Implementazione di un'Architettura Lambda

Davide Massuda, Giulio Raponi

July 2021

Contents

1 Abstract

Con il rapido sviluppo del web, delle app mobili e dei dispositivi IoT, un enorme volume di dati viene creato ogni giorno. Questi dati non sono solo massivi, ma vengono anche generati rapidamente e con una varietà di formati diversi. Molte aziende hanno interesse nell'elaborare questi "big data" in tempo reale o quasi. All'interno di diversi domini applicativi, alcuni dati devono essere elaborati real time, mentre per altri è possibile utilizzare un'elaborazione batch offline. L'Architettura Lambda è una soluzione che unisce sia i vantaggi delle tecniche di elaborazione batch che quelli streaming. In questo lavoro, verrà approfondita la realizzazione di un'architettura Lambda per l'analisi e il monitoraggio di dati nell'ambito delle criptovalute utilizzando diversi software open source.

2 Introduzione

Con l'enorme crescita delle quantità di dati, ci sono state diverse innovazioni sia nella memorizzazione che nell'elaborazione dei big data. A causa di sorgenti come sensori IoT, interazioni nei Social Network, log delle applicazioni e i cambiamenti di stato dei database, il numero di dati da tenere in considerazione è cresciuto esponenzialmente. Questi stream rappresentano flussi di dati continui e senza limiti che richiedono molto spesso un'elaborazione quasi in tempo reale. Il campo dell'analisi dei dati sta crescendo immensamente per trarre preziose intuizioni da grandi quantità di dati grezzi (raw data). Al fine di ricavare informazioni in un sistema di dati, i sistemi di elaborazione risultano essere essenziali, come la necessità di disaccoppiare il calcolo dalla memorizzazione. I frameworks di elaborazione possono essere classificati in tre categorie: elaborazione batch, elaborazione dei dati stream e sistemi di elaborazione dei dati ibridi. L'elaborazione tradizionale dei dati batch dà buoni risultati ma con un'alta latenza. Al fine di ottenere risultati in tempo reale con bassa latenza, una buona soluzione è quella di utilizzare strumenti come, ad esempio, Apache Kafka accoppiato con Apache Spark o con altri sistemi di elaborazione (near) real time. Questi tipi di modelli streaming risultano essere ottimi in termini di alta disponibilità e bassa latenza, ma potrebbero non garantire un'elevata precisione. Nella maggior parte degli scenari, i casi d'uso richiedono sia risultati veloci che un'elaborazione approfondita dei dati. Questo progetto si focalizza su quella che viene chiamata Architettura Lambda, in grado di unificare sia l'elaborazione batch che quella quella streaming [MW15]. Nello specifico la Lambda Architecture si compone di tre strati principali:

- Batch Layer
- Serving Layer
- Speed Layer

In questa relazione viene presentata la realizzazione di un'architettura Lambda, attraverso la sperimentazione di una o più tecnologie che combinano Stream

Analysis e Batch Analysis [Kum20]. In particolare, si vuole realizzare un sistema in grado di estrarre dati da una fonte esterna e analizzarli in modo differente, attraverso l'utilizzo di strumenti che processino i dati in tempi diversi. A seguito della fase di data ingestion, il processo viene diviso in due rami: da una parte, i dati arrivano in flussi e vengono elaborati man mano che arrivano, nell'altro invece, i dati vengono memorizzati "as is" in un dataset persistente e vengono poi applicate delle elaborazioni. Successivamente questi vengono mostrati ed eventualmente consumati.

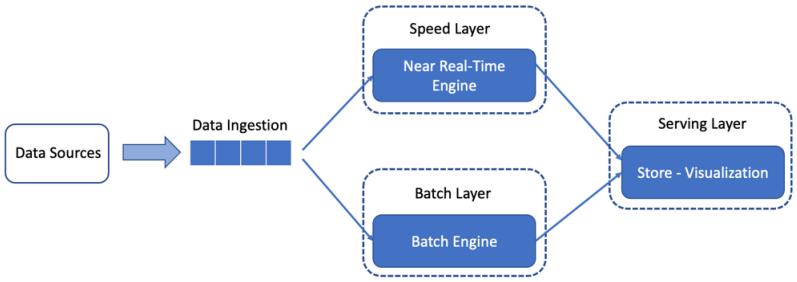


Figure 1: Generic Lambda Architecture

3 Architettura

La realizzazione di questa architettura ha compreso l'utilizzo di un cluster composto da 6 nodi. Il tutto è stato possibile grazie a quattro nodi Raspberry pi3 e due personal computer. Di seguito riportiamo le specifiche tecniche:

- 4x Raspberry pi3: 1.2 GHz ARM Cortex-A53 CPU
- 1x Asus VivoBook S15: Intel Core i7-8565U Processor 1.8 GHz, 16GB di RAM, GPU GeForce MX150
- 1x MacBook Air: 1,1 GHz Quad-Core Intel Core i5, 8 GB di RAM, Intel Iris Plus Graphics 1536 MB

4 Scelta dei Framework

In questa sezione vengono descritte le tecnologie utilizzate per la realizzazione del progetto. Nello specifico, per quanto riguarda l'ingestion dei dati, si è deciso di fare affidamento su Apache Kafka, un sistema di messaggistica Publish-Subscribe in grado di gestire code di flussi provenienti da varie sorgenti. In Kafka è possibile individuare due attori principali: Publisher che producono contenuti memorizzati in Topic e Subscriber, che si iscrivono a questi contenuti e ne usufruiscono nell'ordine e nella velocità che preferiscono. Il ramo relativo

all'analisi (near) real time, ovvero lo Speed Layer, è stato realizzato attraverso l'utilizzo di Apache Flink, un framework open source in grado di elaborare processi streaming con alto throughput e bassa latenza. Parlando invece della parte relativa al Batch Layer, il flusso di dati proveniente da Kafka è stato memorizzato in MongoDB. Quest'ultimo rappresenta un sistema di storage NoSQL aggregate-oriented di tipo document store, basato su paradigma Master-Slave. I dati così memorizzati sono stati poi analizzati attraverso l'utilizzo di Apache Spark, un sistema in grado di elaborare grandi moli di dati, basato sul paradigma di programmazione map-reduce e costruzione di DAG (*directed acyclic graph*). La memorizzazione degli output del livello Batch e del livello Speed, è stata affidata a ElasticSearch, un sistema open source scalabile, che permette di memorizzare, cercare e analizzare grandi volumi di dati rapidamente. Per quanto riguarda la visualizzazione delle informazioni ottenute ci si è affidati a Kibana, un'interfaccia utente open source che permette di visualizzare e navigare attraverso i dati provenienti da Elasticsearch.

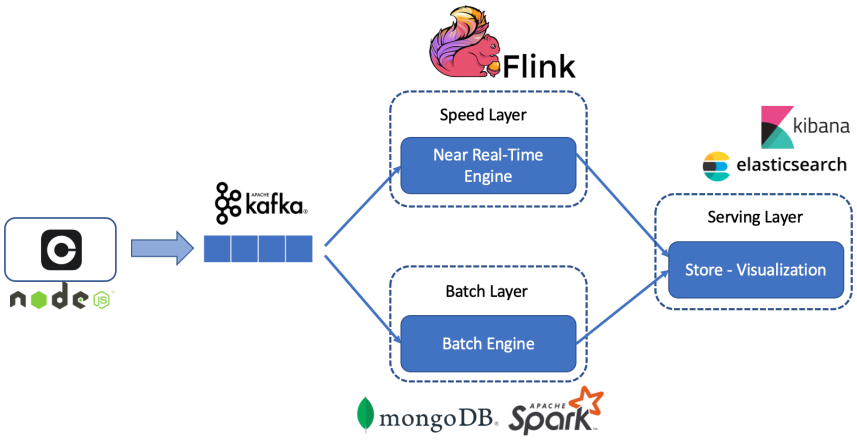


Figure 2: Lambda Architecture

4.1 Apache Kafka

Apache Kafka [Foub] è una piattaforma open source di stream processing scritta in Java e Scala e sviluppata dall'Apache Software Foundation. Il progetto mira a creare una piattaforma a bassa latenza ed alta velocità per la gestione di dati in tempo reale. Si concentra sulla prima fase della pipeline di analisi dei dati, ovvero la "Data Ingestion". Kafka nasce come sistema di messaggistica Publish-Subscribe ed è progettato per essere scalabile, veloce e durevole. L'intero lavoro di Kafka è quello di fornire un "absorber" tra il flusso di eventi e coloro che vogliono consumarli nell'ordine e nella velocità che preferiscono. In questo sistema possiamo individuare due attori principali: i producers, in grado di pubblicare messaggi (records) su uno o più topics, e i consumers, che possono

iscriversi ai topics e processarli (un topic può avere zero, uno o molti consumers iscritti ad esso). Kafka mette a disposizione quattro tipi di APIs:

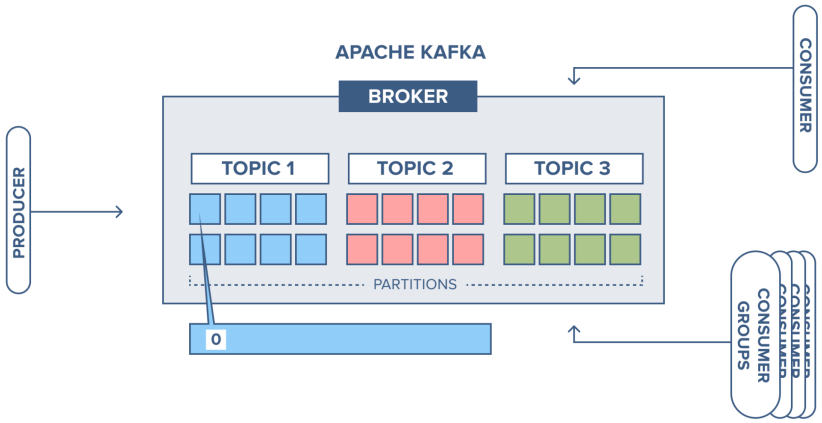


Figure 3: Apache Kafka partitions

- Producer API, che permette ad un'applicazione di pubblicare uno stream di record ad uno o più topic Kafka.
- Consumer API, per iscriversi ad uno o più topics e processare il flusso di record.
- Streams API per consentire il consumo di uno o più stream in input e produrre uno stream in output.
- Connector API in grado in permettere la connessione tra i topic e le varie applicazioni del sistema.

I principali vantaggi di questo sistema sono:

1. Bassa latenza: Apache Kafka offre un basso valore di latenza, poichè permette al subsriber di consumare i messaggi in qualsiasi momento.
2. Alta velocità di trasmissione: grazie alla bassa latenza, Kafka è in grado di gestire grandi volumi di dati ad alta velocità. Supporta inoltre migliaia di messaggi in un secondo.
3. Tolleranza ai guasti: una caratteristica essenziale di Kafka riguarda la resistenza ai guasti dei nodi all'interno del cluster introducendo un meccanismo di replicazione.
4. Durabilità: Kafka offre la funzione di replica, un servizio che permette ai dati o messaggi di persistere in modo duraturo su cluster.

5. Accessibilità: poiché tutti i dati vengono memorizzati in Kafka, diventano facilmente accessibili a chiunque.

6. Sistema distribuito: Apache Kafka contiene un'architettura distribuita che lo rende scalabile. Partizionamento e replicazione sono due aspetti fondamentali del sistema distribuito.

7. Gestione in tempo reale: Apache Kafka è in grado di gestire una pipeline di dati in tempo reale, che include processamento, analisi, archiviazione, ecc.

4.2 Apache Flink

Apache Flink [Foua] è un framework e motore distribuito per elaborare flussi di dati. Scritto in Java e Scala, supporta sia processi batch che streaming, grazie alla sua capacità di elaborare dati con alto throughput e bassa latenza allo stesso tempo. Si basa su *Dataflow Model* e supporta grazie alle *DataStream API*, sia *event time* che processi *out-of-order*. Sono presenti inoltre librerie per elaborazioni su grafi (batch), Machine Learning (batch) e processi basati su eventi (streaming). Supporta inoltre una integrazione con YARN, HDFS, HBase e altri componenti dell'ecosistema Apache Hadoop. Qualsiasi tipo di

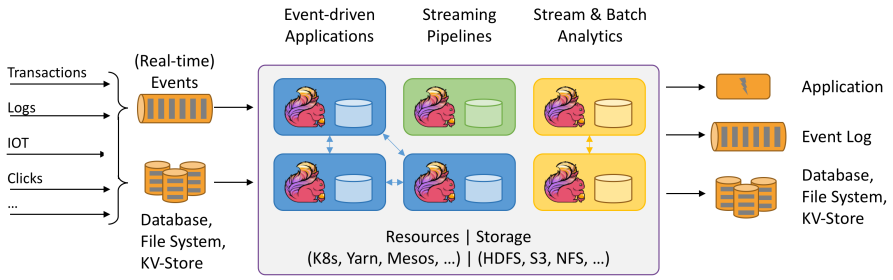


Figure 4: Flink Architecture

dato in Flink [Fab19] è prodotto come un flusso di eventi: Transazioni con carta di credito, misurazioni di sensori, interazioni utente su un sito Web e così via. Questi possono essere elaborati come flussi illimitati o limitati.
I flussi illimitati hanno un inizio ma non una fine; non terminano e forniscono i dati così come vengono generati. I flussi illimitati devono essere elaborati continuamente, e gli eventi devono essere gestiti prontamente dopo essere stati importati. Non è possibile attendere l'arrivo di tutti i dati in input poiché questo è illimitato. Questo tipo di elaborazione spesso richiede che gli eventi vengano inseriti in un ordine specifico, ad esempio l'ordine in cui si sono verificati gli eventi, per poter ragionare sulla completezza dei risultati.
I flussi limitati invece, hanno un inizio e una fine definita e possono essere elaborati inserendo tutti i dati prima di eseguire qualsiasi calcolo. Non risulta

importante l'ordine con il quale vengono importati i flussi poiché questi possono essere ordinati secondo timestamp o altri parametri. Questo tipo di elaborazione è noto anche come elaborazione batch.

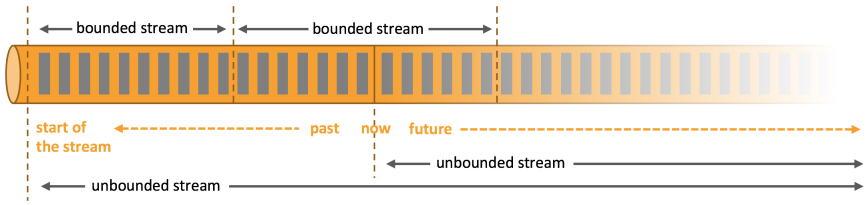


Figure 5: Flink Stream

4.3 MongoDB

MongoDB [Inc] costituisce un DBMS non relazionale, orientato ai documenti (aggregate oriented). Classificato come un database di tipo NoSQL, MongoDB si allontana dalla struttura tradizionale basata su tabelle dei database relazionali, in favore di documenti in stile JSON con schema dinamico (MongoDB chiama il formato BSON).
In questo sistema di storage ogni aggregato ha una chiave e un valore che identifica un documento, completamente visibile e in grado di essere interrogato sulla base dei suoi campi. Le query ad hoc, l'indicizzazione e l'aggregazione in tempo reale offrono efficaci modalità di accesso e analisi dei dati. MongoDB è concepito in origine come database distribuito; l'alta disponibilità, la scalabilità orizzontale e la distribuzione geografica sono quindi native e facili da usare. In questo particolare sistema di storage NoSQL i nodi seguono un'architettura detta *leader-follower*.
Grazie alla sua flessibilità MongoDB costituisce una scelta ottimale in molti casi d'uso, come ad esempio:

- Sistemi per la gestione dei prodotti: le grandi aziende che vendono una vasta gamma di prodotti, anche molto diversi tra loro, possono essere supportate in modo migliore da MongoDB piuttosto che da DB che utilizzano schemi più rigidi.

- Sistemi di gestione dei contenuti: i contenuti dei siti web possono essere molto eterogenei (testi, elementi di stile, immagini, altre risorse ...) quindi MongoDB è una scelta naturale di memorizzazione.

- Operational Intelligence: l'analisi in tempo reale e l'uso di intelligence operativa sono gestiti in modo efficace da MongoDB. Può memorizzare i log e, più in generale, i dati generati dalla macchina ed elaborare rapporti gerarchici in intervalli specifici / minuti / ore / giorni.

Il modello dei dati in MongoDB, come accennato in precedenza, è organizzato secondo alcuni livelli gerarchici: i) I *Database* corrispondono al più alto livello di astrazione. Molto simile ai database SQL, un database Mongo raccoglie idealmente tutti i dati che si riferiscono ad un singolo progetto/applicazione; ii) le *Collezioni* sono l'equivalente delle tabelle nei database SQL. Queste non hanno uno schema preimpostato: la stessa collezione può contenere due documenti con attributi diversi; iii) un *Documento* rappresenta un elemento, o un oggetto, nel dominio di interesse. Può essere visto come una collezione di attributi, ognuno dei quali è una coppia chiave-valore. I valori hanno dei tipi, semplici o complessi (liste, insiemi); anche la nidificazione è supportata.

Figure 6: MongoDB schema design

4.4 Apache Spark

Apache Spark [Fouc] è un framework open source per il calcolo distribuito, sviluppato da AMPlab della Università della California e successivamente donato alla Apache Software Foundation. Apache Spark SQL costituisce uno dei moduli offerti da Spark e consente un'elaborazione dei dati strutturati; basato su DataFrames, può anche fungere da motore di query SQL distribuito. Consente a query Hadoop Hive di essere eseguite fino a 100 volte più velocemente e fornisce inoltre un'integrazione con il resto dell'ecosistema Hadoop. Può essere eseguito in cluster Hadoop tramite YARN o con la modalità *standalone* di Spark e può elaborare dati in HDFS, HBase, Cassandra, Hive e qualsiasi Hadoop *InputFormat*. È progettato per eseguire sia elaborazioni batch (simile a MapReduce) che streaming, query interattive e Machine Learning. Un cluster Spark può essere eseguito su migliaia di nodi (il più grande conosciuto ne possiede circa 8000). Riesce a lavorare bene con dati di dimensione fino ai petabyte e risulta essere molto più veloce di Hadoop MapReduce anche avendo 1/10 delle macchine a disposizione. Le operazioni Spark sono eseguite su memoria e se questa non dovesse bastare, i dati vengono memorizzati momentaneamente su disco, permettendo di eseguire operazioni di grandezza arbitraria. Può essere eseguito in modalità standalone, o su YARN, Mesos o Kubernates.

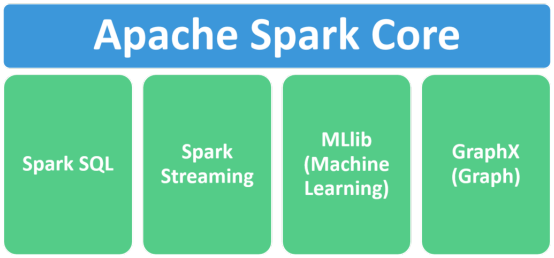


Figure 7: Apache Spark Core

4.5 Elasticsearch

Elasticsearch [NVa] è un sistema di ricerca scalabile, sviluppato in Java e basato su Lucene con supporto alle architetture distribuite. Le funzionalità sono accessibili tramite interfaccia RESTful, mentre le informazioni, sono gestite come documenti JSON. Questa architettura scalabile permette di dividere gli indici in shard, ognuno con una possibile replica. Elasticsearch consente di eseguire e combinare molti tipi di ricerche: strutturate, non strutturate, geografiche, nel modo che si preferisce. Permette inoltre di navigare tra i dati, selezionare l'intervallo temporale di interesse per recuperare tendenze (trend) e modelli. Con Elasticsearch è possibile memorizzare dati localmente o su ambiente distribuito, remoto, o in Cloud per avere sempre spazio a disposizione.

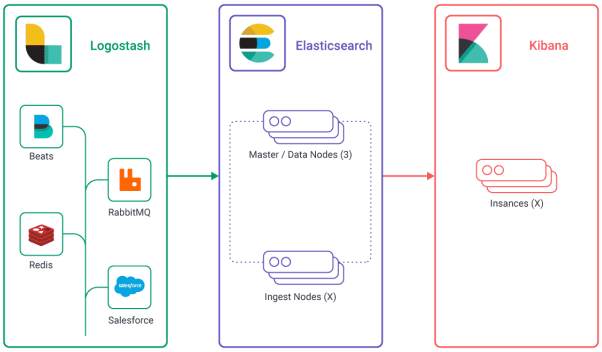


Figure 8: Flusso dati Elasticsearch

4.6 Kibana

Kibana [NVb] è un tool di "data visualization" che ci consente di interrogare il contenuto del cluster di Elasticsearch ed estrapolare le informazioni sotto forma di grafici o valori tabellari, consentendone una facile lettura. Grazie a questo sistema è inoltre possibile definire quelle che prendono il nome di "dashboard";

una dashboard di Kibana mostra una collezione di visualizzazioni e ricerche. E' possibile organizzare, ridimensionare e modificare il contenuto di questa e salvarla in modo da poterla condividere all'occorrenza.

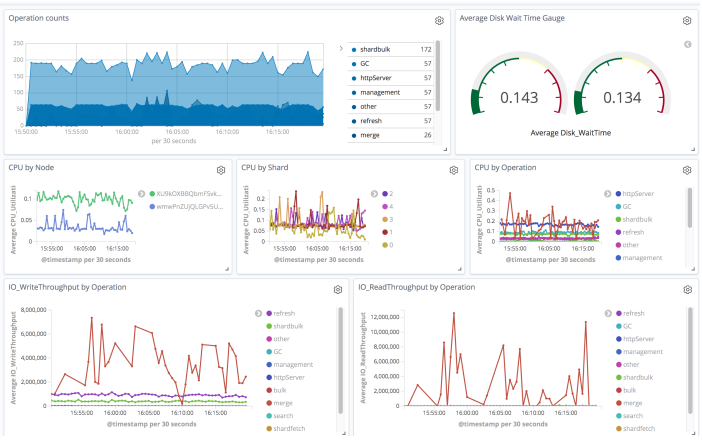


Figure 9: Kibana Dashboard

4.7 Docker

Docker [Doc] è un progetto open source che automatizza il processo di deployment di applicazioni all'interno di contenitori software, fornendo un'astrazione aggiuntiva grazie alla virtualizzazione a livello di sistema operativo di Linux. Un container è un ambiente software in grado di eseguire e isolare dall'esterno l'esecuzione di processi e applicazioni; ha dunque l'obiettivo dello sviluppo di un contenitore facilmente trasferibile che permetta l'esecuzione di una o più applicazioni al suo interno. La separazione data da questo involucro permette alle applicazioni eseguite al suo interno di svolgere i loro compiti in maniera sicura e senza che si debbano preoccupare dell'ambiente esterno. Gli strumenti per la creazione di container, come Docker, consentono il deployment a partire da un'immagine. Ciò semplifica la condivisione di un'applicazione o di un insieme di servizi, con tutte le loro dipendenze, nei vari ambienti. Docker automatizza anche la distribuzione dell'applicazione (o dei processi che compongono un'applicazione) all'interno dell'ambiente containerizzato. E' presente inoltre un servizio offerto da Docker (docker Hub) in cui poter salvare e/o scaricare le immagini direttamente da internet, offendo così una repository di container Docker.

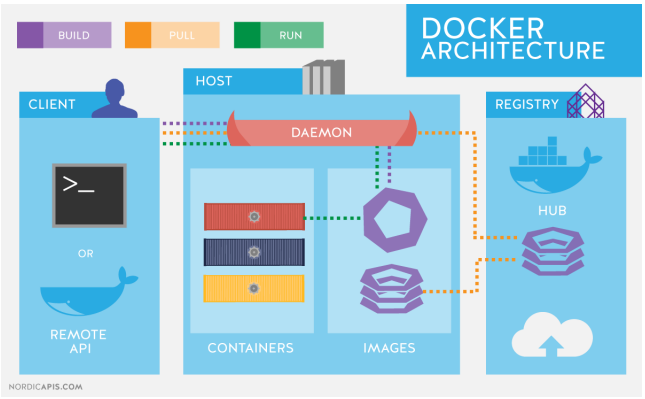


Figure 10: Docker [Nor]

5 Caso d'uso

La prima fase del progetto è stata la parte riguardante la scelta delle sorgenti informative. In particolare, si richiedeva l'utilizzo di sorgenti che garantissero un flusso continuo di dati e per questo si è scelto di utilizzare i dati provenienti dalla vendita e acquisto di BitCoin (ticker). Un exchange che mette a disposizione questo tipo di record è Coinbase Pro, una piattaforma di trading con diverse criptovalute (BTC, ETH, LTC, EOS...). I dati sono stati recuperati attraverso l'utilizzo delle API fornite dal sito stesso. Nonostante questo lavoro sia stato eseguito considerando una singola criptovaluta, il tutto potrebbe essere rieseguito analizzandone molteplici.
Si è quindi creato un producer Kafka che attraverso l'utilizzo di Nodejs, recuperasse i dati provenienti da Coinbase Pro e li inserisse in un topic Kafka. Dovendo gestire un'elaborazione batch e una (near) real time si è deciso di creare due consumer in modo tale da disaccoppiare le velocità di processamento dei dati. Di sotto si riporta un esempio di struttura ottenuta attraverso il producer Kafka.


```
{"type":"ticker","sequence":12037843028,"product_id":"BTC-USD",
"price":"26504.67","open_24h":"26960.72",
"volume_24h":"941.58122430","low_24h":"26371.64",
"high_24h":"27223.71","volume_30d":"42454.67276770",
"best_bid":"26501.30","best_ask":"26504.67","side":"buy",
"time":"2021-07-16T10:52:24.828693Z","trade_id":46746298,
"last_size":"0.00058888"}
```


5.1 Speed Layer

Per questo livello si è deciso di utilizzare Apache Flink come framework di elaborazione, in quanto mette a disposizione numerosi strumenti per analisi

(near) real-time. Questo rappresenta uno dei principali competitor di Apache Spark Streaming, framework approfondito durante il corso di Big Data. Si è scelto dunque di sperimentare un sistema non affrontato a lezione. Apache Flink fornisce un "connector" che gli permette di collegarsi al topic Kafka e consumare quindi i dati come subscriber. I dati ripresi da Kakfa, recuperati in formato Json, sono stati prima "parsati" per ottenere una struttura processabile e poi tramite *event time* disponibili in Flink, attraverso un aggregazione con una finestra temporale di un minuto, è stato creato un modello OHLC estraendo i valori di *open*, *close*, *high*, *low* per ogni minuto (open rappresenta il prezzo di apertura dell'intervallo, close l'ultimo prezzo riportato, high il prezzo più alto ed infine low quello più basso). Questi dati sono stati infine inviati ad Elastichsearch per una visualizzazione tramite Kibana. Di seguito viene riportato un esempio di dato inviato da Flink ad Elastiseach:

```
{high=31738.22, low=31685.25, close=31730.19, open=31691.14,
timestamp=1.626653039675E12}
```

5.2 Batch Layer

Per quanto riguarda invece l'analisi batch, si è creato, attraverso l'utilizzo di Nodejs, un Kafka consumer che riuscisse a recuperare i dati del topic e, attraverso l'instaurazione di una connessione con database MongoDB, li salvasse "as is" nel sistema di storage all'interno della collezione *coinbase*. Dopo aver eseguito con successo la memorizzazione dei dati, il passo successivo ha riguardato la creazione di una connessione tra il sistema di storage e il framework scelto per l'analisi batch, ovvero Apache Spark. Nello specifico, è stato deciso di fare affidamento sul modulo Spark SQL, costruito in cima allo Spark Core, in grado di consentire interrogazioni efficienti sfruttando una sintassi SQL. In particolare, le elaborazioni svolte sono state: recupero del prezzo minimo e massimo, calcolo della variazione percentuale, calcolo della distribuzione cumulativa, conteggio dei record salvati nel database. Gli output di queste elaborazioni sono stati successivamente passati ad Elasticsearch, ovvero il sistema responsabile della memorizzazione nello nello strato di servizio. Questo framework risulta molto intuitivo da utilizzare e configurare e grazie a ciò è stato possibile creare un flusso che collegasse diverse applicazioni.

5.3 Serving Layer

L'ultimo step di questa pipeline ha riguardato l'interrogazione dei dati e la creazione di un'interfaccia grafica che permettesse la visualizzazione delle informazioni. A tal proposito si è deciso di utilizzare Elasticsearch e Kibana, framework che permettono di analizzare grandi volumi di dati, grazie alla loro scalabilità, offrendo una rappresentazione ottimale delle informazioni. Elasticsearch offre la possibilità di definire quelli che prendono il nome di *index pattern*. Kibana richiede un index pattern per accedere ai dati Elasticsearch che si vogliono esplorare. Un index pattern seleziona i dati da utilizzare e permette

di definire le proprietà dei campi. In particolare, è possibile navigare tra i dati ed ottenere così informazioni riguardanti singoli intervalli temporali. Kibana implementa inoltre una dashboard interattiva che permette di organizzare, ridimensionare e modificare il contenuto di questa, senza la necessità di conoscere linguaggi di programmazione come CSS o HTML. E' possibile inoltre selezionare un intervallo temporale nel quale visualizzare i dati.

Dove possibile, sono state utilizzate immagini Docker per la creazione di container che potessero essere installate, eseguite e rimosse senza dover impostare ogni volta l'ambiente (ecosistema) di lavoro. Nello specifico sono state create *docker images* per Kafka e Zookeeper, Kafka producer, Kafka consumer, Elasticsearch e Kibana.
Di seguito si riporta un'immagine che rappresenta come i vari nodi siano stati utilizzati all'interno del progetto:

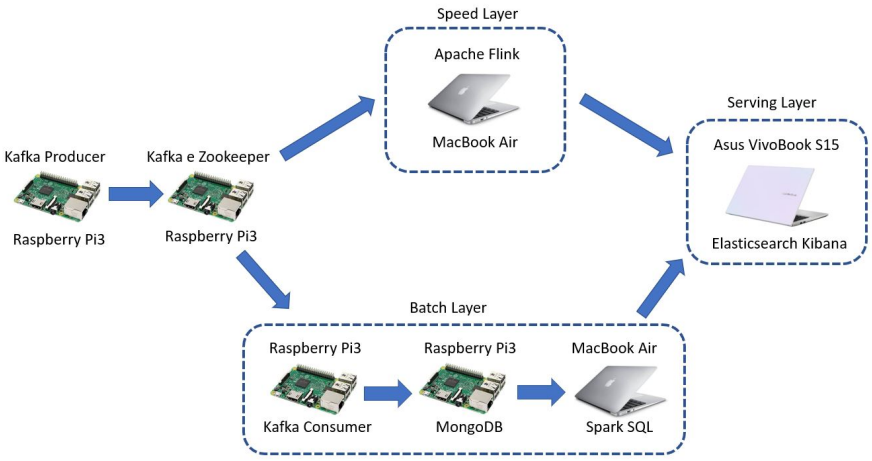


Figure 11: Cluster e Architettura

- Raspberry Pi3: Kafka Producer
- Raspberry Pi3: Kafka e Zookeeper
- Raspberry Pi3: Kafka Consumer
- Raspberry Pi3: Database MongoDB
- MacBook Air: Apache Flink e Spark SQL
- Asus VivoBook S15: ElasticSearch e Kibana

Nonostante il lavoro sia stato eseguito su cluster, in ambiente distribuito, è presente sulla pagina github del progetto una repository che permette di avviare il

tutto localmente, attraverso l'esecuzione di pochi step esplicitati nel README.

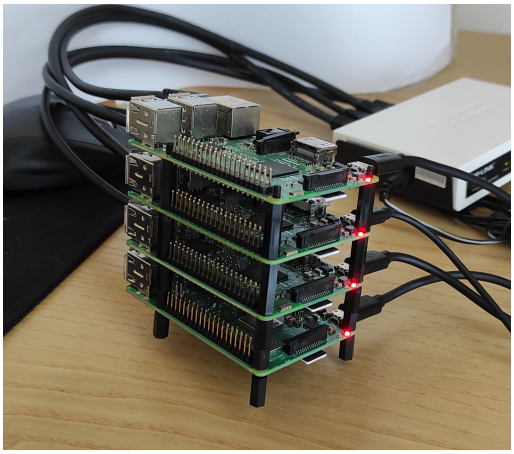
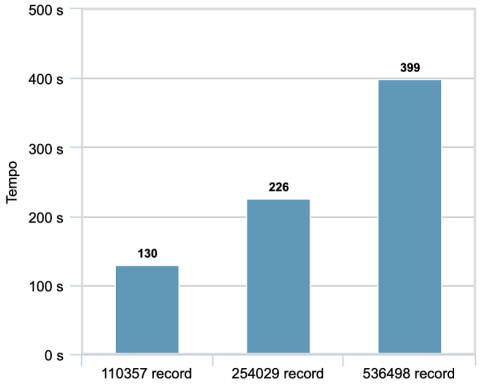


Figure 12: Quattro nodi del Cluster

6 Valutazione

Una volta terminata la progettazione dell'architettura, il sistema è rimasto operativo per una durata di 66 ore, processando in totale 536.500 record circa.
Per quanto riguarda la parte batch, questi dati sono stati analizzati giornalmente, al fine di ottenere statistiche sui tempi di esecuzione di Apache Spark SQL, al crescere delle dimensioni dell'input. Nella figura 13 vengono riportati una tabella e un grafico con i tempi e numero di record processati. I tempi



# Record	Tempo Esecuzione
110357	130 sec
254029	226 sec
536498	399 sec

Figure 13: Tempi Esecuzione

sopra riportati comprendono l'instaurazione di una connessione MongoDB, il recupero dei dati al suo interno, l'analisi e la memorizzazione su ElasticSearch. Parlando invece dell'elaborazione near real time, i dati inviati a Flink sono stati aggregati in intervalli temporali di un minuto, ottenendo così 30519 record ed estraendo un modello OHLC.
Si è provveduto poi alla memorizzazione dei dati, sia per quanto riguarda quelli batch, che streaming, salvati su framework Elasticsearch. La visualizzazione di questi è stata affidata a Kibana, attraverso il quale è stato possibile creare una Dashboard interattiva che permettesse di navigare tra i dati e creare grafici e tabelle con diverse caratteristiche.
Alcuni esempi di grafici implementati sono riportati di seguito:

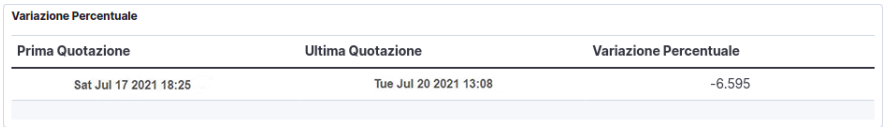


Figure 14: Variazione percentuale rispetto ai dati raccolti

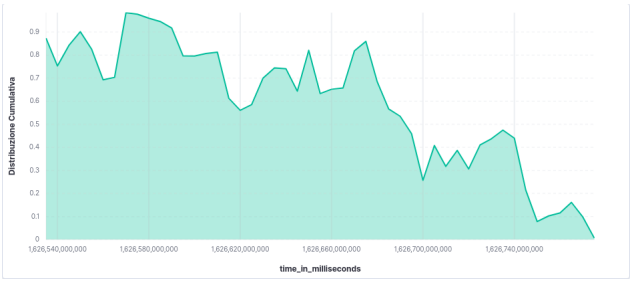


Figure 15: Distribuzione cumulativa

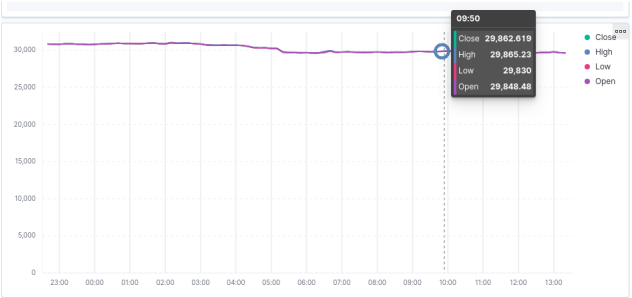


Figure 16: Andamento del bitcoin in tempo reale

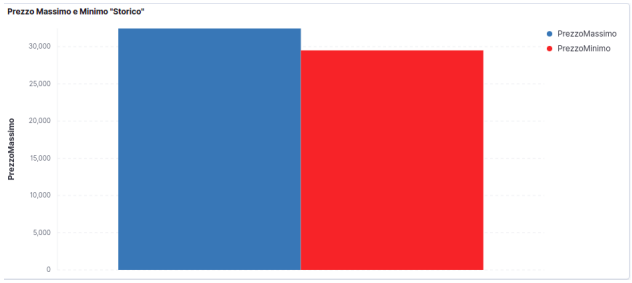


Figure 17: Prezzo minimo e massimo rispetto ai dati raccolti

7 Conclusioni e sviluppi futuri

In questo progetto è stata presentata la realizzazione di un'Architettura Lambda eseguita su ambiente distribuito, che consentisse il monitoraggio e l'analisi di dati provenienti dalla criptovaluta Bitcoin. In particolare, la pipeline ha riguardato da una parte un'elaborazione batch (eseguita attraverso l'utilizzo di database MongoDB e framework Apache Spark SQL), dall'altra un'elaborazione streaming (Apache Flink). Il tutto è stato poi reso disponibile attraverso sistemi come Elasticsearch e Kibana, il primo per lo storage dei dati, il secondo per la visualizzazione. Nonostante i dati raccolti siano inerenti ad un intervallo temporale di soli tre giorni, è stato possibile eseguire statistiche ed analisi che riuscissero ad indagare l'andamento del bitcoin. Con una quantità maggiore di dati a disposizione, uno sviluppo futuro potrebbe riguardare l'utilizzo di tecniche di Machine Learning in grado di prevedere l'evoluzione della criptovaluta nel tempo e fornire un migliore supporto alle decisioni. Sarebbe inoltre utile sperimentare l'utilizzo di più nodi all'interno del cluster, in modo tale da vedere come le varie tecnologie riescano a scalare e a distribuire i dati tra questi per gestire così una ripartizione ottimale delle risorse.

Bibliografia

[MW15] Nathan Marz and James Warren. *BigData: Principles and best practices of scalable realtime data systems*. Manning, 2015. ISBN: 9781617290343. URL: https://www.manning.com/books/big-data.

[Fab19] Vasiliki Kalavri Fabian Hueske. *Stream Processing with Apache Flink: Fundamentals, Implementation, and Operation of Streaming Applications*. Oreilly Associates Inc, 2019. ISBN: 9781491974292. URL: https://www.oreilly.com/library/view/stream-processing-with/9781491974285/?utm_source=dlvr.it&utm_medium=gplus.

[Kum20] Dr. Yuvraj Kumar. *Lambda Architecture – Realtime Data Processing*. 2020. URL: https://www.researchgate.net/publication/338375917.

[Doc] Inc Docker. *Docker*. URL: https://www.docker.com/. (accessed: 07.21.2021).

[Foua] Apache Software Foundation. *Apache Flink*. URL: https://flink.apache.org/. (accessed: 07.21.2021).

[Foub] Apache Software Foundation. *Apache Kafka*. URL: https://kafka.apache.org/. (accessed: 07.21.2021).

[Fouc] Apache Software Foundation. *Apache Spark*. URL: https://spark.apache.org/. (accessed: 07.21.2021).

[Inc] MongoDB Inc. *MongoDB*. URL: https://www.mongodb.com/. (accessed: 07.21.2021).

[Nor] NordicAPIs. URL: https://nordicapis.com/. (accessed: 07.21.2021).

[NVa] Elastic NV. *Elasticsearch*. URL: https://www.elastic.co/elasticsearch/. (accessed: 07.21.2021).

[NVb] Elastic NV. *Kibana*. URL: https://www.elastic.co/kibana/. (accessed: 07.21.2021).