Git hosting events analysis

Progetto finale del corso di Big Data
2017/2018

Emanuele Tusoni, Danilo Ponti
Gruppo Tusonti

Introduzione

Git nasce nel 2005 per opera di Linus Torvalds, inventore del sistema operativo Linux.
È lo standard *de facto* per quanto riguarda i sistemi di controllo di versione (VCS), strumenti il cui scopo è facilitare la gestione del codice delle applicazioni. In generale, aiuta a tenere traccia dell'evoluzione di un insieme di file e permette di navigare la cronologia delle modifiche; rende quindi più facile sperimentare con nuove funzionalità ed, in caso di errore, ripristinare uno stato precedente e consistente senza fatica. Inoltre agevola la condivisione e la modifica simultanea del codice fra più autori.

Github e Gitlab sono piattaforme web di hosting di repository git molto apprezzate ed utilizzate dagli sviluppatori per la gestione e la condivisione dei propri software. Grazie a queste due piattaforme, infatti, risulta molto semplice per degli sviluppatori collaborare ad un progetto, dal momento che permettono la modifica efficiente e sicura di un repository git remoto centrale e la sua gestione. I file caricati in questo repository, inoltre, possono essere scaricati e utilizzati da chiunque, incentivando così la libera circolazione del codice. Risulta evidente, dunque, che un progetto open source trovi il suo habitat ideale in due ambienti come questi.

Il nostro progetto punta all'analisi degli eventi che si verificano sulle principali piattaforme di hosting per repository git. Ci siamo concentrati sul flusso in streaming degli eventi che si verificano sui repository pubblici di github e gitlab, disponibili tramite delle API offerte da queste piattaforme proprio allo scopo di permetterne l'analisi.

La nostra principale ispirazione è stata un'applicazione web, github audio, che agli eventi di github faceva corrispondere

l'apparizione di un cerchio colorato (in una posizione casuale) e la riproduzione di un suono. Il colore, la dimensione del cerchio e il suono ad esso associato dipendevano dal tipo di evento ricevuto (commit, merge, pull request, apertura di issue,...).

Abbiamo cominciato a pensare, quindi, che potesse essere interessante prendere questo flusso di dati e, a differenza di quanto faceva l'applicazione sopra citata, realizzare un'architettura che permettesse di effettuare sia qualche tipo di operazione real time che allo stesso tempo un'analisi in batch degli eventi da vari punti di vista, rispondendo a diverse query sugli stessi.

Obbiettivi

Quello che ci siamo prefissati di realizzare è di fatto un'architettura in grado di gestire un grande flusso di dati e di operare sia operazioni in batch che real time.

Nel fare questo abbiamo cercato di sperimentare quante più tecnologie possibile, iniziando dall'ingestione dei dati, passando per le operazioni in real time e per la memorizzazione distribuita, per poi finire con l'analisi in batch dei dati memorizzati.

Abbiamo cercato anche di mettere a confronto tecnologie che svolgono lo stesso compito per comprendere i loro vantaggi, la facilità di utilizzo e di configurazione delle stesse, nonché la possibilità di integrazione con le altre tecnologie utilizzate nel progetto.

Così facendo abbiamo dato vita a delle architetture parallele che rappresentano alternative valide e all'avanguardia nell'ambito di un progetto Big Data.

Il sistema è stato progettato al fine di essere in grado di analizzare grandi quantità di dati da punti di vista differenti, permettendo query quali

ad esempio: quali sono i linguaggi più utilizzati in una certa finestra temporale? o quali sono i paesi dai quali si ricevono più eventi in un certo lasso di tempo?, quali sono i progetti più attivi sulla piattaforma, ecc.

Architettura

Per perseguire i nostri obiettivi abbiamo fatto riferimento ad uno specifico tipo di architettura chiamata lambda.

Un'architettura lambda ha lo scopo di gestire una grande quantità di dati derivante da una sorgente e supportare analisi sia in batch che real time.

Lo scopo principale di questa architettura è quello di permettere la realizzazione di queste operazioni offrendo ottime garanzie a livello di latenza, throughput e tolleranza ai guasti sfruttando un modello distribuito basato su 3 strati principali:

- **Batch layer**: è lo strato che si occupa dell'analisi in batch di un dataset molto grande sfruttando un sistema di processamento distribuito. Questo strato riceve in input l'intero dataset disponibile e produce in output delle viste che mostrano i risultati delle analisi effettuate. L'output di questo strato è poi memorizzato nel serving layer. Man mano che il dataset viene aggiornato le viste sono ricalcolate sull'intero dataset e sovrascrivono quelle vecchie.

- **Speed layer**: lo speed layer è responsabile del processamento dei dati in real time. In questo senso c'è un compromesso nell'aumento della latenza a discapito del throughput per fare in modo di fornire viste sui dati più recenti.

Questo strato quindi si occupa di colmare la lacuna del batch layer causata dall'incapacità di quest'ultimo di fornire in breve tempo risultati sui dati più recenti.

Evidentemente la forza di questo layer non è certamente la completezza, che piuttosto viene sacrificata in favore di una maggiore velocità con cui vengono processati i dati appena ricevuti.

➢ **Serving layer**: l'output derivante dai due strati precedenti fluisce in quest'ultimo strato, che si occupa di memorizzarlo e rispondere a delle query specifiche sottoposte dall'utente.

Il serving layer deve essere in grado di gestire una grande mole di dati proveniente dalla sorgente batch e quella real time in modo distribuito e scalabile. Per questo principalmente vengono utilizzate tecnologie NoSQL che, a differenza di quelle SQL (o relazionali), soddisfano a pieno tutti questi requisiti.

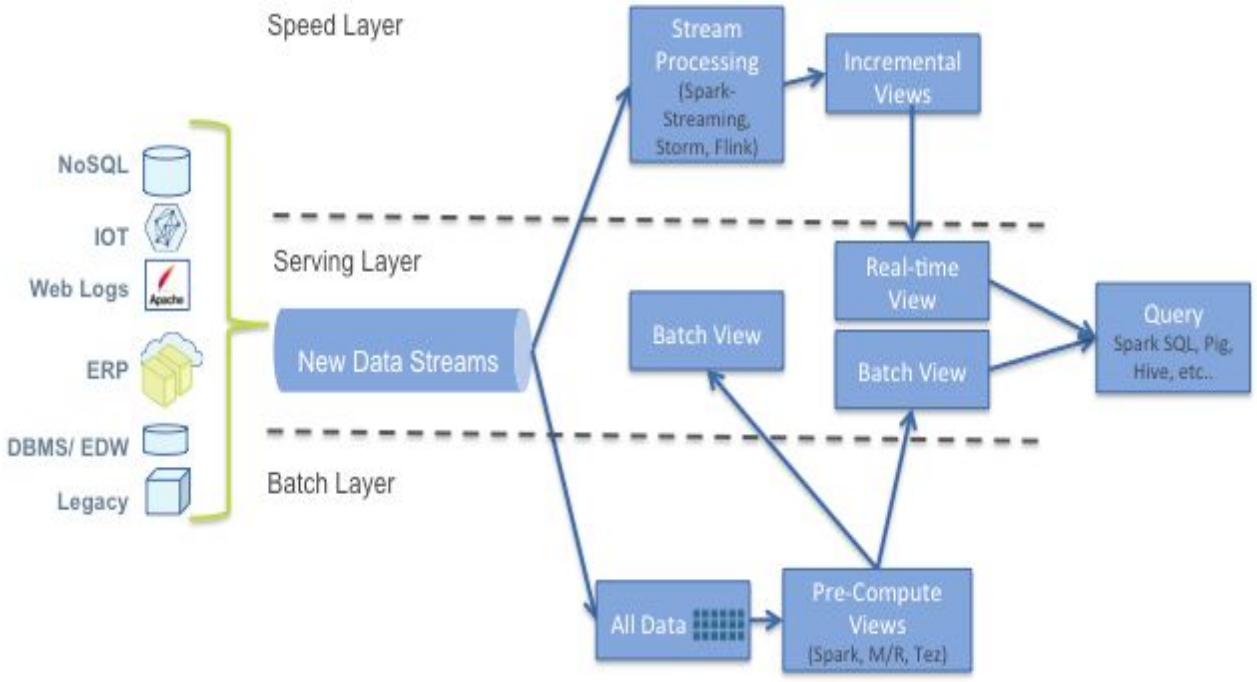

Figura 1. Esempio di architettura lambda

Le limitazioni di questa architettura risiedono nel fatto che sia il batch layer che lo speed layer richiedono una base di codice differente che deve essere manutenuta e sincronizzata per far sì che dia lo stesso risultato nei due differenti percorsi.

Come data absorber per l'ingestione dei dati la nostra scelta è ricaduta su Kafka, mentre per quanto riguarda l'engine per l'analisi in batch abbiamo optato per Spark. Queste due tecnologie rappresentano i punti fermi che non abbiamo voluto mettere in discussione per costruire l'architettura di base del nostro progetto.

Nello speed layer abbiamo voluto sperimentare due tecnologie alternative, entrambe considerate tra le più valide in circolazione nell'ambito di stream processing:

- ➢ Storm
- ➢ Spark streaming

Infine per completare il nostro serving layer ci siamo serviti e abbiamo sperimentato altre due tecnologie alternative fra loro nell'ambito dei database non relazionali, ovvero NoSQL:

- ➢ MongoDB
- ➢ Cassandra

In questa fase iniziale del progetto abbiamo preferito rimandare l'effettiva integrazione con le API di github e di gitlab (per semplificare lo sviluppo ed evitare la gestione di errori di rete, poco significativa ai fini del corso) preferendo invece optare per un generatore pseudo-casuale di json. Il software, sviluppato da Aces, Inc., è da noi controllato tramite due file di configurazione in cui specifichiamo la frequenza di invio dei json, il delay fra un json ed il successivo, la struttura delle chiavi e, per ogni chiave, una lista di valori ammissibili. Lo stream così artificialmente costruito è ragionevolmente simile a quello prodotto dalle due piattaforme di hosting.

Implementazione dell’architettura

L’immagine seguente illustra la connessione fra le varie componenti dell’architettura da noi realizzata :

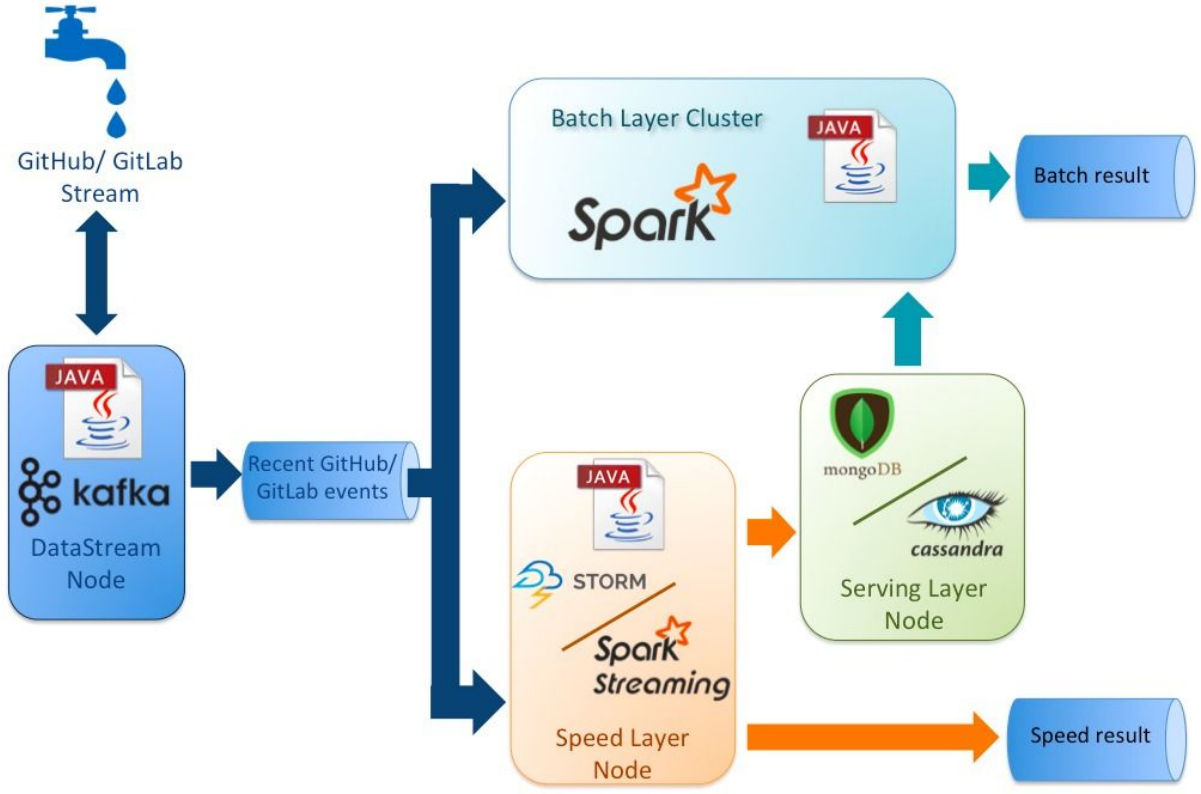

Figura 2. architettura lambda del progetto

Tecnologie

1. Data Stream

Kafka

Lo stream di eventi proveniente da github e gitlab viene incanalato nell'applicazione grazie all'utilizzo di Kafka.

Kafka è una piattaforma di stream processing open source sviluppata da Apache Software Foundation e creata appositamente per gestire flussi di dati in streaming in modo distribuito, scalabile e persistente, con performance a bassa latenza ed alta velocità. Tutt'ora è la tecnologia più utilizzata nello sviluppo di applicazioni per l'elaborazione in real time di stream di dati.

Kafka è basato su un paradigma publish/subscribe che permette di distinguere tre principali entità che giocano un ruolo chiave in questa piattaforma:

- **Producers**: i producers sono entità il cui compito è quello appunto di produrre e "pubblicare" dati (record) in uno o più topic tra quelli esistenti. Essenzialmente si occupano di rifornire il sistema dei dati che poi saranno processati, memorizzati ed analizzati.

- **Topics**: un topic rappresenta una categoria di record al quale è assegnato un nome univoco, ovvero un flusso di dati che devono essere trattati allo stesso modo. I producer pubblicano record all'interno di questi topic, che si occuperanno di far arrivare questi record a tutti gli iscritti al topic.

- **Consumers**: sono i destinatari finali dei record pubblicati nei topic, nonché i veri fruitori di questi ultimi. I consumatori sono organizzati in gruppi e ogni topic invia record solo ad uno dei nodi del gruppo iscritto a quel topic.

Kafka in realtà può offrire principalmente tre diversi servizi:

- ➢ **Messaging system**: a seconda di come si configurano i gruppi il sistema può funzionare come coda (con un solo gruppo) o come publish/subscribe (con molteplici gruppi), generalizzando i due modelli e riuscendo a coniugare i punti forti dei due sistemi rendendoli complementari.

- ➢ **Storage System**: i messaggi inviati ai topic possono essere memorizzati in modo distribuito e resistente ai guasti, in accordo con la natura stessa di Kafka, grazie ad un sistema di replicazione e di acknowledgement delle scritture, che viene inviato solo quando i messaggi vengono replicati su più server.

- ➢ **Stream Processing**: infine Kafka mette a disposizione una libreria per eseguire dei semplici processamenti sui dati che passano per la pipe di Kafka, rendendo così possibile, ad esempio, pre-processare i dati prima di inviarli ad un altro sistema per il processamento dei dati (ad esempio Storm).

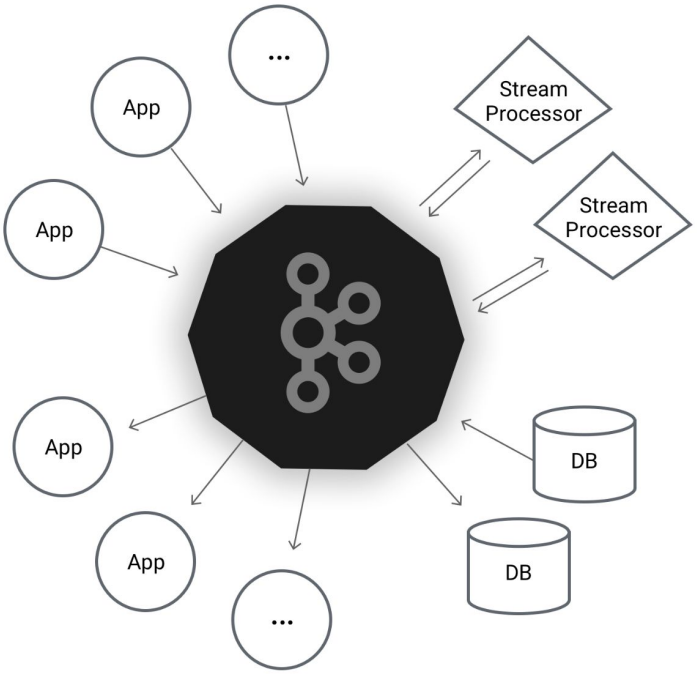

Figura 2. Servizi e topologia di una piattaforma realizzata con Kafka

Nello specifico abbiamo creato un topic “git-stream-input” nel quale vengono pubblicati eventi dai repository pubblici di github e gitlab appena questi si verificano. I record contenenti gli eventi pubblicati sono quindi dati in pasto ad una pipe iscritta come consumatore al topic di input, che dopo un piccolo processamento li inserisce in un altro topic di output chiamato “git-stream-output”.

Quest’ultimo topic memorizza e dispensa i record che gli vengono inviati e fa da presa per ogni altra tecnologia che si integra con Kafka (come ad esempio Storm, MongoDB, Spark Streaming, ecc...), che di fatto le fa da consumer. In questo modo i record possono essere raccolti dal topic e processati dallo speed layer, memorizzati dal serving layer ed infine analizzati dal batch layer.

2. Speed Layer

Per la realizzazione dello speed layer abbiamo sperimentato due tecnologie principali: Storm e Spark Streaming.

Storm

Storm è un sistema open source di processamento di dati real time distribuito che rende possibile gestire flussi di dati praticamente illimitati in modo veloce, scalabile e resistente ai guasti con la garanzia che tutti i dati vengano effettivamente processati.

Grazie a Storm è possibile effettuare operazioni complesse a piacere sui dati in streaming ripartizionando, se necessario, i flussi ad ogni fase del calcolo ed infine generare viste che ne mostrano i risultati.

Un'applicazione Storm è progettata con la topologia di un grafo diretto aciclico(DAG) con due tipologie di vertici:

- **Spout**: sono le sorgenti degli stream di dati.

- **Bolt**: sono le unità che operano le trasformazioni richieste al flusso di dati in entrata. Possono eseguire operazioni come filtraggi, aggregazioni, interazioni con il database ecc. Una pipeline può essere composta da più bolt, ognuno dei quali ad ogni passo opera sempre le stesse trasformazioni sul flusso di dati e manda in output i risultati ad un altro bolt.

Gli archi invece rappresentano il flusso di dati tra spout e bolt o tra due bolt.

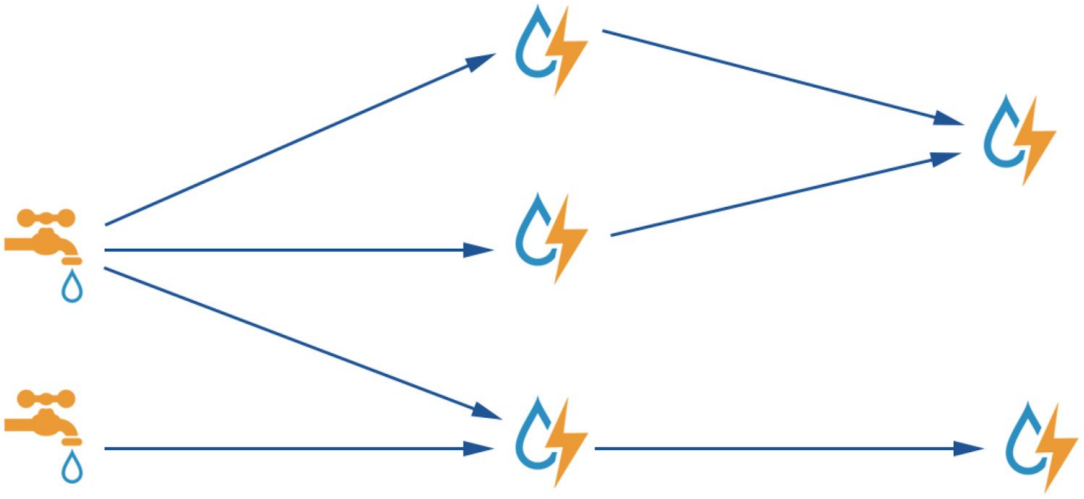

Figura 3. Rappresentazione della topologia(DAG) di un'applicazione Storm.

La nostra idea per un processamento real time è stata quella di notificare l'evento in corso di processamento a tutti i collaboratori del progetto al quale l'evento fa riferimento.

Nello specifico è stato necessario creare una topologia con uno spout e due bolt che lavorano come una catena di montaggio.

Lo spout prende i record derivanti dal topic "github-stream-output" comportandosi così come un consumer kafka e, in accordo con la topologia configurata, manda questi record al primo bolt.

Il "MailExtractorBolt" è il responsabile della prima fase di processamento, che consiste nell'estrazione degli indirizzi e-mail dei collaboratori ad un progetto dai record inviatigli dallo spout. Gli indirizzi estratti sono quindi inviati al secondo bolt per l'ultima fase di processamento.

L'invio vero e proprio della mail conclude la pipeline di Storm e il responsabile è il "MailSenderBolt", che riceve in input gli indirizzi e-mail e demanda ad una classe apposita ("MailSender") la preparazione di una sessione per l'invio della posta elettronica grazie alla libreria "javax.mail".

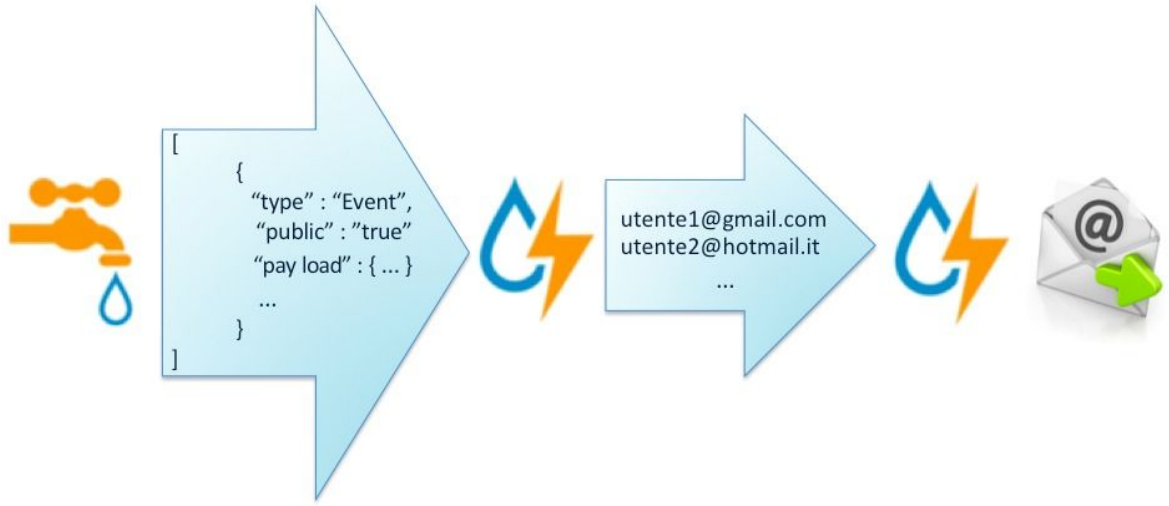

Figura 4. Pipeline per l'invio delle mail ai collaboratori in real time.

Spark Streaming

Spark Streaming, così come Storm, è un sistema che processa grandi flussi di dati in real time in modo da rendere questo processamento veloce, distribuito, scalabile e resistente ed è un'estensione del core delle API di Spark. Proprio grazie a questo le operazioni effettuate con questo engine sono molto simili alle operazioni in batch che possono essere fatte grazie al motore principale Spark.

Spark Streaming, infatti, ingerisce i dati in input e li discretizza in micro-batches (che sono anche degli RDD) da assegnare ai worker presenti nel cluster (con operazioni di load balancing e fails recovery), facendo così in modo da rendere possibile l'utilizzo di codice scritto per operazioni in batch, anche per operazioni real time.

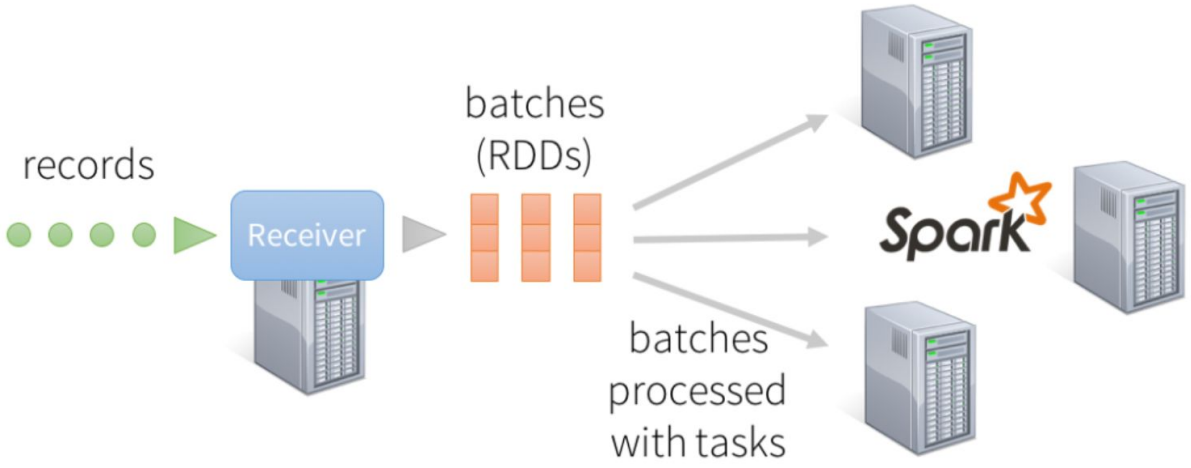

Figura 5. Processamento real-time con l'utilizzo di Spark Streaming.

Spark Streaming si rivela dunque l'ideale per chi vuole utilizzare un singolo framework (Spark) per soddisfare tutte le necessità di processamento delle proprie applicazioni, grazie anche alla sua capacità di rendere riutilizzabile codice già scritto per delle analisi batch con Spark.

3. Batch Layer

Spark

Apache Spark è un motore ed un framework open-source per il calcolo distribuito su cluster.

Opera su un qualsiasi filesystem distribuito compatibile con hdfs.

Il suo modello di esecuzione separa la gestione delle operazioni dalla gestione delle risorse dei nodi del cluster :un elemento, lo sparkContext interagisce con un gestore di cluster; questi alloca le risorse necessarie e invia il codice ai singoli esecutori. Spark si occupa di inviare gli effettivi task da eseguire agli esecutori e riceve i risultati delle computazioni.

È progettato per ottenere alte prestazioni ed essere fault tolerant, sia per analisi batch che per analisi su dati in streaming.

Offre un vasto numero di operazioni per la trasformazione, l'analisi e la persistenza dei dati che lo rendono più flessibile rispetto al modello di mapReduce.

Uno dei principali punti di distacco da mapReduce offerto da questa nuova tecnologia è l'uso intensivo di memorie ram. Mentre in un processamento hadoop mapReduce, ad ogni passo i dati sono necessariamente salvati sul file system distribuito (quindi su uno o più dischi fisici), in spark i risultati dei processamenti intermedi vengono memorizzati in ram, all'interno di strutture dati definite RDD (Resilient Distributed Dataset); come indica il nome si tratta di collezioni immutabili di oggetti distribuite in grado di recuperare porzioni danneggiate o mancanti dalle altre copie esistenti nel cluster, assicurando un buon livello di fault tolerance.

Sfruttando i più brevi tempi di accesso e scrittura delle ram si riescono ad ottenere velocità di elaborazione fra le 10 e 100 volte maggiori rispetto a quelle di mapReduce.

4. Serving Layer

MongoDB

MongoDB è un database non-relazionale afferente alla tipologia document database.

È il database NoSQL più usato al mondo secondo db-engines.com.

I document database non memorizzano i dati in tabelle e non gestiscono le loro relazioni; i document database memorizzano coppie chiave ->

documento, dove per documento si intende il contenuto di un file in un qualche formato (xml, json, testo,...).

Il document database, a differenza dei database key-value, è inoltre in grado di navigare nei campi dei documenti e supporta quindi interrogazioni su questi.

MongoDb accetta documenti in formato BSON, un superset del json che include informazioni sui tipi, su array associativi ed altre semplici strutture dati.

A differenza di altri document database, fornisce nativamente funzionalità di aggregazione sui dati e di analisi in streaming :

- L'aggregation pipeline è un framework per l'aggregazione dei dati basato sul concetti di "pipe" : i documenti passano lungo una serie di pipe che trasformano i loro dati attraverso filtraggi ed aggregazioni. È concettualmente simile al framework map-reduce sebben rispetto alle sue implementazioni (ad esempio quella di hadoop) ha delle limitazioni in termini di memoria e dimensione dei documenti. Ad esempio : ogni documento derivante da un'operazione di aggregazione non può superare i 16 Mb; durante il processamento in una pipe la memoria ram disponibile non può superare i 100 Mb.

- I change streams permettono alle applicazioni di accedere e rilevare in real-time i cambiamenti effettuati su una collezione o su un database.

Per applicazioni che non superino le limitazioni sopra indicate, soprattutto in una fase iniziale di prototipazione, MongoDB può essere quindi una valida alternativa a framework di map-reduce come hadoop o a processamenti più complessi come quelli forniti da spark.

Anche per questa tecnologia abbiamo creato un consumer kafka per integrarla all'infrastruttura generale. Il connettore infatti non fa altro che connettersi al topic del flusso di eventi in output e creare una connessione al database. Dopo un piccolo preprocessamento i dati vengono memorizzati su mongoDB sotto forma di documenti pronti per essere analizzati dal Batch Layer.

L'integrazione fra mongoDB e spark è stata possibile grazie al connettore mongo db connector for apache spark .

Cassandra

Cassandra è un database non-relazionale afferente alla tipologia column-family store database.

A differenza dei tradizionali database relazionali o di quelli document, i database column-store memorizzano e

permettono di accedere ai dati usando una struttura a due livelli : i dati veri e propri vengono divisi in colonne mentre queste sono salvate in aggregati; gli aggregati sono accessibili tramite chiavi. Quindi, per accedere ad un dato :

- ➢ prima si accede all'aggregato tramite la sua chiave
- ➢ poi si sceglie un dato presente nella colonna indicata

In cassandra il contenitore più esterno si chiama keyspace. Un keyspace è caratterizzato da un fattore di replicazione e da una lista di column-families. Ogni keyspace ha almeno una column-family ad esso associata.

Considerazioni finali

Le diverse implementazioni dell'architettura ottenute usando di volta in volta diversi strumenti (storm o spark streaming, database document oriented o column based, ...) ci hanno permesso di mettere mano anche alle tecnologie illustrate nella seconda parte del corso e ci ha permesso di valutare il loro livello di integrazione coi sistemi di processamento ed analisi visti invece nella prima parte. Il tutto cercando di analizzare un dominio, quello delle piattaforme di hosting per repository git, di nostro effettivo interesse.

Le nostre considerazioni più che il risultato di un qualche tipo di benchmark, sono principalmente valutazioni qualitative sull'esperienza pratica da noi riportata.

L'unico parametro su cui potevamo effettivamente operare era quello della frequenza di generazione dei json pseudo-casuali. Al variare di questo parametro non ci sono stati problemi e non si sono riscontrate differenze in termini di prestazioni fra i diversi tipi di database.

Le differenze più significative hanno riguardato la parte di installazione e configurazione del software :

- Per l'utilizzo di Kafka è stato necessario installare lanciare un server zookeeper ma in generale la sua configurazione si è dimostrata molto semplice e veloce, così come l'utilizzo delle sue API. La caratteristica fondamentale di Kafka è che la sua architettura, semplice ma efficace, lo rende capace di integrarsi con moltissime altre tecnologie in modo agevole, aggiungendo, in pratica, solo dei consumatori iscritti ad uno dei suoi topic.

- Anche l'utilizzo di Storm, come quello di Kafka, si è rivelato intuitivo ed efficace: la creazione e la configurazione dello spout e dei bolt è semplice così come quella dell'intera topologia del grafo, permettendo la creazione di una pipeline in cui ogni nodo ha una sua responsabilità precisa e circostanziata.

- Nella nostra esperienza Spark Streaming si è rivelato più difficile da utilizzare e configurare rispetto a Storm, che invece è stato sicuramente più immediato, anche se ha richiesto la scrittura di più codice per eseguire gli stessi processamenti. Nello specifico Spark ha richiesto la configurazione di un considerevole numero di parametri ed in generale i benefici derivanti dagli RDD si pagano con un processamento meno intuitivo e fluido rispetto alla pipe di Storm. Nonostante questo la scelta di Spark Streaming si sarebbe potuta rivelare quella più giusta vista anche la presenza di Spark nel batch layer nella nostra architettura.

- mongoDB è stato ragionevolmente semplice da installare e configurare. L'applicazione per l'interazione tramite shell, *mongo,* si è rivelata intuitiva; la struttura delle query, sebbene non perfettamente identica, è ragionevolmente simile a quella poi disponibile tramite le API in java.
  Infine, anche il connettore fra mongo e spark, le cui funzionalità non sono affatto banali, ha richiesto una configurazione minima.

- Cassandra si è rivelato molto meno intuitivo ed anche solo l'avvio dell'interfaccia shell, *cqlsh,* ha richiesto la configurazione di molteplici campi in *cassandra.yaml* e la documentazione si è rivelata meno accessibile rispetto a quella di mongo.

  Di contro la forma in cui si fanno le interrogazioni è assolutamente identica tramite shell e tramite API java : entrambe accettano in input una stringa simil-sql (in realtà cql, *cassandra query language*).

  Per un progetto simile al nostro, in cui si hanno in input documenti strutturati e in cui le interrogazioni su questi documenti non sono state ancora ben definite e sono soggette ad una rapida evoluzione, è preferibile (e l'abbiamo effettivamente verificato)

usare un db orientato ai documenti rispetto ad uno orientato alle colonne.

Non abbiamo riscontrato particolari vantaggi con la seconda tipologia ed anzi, la necessità di organizzare il json in colonne ha portato ad un ulteriore fase di preparazione prima dell'effettiva scrittura in cassandra.Ci riteniamo quindi più soddisfatti dall'uso di mongoDB.

Sviluppi futuri

Contiamo di continuare a sviluppare il sistema nell'ottica di integrarlo col progetto finale del corso di visualizzazione delle informazioni. I principali sviluppi futuri che prevediamo sono :

1. Miglior controllo sulle query : l'utente dovrà essere in grado di scegliere quali query visionare (ora di default il client mostra i risultati di tutte le query che abbiamo implementato)

2. Aggiungere ulteriori query : al momento rispondiamo solo a "quali sono i 3 linguaggi più usati negli ultimi tot minuti?" e "quali sono le nazioni i cui cittadini stanno generando più eventi sulle due piattaforme d'interesse negli ultimi tot minuti?". Sarà importante implementare la selezione della finestra temporale d'interesse

3. Implementare un sistema di visualizzazione : permetterà in modo semplice per l'utente di comprendere quali siano le funzionalità offerte dal nostro sistema. Immaginiamo che di default verrà visualizzato un planisfero sui cui i singoli eventi rilevati saranno rappresentati come piccoli cerchi opachi all'interno dei confini del paese di origine dell'utente che ha causato l'evento; in basso, in

fondo alla pagina, visualizzeremo invece un'istogramma degli n linguaggi più usati nella finestra temporale selezionata.