Big Data Velocity

Ottenimento e analisi di dati in streaming

Indice

1 Introduzione

Questo progetto nasce con l'obiettivo di ottenere dati in streaming e di sottoporli ad un'analisi in tempo reale utilizzando sistemi scalabili in grado di adattarsi facilmente ad eventuali aumenti della frequenza di ricezione.
Il dominio scelto è quello finanziario, nello specifico quello delle criptovalute in quanto le piattaforme di exchange forniscono API di semplice accesso che consentono di ricevere informazioni di mercato come prezzi e volumi di un certo asset con tempi di ritardo ridotti.
In aggiunta le monete virtuali negli ultimi anni hanno guadagnato una notevole attenzione tanto che ad esempio su servizi come Twitter ogni minuto vengono pubblicate varie decine di tweets relativi all'argomento che è possibile ricevere in streaming grazie alle API fornite, così da poter utilizzare anch'essi per scopi analitici.

2 Dati

2.1 Time-series

I dati acquisiti dall'applicazione sono di tipo **time-series** o serie temporali; si tratta di dati strettamente dipendenti da una coordinata temporale (ad esempio il momento di acquisizione) e che pertanto presentano alcune peculiarità come il fatto che spesso sono i record recenti ad essere più "interessanti" e di conseguenza raramente si ha la necessità di aggiornare quelli più datati.
Ne sono alcuni esempi i dati acquisiti da sensori ad esempio meteorologici, quelli relativi all'utilizzo delle risorse hardware di un personal computer o le quotazioni di titoli finanziari.

2.2 Dati finanziari

I dati finanziari sono ricevuti dalla piattaforma di exchange di criptovalute Binance [1] e comprendono i migliori prezzi di offerta e richiesta di un determinato simbolo e le rispettive quantità; segue un esempio:

```
{
    "u": 5828881697,         // order book updateId
    "s": "BTCUSDT",          // symbol
    "b": "10262.83000000",   // best bid price
    "B": "1.88008400",       // best bid qty
    "a": "10262.94000000",   // best ask price
    "A": "6.48000500"        // best ask qty
}
```

I dati sono forniti in real-time senza nessun parametro temporale; di conseguenza viene aggiunto al momento della ricezione un campo "timestamp" con l'istante corrente come valore. Durante l'analisi verrà principalmente utilizzato il valore "askprice". Il simbolo scelto al quale i prezzi ricevuti si riferiscomo è BTCUSDT.

2.3 Tweets

Anche i tweets possono essere considerati serie temporali data la notevole importanza dell'istante di pubblicazione, sopratutto se analizzati per scopi

legati alla finanza.
Sono ricevuti attraverso le API per sviluppatori fornite da Twitter [2] e si presentano come segue:

```
{
    "created_at": "Wed Sep 09 15:19:42 +0000 2020",
    "id": 1303714706687963136,
    "id_str": "1303714706687963136",
    "text": "... $50 ETH GIVEAWAY ...",
    "source": "...",
    ...
    "user": {
        ...
    }
    "geo": null,
    "coordinates": null,
    "retweeted_status": {
        ...
    }
    "is_quote_status": false,
    "quote_count": 0,
    "reply_count": 0,
    "retweet_count": 0,
    "favorite_count": 0,
    "lang": "en",
    "timestamp_ms": "1599664782579"
}
```

Si nota la presenza di ben 2 campi temporali (”created_at” e ”timestamp_ms”) che rappresentano in realtà il medesimo istante temporale ma con precisione differente. Tuttavia è possibile notare un certo ritardo nella ricezione dei tweets di circa 5-10 secondi, perciò è risultata necessaria l’aggiunta di un campo chiamato ”receivedat” così da tenere traccia del momento di ricezione.

L’analisi verrà successivamente effettuata principalmente sul campo ”text” contente il corpo del tweet.

I tweet sono filtrati ”a monte” tramite le API di Twitter quindi ogni tweet ricevuto risulta già pertinente con il filtro utilizzato, cioè la parola ”bitcoin”.

2.4 Dati strutturati

Lo schema di entrambe le categorie di dati è costante per ogni messaggio rendendoli dati strutturati o semi-strutturati per quanto riguarda i tweets, che presentano uno schema più complicato contente elementi innestati (abbreviati nell'esempio per motivi di spazio) e del testo. Nonostante ciò, anche per il fatto che dell'intero tweet i campi analizzati saranno il testo e le coordinate temporali, è possibile utilizzare una rappresentazione relazionale.

3 Componenti software

Seguono le componenti software utilizzate e realizzate per raggiungere l'obiettivo.

3.1 Kafka

Apache Kafka è una piattaforma di streaming distribuita open-source (Apache 2.0 [3]) che svolge il ruolo di message broker dove dei "producer" possono pubblicare messaggi, che verranno poi consumati per essere analizzati, su specifici topic. Ciò consente a più producer di pubblicare su topic distinti o sul medesimo, utile per esempio nel caso si desideri collezionare gli stessi dati da più fonti (in questo contesto aggiungere fonti secondarie potrebbe frazionare il rischio relativo all'interruzione dei dati riguardanti l'andamento della criptovaluta sul mercato).
La scelta è ricaduta sul suddetto software perché in grado di garantire high-availability, fault-tolerancy, ottime performance scalabili orizzontalmente grazie alla possibilità di configurare un cluster di più nodi. [4]
In più grazie risulta ben integrato con Apache Spark, framework su cui si basa l'applicazione analitica.

3.2 Producers

Il ruolo dei producer consiste nel ricevere dati in streaming e di pubblicarli prontamente su di un topic Kafka.
Entrambi i producer sono stati realizzati in python per semplicità nell'utilizzo delle librerie fornite dalle fonti per l'acquisizione di dati in tempo reale.
Allo scopo di valutare le performance dell'intero progetto entrambi i producer aggiungono ad ogni messaggio un campo contenente l'esatto momento di ricezione.

3.2.1 Binance producer

Binance producer riceve dati dalla piattaforma di exchange Binance [1] avvalendosi della libreria ufficiale per poi pubblicarli su Kafka.

3.2.2 Tweets producer

Tweets producer fa uso della libreria Tweepy [5] per ottenere tweets relativi ad un determinato filtro e poi pubblicarli su Kafka.

3.2.3 Funzionamento

La logica di entrambi i producer si può riassumere nelle seguenti fasi:

1. Connessione alla fonte di dati
2. Connessione a Kafka
3. Definizione di una funzione da eseguire alla ricezione di nuovi messaggi che principalmente si occuperà della loro pubblicazione su Kafka

Segue un estratto di codice da "Tweets producer":

```
# Connessione a Kafka
producer = kafka.KafkaProducer(
                bootstrap_servers=kafka_servers,
                value_serializer=lambda x:
                json.dumps(x).encode('utf-8'))

class TweetsStreamListener(tweepy.StreamListener):
    # Definizione della funzione
    def on_status(self, tweet):
        if verbose:
            print(json.dumps(tweet._json))
        msg = tweet._json
        msg["receivedat"] = time.time()
        producer.send(kafka_topic, value=msg)

tweetsStreamListener = TweetsStreamListener()
# Connessione a Twitter
twStream = tweepy.Stream(auth = auth,
                listener=tweetsStreamListener)
twStream.filter(track=tweets_filter)
```

3.3 TimescaleDB

TimescaleDB è un database relazionale open-source ottimizzato per dati di tipo time-series che offre strutture adeguate volte ad incrementare le performance dalla ricerca all'inserimento fino allo storage; ad esempio permette di impostare un intervallo di tempo dopo il quale i dati vengono compressi [6].
Si basa su PostgreSQL, noto database relazionale open-source [7], più precisamente ne è un estensione. Per questo motivo TimescaleDB risulta relativamente semplice da utilizzare per coloro che hanno già avuto esperienza con PostgreSQL ed è inoltre compatibile con i client da esso supportati.
Ciò semplifica la connessione con l'applicazione Spark che grazie al modulo Spark SQL è in grado di utilizzare driver JDBC per interagire con database relazionali tra cui appunto PostgreSQL [8].

3.4 Spark

Apache Spark è un framework open-source per l'analisi di dati su larga scala che offre ottime performance sia per elaborazioni batch che in streaming. Offre API di alto livello nei linguaggi Scala, Java, Python e R, e librerie aggiuntive come MLlib, che fornisce strumenti utili al machine learning, Spark SQL, modulo per l'elaborazione di dati strutturati, e Structured Streaming basato su Spark SQL ma rivolto ai dati strutturati in streaming [9].
Le API Spark consentono di sviluppare applicazioni scalabili orizzontalmente, highly-available e fault-tolerant; infatti possono essere eseguite su cluster comprendenti diverse macchine tra cui almeno un **master** ed un **worker**.
Spark risulta significativamente più performante del paradigma MapReduce soprattutto per quegli algoritmi che necessitano ripetute letture degli stessi dati poiché al contrario di MapReduce non necessita di rileggere i dati da disco ma utilizza una cache in memoria [10].

3.4.1 StreamApp

Applicazione sviluppata utilizzando le API Scala di Apache Spark. Rappresenta il cuore del progetto: si occupa di ricevere, elaborare e analizzare sia i dati finanziari che i tweets.

3.4.2 TrainApp

Applicazione aggiuntiva anch'essa sviluppata utilizzando le API Scala di Apache Spark che si occupa del training di alcuni modelli utilizzati da StreamApp a partire dai dati ricevuti da essa. Per questo motivo StreamApp è in grado di funzionare (parzialmente) anche in assenza di questi modelli cosicché sia possibile ottenere una quantità di dati iniziale con cui allenare i modelli.

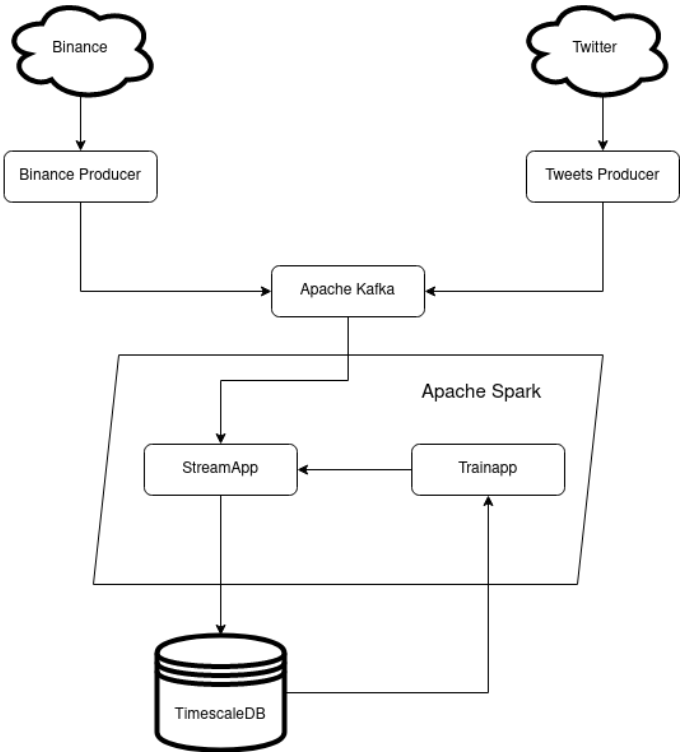

Figura 1: Schema delle componenti principali

4 Applicazione

L'applicazione principale del progetto è la StreamApp composta principalmente da due Streaming Query e da un task eseguito periodicamente.

Grazie a Spark Structured Streaming è infatti possibile trattare dati strutturati ottenuti in real-time da fonti come Apache Kafka similmente a delle query SQL utilizzando strutture chiamate Dataframe.

Nella Figura 2 è riportata gran parte della Streaming Query che si occupa nell'ordine di:

1. Configurare le opzioni di connessione a Kafka.

2. Applicare uno schema corretto ai dati (il corpo dei messaggi Kafka è contenuto nel campo value) rinominandoli opportunamente e utilizzando casting al tipo corretto ove necessario.

3. Aggiungere eventuali informazione ai dati; per esempio viene aggiunto il campo "processedat" con il timestamp corrente allo scopo di facilitare a valutazione delle performance dell'applicazione.

4. Scrivere i dati nel database TimescaleDB (attraverso il driver jdbc di postgresql). Questa operazione viene svolta attraverso la direttiva foreachBatch in quanto attualmente non è possibile scrivere dati da una Streaming Query direttamente in un database, ma è possibile farlo a partire da normali Dataframe. Perciò avvalendosi di foreachBatch si può ovviare a questo problema aggregando iterativamente i dati ottenuti da una Streaming Query in semplici Dataframe il cui contenuto può essere scritto semplicemente in un database.

4.1 Analisi

L'analisi effettuata cerca di valutare la positività dei tweets ricevuti (il target è quindi binario) attraverso modelli allenati a partire dai dati ricevuti utilizzando come target l'andamento del prezzo di ask di BTCUSDT relativo al minuto di pubblicazione del tweet in modo che sia positivo ogni minuto il cui successivo abbia un prezzo di ask medio superiore e negativo altrimenti.

In Figura 2 è illustrato come i dati finanziari contenenti i prezzi di ask vengono ottenuti e salvati nel database, tuttavia la singola Streaming Query non è sufficiente a calcolare il valore medio del prezzo di ask per minuto;

```
// Schema del json
val binance_schema = new StructType()
      ...
      .add("a", "string")
      .add("timestamp", "string")

val binance = spark.readStream
    .format("kafka")
    .option("kafka.bootstrap.servers",
       kafkaBootstrapServers)
    .option("subscribe", pricesTopic)
    .load.select(
    from_json($"value".cast("string"),
       binance_schema).alias("value"))
    .withColumn("askprice", $"value.a".cast(
       DoubleType))
    ...
    .withColumn("timestamp",
    ($"value.timestamp".cast(DoubleType)).cast(
       TimestampType))
    .writeStream
    .foreachBatch {
        (batchDF: DataFrame, batchId: Long) =>
    batchDF.withColumn("processedat",
            current_timestamp())
        .write
        .format("jdbc")
        .option("driver", "org.postgresql.Driver
           ")
        .option("url", jdbcUrl)
        ...
        .save()
    }.start()
```

Figura 2: Esempio di Streaming Query

```
val averagePerMin = priceDB
    .groupBy(window($"timestamp", "1 minute"))
    .agg(avg("askprice"))
    ...

val trendPerMin = averagePerMin
    .join(
        averagePerMin
        //renaming column to new_*
        ...
    )
    .filter(expr("timestamp + interval '1 minute' =
       new_timestamp"))
    .withColumn("asktrend", $"new_avgaskprice" >= $"
       avgaskprice")
```

Figura 3: Calcolo del prezzo di ask medio al minuto

sono infatti necessarie due query: una per salvare i dati ed una seconda per computare il prezzo medio ogni minuto. Per ovviare a quest'ultima esigenza StreamApp istanzia un timer che periodicamente esegue le operazioni necessarie per calcolare il prezzo di ask medio e la polarità di ogni minuto a partire dai dati salvati nel database per poi scriverne i risultati in una specifica tabella. Il procedimento è riassunto in Figura 3.

Infine l'ultima Streaming Query si occupa dei tweets; effettua fondamentalmente le stesse operazioni dell'altra streaming query eccetto che la fase di aggiunta di informazioni è più complicata in quanto esegue l'analisi vera e propria per classificare la polarità dei tweets attraverso modelli precedentemente allenati tramite la TrainApp.

La classificazione risulta essenzialmente una "Sentiment Analysis" supervisionata e sfrutta alcune funzionalità della libreria MLlib di Spark; implica nell'ordine di:

1. Separare l'input in token.
2. Rimuovere le cosiddette "stop-words".
3. Eseguire lo "stemming" su ogni token.

4. Trasformare l'input tramite word embedding in modo da ottenere una rappresentazione utilizzabile come input per allenare un modello di machine learning; è stato utilizzato Word2Vec.

5. Infine fornire la rappresentazione ottenuta tramite Word2Vec come input ad una Regressione Logistica, modello in grado di classificare target binari.

Ognuno di questi step, eccetto lo stemming, utilizza funzioni fornite dalla MLlib di Apache Spark e per questo motivo si tratta di operazioni scalabili orizzontalmente sui vari nodi del cluster. Per quanto riguarda lo stemming invece non è stato possibile utilizzare librerie sviluppate appositamente per l'ultima versione di Spark ma è stato utilizzato Apache OpenNLP, tuttavia riuscendo ugualemente a rendere l'esecuzione di questo step distribuibile su più nodi.

In Figura 4 è illustrato uno schema del funzionamento di StreamApp.

4.2 Training

Il training dei modelli è svolto per conto della TrainApp: a partire dai tweets salvati nel database sono stati eseguiti alcuni degli stessi passaggi necessari anche per l'analisi in streaming, quali:

1. Tokenizzazione.

2. Rimozione delle stop-words.

3. Stemming.

Per poi utilizzare i token rimasti per il training del modello Word2Vec, ed in seguito impiegare la rappresentazione Word2Vec, come input, insieme all'andamento del prezzo di ask al minuto, come target, per allenare il modello di Regressione logistica.

4.3 Performance del modello

La TrainApp stessa prima della creazione del modello finale utilizzando la totalità dei dati ne esegue una validazione dividendo il dataset in trainset e testset e computando il valore AUC (Area Under ROC). Purtroppo i risultati

ottenuti non sono dei migliori: i valori di AUC sono intorno a 0.5, cioè le performance sono paragonabili a una scelta random.

Ciò può significare che il contenuto dei tweets generalmente non è correlato all'andamento del prezzo di ask. Potrebbe perciò rivelarsi utile un filtraggio manuale di un insieme ristretto di tweets in modo da selezionare solo quelli rilevanti.

In aggiunta un'ulteriore operazione manuale di etichettatura potrebbe chiarire se l'andamento dei prezzi sia un parametro adeguato a definire la polarità dei tweets.

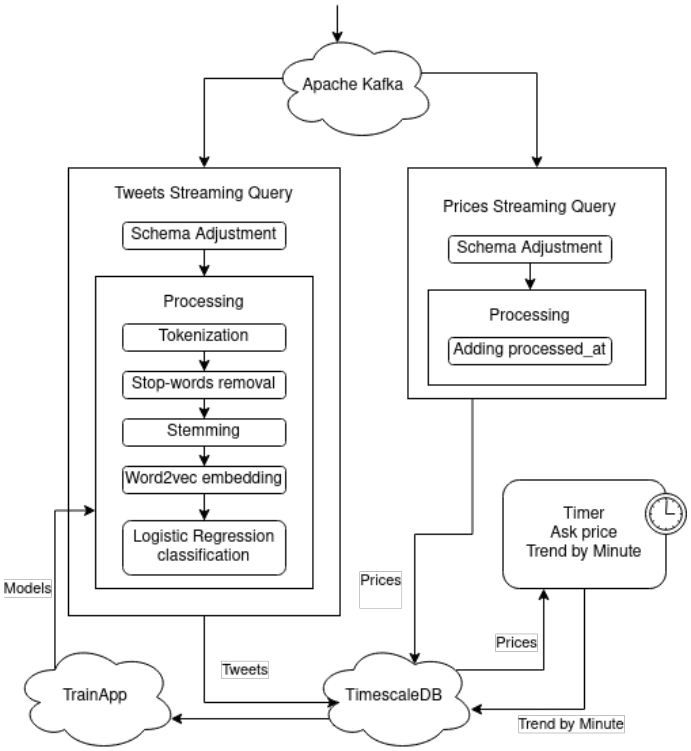

Figura 4: Schema del funzionamento di StreamApp

5 Risultati

Per dimostrare la scalabilità orizzontale dell'applicazione sono state utilizzate le seguenti macchine:

1. un laptop con Intel core i5 e 8gb ram,
2. un laptop con Intel core i7 e 16gb ram

La maggior parte dei servizi necessari all'applicazione sono stati eseguiti sulla prima macchina: TimescaleDB, Kafka, i producer, l'istanza master di Spark, alcune istanze worker ed infine il processo submit di Spark che avvia l'esecuzione della StreamApp all'interno del cluster.

Nella seconda macchina sono stati eseguiti soltanto Spark workers appartenenti allo stesso cluster.

Ogni worker dispone di 1 cpu e 1gb di ram ed il loro numero è stato incrementato gradualmente allo scopo di testare la scalabilità dell'applicazione in questo modo:

- inizialmente (alle 20:50 circa) è stato eseguito un solo worker nella prima macchina
- tre minuti dopo ne è stata eseguita un'altra istanza sulla prima macchina
- d'ora in poi ogni 3 minuti è stato aggiunto un worker sulla seconda macchina fino a a raggiungere un totale di 10 worker sulla seconda macchina più 2 sulla prima
- infine sono stati aggiunti altri 4 worker sulla seconda macchina sempre ad intervalli di tre minuti.

Figura 5: Tempo di elaborazione ed analisi dei dati

Nella Figura 5 è riportato il grafico che mostra il tempo impiegato nell'elaborare ed analizzare tweets (in giallo) e prezzi (in verde) ricevuti calcolato come differenza tra il momento di ricezione (aggiunto dai producer) ed il momento subito precedente all'inserimento nel database.

Si può notare che generalmente i tempi richiesti (mediamente 300ms per i prezzi e 750ms per i tweets) siano pressoché indipendenti dal numero di worker, probabilmente significando che uno o due worker sono in grado di gestire il flusso medio di dati ricevuto (mostrato in Figura 6).

Ed è evidente come aggiungendo worker diminuiscano numero e frequenza dei "picchi", probabilmente dovuti ad oscillazioni nella frequenza dei messaggi ricevuti, in favore della stabilità complessiva del sistema.

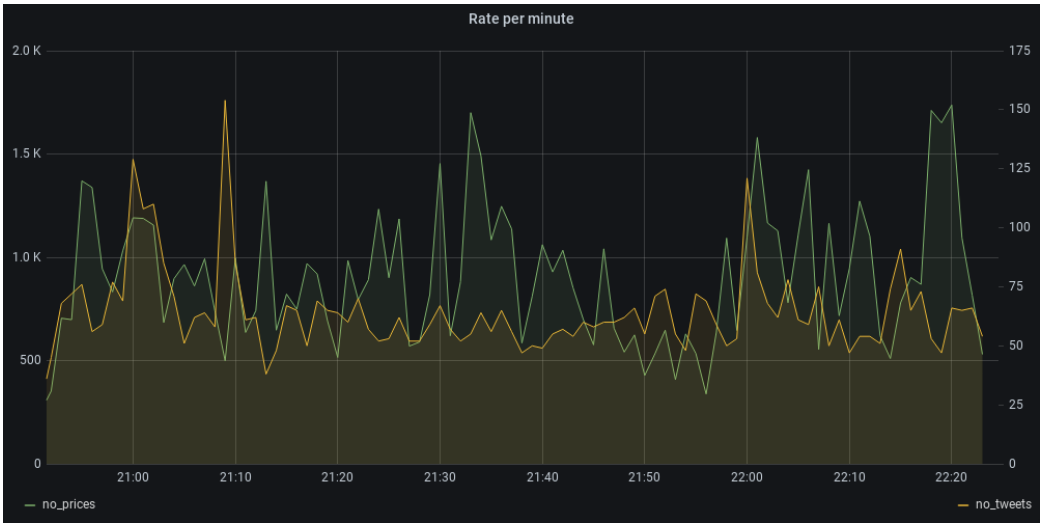

Figura 6: Numero di messaggi ricevuti al minuto

In Figura 6 è riportato il numero di messaggi ricevuti al minuto da Twitter, in giallo sull'asse delle ordinate destro, e da Binance, in verde sulle ordinate a sinistra, nell'intervallo di tempo preso in analisi. Riassumendo sono stati ricevuti ogni minuto in media 895 messaggi relativi ai prezzi con un massimo di 1737, mentre per quanto riguarda i tweets mediamente 64 con un massimo di 154.

È possibile notare una corrispondenza tra i massimi del grafico in Figura 6 con i picchi di quello in Figura 5 che dimiuisce all'aumentare del numero dei worker fino a svanire per i massimi dalle 22 in poi.

Riferimenti bibliografici

[1] "Binance." https://www.binance.com/en/.

[2] "Twitter." https://twitter.com/.

[3] The Apache Software Foundation, "Apache 2.0 license." https://www.apache.org/licenses/LICENSE-2.0.

[4] Apache Software Foundation, "The apache kafka documentation." https://kafka.apache.org/documentation/#uses_messaging.

[5] "Tweepy." https://www.tweepy.org/.

[6] Timescale Inc., "Timescaledb documentation." https://docs.timescale.com/latest/main.

[7] The PostgreSQL Global Development Group, "About postgresql." https://www.postgresql.org/about/.

[8] The Apache Software Foundation, "Jdbc to other databases." https://spark.apache.org/docs/latest/sql-data-sources-jdbc.html.

[9] The Apache Software Foundation, "Apache spark documentation." https://spark.apache.org/docs/latest/.

[10] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," *USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*, 2010.