Tyr: Tool di analisi real-time di una rete sociale sfruttando Apache Storm

Gabriele Belli
Università di Roma Tor Vergata
belligabriele91@gmail.com

Giuseppe Chiapparo
Università di Roma Tor Vergata
giuseppe.chiapparo.91@gmail.com

Federico Di Domenicantonio
Università di Roma Tor Vergata
fededido91@hotmail.it

***Sommario*—In questo articolo verrà presentata una soluzione relativa alla DEBS 2016 Grand Challenge che sfrutta Apache Storm, un sistema libero e open source per la computazione real-time e distribuita. Inoltre, una sezione del paper sarà dedicata alla descrizione dell'applicazione open source *teacup-storm*, che è stata sviluppata per rendere più semplice il deploy di Apache Storm su EC2.**

Per ottenere un processamento veloce di una grande quantità di flussi di dati si è realizzata un'architettura che sfrutta la computazione parallela offerta da Apache Storm. Inoltre, per ottenere prestazioni ancora migliori si è preferito sviluppare un'architettura efficiente dal punto di vista dell'utilizzo delle memoria, limitando al minimo le interazioni con memorie esterne ai nodi, come ad esempio Redis.

La gran parte della computazione necessaria a risolvere la prima query riguarda il calcolo dello score di un post e la generazione delle classifiche parziali dei post più attivi. Si è dunque deciso di sviluppare questa logica all'interno di un nodo, la cui computazione potesse essere distribuita. Questi nodi distribuiti generano le classifiche parziali, che sono poi rielaborate da un nodo unico in grado di calcolare la classifica dei post con più alto score.

Anche per quanto riguarda la seconda query si è cercato di concentrare la computazione all'interno di un nodo che potesse essere distribuito. Infatti, all'interno di tale nodo si è riuscito ad implementare la logica principale della query, che sfrutta l'algoritmo di Bron-Kerbosch[1] per identificare le più grandi comunità attive su di un determinato topic. In questa seconda query si è reso necessario l'utilizzo di Redis per poter memorizzare il grafo delle amicizie. Tuttavia, il nostro approccio ha cercato di evitare il più possibile l'interazione con Redis e sfruttare maggiormente un sistema di caching in-memory all'interno dei diversi nodi della topologia.

I risultati ottenuti dai test effettuati hanno mostrato dei buoni risultati per la query uno, e tempi di processamento elevati per la seconda query a causa del tempo impiegato per effettuare le chiamate a Redis. Un valore aggiunto è in ogni caso rappresentato da *teacup-storm*, in quanto tale tool potrà semplificare di molto lo sviluppo di progetti futuri che prevedono il deploy di Storm su Amazon EC2.

I. Introduzione

L'analisi delle reti sociali è una moderna metodologia di indagine, che trova applicazione nelle scienze come la sociologia, l'antropologia, la psicologia e l'economia. Con l'avvento dei nuovi social network (come Facebook e Twitter), tale metodologia è divenuta un argomento cardine anche per quanto riguardo il mondo della Computer Science.

Non a caso la sesta edizione dell'ACM DEBS Grand Challenge ha avuto come obiettivo proprio quello di estrapolare informazioni in real-time da un ampio volume di dati prodotti da un social network.

Con la crescente diffusione dei social network le comunità scientifiche hanno iniziato ad indagare sul modo migliore per poter estrarre informazioni di valore dalla grande quantità di dati prodotta ogni giorno. Una delle attività principali da eseguire su di un social network è l'identificazione delle comunità che meglio riescono a stimolare l'interesse e l'interazione tra diversi utenti. Spesso le aziende sono interessate a conoscere queste comunità per poter condurre delle campagne pubblicitarie efficaci.

Queste ultime sono sempre alla ricerca dei topic più in voga e vogliono conoscere le reazioni degli utenti, così da poter prendere le decisioni basandosi sui dati. È per tale motivo che è molto importante poter studiare i cambiamenti in un social network in tempo reale, seppur questo rappresenta ancora oggi una grande sfida. Le odierne difficoltà incontrate nell'analisi di una rete sociale sono dovute:

- Alla grande quantità di dati che viene prodotta da una rete sociale.
- Alla dinamicità intrinseca di un social network, che è in continua evoluzione.
- Alle complessità che caratterizzano i grafi, che sono le strutture dati usate per rappresentare i social network.

La soluzione alle difficoltà sopracitate è rappresentata dall'utilizzo di algoritmi efficienti e tecnologie avanzate. Oggi le tecnologie che consentono di poter effettuare un'analisi real-time sono strettamente legate al mondo del cloud computing e allo stream processing.

Sulla base di queste motivazioni la sesta edizione della DEBS Grand Challenge si focalizza proprio sui problemi legati alla creazione di applicazioni in grado di fornire un'analisi real-time di grafi sociali. In particolar modo la sfida riguarda lo sviluppo di un'applicazione in grado di elaborare i dati provenienti da diversi stream rappresentanti eventi di un tipico social network. Nello specifico vengono trattati eventi come amicizie, post, commenti e like, così da poter:

- Identificare i post che al momento sono più attivi nel social network.
- Scoprire le più ampie comunità che sono attualmente coinvolte in un determinato argomento.

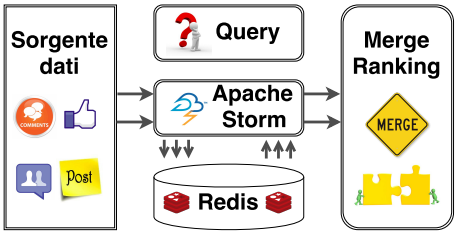

Figura 1. Architettura ad alto livello usata per la risoluzione delle query.

Per mostrare il lavoro svolto per la risoluzione della challenge, si è deciso di suddividere il paper in diverse sezioni. Una prima parte in cui viene descritta la Grand Challenge e le soluzioni alle due query, una seconda parte in cui si descriverà l'applicazione teacup-storm che si è sviluppata per poter effettuare il deploy di Storm su AWS. Infine, verranno mostrati i risultati ottenuti dagli esperimenti e le conclusioni a cui siamo giunti a seguito di questo lavoro.

II. Soluzione alla Grand Challenge

La Debs Grand Challenge 2016 ha come obiettivo quello di analizzare una rete sociale dinamica rappresentata da un grafo.

In particolar modo il problema riguarda l'identificazione dei post che al momento sono più attivi nel social network, e delle più ampie comunità che sono attualmente coinvolte in un topic. Le queries richiedono l'analisi continua di un grafo dinamico attraverso stream multipli che riflettono il modo in cui il grafo evolve.

Per rappresentare l'evoluzione di un social network i dati di input sono organizzati in quattro stream separati: le amicizie, i post, i commenti e i like.

Per riuscire a trovare una buona soluzione, inizialmente c'è stato un grande lavoro di studio teorico del problema. Una volta che si è compreso il problema e si è raggiunta una soluzione si sono dovute fare delle scelte riguardo le tecnologie da utilizzare.

La prima scelta che abbiamo effettuato è sul tipo di sistema di calcolo parallelo da usare. Si sono confrontati Amazon Kinesis, Apache Spark e Apache Storm. La nostra scelta è ricaduta proprio su Storm, in quanto è un sistema di calcolo distribuito, a tolleranza di errore e open source che consente di elaborare i dati in tempo reale. Infatti, Storm si concentra proprio sul processamento a stream, caratterizzato dalla computazione di tipo task-parallel. Lo si è preferito ad Apache Spark, in quanto più responsive e garantisce una latenza molto bassa (sotto al secondo). Mentre per quanto riguarda Amazon Kinesis, nonostante sia già presente su AWS, si è preferito Storm per il fatto che è open source e gratuito.

Studiando la seconda richiesta della Challenge, ci si è resi conto fin da subito della necessità di utilizzare un Database per memorizzare il grafo delle amicizie. In questo caso si è scelto Redis, in quanto è un database key-value open source residente in memoria con persistenza facoltativa.

Dalle scelte fatte durante la fase di progettazione, e per il modo in cui sono state definite le due richieste della challenge, si è ottenuta una soluzione ad alto livello che ben si può esprimere con la figura 1. La soluzione mostra i seguenti elementi:

- Una sorgente dati, che in entrambe le query è stata simulata da uno spout che legge da diversi file e che emette in ordine i diversi eventi da processare.
- Apache storm che consente di elaborare i dati parallelamente.
- Redis, che viene usato nella seconda query per memorizzare il grafo delle amicizie.
- Un nodo di *merge* che confronta i risultati parziali ottenuti dai nodi precedenti ed ottiene i risultati globali.

A. Query uno

L'obiettivo della prima query è proprio quello di computare i 3 post attivi con più alto score, producendo un risultato aggiornato ogni volta che avviene un cambiamento tra i primi tre valori. Lo score totale di un post attivo P è calcolato come la somma del proprio score (in quanto post) più lo score di tutti i suoi commenti. L'arrivo di un post o di un commento apporta l'aumento dello score di 10 unità, per poi decrescere di 1 ogni ventiquattro ore. La scadenza di un post è un evento che si verifica quando si ha un post che raggiunge un valore di punteggio pari a 0.

Un commento apporta un incremento al post P, sia se si tratta di un commento diretto a P, sia se è un commento ad un commento di P. Tale questione è stata oggetto di studio, in quanto nel dataset fornito, un commento di commento non ha il riferimento diretto al post. Per tale motivo nella query uno si è resa necessaria l'implementazione di un *Comment Mapper*. Quest'ultimo riesce a risalire al post che è stato commentato, in quanto memorizza un hash map che collega l'id di un commento al post relativo.

Ne segue che il compito del *Comment Mapper* è quello di inoltrare senza alcuna modifica i post, e mappare i commenti al relativo post cui fanno riferimento. Il motivo per cui anche i post sono inoltrati al *Comment Mapper* è che in questo modo si è sicuri che la temporalità degli eventi ricevuti dai *Ranking Bolt* è corretta.

Se il *Comment Mapper* dovesse mantenere in memoria tutti i riferimenti dei commenti al relativo post, la memoria sarebbe destinata a crescere in maniera indefinita. Fortunatamente all'interno di questa query siamo interessati solamente ai post attivi, ossia a quei post che non hanno mai avuto score pari a 0. Per tale motivo si è reso necessario un sistema di *feedback* in grado di comunicare al *Comment Mapper* la scadenza di un post. Questo meccanismo è stato implementato attraverso la generazione del particolare stream di output *mapper stream* generato dal *Ranking bolt* e ricevuto dal *Comment Mapper*. In questo stream vengono inoltrati i post che scadono, facendo in modo di liberare la memoria occupata dal *Comment Mapper*.

Gran parte della vera computazione della query è stata effettuata all'interno dei *Ranking Bolt*. Il compito di quest'ultimo nodo è di analizzare l'arrivo di post e commenti e creare

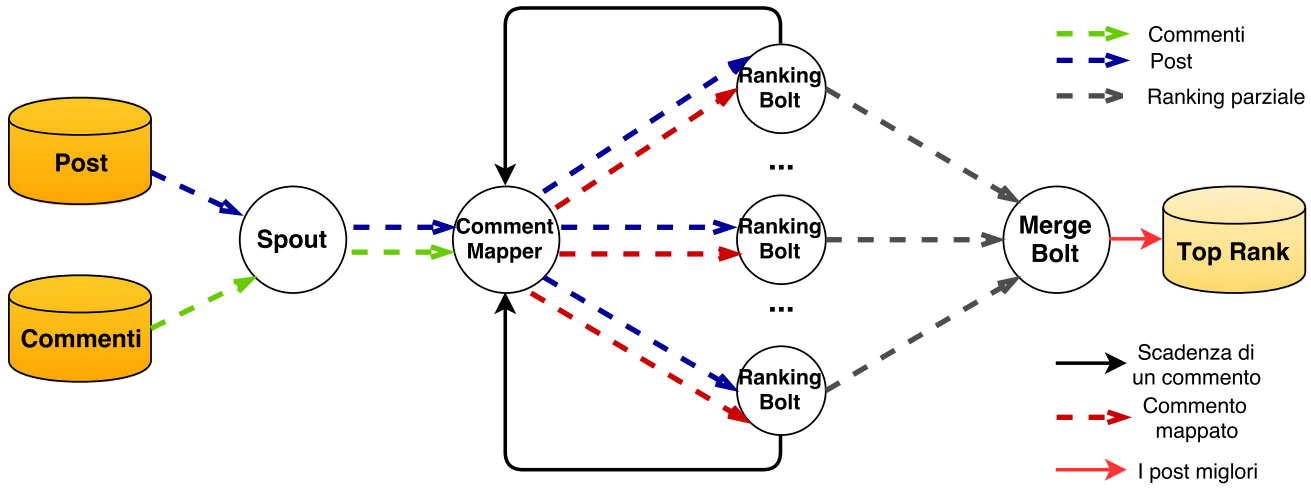

Figura 2. Topologia query 1.

delle classifiche ordinate di post in base allo score. Come anticipato in precedenza, il *Ranking Bolt* è un nodo la cui computazione può essere distribuita, e a tal proposito si è costruita la topologia in modo tale da far analizzare ad ogni singolo bolt, una porzione di post e tutti i commenti relativi a tali post. Ciò è stato possibile tramite l'implementazione del *Field Grouping* messo a disposizione da Storm ed eseguito in base all'id del post.

L'elaborazione eseguita dai *Ranking Bolt* si occupa di:

- Gestire l'arrivo di un nuovo post o commento.
- Controllare se un post è scaduto, e in caso inoltrare al *Comment Mapper* l'avvenuta cancellazione.
- Monitorare i cambiamenti tra le prime tre posizioni, ed aggiornare il *Merge Bolt* di ogni modifica della classifica parziale.

Il tempo logico della query avanza sulla base dei timestamp delle tuple di input e non in base all'orologio di sistema. Questo fattore è stato determinante per la progettazione del *Merge Bolt*. Infatti, osservando la topologia in figura 2 si può evincere che gli aggiornamenti prodotti dai *Ranking Bolt* non giungono necessariamente in ordine al *Merge Bolt*. Per tale motivo il *Merge Bolt* memorizza i diversi aggiornamenti che provengono dai vari bolt, e quando per ogni bolt è presente almeno un aggiornamento si processa quello che ha il timestamp più vecchio.

B. Query due

L'obiettivo della seconda query è quello di identificare i k commenti creati negli ultimi d secondi con il range più alto; dove con range indichiamo la dimensione della più grande componente connessa di utenti che sono amici e che hanno messo mi piace ad uno stesso commento. I valori di k e di d sono forniti come parametro.

La topologia della nostra soluzione è rappresentata nella figura 3. Lo Spout ha un duplice ruolo quello di leggere dalle sorgenti dati: i like, i commenti e le amicizie ed ordinare temporalmente gli eventi prima di inviarli.

Lo stream prodotto dallo *Spout* e contenente like e commenti viene diviso in un numero di partizioni e assegnato tra i diversi *Ranking Bolt* che sono interessati allo stream.

Grazie ad Apache Storm è possibile definire come queste partizioni di tuple devono essere assegnate configurando nella topologia lo *Stream Grouping*.

Il *Fields Grouping*, ovvero lo Stream Grouping da noi scelto, permette di controllare in che modo le tuple sono inviate ai task di un Bolt, tenenendo in considerazione uno o più campi della tupla stessa. Per una certa combinazione di campi di un dato set di valori, Apache Storm garantisce che sia sempre inviata allo stesso Bolt.

E' stato deciso, per minimizzare l'invio non necessario di dati e i tempi di processamento, che un commento ed i suoi corrispettivi like vengano processati dallo stesso Bolt.

Lo stream contenente le amicizie viene consumato sia dal *Friendship Bolt* il quale ha il compito di memorizzare tutte le amicizie in Redis, che dai *Ranking Bolt*, infatti ciascuno di questi, deve ricevere tutte le amicizie in quanto potrebbero far variare il valore del range calcolato fino a quel momento.

Il *Ranking Bolt* rappresenta il cuore della soluzione da noi proposta, ciascuno di questi riceve un evento che può essere un'amicizia, un commento o un like e restituisce quelli che secondo i dati ricevuti sono i k commenti con range più grande negli ultimi d secondi; sarà poi il *Merge Bolt* a calcolare la classifica assoluta tra tutti i valori ottenuti dai *Ranking Bolt*.

La soluzione proposta cerca di minimizzare le richieste a Redis per ottimizzare le prestazioni del sistema stesso; vediamo con maggiore dettaglio quanto appena detto. Il comportamento del *Ranking Bolt* cambia a seconda dell'evento ricevuto:

- All'arrivo di un like, a partire dall'utente che ha generato l'evento viene richiesto a Redis di restituire i suoi amici e i rispettivi amici di quest'ultimi; è da questo insieme che grazie all'algoritmo di *Bron-Kerbosch* viene calcolata la massima clique. Successivamente vengono aggiornate tutte le informazioni del commento che ha ricevuto il like

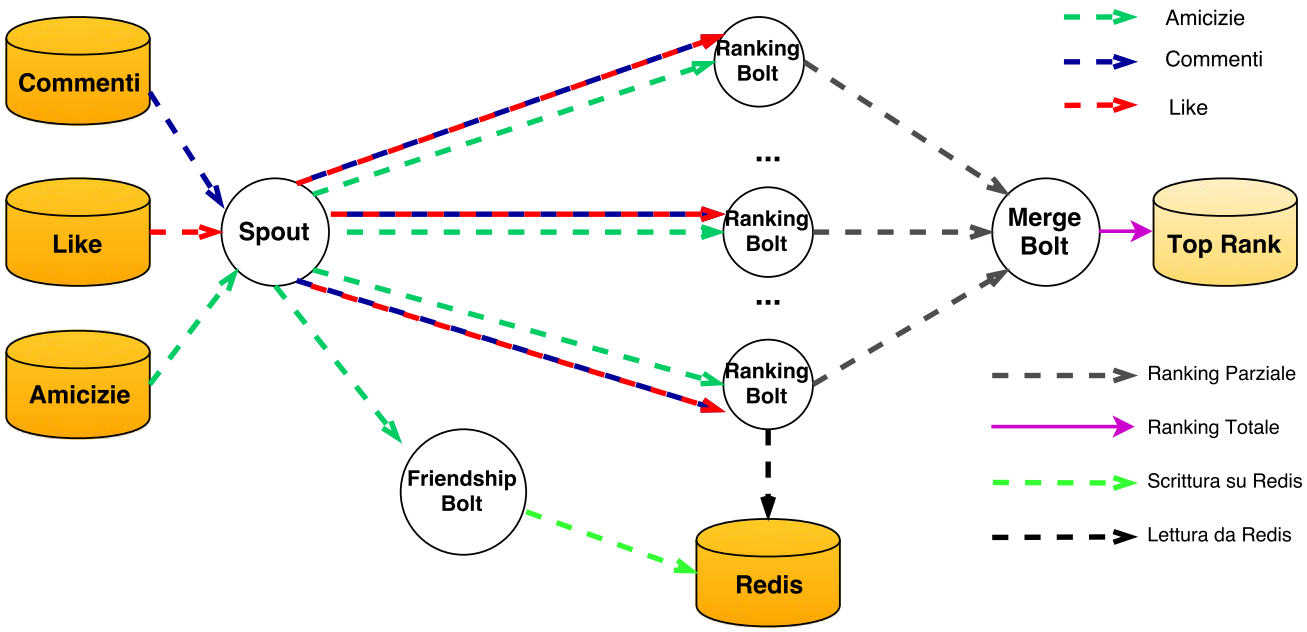

Figura 3. Topologia query 2.

e le clique degli utenti che già avevano lasciato un like al commento.

- Per ogni commento, viene mantenuta per ogni utente che ha messo un like una clique originata da quella precedentemente descritta, ma a partire da questa vengono rimossi gli utenti che non hanno generato un evento sul commento che si sta considerando.
- All'arrivo di un'amicizia, invece, vengono calcolati prima tutti gli utenti per cui l'arrivo dell'amicizia che si sta processando può stravolgere la massima clique. Questi utenti sono quelli per cui nelle massime clique memorizzate c'è almento uno dei due utenti che stanno stringendo l'amicizia. Successivamente, quindi viene ricalcolata la massima clique e per ogni utente che fa parte di questa nuova massima clique viene eseguito un ricalcolo della clique descritta nel punto precedente, ovvero quella la cui dimensione mi darà il range associato al commento.

Per ogni evento viene inoltre valutato se vi sono commenti scaduti e in caso affermativo vengono rimossi. Un evento può anche generare un cambiamento delle prime k posizioni della classifica dei commenti con range maggiore, qualora questo si verificasse viene inoltrato al MergeBolt la modifica che sarà gestitia da quest'ultimo.

Ricordiamo, inoltre, come specificato dalla query in caso di parità tra due commenti viene scelto quello con ordine lessicografico inferiore.

III. Test del sistema

I test che sono stati effettuati per verificare la corretta implementazione delle due query sono stati eseguiti su di un ambiente non uniforme. Infatti, si è preferito effettuare test su cluster differenti per poterne valutare le differenze delle prestazioni.

Per quanto riguarda la query 1 si è riuscito ad effettuare dei test con successo anche su istanze small. Lo small dataset fornitoci dalla query è stato processato senza errori di sovraccarico del sistema pur usando due supervisors su istanze small e con 8 worker in esecuzione. Per tale motivo si è deciso di effettuare dei test al variare del numero di supervisors utilizzati, pur eseguendo questi ultimi sempre su delle istanze di tipo `t2.small`.

Invece, dai test effettuati sulla query 2 si è visto come il sistema non riuscisse a completare l'analisi dello small dataset se eseguito su istanze di tipo `t2.small`. Infatti, riprendendo la documentazione[2] di Redis è consigliato l'utilizzo di istanze `m3.large`. In questo modo infatti siamo riusciti ad evitare gli errori di sovraccarico del sistema precedentemente avuti.

Su di entrambe le query ci si è soffermati a valutare:

- La capacità dei singoli nodi.
- La latenza di esecuzione dei singoli nodi, ossia il tempo medio speso da una tupla nel metodo *execute*.
- La latenza complessiva di una tupla elaborata.

A. Test query1

I test della query uno sono stati effettuati su istanze di tipo `t2.small`. I test eseguiti mostrano i dati raccolti dopo l'esecuzione del 10% dei dati provenienti dal dataset grande della Challenge. In particolar modo si sono confrontati i seguenti casi:

1) TEST 1 = Quattro Supervisor su quattro istanze `t2.small`, che gestiscono **un Ranking Bolt**, uno Spout, un Comment Mapper e un Merge Bolt.
2) TEST 2 = Due Supervisor su due istanze `t2.small`, che gestiscono cinque Ranking Bolt, uno Spout, un Comment Mapper e un Merge Bolt.

3) TEST 3 = Otto Supervisor su otto istanze `t2.small`, che gestiscono cinque Ranking Bolt, uno Spout, un Comment Mapper e un Merge Bolt.

I risultati riportati nella tabella I mostrano un andamento delle prestioni in cui è evidente che i nodi sono sottoutilizzati. In particolar modo si può notare come i nodi che principalmente hanno un carico maggiore sono il Comment Mapper e il Ranking Bolt. Il comment mapper, infatti, deve processare tutte le tuple in arrivo e la gestione dei commenti al crescere del numero di tuple impiega un tempo maggiore.

Per quanto riguarda il Ranking Bolt, si può evincere dai dati che maggiore è il parallelismo e minore risulta la capacità dei singoli Ranking Bolt. Questo risultato è in linea con le nostre aspettative. Allo stesso tempo non deve sorprendere il fatto che all'aumentare del numero di macchine istanziate nel cluster, cresce l'utilizzazione del Comment Mapper, il motivo di questa crescita, è che i Ranking Bolt analizzano i dati più velocemente perché istanziati su più macchine, mentre per il comment Mapper il parallelismo non modifica la propria condizione.

Inoltre, dalle analisi fatte sulla latenza complessiva del sistema, si è osservato che la latenza rimane sostanzialmente invariata al crescere del numero di tuple che si sono elaborate.

B. Test query2

Nei test della query 2 si è utilizzato come parametri di d e k rispettivamente 259200s e 3; inoltre questi si sono eseguiti su un'istanza M3, la quale offre un equilibrio di risorse di calcolo, memoria e di rete ed è un'ottima scelta per molte applicazioni.Tra le sue caratteristiche principali vi sono:

- Processori ad alta frequenza Intel Xeon E5-2670 v2 (Ivy Bridge).
- Storage di istanze basato su SSD per elevate prestazioni I/O.
- Equilibrio di risorse di calcolo, memoria e di rete.

I principali casi d'uso di questa famiglia di istante sono: database di piccole e medie dimensioni, attività di elaborazione dati che richiedono più memoria, caching di flotte ed esecuzione di server back-end per SAP, Microsoft SharePoint, cluster computing ed altre applicazioni per grandi imprese. Come si evince dalla tabella 4, sono stati eseguiti due tipi di test:

- TEST 1 = 4 Supervisor ognuno con due vCPU, 1 Spout, 1 Friendship Bolt, 1 Merge Bolt e 5 Ranking Bolt.
- TEST 2 = 8 Supervisor ognuno con due vCPU, 1 Spout, 1 Friendship Bolt, 1 Merge Bolt e 13 Ranking. Bolt

Come immaginato in partenza, anche i dati ottenuti dai test hanno confermato che la latenza per il Friendship Bolt tende a rimanere costante a differenza di quella del Ranking Bolt (cfr. figura 4). Per quest'ultimo, infatti, la latenza tende ad aumentare al crescere del numero di tuple analizzate, questo perché tendono ad aumentare le chiamate a Redis per ottenere le nuove massime clique. Ovviamente aumentando la latenza per il Ranking Bolt tende ad aumentare anche la latenza complessiva del sistema.

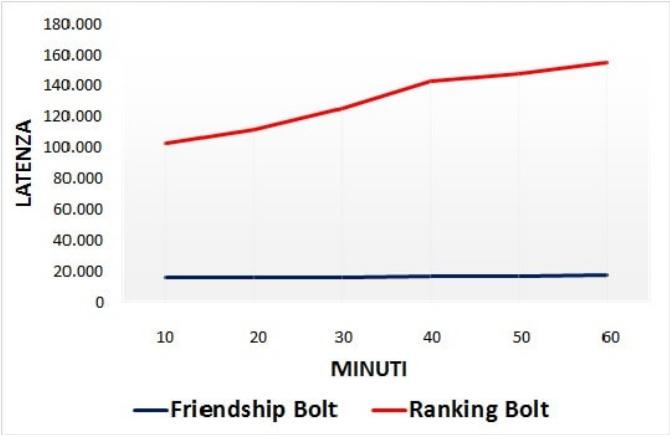

Figura 4. Confronto Ranking Bolt vs Frienship Bolt.

IV. TEACUP-STORM

Sin dalle prime fasi dello sviluppo di questo progetto, la possibilità di sfruttare lo scale-out fornito dalle moderne architetture cloud è sempre stata una tematica centrale, vista la decadenza della limitazione ad una sola macchina imposta dalla DEBS Grand Challenge 2016. Tale possibilità è stata perseguita ancora con più convinzione vista l'esistenza di diversi tool[3] [4], anche indicati dagli sviluppatori di Apache Storm, che permettono il deploy automatizzato sulla piattaforma Amazon Web Services. Tuttavia, nessuno di questi tool è aggiornato alla versione corrente di Apache Storm, e in particolare l'ultimo commit sul repository di `storm-deploy` risale al 2013.

Considerata la necessità di avere un tool semplice per effettuare il deploy su AWS, si è deciso di crearlo *ex-novo*, e di rendere il codice sorgente disponibile pubblicamente mediante la Apache License v. 2.0. Il progetto è stato denominato `teacup-storm`[5], parafrasando l'espressione inglese *storm in a teacup*, utilizzata in relazione a rabbia e preoccupazione espressa verso un problema non importante. Lo sviluppo del tool ha visto la collaborazione di altri studenti della Facoltà di Ingegneria nell'ambito di un nascente team open source, e continuerà ad essere sviluppato in futuro per dare supporto alle numerose tesi e progetti su Apache Storm, oltre che per fornire una alternativa aggiornata, semplice e funzionante agli applicativi già esistenti.

A. Funzionamento

Vista la relativa complessita delle alternative esistenti sulla rete, uno dei principi base adottati nello sviluppo del tool è la semplicità d'uso. All'utente infatti non si richiede altro che la compilazione di un file YAML, inserendo semplici parametri come il numero di worker machines desiderate, il numero di processi per macchina, i security group, le AMI che si desidera lanciare, la chiave ssh da utilizzare per la connessione, la VPC, il tipo di istanza, e le chiavi di accesso ad AWS. L'unica dipendenza richiesta è la libreria `boto3` di Python, e ovviamente un account AWS.

	Test 1		Test 2		Test 3	
	Capacity	**Execute Latency**	**Capacity**	**Execute latency**	**Capacity**	**Execute latency**
Mapper	0,001	0,164	0,009	0,464	0,147	0,591
Merge	0,000	0,131	0,000	0,105	0,000	0,040
Ranking	0,021	0,262	0,001	0,098	0,005	0,156

Tabella I
TEST QUERY 1

In questa tabella sono riportati i risultati ottenuti dai tre test della Query uno che sono stati descritti in precedenza. In particolar modo sono riportati i valori delle capacità dei singoli bolt e le latenze di esecuzione di una tupla. Un valore di capacità maggiore o uguale a 1 indica che il bolt è saturo. I valori delle latenze sono riportati in millisecondi.

	Test 1		Test 2	
	Capacity	**Execute Latency**	**Capacity**	**Execute latency**
Friendship	0,000	5,257	0,000	17,727
Merge	0,000	0,044	0,000	0,028
Ranking	1,392	175,912	1,511	199,400

Tabella II
TEST QUERY 2

In questa tabella sono riportati i risultati ottenuti dai due test della Query due che sono stati descritti in precedenza. In particolar modo sono riportati i valori delle capacità dei singoli bolt e le latenze di esecuzione di una tupla. Un valore di capacità maggiore o uguale a 1 indica che il bolt è saturo. I valori delle latenze sono riportati in millisecondi.

Una volta selezionati questi parametri, la configurazione è completamente automatizzata: inizialmente, il programma controlla se sull'AMI fornita sono presenti storm o zookeeper (a seconda del nodo che si sta considerando); se non sono presenti vengono scaricati in automatico. Se sono presenti, al contrario, si procede direttamente alla configurazione, e questo permette di riutilizzare AMI create in precedenza con un notevole risparmio di operazioni di I/O.

È importante notare che non c'è alcuna necessità di fornire in anticipo gli indirizzi IP delle macchine su cui fare il deploy di Apache Storm, dal momento che la configurazione viene gestita in modo dinamico da `teacup-storm`; questi indirizzi vengono poi forniti al termine della procedura di creazione, e possono quindi essere acceduti dall'utente.

Scendendo in dettaglio, vengono istanziati in macchine separate `storm ui`, un'istanza di `zookeeper`, un'istanza di `storm nimbus` e un numero scelto dall'utente di istanze di `storm supervisor`. Tutti questi processi vengono eseguiti in `supervisord`, dal momento che sia `storm` che `zookeeper` sono fail-fast e necessitano di essere riavviati in caso di errori. In ogni macchina viene inoltre eseguita un'istanza di `Logviewer` per la consultazione dei log tramite `storm ui`.

Come si può notare, si è scelto di istanziare ogni servizio su una macchina dedicata, questo perché si è cercato di avere una configurazione più semplice possibile, ovviamente però con l'addebito di maggiori costi per il cliente.

B. Difficoltà incontrate

Uno dei principali ostacoli incontrati è relativo all'indirizzamento delle macchine all'interno della VPC, dal momento che comunicano tra di loro tramite il DNS privato, pur essendo dotate di DNS e indirizzo IP pubblici. A causa di ciò, è risultato problematico accedere ai log tramite `storm ui`, collegandosi da una macchina remota rispetto alla VPC. Questo problema è stato risolto permettendo all'applicazione di modificare il file `/etc/hosts` in modo da effettuare la traduzione tra DNS privato e IP pubblico della macchina.

C. Estensioni future

Come già detto, `teacup-storm` è un progetto open source che continuerà ad essere sviluppato indipendentemente dal problema affrontato in questa sede. Diverse sono le estensioni già prese in considerazione. Vediamone alcune.

- Migliore gestione dell'indirizzamento tra le macchine: la soluzione attuale va a modificare il file `/etc/hosts`, l'idea è di trovare una soluzione più pulita e più portabile, per permettere il funzionamento anche su sistemi non Unix-like.
- Possibilità di avere cluster di nodi `nimbus` e `zookeeper`: al momento vengono istanziati nodi singoli, più che sufficienti per la maggior parte delle applicazioni; l'introduzione di cluster composti da più nodi comporterebbe un notevole miglioramento dell'affidabilità e delle prestazioni.
- Possibilità di istanziare più servizi su una macchina sola: attualmente ogni servizio è associato ad una macchina singola, ciò tuttavia comporta uno spreco di risorse dal momento che, ad esempio, `nimbus`, `ui` e `zookeeper` devono sopportare un carico lavoro molto inferiore rispetto alle istanze `supervisor`, e di conseguenza potrebbero essere istanziati su un numero minore di macchine.
- Possibilità di scegliere tipi diversi di istanze per i diversi servizi: nella versione attuale si può scegliere un unico tipo per tutti i servizi; questo tuttavia comporta un notevole aumento di costi, dato che come detto le risorse nei servizi di controllo sono sottoutilizzate anche con istanze piccole.

V. IL DEPLOY

Sebbene la DEBS Grand Challenge 2016 prevedesse l'esecuzione delle query su una singola macchina ad alte prestazio-

ni, in questo caso non vi è stata alcuna limitazione sulla scelta delle istanze EC2 su cui effettuare il deploy. Si è pertanto scelto di esplorare le possibilità di scale-out fornite dalla remote cluster mode di Storm, cercando per quanto possibile di effettuare un tailoring del numero e della potenza delle istanze in base all'effettiva potenza di calcolo richiesta, e cercando di ridurre al minimo gli scambi di dati non necessari.

Fatta eccezione per Redis, deployato su un'istanza `t2.small` di ElastiCache, tutto il cluster è stato realizzato facendo uso di `teacup-storm`, riutilizzando le stesse AMI per ridurre al minimo le operazioni di I/O su EBS. Come già detto, la versione attuale di `teacup-storm` permette di selezionare un solo tipo di istanza per il cluster, e ogni servizio ha la propria macchina dedicata.

Per cercare di sfruttare al massimo il parallelismo, e per evitare al tempo stesso di sovraccaricare le macchine, nei supervisor si è scelto un numero di slot pari al numero di vCPU della macchina su cui si esegue. Avremo quindi uno slot per istanze `t2.micro` e `t2.small`, e due slot per istanze `c4.large` e `m3.large`.

Per la query 1 si è inizialmente cercato di realizzare un cluster utilizzando istanze `t2.micro`, tuttavia 1 GB di RAM è risultato essere insufficiente per i nostri scopi; è più che sufficiente invece un cluster con istanze `t2.small`, con 2 GB di RAM; non si è reso necessario aumentare il parallelismo dato che la CPU risulta comunque sottoutilizzata, e l'unico problema era rappresentato dalla RAM.

La topologia della query 2, diversamente dalla prima, richiede delle prestazioni molto più elevate, e soprattutto un maggiore consumo di memoria, che andava ad esaurirsi con istanze piccole. Si è scelto quindi di utilizzare istanze `m3.large`, preferite alle `c4.large` a causa del limite di 5 istanze presente per queste ultime. Si sono effettuati test con cluster contenenti 4 e 8 supervisor, anche se gran parte del tempo di processamento è risultato essere dovuto alle chiamate a Redis, non influenzate da un maggiore parallelismo.

VI. Conclusioni

In questo paper abbiamo voluto fornire una nostra soluzione alla DEBS Grand Challenge 2016. L'idea di poter dare un contributo ad un problema così attuale e così difficile, ci ha permesso di progettare e implementare l'intero progetto con entusiasmo.

Quasi fin dall'inizio del progetto abbiamo avuto due obiettivi:

- poter progettare una soluzione alle due query, di per sé complicate, con strutture dati efficienti, volte a minimizzare la quantità di operazione eseguite e la quantità di memoria occupata.
- realizzare un tool automatizzato in grado di istanziare un cluster per istanziarci le topologie di Storm. Per questo motivo si è realizzato anche l'applicazione open source *teacup-storm*.

Nel lavoro che si è realizzato si sono incontrati diversi tipi di problemi, dalle normali difficoltà riscontrate nell'utilizzo di strumenti nuovi come Apache Storm, i servizi di Amazon e Redis, ma anche piccoli problemi con l'uso corretto delle strutture dati all'interno del progetto. Poter affrontare questi problemi è stato un fattore di crescita per noi importante.

Dai dati forniti dai test si può ben comprendere come effettivamente ci siano diversi miglioramenti da poter apportare al progetto come ad esempio rendere l'utilizzazione di Redis maggiormente efficiente, che tuttora rappresenta il collo di bottiglia della query 2.

Riferimenti bibliografici

[1] C. Bron e J. Kerbosch, *Algorithm 457: finding all cliques of an undirected graph*, Commun. ACM, ACM, 16 (9): 575–577, 1973.
[2] *Redis latency problems troubleshooting*, https://redis.io/topics/latency
[3] *storm-deploy*, https://github.com/nathanmarz/storm-deploy
[4] *storm-deploy*, https://github.com/KasperMadsen/storm-deploy-alternative
[5] *teacup-storm*, https://github.com/hopandfork/teacup-storm