della blockchain. La query risultante sarà all'incirca questa:

INSERT INTO block VALUES(21000,'0x0465...',...)

replace_ wordq e *replace_ wordt* hanno lo stesso compito di replace_wordq solo che invece di scrivere i blocchi vengono scritte le transazioni contenute all'interno dei blocchi stessi. Il codice risulta quindi molto simile nella struttura a quello della funzione vista in precedenza.

Replace_ worda è la funzione che si occupa di salvare i contratti e gli utenti nelle rispettive tabelle.

```
if  dictionary[user] == None :
  stringInsert = "\n"
elif web3.toHex(web3.eth.getCode(dictionary[user])) == "0x":
        stringInsert = insertAccount(dictionary,web3,user)
else :
        stringInsert = insertContract(dictionary,web3,user)
```

Per capire se l'address è di un utente o di un contratto viene lanciata l'API di web getCode e come parametro si passa l'indirizzo da analizzare. Questa API restituisce il bytecode del contratto associato all'address, mentre nel caso l'address sia di un utente Ethereum verrà restituito "0x".
insertContract andrà ad inserire nel database le entry della tabella contract.

```
contractAdd = dictionary[user]
code = getContractCode(contractAdd)
```

per prima cosa ottengo l'address del contratto e il suo codice sorgente.

```
for string in code:
        for char in string:
            charCount = charCount + 1
            if char.lower() == "def" or char.lower() == "
                                              function" :
                funCount = funCount + 1
            if char == "clientOfdAppBridge" \
            or  char == "TownCrier"       \
   or    char == "usingProvable"    \.....:
    oracleUser = True
```

Ora andiamo ad analizzare il codice sorgente per ottenere le informazioni che andremo ad inserire nella entry. I due for annidati ci permettono di iterare una parola per volta del codice; viene quindi tenuto conto del numero

di parole totali e del numero di funzioni presenti nel contratto. La parola chiave che indica se sta venendo definita una funzione è *"def"* per vyper e *"function"* per solidity.

Poi, sempre guardando le parole chiavi, verifichiamo se un contratto è un *oracle contract*, ovvero se un contratto utilizza un servizio di oracoli che lo può mettere in contatto con API esterne. Per verificare se un contratto utilizza dei servizi di oracoli abbiamo controllato se all'interno del codice sono presenti i metodi che riguardano i più famosi strumenti di Ethereum messi a disposizione per comunicare con l'esterno della blockchain. Tramite la funzione *getTokenBalance* se il contratto è un token contract verrrà salvato sul database anche la sua fornitura di token.

insertAccount andrà invece semplicemente a salvare nella tabella account i dati dell'utilizzatore.

Entrambi gli insert contengono una differenza nel codice sql rispetto ai replaceword visti in precedenza per l'inserimento di nuove entry.

```
INSERT INTO account VALUES(.....)ON CONFLICT(address)
DO UPDATE SET
(address,balance,txcount,tokenBalance) = (........)
```

Nel caso in cui la entry sia già presente nel database viene eseguito solo un aggiornamento, non un nuovo inserimento, questo perché è normale che un utente o un contratto siano presenti in più transazioni, anche di blocchi diversi, come mittente o come destinatario, quindi inserirli due volte causerebbe un errore in PostgreSql siccome non ci possono essere due righe con la stessa chiave primaria.

```
def replace_wordFeeBlock(totFee,block):
  return "UPDATE block \n SET totalFee ="+str(totFee)+"\n WHERE
                                blockNumber = "+str(block)+";\
                               n"
```

replace_wordFeeBlock va ad inserire nella tabella block il totale delle fee che ha ottenuto il miner del blocco, questo viene lanciato dopo che tutte le transazioni sono state analizzate visto che ogni singola transazione contiene all'interno la quantità di fee che è stata pagata.

getContract.py restituisce semplicemente il codice sorgente di un contratto utilizzando una delle API messe a disposizione **Etherscan**, un servizio che permette di analizzare dati e statistiche di Ethereum come la variazione del valore dell'ether nel tempo, la capitalizzazione sul mercato e varie informazioni che riguardano token, transazioni, contratti etc..

La Api restituisce il codice pubblicato dallo sviluppatore, Etherscan si occupa di verifcare che il codice pubblicato corrsiponda, in termini di linguaggio macchina, al bytecode presente nella transazione di creazione del

contratto.
getTokens.py restituisce la fornitura totale di token relativa ad uno Smart Contract prendendo come parametro l'indirizzo del contratto stesso.
getPrice.py calcola il valore in dollari dell'ether per una data specifica che può andare dal 9-8-2015 al 15-06-2020. Per fare questo legge i valori da un file .csv esterno scaricato dal sito coindesk.com[19] , sito di notizie specializzato in bitcoin ed altre valute digitali.

Capitolo 5

Analisi Contratti

5.1 Statistiche generali

Andiamo ora a fare alcune osservazioni generali sugli Smart Contracts, in particolare come questi sono cambiati nel corso della storia di Ethereum.

Per far questo vedremo alcuni grafici che contengono informazioni riguardanti gli Smart Contracts ottenibile dalla blockchain o da explorer esterni, con un periodo di analisi che va dal 2016 al 2020 circa.

Il grafico in figura 5.1 è stato disegnato attraverso il blockchain explorer etherscan.io[5] e va ad indicare quanti contratti sono stati verificati,ovvero accettati dal sistema e resi pubblici, nel periodo che va da fine 2015 a fine 2017. Ethereum è diventato pubblico il 30 luglio 2015, ma i primi blocchi risultano quasi tutti vuoti visto che è stato necessario minare unità di criptovaluta prima che gli utenti potessero farne uso. Le prime transazioni risalgono circa a luglio 2015 e, poco dopo,sono stati pubblicati anche i primi Smart Contracts. Nella prima metà del 2017 Ethereum ha vissuto un boom economico che ha portato la il suo valore monetario a crescere del 2200%, supponiamo quindi che sia stato questo il motivo dell'aumento di contratti verificati, che sono passati da qualche decina a centinaia di nuovi contratti ogni giorno. L'aumento del valore ha quindi portato un maggior numero di sviluppatori a credere nell'iniziativa Ethereum e ad investire tempo e risorse nello sviluppo di Smart Contracts.

In fig 5.2 vediamo ora quanti token appartenenti allo standard ERC-20 sono stati trasferiti giorno per giorno dal 2016 fino all'inizio del 2020. Si nota subito un'importante crescita nel giugno del 2017 rispetto ai mesi precedenti, questo è dato dall'esponenziale crescita di popolarità dei token ERC-20 dato dalle ICO(Initial Coin Offering), che rappresentano i finanziamenti per le nuove Dapp e i nuovi servizi da sviluppare per Ethereum. La crescita del numero di trasferimenti è conforme alla crescita della piattaforma, ci sono però 3 picchi piuttosto evidenti. Nel novembre del 2017 erano in atto alcune

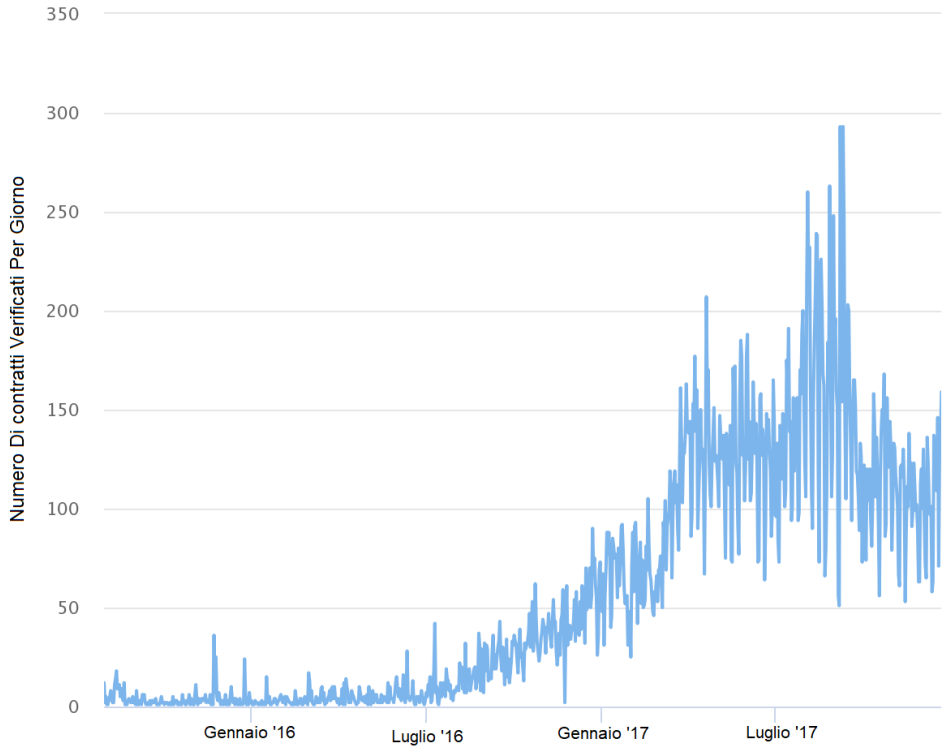

Figura 5.1: numero di contratti pubblicati su Ethereum ogni giorno

delle più redditizie ICO mai lanciate come Filecoin, Tezos, Bancor, Polkadot e Qhash che hanno guadagnato circa 600 milioni di dollari dalla vendita di token ERC20. Si considera il tasso di conversione Ether-dollari alla data di chiusura delle singole ICO.

Nel 2018 la crescita di trasferimenti giornalieri è continuata fino a giugno dove ha subito un calo dato probabilmente dall'abbassamento del valore in borsa della criptovaluta di Ethereum, di conseguenza anche gli acquirenti e utilizzatori di token potrebbero essere stati disincentivati ad investire in questo periodo. Un altro motivo del calo potrebbe essere stato il fallimento della maggior parte delle ICO iniziate nel 2017, Coindesk[19] riporta che l'80% delle ICO iniziate fossero a scopo di truffa o fasulle.

Nel 2019 il valore di Ethereum è aumentato visti gli investimenti di grosse aziende, la crescita del settore del gaming tramite i token collezionabili ERC721 e delle DAOs (organizzazioni decentralizzate). Il numero di token ERC20 scambiati invece ha subito una fluttuazione verso l'estate ma poi è tornato al valore di circa 400 mila unità al giorno. Infine Nel 2020 vediamo di nuovo un piccolo aumento.

Figura 5.2: numero di Token dello standard ERC20 scambiati ogni giorno

5.2 Query sul database

In questa sezione vado ad introdurre le query che ho lanciato sul database per ottenere informazioni di carattere statistico sui campi dei contratti salvati, l'analisi riguarderà dunque la quantità di token creati da un contratto, il numero di funzioni e il numero di caratteri presenti nel codice del contratto.

Prima di eseguire le query sul database ho creato una **VIEW**(tabella virtuale in SQL data dal risultato di una query) che mi permettesse di capire quando un contratto veniva chiamato tramite una transazione per la prima volta. per far questo viene cercato il timestamp più basso tra le transazioni che riguardano lo stesso contratto e viene salvato sulla view insieme all'indirizzo del contratto.

Come alternativa avrei potuto salvare direttamente il timestamp della transazione sulla tabella del contratto oppure, invece di considerare la prima transazione che interagisce con lo Smart Contract, si sarebbe potuto utilizzare la transazione di creazione del contratto. Una transazione di tipo "contract creation" è caratterizzata da destinatario nullo,visto che non è diretta ancora a nessuno, e dall'indirizzo del nuovo contratto salvato nel campo receipt della transazione. Ho scelto di creare una View per evitare di fare delle modifiche all'ultima versione della libreria software utilizzata, visto che già alcune migliaia di blocchi erano stati già caricati sul database universitario e ricominciare da capo la trascrizione avrebbe comportato uno spreco elevato di tempo.

```
CREATE TEMP VIEW data_contratto(address,data) AS
```

```
SELECT DISTINCT C.address ,
   (TIMESTAMP 'epoch' + B.timestamp * INTERVAL '1 second')::TIMESTAMP
FROM Contract C, block B
        JOIN TxSummary Q ON Q.blockNumber = B.blockNumber
WHERE (Q.sender = C.address OR Q.receiver = C.address)
  AND B.timestamp <= ALL(
                SELECT B1.timestamp
                FROM Contract C1, block B1
                   JOIN TxSummary Q1 ON Q1.blockNumber = B1.blockNumber
                WHERE (Q1.sender = C1.address OR Q1.receiver = C1.address)
                AND C1.address = C.address
                );
```

Le query non sono lanciate su tutta la blockchain ma solo una picola parte. Questo perchè, per questioni di tempo, non sono riuscito a caricare tutti i blocchi presenti sulla blockchain di Ethereum, i blocchi presi in considerazione sono stati scelti a campione e sono i blocchi: 0-42000, 2000000 - 2001199 , 6000000 - 6000099.
la prima query calcola la media dei token ERC20 forniti dai token contract presenti sul database. la media è calcolata sui giorni. La query legge tutti contratti presenti sulla tabella virtuale contract e fa il JOIN con data_contratto utilizzando come condizione di JOIN l'uguaglianza tra gli indirizzi del contratto, non vengono considerati i contratti che non hanno mai fornito token perchè si tratta di contratti normali e in questo momento non ci interessano.

```
SELECT   DC.data::DATE ,  AVG(C.tokenTotalSupply),count(*)
FROM   Contract C
        JOIN   data_contratto DC ON DC.address = C.address
WHERE C.tokenTotalSupply <> 0
GROUP BY DC.Data::DATE
```

Tempo di esecuzione : 16.384 s

Data	Token	numero
2018-07-20	$767694516834294 \cdot 10^{60}$	401
2016-08-02	$5028219084126572 \cdot 10^{60}$	13
2016-08-03	$19281831982520778 \cdot 10^{60}$	13

Tabella 5.1: risultati sulla fornitura media di token

Il tempo di esecuzione è molto alto vista la mole di dati da analizzare e l'ordine di grandezza dei token, dai risultati vediamo che la fornitura media dei token è molto alta visto che abbiamo preso in considerazione contratti che sono da parecchio tempo sulla piattaforma e hanno generato parecchi token,

se avessimo preso in considerazione contratti più recenti avremmo ottenuto numeri di ordini di grandezza decisamente più piccolo.

In questa query invece andiamo a contare quante parole contiene il codice di ogni contratto, non vengono considerati i contratti con 0 caratteri perché nell'effettivo non esistono, sono semplicemente contratti i cui sviluppatori hanno deciso di non pubblicare il codice, però sulla blockchain sono salvati e forse sono anche in esecuzione. Anche qui consideriamo la media per giorno.

```
SELECT DC.data::DATE, AVG(C.codeSize), COUNT(*)
FROM   Contract C
           JOIN   data_contratto DC ON DC.address = C.address
WHERE C.codeSize <> 0
GROUP BY DC.Data::DATE
```

Tempo di esecuzione : 16.608 s

Data	caratteri	numero
2018-07-20	1499	551
2016-08-02	3151.3023255813953488	43
2016-08-03	1686.5416666666666667	24

Tabella 5.2: risultati sulla media di caratteri per contratto

Vediamo subito che i contratti del 20-07-2018 e del 3-08-2016 presentano un numero simile di caratteri mentre i contratti del 2-08-2016 ne hanno circa 1500 in più.

L'ultima query conta quante funzioni ha uno Smart Contract, sono contate solo le funzioni di Smart Contract scritti in vyper o solidity e, come fatto prima, consideriamo sempre la media per giorno.

```
SELECT   DC.data::DATE ,  AVG(C.functionNumber ) ,COUNT(*)
FROM   Contract C
           JOIN   data_contratto DC ON DC.address = C.address
WHERE C.functionNumber <> 0
GROUP BY DC.Data::DATE
```

Tempo di esecuzione : 16.255 s
Come per il numero di caratteri anche qui il primo e il terzo giorno presi in esame presentano valori simili, i 36 contratti del 2-08-2016 tuttavia un numero medio di funzioni che si avvicina a quello degli altri due.

Data	**caratteri**	**numero**
2018-07-20	34	551
2016-08-02	25	36
2016-08-03	32	24

Tabella 5.3: risultati sulla media di funzioni presenti in un contratto

Capitolo 6

Conclusioni

Lo studio della piattaforma Ethereum e della tecnologia blockchain è stato molto interessante, il sistema (nato appena nel 2015) è molto innovativo e, in futuro, potrebbe rivoluzionare completamente il mondo dell'informatica e della finanza per come lo conosciamo. Nonostante Ethereum sia nato da poco non è risultato troppo difficile trovare informazioni che riguardassero il suo funzionamento generale e come è organizzata la gestione degli Smart Contracts, i siti e i libri citati nella bibliografia sono piuttosto chiari ed esaustivi anche se alcune informazioni più specifiche che riguardano i contratti sono state difficili da reperire ed è stato necessario più tempo.

Per sviluppare il framework abbiamo cominciato a lavorare durante il periodo di tirocinio presso l'università, la parte di ricerca di librerie funzionali al progetto è stata parecchio dispendiosa a livello di tempo visti i numerosi tentativi falliti dati da software non adatti alle nostre esigenze. Trovare informazioni come il codice del contratto o la quantità di token scambiati non è stato immediato, questi sono campi che non vengono salvati direttamente sulla blockchain ma possono essere ottenuti da full node che eseguono un'istanza dell'EVM e, di conseguenza, compliano ed eseguono tutti i contratti salvando nello stato il risultato di queste operazioni. Alla fine si è deciso di prendere queste informazioni dal block explorer Etherscan tramite delle API fornite proprio da quest'ultimo, chiaramente prenderle dalla blockchain sarebbe stato molto più performante che ricavarle da un sito esterno. Non è stato necessario studiare nuovi linguaggi di programmazione visto che il framweork risulta tutto scritto in Python, come parte finale della tesi,per completare il discorso sugli Smart Contracts, si sarebbe potuto scrivere un esmpio di contratto in solidity. Alcune delle difficoltà affrontate durante lo sviluppo del software sono date anche dal fatto che il sistema Ethereum è in continuo aggiornamento e sviluppo, quindi anche i vari software e articoli online potrebbero non essere al passo con la verisone attuale di Ethereum.

Siccome la blockchain di Ethereum conta più di 10 milioni di blocchi, caricare tutta la blockchain sul database entro la data di consegna della tesi

sarebbe stato impensabile, ho deciso quindi di caricare qualche campione di blocchi per verificare che il software sviluppato funzionasse e svolgesse il compito richiesto, per le query lanciate sarebbe stato meglio avere a disposizione tutta la catena, in modo da ottenere un risultato statistico accurato su una grossa quantità di dati a disposizione.

Lo sviluppo del modello del database e, in seguito, l'implementazione del database stesso non ha dato grossi grattacapi visto che un lavoro simile è stato svolto durante il corso di Basi di Dati seguito presso l'Università di Verona. Il progetto non è sicuramente completo, però ritengo che un lavoro più attento ed accurato possa essere svolto una volta che Ethereum passerà ad ETH 2.0, scaricando solo la vecchia blockchain di Ethereum prima del passaggio definitivo al nuovo protocollo. Così facendo si avrebbe la vecchia catena completa prima dell'aggiornamento.

Bibliografia

[1] https://Ethereum.org/en/learn

[2] https://www.Ethereum-italia.it/white-paper

[3] https://academy.binance.com/it

[4] https://blockgeeks.com/guides/Ethereum-gas

[5] https://etherscan.io

[6] https://medium.facilelogin.com/the-mystery-behind-block-time-63351e35603a

[7] https://www.ilsole24ore.com/art/smart-contract-cosa-sono-e-come-funzionano-clausole-blockchain-ACsDo2P fintastico.com

[8] https://medium.com/validitylabs/how-to-interact-with-the-Ethereum-blockchain-and-create-a-database-with-python-and-sql-3dcbd579b3c0

[9] https://it.ihodl.com/tutorials/2018-03-26/token-erc-20-guida-ico-Ethereum-tutorial

[10] https://www.blockchain4innovation.it/criptovalute

[11] https://www.theblockchainmanagementschool.it/introduzione-alla-crypto-Ethereum

[12] https://solidity.readthedocs.io/en/v0.7.1/introduction-to-smart-contracts.html

[13] https://blockgeeks.com/guides/smart-contracts

[14] https://it.wikipedia.org/wiki/Smart_contract

[15] https://bmcmedgenomics.biomedcentral.com/articles/10.1186/s12920-020-00732-x

[16] https://www.ecovicentino.it/blog/Ethereum-smart-contract-funzionano

[17] https://Ethereum.github.io/yellowpaper/paper.pdf

[18] https://solidity.readthedocs.io/en/v0.7.1

[19] https://www.coindesk.com

[20] https://Ethereumclassic.org

[21] https://Ethereum.stackexchange.com

[22] https://www.reddit.com/r/ethdev/comments/ipzlrx/everything_you_ever_wanted_t

[23] https://Ethereum.stackexchange.com/questions/3417/how-to-get-contract-internal-transactions

[24] https://education.district0x.io/general-topics/understanding-Ethereum

[25] https://kauri.io/connecting-to-an-Ethereum-client-with-java-eclipse/b9eb647c47a546bc95693acc0be72546/a

[26] https://github.com/Great-Hill-Corporation/trueblocks-core

[27] https://quickblocks.io/

[28] https://github.com/xiaoyao1991/presto-Ethereum

[29] https://pypi.org/project/psycopg2/

[30] https://medium.com/@ASvanevik/why-all-these-empty-Ethereum-blocks-666acbbf002

[31] http://ethslurp.com/

[32] https://github.com/Impetus/Kundera/wiki/Getting-Started-in-5-minutes

[33] https://github.com/ConsenSys/eventeum

[34] https://medium.com/coinmonks/solution-to-extract-data-from-Ethereum-52d0b8007d1b

[35] https://ethdocs.org/en/latest/contracts-and-transactions/contracts.html#Ethereum-high-level-languages

[36] https://web3py.readthedocs.io/en/stable/contracts.html

[37] https://web3js.readthedocs.io/en/v1.3.0

[38] https://github.com/web3j/web3j

[39] https://consensys.net/blog/blockchain-explained/a-short-history-of-Ethereum

[40] Solomon, *Ethereum For Dummies*, For Dummies (5 aprile 2019)

[41] Lorenzo Foti, *Capire Blockchain: La guida in italiano per comprendere la tecnologia dietro Bitcoin e molte altre applicazioni che rivoluzionerà il futuro di Internet*, Independently published (16 novembre 2017)

[42] Lorenzo Foti, *Capire Ethereum Smart Contract ICO e DApp: Una panoramica sulle nuove tecnologie che stanno rivoluzionando internet e tanti esempi pratici della loro applicazione*, Independently published (4 gennaio 2018)

[43] MarkGates, *Ethereum: La guida definitiva che vi farà conoscere Ethereum, Blockchain, Contratti Smart, ICO e App decentralizzate. Include delle guide su come comprare Ether*, criptovalute e investire in ICO,Independently published (25 febbraio 2018)

[44] Andreas M. Antonopoulos, *Mastering Ethereum: Building Smart Contracts and DApps*, O'Reilly Media (13 novembre 2018)