POLITECNICO DI TORINO

Dipartimento di Automatica e Informatica

Corso di Laurea Magistrale in Ingegneria Informatica

Tesi di Laurea Magistrale

Progettazione di un sistema per l'analisi di dati real-time

Relatore:
Prof. Paolo Garza

Candidato:
Graziano Maggio

Anno accademico 2017-2018

Indice

Introduzione

Con l'avvento dell'Internet Of Things, il numero di dispositivi e delle apparecchiature connesse alla rete sta aumentando in maniera vertiginosa, basti pensare ai sensori per le automobili, elettrodomestici, telecamere o anche quelli utilizzati per il fitness. Tutti questi dispositivi elettronici scambiano continuamente una grande mole di dati sulla rete globale al fine di ottimizzare in tempo reale i processi produttivi. La diffusione di questo fenomeno, insieme ad altri fattori come l'utilizzo sempre più crescente dei social media, dove gli utenti pubblicano quotidianamente un'enorme quantità di dati come tweet, post, video, immagini, hanno messo in evidenza come le infrastrutture di calcolo tradizionali non soddisfino più in modo adeguato la richiesta crescente di capacità computazionale e storage, al fine di elaborare in tempi brevi questa "esplosione" di dati.
Esistono diversi casi significativi in cui non basta solo processare una grande quantità di dati, ma bisogna farlo il più velocemente possibile. I ritardi sulle previsioni di code di traffico si traducono in una maggiore perdita di tempo per gli automobilisti; rilevare transazioni fraudolente in tempo reale permetterebbe di fermarle prima che queste siano completate; pubblicare annunci basati sulle attività più recenti degli utenti risulterebbero più interessanti, incrementando in tal modo le vendite.
Da sole, comunque, le tecniche di elaborazione di dati streaming non sono sufficienti a creare un sistema per l'analisi in tempo reale. Occorre infatti, disporre di uno storage adeguato che permetta di archiviare e interrogare i dati con il minimo ritardo possibile. Queste basi di dati devono essere capaci di memorizzare centinaia di terabyte di dati e di gestire milioni di richieste al giorno, rimanendo continuamente disponibili.
Tra i numerosi framework open-source utilizzati nel panorama dei Big Data, quello che storicamente ha riscontrato maggior successo è senz'altro Hadoop. Esso è formato da diversi moduli e permette di analizzare in parallelo enormi set di dati distribuiti su un cluster di macchine. Sebbene tale soluzione risulti ideale per processare grandi blocchi di dati, essa non permette di ottenere dei risultati di analisi in tempo reale, in quanto introduce una latenza elevata.
Nasce, quindi, la necessità di sviluppare nuovi sistemi, realizzati per mezzo degli strumenti e delle tecnologie più moderne, per l'elaborazione di dati real-time al fine

di produrre i risultati migliori nel minor tempo possibile, generalmente nell'ordine dei secondi.
L'obiettivo di tesi è quello di progettare e realizzare un'infrastruttura Big Data per l'analisi di dati streaming che permettesse non solo di acquisire flussi di dati generati continuamente, ad alta frequenza, da sorgenti esterne ma anche di ottenere i risultati in modo tempestivo. L'intero lavoro di tesi è stato svolto presso l'azienda Consoft Sistemi S.p.A con sede a Torino, grazie alla quale è stato possibile implementare e testare il sistema su un cluster di cinque macchine, residenti su Amazon EC2.
La tesi è suddivisa in cinque capitoli. Nel primo verrà affrontato lo stato dell'arte relativo all'ambito in cui si svolge la tesi: si partirà da una breve introduzione sul fenomeno dei Big Data, seguita da una presentazione sull'ecosistema Hadoop; verranno poi affrontati i requisiti principali da prendere in considerazione durante la progettazione di un sistema per l'acquisizione di dati in tempo reale e successivamente si illustrearanno alcune tra le più recenti architetture implementate per l'elaborazione di dati streaming. Nel secondo capitolo, verrà definita la fase di progettazione dell'infrastruttura: innanzitutto, si elencheranno le diverse funzionalità che il sistema dovrà garantire e dopo aver effettuato un confronto tra i vari framework open-source esistenti ai giorni nostri, verranno scelti quelli più idonei a svolgere i compiti richiesti, descrivendo inoltre alcuni esempi di funzionamento. Nel terzo capitolo, si affronteranno i passi necessari per il setup, la configurazione e l'integrazione dei vari servizi scelti durante la fase di progettazione; ognuno di essi verrà installato in modalità distribuita sul cluster di macchine residente su Amazon EC2, in modo tale da realizzare un'infrastruttura Big Data che fornisse una forte affidabilità e tolleranza ai guasti, allo scopo di garantire l'elaborazione continua di dati streaming. Nel quarto capitolo, invece, verrà affrontato un caso d'uso per testare il funzionamento del sistema implementato, discutendo i risultati ottenuti. In particolar modo, si è scelto di analizzare l'andamento dei prezzi di alcune azioni in tempo reale nel mercato finanziario e di predire i possibili valori futuri, utilizzando opportuni algoritmi di apprendimento automatico. Successivamente, verrà effettuato un confronto tra le previsioni ottenute e i valore reali, con l'obiettivo di valutare la qualità dei risultati ottenuti sottoponendo il modello di machine learning allo stream di dati. Nell'ultimo capitolo, infine, saranno esposte le conclusioni relative al lavoro di tesi svolto.

Capitolo 1

Stato dell'arte

1.1 Cosa sono i Big Data

Ai giorni nostri, il termine "Big Data" ha raggiunto un elevato grado di popolarità, sebbene spesso non vi sia un'idea chiara su cosa esso rappresenti realmente. Nel corso degli anni sono state proposte diverse espressioni per spiegare tale fenomeno, ma fino ad oggi non esiste una definizione accettata universalmente.
Nel 2001, Doug Laney, analista della celebre società Gartner, pubblicò uno studio sul modello delle"3V" [1] per definire le caratteristiche dei Big Data secondo tre dimensioni differenti. Tale modello comprende: Volume, per indicare la grande mole di dati generata dalle varie sorgenti, caratterizzata, durante gli anni, da una crescita esponenziale (basti pensare che nel 2020 si prevedono oltre 35 miliardi di Terabyte); Variety, per sottolineare le diverse tipologie dei dati utilizzate dalle applicazioni come immagini, video, e-mail, dati GPS e molto altro; Velocity, per definire sia l'elevata frequenza alla quale i dati vengono prodotti sia la velocità alla quale quest'ultimi vengono analizzati, con l'obiettivo finale di ottenere i risultati migliori nel più breve tempo possibile.
Successivamente, sono stati introdotti i concetti di Veracity [2] e Value [3]. Mentre il primo si riferisce alla qualità dei dati, includendo fattori come rilevanza, accuratezza e affidabilità, il secondo evidenzia la possibilità di estrarre valore dai dati allo scopo di ottenere opportunità di bussiness.
Nel 2011, il McKinsey Global Institute definì i Big Data come quel sistema che opera su enormi dataset, la cui dimensione va oltre la capacità degli strumenti software, forniti dai database tradizionali, per catturare, memorizzare, gestire e analizzare i dati[4].
Un'ulteriore definizione di Big Data è stata proposta da Andrea De Mauro, Marco Greco e Michele Grimaldi nell'articolo "A Formal definition of Big Data based on its essential features" [5]. Tale definizione descrive i Big Data come l'insieme delle

tecnologie e dei metodi di analisi, indipendenti dal campo di applicazione, necessari a trasformare un'enorme quantità di dati eterogenei, con l'obiettivo di estrarre delle dei legami nascosti utili alle società e alle imprese.
Il ciclo di vita dei Big Data può essere rappresentato da una catena di processi che step-by-step consente di trasformare la grande quantità di dati in input in un un nuovo flusso di informazioni in output. La figura 1.1 descrive una serie di fasi necessarie a estrarre valore dai dati all'interno di un sistema Big Data, definendo un modello diviso nelle seguenti attività principali: Data Acquisition, Data Analysis, Data Curation, Data Storage, Data Usage.

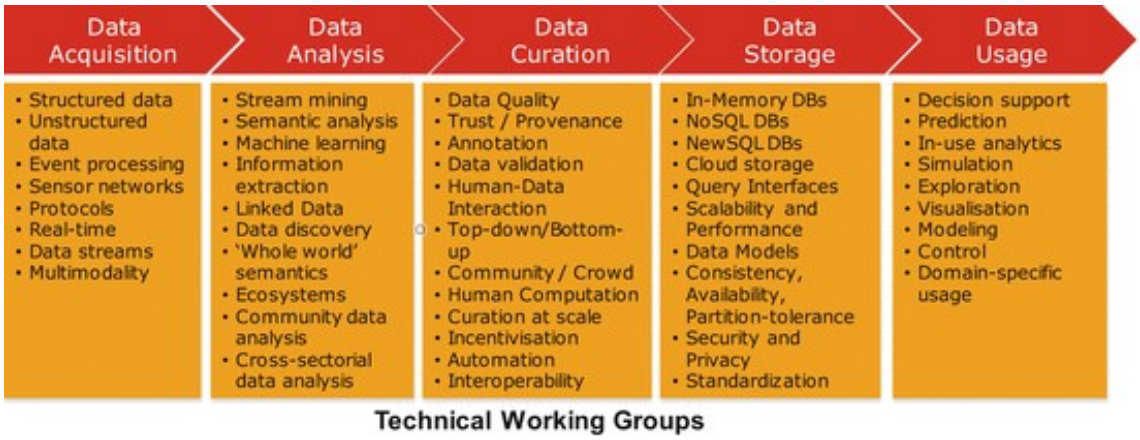


Figura 1.1. The Big Data Value Chain [6]

La prima attività comprende processi di raccolta e di pulizia dei dati prima che questi passino allo step successivo. E' uno dei principali requisiti da prendere in considerazione nella realizzazione di un'infrastruttura Big Data e deve permettere l'acquisizione sia di strutture dati dinamiche che di grandi volumi di dati mantenendo al tempo stesso una latenza bassa. Questa fase richiede protocolli che permettono di raccogliere informazioni di qualsiasi tipo (strutturate, semi-strutturate, non strutturate) generate da diverse sorgenti come IoT, social network (Facebook, Twitter), portali di e-commerce (Amazon, eBay), mercati finanziari.
La parte relativa al Data Analysis, invece, si occupa di trasformare e modellare i dati raw acquisiti con l'obiettivo di estrarre informazioni nascoste ma potenzialmente utili da un punto di vista economico. Questa attività include processi come machine learning, data mining, information extraction e data discovery. Gli algoritmi di machine learning, ad esempio, usano i dati per apprendere in maniera automatica e quindi eseguire attività importanti come predizioni, detenzione di anomalie e classificazione.
Durante la fase di Data Curation si vuole migliorare l'accessibilità e la qualità dei

dati a seconda del loro uso effettivo e garantire così il rispetto di determinati requisiti. Uno dei prinicipi chiave del data analytics è che la qualità di un'analisi dipende dalla qualità delle informazioni analizzate. Creazione del contenuto, selezione, classificazione e validazione sono alcuni dei processi che comprende.
Data Storage riguarda l'archiviazione e la gestione dei dati in maniera scalabile tale da garantire un accesso veloce ai dati alle applicazioni. Per molti anni, i database relazionali (RDBMS) sono stati l'unica soluzione utilizzata per il salvataggio su storage persistente, ma con la crescita del volume e la complessità della tipologia di dati raccolti si sono rivelati inefficienti da un punto di vista delle performance. I database NOSQL sono stati progettati per risolvere tali problemi; essi garantiscono un'elevata flessibilità dal punto di vista dello schema in cui vengono memorizzati e interrogati i dati.
Data Usage fornisce gli strumenti necessari per integrare i dati analizzati con le attività di bussiness. L'uso di tali informazioni nel decision making può migliorare la competitività di un'azienda grazie alla riduzione dei costi. Questa fase comprende attività di supporto alle decisioni, predizione, simulazione, visualizzazione ed esplorazione dei dati.
Per far fronte al fenomeno sempre più crescente dei Big Data, nel corso degli anni sono stati proposti diversi framework per gestire il grande volume di dati strutturati e non strutturati, generati dalle sorgenti di informazioni disponobili, come IoT e social media. Apache Hadoop è stata una delle prime e più importanti soluzioni adottate in tale ambito e trova un notevole riscontro nella maggior parte delle architetture Big Data utilizzate al giorno d'oggi.

1.2 Apache Hadoop

Software open source sviluppato sotto licenza Apache, ispirato dal Google File System (GFS) e dalla MapReduce di Google, Hadoop è un framework scritto in Java che fornisce gli strumenti necessari per il calcolo distribuito di grandi dataset suddivisi in cluster di computer garantendo allo stesso tempo scalabilità e affidabilità [**?**].
E' stato progettato per elaborare dati in maniera efficiente sia su un singolo server che su migliaia di macchine, utilizza una libreria che si occupa della gestione e del rilevamento dei guasti che possono verificarsi su ogni singolo nodo del cluster.
Il framework include diversi moduli tra cui:

- Hadoop Common: comprende un set di funzionalità comuni (I/O utilities, error detection, ecc..) insieme ad alcuni tool di configurazione necessari agli altri moduli di Hadoop.
- Hadoop Distributed File System (HDFS): file system distribuito progettato per memorizzare grandi quantità di dati in maniera ridondante su più nodi del

cluster in modo tale da garantire fault tolerant e high availability. Si basa su un'architettura Master Slave: mentre i DataNode si occupano di mantenere e recuperare i blocchi di dati, i NameNode gestiscono le richieste del client avendo una visione completa dell'organizzazione dei dati all'interno del cluster.

- Hadoop YARN ("Yet Another Resource Negotiator"): responsabile della gestione delle risorse del cluster, permette una separazione tra l'infrastruttura e il modello di programmazione. Quando occorre eseguire un'applicazione, viene fatta richiesta al ResourceManager, il quale è incaricato di trovare un NodeManager libero. Quest'ultimo si occuperà di avviare l'applicazione all'interno di un contenitore.

- Hadoop MapReduce: software per il processamento di grandi quantità di dati distribuiti su un cluster di nodi. Il dataset di ingresso viene diviso in chunck (tipicamente di 128 MB), i quali vengono elaborati in parallelo dalla funzione di Map che si occupa di organizzare i dati in coppie chiave-valore. Il risultato viene mandato in input alla funzione di Reduce che produrrà il risultato finale in output.

Hadoop è molto efficiente per il batch processing; come è possibilie notare dalla figura 1.2, nell'orbita di tale framework sono emersi numerosi progetti open-source, tra i quali ricordiamo: Apache HBase, database non relazionale, scalabile, distribuito, esegue su HDFS, permette un accesso ai dati veloce sia in lettura che in scrittura; Apache Pig, un motore per eseguire flussi di dati in parallelo su Hadoop, fornisce numerose operazioni sui dati (filter, join, sort, ecc..) che vengono definite attraverso script Pig Latin, eseguibili in uno o più job di MapReduce; Apache Hive, infrastruttura di data warehouse sviluppata da Facebook, fornisce un linguaggio SQL-like, chiamato HiveQL, per interrogare i dati su HDFS; Apache Spark, framework di analisi dati in un ambiente di calcolo distribuito, sviluppato all'Università di Berkeley e scritto in Scala, fornisce processamento in-memory per incrementare la velocità di elaborazione dati rispetto MapReduce, inoltre, supporta una ricca libreria per il machine learning, denominata Spark MLlib; Apache ZooKeeper, permette il coordinamento e la sincronizzazione tra vari servizi in un ambiente distribuito; Apache Solr e Lucene, servizi utilizzati per la ricerca e l'indicizzazione all'interno dell'ecosistema Hadoop; Apache Ambari, software per il provisioning, la gestione e il monitoraggio di un cluster Hadoop.
Infine, per quanto riguarda l'ingestion dei dati, Apache Sqoop permette di esportare su HDFS dati strutturati, come tabelle provenienti da database relazionali, mentre Apache Flume consente di acquisire dati semi-strutturati o non strutturati provenienti da varie sorgenti come file di log, network traffic e molto altro. A tal proposito, nel paragrafo successivo verranno discussi i principali requisiti che richiede un'architettura Big Data per consentire la corretta acquisizione di stream di dati.

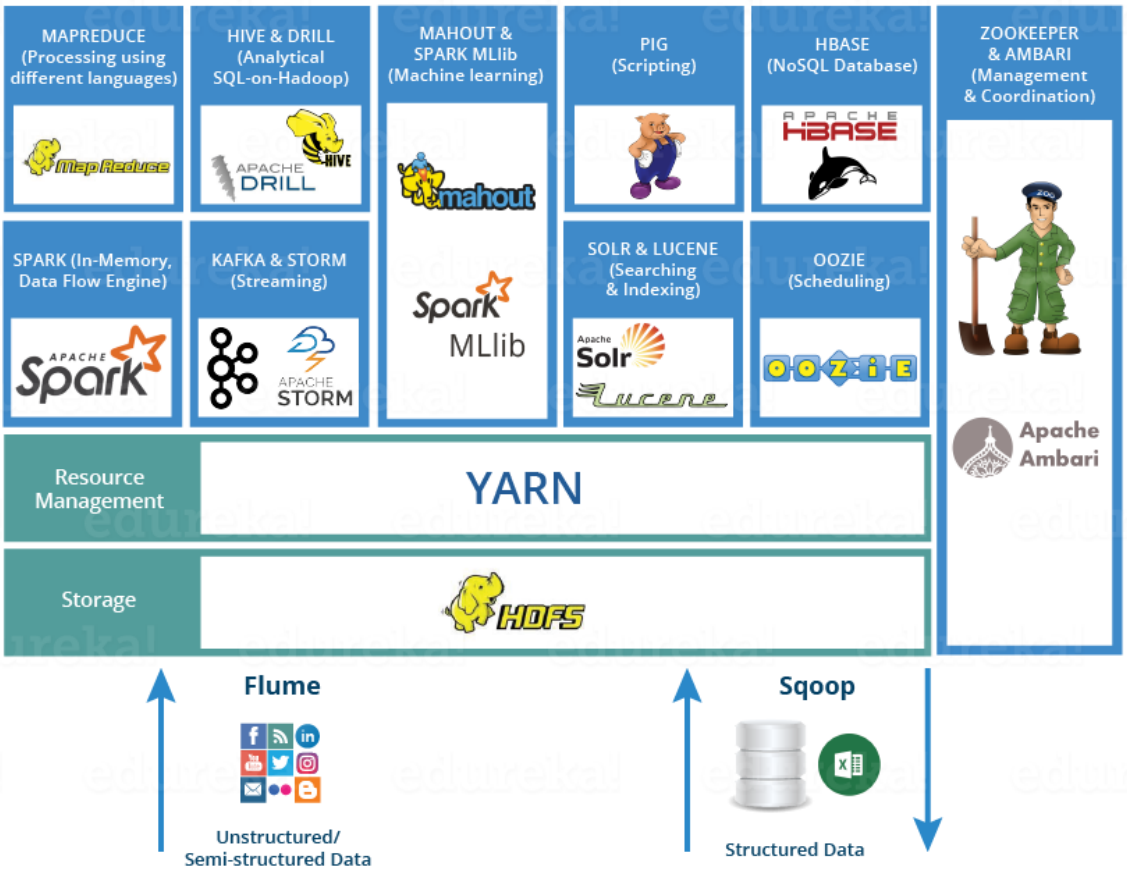


Figura 1.2. Hadoop Ecosystem [7]

1.3 Requisiti per l'ingestion real-time

Le applicazioni Big Data si stanno muovendo rapidamente da un modello di esecuzione batch-oriented verso un modello di esecuzione streaming con l'obiettivo di estrarre valore dai dati in tempo reale. Molto spesso pero questo non è sufficiente e alcuni servizi necessitano di aggregare il flusso di dati corrente con altri dati precedentemente raccolti e archiviati nel sistema allo scopo di migliorare la qualità delle informazioni ottenute.
Tipicamente un'infrastruttura Big Data per l'analisi real-time si basa su tre componenti principali: ingestion, processing e storage. Ogni strato è specializzato in un compito ben preciso e può essere indipendente dalla scelta degli altri [8]. L'ingestion si occupa principalmente di acquisire lo stream dati da sorgenti esterne al sistema prima di passare allo strato di processing in cui avviene l'analisi dei dati

vera e propria. A differenza della componente di storage, l'ingestion non permette di memorizzazione i dati in maniera permanente all'interno dell'architettura ma consente solamente un accesso temporaneo tramite l'uso di opportuni buffer di memoria. Inoltre, in alcuni casi, fornisce semplici operazioni di pre-processing dei dati prima che questi arrivino allo strato successivo.
Nella realizzazione di un'infrastruttura Big Data per l'ingestion real-time esistono diversi requisiti da prendere in considerazione:

- Data Size: si riferisce alla dimensione dello stream di dati (che può andare dalle decine di Byte fino a pochi Megabyte). Rappresenta una caratteristica importante per le performance dell'architettura poichè influenza il throughput del sistema.

- Data Access Pattern: definisce la modalità di accesso ai dati (ad esempio Apache Kafka tiene traccia dell'offset per decidere a quale record accedere, mentre Redis permette di utilizzare query multi-gruppo).

- Data Model: descrive la rappresentazione del flusso di dati (semplici record oppure set di righe in una tabella). Questa caratteristica influisce sul throughput del sistema che può essere migliorato utilizzando tecniche di compressione dei dati.

- Supporto al partizionamento del datastream: la suddivisione dello stream di dati in partizioni formate da sequenze di record, preferibilmente ordinati, è una tecnica che consente di migliorare scalabilità e throughtput.

- Supporto al message routing: permette di definire il percorso che uno stream di dati deve seguire, affinchè sia permessa l'analisi e l'eventuale memorizzazione dei dati su storage.

- Supporto al backpressure: consente di far fronte a situazioni in cui, in un intervallo di tempo, lo stream di dati arriva con una frequenza talmente elevata che il sistema non riesce più a sostenere. E' consigliabile utilizzare dei buffer in cui salvare i dati acquisiti temporaneamente in modo che il sistema possa recuperarli in un secondo momento.

- Latency vs throughput: definisce un trade-off da prendere in considerazione per migliorare le performance del sistema. L'ideale sarebbe ottenere una bassa latenza mantenendo allo stesso tempo un alto throughput, in modo da prendere le decisioni di bussiness migliori nel minor tempo possibile.

- Scalability: rappresenta la capacità del sistema di rimanere stabile a fronte di un incremento della quantità di dati in ingresso. La scalabilità orizzontale

si ottiene distribuendo il carico di lavoro tra i nodi del cluster, mentre si parlerà di scalabilità verticale quando si esegue un incremento delle risorse di elaborazione di una singola macchina.

- Availability: garantisce il corretto funzionamento del sistema anche in presenza di guasti, senza che vi sia nessuna perdita di grandi quantità di dati, in modo tale da garantire la corretta analisi di ogni dato acquisito dal sistema.

Nel paragrafo successivo saranno esaminate alcune piattaforme Big Data progettate per l'ingestion e l'analisi in real-time, facendo attenzione alle diverse componenti che vengono integrate all'interno dei vari sistemi.

1.4 Architetture per stream processing

Processare stream di dati in tempo reale rappresenta un aspetto cruciale per molte applicazioni, in quanto ottenere risultati nel minor tempo possibile equivale a prendere decisioni migliori. Il problema principale risiede nel fatto che la maggior parte dei sistemi big data attuali è basata sulla tecnologia Hadoop che sebbene rappresenti un framework altamente scalabile e fault-tolerant, non risulta appropriato per stream processing. Di conseguenza vi è la necessità di architetture che permettano di elaborare i dati in real-time fornendo bassa latenza.
Ai giorni nostri, esistono principalmente due pattern di architetture Big Data basate sul real-time processing, chiamate Lambda e Kappa.
La prima è un framework progettato da Nathan Marz, con l'obiettivo di creare un sistema ibrido che unificasse batch processing con real-time processing.
Come mostrato in figura 1.3, è possibile suddividere l'architettura in tre strati principali:

1. Batch layer: si occupa di archiviare su storage l'intero dataset, aggiungendo di volta in volta il nuovo stream di dati che arriva in ingresso al sistema, e di pre-calcolare determinate query functions, chiamate viste batch. Questo strato può essere implementato ad esempio da Hadoop, usando HDFS per memorizzare i dati in maniera persistente e Map Reduce per la creazione delle viste.

2. Serving layer: permette di ridurre la latenza del sistema, recuperando le informazioni contenute nelle viste batch tramite opportune query ad-hoc eseguite in real-time non appena arrivano i dati. Tipicamente questo livello dell'architettura viene realizzato tramite database NOSQL come HBase o Cassandra.

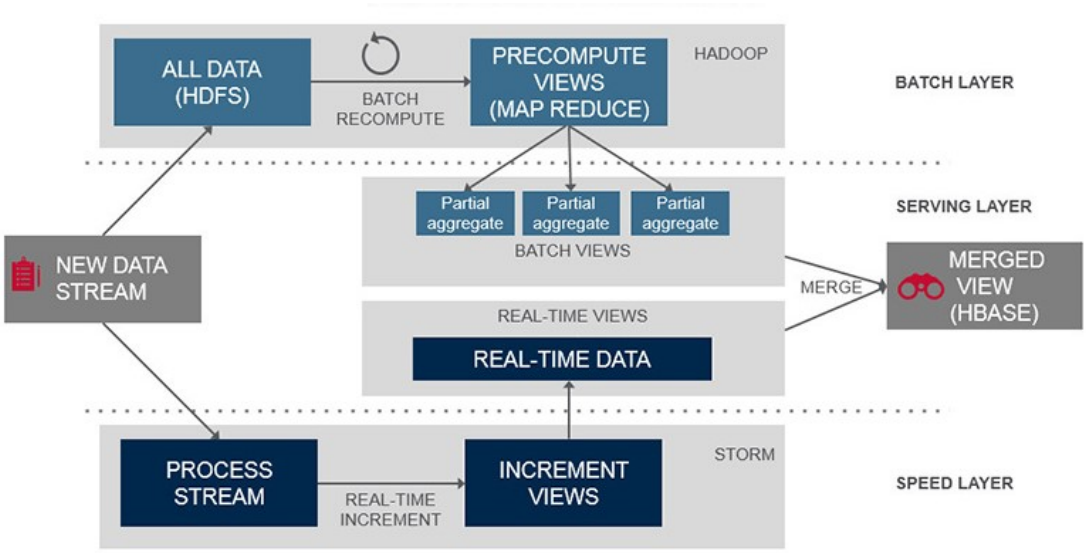


Figura 1.3. Lambda Architecture [9]

3. Speed layer: a differenza del batch layer, questo strato lavora solo con i dati più recenti e accetta solo richieste che sono soggette a una bassa latenza, usando algoritmi veloci per la generazione di viste incrementali. Le tecnologie di stream processing adoperate per questo strato possono essere ad esempio Apache Storm o Spark Streaming.

L'architettura Kappa, invece, è stata ideata da Jay Kreps nel 2014 ed è incentrata soltanto sullo stream processing dei dati. Può essere immaginata come una versione semplificata della Lambda Architecture in cui non è presente il Batch layer.
L'architettura Kappa è composta da due strati principali: Stream-processing layer che esegue operazioni non appena il flusso di dati arriva al sistema e il Serving Layer che viene usato per rispondere alle richieste del client interrogando i risultati ottenuti dallo stream process engine [10].
Sebbene entrambi i sistemi garantiscano scalibility e fault tolerant, l'architettura Kappa non utilizza uno storage persistente ma è più performante per quanto riguarda il processing dei dati in real-time. Nell'architettura Lambda, i risultati non sono immediatamente consistenti con i dati in ingresso e inoltre, la logica di bussiness è implementata due volte, sia nel Batch layer che nello Speed Layer, per cui gli sviluppatori necessitano di mantenere il codice in due parti separate del sistema.
Partendo dal confronto dei due sistemi, è stata proposta una nuova architettura che permettesse di unire i vantaggi esposti in precedenza [11]. Tale sistema presenta cinque strati principali: Integration Layer, si occupa di raccogliere i dati provenienti dall'esterno, indipendentemente dal loro formato; Filtering layer, realizza la fase di

pre-processing che si occupa di pulire i dati, elimando ad esempio campi inconsistenti; Real-Time processing layer, formato da Storm, usato per processare grandi quantità dati con una bassa latenza, e dal machine-learning, per acquisire continuamente conoscenza sui nuovi dati in ingresso; Storage layer, implemenatato tramite HBase per memorizzare i risultati ottenuti dalla fase di processing; Presentation Layer, utilizzato, infine, per mostrare i dati finali all'utente.
Un altro esempio di architettura Big Data per l'analisi dei dati in real-time viene presentato nell'articolo "Developing a Real-time Data Analytics Framework For Twitter Streaming Data" [12]. L'obiettivo è quello di analizzare in tempo reale migliaia di tweets generati dagli utenti ogni secondo utilizzando una piattaforma scalabile e distribuita. Il framework proposto consiste di tre sezioni: data ingestion, data processing e data visualization.
Come mostrato in figura, 1.4 prima parte dell'architettura è implementata tramite Apache Kafka e permette di recuperare i tweets interrogando direttamente le API Twitter. I producers si occupano di pubblicare sui topic stream di dati in ingresso al sistema, mentre i consumers consentono di recuperare tali informazioni e passarle allo strato successivo dell'architettura. Le operazioni di lettura e scrittura sui topic sono gestite dai broker che compongono il cluster Kafka. Infine, Zookeeper permette il coordinamento tra i vari nodi del cluster, garantendo il corretto funzionamento nel caso in cui un broker diventi inattivo.

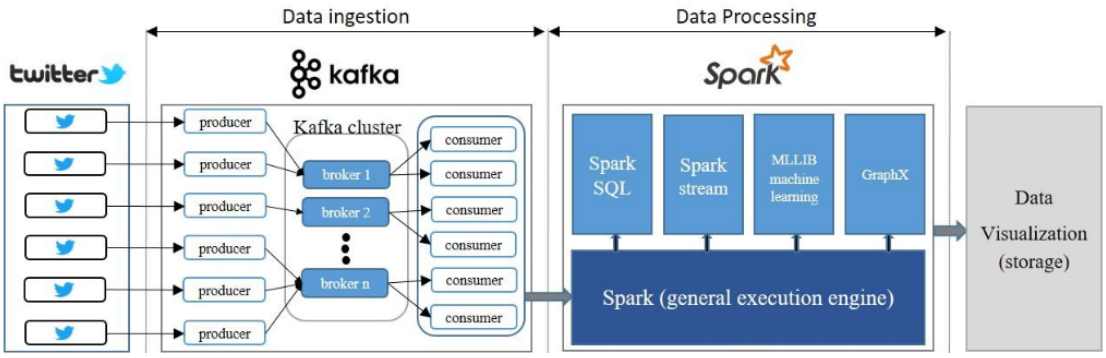


Figura 1.4. Real-time Data Analytics Framework for Twitter Streaming Data [12]

Per quanto riguarda la sezione di data processing, questa è implementata utilizzando Apache Spark. Internamente, vengono svolte le seguenti operazioni: Spark Streaming, una libreria che permette il processamento di data stream in real-time, riceve i dati tramite Kafka consumers e li divide in set di mini-batches, i quali vegono elaborati da Spark engine per generare i risultati finali che potranno essere salvati su database o file system (come HDFS). Apache Spark include librerie aggiuntive come MLlib per il machine learning, che permettono di eseguire elaborazioni più

complesse al fine di migliorare la qualità dei risultati ottenuti.
La parte che riguarda il data visualization, invece, dipende dalla modalità in cui verrano memorizzati i risultati ottenuti. Questa sezione è implementata tramite un database NOSQL in quanto i tweets possono assumere diversi formati (testo, immagini, video) e cambiare a seconda della situazione corrente.
"An Ingestion and Analytics Architecture for IoT applied to Smart City Use Cases" [13] propone un'architettura per il real-time processing progettata per applicazioni IoT. In generale, tali applicazioni richiedono di reagire agli eventi in tempo reale sulla base della conoscenza degli eventi passati. Queste informazioni sono essenziali per capire quale comportamento ci si aspetta, in modo da identificare eventuali anomalie. Per tale motivo occorre che lo storico dei dati sia analizzato prima di procedere con l'analisi in real-time.
Come mostrato in figura 1.5, l'architettura proposta separa due flussi di dati, batch e stream, permettendo di operare su di essi in maniera separata e indipendente.
Il primo step riguarda l'acquisizione dei dati, realizzata tramite Node Red. Tale software permette di collezionare informazioni da diversi dispositivi o sorgenti esterne, come servizi web RESTful, senza nessuna limitazione riguardo il formato dei dati, sia esso XML piuttosto che JSON. I dati raccolti conterranno informazioni ridondanti che possono essere eliminate tramite funzionalità di pre-processing. A questo punto, Node-Red potrà pubblicare i risultati ottenuti su Kafka.
La fase successiva riguarda l'ingestion dei dati tramite un tool open-source, chiamato Secor, che permette di trasferire i messaggi dal message broker verso OpenStack Swift, agendo così da connettore tra Kafka e object storage. Poichè verranno memorizzate enorme quantità di dati, si utilizza un meccanismo per annotare con metadati gli oggetti all'interno di Swift, con l'obiettivo di dare la possibilità di effettuare ricerche sui dati grazie all'uso di Elastic Search, basato su Apache Lucene.
Lo step successivo utilizza come software di analisi batch Apache Spark che non solo permette di recuperare i dati da una varietà di sistemi di storage, come Swift per l'appunto, ma fornisce anche librerie per il machine learning. Per quanto riguarda invece il processamento in real-time, viene usato il framework Complex Event Processing (CEP) per correlare eventi indipendenti in ingresso al sistema e generare un Complex Event. Un CEP Engine fornisce una serie di plugin in modo da migliorare l'acquisizione dei dati da fonti esterne e implemetare una logica di bussiness imponendo alcuni tipi di regole all'interno del sistema, in modo tale da creare l'output finale. Questo è il componente dell'architettura in cui il flusso di dati stream viene elaborato insieme al risultato ottenuto dall'analisi batch.

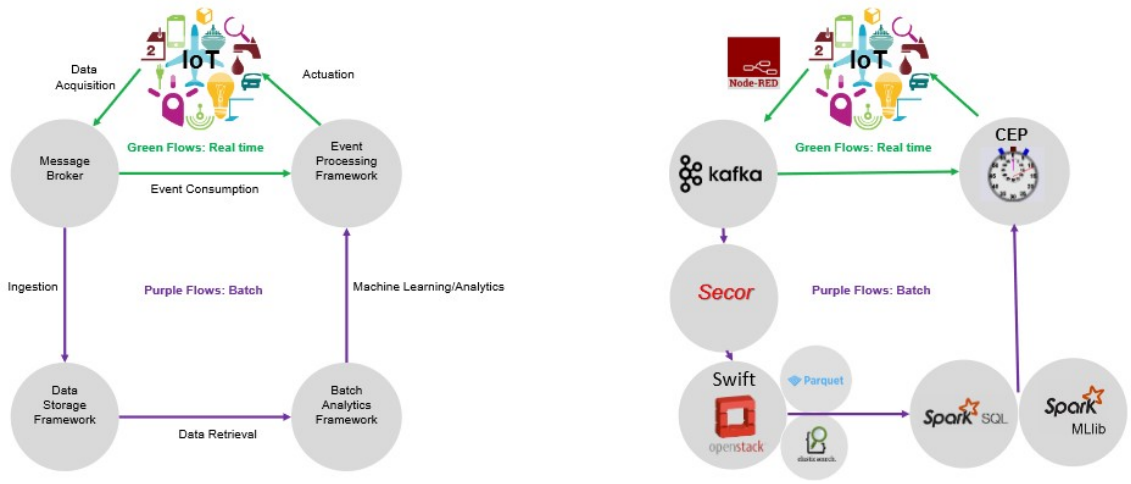


Figura 1.5. Ingestion and Analytics Architecture for IoT [13]

Capitolo 2

Progettazione di un'infrastruttura per l'analisi di dati streaming

In questo capitolo ci occuperemo di descrivere il sistema realizzato sia da un punto di vista architetturale che del funzionamento. Una sezione sarà dedicata al confronto tra le varie soluzioni open-source disponibili per l'implementazione di ogni step dell'architettura, in modo tale da esporre le motivazioni che hanno portato alla scelta di determinati framework piuttosto che di altri.

2.1 Architettura del sistema

Per definire l'architettura del sistema si è deciso di utilizzare uno schema a blocchi in cui vengono rappresentate le varie entità che fanno parte dell'architettura. In particolar modo si è deciso di suddividere il sistema in sei blocchi principali come mostrato in figura 2.1.
Il primo componente è rappresentato dal data ingestion real-time e viene utilizzato per acquisire i dati da sorgenti esterne e instradarli verso la corretta destinazione. A differenza dell'ingestion batch, dove i dati vengono raccolti per un intervallo di tempo e poi emessi in output, in uno scenario real-time, ogni volta che un evento arriva questo viene immediatamente inviato allo strato successivo dell'architettura cercando di garantire una latenza quanto più bassa possibile.
Questo componente potrebbe essere utilizzato anche per operazioni di pre-processing sui dati, nel caso ad esempio in cui si vogliano eliminare informazioni superflue oppure trasformare il formato dei dati, in modo da prepararli allo step successivo dell'infrastruttura.
Il componente successivo riguarda il message broker e viene utilizzato per la comunicazione tra i vari blocchi del sistema, attraverso lo scambio di messaggi caratterizzati

da una struttura ben definita. Inoltre, rappresenta il percorso che i dati devono seguire, affinchè venga rispettata la corretta sequenza delle operazioni che il sistema dovrà compiere per ottenere il risultato finale. Un'altra caratteristica fondamentale risiede nel fatto che il message broker permette di ridurre al minimo la dipendenza tra i vari strati dell'architettura, per cui, ogni componente dovrà preoccuparsi solamente di come inviare e ricevere messaggi verso il message broker piuttosto che conoscere la modalità di interfacciamento verso ogni entità del sistema.
Successivamente vi è la componente che si occupa di stream processing. Questa parte viene utilizzata per elaborare continuamente stream di dati attraverso una serie di operazioni. Riguarda il processamento dei dati "in motion", ovvero dei dati per come sono prodotti o ricevuti dal sistema. Uno degli aspetti più importanti di tale fase riguarda la possibilità di processare i dati in parallelo senza problemi di dipendenza dei dati o sincronizzazione.
Alla ricezione di un evento, la fase di di stream processing permette al sistema di reagire istantaneamente a tale evento: ad esempio è possibile lanciare un allarme, aggiornare attributi sui dati, o "ricordare" l'evento stesso per riferimenti futuri.
Per quanto riguarda la componente di machine learning, si è scelto di aggiungere questa funzionalità per offrire al sistema la capacità di apprendere in maniera automatica alcuni "pattern" che caratterizzano l'insieme dei dati analizzati al fine di effettuare previsioni sul nuovo stream dati in ingresso al sistema. Maggiore è la quantità dei dati a disposizione su cui allenare il modello di machine learning, più accurato sarà l'apprendimento e quindi migliori saranno le previsioni sui dati. Possiamo quindi affermare che il machine learning equivale al motore del valore dei Big Data.
Il prossimo blocco riguarda il data storage. Tale blocco si occupa di memorizzare i dati durante il loro ciclo di vita all'interno del sistema; ad esempio, dopo la fase di ingestion saranno disponibili raw data, ovvero i dati nella forma in cui sono stati prodotti dalla sorgente, mentre dopo la fase di processing avremo i dati elaborati, arricchiti di informazioni utili.
All'interno dell'architettura si è deciso di utilizzare due componenti diverse per il data storage: la prima permette di archiviare l'intero storico dei dati in maniera scalabile, fault-tolerant e con high availability, ma non viene utilizzata per interrogare i dati poichè richiederebbe tempi di attesa elevati per un'infrastruttura real-time; la seconda, invece, rigurarda il data warehouse, mantiene un volume di dati inferiore rispetto alla parte precedente, relativo solamente alle attività più recenti eseguite dal sistema, ma risulta migliore per svolgere l'interrogazione dei dati al fine di visualizzare i risultati ottenuti nel minor tempo possibile. Inoltre, dallo storico dei dati è possibile ricavare il dataset da utilizzare per le attività di machine learning.

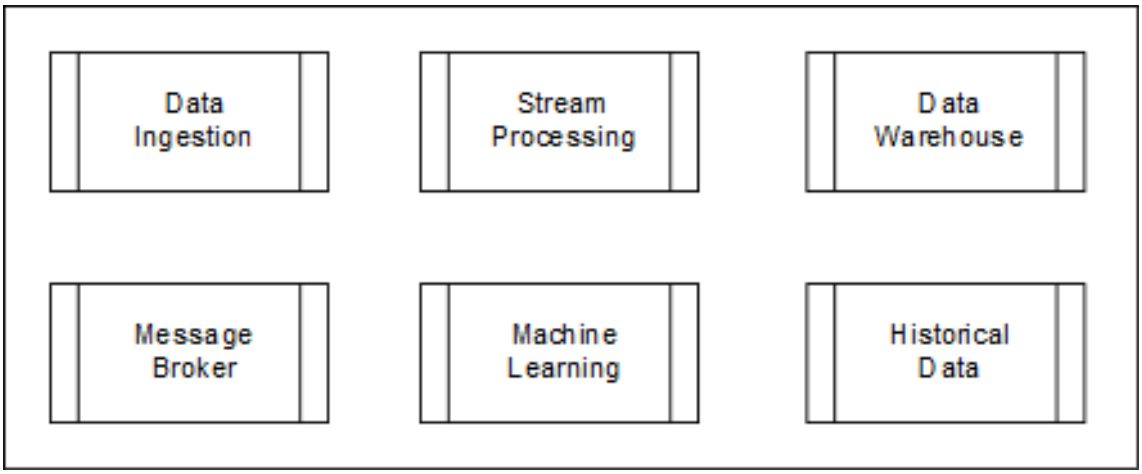


Figura 2.1. Architectural Components

2.2 Scelta delle componenti

In questa sezione si è scelto di effettuare un confronto sui vari software open-source che operano intorno all'ecosistema Hadoop con l'obiettivo di realizzare un'infrastruttura Big Data per l'ingestion real-time. In particolare, ogni sezione sarà dedicata a uno dei seguenti step dell'architettura: data ingestion, message broker, stream processing, data warehouse e machine learning. Per quanto rigurda la parte di data storage che si occupa mantenere lo storico dei dati si è scelto di utilizzare come soluzione il file system distribuito di Hadoop.
Per ogni sezione, si evidenzieranno vantaggi e svantaggi relativi ai software analizzati e verrà proposto il framework più idoneo all'implementazione dell'architettura, esponendo le motivazioni che hanno portato a tale scelta.

2.2.1 Data Ingestion

Rappresenta il primo step dell'architettura, si occupa di collezionare i dati, provenienti da una o più sorgenti e con formati differenti, e può includere una parte di pre-processing per migliorare la qualità dei dati, ad esempio attraverso operazioni di filtraggio che riducano la ridondanza delle informazioni ottenute. Per questa fase si è scelto di effettuare un confronto tra Apache Flume ed Apache NiFi.
Apache Flume è un sistema distribuito che può essere usato per collezionare, aggregare e trasferire una grande quantità di eventi streaming. Usa un semplice modello di dati costituito dal Flume Agent, un processo che ospita delle componenti che permettono di trasferire gli eventi da una sorgente, per esempio un web server, fino a una destinazione esterna, come HDFS. In particolare, utilizza una Source per ricevere dei dati dall'esterno e memorizzarli su uno o più Channel finchè gli stessi non

verranno consumati da un componente chiamato Sink, il quale sarà responsabile di pubblicare l'evento su un sistema esterno. Inoltre, definisce dei componenti chiamati Interceptors, per eseguire semplici trasformazioni su stream dati.
Apache Flume è ideale per casi come il trasferimento di stream di logs provenienti da più sorgenti verso un data store centrale. Uno dei vantaggi di questo framework è quello di essere supportato nella maggioranza delle distribuzioni commerciali di Hadoop presenti al giorno d'oggi.
Apache NiFi è un software open-source scritto in linguaggio Java e basato sul progetto"NiagaraFiles" sviluppato nel 2014 dalla NSA. [14] Utilizza delle componenti, chiamate processori, che permettono di effettuare operazioni come: catturare dati da sorgenti, trasformare e pubblicare il flusso di dati su sistemi esterni. Ad oggi, nel framework sono disponibil circa 200 processori diversi.
NiFi fornisce controllo in real-time per gestire lo spostamento dei dati tra più sistemi e offre la possibilità di lavorare in modalità cluster, garantendo proprietà come scalabilità, fault tolerant e availability. Altri aspetti chiave di questo framework sono: provenienza dei dati, utile per seguire il percorso del dataflow dalla sorgente alla destinazione; interfaccia utente web-based, per un debugging migliore dell'applicazione;supporto al back-pressure (tramite appositi buffer), fondamentale per gestire diverse frequenze di arrivo dei dati all'interno del sistema; elevata configurabilità, in quanto permette di impostare diversi trade-off per migliorare le performance del sistema, come ad esempio bassa latenza contro alto throughput ma anche garanzia di consegna contro tolleranza alle perdite; sicuro, grazie all'utilizzo del multi-tenant e di protocolli di sicurezza come HTTPS ed SSL; estensibile, poichè è possibile sviluppare ed utilizzare nuovi processori oltre quelli già disponibili, semplicemente creando delle classi Java che estendono opportune interfacce.
Sia NiFi che Flume offrono grandi performance, scalano bene orizzontalmente e permettono di estendere le loro funzionalità attraverso componenti aggiuntivi personalizzabili. Con Flume i dati non vengono replicati sui vari nodi: configurando un channel in modo tale da avere alto throughput, se durante il trasferimento vi è un guasto sul nodo i dati saranno persi, mentre se volessimo un canale affidabile, la latenza potrebbe aumentare in maniera eccessiva. Anche NiFi non permette di replicare i dati tra i nodi del cluster, perlomeno non ancora. Quando un nodo va in crash, il flusso di dati può essere diretto ad un altro nodo, ma i dati accodati sul nodo guasto dovranno attendere finchè questo non ritorni attivo. A differenza di Flume, è possibile gestire messaggi di dimensioni variabili. NiFi può essere utilizzato per il simple event processing, in quanto permette di effettuare semplici operazioni sui dati. Sebbene non permetta il complex event processing, si integra molto bene con framework come Storm, Spark Streaming e Flink tramite gli opportuni processori che mette a disposizione.
Nalla tabella 2.1, vengono riassunte le caratteristiche dei due tool di ingestion affrontate finora.

Tool	Features	Limits	Use Cases
Flume	Sources, channels and sinks Interceptors	Data loss scenarios No data replication Data size (KB)	Moving high-volume streaming events into Hadoop
NiFi	User Interface Many processors Data provenance Back-Pressure	Not for complex event processing No data replication	Dataflow management online Data routing between various systems Arbitrary data size

Tabella 2.1. Comparison between Ingestion Tools

In conclusione, NiFi presenta maggiori potenzialità rispetto Apache Flume per quanto riguarda l'ingestion dei dati in real-time. Infatti, oltre a fornire funzionalità aggiuntive come un'interfaccia utente, informazioni sulla provenienza dei dati e la possibilità di migliorare la gestione dello stream dati in tempo reale senza effettuare il deploy delle nuove impostazioni del sistema, fornisce un framework fault-tolerant e scalabile capace di gestire migliaia di eventi al secondo. Inoltre, con NiFi è possibile utilizzare degli agenti, chiamati MiNiFi, che permettono di raccogliere flussi dati provenienti da più sorgenti contemporaneamente, mentre un Flume Agent opera esclusivamente tra una sorgente e una destinazione.

2.2.2 Message Broker

Questo strato dell'architettura ha lo scopo di distribuire i dati, trasportati all'interno di messaggi, tra le varie componenti del sistema, definendo così il percorso che uno stream deve eseguire all'interno del sistema, affinchè tutte le operazioni richieste sui dati siano eseguite in maniera corretta e ordinata. In questo contesto, si è scelto di effettuare un confronto tra RabbitMQ e Apache Kafka.
RabbitMQ è un middleware message-oriented, scritto in Erlang, supporta diversi protocolli standard di messaggistica, tra cui Advaced Message Queieng Protocol (AMQP), Streaming Text Oriented Messagging Protocol (STOMP) e Message Queue Telemetry Transport (MQTT). Fornisce scalabilità, bassa latenza e availability, quest'ultima ottenuta tramite message replication. Tra i vari aspetti di RabbitMQ ricordiamo: interfaccia utente web-based, per il monitoraggio delle varie code di messaggi; message priority; routing dei messaggi flessibile tra produttori e consumatori; supporto alle transazioni, fornisce funzionalità di commit e rollback; garanzia ordine dei messaggi all'interno delle code. Inoltre, il sistema tiene traccia dello stato di ogni messaggio tramite un meccanismo esplicito degli ack, infatti è compito del consumatore segnalare al message broker la corretta ricezione dei messaggi.

Apache Kafka è una piattaforma open-source, scritta in Java e Scala, sviluppata originariamente da LinkedIn nel 2011, permette di realizzare pipeline di dati real-time e applicazioni di streaming. A differenza di RabbitMQ, non supporta protocolli standard come AMQP o MQTT, ma implementa il proprio protocollo di messaggistica e inoltre, invece delle code dei messaggi utilizza delle strutture dati chiamate topic, che possono essere divise in una o più partizioni.
Kafka fornisce le seguenti API chiave:

- Producer API: permette ad un'applicazione di pubblicare uno stream di dati sul cluster Kafka. Il producer è responsabile di scegliere quale record assegnare a quale partizione di un determinato topic (ad esempio tramite una modalità round-robin per favorire il load balancing all'interno del sistema).
- Consumer API: consente di ricevere i messaggi pubblicati su uno o più topic Kafka. Ogni consumer appartiene a un consumer group, ciò garantisce che più applicazioni possano leggere in parallelo un record alla stessa posizione all'interno del topic. Inoltre, i messaggi di un topic verranno distribuiti tra più istanze consumer dello stesso gruppo.
- Streams API: fornisce le funzioni per creare applicazioni di streaming, in modo da processare uno stream di record presenti su un determinato topic e salvare i risultati ottenuti su un topic di output.
- Connector API: permette di collegare sistemi di dati esistenti con il cluster Kafka. Ad esempio, è possibile importare i dati presenti su un database MySQL, ma anche pubblicare uno stream di record su Cassandra.

Apache Kafka si appoggia su un commit log in cui viene memorizzato l'offset dei messaggi che sono stati pubblicati e letti rispettivamente dai producers e consumers. Viene definito come "durable message store" in quanto i client possono chiedere di ricevere lo stesso stream di eventi più volte, al contrario dei tradizionali message broker che rimuovono i messaggi dalla coda una volta ricevuto l'ack di avvenuta consegna [15].
Permette di impostare la consegna dei messaggi scegliendo tra tre diverse modalità: at most once, in cui i messaggi verranno trasmessi al consumatore solo una volta ma in caso di perdite il messaggio non arriverà mai al consumer; at least once: ritrasmissione dei messaggi con possibilità di record duplicati al consumatore; exactly once: assicura che i messaggi vengano consegnati e che in caso di perdite non vi siano record duplicati al consumer.
Dal punto di vista delle performance, Kafka garantisce un alto throughput dei messaggi mantenendo bassa la latenza, sebbene non garantisca l'ordine dei record fra più partizioni dello stesso topic, ma solo all'interno della singola partizione. Inoltre, rispetto RabbitMQ supporta meglio la scalabilità orizzontale garantendo un

miglioramento delle prestazioni incrementando il numero di macchine che formano il cluster.
Sebbene RabbitMQ sia un framework auto-sufficiente, Kafka necessita di ZooKeeper per il coordinamento tra i vari broker. In un cluster Kafka, per ogni partizione viene eletto un broker come leader, mentre i restanti broker saranno marcati come follower. Il leader si occupa di gestire tutte le richieste di lettura e scrittura verso quella determinata partizione e, in caso di guasti, uno dei followers verrà eletto come nuovo leader.
In definitiva, Kafka offre una soluzione che si integra meglio in una piattaforma Big Data per ingestion real-time in quanto, a differenza di RabbitMQ, permette il disaccoppiamento tra produttore e consumatore e quindi la quantità di messaggi e la frequenza con cui questi arrivano non influiscono sulle performance del sistema, eliminando così il problema di avere consumers più lenti dei producers. In più, Kafka garantisce prestazioni migliori a livello di throughput e scalabilità. Se consideriamo un batch formato da 100 messaggi, con RabbitMQ si ottiene un throughput inferiore a 10 mila messaggi al secondo, mentre con Kafka si arriva fino a circa 90 mila messaggi al secondo [16].

Feature	Kafka	RabbitMQ
Need a **durable message store** and **message replay capability**	Y	N
Need ordered storage and **delivery**	Y*	N
Need **multiple different consumer of same data**	Y	N
Need to handle **throughput** of all my data well even at web scale and not a smaller set of messages	Y	N
Need a high throughput with **low latency**	Y	N
Need to **decouple producers** and **consumers** from a **performance perspective** to avoid the "slow consumer problem"	Y	N
Need good **integration** with the **Hadoop ecosystem** and **modern stream processing frameworks**	Y	N
Need a **good buffer for batch systems** to scale well to large backlogs	Y	N
Need **same tool** for building data pipelines and streaming data applications without the help of additional software.	Y	N
Need **support for multiple protocols**: AMQP, STOMP, JMS, MQTT, HTTP, JSON-RPC, ...	N	Y
Need **message priority:** producers can specify the priority of messages to consumers	N	Y
Need **explicit delivery processing acknowledgements** of messages from consumers	N	Y
Need **flexible routing**: producers direct messages to appropriate consumers	N	Y
Need **transaction support:** provide commit and rollback functionality for local transactions	N	Y
Need **native tracing** support to let me find out what's going on if things are misbehaving.	N	Y
Need **a browser-based UI** for management and monitoring of my message brokers.	N	Y
Need a **self-sufficient message broker** without additional tool such as Zookeeper	N	Y

Figura 2.2. Comparison between Apache Kafka and RabbitMQ [17]

La tabella mostrata in figura 2.2 mette a confronto Apache Kafka e RabbitMQ, riassumendo i vari aspetti affrontati finora.

A questo punto, è lecito chiedersi se vi sia sovrapposizione tra le funzionalità offerte da NiFi e Kafka. Come vedremo, questi due framework risultano complementari. In molti casi, infatti, le sorgenti che generano i dati, pensiamo ad esempio ai sensori IoT, non conoscono il protocollo utilizzato da Kafka per cui risulta impossibile trasmettere i dati direttamente al sistema. MiNiFi, in tal senso, fornisce gli strumenti necessari per raccogliere dati direttamente dai vari dispositivi e trasmetterli a NiFi, il quale può effettuare eventuali operazioni di filtering o arricchimento prima di inoltrare il flusso di dati a Kafka, utilizzando il processore PublishKafka. Inoltre, con tale meccanismo se volessimo bloccare temporaneamente lo stream di dati in ingresso a Kafka non occorrerà spegnere le istanze di MiNiFi sparse sui vari dispositivi IoT, ma basterà semplicemente stoppare il processore utilizzato da NiFi [18].

2.2.3 Stream Processing

In questa sezione si è scelto di effettuare un confronto tra Apache Storm e Apache Flink con lo scopo di trovare il framework per stream processing più adatto alla realizzazione di un'architettura Big Data per ingestion ed analisi real-time.
Per quanto riguarda Apache Storm, la sua prima release risale al 17 settembre 2011 ed è stato progettato con l'idea di realizzare un sistema di calcolo distribuito utilizzato per analisi dati in real-time. Permette il processamento dello stream dati con una latenza molto bassa, inferiore al secondo, e si integra molto bene con le tecnologie che ruotano attorno all'ecosistema Hadoop.
Storm offre garanzie di elaborazione at least once, in quanto, ogni stream dati viene processato almeno una volta e in caso di guasti non si verifica nessuna perdita di dati, ma allo stesso tempo si possono creare situazioni in cui si verifichino la presenza di record duplicati.
Apache Storm utilizza un meccanismo di record acknowledgement per garantire che i messaggi siano riprocessati dopo un guasto: ogni record processato da un operatore ritorna un ack all'operatore precedente e ciò genera un pesante overhead sul sistema. Per tale motivo, non può garantire un throughput elevato e può presentare problemi con il controllo del flusso poichè il meccanismo degli ack duplicati potrebbe annunciare guasti falsi.
Apache Flink è stato sviluppato la prima volta l'8 marzo 2016 ed è stato considerato il 4G dei framework di analisi per Big Data. Fornisce diversi casi di processamento distribuito dei dati: batch, streaming, query SQL, machine learning e graph processing.
Flink permette la gestione di eventi ricevuti fuori ordine, ad esempio provenienti da diverse partizioni dello stesso topic di Kafka, grazie alle diverse nozioni di tempo: Event time, indica il tempo in cui viene creato un evento ed è inserito dal sensore di produzione; Ingestion time, definisce il tempo in cui un evento giunge al primo operatore di Flink; Processing Time, tempo in cui un evento viene processato da un

particolare operatore time-based, come ad esempio l'operatore di window.
Come ambiente di esecuzione per applicazioni, Flink utilizza due tipi di processi: JobManager, si occupano di coordinare l'esecuzione in modalità distribuita non appena ricevono la richiesta da parte del client; TaskManager, si occupano di eseguire i task all'interno di opportuni task slot, in cui è possibile configurare il processamento di più thread in parallelo. E' possibile garantire high-availability utilizzando più JobManager, grazie all'ausilio di Apache ZooKeeper.
Con la release 1.4 avvenuta nel dicembre 2017, Apache Flink fornisce il supporto verso Kafka tramite un connettore che permette meccanismi di consegna exactly-once, sia per quanto riguarda la ricezione di dati provenienti dal message broker (ovvero Flink in modalità Kafka consumer) che per la pubblicazione di messaggi (Flink in modalità Kafka producer).
Come Storm garantisce una latenza molto bassa (si parla di sub-second latency) e si integra facilmente nell'ecosistema Hadoop, ma a differenza di quest'ultimo permette di ottenere un throughput molto più elevato come mostrato in figura 2.3.
Grazie al meccanismo degli snapshot distribuiti offre garanzie di processamento exactly once: il recupero di un guasto equivale a ripristinare l'ultimo snapshot disponibile del dato senza nessuna perdita di informazioni e ciò introduce un overhead sul sistema minore rispetto quello che accadeva con Storm. Inoltre, cambiando l'intervallo di snapshotting non si hanno ripercussioni sui risultati ottenuti dal job streaming e quindi si ottiene una separazione tra flow control e throughput.

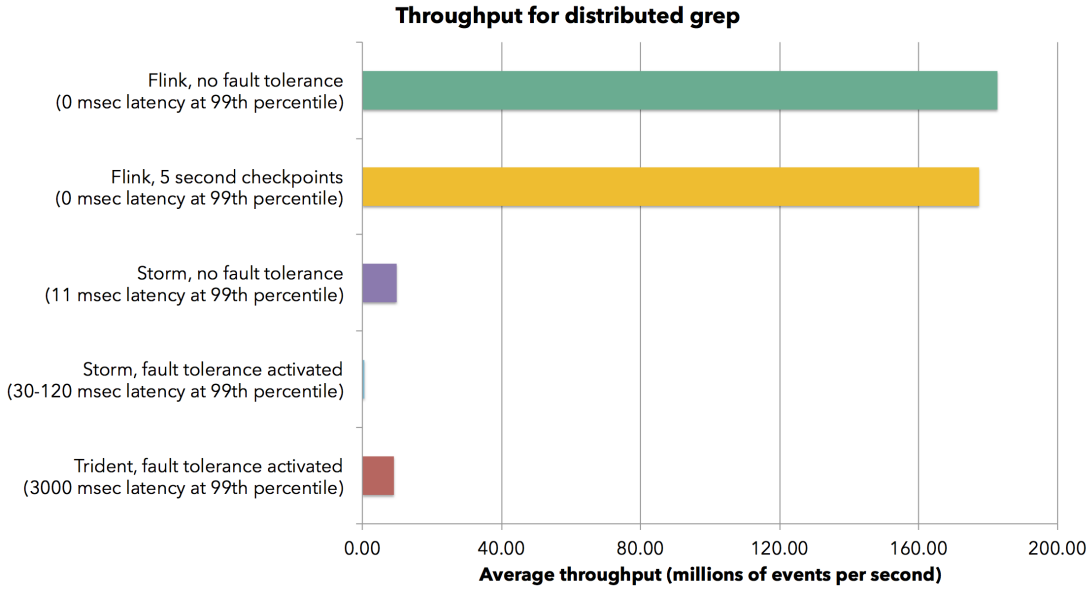


Figura 2.3. Comparison between Apache Flink and Apache Storm [19]

Per questi motivi, pertanto, si è scelto di implementare la parte relativa allo stream

processing utilizzando il framework Apache Flink.
Poichè Kafka stream API offre funzionalità simili alle API fornite da Flink per il processamento dello stream dati, ci si potrebbe domandare se non fosse sufficiente utilizzare la libreria offerta da Kafka per rendere l'architettura più semplice possibile. In realtà, la differenza sostanziale consiste nel fatto che Flink presenta il vantaggio di poter configurare come le applicazioni devono essere distribuite, gestite e come viene coordinato il processamento parallelo (e quindi il fault-tolerant) all'interno dell'infrastruttura. Ciò permette di rendere indipendente l'applicazione dall'ambiente di esecuzione sia esso YARN, Mesos o Docker container [20]. Kafka Stream API è una libreria progettata per consumare record da uno o più topic Kafka, trasformare i dati e pubblicare il nuovo stream su uno o più topic; richiede, quindi, un'installazione Kafka esistente e non permette di interagire con sistemi di dati esterni, se non attraverso Kafka Connect. Inoltre, processa un elemento per volta in modalità sincrona, di conseguenza occore fare attenzione alle operazioni che potrebbero bloccare l'intero stream. Flink, invece, permette il processamento asincrono degli eventi, consente di combinare più stream provenienti da diverse sorgenti e di pubblicare i risultati su più sistemi come database, file system, message logs.
Infine, se dovessimo paragonare le funzionalità offerte da NiFi con quelle offerte da Flink, possiamo osservare che mentre il primo permette una fase di pre-processing, elaborando un evento per volta attraverso semplici operazioni, il secondo consente di eseguire operazioni più complesse sui dati (come window) ma richiede la scrittura del codice. Di conseguenza, NiFi semplifica l'ingestion dei dati da più sorgenti tramite una serie di processori disponibili ma limita la fase di processing dei dati.

2.2.4 Data Warehouse

Rappresenta la parte di architettura che si occupa di conservare i dati in modo tale da garantire, utlizzando delle query quanto più specifiche possibili, la corretta visualizzazione dei risultati ottenuti.
Per questa sezione si è scelto di effettuare un confronto tra due tipi di database non relazionali, in quanto, a differenza dei database SQL forniscono i seguenti vantaggi: supporto per dati semi-strutturati; diversi modelli per memorizzare i dati (come ad esempio documenti JSON, wide column stores, coppie key-value, grafi); nessun record deve possedere obbligatoriamente le stesse proprietà degli altri, possibilità di aggiungere nuove proprietà on the fly; schemi flessibili e dinamici; scalabilità orizzontale; miglioramento delle performance del sistema attraverso un trade-off tra consistenza e availability. In tale contesto, i due database NOSQL selezionati per il confronto sono MongoDB e Cassandra.
Progettato dalla società di software 10gen (ora conosciuta come MongoDB Inc.) nell'Ottobre 2007, MongoDB è scritto in C++ e rilasciato sotto licenza GNU AGPL v3.0 che lo rende un software libero e open-source. E' un database non relazionale,

di tipo document oriented, infatti, si allontana dalla struttura tradizionale utilizzata dai database relazionali basata su tabelle a favore di documenti in stile JSON con schema dinamico (formato BSON), rendendo così l'integrazione di dati di alcuni tipi di applicazioni più facile e veloce.
Caratteristiche chiave di MongoDB includono: schemi che cambiano in base all'evoluzione delle applicazioni (schema-free), full index support per alte prestazioni, replicazione e failover insieme ad availability, scalabilità.
MongoDB utilizza il metodo Sharding che consente di distribuire i dati su server multipli rendendo il sistema altamente scalabile. Inoltre, grazie al processo di Replication permette la sincronizzazione dei dati su più macchine dello stesso cluster. La replicazione migliora l'affidabilità del sistema proteggendo il database dalle perdite di dati su un singolo server e può essere usata per migliorare le capacità di lettura dati.
Apache Cassandra è un database NOSQL di tipo wide column store: il keyspace è il contenitore di dati più esterno in cui è possibile memorizzare una o più column families. Ogni famiglia di colonne può contenere una o più righe caratterizzate da: row key, identifica univocamente una riga all'interno della column family; una lista ordinata di coppie (column-key, column-value).
Aspetti chiave di Cassandra includono:

- Commit-log: contiene la lista delle operazioni sui dati prima che queste vengano eseguite; è il meccanismo utilizzato per il crash-recovering.
- Mem-table: struttura dati residente in memoria; mantiene una collezione di colonne raggiungibile attraverso l'opportuna riga.
- SSTable: file su disco in cui vengono memorizzati i dati presenti su mem-table al raggiungimento di un determinato valore soglia.
- Partitioner: determina come distribuire le repiche dei dati tra i vari nodi del cluster. Utilizza una funzione di hash per ottenere un token relativo a una particolare riga.
- Snitch: permette di impostare la topologia del datacenter, definendo quali nodi appartengono. Usa un processo che monitora le performance del sistema e sceglie la miglior replica dati da utilizzare nelle operazioni di lettura.

A differenza di MongoDB, dove si utilizza un modello di architettura master-slave in cui solo il nodo master è incaricato di effettuare le operazioni di scrittura, in Cassandra ogni nodo del cluster assume lo stesso ruolo e può accettare richieste di lettura e scrittura. Di conseguenza MongoDB limita fortemente la scalabilità per quanto riguarda le operazioni di scrittura sui nodi.
Cassandra utilizza un protocollo di comunicazione peer-to-peer, chiamato Gossip,

che permette di scambiare continuamente informazioni sullo stato dei nodi. Ciò garantisce la rilevazione dei guasti all'interno del sistema e di recuperare un'eventuale perdita di dati tramite appositi meccanismi di recovery. Inoltre, esiste un processo di data replication che permette di distribuire una copia degli stessi dati su più nodi del cluster, a seconda del fattore di replicazione impostato.
Cassandra fornisce supporto a un linguaggio molto simile a SQL chiamato Cassandra Query Language (CQL) che comunque possiede alcune limitazioni (come ad esempio non permette il join), mentre in MongoDB le query sono strutturate come frammenti JSON.
Alcuni casi d'uso di Cassandra includono: Netflix (2500 nodi, 420 TB di dati, oltre 1 trilione di richieste al giorno), Apple (75000 nodi, 10 PB di dati), eBay (oltre 100 nodi, 250 TB di dati) [21].

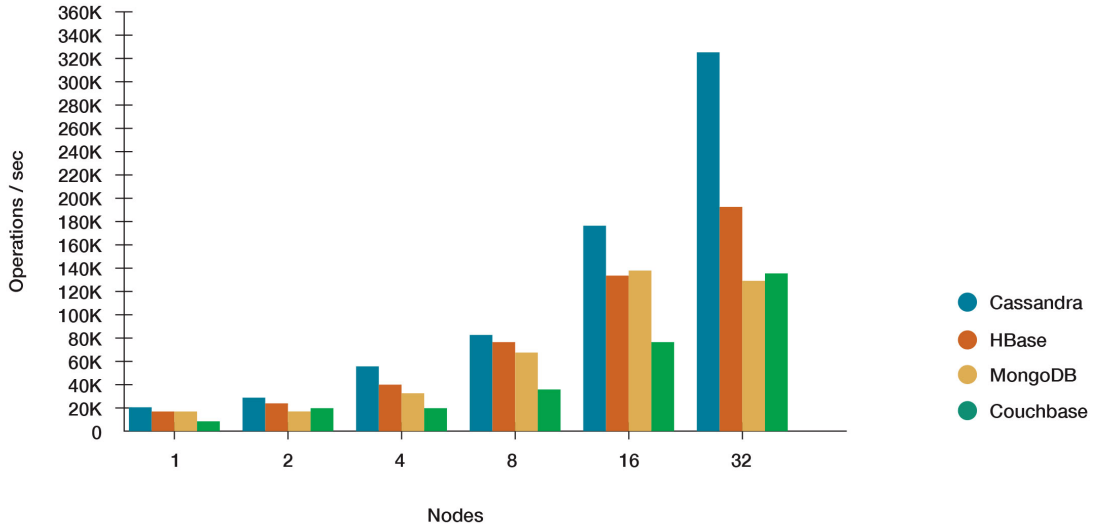


Figura 2.4. Comparison between Cassandra, HBase, MongoDB and Couchbase [22]

Come mostrato in figura 2.4, al variare del numero di nodi che formano il cluster, Cassandra riesce sempre a fornire un throughput più elevato rispetto agli altri tipi di database. La differenza diventa sempre più evidente con l'aumentare delle dimensioni del cluster, arrivando addirittura per 32 nodi ad un throughput di circa 300 mila operazioni al secondo contro quello di circa 150 mila operazioni al secondo fornito da MongoDB. Questo motivo, unito alla caratteristica di possedere "no single point of failure" e ad una maggiore scalabilità per quanto riguarda le operazioni di scrittura, sono i principali motivi che hanno portato alla scelta di implementare la parte di data warehouse utilizzando Cassandra.

2.2.5 Machine Learning

Rappresenta la parte di infrastruttura che si occupa di acquisire una conoscenza nascosta dai dati per effettuare predizioni e quindi prendere decisioni riguardo possibili eventi futuri. In questa fase sono necessari tre passi fondamentali: la costruzione di un modello a partire da un dataset completo, definito training set; la valutazione ed eventuale configurazione del modello; esportare il modello in produzione per estrarre informazioni utili dal nuovo stream dati in ingresso al sistema.
L'obiettivo di questo paragrafo è quello di effettuare un confronto tra due framework che forniscono strumenti per il machine learning. A tal proposito, si è scelto di effettuare un confronto tra Spark MLlib e H2O.
La prima è una libreria di machine learning fornita dalla piattaforma Apache Spark che include un'ampia selezione di algoritmi tra cui: logistic regression, linear regression, decision tree, random forest, support vector machine, naive Bayes, k-means. Spark MLlib offre, quindi, delle opzioni interessanti per diverse necessità; inoltre garantisce sia una buona scalabilità che velocità di esecuzione al variare delle dimensioni del dataset e permette l'interfacciamento con diversi linguaggi di programmazione come Java, Python e Scala.
H2O è una piattaforma open-source per il machine learning, utilizza processamento in-memory ed è scalabile, veloce e distribuita. A differenza di Spark MLlib fornisce una minor selezione di funzioni per l'analisi predittiva, ma permette di utilizzare algoritmi per il deep learning. Per quanto riguarda gli algoritmi di tipo Supervised sono disponibili: distributed random forest, generalized linear model, gradient boosting machine, naive Bayes, stacked ensemble e deep learning; mentre gli algoritmi Unsupervised sono: generalized low rank models, k-means clustering e principal component analysis.
H2O permette l'interfacciamento con i seguenti linguaggi di programmazione: Java Python, R, Scala; fornisce un'interfaccia utente web-based, chiamata H2O Flow UI, che permette la creazione, configurazione e valutazione di modelli per il machine learning in maniera semplice e veloce. Inoltre, facilita l'esportazione del modello costruito tramite la conversione in formato Plain Old Java Object (POJO) o Model Object Optimized (MOJO).
Altre caratteristiche chiave di H2O comprendono: la possibilità di importare dati in formati diversi, come CSV, AVRO, Parquet, e da diverse sorgenti dati tra cui Local File System, HDFS, JDBC, Amazon S3; Sparkling Water: progetto che permette l'integrazione tra Apache Spark e H2O; Deep Water: permette l'integrazione con altri framework che forniscono librerie di deep learning, tra cui: TensorFlow, Caffe, MXNet; AutoML: interfaccia per i non esperti che automatizza il processo di training e di tuning utilizzando più algoritmi di machine learning, ritorna la classifica dei modelli generati, ordinata secondo una metrica di valutazione prescelta; Grid

Search: poichè per ogni parametro di configurazione del modello è possibile impostare diversi valori, permette di costruire modelli basati su ogni combinazione dei vari parametri impostati dall'utente.

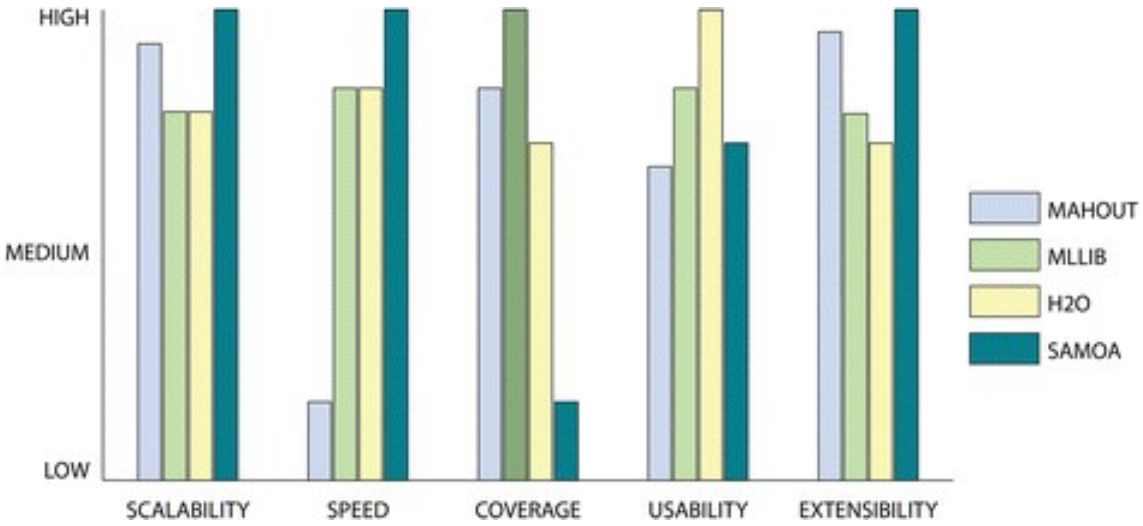


Figura 2.5. Comparison between frameworks for machine learning [23]

In figura 2.5, viene illustrato il confronto tra alcuni framework per il machine learning secondo i criteri di scalabilità, velocità, copertura, usabilità ed estensibilità. Come si può notare dal grafico, Spark MLlib e H2O risultano equivalenti dal punto di vista della velocità di elaborazione e della scalabilità, ovvero al variare delle dimensioni del dataset e della complessità dei dati. MLlib offre una copertura migliore, in quanto offre una maggiore selezione di algoritmi per il machine learning anche se non offre il supporto per il deep learning. In termini di usabilità invece, H2O prevale, in quanto, oltre al fatto di offrire un'interfaccia utente, permette di interfacciarsi con più linguaggi di programmazione rispetto MLlib. Infine, l'estensibilità rappresenta il numero di parametri di tuning richiesti prima che il modello venga deployato; possiamo notare come in tal caso i due framework siano molto simili sotto questo punto di vista.
Concludendo, per questa parte dell'architettura si è scelto di utilizzare H2O in quanto rappresenta un framework indipendente (MLlib invece richiede l'installazione della piattaforma Apache Spark), con un'interfaccia utente che semplifica l'utilizzo e permette di deployare facilmente il modello ottenuto. Inoltre, fornisce dei moduli per l'integrazione con Spark e altri framework per il deep learning che permettono di affrontare un'ampia varietà di casi d'uso.

2.3 Funzionamento del sistema

Come evidenziato nella figura 2.6, il funzionamento del sistema può essere definito dall'insieme delle operazioni che vengono effettuate in sequenza sullo stream dati una volta che questo arriva in ingresso all'infrastruttura.

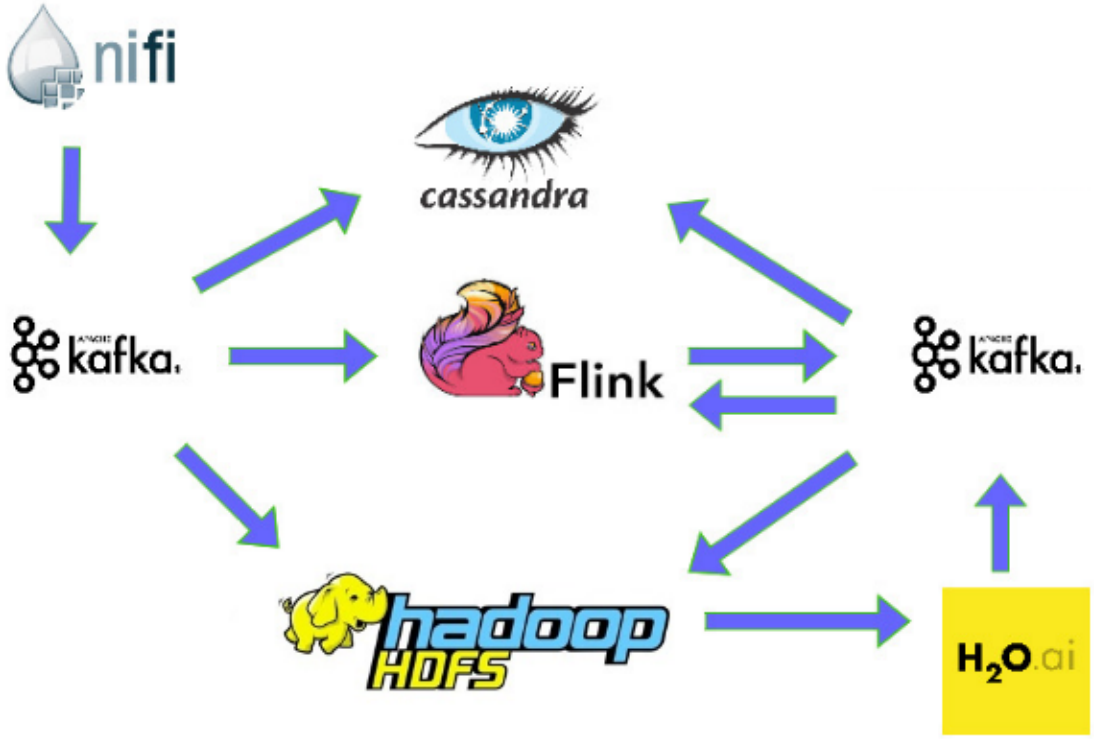


Figura 2.6. Big Data Architecture for Real-Time Ingestion

La prima componente che viene chiamata in causa riguarda la parte che si occupa dell'acquisizione dei dati. Come detto più volte, grazie a questo processo, il sistema riesce a catturare continuamente flussi di dati etorogenei provenienti da più di un'unica sorgente. In base al confronto effettuato nella sezione precedente, si è deciso di implementare tale funzionalità con il framework Apache NiFi. Dopo aver catturato i dati, NiFi permetterà, tramite opportuni processori, di effettuare eventuali operazioni prima di pubblicare i dati sul topic Kafka desiderato.
A questo punto, si passa alla fase successiva in cui il message broker è incaricato di trasmettere i dati raw a tre diverse componenti del sistema: Cassandra, HDFS e Flink. Le prime due entità si occuperanno di memorizzare i dati ricevuti, rispettivamente, su tabella e file system distribuito, mentre Flink svolgerà la fase di stream processing. Quest'ultima consiste nel produrre, tramite opportune funzioni,

un nuovo flusso di dati che potrà essere, ad esempio, lo stesso stream dati in input arricchito con nuovi attributi oppure ripulito da informazioni ridondanti.
Una volta ottenuto il nuovo stream dati, Flink si occuperà di pubblicare i risultatti ottenuti su un nuovo topic Kafka, diverso da quello utilizzato in precedenza per bufferizzare i dati raw. Ancora una volta il message broker trasferirà il nuovo stream dati su Cassandra e HDFS, utilizzando gli opportuni connettori.
Sebbene Cassandra e HDFS svolgano lo stesso compito, ossia quello di memorizzare i dati consegnati da Kafka, queste due componenti sono utilizzate anche per funzionalità aggiuntive. Infatti, Cassandra permetterà, tramite specifiche query in linguaggio CQL, di interrogare e visualizzare per mezzo di query i dati più recenti mantenuti nelle varie tabelle al fine di visualizzare i risultati ottenuti, garantendo brevi tempi di attesa. HDFS, invece, oltre a fornire un file system con high availability, distribuito e fault-tolerant, in cui verrà archiviato lo storico dei dati, verrà utilizzato come sorgente per importare i dati all'interno del cluster H2O, il framework che si occupa della parte di machine learning.
Non appena viene caricato l'intero dataset, sarà possibile scegliere tra un'ampia varietà di algoritmi offerti da H2O per creare il modello di machine learning desiderato. Dopo aver svolto le corrette valutazioni sul modello realizzato, occorre effettuare il deploy affinchè sia possibile estrarre una conoscenza nascosta dal nuovo stream dati. A tal proposito, H2O consente di esportare il modello in diverse modalità tra cui un oggetto Java, denominato POJO.
Per sottoporre il modello di machine learning sul nuovo flusso di dati si utilizzerà un'applicazione Java basata sulle API Kafka Stream. In tal modo, sarà possibile, ad esempio, effettuare predizioni in real-time a partire dal datastream processato in precedenza da Flink. Anche in questo caso, i risultati ottenuti saranno memorizzati su un nuovo topic Kafka così da poter essere inviati ad altre componenti del sistema come HDFS, Flink e Cassandra.
Infine, tramite Flink, sarà possibile elaborare nuovamente i dati ottenuti, sottoponendo il modello di machine learning, in modo da effettuare un nuovo tipo di analisi sui dati. Ad esempio, sarà possibile confrontare le predizioni sullo stream dati attuale con i valori reali ricavati dall'analisi successiva di stream futuri.

2.3.1 NiFi Processors

Apache NiFi utilizza entità, chiamate processori, per permettere una grande varietà di operazioni, tra cui: ingestion dei dati da sorgenti esterne, traformazioni del formato dei dati e pubblicazione del contenuto su sistemi differenti. Per stabilire un collegamento tra i vari processori utilizzati, NiFi mette a disposizione delle entità chiamate Connection. Queste agiscono come buffer intermedi e permetteno ai diversi processori di lavorare a diverse frequenze.

Per quanto riguarda l'acquisizione dei dati, alcuni dei processori che NiFi mette a disposizione sono:

- GetFile: crea dei FlowFile per ogni file contenuto all'interno della directory presente sul file system locale.
- GetFTP: permette di recuperare i file da un server che implementa il File Transfer Protocol. Esiste anche la versione GetSFTP per trasferimento file sicuro tramite SSH.
- GetHDFS: cattura i dati presenti sul file system distribuito di Hadoop.
- GetHTTP: prende i dati da un url HTTP o HTTPS e li scrive come contenuto del FlowFile. Una volta che cattura un oggetto da un determinato url, non lo riacquisce finchè non è cambiato il contenuto del dato sul server remoto.
- GetMongo: permette di recuperare documenti da un sistema MongoDB.
- GetKafka: riceve i messaggi da Apache Kafka e li inserisce come contenuto nel FlowFile.
- GetTCP: cattura i dati disponibili presso un endpoint raggiungibile tramite il protocollo TCP.
- ListenUDP: permette di catturare i pacchetti UDP in arrivo su una porta specificata.
- FetchS3Object: recupera il contenuto di un oggetto da Amazon Simple Storage Service (S3).
- GetSolr: interroga Solr e invia i risultati in formato XML all'interno di un FlowFile.

Inoltre, è possibile implementare un processore a seconda delle proprie esigenze, creando un'applicazione Java che utilizzi le API messe a disposione da NiFi.
Poichè la nostra infrastruttura si occupa di acquisire dati da diverse sorgenti, è possibile ricevere le informazioni in formati diversi come: AVRO, JSON, XML, CSV. Nel caso in cui volessimo ottenere un unico formato per tutti i dati prima che questi siano pubblicati su Kafka, NiFi permette di trasformare il flusso di dati tramite processori come ad esempio: ConvertAvroToJson, per tradurre un record Avro in un oggetto Json; ConvertCsvToAvro, per convertire file CSV in Avro secondo uno schema avro definito; ConvertRecord, per trasformare i dati che possiedono uno schema compatibile al RecordReader in un nuovo formato messo a disposizione dal RecordWriter.

Per pubblicare i dati su Kafka, invece, sono disponibili i seguenti processori: PublishKafka, permette di inviare verso Kafka l'intero contenuto del FlowFile come un singolo messaggio; PublishKafkaRecord, consente di convertire il formato dei dati all'interno del FlowFile, secondo uno schema fornito dal RecordWriter, prima di incapsularli in un messaggio da trasmettere su un topic Kafka.

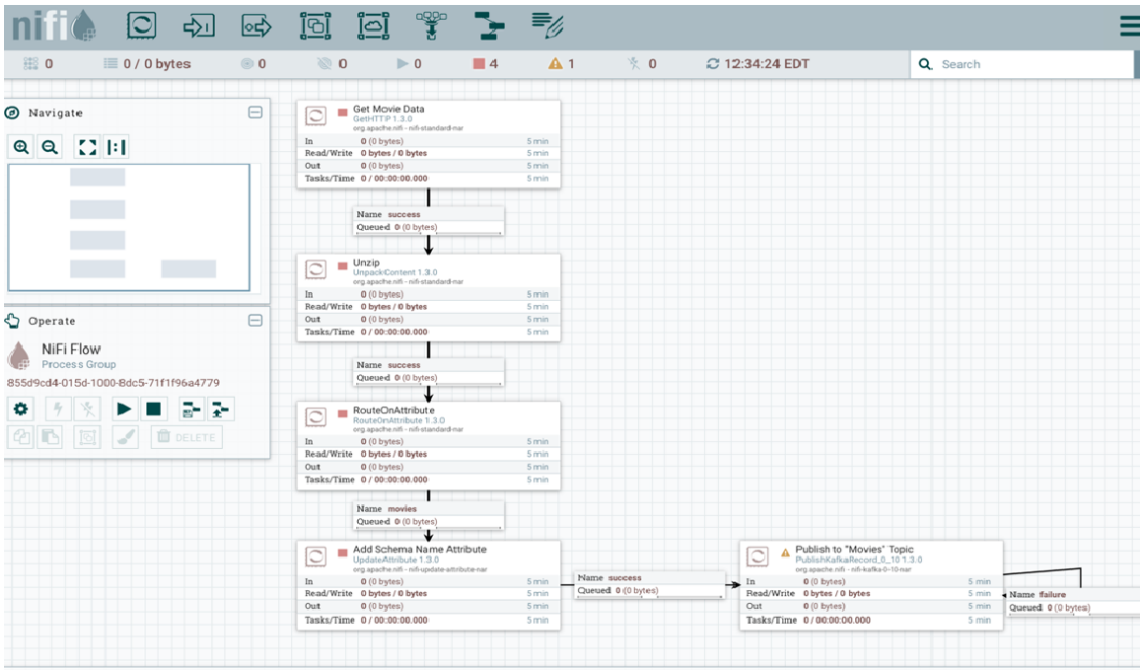


Figura 2.7. Apache NiFi: Data Ingestion From HTTP

In figura 2.7, viene mostrato un caso di data ingestion realizzato per mezzo dell'interfaccia utente Apache NiFi. Partendo dall'alto, il primo processore che incontriamo è GetHTTP, utilizzato per acquisire i dati da un url HTTP che in questo caso espone dati sui film. Poichè questi sono compressi all'interno di un file zip, successivamente troviamo UnpackContent che ci consente di estrarre i dati ricevuti. Proseguendo abbiamo il processore RouteOnAttribute che ci permette di filtrare i dati ottenuti, in modo da selezionare solo quelli di nostro interesse. Tramite il processore UpdateAttribute, invece, è possibile aggiornare gli attributi del nostro FlowFile, in questo caso ci si occupa di aggiungere il nome dello schema a cui si riferiscono i dati. Infine, grazie al processore PublishKafkaRecord, i dati verranno pubblicati su Kafka all'interno del topic Movies.

2.3.2 Schema Registry e Kafka Connect

Per quanto riguarda la parte relativa a Kafka, si è pensato di realizzare un sistema che funzionasse nel caso in cui i dati sarebbero stati raccolti (o successivamente trasformati tramite opportuni processori NiFi) in formato Avro. Uno dei motivi principali che ha portato a tale scelta è dovuto al fatto che per pubblicare i dati su HDFS e Cassandra tramite connettori Kafka non è possibile esportare i dati in qualsiasi formato. In base a tali considerazioni, si è reso necessario utilizzare il servizio Schema Registry, disponibile attraverso la piattaforma open-source Confluent, sviluppata dagli ideatori di Apache Kafka.
Confluent Schema Registry memorizza schemi Avro per Kafka consumers e producers; per ogni schema associa una versione e fornisce il controllo sulla compatibililtà dei vari schemi. Quando un sistema vuole pubblicare dei dati in formato Avro su Kafka, prima di tutto deve occuparsi di registrare lo schema di riferimento per quei dati, tramite opportune API RESTful fornite dal servizio Schema Registry, il quale in caso di successo ritornerà l'id associato a tale schema. A questo punto, Kafka Producer crea un messaggio contenente l'id dello schema registrato e i dati e li serializza in formato Avro, utilizzando un processo chiamato AvroConverter. Una volta che i dati vengono memorizzati all'interno di un topic Kafka, quando un sistema vuole leggere i dati da quel topic, per deserializzare correttamente il contenuto dei messaggi dovrà recuperare lo schema Avro di riferimento, interrogando il servizio Schema Registry.
Una volta che i dati vengono pubblicati su un topic, sarà compito di Kafka inviarli verso HDFS, Flink e Cassandra. A tal proposito, Kafka Connect è un tool che permette di importare ed esportare dati in real-time rispettivamente da e verso sistemi esterni in modo scalabile e sicuro. Nel primo caso si parlerà di Source Connectors, mentre nel secondo di Sink Connectors; questi non sono altro che Kafka producers e Kafka consumers che permettono agli sviluppatori di concentrarsi sullo spostamento dei dati tra Kafka e il mondo esterno. Per realizzare le funzionalità richieste dal nostro sistema si è reso necessario utilizzare i seguenti connettori: HDFS Sink, per esportare i dati sul file system distribuito di Hadoop; Cassandra Sink, per spostare i dati verso Cassandra; flink-connector-kafka, per trasferire i dati tra Flink e Kafka.
Per quanto riguarda invece i topic Kafka che vengono utilizzati all'interno del sistema, possiamo definire almeno tre topic diversi: il primo, topicRawData, si occuperà di memorizzare i dati acquisiti da NiFi; il secondo, topicCleanData, avrà il compito di memorizzare i risultati ottenuti dopo la fase di stream processing; il terzo, topicPredictedData, conterrà le informazioni estratte dai dati sottoponendo i risultati di stream processing al modello di machine learning.
Come vedremo nella fase di implementazione, il sistema utilizzerà Kafka in modalità cluster con tre broker; ogni topic sarà diviso in tre partizioni e i dati saranno replicati di un fattore tre, in modo da garantire scalabilità, high availability e fault

tolerant.

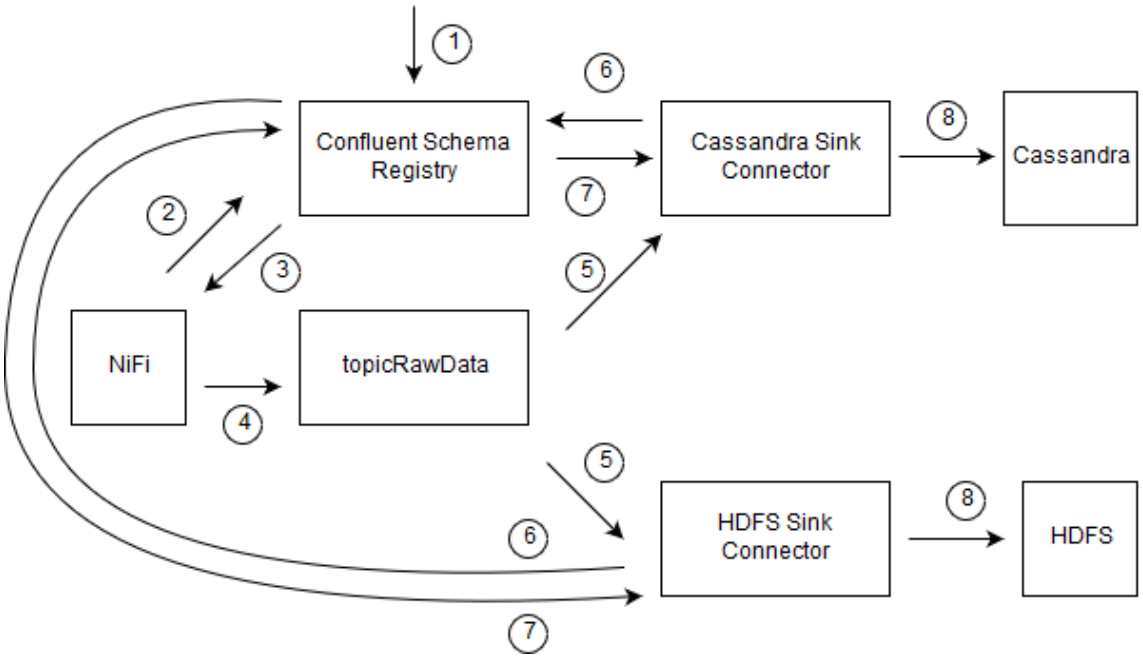


Figura 2.8. Schema Registry and Connectors in Apache Kafka

Nell'esempio di figura 2.8 è possibile osservare come viene gestito da Kafka il trasferimento dei dati raw, utilizzando i servizi Kafka Connect e Schema Registry. Per semplicità si è deciso di eliminare la parte relativa al connettore utilizzato dal sistema per pubblicare i dati su Flink. Supponendo di conoscere lo schema Avro con cui i dati dovranno essere memorizzati sul topic Kafka, la prima operazione da fare è quella di registrare lo schema all'interno di Confluent Schema Registry, invocando le API RESTful messe a disposizione dal servizio. A questo punto, quando NiFi vorrà pubblicare dei dati su Kafka dovrà prima di tutto convertirli nel formato Avro richiesto da quel determinato topic. Per fare ciò, interroga lo Schema Registry che ritornerà l'id associato al nome dello schema Avro richiesto. Ricevuta la risposta dal servizio, NiFi potrà serializzare i dati nel modo corretto e pubblicarli sul topic Kafka specificato, nel nostro caso chiamato topicRawData. Successivamente, grazie ai connettori Cassandra Sink e HDFS Sink, dopo aver recuperato lo schema Avro dal servizio Schema Registry, sarà possibile deserializzare i dati nel formato di partenza e inviarli rispettivamente a Cassandra e HDFS.

2.3.3 Flink Operators

Grazie a Flink è possibile creare applicazione che eseguono trasformazioni su stream di dati, come ad esempio: filtering, definizione di finestre, aggiornamento dello stato, aggregazione. Ogni programma necessita di cinque parti: dichiarare l'ambiente di esecuzione; importare uno o più stream dati; specificare le trasformazioni su questi dati; indicare dove scrivere i risultati ottenuti; scatenare l'esecuzione del programma. Per quanto riguarda le trasformazioni messe a disposizione dalla libreria DataStream API di Flink, tra le più importanti ricordiamo:

- Map: permette di prendere un elemento in input e trasformarlo in un nuovo elemento. Ad esempio, preso un datastream di interi vogliamo ottenere un nuovo stream in cui per ogni dato in ingresso si ottiene il corrispettivo valore raddoppiato.

- FlatMap: dato un elemento ritorna zero, uno o più elementi in output. Un possibile caso d'uso riguarda lo split di una frase in parole singole.

- Filter: utile per ritornare solo gli elementi che verificano una determinata funzione booleana. Ad esempio, selezionare solo gli elementi diversi da zero a partire da un datastream di interi.

- KeyBy: permette di dividere, a livello logico, uno stream in diverse partizioni, ognuna delle quali contiene solo gli elementi con la stessa chiave. Trasforma un'istanza di DataStream in un nuovo oggetto di tipo KeyedStream. Operazione spesso necessaria, in quanto, alcune trasformazioni per essere applicate richiedono che sia definita una chiave su una collezione di elementi.

- Reduce: per ogni insieme di dati con la stessa chiave, combina i vari elementi in un unico risultato finale. Se consideriamo il caso in cui una frase viene splittata in parole singole e utilizziamo come chiave la parola stessa, allora, tramite quest'operatore sarà possibile, ad esempio, contare il numero di occorrenze per ogni parola all'interno della frase.

- Aggregations: per applicare operazioni di aggregazione (come somma, ricerca del minimo e del massimo) su una collezione di elementi che possiedono la stessa chiave.

- Union: permette di unire due o più datastream in un unico stream che contiene tutti gli elementi.

- Split: separa un datastream in due o più stream secondo specifiche condizioni. Ritorna un oggetto di tipo SplitStream.

- Iterate: permette di ripetere un insieme di operazioni su un datastream che viene continuamente aggiornato ad ogni iterazione.

- Extract Timestamps: per estrarre il timestamp presente in un record; utilizzato per lavorare con window che usano la semantica Event Time.

Un discorso a parte, invece, merita l'operatore di Window. Esso permette di dividere uno stream di dati in "contenitori" di dimensione finita, sui quali è possibile effettuare diverse operazioni. Una window viene creata non appena arriva il primo elemento appartenente al datastream su cui vogliamo applicare l'operatore, mentre termina il suo ciclo di vita quando viene oltrepassato un determinato istante temporale, che può essere ritardato di una quantità specificata dall'utente. Inoltre, l'operatore di window permette di lavorare anche su datastream partizionati logicamente secondo una chiave. In quest'ultimo caso, sarà possibile eseguire le operazioni all'interno della finestra in parallelo, in quanto ogni stream logico potrà essere processato in maniera indipendente dagli altri.
Flink permette di assegnare ogni elemento in ingresso a uno o più tipi di windows. E' possibile utilizzare quelle messe a disposizione dal framework oppure implementare la propria window personalizzata estendendo la classe WindowAssigner. Quelle predefinite fornite da Flink, a eccezione della global window, permettono di assegnare gli elementi in base al tempo di arrivo e vengono chiamate: Tumbling Windows, Sliding Windows, Session Windows.
Tumbling Windows sono caratterizzate da: una window size, ad esempio cinque minuti, che stabilisce quanto dura il ciclo di vita della finestra, e quindi, ogni quanto tempo viene creata una nuova window; nessuna sovrapposizione, ovvero ogni elemento che arriva sarà assegnato ad un'unica finestra.
Sliding Windows definiscono finestre di dimensione predefinita, tramite il parametro window size, ma a differenza del caso precedente, permettono di stabilire ogni quanto intervallo di tempo è possibile avviare una nuova finestra, tramite il parametro window slide. Quindi, è possibile che si verifichi una sovrapposizione tra più finestre nel caso in cui window slide risulta minore di window size; in tal caso gli elementi saranno assegnati a più finestre.
Session Windows si discostano notevolmente dal comportamento delle precedenti windows, in quanto presentano: dimensione variabile per ogni finestra, non vi sono un istante di tempo iniziale e finale prefissati; nessuna sovrapposizione fra più windows. Una session window termina quando essa non riceve elementi per un determinato intervallo di tempo; a questo punto se arriveranno nuovi dati verrà creata una nuova session window. Il parametro che definisce il periodo di inattività della finestra viene chiamato session gap.
Infine, le Global Windows assegnano tutti gli elementi ad un'unica finestra globale. Vengono utilizzate insieme ad altri costrutti, chiamati trigger, altrimenti nessuna

operazione verrà mai eseguita, in quanto, tali finestre non presentano una fine naturale.

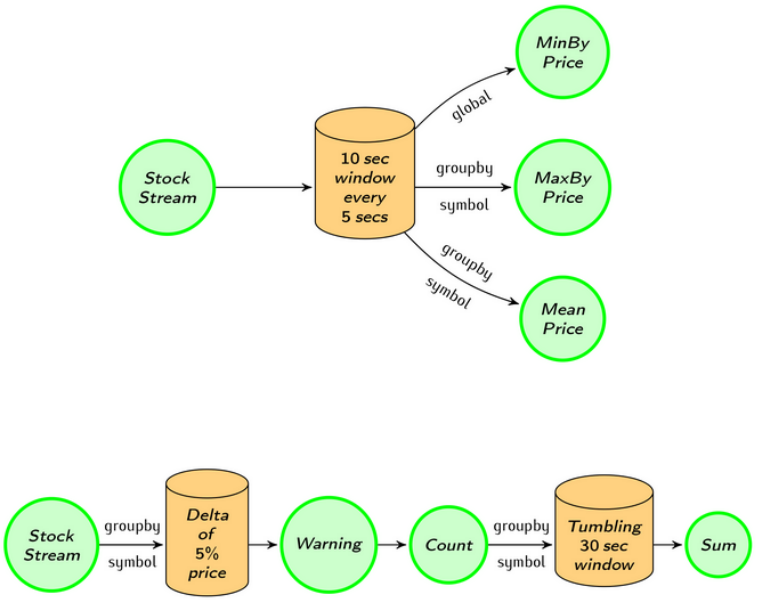


Figura 2.9. Flink Use Cases [24]

Nel caso del nostro sistema, gli operatori di window risulteranno molto utili per l'analisi di stream di dati. Consideriamo, infatti, il caso di uno stream dati che trasporta continuamente informazioni sui valori delle azioni in tempo reale. In figura 2.9, vengono mostrati due possibili casi d'uso realizzabili tramite Flink. Nel primo caso, utilizzando una sliding window con window size di dieci secondi e window slide di cinque secondi, è possile ottenere ogni cinque secondi delle statistiche riferite agli ultimi dieci secondi come ad esempio il prezzo minimo tra tutte le azioni oppure il valore massimo per ogni azione.
Nel secondo caso, invece, viene utilizzata una finestra per calcolare la variazione tra il prezzo precedente e quello corrente per ogni azione. Se il delta supera il 5%, l'evento verrà segnalato tramite un warning. Inoltre, una tumbling window, ci permetterà di contare il numero di warning lanciati negli ultimi trenta secondi per ogni azione.

2.3.4 Cassandra Keyspace

CQL è un linguaggio tipizzato simile a SQL che permette di definire un insieme di operazioni per la gestione dei dati all'interno di Cassandra. Tra i vari tipi di dati supportati abbiamo: ascii, per rappresentare una stringa di caratteri ascii; bigint, equivale a un long con segno su 64 bit; blob, sequenza di byte arbitraria; boolean; double; float; int; text, equivale a una stringa con codifica UTF8; timestamp, per indicare data e ora; uuid, universally unique identifier. Inoltre, CQL supporta tuple, tipi definiti dall'utente e tre tipi di collezioni: mappe, coppie key-value ordinate per chiave; set, sequenze ordinate di valori unici; e liste, insieme di elementi non-unici.
Prima di ricevere i record da Kafka, occorre creare le strutture dati necessarie a memorizzare lo stream di dati. A tal proposito, Cassandra permette di creare delle tabelle, raggruppate in keyspace. Quest'ultimo definisce due opzioni valide per tutte le tabelle contenute all'interno: replication, indica la strategia di replicazione e il fattore di replicazione; durable writes, permette di scegliere se usare il log di commit per le scritture nel keyspace. Nel caso di SimpleStrategy con replication factor pari a tre, si ottengono tre repliche dei dati all'interno del cluster per tutte le tabelle appartenenti a quel keyspace.
La creazione di una tabella, invece, avviene definendo un insieme di colonne; per ognuna sarà necessario specificare nome e tipo di dato, che indica quali valori sono accettati per quella colonna. All'interno di una tabella, una riga è identificata univocamente dalla sua primary key che può essere composta da una o più colonne; nel secondo caso, l'ordine è importante, in quanto, la prima colonna definisce la partition key, mentre le restanti rappresentano le cosiddette clustering columns. Tutte le righe di una tabella con la stessa partition key sono memorizzate sugli stessi nodi del cluster, di conseguenza effettuare delle query su tali righe richiede di contattare il minor numero di nodi. La scelta della primary key è un aspetto fondamentale in Cassandra poichè influenza le performance del sistema quando vengono eseguite le query.
L'ordine delle clustering columns definisce come le righe all'interno della stessa partizione vengono ordinate, ovvero il clustering order. Di default, si utilizza un ordinamento ascendente per tutte le clustering columns, altrimenti è possibile specificare il proprio clustering order, specificando per ogni colonna il tipo di ordinamento desiderato. Oltre a determinare il modo in cui saranno memorizzate le righe, il clustering order influisce anche sulla modalità in cui verrano ritornati i risultati. Infatti, quando viene effettuata una query, l'ordine con cui vengono restituiti i risultati equivale al clustering order all'interno della partizione. Se viene utilizzata la clausola ORDER BY, è possibile invertire l'ordine di visualizzazione, anche se ciò potrebbe avere un piccolo impatto sulle performance.
Per visualizzare i dati presenti in una tabella, CQL fornisce l'istruzione SELECT

seguita da una clausola di selezione. Quest'ultima permette di scegliere quali colonne visualizzare di una o più righe della tabella. Inoltre, è possibile applicare ai risultati ottenuti delle funzioni come Count, Max, Min, Sum, Avg o altre specificate dall'utente. E' possibile includere anche: la clausola WHERE, per definire quali righe devono essere interrogate; la clausola GROUP BY, per raggruppare i risultati in una singola riga; la clausola ORDER BY, per visualizzare i risultati secondo l'ordine desiderato, in relazione al clustering order; la clausola LIMIT, per limitare il numero di righe ritornate dalla query.

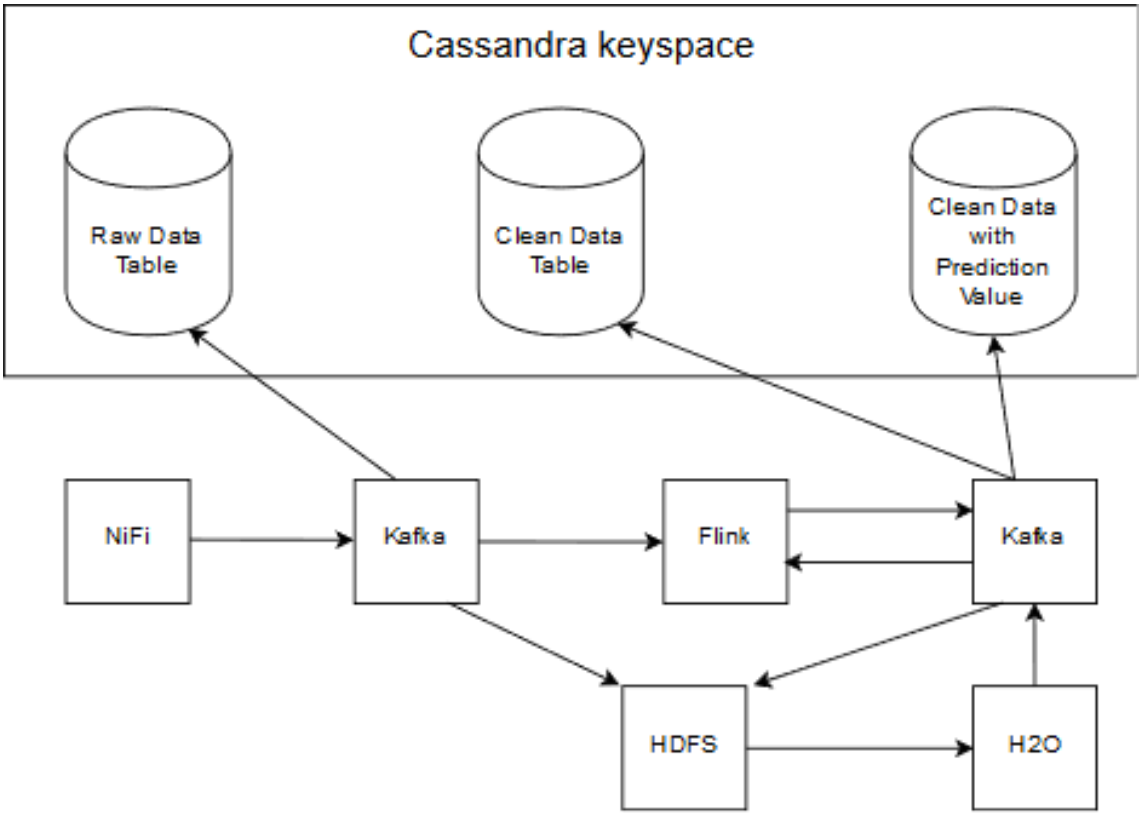


Figura 2.10. Tables in Cassandra Keyspace

Come mostrato in figura 2.10, nel nostro sistema, sarà necessario creare almeno tre diverse tabelle: la prima contenente i dati raw generati dalle sorgenti e ottenuti tramite Apache NiFi; la seconda, invece, memorizzerà i risultati prodotti dalla fase di stream processing; la terza verrà utilizzata per salvare i dati ottenuti dopo aver sottomesso il modello di machine learning allo stream. Ovviamente, per ogni tabella disponibile su Cassandra, esisterà la corrispettiva directory su HDFS che si occuperà di archiviare lo storico dei dati. Infine, prima di iniziare il trasferimento dei dati su

Cassandra sarà necessario creare, per ogni tabella, un set di colonne caratterizzate da tipi coerenti con le informazioni contenute all'interno del particolare stream dati.

2.3.5 H2O Model

H2O Flow rappresenta l'interfaccia utente per svolgere le diverse funzionalità offerte dal framework di machine learning come: importare file, creare e valutare modelli, effettuare predizioni, esportare modelli nell'ambiente di esecuzione.
Per quanto riguarda il nostro sistema, ci occuperemo di importare su H2O i file residenti su HDFS specificando il percorso della directory, all'interno del file system distribuito, in cui sono archiviati i dati per addestrare il modello di machine learning. Una volta caricati i file, è necessario passare alla fase di parsing che permette ad H2O di riconoscere il tipo dei dati importati. La lista dei parser disponibili include diverse opzioni tra cui AVRO, PARQUET e CSV. In caso di dati mancanti all'interno del dataset, potremmo rimpiazzarli con nuovi valori tramite il processo di data imputation fornito da H2O.
Per quanto riguarda invece la creazione dei modelli di machine learning, abbiamo due scelte possibli: optare per uno degli algoritmi disponibili oppure selezionare la modalità AutoML. Nel primo caso, sarà necessario settare molti parametri (di base, avanzati e per esperti) che possono cambiare a seconda dell'algoritmo scelto, mentre nel secondo caso H2O richiederà all'utente di configurare solo poche impostazioni e si occuperà di effettuare in maniera automatica il training e il tuning di diversi modelli, testando più algoritmi. I parametri richiesti per l'Automatic Machine Learning sono:

- y: per indicare il nome della response column.
- training frame: per specificare il training set.
- max runtime secs: controlla l'intervallo di tempo massimo consentito per l'esecuzione di AutoML.
- max models: definisce il numero massimo di modelli che potranno essere costruiti durante l'esecuzione di AutoML.

Una volta creato il nostro modello, ci occuperemo di convertirlo in Plain Old Java Object (POJO) oppure Model Object Optimized (MOJO) in modo tale da esportarlo facilmente in un ambiente Java. Il nostro obiettivo è quello di sottomottere il modello di machine learning allo stream dati, processato da Flink, tramite un'applicazione Java che utilizzi le API Kafka Streams. In questo modo riusciremo ad ottenere delle predizioni in tempo reale che arricchiranno ulteriormente la quantità di informazioni che sarà possibile estrarre dai dati. Inoltre, grazie al connettore tra Flink e Kafka, sarà possibile importare i nuovi risultati all'interno di Flink e confrontare in tempo reale le predizioni con i valori reali ottenuti processando il nuovo

stream di dati.
Poichè la qualità del modello dipende dal dataset utilizzato per l'apprendimento automatico, maggiore sarà la quantità di dati contenuta all'interno del dataset, più accurate saranno le predizioni effettuate sul nuovo stream in ingresso al sistema. Di conseguenza, per migliorare la qualità delle informazioni che è possibile ottenere, di volta in volta sarà necessario creare un nuovo modello di machine learning, esportarlo nuovamente in formato POJO/MOJO e riavviare l'applicazione Kafka Stream che si occupa di sottomettere il modello al nuovo stream dati. Quest'ultima fase non interromperà il processo di analisi sui dati eseguito da Flink, in quanto saranno utilizzate due applicazioni separate e indipendenti; inoltre dopo il riavvio l'applicazione Kafka Stream riprenderà da dove aveva terminato in quanto viene tenuta traccia dell'ultimo offset consumato.

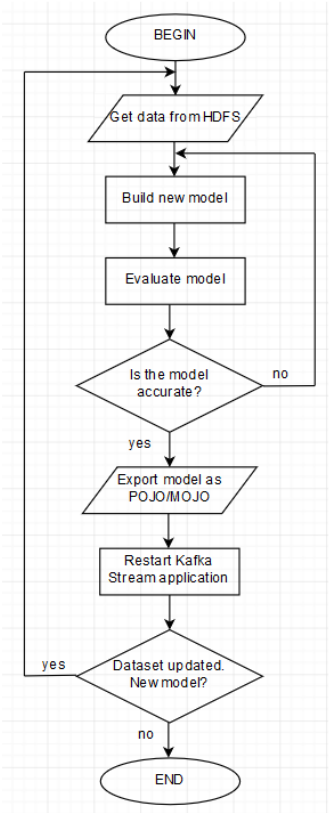


Figura 2.11. Lifecycle machine learning model

In figura 2.11, viene rappresentato il ciclo di vita di un modello di machine learning.

Capitolo 3

Realizzazione di un'infrastruttura per l'analisi di dati streaming

In questo capitolo ci occuperemo di descrivere come è stata realizzata l'infrastruttura Big Data per ingestion real-time, discutendo i vari step di configurazione per ogni framework utilizzato nell'architettura. L'obiettivo principale è quello di implementare un sistema che garantisse continuamente il corretto funzionamento dei servizi richiesti, anche in caso di possibili guasti. A tal proposito, in aggiunta ai framework scelti durante la fase di progettazione, si è reso necessario installare Apache ZooKeeper per fornire supporto ai servizi: NiFi, Kafka, Flink, HDFS.
Per quanto riguarda le risorse di elaborazione, sono state utilizzate cinque istanze di server virtuali attraverso il servizio Web di Amazon Elastic Compute Cloud (Amazon EC2). Ogni macchina presenta le stesse capacità; in particolar modo viene utilizzato il modello di istanza t2.large che fornisce 2 cpu virtuali ad alta frequenza Intel Xeon e 8 GB di RAM, in modo da garantire prestazioni elevate per lunghi periodi di tempo. La capacità di storage per ogni macchina equivale a 50 GB. Il sistema operativo installato è Red Hat Enterprise Linux versione 7.4_HVM_GA mentre per quanto riguarda la configurazione di rete abbiamo: abilitato l'accesso a internet su ogni macchina; utilizzato una rete privata per la comunicazione all'interno del cluster; consentito l'accesso alle macchine dall'esterno tramite SSH. Inoltre, si è reso necessario abilitare le connessioni in ingresso sulle porte in cui vengono esposte le interfacce utente web fornite dai servizi: NiFi, Flink, HDFS, H2O.
Per ogni macchina abbiamo modificato il file /etc/hosts per far corrispondere all'indirizzo ip privato di ogni macchina un nome di dominio dell'host, allo scopo di facilitare la configurazione dei servizi all'interno del cluster.
Nella tabella 3.1 vengono elencate, per ogni macchina all'interno del cluster, le associazioni effettuate tra hostname e indirizzo ip privato corrispondente.

ID	Hostname	Private IP address
1	hadoop1	172.31.16.37
2	hadoop2	172.31.23.139
3	hadoop3	172.31.20.178
4	hadoop4	172.31.29.7
5	hadoop5	172.31.16.220

Tabella 3.1. Hostname and private IP address for each machine

3.1 Apache ZooKeeper

ZooKeeper permette ai processi distribuiti di coordinarsi attraverso una struttura dati gerarchica condivisa simile a un file system ma mantenuta in memoria, garantendo alte performance. E' un servizio che utilizza un'insieme di server su cui vengono replicati i dati mantenuti all'interno, in modo da fornire high availability in caso di guasti. Ogni server ZooKeeper può servire le richieste del client eccetto le operazioni di scrittura che potranno essere eseguite solamente da un server eletto come leader, a differenza degli altri che saranno definiti come followers.
L'affidabilità del sistema viene garantita quando la maggioranza dei server che formano il cluster è attiva, quindi, se abbiamo tre macchine su cui è installato ZooKeeper, questo può gestire il guasto di un solo server; per un cluster formato da cinque macchine, invece, il sistema potrà tollerare la perdita di due server ZooKeeper. Quest'ultimo caso è particolarmente utile quando occorre effettuare la manutenzione di un server, infatti, con sole tre macchine, se dovessimo spegnere un server per manutenzione renderemmo l'intero sistema vulnerabile ai guasti. Per tale motivo, si è deciso di installare ZooKeeper su ogni macchina del cluster e avere così cinque istanze di server continuamente in esecuzione durante il funzionamento del sistema.
Per quanto riguarda l'installazione, è stata utilizzata l'ultima release stabile disponibile durante il lavoro di tesi, equivalente alla versione 3.4.10. I requisiti software per tale servizio includono una versione di Java Development Kit uguale o superiore alla 1.6 e almeno tre istanze di server ZooKeeper, possibilmente che eseguono su macchine separate.
Dopo aver effettuato il download del package contenente il software e aver estratto il contenuto, si è reso necessario editare due file di configurazione. Il primo è chiamato java.env e viene utilizzato per cercare di eliminare lo swapping che potrebbe degradare in maniera evidente le performance di ZooKeeper. Nel nostro caso si è deciso di settare a 2 GB la dimensione massima dell'heap utilizzabile dalla JVM, inserendo la riga `export JVMFLAGS="-Xmx2048m"`.
Il secondo file, chiamato zoo.cfg, è stato configurato, come mostrato in figura 3.1, con le seguenti proprietà: tickTime, utilizzato per regolare le sessioni di timeout,

misura in millisecondi la durata di un'unità di tempo per ZooKeeper, chiamata tick; dataDir, indica la posizione in cui vengono memorizzate le informazioni necessarie al funzionamento di ZooKeeper; clientPort, definisce il numero di porta verso cui devono connettersi i client per poter usufruire dei servizi offerti da ZooKeeper; initLimit, definisce il numero massimo di tick per consentire ai followers di connettersi a un leader (in questo caso è stato settato a 10 secondi), maggiore è la quantità di dati gestita da ZooKeeper più grande deve essere questo valore, in quanto, alla prima connessione vi è la necessità di scambiare più informazioni; syncLimit, specifica il numero massimo di tick affinchè un follower possa sincronizzarsi con ZooKeeper. Infine, vi è una lista di `server.id=host:port:port` per indicare la lista dei server su cui viene eseguito ZooKeeper. In particolare, viene specificato un id diverso per ogni server, il nome dell'host associato, il numero di porta 2888 per la comunicazione tra followers e leader e il numero 3888 per l'elezione del leader.

```
tickTime=2000
dataDir=/tmp/zookeeper
clientPort=2181
initLimit=5
syncLimit=2

server.1=hadoop1:2888:3888
server.2=hadoop2:2888:3888
server.3=hadoop3:2888:3888
server.4=hadoop4:2888:3888
server.5=hadoop5:2888:3888
```

Figura 3.1. Zoo.cfg properties for each ZooKeeper node

Infine, per ogni server su cui esegue ZooKeeper abbiamo creato il file myid, nel percorso specificato da dataDir. Tale file contine l'id associato a quella macchina, come specificato in zoo.cfg (ad esempio, su hadoop1 myid conterrà `1`, mentre su hadoop2 `2` e così via).

3.2 Apache NiFi

NiFi utilizza il paradigma Zero-Master Clustering in cui i dati vengono divisi in set e ogni nodo esegue le stesse operazioni su diversi insiemi di dati. Tramite ZooKeeper, uno dei nodi sarà eletto come Cluster Coordinator, il quale sarà responsabile di fornire la versione più aggiornata del fusso di dati agli altri nodi. Ogni cluster definisce anche un Primary Node attraverso il quale è possibile eseguire uno o più

processori in modalità isolata, evitando così che gli altri nodi eseguano la stessa operazione sui dati. Nel caso in cui si verifichi un guasto su uno dei nodi, sia esso Cluster Coordinator o Primary Node, grazie a ZooKeeper verrà eletto un sostituto per tale ruolo senza interrompere il funzionamento del sistema.
Per quanto riguarda l'implementazione della fase di ingestion, si è scelto quindi di configurare NiFi in modalità cluster in modo tale da poter processare una quantità di dati maggiore, distribuendo il carico tra più nodi all'interno del cluster. A tale scopo si è deciso di installare il framework sui nodi: hadoop1, hadoop2, hadoop3. La release utilizzata è la 1.4.0 e richiede una versione Java uguale o superiore alla 8. Per abilitare la modalità cluster in NiFi, si è reso necessario modificare il file di configurazione nifi.properties. In figura 3.2, vengono mostrate alcune proprietà impostate per la macchina hadoop1.

```
# State Management #
nifi.state.management.embedded.zookeeper.start=false

# Site to Site properties
nifi.remote.input.host=hadoop1
nifi.remote.input.socket.port=9997

# web properties #
nifi.web.http.port=8080
nifi.web.http.network.interface.default=

# cluster node properties #
nifi.cluster.is.node=true
nifi.cluster.node.address=hadoop1
nifi.cluster.node.protocol.port=9998
nifi.cluster.flow.election.max.candidates=3

# zookeeper properties, used for cluster management #
nifi.zookeeper.connect.string=hadoop1:2181,hadoop2:2181,hadoop3:2181,
hadoop4:2181,hadoop5:2181

nifi.zookeeper.root.node=/nifi
```

Figura 3.2. Nifi.properties for hadoop1

Per quanto riguarda la sezione relativa allo State Management, NiFi da la possibilità di eseguire anche in un ambiente in cui non è presente ZooKeeper, avviando un'istanza all'interno di un server ZooKeeper embedded. Poichè questo non è il nostro

caso, abbiamo settato a false la proprietà corrispondente.
La sezione Site to Site Properties definisce come avviene la comunicazione tra i nodi per fornire il bilanciamento del carico all'interno del cluster, definendo per ogni macchina hostname e numero di porta utilizzati. Nella parte relativa alle Web Properties, invece, viene indicato l'indirizzo web su cui viene esposta l'interfaccia utente web-based. Poichè non abbiamo un indirizzo ip pubblico fisso (in quanto Amazon EC2 cambia DNS pubblico a ogni riavvio della macchina), per collegarsi all'interfaccia utente web dall'esterno, si è reso necessario fare in modo che NiFi ascoltasse alla porta specificata su tutte le interfacce di rete della macchina disponibili (lasciando la proprietà `nifi.web.http.network.interface.default` vuota).
Nella sezione Cluster node, vengono indicate le proprietà utilizzate per la comunicazione tra i nodi all'interno del cluster, ad esempio, durante la fase di elezione del Coordinator e Primary Node. Infine, per quanto riguarda le ZooKeeper Properties abbiamo impostato la stringa di connessione utilizzata dai client per il collegamento ai server ZooKeeper e la root node, in cui vengono mantenute le informazioni per il coordinamento tra i nodi che formano il cluster NiFi.
A questo punto, una volta avviati ZooKeeper e NiFi, verranno selezionati un Cluster Coordinator e un Primary Node tra i nodi hadoop1, hadoop2 e hadoop3. Nella figura 3.3 viene mostrata l'interfaccia utente di NiFi Cluster da cui è possibile osservare come il nodo hadoop1 sia stato eletto come Coordinator mentre hadoop2 come Primary.

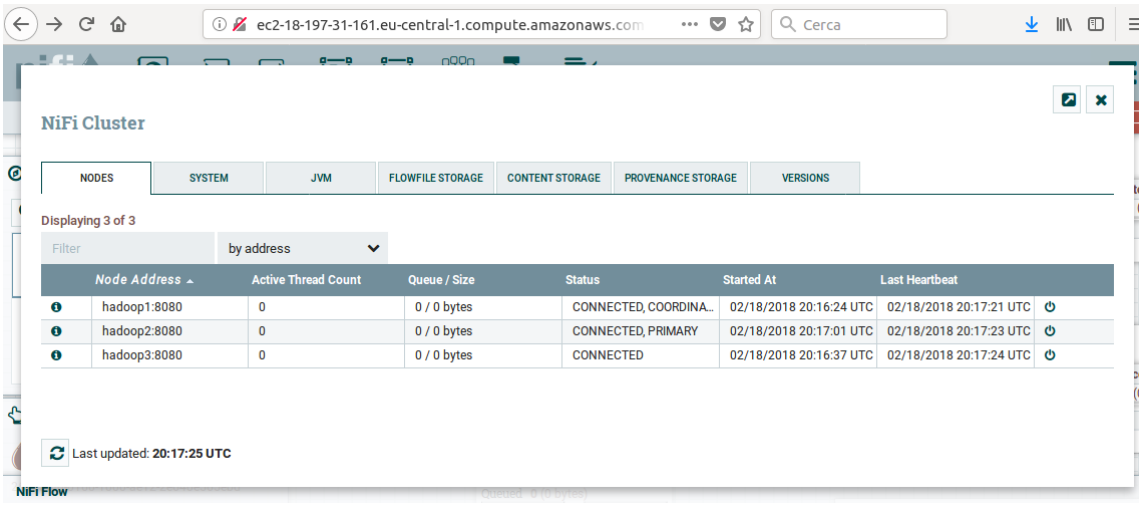


Figura 3.3. NiFi Cluster User Interface

Nel caso in cui si verifichi un guasto su uno dei nodi che formano il cluster NiFi, ZooKeeper garantisce che il sistema continui a funzionare correttamente. Ritornando all'esempio precedente, supponiamo che il nodo hadoop1 non sia più disponibile;

di conseguenza occorrerà rieleggere un nuovo Cluster Coordinator fra i nodi rimanenti. Come si vede dalla figura 3.4, in tal caso hadoop2 è stato eletto come nuovo Coordinator.

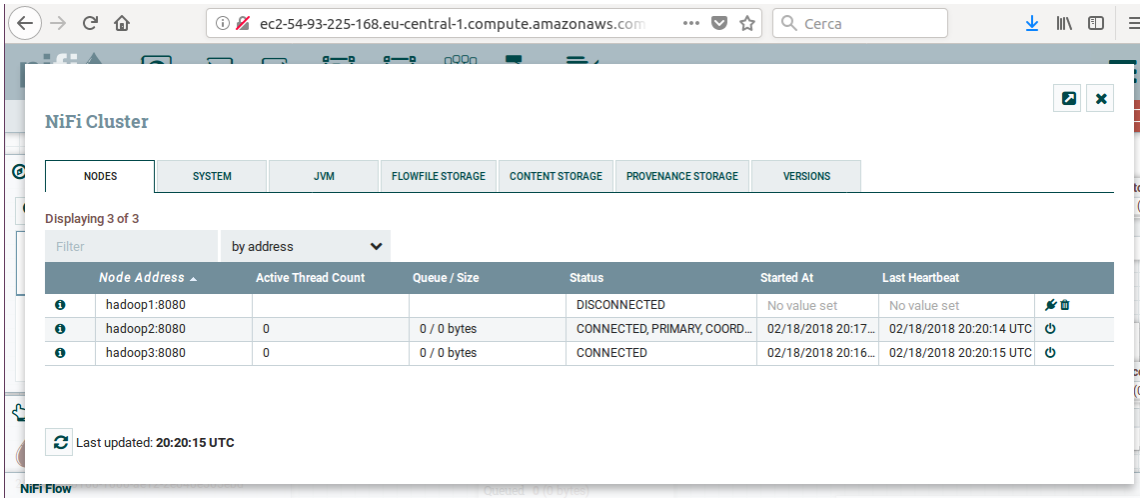


Figura 3.4. NiFi Cluster after hadoop1 goes down

3.2.1 PublishKafkaRecord Processor

Per ogni processore disponibile, NiFi definisce un'insieme di proprietà che è possibile settare. In figura 3.5, vengono mostrate alcune opzioni di configurazione per il processore PublishKafkaRecord, in accordo al nostro sistema: Kafka Brokers, per indicare la lista dei broker utilizzati dal cluster Kafka; Topic Name, dichiara su quale topic verrà pubblicato il contenuto del FlowFile; Record Reader, definisce lo schema con cui sono letti i dati all'interno del FlowFile; RecordWriter, indica il formato in cui verranno serializzati i record prima di essere inviati a Kafka; Use Transactions, permette di decidere se NiFi dovrebbe utilizzare funzioni di transazione di per la comunicazione con Kafka; Delivery Guarantee, permette di impostare i requisiti per la garanzia di consegna dei messaggi; Security Protocol, determina quale protocollo utilizzare per comunicare con i broker. Altre proprietà non visibili dall'esempio in figura sono: Max Request Size: per indicare la dimensione massima in byte dei messaggi; Compression Type: permette di specificare la codifica di compressione che per tutti i dati trasmessi su Kafka.

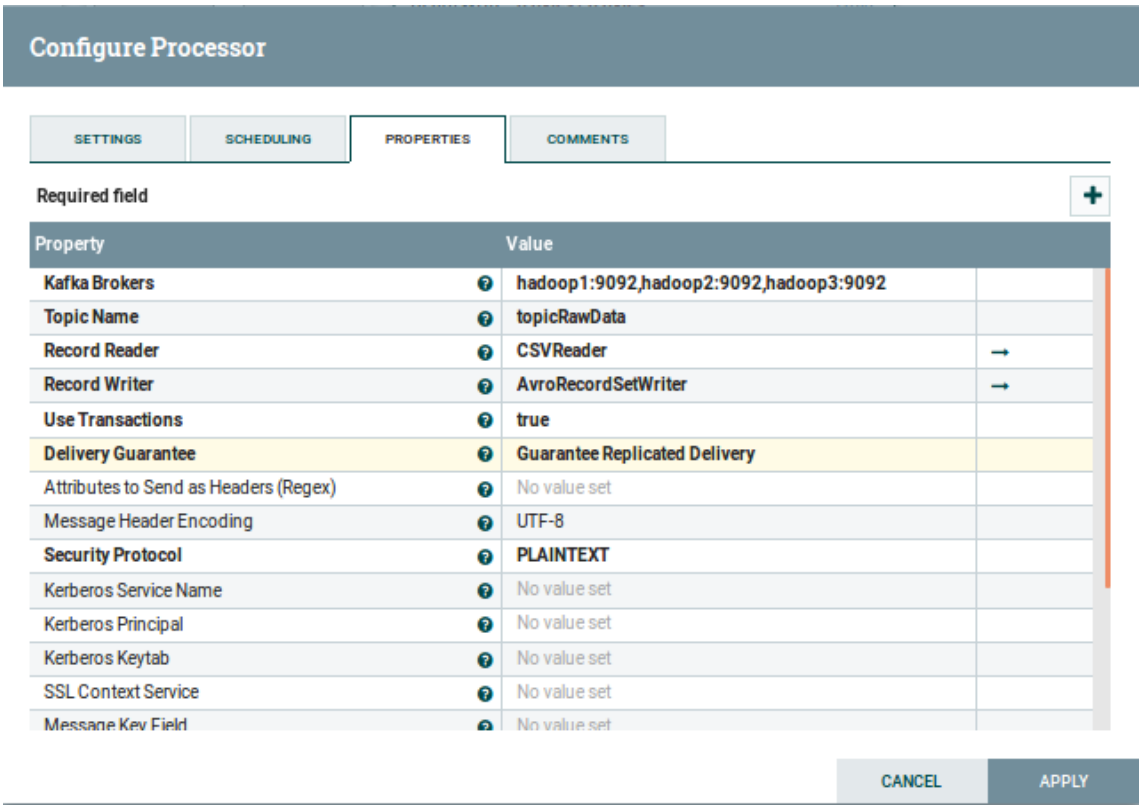


Figura 3.5. PublishKafkaRecord Processor

3.3 Apache Kafka

Per implementare le funzionalità offerte dallo strato di Message Broker, si è reso necessario installare Kafka insieme ad alcuni package distribuiti dalla piattaforma open-source Confluent. In modo particolare, verrà utilizzato il servizio schema-registry per fornire supporto allo serializzazione e deserializzazione di dati espressi in formato Avro. Schema Registry permette di registrare e recuperare una lista di schemi Avro utilizzabili da Kafka per il trasferimento dei messaggi. Il motivo principale di tale scelta è dovuto al fatto che per pubblicare i dati su HDFS e Cassandra il formato Avro risulta la soluzione migliore.
Kafka Connect, invece, è un tool che permette di scambiare stream di dati tra Kafka e sistemi esterni in modo scalabile e affidabile. Permette di eseguire dei connettori in cui è possibile definire un insieme di proprietà per il trasferimento di dati da Kafka verso un sistema esterno o viceversa. Per pubblicare i dati su HDFS verrà utilizzato HDFS Connector fornito dalla piattaforma Confluent, mentre per scrivere su Cassandra verrà utilizzato il connettore Cassandra Sink sviluppato da DataMountaineer.

Per quanto riguarda la modalià di installazione, si è scelto di implementare un cluster Kafka formato da tre broker per fornire al servizio proprietà di scalabilità, fault-tolerant e high availability. A tal proposito, si è reso obbligatorio ZooKeeper per il coordinamento tra i vari nodi. Le macchine scelte per formare il cluster Kafka sono: hadoop1, hadoop2, hadoop3. Su ognuna di esse saranno anche installati i servizi Kafka Connect e Schema Registry.
Per implementare i servizi descritti finora abbiamo effettuato il download della versione 4.0 della piattaforma open-source Confluent. Una volta disponibile, il primo step è stato quello di modificare il file di configurazione server.properties.
Nell'esempio di figura 3.6, è possibile notare alcune proprietà impostate per la macchina hadoop1.

```
broker.id=1

listeners=PLAINTEXT://hadoop1:9092

log.dirs=/tmp/kafka-logs

log.retention.hours=168

zookeeper.connect=hadoop1:2181,hadoop2:2181,hadoop3:2181,
hadoop4:2181,hadoop5:2181/kafka
```

Figura 3.6. Server.properties for hadoop1

Per quanto riguarda la proprietà `broker.id`, essa permette di distinguere i vari broker che formano il cluster (per hadoop2 e hadoop3 abbiamo assegnato come id rispettivamente `2` e `3`). La proprietà `listeners` rappresenta l'indirizzo su cui è in ascolto il socket insieme al protocollo di sicurezza; la proprietà `log.dirs`, invece, definisce il percorso della directory in cui verranno memorizzati i file di log per tenere traccia degli offset su ogni topic per ogni producer e consumer Kafka. Passando alla proprietà `log.retention.hours`, essa specifica l'intervallo di tempo dopo il quale devono essere cancellati i file di log (di default è impostata a 168, che equivale a una settimana). Successivamente, si è reso necessario modificare la proprietà `zookeeper.connect` per definire l'indirizzo dei server su cui esegue ZooKeeper, specificando anche la root directory in cui verranno registrate le informazioni necessarie per il coordinamento dei broker Kafka.
Per quanto riguarda, invece, il servizio schema-registry, abbiamo modificato il file schema-registry.properties. Per ogni istanza Kafka, la proprietà `listeners` è stata settata allo stesso modo di come fatto per il file server.properties con l'eccezione di aver cambiato la porta su cui è disponibile il servizio da `9092` ad `8081`, mentre la

proprietà `kafkastore.bootstrap.server`, rappresenta la lista degli indirizzi su cui ascoltano i broker Kafka, ovvero, `hadoop1:9092,hadoop2:9092,hadoop3:9092`.
Passando alle proprietà relative al servizio Kafka Connect, si è reso necessario configurare, su ogni nodo del cluster, il file connect-avro-distributed.properties. Innanzitutto, per la proprietà `bootstrap.servers` abbiamo settato lo stesso valore usato per `kafkastore.bootstrap.server`; per quanto riguarda le proprietà `key.converter.schema.registry.url` e `value.converter.schema.registry.url` abbiamo indicato la lista degli indirizzi dei server su cui viene eseguito schema-registry, cioè, `http://hadoop1:8081,http://hadoop2:8081,http:hadoop3:8081`; la proprietà `rest.hostname` indica su quale hostname è in ascolto il servizio Kafka Connect offerto dal server che stiamo configurando, per cui equivale ad `hadoop1` nel caso in cui ci stiamo riferendo al broker Kafka con id `1`, e così via; `rest.port`, invece, indica il numero di porta su cui viene esposto il servizio e quindi si è scelto il numero `8083`; la proprietà `plugin.path` permette di definire una lista di percorsi in cui si trovano i file jar contenenti le classi Java che definiscono i diversi connettori, nel nostro caso si ha `$CONFLUENT_HOME/share/java,$CONFLUENT_HOME/plugin`. Il primo percorso localizza il file jar relativo al connettore HDFS Sink, mentre il secondo rappresenta la cartella in cui abbiamo scaricato il connettore Cassandra Sink.

3.3.1 HDFS Sink Connector

Il connettore HDFS Sink esporta continuamente i dati dai topic Kafka e li scrive su HDFS; inoltre, si integra con Hive per fare in modo che questi siano immediatamente accessibili attraverso query in HiveQL. HDFS connector include numerosi aspetti tra cui: garanzia di consegna exactly once, il connettore tiene traccia degli offset relativi ai record che sono stati già scritti su HDFS, cosicchè, in caso di guasti, quando si riavvia il task, questo prosegue da dove era rimasto; formato dati estensibile, è possibile scrivere dati su HDFS in formato Avro o Parquet, oppure altri formati che estendano la classe Format; schema evolution, per la compatibilità tra diversi schemi; Secure HDFS, supporta l'autenticazione tramite Kerberos; Pluggable Partitioner, permette di definire la modalità in cui saranno partitizionati i file (che conterranno i record ricevuti da Kafka) su HDFS.
In figura 3.7, è possibile osservare il file di configurazione usato per esportare i dati raw da Kafka al file system distribuito di Hadoop. Tra le varie proprietà abbiamo: `name`, per indicare il nome del connettore; `connector.class`, rappresenta il nome della classe in cui sono implementate le funzionalità fornite dal connettore; `format.class`, definisce il formato dei dati usato per scrivere su HDFS; `tasks.max`, per indicare il numero di massimo di task che si occuperanno di trasferire i dati; `topics`, definisce i topic Kafka da cui vogliamo esportare i dati; `hdfs.url`, url di connessione verso HDFS; `hadoop.conf.dir`, specifica il percorso della directory in cui si trovano i file di configurazione di Hadoop; `flush.size`, definisce il numero

di record da scrivere su HDFS prima di invocare il commit; `logs.dir`, nome della directory in cui saranno memorizzati i log che terranno traccia delle operazioni di scrittura su HDFS per realizzare garanzia di consegna exactly-once; `topics.dir`, indica il nome della directory padre su HDFS in cui saranno creati i file che conterranno i dati ricevuti da Kafka.

```
name=hdfs-sink-topicrawdata

connector.class=io.confluent.connect.hdfs.HdfsSinkConnector
format.class=io.confluent.connect.hdfs.avro.AvroFormat

tasks.max=1
topics=topicRawData

hdfs.url=hdfs://ha-cluster
hadoop.conf.dir=$HADOOP_HOME/etc/hadoop/

flush.size=100

logs.dir=/topicRawData-logs
topics.dir=/topicRawData-topics
```

Figura 3.7. HDFS Sink Connector properties

3.3.2 Cassandra Sink Connector

Cassandra Sink Connector è un tool sviluppato da DataMountaineer per semplificare la scrittura dei dati da Kafka verso Cassandra. Internamente, si occupa di convertire il record in formato Json e quindi trasferisce i dati su Cassandra in modalità asincrona. Il connettore fornisce gli aspetti seguenti: Field Selection, permette di selezionare tramite Kafka Connect Query Language (KCQL) quali campi di un record scrivere su Cassandra; Error policies, defisce la politica adoperata per la gestione dei guasti, ad esempio informando il connettore di redistribuire i messaggi in caso di errori durante il trasferimento; Payload support, permette il supporto verso messaggi che trasportano le informazioni sullo schema, come ad esempio Avro, oppure messaggi senza schema ma con payload Json; Optional TTL, si riferisce all'iintervallo di tempo massimo consentito per scrivere dati su una tabella Cassandra; SSL support, per connessioni sicure; cancellazione di un record Kafka trasmesso con payload vuoto.
In figura 3.8, viene mostrato il contenuto del file di configurazione usato per trasferire

i dati raw da Kafka verso Cassandra. In particolar modo, si è scelto di settare le seguenti proprietà: `name`, per indicare il nome del connettore; `connector.class`, specifica il nome della classe che implementa le funzionalità del connettore; `task.max`, indica il numero di processi che si occupa del trasferimento di dati tra i due sistemi; `topics`, indica il nome del topic Kafka da cui devono essere letti i dati; `connect.cassandra.kcql`, rappresenta l'istruzione scritta in kafka connect query language per selezionare quali dati copiare da topicRawData verso una tabella di Cassandra, chiamata tablerawdata; `connect.cassandra.contact.points`, indica gli hostname utilizzati dal cluster Cassandra; `connect.cassandra.port`, definisce il numero di porta su cui sono in ascolto i nodi che eseguono Cassandra; `connect.cassandra.key.space`, per specificare il nome del keyspace a cui appartiene la tabella su cui vengono esportati i dati; `connect.cassandra.username` e `connect.cassandra.password` definiscono le credenziali necessarie per connettersi a Cassandra.

```
name=cassandra-sink-topicrawdata

connector.class=com.datamountaineer.streamreactor.connect.cassandra
.sink.CassandraSinkConnector

tasks.max=1
topics=topicRawData

connect.cassandra.kcql=INSERT INTO tablerawdata SELECT * FROM topicRawData

connect.cassandra.contact.points=hadoop3,hadoop4,hadoop5
connect.cassandra.port=9042

connect.cassandra.key.space=mykeyspace
connect.cassandra.username=cassandra
connect.cassandra.password=cassandra
```

Figura 3.8. Cassandra Sink Connector properties

3.4 Apache Flink

Come discusso nel capitolo precedente, Apache Flink è il framework scelto per realizzare la fase di stream processing. Non solo permette di utilizzare le librerie necessarie

per sviluppare applicazioni che operano su stream di dati in modo scalabile e fault-tolerant, ma fornisce anche un ambiente di esecuzione distribuito che può essere definito da due tipi di processi: JobManagers, si occupano di schedulare i task, gestire le risorse, coordinare il recupero in caso di guasti; TaskManagers, eseguono i task e bufferizzano lo scambio di stream di dati. Entrambi possono essere eseguiti in vari modi: cluster standalone, Docker container, oppure gestiti da framework come YARN o Mesos. In un tale contesto, il client si occuperà soltanto di sviluppare l'applicazione di streaming e di sottometterla al JobManager. Possiamo affermare che tale processo rappresenta un componente fondamentale all'interno del sistema, in quanto, se il JobManager termina la sua esecuzione in caso di guasti, la parte di stream processing non funzionerebbe più come ci si aspetta. Per evitare che il JobManager crei un singolo punto di guasto, si è scelto di configurare Flink in modalità High Availability. Ciò permette di avere contemporaneamente un JobManager leader e più JobManager standby, in modo tale che quando il leader va in crash, viene subito eletto un nuovo master tra le istanze di JobManager in standby, cosicchè il sistema possa continuare a funzionare senza problemi. Per il coordinamento tra i processi JobManager distribuiti sui nodi del cluster, Flink utilizza ZooKeeper. In modo particolare, quest'ultimo si occuperà di rendere fortemente affidabile la fase di elezione del leader.
Nel nostro caso si è scelto di utilizzare due JobManager e tre TaskManager per consentire sia di rendere immune il sistema da un possibile guasto sul JobManager sia per poter eseguire in modo parallelo più task di streaming. I processi di JobManager saranno disponibili per le macchine hadoop1 e hadoop2, mentre quelli di TaskManager per le macchine hadoop3,hadoop4,hadoop5.
Per configurare il cluster Flink in modalità High Availability si è reso necessario editare il file masters inserendo le righe `hadoop1:9081` e `hadoop2:9081` per indicare hostname e numero di porta (alla quale è disponibile l'interfaccia utente web-based) dei JobManager. Invece, per indicare le informazioni relative ai TaskManager, è stato creato il file slaves contenente le seguenti righe: `hadoop3`, `hadoop4`, `hadoop5`. Infine, come mostrato in figura 3.9, all'interno del file flink-conf.yaml sono state impostate le seguenti proprietà: `high-availability`, per abilitare la modalità high availability all'interno del cluster Flink; `high-availability.zookeeper.quorum`, indica il gruppo di server ZooKeeper; `high-availability.zookeeper.path.root`, definisce il root node in ZooKeeper in cui vengono memorizzati i dati relativi al cluster Flink, `high-availability.zookeeper.storageDir`, directory per lo storage dei metadati utilizzati dal JobManager.
Per quanto riguarda il connettore che permette ad Apache Flink di leggere e scrivere dati da e verso Kafka, si utilizzerà la libreria flink-connector-kafka-0.11_2.11 che include le classi FlinkKafkaConsumer011 e FlinkKafkaProducer011.

```
high-availability: zookeeper

high-availability.zookeeper.quorum: hadoop1:2181,hadoop2:2181,
hadoop3:2181,hadoop4:2181,hadoop5:2181

high-availability.zookeeper.path.root: /flink

high-availability.zookeeper.storageDir: /tmp/flink
```

Figura 3.9. Flink-conf.yaml properties

3.4.1 Flink Kafka Consumer

FlinkKafkaConsumer011 permette di specificare: una lista di topic da cui recuperare i dati; un DeserializationSchema, per sapere come deserializzare il contenuto dei dati, rappresentato in byte, nell'oggetto Java desiderato; la lista dei broker Kafka e l'id del consumer group. Inoltre, è possibile configurare l'offset di partenza da cui vogliamo iniziare a leggere i dati di un topic Kafka. Per quanto riguarda, invece, la tolleranza ai guasti, Flink Kafka Consumer si integra con il meccanismo dei checkpoint per fornire garanzie di processamento exactly-once. In modo particolare, internamente Flink si occuperà di tener traccia degli offset dei record consumati da Kafka; in caso di job failure, il task di stream processing ripartirà dall'ultimo checkpoint e inizierà a consumare i record da Kafka partendo dagli offset che erano stati memorizzati durante l'ultimo checkpoint. Se il meccanismo dei checkpoint viene disabilitato, Kafka Consumer effettuerà periodicamente il commit degli offset consumati su ZooKeeper.
In figura 3.10, viene mostrato il codice Java utilizzato dalla nostra applicazione per esportare i dati raw dal topic Kafka verso Flink. Inizialmente, viene settato l'ambiente di esecuzione e abilitato il meccanismo dei checkpoint con un intervallo di 5 secondi. Nelle righe successive, vengono impostate le proprietà riguardanti il cluster Kafka: indirizzo dei broker; indirizzo dei server per la connessione a ZooKeeper; Kafka consumer group. Per quanto riguarda FlinkKafkaConsumer, si specifica: topic Kafka da cui leggere i dati; un DeserializationSchema che permetta di deserializzare i messaggi ricevuti in istanze dell'oggetto MyRawData; le proprietà relative al cluster Kafka. Come posizione di partenza da cui consumare i dati, viene settato il primo offset disponibile all'interno del topic. Infine, FlinkKafkaConsumer viene agganciata come sorgente del DataStream che trasporta oggetti di tipo MyRawData e viene invocata l'esecuzione dell'applicazione.

```
final StreamExecutionEnvironment env =
 StreamExecutionEnvironment.getExecutionEnvironment();

env.enableCheckpointing(5000);

Properties properties = new Properties();

properties.setProperty("bootstrap.servers",
"hadoop1:9092,hadoop2:9092,hadoop3:9092");

properties.setProperty("zookeeper.connect", "hadoop1:2181,
hadoop2:2181,hadoop3:2181,hadoop4:2181,hadoop5:2181/kafka");

properties.setProperty("group.id", "anonymous");

String schemaRegistryUrl =
 "http://hadoop1:8081,http://hadoop2:8081,http://hadoop3:8081";

String sourceTopic = "topicRawData";

FlinkKafkaConsumer011<MyRawData> myKafkaConsumer =
new FlinkKafkaConsumer011<>(
sourceTopic,
new MyKafkaAvroDeserializationSchema(sourceTopic,
schemaRegistryUrl, MyRawData.class),
properties);

myKafkaConsumer.setStartFromEarliest();

DataStream<MyRawData> rawDataStream = env.addSource(myKafkaConsumer);
env.execute("Flink Streaming Java API");
```

Figura 3.10. Flink Kafka Consumer example

3.4.2 Flink Kafka Producer

La classe FlinkKafkaProducer011, invece, permette di scrivere uno stream di record su uno o più topic Kafka. In questo caso, occorrerà specificare: lista dei broker Kafka; topic desiderato; SerializationSchema, per definire come serializzare l'oggetto Java in uno stream di byte. Inoltre, è possibile assegnare ogni record ad una partizione del

topic specifica estendendo la classe FlinkKafkaPartitioner e allegare un timestamp al record. Anche in questo caso, è possibile stabilire garanzie di consegna exactly-once abilitando il meccanismo dei checkpoint che Flink mette a disposizione.
In figura , viene mostrato un frammento di codice utilizzato per pubblicare su Kafka il contenuto di un datastream, processato da Flink. Supponendo di aver mantenuto le stesse proprietà per il cluster Kafka, definite nel sottoparagrafo precedente, dopo aver settato il topic su cui dovrà essere pubblicato lo stream di dati, verrà definito FlinkKafkaProducer in modo tale da: scrivere sul topic desiderato; serializzare i dati secondo il nostro SerializationSchema, utilizzare le proprietà che definiscono il cluster Kafka. Una volta istanziata tale classe, sarà possibile agganciare il nostro stream di dati al connettore Kafka in modo da avviare il trasferimento.

```
DataStream<MyCleanData> cleanDataStream = ...;

String destinationTopic = "topicCleanData";

FlinkKafkaProducer011<MyCleanData> myKafkaProducer = (destinationTopic,
new KafkaAvroSerializationSchema(destinationTopic,
schemaRegistryUrl),
properties));

cleanDataStream.addSink(myKafkaProducer);
```

Figura 3.11. Flink Kafka Producer example

3.5 Hadoop Distributed File System

Come sappiamo l'architettura di HDFS segue un modello di tipo Master/Slave in cui i processi NameNode si occupano di coordinare le attività necessarie per soddisfare le richieste dell'utente, mentre i DataNode sono responsabili dell'archiviazione dei dati all'interno del sistema. Solitamente, si utilizza un NameNode e più DataNode in modo tale da poter replicare i dati su un cluster di macchine, cosicchè non si abbia una perdita di informazioni nel caso in cui uno dei DataNode non sia più disponibile. In tal caso però un solo NameNode rappresenterebbe un singolo punto di guasto in quanto tutte le funzionalità offerte da HDFS all'utente dipendono dall'attività del NameNode, se quest'ultimo va in crash non sarà più possibile accedere ai dati archiviati nel sistema nonostante siano presenti più DataNode.
Per far fronte ai problemi appena discussi, si è reso necessario installare due NameNode in modo da realizzare un cluster HDFS con High Availability. In ogni

momento, soltanto uno dei NameNode sarà Active mentre l'altro sarà in modalità Standby: quando l'Active non è più raggiungibile a causa di un guasto, il NameNode Standby verrà eletto come nuovo Active. Per garantire che soltanto uno dei due NameNode rimane Active in un certo istante di tempo si usa il processo di Fencing, ciò evita uno scenario in cui un cluster risulta diviso in due cluster più piccoli , dove ognuno crede di essere l'unico cluster attivo. NameNode Active e Standby sono sincronizzati attraverso un gruppo di processi chiamati JournalNode. Questi sono responsabili, tra le altre cose, di eseguire il processo di fencing all'interno del cluster HDFS.
Una volta che il NameNode Active cade, viene avviata la procedura di Automatic Failover per trasferire il controllo in maniera automatica all'altro NameNode. Tale funzionalità viene svolta dal processo ZooKeeperFailoverController (ZKFC), il quale non è altro che un'istanza client di ZooKeeper che monitora e gestisce lo stato corrente dei NameNode.
Per implementare tutte le funzionalità descritte finora, si è reso necessario istanziare: NameNode su hadoop1 e hadoop2; Datanode su hadoop3, hadoop4,hadoop5; JournalNode su ogni macchina del cluster; ZKFC su hadoop1 e hadoop2.
Per quanto riguarda la configurazione dei file, innanzitutto si è reso necessario modificare il file core-site.xml, come mostrato in figura 3.12, per definire il nome del file system e la directory in cui saranno memorizzati i dati utilizzati dal processo JournalNode. Successivamente, sono state settate le seguenti proprietà per il

```
<configuration>
  <property>
    <name>fs.defaultFS</name>
    <value>hdfs://ha-cluster</value>
  </property>
  <property>
    <name>dfs.journalnode.edits.dir</name>
    <value>$HADOOP_HOME/hdfs/data/journal</value>
  </property>
</configuration>
```

Figura 3.12. Core-site.xml properties

file hdfs-site.xml. Come si può vedere dalla figura 3.13, abbiamo definito: lista dei server per la connessione a ZooKeeper; directory in cui vengono mantenute le informazioni relative ai namenode e ai datanode; fattore di replicazione dei dati all'interno del cluster hdfs; lista dei nameservice; lista dei server che eseguono il processo JournalNode; abilitazione dell'automatic failover.

```
<configuration>
  <property>
    <name>ha.zookeeper.quorum</name>
    <value>hadoop1:2181,hadoop2:2181,hadoop3:2181,
              hadoop4:2181,hadoop5:2181</value>
  </property>
  <property>
    <name>dfs.namenode.name.dir</name>
    <value>$HADOOP_HOME/hdfs/data/namenode</value>
  </property>
<property>
    <name>dfs.datanode.data.dir</name>
    <value>$HADOOP_HOME/hdfs/data/datanode</value>
  </property>

  <property>
    <name>dfs.replication</name>
    <value>3</value>
  </property>
  <property>
    <name>dfs.nameservices</name>
    <value>ha-cluster</value>
  </property>
  <property>
    <name>dfs.ha.namenodes.ha-cluster</name>
    <value>hadoop1,hadoop2</value>
  </property>
  <property>
    <name>dfs.namenode.shared.edits.dir</name>
    <value>qjournal://hadoop1:8485;hadoop2:8485;hadoop3:8485;
              hadoop4:8485;hadoop5:8485/ha-cluster</value>
  </property>
  <property>
    <name>dfs.ha.automatic-failover.enabled</name>
    <value>true</value>
  </property>
</configuration>
```

Figura 3.13. Hdfs-site.xml properties

3.6 Apache Cassandra

All'interno del sistema, Cassandra viene utilizzato in modo tale da distribuire grandi carichi di lavoro attraverso più nodi senza un singolo punto di guasto. In un cluster Cassandra tutti i nodi sono equivalenti e possono svolgere operazioni di scrittura e lettura sui dati. Ogni nodo scambia continuamente informazioni con gli altri attraverso il protocollo peer-to-peer di Gossip. Prima di eseguire un'operazione di scrittura, questa viene aggiunta su un commit log, in modo da poter ripristinare l'operazione in caso di fallimenti. Successivamente, i dati vengono scritti su una struttura in-memory, chiamata memtable, e replicati automaticamente su più nodi all'interno del cluster, per garantire affidabilità e tolleranza ai guasti. Quando la memtable è piena i dati vengono salvati su disco, in una struttura chiamata SSTable. Quando a un nodo viene fatta una richiesta da un client, il nodo agisce da proxy tra l'applicazione client e i nodi proprietari dei dati richiesti.
Nella terminologia di Cassandra, un datacenter rappresenta un gruppo di nodi su cui viene diviso il carico di lavoro e per il quale viene determinato il fattore di replicazione; il partitioner invece è una funzione hash che calcola un token, prendendo in input una primary key di una riga, per determinare quale nodo riceverà la prima replica dei dati e come distribuire le altre repliche all'interno del datacenter. Infatti, per ogni nodo vengono assegnati un numero di token che indicano in proporzione la quantità di dati memorizzata rispetto all'intero dataset. Lo snitch, infine, definisce il gruppo di nodi che appartengono a un datacenter e un rack; questa topologia viene utilizzata per piazzare correttamente le repliche dei dati sulle macchine.
Nel nostro caso, si è deciso di implemetare un datacenter Cassandra formato da tre nodi: hadoop3, hadoop4, hadoop5. Per far questo, si è reso necessario configurare il file cassandra.yaml. Nella figura 3.14, vengono mostrate alcune proprietà di configurazione per la macchina con hostname hadoop3.

```
num_tokens: 256

seed_provider:
    - class_name: org.apache.cassandra.locator.SimpleSeedProvider
      parameters:
          - seeds: "hadoop4,hadoop5"

listen_address: hadoop3
partitioner: org.apache.cassandra.dht.Murmur3Partitioner
endpoint_snitch: GossipingPropertyFileSnitch
```

Figura 3.14. Cassandra.yaml for hadoop3

In particolare, possiamo osservare: il numero di token assegnato a questo nodo; la lista dei provider seed, il cui unico scopo è quello di avviare il protocollo di Gossip sui nuovi nodi che si uniscono al cluster (sebbene sia raccomandato avere più nodi seed per fault tolerance, rendere ogni nodo seed ridurebbe le prestazioni); l'indirizzo su cui è esposto il servizio; il tipo di partitioner scelto, ovvero la funzione hash utilizzata per calcolare il token sulla chiave primaria della riga; il tipo snitch utilizzato, che in tal caso permette ad hadoop3 di scambiare informazioni con gli altri nodi definendo datacenter e rack di appartenenza.
Infine, nella figura 3.15, è possibile osservare lo stato del cluster Cassandra una volta che hadoop3, hadoop4 e hadoop5 vengono avviati. Poichè all'interno è stato creato un keyspace con fattore di replicazione 3 tramite il comando:
`CREATE KEYSPACE mykeyspace WITH replication={'class':'SimpleStrategy', 'replication_factor':3};`, ogni nodo sarà proprietario di tutti i dati presenti in mykeyspace, in quanto ogni replica è presente su una macchina del cluster.

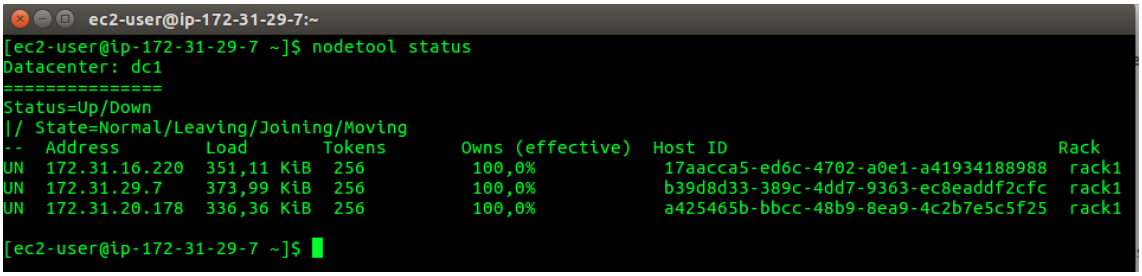


Figura 3.15. Cassandra cluster

3.7 H2O

Per quanto riguarda il framework utilizzato per il machine learning, H2O presenta una piattaforma scalabile, veloce, distribuita, in-memory che fornisce strumenti sia per creare modelli di machine learning che per effettuare l'analisi predittiva.
Il framework è scritto in Java e si compone di due sezioni: REST API Client e componenti H2O che eseguono in un processo JVM. La prima parte permette di scrivere script in R, Python o JavaScript (con cui è scritta l'user interface H2O Flow) e di visualizzare i risultati tramite Tableau. Ogni processo JVM è diviso in tre strati: linguaggio, include un motore di valutazione per le richieste client; algoritmi, utilizzati per implementare funzionalità offerte da H2O, come parsing dati, creazione e valutazione dei modelli, predizioni; infrastruttura, per la gestione delle risorse come CPU e memoria.
Un Data Frame è l'unità base utilizzata in H2O per memorizzare i dati secondo un

formato a colonne compresso, in modo tale da poter aggiungere, modificare, rimuovere elementi in maniera fluida. I frame vengono suddivisi tra i nodi del cluster attraverso uno store distribuito chiave-valore. Inoltre, H2O elabora i dati in Job, suddivisi tramite in-memory MapReduce Task e utilizza operazioni di Fork/Join per il multi-threading.
Un cluster H2O consiste di uno o più nodi, ognuno dei quali esegue un processo JVM. Per tali ragioni si è deciso di creare un cluster H2O formato da tre macchine: hadoop2, hadoop4, hadoop5.
A tal proposito, si è reso necessario aggiungere un file di configurazione, chiamato flatfile.txt, contenente la lista dei nodi che formano il cluster, specificare ip e porta di un nodo per ogni riga. In figura 3.16, viene mostrato il contenuto del file utilizzato all'interno del sistema.

```
hadoop2:54321
hadoop4:54321
hadoop5:54321
```

Figura 3.16. Flatfile.txt for hadoop2, hadoop4, hadoop5

Per le migliori performance, occorrerebbe dimensionare il cluster con una quantità di memoria circa quattro volte le dimensioni dei dati (in modo da eliminare swapping); stabilire per ogni nodo la stessa porzione di memoria, in quanto il framework esegue meglio con nodi simmetrici. Per assegnare la quantià di memoria desiderata al processo JVM che esegue H2O, durante l'avvio da terminale, per ogni nodo, si è aggiunta l'opzione `-Xmx2g`. A questo punto, una volta che il cluster è attivo si avrà a disposizione una memoria totale di 6 GB.
Nella figura 3.17, viene illustrato lo stato del cluster H2O tramite interfaccia utente fornita da H2O.

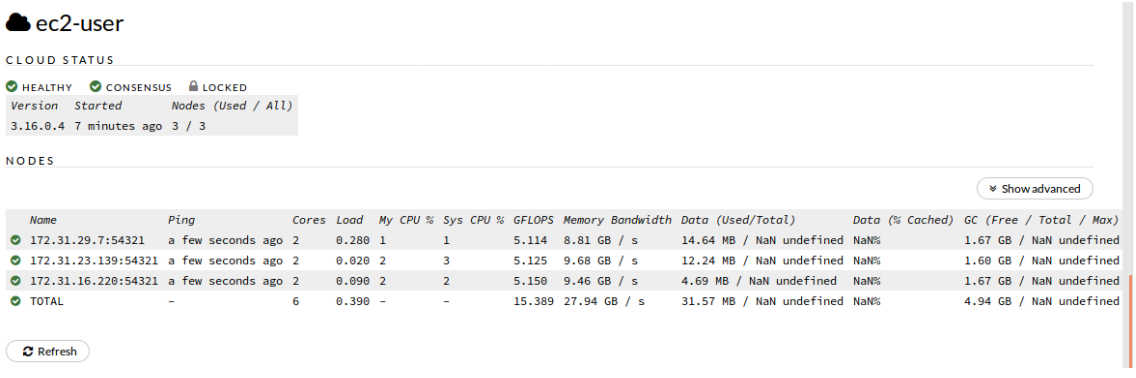


Figura 3.17. H2O Cluster status

Come anticipato nel capitolo precedente, H2O permette di esportare facilmente il modello di machine learning in formato POJO o MOJO, con l'obiettivo di rendere utilizzabile tale modello in un'applicazione Java ed effettuare predizioni sullo stream dati in tempo reale. Sebbene il tipo POJO non sia ancora supportato per modelli XGBoost, Stacked Ensemble, AutoML, il formato MOJO, invece, è disponibile per la maggior parte degli algoritmi, incluso Stacked Ensemble. Inoltre, POJO non offre supporto per file sorgenti di grandi dimensioni (superiori a 1 GB) ma funziona bene per predizioni di tipo binomiale e regressione su piccoli modelli, mentre MOJO risulta almeno 2-3 volte più veloce quando le dimensioni del modello sono molto più grandi (5000 trees / 25 depth).
Per realizzare un'applicazione che permetti di effettuare predizioni utilizzando il modello in formato POJO/MOJO, H2O mette a disposizione una libreria Java chiamata h2o-genmodel. Inoltre, per sottomettere tale modello allo stream dati, si è reso necessario utilizzare le API messe a disposizione da Kafka Streams.

3.7.1 Kafka Streams Machine Learning

In figura 3.18, viene mostrato un esempio di codice utilizzato per effettuare predizioni su uno stream dati.
Inizialmente, il modello di machine learning, chiamato nel nostro caso "myPojoModel", viene caricato in un oggetto Java di tipo EasyPredictModelWrapper. A questo punto, viene creato un oggetto di tipo KStream, per rappresentare uno stream di record proveniente da topicCleanData tramite delle coppie key-value rispettivamente di tipo String e MyCleanData. La funzione foreach ci permette di iterare su ogni coppia dello stream; per ogni iterazione viene creato un nuovo oggetto di tipo Row in cui dovranno essere inseriti i campi del record corrente che andranno in input al modello di machine learnig. In questo esempio, si effettua sui dati una predizione binomiale; una volta restituito il risultato, si aggiunge l'informazione ottenuta ad un nuovo stream di record che verrà pubblicato su topicPredictedData.

```
private static String predictedValue = "";

hex.genmodel.GenModel rawModel =
(hex.genmodel.GenModel) Class.forName("myPojoModel").newInstance();

EasyPredictModelWrapper myModel =
 new EasyPredictModelWrapper(rawModel);

final KStreamBuilder builder = new KStreamBuilder();
final KStream<String, MyCleanData> cleanData =
 builder.stream("topicCleanData");

cleanData.foreach((key, value) -> {
RowData row = new RowData();
row.put("FirstField", value.getFirstField());
row.put("SecondField", value.getSecondField());

BinomialModelPrediction prediction = myModel.predictBinomial(row);
predictedValue = prediction.label;
});

KStream<String, MyPredictedData> predictedData =
 cleanData.mapValues( ... );

predictedData.to("topicPredictedData");
```

Figura 3.18. Kafka Streams application for machine learning

Capitolo 4

Caso d'uso: analisi real-time mercato finanziario

In questo capitolo, ci occuperemo di testare il funzionamento dell'infrastruttura realizzata considerando come caso d'uso l'ingestion e l'analisi real-time di dati provenienti dal mercato finanziario. In particolar modo, lo stream dati che analizzeremo sarà generato da una sorgente che produrrà continuamente informazioni relative alle quote di un gran numero di azioni in tempo reale. Per poter acquisire tali informazioni, si è reso necessario utilizzare le API messe a disposizione da Investors Exchange (IEX) [25]. Esse permettono di recuperare i dati in tempo reale da un sistema con alte prestazioni (picchi di 3.5 milioni di messaggi al secondo), fortemente affidabile (99.981% web uptime), con un enorme base di dati (oltre 1 trilione di record) continuamente aggiornato (circa 300 GB aggiunti al giorno).
IEX API mette a disposizione l'endpoint stock su cui è possibile effettuare diverse richieste tra cui: chart, permette di recuperare lo storico delle quote di ogni azione (fino a 5 anni precenti al giorno corrente), news, fornisce le ultime notizie del mercato finanziario; quote, ritorna i prezzi in tempo reale di ogni azione insieme ad altre informazioni come prezzo di apertura, timestamp relativo all'ultimo aggiornamento, miglior offerta e miglior richiesta insieme al numero di azioni proposte, cambio rispetto alla chiusura del giorno precedente; batch, pemette di richiedere tramite un'unica richiesta più informazioni su più azioni contemporanemante.
Nel nostro caso, tra le oltre 8000 azioni continuamente aggiornate su IEX, abbiamo selezionato quelle relative solo ad alcuni settori di mercato come: technology hardware equipment, software services, media, retailing, telecommunication services, semiconductors. Solo per citare alcuni simboli, abbiamo: aapl (Apple), amzn (Amazon), fb (Facebook), googl (Google), ibm (IBM), intc (Intel), qcom (Qualcomm).
Una volta acquisito lo stream dati all'interno del nostro sistema tramite Apache

NiFi, questi verranno pubblicati su un topic Kafka, chiamato rawStock. Successivamente, verranno distribuiti su HDFS, Cassandra e Flink, il quale si occuperà della fase di processing. In particolar modo, l'applicazione di stream processing produrrà, su due topic Kafka separati, due nuovi stream di dati: il primo verrà emesso non appena arriva uno stream dati in input a Flink e conterrà solamente gli ultimi prezzi delle azioni acquisite; il secondo, invece, sarà pubblicato dopo un minuto e, oltre alle quote in real-time, sarà arricchito con il valore dell'azione aapl (Apple) relativo al minuto successivo. Il primo risultato verrà utilizzato dall'applicazione Kafka Stream per predire il valore di Apple al minuto successivo, mentre il secondo verrà aggiunto allo storico dati su HDFS che costituisce il set di dati utilizzato per addestrare il modello di machine learning su H2O. Infine, si effettuerà un confronto tra valore reale e valore predetto e verranno discussi i risultati ottenuti.

4.1 NiFi DataFlow

In questa sezione ci occuperemo di descrivere i passi principali effettuati nella realizzazione del DataFlow tramite l'interfaccia utente fornita da NiFi. I processori utilizzati a tal proposito sono stati: GetHTTP, ReplaceText, UpdateAttribute, PublishKafkaRecord.
Per quanto riguarda GetHTTP, questo ci ha permesso di recuperare le informazioni relative alle quote delle azioni in real-time esposti dalle API IEX. Come intervallo temporale tra due richieste consecutive si è scelto 15 secondi, in quanto i prezzi delle azioni vengono continuamente aggiornati e a noi interessa conoscere il loro valore in tempo reale. Inoltre, abbiamo settato come Primary Node la modalità di esecuzione di tale processore, poichè diversamente avremmo ricevuto gli stessi dati su tutti i nodi che formano il cluster NiFi. Per interrogare le API IEX, come URL si è scelto: `https://api.iextrading.com/1.0/stock/market/batch?symbols=<lista-simboli>&types=quote`, dove per `<lista-simboli>` si intende la stringa delle azioni (separata da virgola) di cui vogliamo ricevere le quote in real-time. Tale query restituirà i dati in formato JSON.
In figura 4.1, viene mostrato un esempio di record JSON restituito da IEX; per problemi di spazio, vengono visualizzate solamente pochi campi relativi solo alle quote di Apple mentre in realtà tale record conterrà tutte le informazioni disponibili per ogni azione specificata in `<lista-simboli>`. Inoltre, come si vede in alto a destra, come attributo Fiilename del FlowFile si è scelto di impostare il timestamp corrente, in modo tale che ogni volta che catturiamo un nuovo stream dati, questo sarà identificato in maniera univoca all'interno di NiFi.
Poichè il record JSON ricevuto non è ancora adatto per essere pubblicato su Kafka, si è reso necessario utilizzare il processore ReplaceText per modificare la struttura dei dati come da noi desiderato. In particolar modo, l'obiettivo è stato quello di

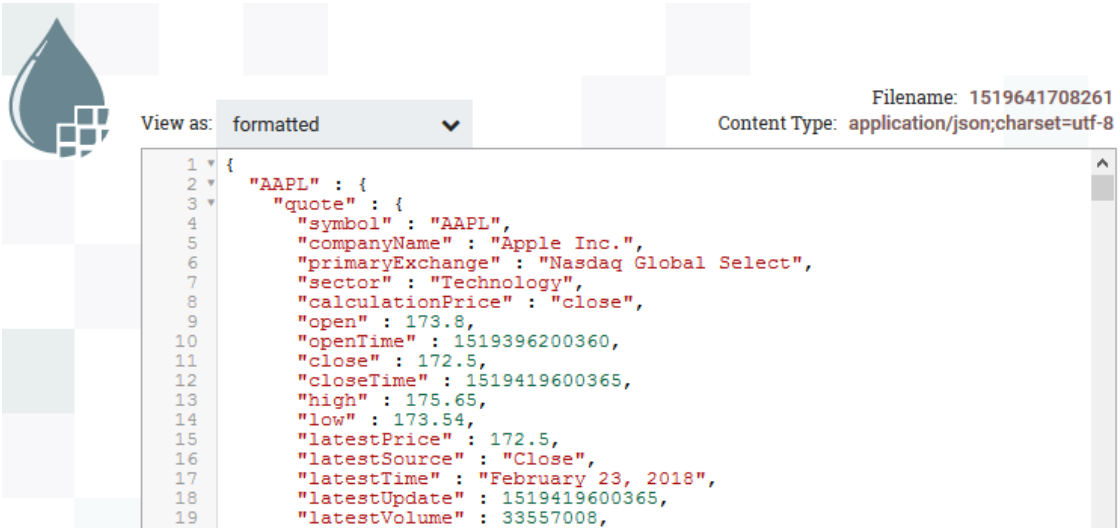


Figura 4.1. FlowFile Content: Apple quote

trasformare il contenuto del FlowFile in un array JSON formato da un insieme di record caratterizzato dagli stessi campi. Ogni record conterrà tutte le informazioni disponibili per una determinata azione, identificata dal campo `symbol`. Un altro processore ReplaceText, invece, è stato utilizzato per sostituire i valori di tipo null con delle stringhe, per facilitare la conversione dei record in formato Avro prima che siano pubblicati sul topic Kafka. Inoltre, si è scelto di aggiungere per ogni record JSON, un nuovo campo chiamato `"time"` per associare il timestamp relativo all'istante di tempo in cui è stato ricevuto quel record da NiFi.
Per quanto riguarda il processore UpdateAttribute, questo è stato utilizzato per aggiungere un attributo al nostro FlowFile. Infatti, come accennato nei capitoli precedenti, per scrivere i dati in formato Avro su un topic Kafka, occorre associare al record da pubblicare anche il nome dello schema Avro corrispondente, pubblicato precedentemente su Schema Registry. Di conseguenza, tramite UpdateAttribute abbiamo definito un nuovo attributo chiamato `schema.name` con valore `rawStock-value`, in quanto il nome dello schema Avro registrato deve contenere il nome del topic corrispondente seguito da key o value, a seconda di dove vengono trasportate le informazioni all'interno del record Kafka.
Infine, abbiamo configurato il processore PublishKafkaRecord come mostrato nel capitolo precedente in figura fig:pubkafka ad eccezione del campo del campo `Topic Name` settato con valore `rawStock` e del campo `Record Reader` settato con valore `JsonTreeReader`. Tale processore, sebbene riceva un array JSON permetterà di scrivere sul topic Kafka ogni singolo record contenuto all'interno, separando quindi le informazioni per ogni azione.

4.2 Record in Kafka

Per quanto riguarda la fase di message broker, si è deciso di utilizzare diversi topic Kafka, ognuno formato da tre partizioni e con fattore di replicazione pari a tre, al fine di garantire bilanciamento del carico e tolleranza ai guasti. Per ogni partizione, Kafka eleggerà come leader uno dei broker che formano il cluster, il quale si occuperà di rispondere alle richieste di lettura e scrittura per quella partizione, mentre i broker restanti assumeranno il ruolo di follower e dovranno rimanere sincronizzati con le operazioni svolte dal leader. In caso di guasti, infatti, quest'ultimo verrà subito rimpiazzato con uno dei follower disponibili, in modo da non pregiudicare il funzionamento del sistema.
Sostanzialmente, sono stati realizzati quattro topic Kafka: il primo, chiamato rawStock, memorizza i dati grezzi provenienti da NiFi, in tal caso avremo un record per ogni azione disponibile contenente le informazioni relative alle quote in tempo reale; il secondo, chiamato cleanStock, salva i risultati prodotti dalla fase di stream processing, quindi per ogni record Kafka avremo solo i prezzi di tutte le azioni per un determinato istante di tempo; il terzo, chiamato futureStock, viene usato per memorizzare, oltre alle informazioni contenute nel record precedente, anche il valore di Apple al minuto successivo; mentre l'ultimo, chiamato predictionStock, riceverà i dati ottenuti dopo la fase di machine learning, per cui avremo un record formato sia dai prezzi delle azioni per un determinato istante di tempo, sia il valore di Apple predetto per il minuto successivo.

Caso	**Contenuto Record**	**Kafka Topic**	**Cassandra Table**
1	Tutte le informazioni acquisite in un determinato istante di una sola azione	rawStock	rawStocksTable
2	Gli ultimi prezzi disponibili nello stesso istante per tutte le azioni	cleanStock	cleanStocksTable
3	Come nel caso 2, insieme all'ultimo prezzo di Apple del minuto successivo	futureStock	futureStocksTable
4	Come nel caso 2, insieme al prezzo di Apple predetto per il minuto successivo	predictionStock	predictionStocksTable

Tabella 4.1. Association between Kafka Topics and Cassandra Tables

Per ogni topic, sono stati configurati: un connettore HDFS Sink per trasferire i record da Kafka verso una directory del file system distribuito di Hadoop, chiamata `stocks-topics`; un connettore Cassandra Sink per esportare i dati all'interno di un keyspace, chiamato `stocks`. Come si può vedere dalla tabella 4.1, si è preferito creare per ogni topic Kafka esistente un' area di memoria corrispondente su Cassandra in modo da poter interrogare e visualizzare lo stato dei dati streaming ogni volta che viene effettuata un'operazione su di essi.
Nel caso del topic rawStock, per ogni azione avremo un record contenente tutte le informazioni disponibili in tempo reale su IEX, tra cui:

- symbol: identifica l'azione tramite una serie univoca di lettere e numeri;
- company: nome della compagnia;
- primaryExchange: nome della borsa valori primaria in cui è negoziata l'azione;
- sector: settore dell'azione;
- open: prezzo dell'azione durante l'apertura del mercato;
- close: ultimo prezzo dell'azione alla chiusura del mercato;
- high: prezzo più alto raggiunto durante la stessa giornata;
- low: prezzo più basso raggiunto durante la stessa giornata;
- latestPrice: ultimo prezzo dell'azione aggiornato in tempo reale;
- latestSource: definisce da dove proviene l'informazone relativa all'ultimo prezzo dell'azione disponibile;
- latestUpdate: data e ora relativi all'istante di tempo in cui è avvenuto l'ultimo aggiornamento del prezzo dell'azione;
- iexBidPrice: miglior prezzo di vendita attuale, dove per migliore si intende il più alto possibile;
- iexAskPrice: miglior prezzo di acquisto attuale, dove per migliore si intende il più basso possibile;

Nell'esempio di figura 4.2, viene mostrato il contenuto di due record memorizzati all'interno del topic rawStock. Ognuno di essi è formato dall'insieme di informazioni raccolte da IEX (tramite NiFi), in un determinato istante di tempo (rappresentato dal valore del campo `time`). Mentre il primo record (di cui i campi sono evidenziati in figura) è relativo alle azioni della società Apple, il secondo si riferisce alle azioni della compagnia Amazon.

Figura 4.2. Topic rawStock: Apple and Amazon records

4.3 Flink Streaming Job

In questo paragrafo verranno mostrati alcuni frammenti di codice Java utilizzati per realizzare l'applicazione che si occupa dell'analisi di dati streaming. A tal proposito, ci siamo serviti delle API DataStream, messe a disposizione dal framework Apache Flink, che oltre a fornire gli operatori necessari all'elaborazione, permettono anche di importare ed esportare i dati da e verso sistemi esterni, come Kafka nel nostro caso.

La nostra applicazione si occuperà principalmente di: acquisire i dati dal topic rawStock; estrarre da ogni record le informazioni relative al tempo in cui sono stati acquisiti i dati dalla sorgente esterna, simbolo dell'azione e ultimo prezzo disponibile; creare un nuovo stream di dati contenente solamente i valori di ogni azione per un determinato istante di tempo; pubblicare i risultati sul topic cleanStock; utilizzare una finestra per aggiungere ai risultati precedenti l'informazione riguardante il valore dell'azione Apple al minuto successivo; pubblicare i nuovi risultati sul topic futureStock.

In figura 4.3, viene mostrato il frammento di codice utilizzato per ottenere il nuovo stream di dati da pubblicare sul topic cleanStock. Dopo aver definito i topic Kafka da cui importare ed esportare i dati, attraverso il costruttore FlinkKafkaConsumer011 vengono acquisti i dati dal topic rawStock.

```
String inputTopic = "rawStock";
String outputTopic = "cleanStock";
String outputTopic2 = "futureStock";

DataStream<MyRawStock> rawStock =
   env.addSource(new FlinkKafkaConsumer011<>(inputTopic,
      new KafkaAvroDeserializationSchema(inputTopic,
         schemaRegistryUrl, MyRawStock.class),
      properties));

DataStream<Tuple3<String,String,String>> streamTuples =
  rawStock.map(new MapFunction<MyRawStock,
    Tuple3<String,String,String>>(){
      public Tuple3<String, String, String> map(MyRawStock element)
         throws Exception {
            Tuple3<String,String,String> result =
               new Tuple3<String,String,String>(
new String(element.getTime().toString()),
new String(element.getSymbol().toString()),
new String(element.getLatestPrice().toString()));
            return result;
  } });

DataStream<Tuple2<String,Map<String,String>>> streamResult =
   streamTuples.windowAll(TumblingProcessingTimeWindows
    .of(Time.seconds(5))).apply(new MyWindowFunction());

DataStream<GenericRecord> cleanStocks =
   streamResult.map(new MapToAvro());

cleanStocks.addSink(new FlinkKafkaProducer011<>(outputTopic,
   new KafkaAvroSerializationSchema(outputTopic,
      schemaRegistryUrl), properties));
```

Figura 4.3. Flink Streaming Application: cleanStocks DataStream

Poichè questi sono memorizzati su Kafka in formato Avro, abbiamo definito uno schema di deserializzazione per incapsulare le informazioni raccolte in un oggetto Java da noi prestabilito, appartenente alla classe MyRawStock. Per estrarre i valori di nostro interesse, si è scelto di utilizzare una funzione map e convertire così ogni oggetto appartenente allo stream dati in una tupla di tre elementi: il primo rappresenta il timestamp in cui sono state ricevute le informazioni dalla sorgente IEX; il secondo indica il simbolo relativo all'azione; il terzo, invece, costituisce l'ultimo prezzo disponibile per la relativa azione. Successivamente, abbiamo utilizzato una finestra di tipo tumbling di durata 5 secondi per attendere eventuali record che giungessero in ritardo da Kafka. Infatti, sebbene ogni 15 secondi, NiFi acquisisce i dati di più azioni dalla sorgente esterna e li converte in un array JSON, sul topic rawStock pubblicherà un record per ogni azione contenuta all'interno dell'array e di conseguenza potrà accadere che uno o più record arrivino in ritardo rispetto ad altri. Per tutti gli elementi appartenenti alla finestra, inoltre, verrà applicata una funzione definita nella nostra classe MyWindowFunction, allo scopo di produrre un nuovo stream di dati contenente per ogni istante di tempo in cui è stato fatto l'ingestion da NiFi, una mappa che associ ad ogni simbolo di un'azione il relativo prezzo aggiornato in tempo reale. A questo punto, il risultato verrà convertito in nuovo oggetto di tipo GenericRecord tramite la funzione di mapping definita nella classe MapToAvro in modo tale da essere pubblicato sul topic cleanStock, secondo uno schema Avro da noi prestabilito.
Nella figura 4.4, invece, viene mostrato il frammento di codice realizzato per pubblicare su Kafka un dato aggiuntivo oltre alle informazioni ottenute nella fase di elaborazione precedente. Nel nostro caso, si è deciso di catturare il valore delle azioni di Apple al minuto successivo rispetto l'istante temporale corrente a cui si riferiscono i prezzi di tutte le azioni acquisite dalla sorgente esterna. Poichè in questo caso si è reso necessario attendere 60 secondi prima di trovare il valore desiderato, abbiamo preferito non includere tale informazione nello stream dati pubblicato su cleanStock, in quanto esso rappresenta il topic di input da cui verranno presi i dati per effettuare le predizioni in tempo reale.
Tornando alla parte di codice, in questo caso, si è scelto di utilizzare una SlidingWindow per raccogliere tutti i dati in ingresso alla finestra nell'ultimo minuto ed emettere ogni 15 secondi il risultato ottenuto applicando una funzione di reduce. Quest'ultima permette di restituire un'unico elemento combinando tutti quelli arrivati all'operatore di window negli ultimi 60 secondi, iterando la stessa operazione su due elementi per volta. Nel nostro caso abbiamo definito una funzione che, dopo aver confrontato il timestamp relativo ai due oggetti in input, ritornasse l'oggetto con il timestamp meno recente solo dopo aver aggiunto (o aggiornato) all'interno della sua mappa (contenente per ogni azione l'ultimo prezzo corrispondente) il valore dell'azione Apple dell'oggetto con data e ora più recenti. Tale valore sarà associato

```
DataStream<Tuple2<String,Map<String,String>>> futureResult =
 streamResult.windowAll(SlidingProcessingTimeWindows
  .of(Time.seconds(60), Time.seconds(15)))
  .reduce(new ReduceFunction<Tuple2<String,Map<String,String>>>() {
    public Tuple2<String, Map<String, String>> reduce(
     Tuple2<String, Map<String, String>> value1,
     Tuple2<String, Map<String, String>> value2) throws Exception {
   Long timeFirstStockRecord = Long.parseLong(value1.f0);
   Long timeSecondStockRecord = Long.parseLong(value2.f0);
   if(timeFirstStockRecord < timeSecondStockRecord) {
     value1.f1.put(new String("aaplnext"),
               new String(value2.f1.get("aapl")));
     return value1;
   } else {
      value2.f1.put(new String("aaplnext"),
                new String(value1.f1.get("aapl")));
      return value2;
   }}});

DataStream<GenericRecord> futureStock = futureResult
   .map(new MapToAvro2());

futureStock.addSink(new FlinkKafkaProducer011<>(outputTopic2,
   new KafkaAvroSerializationSchema(outputTopic2,
      schemaRegistryUrl), properties));
```

Figura 4.4. Flink Streaming Application: futureStocks DataStream

con la chiave aaplnext, per indicare il prezzo di Apple successivo a quello corrente. Infine, come nel caso precedente, dopo aver convertito il risultato ottenuto in un oggetto di tipo GenericRecord tramite una nuova funzione definita nella classe MapToAvro2, è stato possibile pubblicare i nuovi dati sul topic futureStock, secondo lo schema Avro associato all'oggetto GenericRecord durante la fase di mapping.

4.4 Risultati di stream processing su Cassandra

In questo paragrafo, ci occuperemo di verificare la correttezza dei valori ottenuti durante la fase di stream processing, confrontando il contenuto di due tabelle su Cassandra appartenenti al keyspace stocks: rawstockstable e futurestockstable.

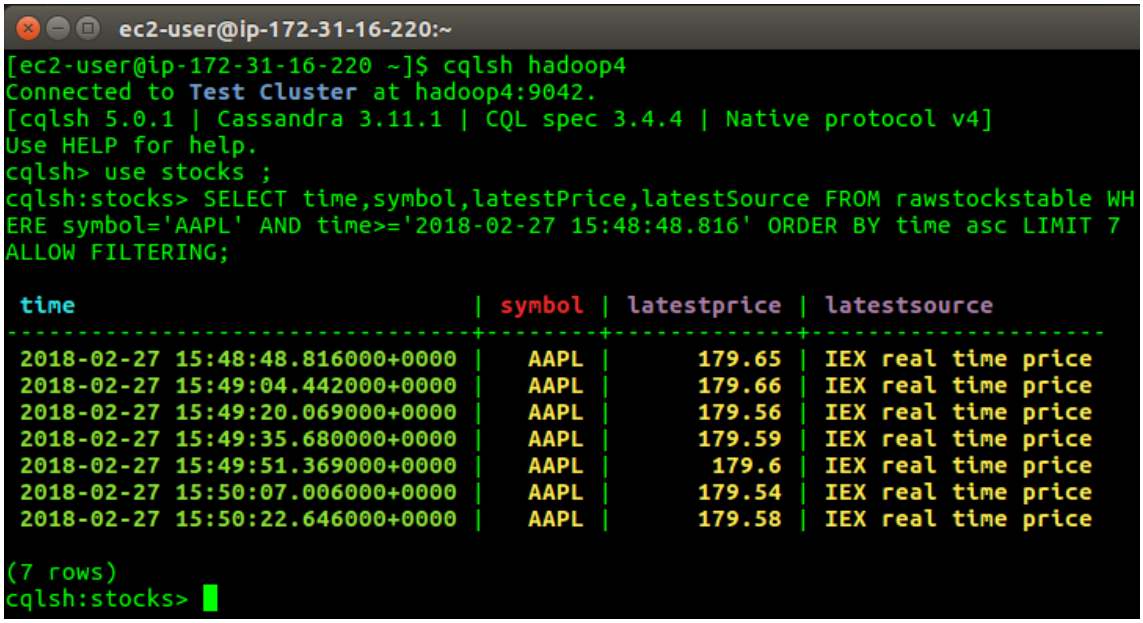


Figura 4.5. Cassandra: Raw Stocks Table

In figura 4.5, viene mostrato il risultato di una query scritta in linguaggio CQL effettuata sulla tabella rawstockstable. Come descritto nei paragrafi precedenti, tale tabella conterrà per ogni riga tutte le informazioni relative ad una precisa azione per un determinato istante di tempo in cui abbiamo acquisito i dati dalla sorgente esterna.
Per quanto riguarda la query, innanzitutto, come è possibile notare dalla clausola SELECT, si è deciso di selezionare per ogni riga solamente i campi time, symbol, latestprice e latestsource. La clausola WHERE, invece, ci ha permesso di filtrare tra le innumerevoli righe, scegliendo solamente quelle con valore di symbol pari ad aapl (ovvero solo righe relative alle azioni di Apple) e con valore di time maggiore o uguale alle 15:48:48.816 del 27 febbraio 2018. Tramite la clausola ORDER BY abbiamo ordinato i risultati in ordine di tempo dall'istante meno recente verso quello più recente; successivamente, LIMIT ci ha permesso di visualizzare solamente le prime sette righe appartenenti al risultato finale.
Possiamo quindi affermare che alle 15:48:48.816 del 27 febbraio 2018, l'ultimo prezzo relativo alle azioni di Apple, aggiornato da IEX in tempo reale, equivale a 179.65, mentre circa 15 secondi dopo, ovvero alle 15:49:04.442 dello stesso giorno, il valore di tali azioni risulta 179.66.

A questo punto, se volessimo verificare la correttezza dell'analisi di streaming dati per quanto riguarda i record pubblicati da Flink sul topic futureStock, ovvero quelli contenenti per ogni istante di tempo il valore delle azioni di Apple al minuto successivo, come si vede dalla stessa figura, dovremmo dimostrare che:

1. per l'istante di tempo 15:48:48.816 del 27 febbraio 2018, il prezzo corrente di Apple è 179.65 mentre quello al minuto successivo sarà 179.59

2. per l'istante di tempo 15:49:04.442 dello stesso giorno, il prezzo di Apple è 179.66 mentra al minuto successivo sarà 179.6

3. per l'istante di tempo 15:49:20.069 il prezzo di Apple è 179.56 mentre al minuto successivo è 179.54

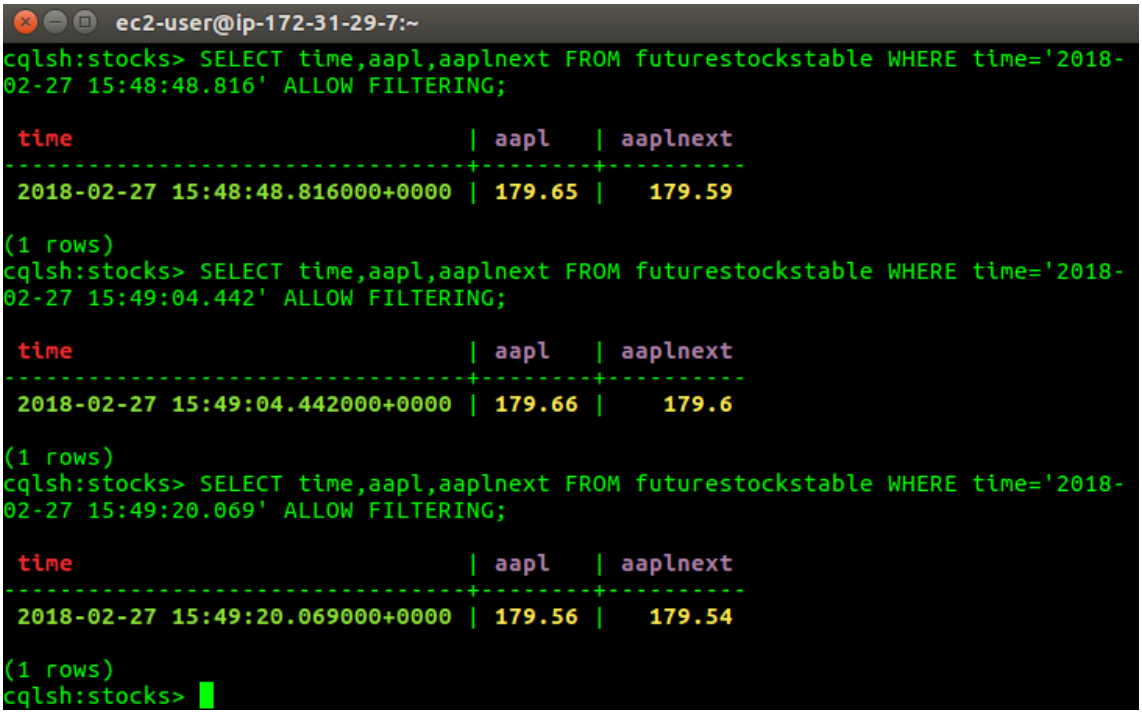


Figura 4.6. Cassandra: Future Stocks Table

Come viene mostrato in figura 4.6, effettuando le opportune query sulla tabella futurestockstable, possiamo verificare come i valori delle azioni Apple al minuto successivo, identificate dalla colonna aaplnext, siano coerenti con le ipotesi fatte in precedenza.

4.5 Realizzazione del modello con H2O

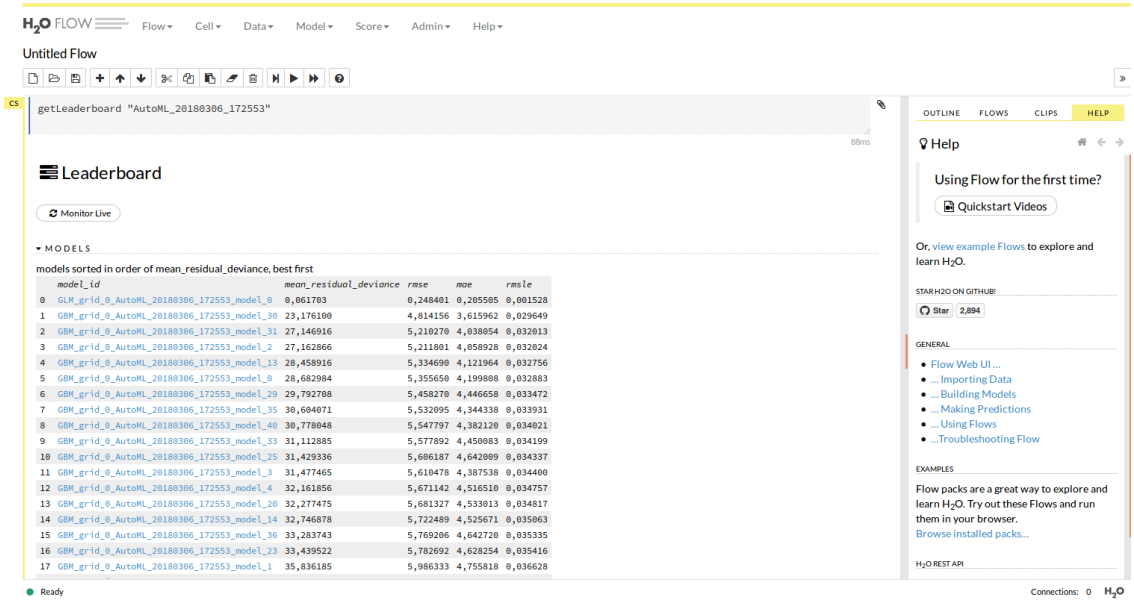
In questa sezione, ci occuperemo di descrivere il processo di creazione del modello di machine learning utilizzando il framework H2O. Il nostro obiettivo sarà quello di ottenere delle predizioni sui prezzi futuri delle azioni di Apple, a partire dai valori, acquisiti in tempo reale, di alcune azioni appartenenti allo stesso settore di mercato di Apple. Ciò può essere descritto come un problema di regressione, in quanto desideriamo trovare il valore di una variabile reale.
Per addestrare il modello, abbiamo scelto di organizzare il nostro dataset inserendo per ogni riga il valore, in un determinato minuto, delle azioni di 80 compagnie (come Cisco, HP, Microsoft), insieme al prezzo di Apple al minuto successivo. I dati raccolti sono relativi ad un intervallo temporale di 4 mesi, da giorno 1 novembre 2017 a giorno 2 marzo 2018 (esclusi i giorni di sabato, domenica e festività); per ogni giorno di mercato, i prezzi sono disponibili minuto per minuto, dalle 9:30 alle 15:59, secondo il fuso orario presente negli Stati Uniti.
In figura 4.7, vengono mostrate le prime righe del set di dati utilizzato. Come si può notare, nella prima colonna vengono elencati in ordine crescente gli istanti temporali, secondo il formato anno, mese, giorno, ora e minuto, per cui sono disponibili i valori delle azioni; nella seconda colonna, chiamata aaplnext, sono presenti i prezzi di Apple al minuto successivo rispetto quello corrente; nelle restanti colonne, invece, per ogni compagnia, identificata dal simbolo utilizzato come nome della colonna, abbiamo i prezzi per il minuto corrente.

	time	aaplnext	aapl	adbe	amat	amzn	apa	apc	avgo	baba	ba	bby	bidu	caa	cat	chtr	c	clr	cmcsa	cop
1	20171101-09:30	169.864	169.691	176.463	56.976	1105.299	41.724	50.325	264.095	187.591	257.908	56.035	248.305	49.468	136.385	335.465	73.963	41.483	36.304	51.6
2	20171101-09:31	169.501	169.864	176.463	57.14	1107.036	41.608	50.325	264.023	187.094	258.13	56.035	247.92	49.47	136.492	337.23	74.025	41.483	36.357	51.85
3	20171101-09:32	169.355	169.501	175.67	57.213	1107.775	41.63	50.377	264.984	187.492	258.76	56.035	247.291	49.47	136.863	338.535	74.031	41.483	36.363	51.89
4	20171101-09:33	169.311	169.355	175.428	57.211	1107.585	41.635	50.251	264.73	187.761	258.985	56.035	247.481	49.47	136.885	338.563	74.031	41.6	36.315	51.92
5	20171101-09:34	169.338	169.311	175.599	57.211	1106.981	41.665	50.045	265.02	187.859	258.985	56.035	248.118	49.31	136.94	338.34	74.093	41.687	36.3	51.91
6	20171101-09:35	169.604	169.338	175.732	57.282	1105.851	41.684	50.193	265.601	187.75	258.625	56.035	247.861	49.31	136.84	338.645	74.17	41.669	36.3	51.84
7	20171101-09:36	169.706	169.604	175.543	57.239	1104.964	41.682	50.319	264.975	187.956	258.618	56.035	247.964	49.34	136.833	337.53	74.236	41.65	36.231	51.91
8	20171101-09:37	169.736	169.706	175.326	57.202	1104.645	41.681	50.285	264.975	188.284	258.701	56.035	248.09	49.333	136.77	337.333	74.127	41.65	36.224	51.91
9	20171101-09:38	169.634	169.736	175.326	57.14	1104.211	41.656	50.09	264.22	188.478	258.701	55.96	248.084	49.333	136.625	337.333	74.285	41.72	36.22	52
10	20171101-09:39	169.587	169.634	175.326	57.177	1104.122	41.656	50.091	264.22	188.788	258.615	55.965	248.77	49.335	136.636	337.643	74.285	41.72	36.204	52.01
11	20171101-09:40	169.281	169.587	175.505	57.115	1103.482	41.782	50.111	263.904	188.577	258.575	55.965	248.808	49.34	136.711	338.085	74.24	41.82	36.22	52.01
12	20171101-09:41	169.265	169.281	175.4	57.028	1102.68	41.826	50.13	263.886	188.121	258.575	55.965	248.725	49.355	136.787	338.09	74.23	41.797	36.237	52.11
13	20171101-09:42	169.4	169.265	175.326	57.074	1102.866	41.99	50.15	264.009	188.628	258.575	55.965	249.161	49.355	137	338.253	74.26	41.797	36.248	52.15
14	20171101-09:43	169.153	169.4	175.363	57.145	1103.22	42.14	50.15	264.009	188.363	258.575	55.965	249.588	49.355	136.98	338.39	74.26	41.93	36.292	52.17
15	20171101-09:44	169.06	169.153	175.49	57.297	1103.799	42.14	50.207	264.25	188.51	259.045	55.965	249.838	49.355	136.98	338.352	74.315	41.93	36.305	52.14
16	20171101-09:45	169.055	169.06	175.485	57.297	1103.219	42.146	50.238	264.053	188.301	259.045	55.965	249.62	49.405	136.968	338.352	74.364	41.93	36.256	52.25
17	20171101-09:46	168.971	169.055	175.743	57.297	1103.878	42.285	50.309	263.98	188.379	259.173	55.965	249.504	49.39	136.988	338.803	74.38	41.93	36.267	52.24
18	20171101-09:47	168.88	168.971	176.11	57.15	1104.558	42.383	50.3	264.01	188.582	259.173	56.15	249.533	49.41	136.995	339.493	74.4	41.93	36.258	52.20
19	20171101-09:48	168.716	168.88	176.395	57.188	1104.999	42.357	50.253	264.08	188.591	259.173	56.27	249.622	49.41	136.942	339.543	74.4	41.847	36.301	52.12

Figura 4.7. Dataset sui prezzi delle azioni minuto per minuto

Successivamente, si è reso necessario suddividere il nostro insieme di dati in 3 parti: set di training, set di validazione, set di test. Il primo rappresenta un insieme di esempi di apprendimento utilizzati dall'algoritmo di predizione per adattare i parametri del modello in modo da ottenere l'output richiesto; il secondo e il terzo sono necessari per valutare il modello realizzato durante la fase di addestramento: il set di validazione è utilizzato per regolare i parametri e selezionare il modello finale,

in base al minor tasso di errore stimato su tale insieme di dati; il set di test, invece, serve per misurare l'accuratezza del modello, selezionato nella fase di validazione, su un nuovo insieme di dati. Per tali motivi, si è scelto: come set di training i dati relativi ai mesi di novembre, dicembre 2017 e gennaio 2018; come set di validazione quelli dal primo al 16 febbraio 2018; come set di test i dati dal 19 febbraio al 2 marzo 2018.
A questo punto, dopo aver importato i vari set di dati su H2O, abbiamo utilizzato l'interfaccia AutoML per automatizzare il processo di addestramento del modello e di regolazione dei parametri su una grande selezione di algoritmi. Dopo aver settato come colonna di risposta l'attributo `aaplnext` e aver specificato i vari dataset di addestramento, validazione e test, abbiamo impostato come tempo massimo di esecuzione 15 minuti. Una volta terminato l'algoritmo di AutoML, abbiamo ottenuto l'elenco dei modelli elaborati da H2O, come mostrato in figura 4.8, in ordine di: devianza (che in questo caso coincide con l'indice statistico mse: media dell'errore al quadrato), rmse (radice dell'errore quadratico medio), mae (errore assoluto medio), rmsle (radice dell'errore logaritmico quadratico medio).



	model_id	mean_residual_deviance	rmse	mae	rmsle
0	GLM_grid_0_AutoML_20180306_172553_model_0	0,061703	0,248401	0,205505	0,001528
1	GBM_grid_0_AutoML_20180306_172553_model_30	23,176100	4,814156	3,615962	0,029649
2	GBM_grid_0_AutoML_20180306_172553_model_31	27,146916	5,210270	4,038054	0,032013
3	GBM_grid_0_AutoML_20180306_172553_model_2	27,162866	5,211801	4,058928	0,032024
4	GBM_grid_0_AutoML_20180306_172553_model_13	28,458916	5,334690	4,121964	0,032756
5	GBM_grid_0_AutoML_20180306_172553_model_0	28,682984	5,355650	4,199808	0,032883
6	GBM_grid_0_AutoML_20180306_172553_model_29	29,792708	5,458270	4,446658	0,033472
7	GBM_grid_0_AutoML_20180306_172553_model_35	30,604071	5,532095	4,344338	0,033931
8	GBM_grid_0_AutoML_20180306_172553_model_40	30,778048	5,547797	4,382120	0,034021
9	GBM_grid_0_AutoML_20180306_172553_model_33	31,112885	5,577892	4,450083	0,034199
10	GBM_grid_0_AutoML_20180306_172553_model_25	31,429336	5,606187	4,642009	0,034337
11	GBM_grid_0_AutoML_20180306_172553_model_3	31,477465	5,610478	4,387538	0,034400
12	GBM_grid_0_AutoML_20180306_172553_model_4	32,161856	5,671142	4,516510	0,034757
13	GBM_grid_0_AutoML_20180306_172553_model_20	32,277475	5,681327	4,533013	0,034817
14	GBM_grid_0_AutoML_20180306_172553_model_14	32,746878	5,722489	4,525671	0,035063
15	GBM_grid_0_AutoML_20180306_172553_model_36	33,283743	5,769206	4,642720	0,035335
16	GBM_grid_0_AutoML_20180306_172553_model_23	33,439522	5,782692	4,628254	0,035416
17	GBM_grid_0_AutoML_20180306_172553_model_1	35,836185	5,986333	4,755818	0,036628

Figura 4.8. Risultati AutoML in H2O

Se consideriamo i primi due modelli dell'elenco, possiamo notare come il primo sia stato realizzato mediante l'algoritmo GLM (Modello Lineare Generalizzato) mentre il secondo per mezzo del GBM (Gradient Boosting Machine); per quanto rigurda gli indici statistici usati per valutare la qualità del modello, possiamo vedere come nel primo caso si ottengano dei valori molto inferiori rispetto al modello GBM, di conseguenza sembra che GLM fornisca delle predizioni molto più vicine ai valori

reali rispetto al secondo modello.
Nella tabella 4.2 vengono riportate le formule matematiche utilizzate per calcolare tali indici statistici utilizzati dall'AutoML per valutare la qualità del modello; si noti che il simbolo e_t rappresenta l'errore come differenza tra il valore reale r_t e quello predetto p_t.

Mean squared error	$\text{MSE} = \frac{1}{n}\sum_{t=1}^{n} e_t^2$
Root mean squared error	$\text{RMSE} = \sqrt{\frac{1}{n}\sum_{t=1}^{n} e_t^2}$
Mean absolute error	$\text{MAE} = \frac{1}{n}\sum_{t=1}^{n} \lvert e_t \rvert$
Root mean squared logarithmic error	$\text{RMSLE} = \sqrt{\frac{1}{n}\sum_{t=1}^{n} (\log(p_t + 1) - \log(r_t + 1))^2}$

Tabella 4.2. Parametri di valutazione utilizzati dall'AutoML H2O

Rispetto la media dell'errore assoluto (MAE), l'MSE penalizza maggiormente grandi errori. Ciò è preferibile nei casi in cui non sono particolarmente desiderabili grosse differenze tra valori reali e predetti. Per quanto riguarda l'RMSLE invece, esso può essere usato quando non vogliamo penalizzare grandi differenze tra i valori stimati e reali quando questi assumono numeri elevati. In generale, sebbene tali indici statistici ci forniscano un'idea su quanto i valori predetti si discostino dai valori reali, essi dipendono dall'intervallo di valori in cui variano i dati stimati e quelli reali e quindi non possono rappresentare indici assoluti di quanto sia affidabile la predizione effettuata.

4.6 Confronto tra predizioni e valori reali

In questo sezione ci occuperemo di analizzare i risultati di test ottenuti sottoponendo ai dati stream diversi modelli di machine learning realizzati mediante H2O. In particolar modo, si è scelto di effettuare delle predizioni in tempo reale non solo sui prezzi futuri delle azioni di Apple ma anche su quelli relativi alle azioni di Microsoft.
Per quanto riguarda le predizioni sui prezzi di Apple, abbiamo utilizzato i primi due modelli restituiti dall'algoritmo AutoML di H2O, come mostrato nel paragrafo

precedente, ovvero il Modello Lineare Generalizzato (GLM) e il Gradient Boosting Machine, mentre per ottenere delle predizioni sui prezzi di Microsoft, abbiamo, innanzitutto, modificato il set di dati di partenza aggiungengo un'ulteriore colonna, chiamata `msftnext`, contenente per ogni riga il valore delle azioni di Microsoft al minuto successivo. A questo punto, dopo aver diviso i dati in set di training, di validazione e di test, abbiamo utilizzato nuovamente l'algoritmo AutoML di H2O impostando gli stessi parametri, questa volta però ignorando i valori della colonna `aaplnext` e impostando come colonna di risposta `msftnext`. Anche in questo caso, abbiamo ottenuto dei risultati simili al caso di Apple poichè dalla lista dei modelli restitituiti da H2O e ordinati per devianza, rmse, mae, rlmse, abbiamo ottenuto come primi due, rispettivamente, GLM e GBM.
Per valutare l'affidabiità dei modelli di machine learning ottenuti, abbiamo deciso di calcolare la percentuale dell'errore di predizione, ottenuta come $\frac{(r_t - p_t)}{p_t} * 100$, dove r_t rappresenta il valore reale delle azioni al minuto successivo all'istante t, mentre p_t indica il valore predetto dal modello in questione all'istante t.
Inoltre, abbiamo simulato il comportamento di un operatore finanziario (trader) che compra e vende azioni in base ai valori predetti: ogni volta che il modello di machine learning stima un prezzo maggiore rispetto al valore delle azioni al minuto corrente, verranno acquistate delle azioni, viceversa, se il prezzo predetto per il prossimo minuto è minore allora lo stesso numero di azioni sarà venduto. All'avvio della nostra applicazione, ipotizzeremo a disposizione del trader un budget e un numero di azioni illimitati, cosicchè sia possibile sin da subito comprare o vendere; per quanto riguarda il numero di azioni cedute o acquistate ad ogni transizione si è stabilito di usare come cifra 100. L'obiettivo finale sarà quello di confrontare il guadagno totale, ottenuto come somma dei singoli guadagni per ogni transazione effettuata, per tre casi diversi:

- caso ideale: supponendo di conoscere per ogni istante i valori reali dei prezzi al minuto successivo, equivale al massimo guadagno possibile;
- caso predetto: supponendo che tutte le predizioni siano corrette, equivale al guadagno massimo promesso dal modello;
- caso reale: ad ogni transizione, si avrà un guadagno solamente quando l'acquisto o la vendita predetti coincidono con quelle nel caso ideale, altrimenti si avrà una perdita. Nel calcolo, verranno considerati i prezzi reali delle azioni al minuto successivo.

Per chiarire meglio, supponiamo che al minuto x, le azioni di Apple abbiamo un valore di 170 dollari e che il prezzo al minuto successivo $x+1$ sia 171 mentre quello predetto sia 172, allora in tal caso avremo: guadagno caso ideale $= (171-170)*100 = 100$ dollari, in quanto acquisteremo 100 azioni che al minuto x costano 170, mentre al minuto $x+1$ valgono 171; guadagno caso predetto $= (172-170)*100 = 200$; guadagno

caso reale = guadagno caso ideale, in quanto il prezzo stimato è maggiore di quello al minuto x per cui seguendo il modello compriamo 100 azioni che effettivamente saliranno di valore nel minuto successivo. Se invece avessimo predetto un prezzo di 168, allora: il guadagno nel caso ideale non sarebbe cambiato, in quanto esso dipende soltanto dai prezzi reali al minuto x e $x+1$; per il caso predetto, visto che il nostro modello annuncia un abbassamento dei prezzi, ci ritroveremo a vendere 100 azioni al minuto corrente prima che queste perdano valore ottenendo un guadagno di (170-168)*100=200 dollari; infine nel caso reale otterremo una perdita equivalente a (171-170)*100=100 dollari, in quanto seguendo le predizioni del modello venderemo 100 azioni che in realtà al minuto successivo sarebbero aumentate di valore.
Come approfondito nei paragrafi precedenti, ogni 15 secondi il nostro sistema si occuperà di catturare i prezzi in tempo reale delle varie azioni da IEX ed effettuare predizioni tramite Kafka Streams sottoponendo il modello H2O ai dati stream, salvando i risultati ottenuti sia su HDFS che su Cassandra. Di conseguenza, dopo il primo minuto di test è stato possible ottenere circa 3-4 predizioni, oltre 200 predizioni dopo la prima ora, mentre dopo quattro ore quasi 1000 prezzi stimati per le azioni Apple e Microsoft.

```
cqlsh:stocks> SELECT ingestiontime,model,symbol,nextprice,predictedprice,predictionerror,totmaxgain,totpredictedgain,totrealgain from predictionstockstable WHERE predict
ionid='214' AND model='Generalized Linear Model' ALLOW FILTERING;

 ingestiontime                | model                    | symbol | nextprice | predictedprice | predictionerror | totmaxgain | totpredictedgain | totrealgain
------------------------------+--------------------------+--------+-----------+----------------+-----------------+------------+------------------+-------------
 2018-03-09 18:31:00.473000+0000 | Generalized Linear Model |   aapl |    179.26 |         178,92 |           0,19% |     964,50 |          6686,34 |      -36,50
 2018-03-09 18:31:00.473000+0000 | Generalized Linear Model |   msft |    96.105 |          96,50 |          -0,41% |     495,00 |          9327,42 |      124,00

(2 rows)
cqlsh:stocks> SELECT ingestiontime,model,symbol,nextprice,predictedprice,predictionerror,totmaxgain,totpredictedgain,totrealgain from predictionstockstable WHERE predict
ionid='214' AND model='Gradient Boosting Machine' ALLOW FILTERING;

 ingestiontime                | model                    | symbol | nextprice | predictedprice | predictionerror | totmaxgain | totpredictedgain | totrealgain
------------------------------+--------------------------+--------+-----------+----------------+-----------------+------------+------------------+-------------
 2018-03-09 18:31:00.473000+0000 | Gradient Boosting Machine |   aapl |    179.26 |         179,21 |           0,03% |     964,50 |          1506,45 |       -3,50
 2018-03-09 18:31:00.473000+0000 | Gradient Boosting Machine |   msft |    96.105 |          94,67 |           1,49% |     495,00 |         26345,03 |      -124,00
```

Figura 4.9. Risultati ottenuti dopo la prima ora

```
cqlsh:stocks> SELECT ingestiontime,model,symbol,nextprice,predictedprice,predictionerror,totmaxgain,totpredictedgain,totrealgain from predictionstockstable WHERE predict
ionid='961' AND model='Generalized Linear Model' ALLOW FILTERING;

 ingestiontime                | model                    | symbol | nextprice | predictedprice | predictionerror | totmaxgain | totpredictedgain | totrealgain
------------------------------+--------------------------+--------+-----------+----------------+-----------------+------------+------------------+-------------
 2018-03-09 21:30:14.614000+0000 | Generalized Linear Model |   aapl |    179.85 |         179,47 |           0,21% |    3640,50 |         30730,98 |      -153,50
 2018-03-09 21:30:14.614000+0000 | Generalized Linear Model |   msft |     96.26 |          96,67 |          -0,42% |    1993,00 |         41658,60 |       205,00

(2 rows)
cqlsh:stocks> SELECT ingestiontime,model,symbol,nextprice,predictedprice,predictionerror,totmaxgain,totpredictedgain,totrealgain from predictionstockstable WHERE predict
ionid='961' AND model='Gradient Boosting Machine' ALLOW FILTERING;

 ingestiontime                | model                    | symbol | nextprice | predictedprice | predictionerror | totmaxgain | totpredictedgain | totrealgain
------------------------------+--------------------------+--------+-----------+----------------+-----------------+------------+------------------+-------------
 2018-03-09 21:30:14.614000+0000 | Gradient Boosting Machine |   aapl |    179.85 |         179,75 |           0,05% |    3640,50 |          5750,02 |       39,50
 2018-03-09 21:30:14.614000+0000 | Gradient Boosting Machine |   msft |     96.26 |          94,66 |           1,66% |    1993,00 |        139160,12 |      -205,00
```

Figura 4.10. Risultati ottenuti dopo quattro ore

In figura 4.9 e 4.10, vengono mostrati i risultati ottenuti rispettivamente dopo un'ora e quattro ore di test, salvati sulla tabella `predictionstockstable` di Cassandra, suddivisi per modello di machine learning utilizzato (colonna `model`) e tipo di azioni per cui sono state effettuate le predizioni (colonna `symbol`). Per ogni riga, è inoltre possibile visualizzare: `ingestiontime`, tempo di acquisizione dei dati dalla sorgente esterna (IEX); `nextprice`, valore reale delle azioni al minuto successivo;

`predictedprice`, prezzo stimato dal modello; `predictionerror`, percentuale dell'errore di predizione, calcolata come definito in precedenza; `totmaxgain`, guadagno totale caso ideale; `totpredictedgain`, guadagno totale caso predetto; `totrealgain`, guadagno totale caso reale. Ovviamente, tutti i guadagni totali sono calcolati come la somma dei rispettivi guadagni ottenuti durante le predizioni precedenti fino a quella corrente.
Dai test effettuati possiamo notare come in entrambi i casi i prezzi predetti risultano minori rispetto ai valori reali relativi ai prezzi delle azioni al minuto successivo, eccetto il caso riguardante il Modello Lineare Generalizzato per Microsoft; questo comportamento si è ripetuto per quasi tutte le predizioni effettuate. Di conseguenza, sebbene dai primi tre modelli vengano indovinate quasi tutte le vendite, il numero di acquisti per il caso predetto risulterà molto più basso rispetto a quello ottenuto per il caso ideale. Nel caso invece del Modello Lineare Generalizzato utilizzato per predire i prezzi delle azioni di Microsoft, quasi sempre viene predetto un prezzo maggiore rispetto a quello reale: in tal caso, nel caso predetto vengono indovinati quasi tutti gli acquisti, mentre il numero di vendite è molto minore rispetto quello ottenuto per il caso ideale.
Uno degli aspetti positivi, invece, è che molto spesso si riescono ad ottenere delle stime molto vicine ai prezzi reali con errori di predizione dell'1,8% per Microsoft e dello 0,3% per Apple nel caso peggiore, mentre dello 0,4% e dello 0,02% nel caso migliore rispettivamente per Microsoft ed Apple. Da qui, possiamo notare come, in generale, le predizioni per Apple risultano in tutti i casi più accurate rispetto quelle ottenute per Microsoft; ciò è verificabile confrontando anche i guadagni nel caso ideale con quelli nel caso predetto, infatti, questi si discostano maggiormente per le stime dei prezzi di Microsoft.
Possiamo concludere affermando che sia dopo un'ora che quattro ore di test, come dimostrato dai guadagni totali mostrati per il caso reale, si ottengono delle predizioni più accurate con il Modello Lineare Generalizzato per quanto riguarda le azioni di Microsoft. Viceversa, il Gradient Boosting Machine sembra essere il modello migliore per effettuare delle predizioni sulle azioni di Apple, sebbene secondo l'algoritmo AutoML di H2O il Modello Lineare Generalizzato fosse stato considerato più idoneo a tale scopo.

Capitolo 5

Conclusioni e sviluppi futuri

Il fenomeno dei Big Data è in continua espansione ed è destinato a cambiare il mondo dell'Information Technology in maniera sempre più rilevante. L'analisi di dati streaming consente alle aziende di prendere le strategie di bussiness migliori in brevissimo tempo, non appena si stanno verificando dei cambiamenti, ottimizzando così i processi di produzione e incrementando i profitti ottenuti.
In questo lavoro di tesi, durante una prima fase di progettazione sono state proposte una serie di soluzioni software open-source per lo sviluppo di un'architettura Big Data che permettesse l'acquisizione, l'elaborazione, l'archiviazione e la visualizzazione dei dati in tempo reale. Dopo aver confrontato e motivato con diverse argomentazioni la scelta dei framework effettuata per ogni funzionalità richiesta dal nostro sistema, si è passati alla fase di implementazione in cui abbiamo realizzato una piattaforma per l'analisi di dati streaming in modalità completamente distribuita, scalabile e affidabile, attraverso l'installazione e la configurazione dei diversi servizi su un cluster di cinque macchine residenti su Amazon EC2.
Durante la fase di sperimentazione, i risultati emersi hanno dimostrato come sia stato possibile elaborare i dati streaming, provenienti continuamente dalla sorgente esterna, mantenendo una bassa latenza; allo stesso modo, siamo riusciti ad ottenere delle stime sui prezzi futuri delle azioni di Apple e Microsoft molto vicine ai valori reali, con errori di predizione minori dell'1% nella maggior parte dei casi. Tali valori predetti però, non sembrano essere sufficientemente precisi, in quanto spesso ci siamo ritrovati in situazioni in cui il modello di machine learning non riuscisse a stimare correttamente i casi in cui dover effettuare un acquisto piuttosto che una vendita di azioni e viceversa.
Nonostante si sia scelto di affrontare l'analisi in tempo reale sull'andamento dei prezzi delle azioni quotate nel mercato finanziario, non si esclude, la possibilità di utilizzare la stessa architettura per numerosissimi casi d'uso differenti, come ad esempio l'ottimizzazione dei prezzi in base all'andamento attuale delle vendite per incrementare i profitti di una compagnia B2B, oppure la rilevazione delle frodi prima

che una transazione venga portata a termine.
Possiamo affermare che sebbene il progetto realizzato rappresenti un lavoro concluso, non è da escludere la possibilità che esso possa essere utilizzato come punto di partenza per eventuali sviluppi futuri. Uno degli aspetti che si potrebbe perfezionare riguarda la fase di apprendimento automatico per ottenere delle predizioni più accurate: grazie al modulo Deep Water, H2O permette l'integrazione con alcuni framework per il deep learning, come ad esempio TensorFlow, per ricavare nuove informazioni da una grande quantità di dati non strutturati come immagini, video, suoni e testo o dati strutturati da database transazionali come dati finanziari e serie storiche. Un ulteriore funzionalità da aggiungere al sistema potrebbe riguardare la visualizzazione e l'approfondimento dei risultati di analisi ottenuti attraverso l'utilizzo di un'interfaccia utente, come quella fornita dal software Tableau; in questo caso, sarà possibile integrare facilmente tale componenente con il resto dell'infrastruttura, semplicemente collegando Tableau con Cassandra attraverso il driver ODBC.

Bibliografia

[1] Douglas Laney, 3D Data Management: Controlling Data Volume, Velocity and Variety

[2] IBM, The Four V's of Big Data, ibm, 24 agosto 2012

[3] The 5 Vs of Big Data - Watson Health Perspectives, in Watson Health Perspectives, 17 settembre 2016

[4] Big data: The next frontier for innovation, competition, and productivity, https://www.mckinsey.com/~/media/McKinsey/Business%20Functions/McKinsey%20Digital/Our%20Insights/Big%20data%20The%20next%20frontier%20for%20innovation/MGI_big_data_exec_summary.ashx

[5] Andrea De Mauro, Marco Greco, Michele Grimaldi, (2016) "A formal definition of Big Data based on its essential features", Library Review, Vol. 65 Issue: 3, pp.122-135

[6] Curry, E., Ngonga, A., Domingue, J., Freitas, A., Strohbach, M., Becker, T., et al. (2014). D2.2.2. Final version of the technical white paper. Public deliverable of the EU-Project BIG (318062; ICT-2011.4.4).

[7] HADOOP ECOSYSTEM ,https://www.edureka.co/blog/hadoop-ecosystem

[8] Ovidiu-Cristian Marcu, Alexandru Costan, Gabriel Antoniu, Maria S. Perez-Hernandez,Radu Tudoran, Stefano Bortoli and Bogdan Nicolae Inria Rennes - Bretagne Atlantique, France IRISA / INSA Rennes, France Universidad Politecnica de Madrid, Spain Huawei Germany Research Center, Towards a Unified Storage and Ingestion Architecture for Stream Processing, 2017 IEEE International Conference on Big Data (BIGDATA)

[9] Lambda Architecture, https://mapr.com/developercentral/lambda-architecture/

[10] J. Kreps, Questioning the Lambda Architecture, O'Reilly. pp. 1–10, 2014

[11] Soumaya Ounacer, Mohamed Amine TALHAOUI, Soufiane Ardchir, Abderrahmane Daif and Mohamed Azouazi, Laboratoire Mathématiques Informatique et Traitement de Information MITI Hassan II University, Faculty Of Sciences Ben m'Sik Casablanca, Morocco, A New Architecture for Real Time Data Stream Processing, (IJACSA) International Journal of Advanced Computer Science and Applications, Vol. 8, No. 11, 2017

[12] Babak Yadranjiaghdam, Seyedfaraz Yasrobi, Nasseh Tabrizi, Department of computer science, East Carolina University, Greenville, NC, Developing a Real-time Data Analytics Framework For Twitter Streaming Data, 2017 IEEE 6th International Congress on Big Data

[13] Paula Ta-Shma, Adnan Akbar, Guy Gerson-Golan, Guy Hadash, Francois Carrez, and Klaus Moessner, An Ingestion and Analytics Architecture for IoT applied to Smart City Use Cases, IEEE INTERNET OF THINGS JOURNAL, VOL. X, NO. X, XX 2017

[14] "NSA Releases First in Series of Software Products to Open Source Community". www.nsa.gov

[15] Understanding When to use RabbitMQ or Apache Kafka, https://content.pivotal.io/blog/understanding-when-to-use-rabbitmq-or-apache-kafka

[16] Apache Kafka v/s RabbitMQ – Message Queue Comparison ,http://www.cloudhack.in/2016/02/29/apache-kafka-vs-rabbitmq/

[17] Apache Kafka vs RabbitMQ, https://www.slideshare.net/sbaltagi/apache-kafka-vs-rabbitmq-fit-for-purpose-decision-tree

[18] Nifi or Kafka. Which is the better edge option?, https://community.hortonworks.com/questions/118801/nifi-or-kafka-which-is-the-better-edge-option.html

[19] High-throughput, low-latency, and exactly-once stream processing with Apache Flink™, https://data-artisans.com/blog/high-throughput-low-latency-and-exactly-once-stream-processing-\with-apache-flink

[20] Apache Flink and Apache Kafka Streams: a comparison and guideline for users, https://www.confluent.io/blog/apache-flink-apache-kafka-streams-comparison-guideline-users/

[21] What is Cassandra?, http://cassandra.apache.org/

[22] NoSQL Performance Benchmarks, Cassandra vs. MongoDB vs. Couchbase vs. HBase ,https://www.datastax.com/nosql-databases/benchmarks-cassandra-vs-mongodb-vs-hbase

[23] A survey of open source tools for machine learning with big data in the Hadoop ecosystem, https://journalofbigdata.springeropen.com/articles/10.1186/s40537-015-0032-1

[24] Introducing Flink Streaming ,https://flink.apache.org/news/2015/02/09/streaming-example.html

[25] "Data provided for free by IEX.", https://iextrading.com/developer/