Definizione ed Implementazione di un Sistema di Raccomandazione Distribuito per film e Modellazione di Eventi Complessi

Prof. Ing. Tommaso di Noia
Prof.ssa Marina Mongiello
Mauro Losciale
Pietro Tedeschi

Logica e Intelligenza Artificiale
Ingegneria del Software Avanzata
Laurea Magistrale in Ingegneria Informatica
Politecnico di Bari
A.A 2015 - 2016

Indice

1 Introduzione

2 Stato dell'arte

2.1 Introduzione ai sistemi CEP

2.2 Sistemi di Raccomandazione

2.2.1 Filtro collaborativo

2.3 Introduzione allo Stream Processing

2.3.1 Il paradigma Publish-Subscribe

2.4 Il pattern Facade

2.5 Il pattern Singleton

2.6 Il pattern Model-View-Controller (MVC)

2.7 La tecnologia WebSocket

3 Analisi del progetto

4 Soluzione proposta

4.1 La libreria Spark

Apache Spark è un sistema di cluster computing di tipo general-purpose, scalabile e veloce. Dispone di API di alto livello in **Java**, **Scala**, **Python** ed **R**, e un engine ottimizzato che supporta grafi di esecuzione generici. Integra inoltre un ampio set di tool come **Spark SQL**, per structured data processing, **MLlib** per il machine learning e **Spark Streaming**, descritti nelle sezioni successive. Spark è eseguibile sia su sistemi Windows che UNIX-like (Linux, Mac OS).

Una delle possibili configurazioni di un sistema Spark è la modalità *cluster*, mostrata in Figura 1.

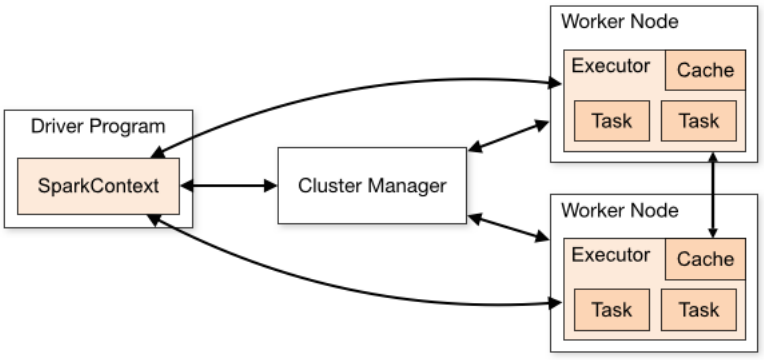

Figura 1: Configurazione in Spark di tipo Cluster Mode

Le applicazioni Spark sono eseguite come un set di processi indipendenti sul cluster, coordinati dall'oggetto *SparkContext* del programma sorgente (detto **driver program**). Precisamente, il programma driver può connettersi su diversi tipi di *cluster managers* (ad esempio un cluster di tipo Standalone, Mesos o YARN), il quale alloca le risorse a disposizione delle applicazioni. Una

volta connesso, Spark scansiona i nodi del cluster alla ricerca degli **executor** (detti anche *worker node*), i quali eseguono effettivamente i task e il data storage delle applicazioni. A questo punto il driver invia il codice dell'applicazione agli executor (tipicamente un file JAR o Python incluso nello SparkContext) e schedula i task per l'esecuzione parallela.

Alcune considerazioni riguardo tale architettura sono:

- Ogni applicazione gestisce i propri workers, i quali restano attivi durante tutto il ciclo di vita ed eseguono task multipli in thread multipli. Questo implica un isolamento tra le applicazioni, sia lato scheduling (ogni driver schedula i propri tasks) sia lato executor (tasks relativi ad applicazioni differenti risiedono in JVM differenti). Tuttavia ciò implica che non è possibile condividere nativamente i dati tra applicazioni diverse, a meno di utilizzare uno storage system esterno;
- Il driver deve poter gestire le connessioni con i workers durante l'intero ciclo di vita dell'applicazione. Per questo motivo dev'essere sempre garantita la visibilità a livello di rete tra driver e workers durante l'esecuzione;
- È necessario che driver e worker abbiano, a livello di rete, una distanza relativamente breve, preferibilmente nella stessa LAN, affinché lo scheduling sia rapidamente eseguito.

Il principio di funzionamento di Spark si basa sostanzialmente sul concetto di *Resilient Distributed Dataset* (**RDD**). Un RDD è una collezione di dati su cui è possibile operare parallelamente, ed è distribuita su tutti i nodi del cluster come file system Hadoop oppure è generata da una collezione esistente in Java o Scala.

Una seconda astrazione è rappresentata dalle variabili condivise (*shared variables*), utilizzate nelle computazioni parallele. Di default Spark tiene traccia delle variabili istanziate nei vari task, e consente se necessario di condividerle fra task o fra task e driver. Le variabili condivise possono essere di due tipi: di tipo *broadcast*, il cui valore viene salvato nella cache per ogni nodo, e di tipo *accumulatore*, per esempio contatori o sommatori.

4.1.1 Spark Streaming

Spark Streaming è un'estensione delle Core API di Spark per lo **stream processing** di live data streams ad alto throughput. Supporta molteplici sorgenti di data stream come **Kafka**, Flume, Twitter, ZeroMQ, Kinesis o socket TCP, i quali possono essere processati tramite direttive come *map*, *join*, *reduce* e *window*. Nel post processing è possibile salvare i data stream in un file system, in un database o visualizzarli in una live dashboard. Come ulteriore fase nella pipeline di operazione rientra anche il machine learning ed il graph processing. In Figura 2 viene riassunta l'architettura descritta.

Figura 2: Architettura di Spark Streaming

Nello specifico, i data streams ricevuti vengono suddivisi in frammenti (*batches*), processati da Spark per generare lo stream finale risultante in batches, come mostrato in Figura 3.

Figura 3: Spark Streaming Data Stream Processing

A livello alto il flusso continuo di dati è rappresento da una struttura astratta detta *discretized stream* o *DStream*, il cui contenuto è rappresentato da tutte le sorgenti collegate eventualmente con Spark, o da stream risultati da altri DStream. Internamente, un DStream è rappresentato tramite una sequenza di RDD.

4.1.2 Spark MLlib

4.1.3 Spark SQL

Spark SQL è un modulo di Spark per il processing di dati strutturati. A differenza delle API RDD la sua interfaccia consente di descrivere in maniera dettagliata la struttura e le operazioni da eseguire sui dati, in stile SQL. Le strutture astratte di riferimento per questo modulo sono i **Dataframes** e i **Datasets**.

Un **Dataframe** è un collezione distribuita di informazioni organizzata in colonne e attributi. Concettualmente è equivalente ad una tabella in un database relazionale, con caratteristiche aggiuntive fornite da Spark. In accordo con tale definizione sono presenti, oltre alle classiche funzionalità SQL, la creazione di DataFrame a partire da un RDD o un oggetto JSON e viceversa.

Un **Dataset** invece è un'interfaccia sperimentale introdotta nella versione 1.6 di Spark, mirata all'integrazione delle API RDD con l'engine SQL. Attualmente il supporto è limitato alle API Java e Scala.

4.2 Apache Kafka

Apache Kafka è un sistema di messaggistica di tipo publish-subscribe, orientato alla distribuzione. La sua architettura consente ad un singolo cluster di agire da backbone centrale per i dati di grandi organizzazioni, e gli stream vengono partizionati e distribuiti lungo tutti i nodi del cluster, sfruttando la potenza di calcolo di ogni singola macchina. Di seguito si introduce la terminologia utilizzata in Kafka:

- Kafka organizza i flussi dei messaggi in categorie chiamate *topics*;
- I processi che si occupano di pubblicare i messaggi in Kafka sono chiamati *producers*;
- I processi che effettuano una sottoscrizione ad un topic ed elaborano i messaggi pubblicati sullo stesso sono chiamati *consumers*;
- I nodi all'interno del cluster(producers e consumers) sono coordinati da uno o più server chiamati *brokers*.

A livello concettuale il funzionamento di Kafka è riassunto in Figura 4. La comunicazione tra client e server avviene tramite protocollo TCP. Sono presenti varie implementazioni del client Kafka, disponibili in vari linguaggi tra cui Java, Javascript e PHP.

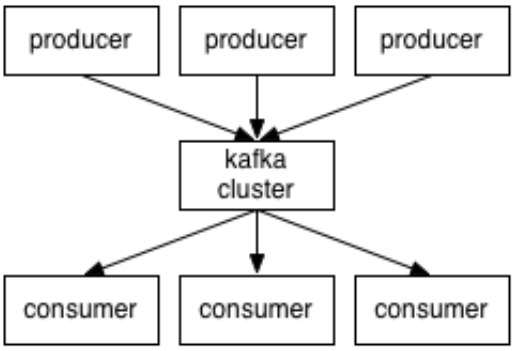

Figura 4: Architettura di Apache Kafka

Di seguito viene mostrata nel dettaglio la struttura di un topic in Figura 5. Per ogni topic Kafka effettua un partizionamento dei messaggi in arrivo, tenendo traccia dell'ordine di arrivo con un sistema di log.

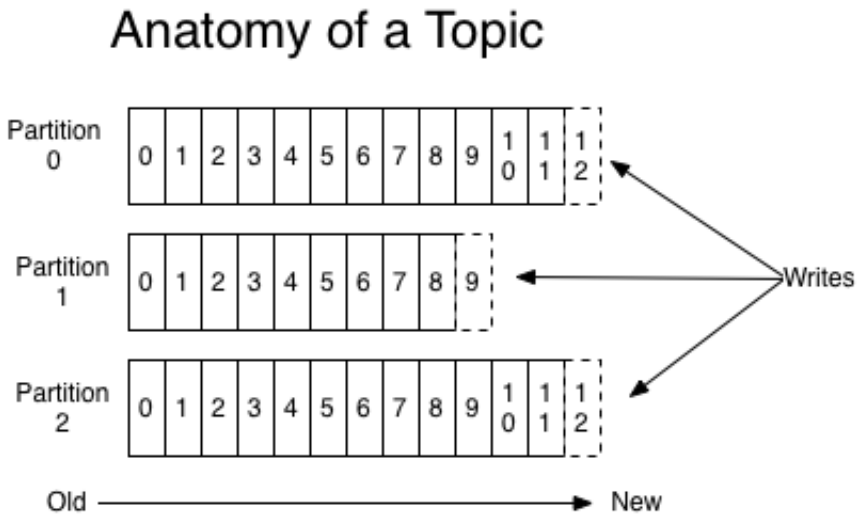

Figura 5: Struttura di un topic in Kafka

Ogni partizione è una sequenza immutabile di messaggi, registrata e ordinata in un commit log. Ad ogni messaggio presente viene assegnato un id sequenziale, detto *offset*, che identifica univocamente un messaggio nella relativa partizione.

Il cluster di Kafka conserva in memoria tutti i messaggi pubblicati, letti o non dai consumers, per un periodo di tempo configurabile. In pratica, il server registra dei metadati relativi ad ogni posizione del consumer nel log, chiamato appunto offset. Tale offset è fissato dal consumer, il quale può leggere i messaggi nell'ordine preferito.

Riferimenti bibliografici

[1] Apache™. Hadoop Official Website. https://hadoop.apache.org/, 2015. [Online; Ultimo accesso 10 Ottobre 2015].

[2] Fernando Kakugawa, Liria Sato, Mathias Brito. Architecture to integrating heterogeneous databases using grid computing. In *21st International Conference on Systems Engineering*, 2011.

[3] Leonardo Candela, Donatella Castelli, and Pasquale Pagano. gCube: a service-oriented application framework on the grid. *ERCIM News*, 72:48–49, 2008.

[4] Leonardo Candela, Donatella Castelli, and Pasquale Pagano. Managing big data through hybrid data infrastructures. *ERCIM News*, 89:37–38, 2012.

[5] Carmela Comito and Domenico Talia. GDIS: A service-based architecture for data integration on grids. In *On the Move to Meaningful Internet Systems 2004: OTM 2004 Workshops*, pages 88–98. Springer, 2004.

[6] J.C.S. Dos Anjos, G. Fedak, and C.F.R. Geyer. Bighybrid – a toolkit for simulating mapreduce in hybrid infrastructures. In *Computer Architecture and High Performance Computing Workshop (SBAC-PADW), 2014 International Symposium on*, pages 132–137, Oct 2014.

[7] D. Garlasu, V. Sandulescu, I. Halcu, G. Neculoiu, O. Grigoriu, M. Marinescu, and V. Marinescu. A big data implementation based on grid computing. In *Roedunet International Conference (RoEduNet), 2013 11th*, pages 1–4, Jan 2013.

[8] L. Hluchy, O. Habala, V. Tran, P. Krammer, and B. Simo. Using ADMIRE framework and language for data mining and integration in environmental application scenarios. In *Fuzzy Systems and Knowledge Discovery (FSKD), 2011 Eighth International Conference on*, volume 4, pages 2437–2441, July 2011.

[9] University of Chicago. Globus Toolkit Home Page. http://toolkit.globus.org/toolkit/, 2015. [Online; Ultimo accesso 10 Ottobre 2015].

[10] The University of Edinburgh. OGSA-DAI Official Website. http://www.ogsadai.org.uk/, 2015. [Online; Ultimo accesso 10 Ottobre 2015].

[11] Tapio Niemi, Marko Niinimaki, Vesa Sivunen. Integrating distributed heterogenous databases and distributed grid computing. In *ICEIS 5th International Conference on Enterprise Information Systems*, 2003.

[12] T. M. Sloan, A. Carter, P. J. Graham, D. Unwin, and I. Gregory. First data investigation on the grid: Firstdig.

[13] Sourceforge. FUSE : File system in userspace. http://fuse.sourceforge.net/, 2015. [Online; Ultimo accesso 10 Ottobre 2015].

[14] Yukako Tohsato, Takahiro Kosaka, Susumu Date, Shinji Shimojo, and Hideo Matsuda. H: Heterogeneous database federation using grid technology for drug discovery process. In *In Grid Computing in Life Science (LSGRID2004) Edited by: Konagaya A, Satou K.* Springer.

[15] Nicolas Viennot, Mathias Lécuyer, Jonathan Bell, Roxana Geambasu, and Jason Nieh. Synapse: A microservices architecture for heterogeneous-database web applications. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, pages 21:1–21:16, New York, NY, USA, 2015. ACM.

[16] Ǧorgi Kakaševski, Anastas Mišev, Boro Jakimovski. Heterogeneous distributed databases and distributed query processing in grid computing. *ICT Innovations Web Proceedings ISSN 1857-7288*, 2011.