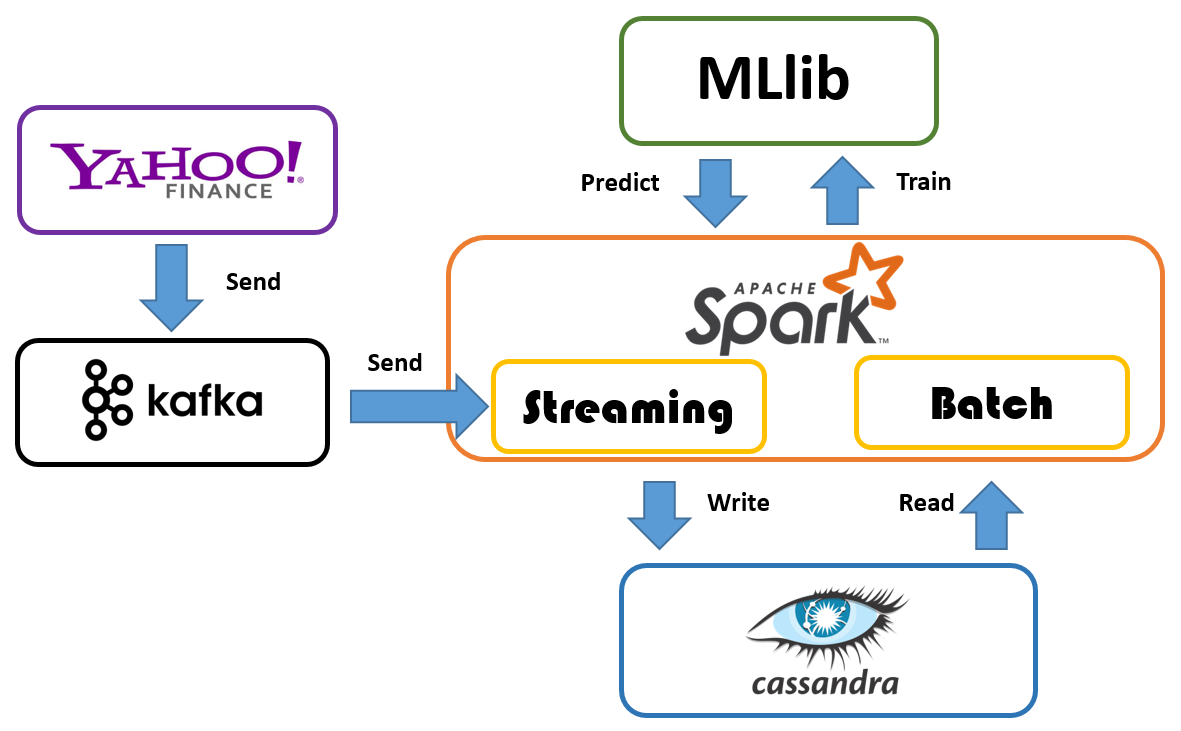
Analisi Streaming su Dati Finanziari con tecniche di Machine Learning

|  |  |
| --- | --- |
| **Andrea Bergonzo 448428** | **Luca Massuda 450710** |

# Overview

In questo progetto è stata progettata ed implementata un’intera architettura lambda per l’analisi di dati finanziari in streaming. Oltre al carattere analitico il progetto presenta una forte componente tecnologica, infatti fra gli strumenti utilizzati troviamo:

* **Spark Streaming**, per le analisi in tempo reale
* **Spark Core** (Batch), per le analisi offline
* **MLlib**, per l’utilizzo di tecniche di Machine Learning, sia in streaming che offline
* **Cassandra**, per la persistenza di aggregati creati dallo streaming
* **Kafka**, per la gestione dei messaggi tra la fonte di dati e Spark Streaming
* **Zookeeper**, per la gestione delle macchine Kafka
* **Yahoo Finance API**, libreria che consente l’accesso a dati finanziari in tempo reale



In generale, un architettura lambda è un’architettura per il processamento dei dati disegnata per gestire grandi quantità di dati usufruendo sia di metodologie batch che streaming. Questo approccio tenta di bilanciare la latenza, il throughput e la resistenza ai guasti utilizzando il processamento batch per garantire analisi accurate sui dati e contemporaneamente usando l’approccio streaming per offrire analisi real-time. E’ formata da tre livelli:

* **Speed Layer**: processa stream di dati in tempo reale. Sacrifica il throughput per minimizzare la latenza. E’ usato per riempire il gap del ritardo del batch layer.
* **Batch Layer**: processa tutti i dati storici. Ha maggior throughput ma anche maggior latenza rispetto allo Speed Layer.
* **Serving Layer**: combina gli output dei layer precedenti per mostrare delle visualizzazioni.

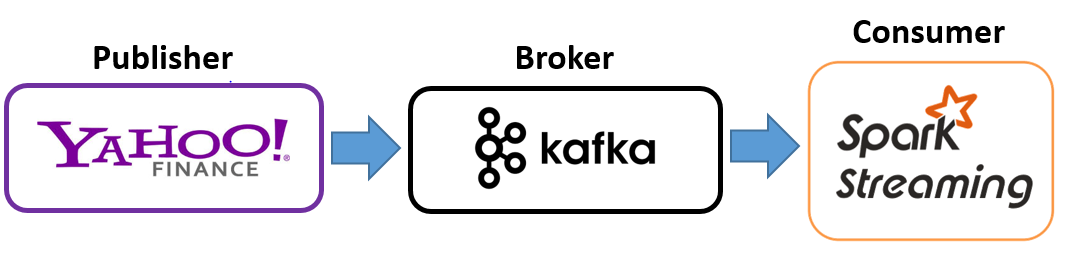
Il nostro progetto segue questa architettura ed è distribuito su più cluster che svolgono funzioni diverse. La prima parte prende i dati da Yahoo Finance e li mette in una coda Kafka, gestita su un cluster di sole macchine Kafka organizzate da Zookeeper. Nella seconda parte i dati vengono processati su un secondo cluster, lo speed layer, che utilizza Spark Streaming. Successivamente i dati aggregati delle analisi effettuate vengono salvati su Cassandra, creando uno storico utilizzato poi dal batch layer. Per alcune delle analisi effettuate sia speed layer che batch layer utilizzano MLlib, la libreria di Machine Learning di Spark, per effettuare analisi complesse.

# Yahoo Finance

Le migliori API gratuite su dati finanziari real-time sono quelle di “[Quotes API for Yahoo Finance](http://financequotes-api.com/)”, esse permettono di ottenere i dati finanziari di [Yahoo Finance](http://finance.yahoo.com/) su tutti gli stock exchange (borse valori) di tutto il mondo in tempo reale effettuando semplici richieste periodiche. Fonti di dati migliori che permettano di non dover effettuare richieste periodiche ma che avvisino al cambiamento dei valori dei prezzi sono tute vendute a caro prezzo.

# Kafka (+ Zookeeper)

Abbiamo utilizzato Apache Kafka, un Message Broker, ovvero un modulo che fa da intermediario traducendo un messaggio dal protocollo del mittente (Producer) al protocollo del destinatario (Consumer). L’utilizzo di un middleware orientato ai messaggi come Kafka ci ha permesso di disaccoppiare l’applicazione dalla fonte di dati, che può quindi essere cambiata con il minimo impatto sull’architettura. Allo stesso modo anche lo strumento di analisi può essere facilmente cambiato. Kafka può anche essere utilizzato per reimmettere da Spark Streaming i dati delle analisi effettuate così che possano essere utilizzati da altri servizi come ad esempio uno strumento di visualizzazione. Zookeeper invece è un servizio che serve a mantenere le informazioni di configurazione e la sincronizzazione tra i nodi Kafka.



# Spark

## Streaming

Abbiamo scelto di utilizzare Spark Streaming invece di Storm o Flink anche se la scelta non è stata per niente facile, infatti non esiste un vero e proprio strumento migliore dei concorrenti. Molti benchmark online mostrano risultati contrastanti, infatti alcuni mostrano come il più veloce sia Spark Streaming mentre altri affermano che sia Storm.

La nostra scelta è quindi ricaduta su Spark Streaming poichè è uno strumento molto maturo ed ha alle spalle non solo una grande community ma presenta anche molte librerie ben documentate di integrazione con altri software come Kafka e Cassandra. Un altro motivo della scelta è dato dal fatto che i nostri dati sono ottenuti da Yahoo Finance tramite delle richieste periodiche e quindi l’utilizzo di uno strumento come Storm che lavora sui singoli eventi invece che su micro-batch come Spark non avrebbe portato nessun vantaggio.

## Batch

Per la parte di analisi offline abbiamo optato per l’utilizzo di Spark per mantenere l’omogeneità nell’architettura avendo utilizzato Spark Streaming per le analisi real time.

# Analisi

## Dominio

I dati che analizziamo sono strutturati nel modo seguente:

* **Nome**: rappresenta il nome dello stock. (es: Apple Inc.)
* **Simbolo**: rappresenta l’abbreviazione del nome dello stock. (es: AAPL)
* **Currency**: rappresenta la valuta di riferimento dello stock. (es: USD)
* **Stock Exchange**: rappresenta la borsa a cui il simbolo fa riferimento. (es: NASDAQ)
* **Ask**: rappresenta il prezzo al quale viene richiesto uno stock.
* **Ask Size**: numero di stock richiesto al prezzo di ask.
* **Bid**: rappresenta il prezzo al quale viene offerto uno stock.
* **Bid Size**: numero di stock offerto al prezzo di bid.
* **Prezzo**: rappresenta il valore effettivo dell’ultimo scambio.
* **Volume**: rappresenta il numero di quote scambiate.
* **Trade Time**: rappresenta il momento in cui avviene la transazione.
* Altri valori secondari e o specifici del dominio che non prendiamo in considerazione...

In ogni finestra temporale possiamo definire rispetto ad un singolo stock 4 valori:

* **Open**, il prezzo all’inizio della finestra temporale
* **High**, il prezzo più alto raggiunto nella finestra temporale
* **Low**, il prezzo più basso raggiunto nella finestra temporale
* **Close**, il prezzo alla fine della finestra temporale

Questi valori sono spesso usati in Analisi Tecnica, ovvero lo studio dell’andamento dei prezzi dei mercati finanziari nel tempo con lo scopo di prevedere tendenze future, mediante principalmente metodi grafici e statistici.

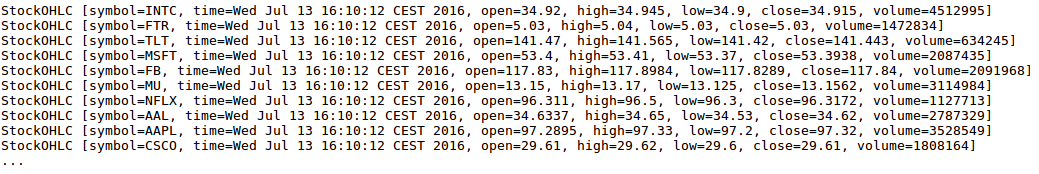
Una visualizzazione classica dei dati finanziari è quella “candlestick” dove il corpo centrale (Real-Body) rappresenta la differenza tra open e close, mentre le linee agli estremi (Shadows) rappresentano high e low. Il colore invece rappresenta la tendenza, a salire o a scendere, del prezzo.



## Streaming

La prima analisi streaming effettuata è stata il calcolare i valori **OHLC** (Open-High-Low-Close) di diversi stock rispetto ad una finestra temporale, con lo scopo di persisterli ed utilizzarli per successive analisi. Già solo questa analisi ha di per se un gran valore, infatti questi aggregati vengono venduti da privati per migliaia di euro. Ad esempio una collezione completa di 16.000 simboli della durata di un anno con intervallo temporale di un minuto viene venduta per circa $4.000 (es: [quantquote.com](https://quantquote.com/purchase.php)).

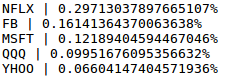
Per esempio questo è un aggregato OHLC da noi calcolato periodicamente su una finestra temporale di 1 minuto utilizzando dati della granularità di 1 secondo.



Questi valori sono poi utilizzati per effettuare altre analisi su determinate finestre temporali come:

* Quali azioni abbiano un trend positivo di maggior rilevanza.
* Quali azioni abbiano un trend negativo di maggior rilevanza.
* Quali azioni siano maggiormente volatili (maggior variazione di volume).

Per esempio in figura vengono mostrati i 5 stock che in tempo reale avevano una variazione percentuale positiva (trend up, crescita) maggiore nella finestra temporale relativa all’ultimo minuto.



## Queste sono state analisi di carattere prettamente descrittivo, le altre analisi effettuate sono più complesse e si basano su analisi predittive rese possibili grazie ad MLlib.

## MLlib

Abbiamo scelto di utilizzare MLlib, la libreria di Machine Learning nativa di Spark al fine di avere un ambiente di sviluppo omogeneo e coerente con le tecnologie precedenti, Spark Streaming e Spark Batch.

Abbiamo utilizzato diverse tecniche di Machine learning per due principali analisi:

* **Classificazione**, tentando di prevedere quali stock sarebbero saliti o scesi di prezzo nel tempo.
* **Clustering**, raggruppando gli stock con comportamento simile, prevedendo quindi quali stock continueranno ad avere lo stesso andamento tentando di trovare relazioni su stock apparentemente sconnessi.

### Classificazione

MLlib mette a disposizione per lo streaming un solo algoritmo di classificazione “[StreamingLogisticRegressionWithSGD](https://spark.apache.org/docs/latest/api/java/org/apache/spark/mllib/classification/StreamingLogisticRegressionWithSGD.html)” che utilizza lo [Stochastic gradient descent](https://en.wikipedia.org/wiki/Stochastic_gradient_descent) aggiornando il modello con ogni micro-batch dello stream, rendendolo quindi sempre più preciso con l’avanzare del tempo. Un difetto di questo algoritmo di classificazione streaming è dovuto al fatto che non è in grado di classificare in maniera multiclasse, utilizzando quindi solamente due label (label binaria), costringendo quindi adi allenare più modelli per avere una classificazione multiclasse.

Abbiamo deciso di classificare in base alle seguenti feature:

* Variazione percentuale tra High e Low.
* Variazione percentuale tra Open e High.
* Variazione percentuale tra Open e Low.

Il valore che tentiamo di predirre è la variazione percentuale tra Open e Close, ovvero l’andamento dello stock (crescente o decrescente).

Purtroppo abbiamo notato che utilizzando tali feature il modello tende ad avere una precisione media intorno al 50%, come un classificatore randomico. Ciò è dato principalmente dal fatto che prevedere l’andamento di un’azione è un task molto complesso e quindi richiede l’utilizzo di feature più complesse che necessitano una conoscenza del dominio molto più profonda.

### Clustering

MLlib mette a disposizione per lo streaming un solo algoritmo di clustering “[StreamingKMeans](http://spark.apache.org/docs/latest/mllib-clustering.html)” che si basa appunto su [K-Means](https://en.wikipedia.org/wiki/K-means_clustering). Un difetto di questo algoritmo è che il numero di cluster deve essere dato come input e quindi conosciuto a priori.

La nostra analisi di clustering si basa sul raggruppare le azioni che hanno andamenti simili, quindi abbiamo scelto le seguenti feature:

* Variazione percentuale tra High e Low.
* Variazione percentuale tra Open e High.
* Variazione percentuale tra Open e Low.
* Variazione percentuale tra Open e Close.

Da questa clusterizzazione otteniamo quindi degli oggetti che rappresentano a quale cluster appartenga un determinato stock in un determinato intervallo temporale. Questi dati vengono salvati su Cassandra, organizzati per simbolo, timestamp e cluster, permettendo poi analisi su periodi temporali più lunghi.

Questi dati sono successivamente usati sia in streaming che offline per effettuare un’analisi di quali stock siano nello stesso cluster più frequentemente e quindi quali stiano avendo lo stesso andamento. Un esempio è il seguente.

2016-07-13-17:30:58-Selection.png

Il numero iniziale rappresenta quante volte nella finestra temporale in analisi (in questo caso 2 minuti) gli stock sono stati clusterizzati nello stesso insieme. Quindi in questo caso possiamo notare dalla prima riga come Netflix (NFLX), Zinga (ZNGA) ed altri sono stati per 28 volte nello stesso cluster e quindi stiano avendo un andamento simile. Mentre dalla seconda e terza riga vediamo come Microsoft (MSFT), Yahoo (YHOO) ed altre due sono state per 21 volte insieme nello stesso cluster e per 17 volte anche Facebook (FB) è stato clusterizzato insieme a loro.

## Batch

Tutte le analisi in streaming sono state implementate anche per essere eseguite offline sui dati persistiti in Cassandra permettendo analisi con maggiore precisione (avendo a disposizione intervalli temporali più ampi) ma con minor “attualità” (basandosi sui dati non real time).

# Cassandra

Per il nostro caso d’uso stavamo cercando un DataBase che offrisse Availability e Partition-Tolerance, quindi che continuasse il suo funzionamento anche con la caduta di qualche nodo e di una eventuale partizione a causa della rete. Rinunciamo quindi alla Consistency poichè non abbiamo aggiornamenti sui dati ma solo molte scritture.

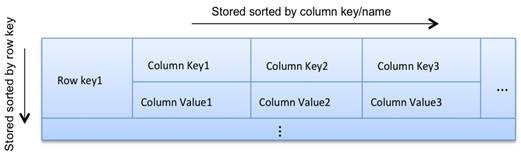
Il tipo principale di dati che salviamo sono serie temporali (time series data), in questo Cassandra è un ottimo strumento. Infatti offre un concetto di wide-row dove:

* E’ permesso un ordinamento e una scansione tra intervalli efficiente.
* I dati vengono richiesti insieme, quindi è possibile raggruppare i dati in una wide row che può essere letta efficientemente.
* E’ possibile denormalizzare una relazione one-to-many come una wide-row senza duplicazione dei dati.

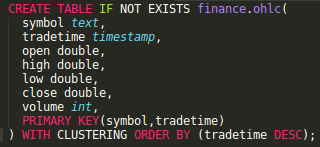
Abbiamo valutato anche l’utilizzo di un Document DB, come MongoDB o CouchDB, che però non ci permettevano di recuperare dati basandoci su intervalli temporali senza compromessi in termini di prestazioni.

Abbiamo strutturato lo schema come segue:

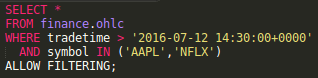
* Una Row Key per ogni simbolo, utilizzati insieme ai vari timestamp per creare una wide row.
* Una Column Key per ogni timestamp relativo al simbolo della Row Key che crea quindi una colonna diversa per quel simbolo.
* Un Column Value in ogni colonna composto dai valori (open, high, low, close, volume) per ogni timestamp.



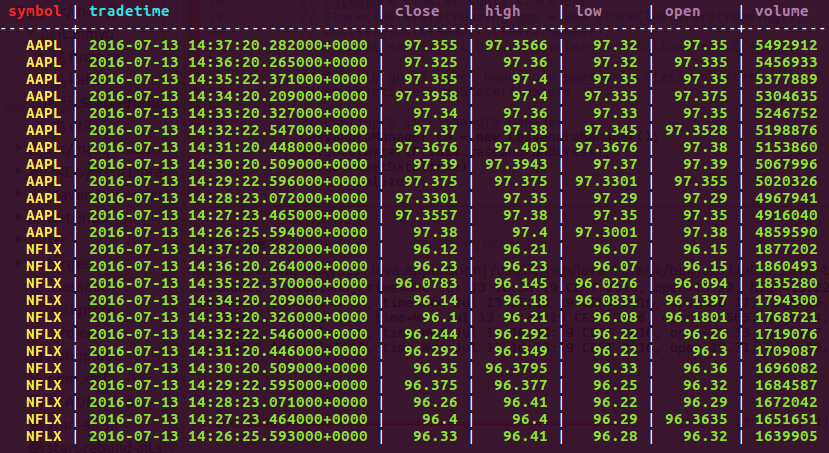
Lo schema utilizzato per gli stock OHLC è il seguente. Grazie ad esso tutti i dati sono ordinati di default per simbolo e le colonne relative ad uno stesso simbolo sono sempre memorizzate sullo stesso nodo.



Una query possibile su questi aggregati è la seguente. Una selezione relativa agli stock di Apple (AAPL) e Netflix (NFLX) in un determinato intervallo temporale da un certo momento in poi.

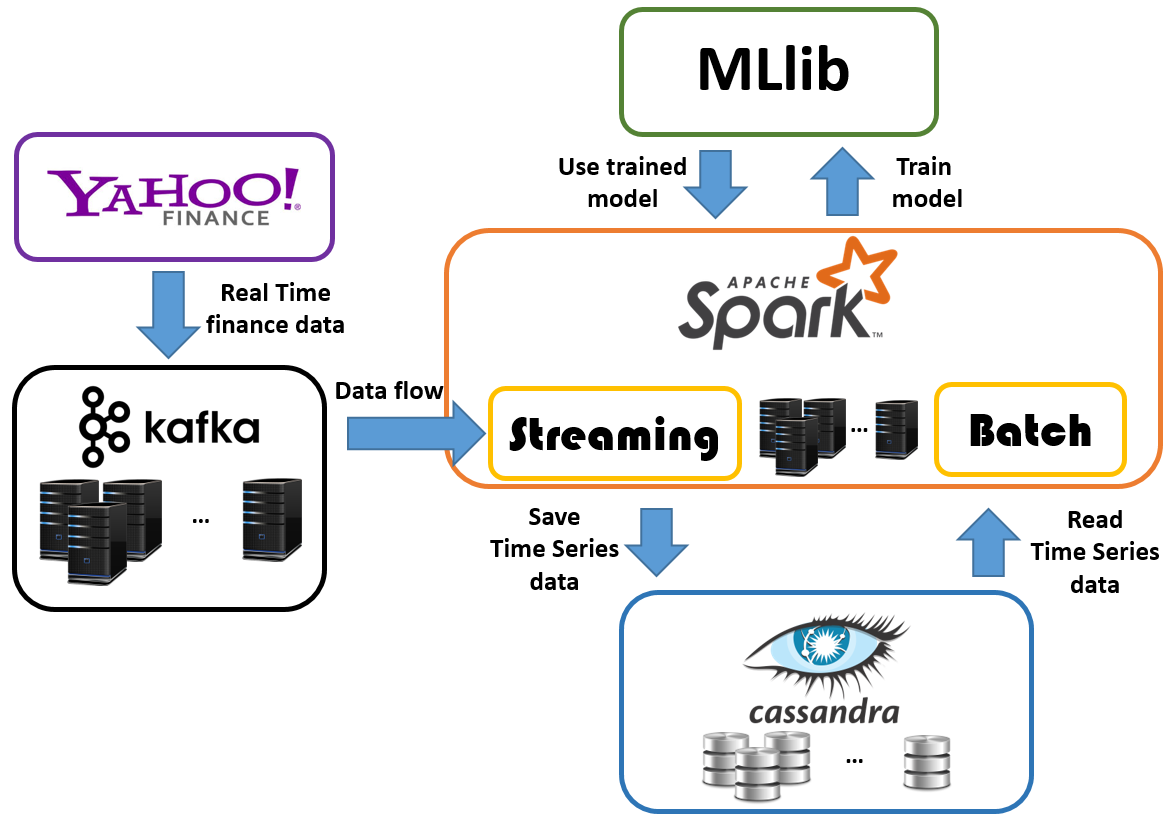


Il risultato è mostrato ordinato prima per simbolo (row key, in rosso) e per tempo (wide row, in blu). In viola invece troviamo i column value.



# AWS

L’intera architettura è stata deployata su AWS grazie al grant studenti di Roma Tre. Per eseguire le nostre analisi su un totale di circa 5000 stock diversi al secondo è stata utilizzata la seguente configurazione: una macchina Kafka (EC2 m4.large con 2 cpu e 8 GB di RAM) per la coda di messaggi, un cluster Spark di 3 nodi (EMR 3x m4.large con 2 cpu e 8 GB di RAM, 1 master e 2 slave) per le analisi e una macchina Cassandra (EC2 m4.large con 2 cpu e 8 GB di RAM + EBS General Purpose gp2) per la persistenza. Per aumentare il numero di stock o eseguire analisi più pesanti basta aumentare il numero di macchine dei vari cluster.



## Script

Per automatizzare il procedimento di inizializzazione dei cluster sono stati scritti degli script riportati in allegato. Per ogni cluster abbiamo diversi script:

* Kafka
  + **aws-kafka-installation.sh**, installa sulla macchina EC2 Java 8 e Kafka, configurando le relative variabili d’ambiente
  + **aws-kafka-start.sh**, configura in Kafka il proprio indirizzo IP e fa partire il Broker
  + **aws-kafka-stop.sh**, termina il servizio Kafka
* Cassandra
  + **aws-cassandra-installation.sh**, installa sulla macchina EC2 Java 8 e Cassandra, configurando le relative variabili d’ambiente
  + **aws-cassandra-start.sh**, fa partire il servizio Cassandra
  + **aws-cassandra-stop.sh**, termina il servizio Cassandra
* Spark
  + **aws-spark-installation.sh**, configura il cluster Spark per utilizzare Java 8 invece del default Java 7
  + **aws-spark-start.sh**, fa partire le analisi in streaming.   
    N.B. bisogna impostare gli IP dello script con gli IP pubblici relativi alle macchine EC2 di Kafka e Cassandra e controllare che le porte dei servizi siano visibili dalle altre macchine

# Panoramica del Progetto

Vediamo ora quali sono i punti più importanti dell’implementazione del progetto.

Il progetto è diviso in due sottoprogetti:

1. **finance-kafka**, programma java che effettua richieste alle API di Yahoo Finance e immette nella coda Kafka i messaggi relativi agli stock.
2. **finance-spark**, la parte corposa del progetto, dove Spark Streaming prende i dati dalla coda Kafka, li processa attraverso diverse analisi e li persiste. Spark Batch invece utilizza gli aggregati persistiti in Cassandra per effettuare analisi storiche.

La parte 1) ha seguente struttura:

* Package **kafka**:
  + Classe **StockProducer**: effettua le richieste alle API Yahoo Finance tramite il package finance e immette nella coda Kafka i dati ricevuti.
* Package **finance**:
  + Classe **YahooFinance**: effettua le chiamate alle API e restituisce un oggetto Stock.
  + Classe **Stock** : oggetto che contiene i dati relativi ad un determinato stock in un determinato istante (prezzo, volume, timestamp, etc..).
* Package **utils** :
  + Classe **CompaniesLister**: processa un file contenente una lista dei nomi degli stock da richiedere a Yahoo Finance.
* Cartella **resources**: contiene tutti i file csv usati da CompaniesList, come la lista degli stock delle borse di New York, Milano, Londra e Francoforte.

La parte 2) ha seguente struttura:

* Package **kafka**:
  + Classe **KafkaConnector**: gestisce l’interfacciamento di Spark con Kafka sia nella fase di ricezione di messaggi sia nella fase di invio di messaggi.
* Package **spark**:
  + Package **analysis.streaming**:
    - Classe **SparkStreamingAnalytics**: imposta le configurazioni usate per le varie analisi streaming.
    - Classe **AnalysisStreamingRunner**: contiene tutte le analisi streaming come il calcolo degli stock OHLC.
  + Package **analysis.batch**:
    - Classe **SparkBatchAnalytics**: utilizzata per impostare le configurazioni usate per le varie analisi.
    - Classe **AnalysisBatchRunner**: contiene gli effettivi metodi di analisi.
  + Package **ml.classification**:
    - Classe **StockClassifierStreaming**: classificatore di dati relativi agli stock in streaming utilizzando l’algoritmo SGD.
    - Classe **StockClassifier**: classificatore per la parte batch.
  + Package **ml.clustering**:
    - Classe **StockClustererStreaming**: effettua il clustering su dati in streaming con l’algoritmo k-means.
* Package **stock**:
  + Diverse classi che rappresentano i diversi aggregati analizzabili sugli stock.

Importanti sono inoltre i file README.md dei 2 sottoprogetti che contengono tutte le informazioni utili per la replicazione dell’infrastruttura in locale.

# Difficoltà Incontrate

Le difficoltà incontrate sono state molteplici. Le prime sono state sulla scelta delle tecnologie, infatti spesso non esistono strumenti migliori di altri su tutti i fronti e la scelta ricade quindi sul capire quale sia il compromesso migliore.

Altre difficoltà sono state quelle sistemistiche per il deploy su AWS. E’ stato infatti molto difficile configurare e far comunicare tra di loro tutte le tecnologie, soprattutto su macchine diverse.

Un’altra difficoltà ha riguardato la parte Streaming di MLlib che si è rivelata ancora molto acerba, infatti:

* Non esiste nativamente la classificazione Multiclasse, quindi per predirre su più di due classi bisogna allenare N modelli diversi dove N è il numero di classi (approccio [One vs Rest](https://en.wikipedia.org/wiki/Multiclass_classification#One-vs.-rest)).
* Non vi sono metodi semplici per il calcolo di precision e recall che invece esistono per i metodi di Machine Learning batch.

# Conclusioni

Il risultato finale è stata l’implementazione di un’intera architettura lambda tramite l’utilizzo di numerose tecnologie importanti nel mondo Big Data come Kafka, Spark Streaming e Cassandra. Il lavoro svolto si è focalizzato principalmente sullo speed layer, e quindi Spark Streaming, utilizzando algoritmi di Machine Learning per classificazione e clustering in streaming. I risultati ottenuti sono positivi sia dal punto di vista architetturale che analitico. L’architettura basata su cluster è in grado di gestire moli ingenti di dati ogni secondo. La sola architettura in grado di persistere analisi semplici come OHLC potrebbe creare un profitto. Riguardo le analisi più complesse, il clustering ha raggiunto il risultato prefissato, mentre la classificazione, sia per la difficile predicibilità del dominio finanziario, sia per la mancanza di conoscenze approfondite del settore, avrebbe potuto portare ad un risultato migliore.

Siamo molto soddisfatti dei risultati ottenuti e di aver avuto la possibilità di sperimentare molte tecnologie emergenti.

# Possibili Sviluppi Futuri

Questo progetto potrebbe infine essere esteso e migliorato in molti modi, tra cui:

* Utilizzare una fonte di dati alternativa (a pagamento) a Yahoo Finance che fornisca dati in modalità push invece che pull, ovvero che non richieda di eseguire richieste periodiche, bensì invii dati ogni qual volta siano aggiornati.
* Migliorare la Machine Learning batch allenando con dataset più grandi (a pagamento).
* Sviluppare analisi più complesse, consultando anche esperti del settore finanziario.