Sviluppo di un tool per query alla blockchain Ethereum

UNIVERSITÀ DI CAMERINO

Autori
Matteo Cetraro
Massimiliano Sampaolo

Docenti
Barbara Re
Andrea Morichetta

Indice

1 Introduzione **3**

2 Background **4**
2.1 Blockchain 4
2.1.1 Blocchi 4
2.1.2 Transizioni 4
2.2 Ethereum 5
2.3 Web3.js 6
2.4 Angular 6
2.4.1 Typescript 6
2.4.2 UI 6
2.4.3 Business logic 6
2.4.4 Angular/RxJs 6

3 Progettazione ed implementazione **7**
3.1 Analisi dei requisiti 7
3.2 Scelte fatte 7
3.3 Descrizione progetto 7
3.4 Esempi di query 10
3.5 Problemi incontrati 11
3.6 Soluzioni adottate 11

4 Conclusione **13**
4.1 Sviluppi futuri 13
4.2 Conclusioni 13

Capitolo 1

Introduzione

Nell'ambito del Project ci siamo occupati della realizzazione di un tool per l'interrogazione e l'integrazione di dati residenti sulla blockchain Ethereum.

Il primo passo è stato quello di costruirsi un solido background teorico: siamo partiti dallo studio dal concetto di blockchain approfondendo i suoi principi e le sue caratteristiche. Siamo poi passati ad Ethereum cercando di capire le differenze con le altre piattaforme disponibili sul mercato, con relativi vantaggi e svantaggi.

Nello step successivo ci siamo dedicati alla parte più tecnica, abbiamo scelto Angular come framework di sviluppo, la libreria web3.js per l'accesso ai dati della blockchain e la libereria rxjs per la costruzione di uno stream di dati che viene filtrato progressivamente durante la query.

Il repository del progetto è disponibile al link:
https://github.com/MaxUnicam/QueryingEthereum

Capitolo 2

Background

2.1 Blockchain

La blockchain è una tecnologia che si basa sul concetto di Distributed Ledger Technology (DLT), ossia un registro distrbuito digitale. Alcune caratteristiche fondamentali:

- **decentralizzazione**: non c'è un single-point of failure, non ci si affida a servizi proprietari, essa è composta da nodi e chiunque può mettere in piedi diversi nodi molto semplicemente (per questo si legge spesso che la blockchain è "democratica")
- **trasparenza**: chiunque ha accesso ad ogni transizione mai registrata in questo registro distribuito
- **sicurezza**: ogni informazione ha un hash associato, questo permette di validare i dati ogni volta è necessario
- **immutabilità**: è molto complesso modificare una transizione salvata sulla blockchain (è necessario che la maggioranza dei nodi accettino questa modifica)

Letteralmente blockchain significa "catena di blocchi", infatti questo registro distribuito è composto concatenando una serie di blocchi, ovvero "pacchetti" di transizioni (e altri dati necessari a costruire una catena).

2.1.1 Blocchi

Questo 'libro mastro' cresce accodando nuovi blocchi (o record), ognuno dei quali fa riferimento al blocco precedente attraverso un hash. Ogni blocco, oltre che contenere una lista di transizioni, ha anche un header in cui sono presenti campi di gestione del blocco stesso. Alcuni dei campi più importanti contenuti nell' header sono:

- **Hash**: l'hash con cui identifichiamo il blocco
- **PrevHash**: è un hash che serve per fare riferimento al precedente blocco della catena
- **Merkle root**: hash di tutti gli hash di tutte le transazioni nel blocco
- **Timestamp**: rappresenta il time stamp dell'ultima transazione con un algoritmo conosciuto come Unix hex timestamp
- **Numero di transazioni**: indica il numero di transizioni contenute nel blocco

2.1.2 Transizioni

La transizione è forse l'entità più interessante della blockchain. Infatti essa definisce il trasferimento di informazioni da un determinato account ad un altro. Ad esempio nei bitcoin è il passaggio di valuta da un account ad un altro; in questo caso la transizione dovrà contenere i riferimenti agli account e la quantità di bitcoin scambiata.

2.2 Ethereum

ethereum

Ethereum è una piattaforma che permette a chiunque di costruire applicazioni decentralizzate eseguite sulla blockchain. É un progetto open-source, questo significa che nessuno è proprietario di Ethereum e che è stato sviluppato da persone di tutto il mondo.

Si può dire che Ethereum è una piattaforma perchè è stato progettato per essere flessibile e per adattarsi a tutti i contesti, infatti chiunque può fargli eseguire una propria applicazione, uno Smart Contract. Alla base di questa flessibilità c'è la EVM (Ethereum virtual machine) che è il componente che si occupa di eseguire gli smart contracts.

Gli smart contracts sono dei pezzi di codice salvati sulla blockchain e che vengono eseguiti a richiesta. Solidity è il linguaggio con cui è possibile implementarli, viene definito un linguaggio "contract-oriented". Dal punto di vista tecnico i contratti non sono altro che account: in Ethereum ci sono due tipi di account, gli EOAs (Externally owned accounts) che sono account associati a persone ed i Contract Account che sono governati dal loro codice interno ma debbono essere attivati da un EOA. Ogni account ha un saldo associato in ETH (la moneta ufficiale di Ethereum); nel caso di EOA il saldo viene aggiornato ad ogni transizione in entrata o in uscita, mentre nel caso dei Contract Account ad essi viene addebitata un costo per ogni volta che vengono eseguiti (si paga un costo computazionale).

Altra carta vincente di Ethereum sono le DAPP: sostanzialmente sono delle semplici applicazioni (mobile, web, desktop ecc) che interagendo con gli smart contracts vanno ad aggiungere transizioni sulla blockchain. Praticamente queste applicazioni usano la blockchain come un "server" decentralizzato con tutti i vantaggi conseguenti.

Creare un nodo di Ethereum è molto semplice, è sufficiente installare uno dei suoi client ed aspettare che si sincronizzi. Ci sono numerosi client, tra i più famosi: Parity, go-ethereum, cpp-ethereum, pyethapp, ruby-ethereum... Questi client mettono a disposizone diverse funzionalità tra cui l'accesso alla blockcahin, esponendo delle api da richiamare per leggere i dati della blockchain, la gestione delle DAPP ed il mining. Il mining consiste nella possibilità di mettere a disposizione la propria potenza computazionale per la blockchain: praticamente il nodo miner si occupa di ricevere, propagare, validare ed eseguire transizioni, di raggrupparle in blocchi e poi compete con altri miner per fare in modo che il prossimo blocco della blockchain sia quello appena creato. Il miner viene compensato con degli ETH per ogni blocco aggiunto.

2.3 Web3.js

Web3.js è una libreria Javascript che espone delle semplici api che ci permettono di interagire con Ethereum. Internamente la libreria non fa altro che mappare le proprie funzioni con le api esposte da un nodo Ethereum effettuando le relative chiamate RPC. Può far riferimento a nodi sia locali che remoti, è sufficiente cambiare opportunamente l'url in cui è esposto il nodo.

Progetto open-source disponibile su GitHub: https://github.com/ethereum/web3.js/

2.4 Angular

Angular è un framework per lo sviluppo di applicazioni, originariamente pensato per il web, si è poi esteso anche al mondo mobile e desktop. Anche questo è un progetto open-source al cui sviluppo/mantenimento partecipa in maniera molto importante, tra gli altri sviluppatori e aziende, il Team Angular di Google.

Angular rende più semplice creare applicazioni e lo fa combinando l'approccio dichiarativo per la definizione della UI a pattern tipici dei linguaggi object-oriented come la dependency-injection. A tutto questo aggiunge una serie di componenti e servizi già implementati come ad esempio tutti i componenti grafici che seguono le linee guida del Material Design di Google o il servizio http, molto utile per l'interazione con delle api di backend.

2.4.1 Typescript

Il linguaggio usato nello sviluppo con Angular è Typescript. Il typescript è un linguaggio compilato superset del javascript, il risultato della sua compilazione è proprio codice javascript minificato e che segue tutte le best practices. Quindi praticamente la nostra applicazione Angular al momento dell'esecuzione altro non è che un'enorme applicazione javascript.

L'uso del typescript mette a disposizione dello sviluppatore tutta una serie di utility tipiche dei linguaggi object-orented: classi, interfacce, ereditarietà... Permette quindi di ottenere applicazioni complesse con codice ben strutturato.

2.4.2 UI

In angular la ui è dichiarativa e si definisce usando l'html con l'aggiunta di alcuni costrutti e pattern specifici del framework come ad esempio il **Data Binding**. Il Data binding ci permette di creare un legame tra un componente grafico ed una nostra classe scritta in Typescript. Questo approccio è molto scalabile e permette di risparmiare molto codice in quanto è possibile modificare quanto mostrato a schermo soltanto modificando una variabile in memoria. Affinchè questo possa accadere angular effettua dei cicli di re-rendering alla modifica di determinate variabili. Per quanto riguarda la logica di iterfaccia del nostro applicativo è definita nei **Componenti**, che sono classi a cui applichiamo il decoratore **Component**.

2.4.3 Business logic

La logica di business è implementata nei **Servizi**, ovvero classi typescript che usano il decoratore **Injectable** di Angular. Come definito dal decoratore injectable i servizi possono essere iniettati attraverso la **Dependency Injection**. L'applicazione angular è organizzata in moduli ed almeno un modulo root deve esistere. Ogni modulo importa ed esporta una serie di componenti e servizi (oltre ad altri strumenti secondari).

2.4.4 Angular/RxJs

Angular ha aggiunto nelle ultime versioni il pacchetto RxJs: questa libreria permette l'uso del "Reactive programming", ovvero di un paradigma di programmazione asincrona che si occupa dello stream e propagazione di dati.

Capitolo 3

Progettazione ed implementazione

3.1 Analisi dei requisiti

L'obiettivo da raggiungere era quello di realizzare un tool che, attraverso una UI semplice e gradevole, ci permettesse di interrogare la blockchain in base a parametri, valori e regole arbitrarie.

3.2 Scelte fatte

Il primo strumento scelto è stato il framework di sviluppo: Angular; la scelta è stata semplice in quanto è uno dei migliori strumenti sul mercato, ha una grande community, è ben documentato è semplice da usare e permette di tenere il codice pulito e ben strutturato.

Successivamente abbiamo scelto web3.js come "accesso" alla blockchain; avevamo diverse alternative come effettuare le chiamate RPC esposte dal nodo Ethereum a mano ma questa opzione ci sembrava particolarmente costosa per i tempi di sviluppo, oppure avremmo potuto leggere i dati direttamente dal database locale del nodo ma in questo caso lo sviluppo sarebbe stato complesso, lungo e soprattutto avremmo ottenuto una soluzione estremamente "verticale" e non modulare. Un'altra opzione era quella di appoggiarci a servizi terzi come ad esempio Etherscan che espone alcune api per leggere dati dalla blockchain: pur garantendoci il vantaggio di non doverci occupare della gestione del nodo locale ci sembrava comunque una solzione non modulare ed inoltre queste api, essendo gratuite, sono filtrate e ci sono dei limiti di traffico da rispettare. Web3.js ci garantisce modularità, è molto semplice da usare e completamente compatibile con Angular.

L'ultimo strumento scelto è stato Parity che è uno dei numerosi client di Ethereum. É stato scelto perchè uno dei più famosi ed usati.

3.3 Descrizione progetto

Il primo passo dello sviluppo è stata la modellazione delle entità principali del dominio: account, blocchi e transizioni.

```
export interface Transaction {
  hash: string;
  nonce: number;
  blockHash: string;
  blockNumber: number;
  transactionIndex: number;
  from: string;
  to: string;
  value: BigNumber;
  gas: number;
```

```
  gasPrice: BigNumber;
  input: string;
}
```

Nel listato precedente possiamo vedere la modellazione delle transizioni. Alcune proprietà sono di tipo BigNumber: questo tipo è definito dall'omonima libreria javascript ed è usata quando è necessario lavorare con numeri a precisione arbitraria.

Dopo aver modellato l'ambiente sono stati implementati i servizi, ovvero quei componenti software che si occupano di implementare la logica di business. Ogni servizio implementa un'interfaccia per rendere il codice modulare e più facilmente usabile con la dependency injection. In questo caso sono stati implementati:

- **Localdataprovider**: il suo compito è quello di recuperare i dati dalla blockchain e fornirli, attraverso un observable, ad altri componenti software. Ecco i metodi più interessanti.

```
getBlocks(start: number, end: number): Observable<Block> {
  return new Observable((observer) => {
    for (let i = start; i < end; i ++) {
      setTimeout(() => {
        const block = this.web3.eth.getBlock(i) as Block;
        block.difficulty = this.normalizeEth(block.difficulty);
        block.totalDifficulty = this.normalizeEth(block.
            totalDifficulty);
        observer.next(block);
        if (i + 1 === end) {
          observer.complete();
        }
      }), this.delayInms);
    }
  });
}

getTransactions(start: number, end: number): Observable<Transaction>
    {
  return new Observable((observer) => {
    for (let j = start; j < end; j ++) {
      const block = this.web3.eth.getBlock(j) as Block;
      if (!block) {
        continue;
      }

      for (let i = 0; i < block.transactions.length; i ++) {
        setTimeout(() => {
          const hash = block.transactions[i];
          const item = this.web3.eth.getTransaction(hash) as
              Transaction;
          item.value = this.normalizeEth(item.value);
          observer.next(item);

          if (j + 1 === end && i + 1 === block.transactions.length) {
            observer.complete();
          }
        }), this.delayInms);
      }
    }
  });
}
```

In questo servizio viene usato web3.js mentre il resto dell'applicazione praticamente non ha idea di come i dati vengono recuperati. Possiamo notare anche una particolarità di web3.js: i valori di tipo BigNumber restituiti sono misurati in 'wei' e non in ETH, per questo viene effettuata la conversione dividendo per 1000000000000000000. Inoltre è possibile vedere che viene ritornato un observable e i dati vengono serviti in modo asincrono all'observer man mano che vengono recuperati.

- **Dataprojector**: Questo servizio invece si occupa di effettuare la 'proiezione', usando l'accezione tipica del mondo dei database, dei dati. Sostanzialente dato un oggetto e una lista di proprietà ritorna i valori di quelle proprietà dell'oggetto. Ecco il metodo più interessante di questo componente:

```
getValues(source: any, properties: string[]): any[] {
    if (!source || !properties) {
      return [];
    }

    const values: any[] = [];
    properties.forEach(prop => {
        values.push(source[prop as string]);
    });
    return values;
}
```

- **Dataselector**: Anche in questo caso il nome è preso in prestito dal mondo SQL dato che questa classe si occupa di effettuare la selezione dei dati; quindi si occupa di verificare se un determinato oggetto rispetta delle regole. Le regole in questione sono state formalizzate sotto forma di 'Constraint' in questo modo:

```
export class Constraint {
    property: string;
    operator: string;
    value: any;
    logicalOperator?: LogicalOperator = null;
}
```

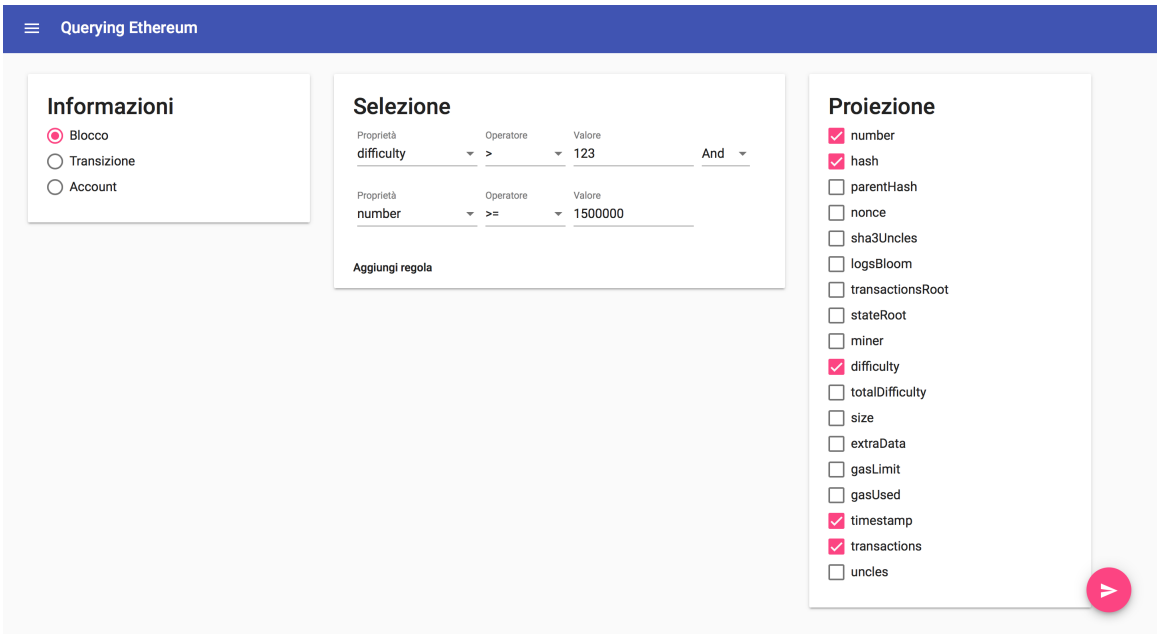
  Questo invece è il metodo che ricevuta una lista di oggetti ed una lista di regole, filtra solo gli oggetti che seguono le regole specificate.

```
filter(source: any[], constraints: Constraint[]): any {
    if (!source || source.length <= 0) {
        return [];
    }

    if (!constraints || constraints.length <= 0) {
        return source;
    }

    const result = [];
    for (let i = 0; i < source.length; i++) {
        const item = source[i];
        if (this.respectsAllConstraints(item, constraints)) {
            result.push(item);
        }
    }

    return result;
}
```

- **Camquerist**: L'ultimo servizio implementato è stato il servizio che si occupa di gestire e coordinare tutti quelli che abbiamo visto in precedenza. Praticamente recupera i dati dal data provider, li filtra con il data selector e li proietta affidandosi al data projector. La sua utilità è quella di fornire un unico punto di accesso per far partire la query nascondendo il coordinamento dei diversi servizi alle classi che lo utilizzano.

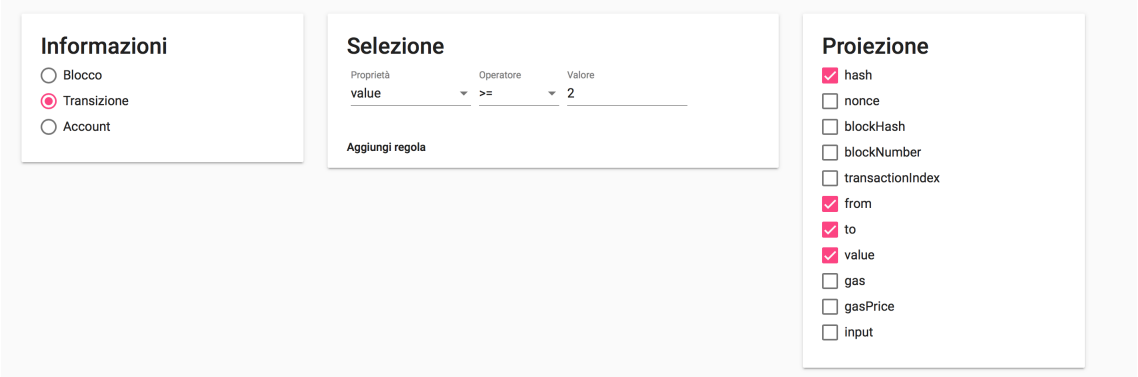
- **ReportGenerator**: In parallelo ai servizi per la costruzione e l'esecuzione delle query è stato progettato anche un semplice servizio che permettesse la generazione di report partendo dai risultati delle query stesse. Al momento è possibile effettuare un export di queste informazioni in formato .csv o .json, ma altri sarebbero semplici da implementare.

Per quanto riguarda la UI abbiamo usato i componenti di Angular/Material. Di seguito la schermata principale del nostro strumento in cui possiamo vedere diversi componenti tipici del Material design: si possono vedere la toolbar blu in alto, il burger-menu le cardview che contengono le diverse sezioni delle query ed infine il floating button per lanciare l'esecuzione dell'interrogazione.

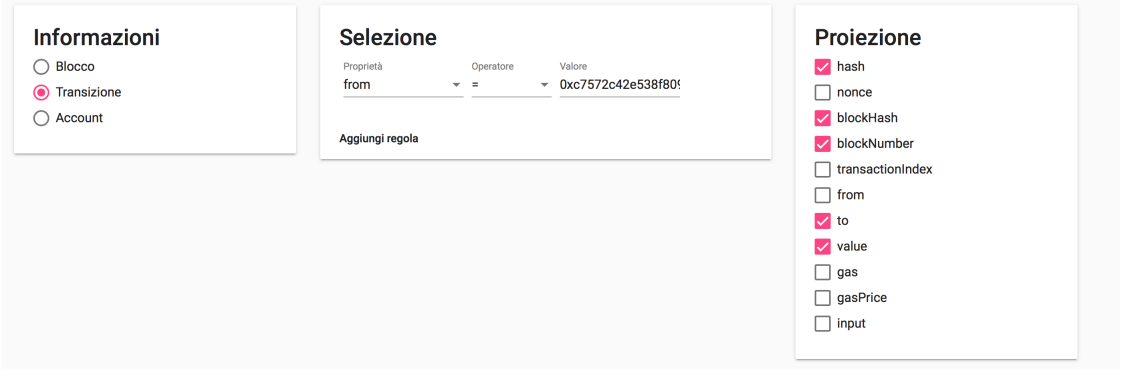


3.4 Esempi di query

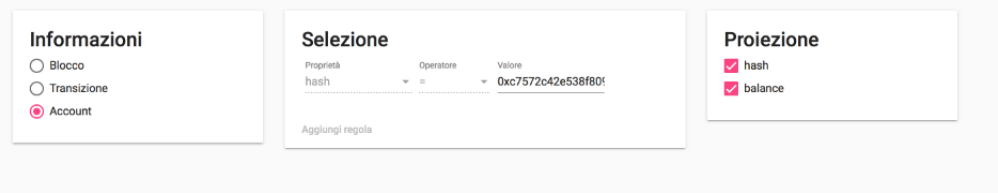
Ecco di seguito alcuni esempi di query che è possibile realizzare utilizzando questo tool:



In questa prima query si stanno cercando tutte le transizioni di importo maggiore o uguale a 2 ETH. Di queste trasizioni siamo interessati all'hash, da chi è partito l'importo e chi l'ha ricevuto, oltre che l'importo stesso.



Nell'immagine sopra vogliamo ottenere l'hash, il numero del blocco, il ricevente ed il valore delle transizioni che sono partite da un determinato account.



In questo ultimo esempio stiamo chiedendo ad Ethereum il bilancio di un determinato account.

Ovviamente questi sono soltanto esempi indicativi, è possibile realizzare un gran numero di query oltre a quelle illustrate.

3.5 Problemi incontrati

Dopo lo studio iniziale ed una prima implementazione grossolana del progetto abbiamo riscontrato un problema che avevamo trascurato: effettuare query all'intera blockchain è un'operazione molto costosa dal punto di vista computazionale e di conseguenza lenta (potenzialmente anche diversi giorni).

Indagando abbiamo stimato che usando web3.js ogni chiamata alla libreria impiega mediamente 100 ms: prendendo il valore da solo è un tempo più che accettabile il problema nasce dal fatto che web3.js mappando le chiamate RPC esposte dai nodi Ethereum fornisce delle api estremamente granulari e specifiche che non permettono alcuna aggregazione di dati. Questo significa che è necessario effettuare un gran numero di chiamate per poter ottenere dati aggregati: per dare un'idea della cardinalità delle chiamate necessarie basti pensare che per ottenere le transizioni di 50 blocchi sono necessarie (mediamente) diverse migliaia di chiamate (si può immaginare per 10000 o più blocchi).

Ovviamente questo non è un problema evitabile essendo una conseguenza della natura stessa della Blockchain ma abbiamo trovato una soluzione che ci permette di mitigare il probelma rendendo usabile lo strumento anche con tempi lunghi.

Ci siamo imbattuti poi in un nuovo problema. Andando a fare così tante operazioni al secondo la CPU della macchina che eseguiva l'applicazione raggiungeva presto il 100% di utilizzo. Questo oltre a rallentare il sistema operativo, surriscaldare la macchina fisica portava anche la nostra applicazione a freezarsi, ovvero a non rispondere più all'interazione con l'utente.

3.6 Soluzioni adottate

Per risolvere il problema della durata della query abbiamo valutato di tornare indietro sulla scelta di web3.js. Abbiamo rivalutato di nuovo le varie possibilità ancora una volta ma la scelta è restata la medesima in quanto ognuna delle altre possibilità aveva delle controindicazioni. L'unica soluzione percorribile sarebbe stata quella di creare un servizio per l'indicizzazione della Blockchain: avrebbe di sicuro migliorato di gran lunga le performance ma oltre ad essere un progetto piuttosto complesso (gestione degli errori, continua sincronizzazione con la blockchain, sincronizzazzione incrementale ...) significava realizzare una seconda applicazione il cui 'compito' esulava dagli obbiettivi del Project (non ci interessa indicizzare la Blockchain!).

La soluzione adottata invece è stata di natura puramente tecnica: accettando il fatto che query su tanti blocchi sono lente possiamo però comunque rendere lo strumento usabile mostrando all'utente i risultati parziali della query e dandogli feedback sul fatto che la query è ancora in esecuzione. L'utente sa che la query è in esecuzione attraverso un indicatore di lavoro e ma man mano che vengono trovati record che rispettano la query vengono mostrati all'utente. Questo migliora notevolmente l'esperienza e di conseguenza l'usabilità. Praticamente l'esecuzione della

query ora è uno streaming di dati sotto forma di pipe. Dal punto di vista tecnico questa modifica è stata possibile grazie all'uso degli Observable e della libreria Rxjs.

Il secondo problema invece è stato risolto utilizzando la funzione Javascript 'setTimeout'. Sostanzialmente invece che eseguire subito la chiamata a web3.js in modo sincrono la facciamo in un secondo momento dando modo alla CPU di fermarsi o dedicarsi ad altri processi.

Capitolo 4

Conclusione

4.1 Sviluppi futuri

Questo progetto può essere migliorato sotto diversi aspetti. Ad esempio nella costruzione della query si possono concatenare diversi Constraint però l'ordine in cui vengono eseguiti è lo stesso in cui vengono aggiunti. Pur essendo accettabile sarebbe sicuramente meglio costruire query complesse in cui l'ordine possa essere definito anche dall'utente attraverso l'uso delle parentesi. Questa modifica dal punto di vista tecnico è piuttosto semplice, è sufficiente costruire un albero invece di una lista, più complessa invece è la resa grafica, ovvero trovare un modo di aggiungere parentesi in modo arbitrario che sia anche semplice ed intuitivo.

Altra modifica interessante potrebbe essere quella di aggiungere una gestione degli utenti, questo permetterebbe di mantenere uno storico di query eseguite, i loro risultati, preferenze che durano per diverse sessioni ecc.

Bisognerebbe indagare e studiare in modo approfondito i Web Workers che sono strumenti che dovrebbero permettere il multithreading in Javascript. Potrebbero essere un ulteriore miglioramento alle prestazioni del tool sviluppato.

Un possibile sviluppo potrebbe essere la gestione di nuovi operatori che al momento non sono stati implementati: join, count, max, min, first ecc.

Dal punto di vista dell'utente invece sarebbe utile avere a disposizione degli strumenti per stoppare la query. Magari i risultati ottenuti fino a quel momento sono sufficienti, oppure si vuole produrre un report parziale. Sarebbe ancora più intrigante aggiungere la possibilità di riprendere l'esecuzione di una query stoppata in precedenza.

4.2 Conclusioni

Il progetto è risultato piuttosto 'challenging' sia per la necessità di acquisire competenze teoriche sui temi della blockchain e sia dal punto di vista tecnico, non tanto per l'uso di Angular ma per la necessità di trovare una soluzione a problemi che erano conseguenza della natura delle tecnologie che abbiamo usato.

Proprio per questo è stato ancora più interessante del previsto!