Cálculo del exponente de Hurst utilizando Spark Streaming: enfoque experimental sobre un flujo de transacciones de criptomonedas

María José Basgall[1,3], Waldo Hasperué[1,2], Marcelo Naiouf[1,2], Aurelio F. Bariviera[4]

[1]Instituto de Investigación en Informática (III-LIDI), Facultad de Informática - Universidad Nacional de La Plata
[2]Comisión de Investigaciones Científicas (CIC)
[3]UNLP, CONICET, III-LIDI, La Plata, Argentina
[4]Department of Business, Universitat Rovira i Virgili, Av. Universitat 1, 43204 Reus, Spain

{mjbasgall, whasperue, mnaiouf}@lidi.info.unlp.edu.ar
aurelio.fernandez@urv.cat

Resumen. Actualmente es cada vez más común encontrarse con problemas de Big Data, donde las aplicaciones desarrolladas para resolver dichos problemas son implementadas en frameworks específicos. Uno de los que más se utiliza es Apache Spark, que posee el módulo Spark Streaming el cual permite el tratamiento de datos provenientes de un flujo de información potencialmente infinito. En este trabajo se presenta una aplicación implementada en Spark Streaming que realiza el cálculo del exponente de Hurst, un indicador muy utilizado en el análisis de mercado para la detección de memoria a largo plazo. Los ensayos realizados se hicieron sobre flujos simulados de transacciones de criptomonedas que demuestran la capacidad de Spark Streaming para el tratamiento de este tipo de flujos.

Palabras clave: Spark Streaming, Stream processing, Criptomonedas, Exponente de Hurst.

1 Introducción

En la actualidad, se ha incrementado la capacidad de recolectar y almacenar información por lo que trabajar con grandes cantidades de datos es cada vez más usual. Debido a que las técnicas tradicionales de análisis de datos que son ejecutadas en una computadora no son capaces de lidiar con estos volúmenes de datos por cuestiones de escalabilidad, surge el término Big Data, que se refiere a la aparición de una enorme cantidad de datos heterogéneos y las técnicas de procesamiento y análisis que se requieren para analizarlos en tiempos razonables de acuerdo al problema [1][2][3].

Actualmente existen diferentes herramientas que simplifican la tarea de extraer conocimiento frente a estos escenarios. Una de las herramientas más utilizadas es Hadoop MapReduce [4][5], construida sobre los principios del procesamiento

paralelo y distribuido. MapReduce trabaja sobre el sistema de archivos HDFS [6], el cual es distribuido, ofrece eficiencia, almacenamiento tolerante a fallos y es adecuado para aplicaciones que utilizan grandes volúmenes de datos ya que proporciona un alto rendimiento de acceso a los mismos. El framework MapReduce realiza las tareas de distribución y paralelización de manera automática y transparente al programador, convirtiéndola en una poderosa herramienta para implementar soluciones distribuidas.

A pesar de su popularidad, se han encontrado múltiples limitaciones para desarrollar programas escalables con MapReduce, ya que es ineficiente para aplicaciones que comparten datos a través de varias etapas de un algoritmo, como en tareas iterativas, debido al uso intensivo de disco generando un importante decremento de performance. Además MapReduce tiene su punto fuerte en el procesamiento off-line, lo cual lo convierte en una herramienta no muy útil en problemas donde se necesita una respuesta dinámica y en tiempo real.

Por lo anterior, han surgido diferentes plataformas que sobrellevan los problemas que presenta MapReduce, una de ellas es Apache Spark [7][8], la cual está siendo cada vez más usada debido a que es una de las más flexibles y poderosas para realizar cómputo distribuido de manera rápida en escenarios Big Data mediante el uso intensivo de memoria RAM, permitiendo leer los datos en ella y consultarlos repetidamente, siendo una característica deseable para algoritmos que usan los datos de manera iterativa.

Spark tiene dos modos de trabajo: batch y online. Este último permite analizar los datos provenientes de un flujo de datos con la posibilidad de ofrecer una respuesta online. Estos flujos de datos pueden ser de diversas fuentes: publicaciones de usuarios en redes sociales, logs generados por servidores, datos provenientes de sensores, navegación web o transacciones monetarias como lo pueden ser el reciente uso del comercio utilizando criptomonedas.

En relación a las criptomonedas, en los últimos años se ha comenzado a utilizar un nuevo tipo de moneda digital que se podría caracterizar como "sintéticas" en el sentido de que no surgió por decisión de un Estado sino que emergió de un acuerdo privado y facilitado por el anonimato que se puede obtener de internet. Las monedas digitales o criptomonedas permiten pagos instantáneos a cualquiera y en cualquier lugar del mundo.

La más importante de ellas es la llamada Bitcoin (BTC) [9] cuyo software fue creado por Satoshi Nakamoto bajo licencia MIT [11]. Y la idea consiste en usar criptografía para controlar la creación y transferencia de dinero de manera descentralizada y sin tener que confiar en las autoridades centrales.

Existen alrededor de 750 criptomonedas, además de Bitcoin. Las que siguen en popularidad son Ethereum (ETH), Ripple (XRP), Litecoin (LTC). Sin embargo, se puede observar en la figura 1 donde se muestra el porcentaje de capitalización bursátil que BTC representa el 47% del mercado, seguido por ETH con un 23%.

En ciencias económicas un indicador que resulta muy útil es el de determinar la presencia de memoria o correlación a largo plazo en un mercado para saber si se pueden construir estrategias de negociación que permitan extraer un rendimiento superior al mercado. Una forma de realizar esa tarea es mediante el cálculo del exponente de Hurst [12]. El poder asegurar que existe una correlación a largo plazo es

un desafío al modelo financiero establecido, ya que de acuerdo a la teoría económica estándar, en un mercado competitivo los precios se debería mover aleatoriamente, reflejando una serie temporal sin memoria. Lograr este objetivo es de mucho interés para las personas que invierten en estos tipos de mercado. En un trabajo reciente [13], se realizó un estudio del exponente de Hurst durante el período 2011-2017, detectando una variabilidad en dicho exponente a lo largo del tiempo. Por otro lado, se comprobó que dicho exponente tiene un comportamiento similar cuando es medido en distintas escalas temporales (rendimientos de 5 a 12 horas). Por ello, un avance natural en esta línea de investigación consiste en estudiar el comportamiento del exponente de Hurst en un entorno online.

En este artículo se presenta el cálculo del exponente de Hurst mediante el método DFA (Detrended Fluctuation Analysis) para un flujo de datos simulado compuesto por transacciones de criptomonedas utilizando el framework Spark Streaming. Este trabajo es a modo experimental ya que si bien es posible encontrar en internet bases de datos de transacciones de criptomonedas, aún no es posible obtener de manera online los datos de las transacciones que se necesitan para el cálculo del exponente de Hurst. Sería esperable, dado el avance de las tecnologías de la información y la comunicación, que en un futuro este tipo de flujo esté disponible en acceso abierto para diferentes análisis.

El resto del artículo se divide de la siguiente manera: en la sección 2 se describe el framework Spark streaming. En la sección 3 se describe el algoritmo del cálculo del exponente de Hurst y su implementación en Spark Streaming. En la sección 4 se describen los ensayos realizados sobre flujos simulados de transacciones usando criptomonedas y finalmente en la sección 5 se mencionan las conclusiones y los trabajos a futuro en los que se está trabajando.

2 Spark Streaming

El procesamiento de flujos de datos (*stream processing*) es un área muy estudiada en los últimos años. Stream processing permite llevar a cabo tareas sobre un flujo continuo y potencialmente infinito de datos [14][15][16][17]18].

El objetivo de este tipo de procesamiento es permitir que las tareas analicen los datos de un flujo de forma online, brindando respuestas en tiempos muy cercanos al tiempo real. La principal característica de esta forma de trabajar sobre un flujo de datos, es que los datos del flujo llegan a una velocidad tal que no es posible almacenarlos en su totalidad y, si se pueden almacenar, el volumen de datos es tan grande que presenta la dificultad de analizarlo en tiempos de respuesta cortos [19]. La velocidad es un concepto a tener en cuenta en los escenarios de Big Data, más aún en los escenarios de stream processing.

Existen dos tipos de modelos de procesamiento en stream processing, un dato a la vez y micro-batching. En el primer enfoque, los datos se procesan a medida que llegan y luego se descartan. Por otro lado, con el modelo de micro-batching el cómputo se lleva a cabo sobre pequeños lotes de datos capturados cada un

determinado intervalo de tiempo. Spark Streaming [20] es un framework que implementa el modelo de procesamiento de datos por micro-batching.

Spark Streaming permite el procesamiento de flujos de datos de manera escalable, de alto rendimiento y tolerante a fallos. La ingesta de los datos puede llevarse a cabo mediante diferentes fuentes de datos como puede ser Kafka, Flume, Twitter, ZeroMQ, Kinesis, o sockets TCP. Estos datos pueden ser procesados mediante algoritmos que utilicen las funciones de alto nivel provistas por la librería como son map, reduce, join, etc, para luego ser escritos en archivos, base de datos o mostrados por pantalla.

En Spark se presenta un modelo de abstracción llamado Resilient Distributed Dataset (RDD), el cual es una colección distribuida de datos tolerante a fallas que permite persistir resultados intermedios en memoria. Los RDD también pueden ser almacenados en memoria caché como en disco, según sea necesario. Spark streaming trabaja con los denominados discretized stream (DStream), abstracciones de alto nivel que provee Spark Streaming. Éstos representan un flujo continuo de datos e, internamente, están representados por una secuencia de RDD (el cual es una colección de elementos tolerantes a fallas que puede ser operada en paralelo mediante transformaciones y acciones).

La gran ventaja de utilizar Spark Streaming es que el framework puede ser ejecutado en un cluster, donde cada nodo del cluster se encarga de ejecutar las transformaciones y acciones sobre diferentes RDD, dando lugar al procesamiento de datos en paralelo y distribuido.

Además es posible configurar el tiempo de la ventana temporal al momento de recolectar los datos que serán utilizados en cada micro-batch.

3 Cálculo del exponente de Hurst en Spark Streaming

Las transacciones de criptomonedas se almacenan como tripletas <TS, V, P> guardando el timestamp (TS), el volumen de comercio de criptomonedas (V) y el precio (P) [21]. Para el cálculo del exponente de Hurst solo se utiliza el timestamp (segundos transcurridos desde las 0 horas del 1 de enero de 1970 GMT).

El flujo de datos compuesto por tripletas <TS, V, P> se divide en ventanas temporales de la misma longitud (*LV*) y no solapadas, llamadas batchs o lotes. Cada batch B_i comienza en el tiempo ti_i y finaliza en el tiempo tf_i, siendo $(tf_i - ti_i) = LV$, además $ti_i = 1+tf_{(i-1)}$ Dentro de cada batch es posible que se obtenga del flujo una o más transacciones. De cada batch se extrae la transacción T_i donde su respectivo timestamp (TS_i) es el más cercano a tf_i.

Se calcula la distancia entre la hora de arribo de la transacción y el tiempo del fin del batch como $D_i = tf_i - TS_i$.

Con los D_i se calcula el exponente de Hurst usando el método llamado Detrended Fluctuation Analysis (DFA) que es más apropiado cuando se trabaja con datos no estacionarios [12].

Se calculan los *x(i)* que representan una serie autosimilar de la serie original de precios como lo muestra la ecuación 1.

$$x(i) = \sum_{t=1}^{i} [D_i - \overline{D}] \quad (1)$$

donde $\overline{D}$ es la media aritmética de los D_i, i=1..M.

La serie de los $x(i)$ se divide en M/m submuestras y de cada submuestra se realiza un ajuste polinómico, que por lo general es un ajuste lineal, para obtener los $x_{fit}(i,m)$. m son las longitudes de intervalos dentro de la submuestra, para permitir el comportamiento fractal de la serie temporal.

La ecuación 2 muestra la forma de calcular los coeficientes $F(m)$ utilizados para el cálculo del exponente de Hurst.

$$F(m) = \sqrt{\frac{1}{M} \sum_{i=1}^{M} \left[x(i) - x_{fit}(i,m) \right]^2} \quad (2)$$

Con los pares (ln(m), ln($F(m)$)) se realiza una regresión lineal y la pendiente de dicha regresión es el exponente de Hurst.

3.1 Simulador de flujos

Debido a las dificultades de tomar datos de transacciones de criptomonedas en el mercado real, para el desarrollo de los diferentes experimentos se utilizó un flujo simulado. Si bien es difícil tomar los datos online, éstos se pueden descargar los datos desde los sitios oficiales de las empresas manejadoras de criptomonedas. Por lo tanto la simulación del flujo consiste en leer las transacciones desde un archivo y enviarlas a un socket TCP, ya que Spark Streaming es capaz de procesar datos escuchando estos sockets.

Para esto se implementó un script bash que lee el archivo de las transacciones línea por línea y las envía a la utilidad Netcat [22] presente en la mayoría de los sistemas GNU/Linux para que reciba cada línea del archivo utilizando un puerto específico y, a su vez, en la aplicación Spark Streaming se establece en el contexto que escuche ese mismo puerto para poder recibir los datos que está escuchando el comando Netcat.

Este simulador también permite el envío continuo de transacciones ficticias, creadas aleatoriamente.

3.2 Implementación en Spark streaming

Primero se configura el tamaño de ventana de cada micro-batch al tamaño de ventana deseado para el cálculo del exponente de Hurst. La longitud de la ventana no debe ser inferior a 500 observaciones, a fin de evitar sesgos en la estimación del exponente.

Con esto nos garantizamos que Spark recolecte todas las transacciones del flujo que caen dentro de cada ventana asegurando que todas las ventanas sean del mismo tamaño, condición necesaria para el cálculo del exponente de Hurst.

La implementación realizada en este trabajo está hecha en Java 7, debido a la complejidad del código, se comenta e ilustra con pseudocódigo las diferentes transformaciones y acciones realizadas para el cálculo del exponente de Hurst.

Mediante funciones de mapeo y reducción se obtiene la transacción T_i que representa la transacción más cercana al tiempo de finalización del batch, utilizando para ello una clave única, ya que la siguiente acción a tomar es la reducción de todas las tuplas recolectadas en la ventana:

```
mapeo = batch.mapToPair(tupla -> (1,tupla))

ultima_transacción = mapeo.reduceByKey
              (a, b -> (a.timestamp > b.timestamp)?a:b)
```

Luego se genera, con la última transacción obtenida, una tupla usando el timestamp de la transacción como clave y la distancia D_i como valor. Esta tupla se persiste ya que es necesaria tener almacenado el "historial" de las últimas transacciones de cada batch del flujo procesado. Cabe aclarar que las siguientes operaciones se realizan por cada una de las ventanas de datos analizadas, donde cada ventana agrega una nueva transacción al "historial" de transacciones.

Utilizando la clase StatCounter provista por Apache Spark se calcula la media de las distancias D_i y se obtiene la menor de ellas, esta última es usada para calcular el número de lote (valor *i*) al cual pertenece la transacción obtenida en la última ventana:

```
map_stats = historial.updateStateByKey
 (values, current -> (new StatCounter(values, current))

stats = map_stats.mapToPair(sc -> (1, sc.mean))
```

Para el cálculo de los $x(i)$ primero se mapea la diferencia $(D(i)-\overline{D})$ usando el valor de *i* como clave:

```
dif_D = historial.mapToPair(t -> (i, t - sc.mean))
```

Luego se hace el producto cartesiano de dif_D para crear una relación que permita calcular la sumatoria de las diferencias $(D(i)-\overline{D})$ y así obtener la serie *x(i)*.

```
prod_carte = dif_D.transformWith(dif_D)
               (rdd_a, rdd_b -> rdd_a.cartesian(rdd_b))
```

Cada tupla de prod_carte (i_a, D_i_a, i_b, D_i_b) se mapea a un par usando i_a como clave y Di_b como valor si $i\_b \leq i\_a$ o 0 en caso contrario. La reducción es usada para sumar los valores y así obtener la serie *x(i)*.

```
x(i) = prod_carte.reduceByKey(a, b -> a+b)
```

Para el cálculo de la regresión la serie *x(i)* se mapea usando un par (*m*,*l*) como clave y *x(i)* como valor, donde *l* es el número de submuestra al que pertenece cada valor *x(i)*. La idea es poder aprovechar al máximo la capacidad de Spark Streaming distribuyendo, por cada valor de *m* y submuestra *l*, el correspondiente cálculo de la regresión polinómica (lineal en este trabajo) y el cálculo de la serie $x_{fit}(i,m)$.

```
map_regres = x(i).flatMap(a -> [(m, l), a.x_value])
```

Como en Spark la reducción se hace por pares de valores no pudiendo acceder a todos los mismos en una misma operación, nos vimos obligados a hacer un mapeo extra para juntar todos los *x(i)* en un único string y así poder acceder a todos los *x(i)* pertenecientes a un mismo submuestreo para realizar la regresión lineal.

```
map_aux = map_regres.mapToPair
                      (a -> (a.m, a.l), a.x.toString()])

submuestras = map_aux.reduceByKey(a, b -> a+b)
```

Luego se usa la operación flatMap para generar los $x_{fit}(i,m)$. También se aprovecha la operación para devolver una tupla con el valor *i* como clave y como valor la diferencia $(x(i) - x_{fit}(i, m))^2$.

```
xfit_i_m = submuestras.flatMap(a -> (a.m, xfit))
```

Luego se hace un mapeo por valor de *m* y la reducción correspondiente para el cálculo de la sumatoria.

```
map_sum = xfit_i_m.mapToPair(a -> (a.m, xfit))

suma = map_sum.reduceByKey(a, b -> a+b)
```

Así obtenemos una tupla por cada valor de *m* utilizado. El anteúltimo paso consiste en el mapeo a un par usando una clave única y los valores de *m* y los *F(m)* como valor. Como también necesitamos todos estos valores al mismo tiempo para el cálculo de la regresión lineal se convierten a string para concatenarlos y poder usarlos en la etapa posterior.

```
map_F_m = suma.mapToPair
               (a -> (1, m.toString(), F(m).toString()))

pares_m_F_m = map_F_m.reduceByKey(a, b -> a+b)
```

El último paso es el que recibe la única tupla producto de la operación anterior y con cada par (*m*, *F(m)*) recibido como valor hace la regresión lineal para el cálculo del exponente de Hurst.

```
hurst = pares_m_F_m.map(a, hurst_value)
```

4 Resultados

Las pruebas realizadas se llevaron a cabo utilizando un cluster de máquinas virtuales, cada una de 2 cores y 7 GB de RAM. Los nodos tienen sistema operativo Ubuntu 16.04, Spark 1.6.3 y Hadoop 2.6.5. El script generador de flujo implementado y utilizado en los ensayos genera alrededor de 1400 transacciones por segundo.

Uno de los interrogantes que se planteó durante el desarrollo de este trabajo es cómo varía la performance de Spark cuando se varía el tamaño de la ventana. Como con cada batch recolectado se guarda una transacción, al cabo de n batches se tendrán n transacciones con las cuales operar para el cálculo del exponente de Hurst. Esto provoca que el cálculo del exponente tarde cada vez más tiempo, ya que con cada ventana analizada se obtiene una nueva transacción, la cual obliga a realizar todos los cálculos.

Como primer experimento se simuló un flujo de transacciones utilizando dos tamaños de ventana diferentes para medir la performance del procesamiento de Spark. Se ejecutó el proceso en dos clusters configurados con uno y dos nodos. Los tiempos de ejecución de cada batch se pueden observar en la figura 1. Se puede observar que el tiempo de procesamiento se mantiene dentro del tiempo de la ventana temporal y que si bien al comienzo hay un overhead, producto del lanzamiento del cluster y la distribución inicial de tareas, luego se estabiliza, demostrando la capacidad de Spark de tratar flujos de alta frecuencia de datos.

5 Conclusiones y trabajo a futuro

Se presentó un algoritmo implementado en el framework Spark Streaming que obtiene de manera online, el cálculo del exponente de Hurst en un flujo simulado de transacciones de criptomonedas.

Los resultados obtenidos muestran la eficiencia y la capacidad de Spark para tratar con flujos cuya frecuencia es de 1400 transacciones por segundo pudiendo dar respuestas online usando ventanas de tamaño de 5 y 10 segundos. En el mercado real estos flujos tienen una frecuencia mucho menor que la simulada en este trabajo, por lo que el algoritmo presentado en este trabajo al ser ejecutado en Spark Streaming podría tratar de manera online los flujos del mercado real sin ningún problema.

Una de las condiciones que debe cumplir el flujo de transacciones para el cálculo del exponente es que cada batch debe por lo menos tener una transacción. En este trabajo se simuló el flujo de manera de asegurarse que cada batch analizado tenga al menos una transacción. En flujos reales donde la distancia en el tiempo de llegada entre diferentes transacciones no se conoce a priori, es necesario calcular, la distancias máxima entre transacciones.

Como trabajo a futuro se está implementando una aplicación que es capaz de almacenar parte del flujo para el cálculo dinámico del tamaño de ventana LV y así

poder calcular de manera correcta el exponente de Hurst. Esto trae aparejado el desafío de poder almacenar la mayor parte del flujo en memoria, para lo cual el procesamiento paralelo y distribuido que proporciona Spark Streaming es una gran ayuda. Problema que los algoritmos secuenciales no podrían lidiar en tiempo real para flujos de alta frecuencia de llegada de datos.

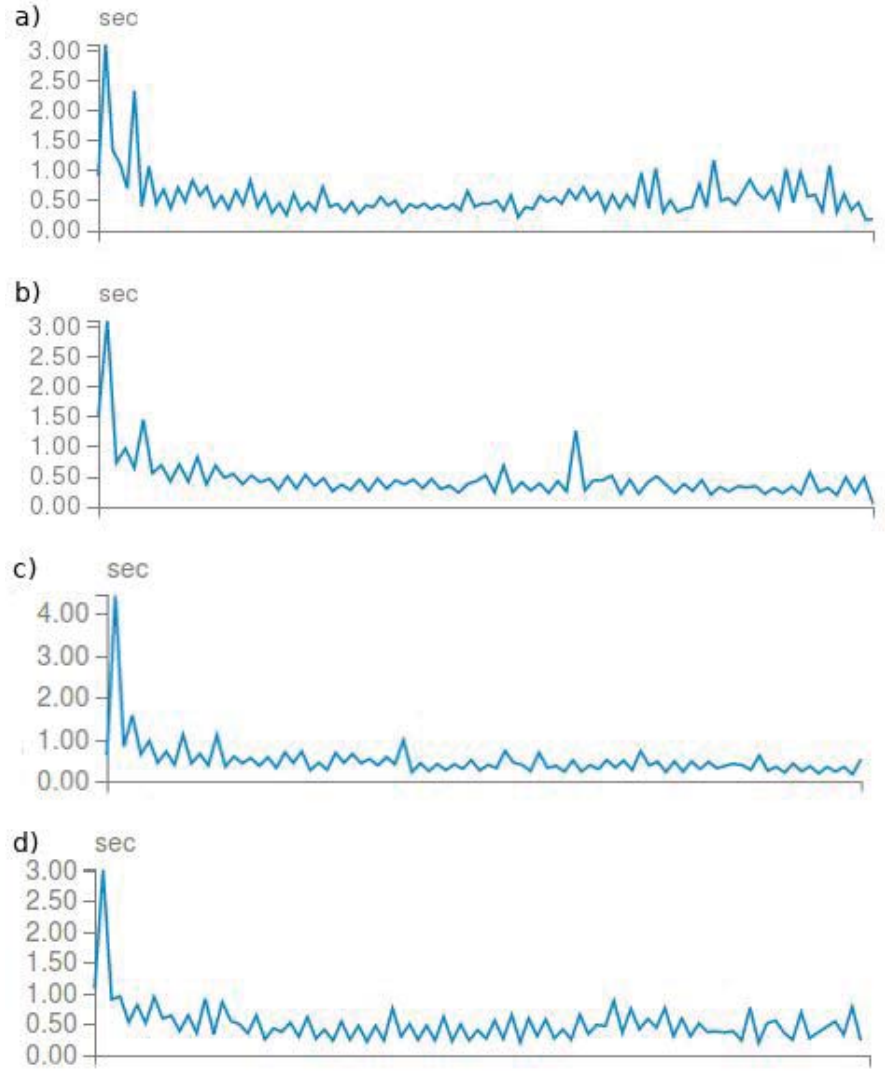

Fig. 1. Tiempo de ejecución de cada batch para:
a) 1 master y 1 slave con ventana de 5 segs. **b)** 1 master y 1 slave con ventana de 10 segs.
c) 1 master y 2 slaves con ventana de 5 segs. **d)** 1 master y 2 slaves con ventana de 10 segs.

Referencias

1. D. Laney, “3D data management: Controlling data volume, velocity, and variety,” tech. rep., META Group, February 2001.
2. S. Kaisler, F. Armour, J. A. Espinosa, and W. Money, “Big data: Issues and challenges moving forward,” 2013 46th Hawaii International Conference on System Sciences (HICSS 2013), vol. 00, no. undefined, pp. 995–1004, 2013.
3. B. Saha and D. Srivastava, “Data quality: The other face of big data” 2014 IEEE 30th International Conference on Data Engineering (ICDE), undefined, pp. 1294–1297, 2014.
4. J. Dean and S. Ghemawat, “MapReduce: Simplified Data Processing on Large Clusters,” Commun. ACM, vol. 51, pp. 107–113, Jan. 2008.
5. Hadoop, White, T.: Hadoop: The Definitive Guide, 4th edn. O’Reilly Media Inc, 2015
6. HDFS. http://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html. Accedido en 07/2017.
7. Apache Spark. http://spark.apache.org/ . Accedido en 07/2017.
8. Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M.J., Shenker, S., Stoica, I.: Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In: Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation, pp. 1–14. USENIX Association (2012)
9. Bitcoin https://en.bitcoin.it/wiki/Main_Page Accedido en 07/2017.
10. https://coinmarketcap.com/currencies/bitcoin/ Accedido en 07/2017.
11. S. Nakamoto, Bitcoin: A peer-to-peer electronic cash system, 2009.
12. A. Carbone, G. Castelli, H.E. Stanley, Time-dependent Hurst exponent in financial time series, Physica A: Statistical Mechanics and its Applications, Vol 344, Issue 1, 2004, pp. 267-271.
13. A.F. Bariviera, M.J. Basgall, W. Hasperué, M. Naiouf, Some stylized facts of the Bitcoin market, Physica A: Statistical Mechanics and its Applications, Vol 484, 2017, pp. 82-90,
14. N. Takahashi et al., “A parallelized data stream processing system using dynamic time warping distance,” in 2009 International Conference on Complex, Intelligent and Software Intensive Systems, Fukuoka, Japan, March 16-19, 2009, pp. 1100–1105.
15. Y. Noh et al., “Real-time data stream processing for ubiquitous home network systems,” in 4th International Conference on Multimedia and Ubiquitous Engineering, MUE 2010, Cebu, Philippines, 11-13 August, 2010.
16. C. Kuka, “Processing the uncertainty: Quality-aware data stream processing for dynamic context models,” in Pervasive Computing and Communications Workshops (PERCOM Workshops), 2012 IEEE International Conference on, pp. 560–561, March 2012.
17. D. Bonino and F. Corno, “spchains: A declarative framework for data stream processing in pervasive applications,” Procedia Computer Science, vol. 10, 2012.
18. J. Stefanowski et al., “Processing and mining complex data streams,” Inf. Sci., vol. 285, pp. 63–65, 2014.
19. A. Rajaraman and J. D. Ullman, Mining of Massive Datasets. New York, NY, USA: Cambridge University Press, 2011.
20. http://spark.apache.org/docs/latest/streaming-programming-guide.html Accedido en 07/2017.
21. http://api.bitcoincharts.com/v1/csv/ Accedido en 07/2017.
22. Netcat http://netcat.sourceforge.net/ Accedido en 07/2017.