Università degli Studi di Bari

DIPARTIMENTO DI INFORMATICA
Corso di Laurea Triennale in Informatica

Piattaforma cloud per l'analisi di big data provenienti da social network

Tecnologie per l'individuazione di contenuti a sfondo discriminatorio

Relatori:
Prof. Pasquale Lops
Dott. Pierpaolo Basile

Candidato:
Gianvito Taneburgo
Matricola 587645

Anno Accademico 2013 – 2014

Indice

Parte I

Prefazione

Introduzione

Il progresso tecnologico degli ultimi anni ha rivoluzionato il modo con cui gli esseri umani conoscono, lavorano, si relazionano con gli altri e, più in generale, interagiscono in un contesto sociale. Il fattore decisivo per questo cambiamento è stato Internet, la più grande rete di computer del mondo, accessibile dal 40% circa della popolazione mondiale [33].

Nato nel 1991 per opera di Tim Berners-Lee, il World Wide Web si è evoluto molto velocemente fino a diventare la più ricca sorgente di informazioni della storia dell'uomo, accessibile consultando liberamente miliardi di pagine web. Lo sviluppo di complesse tecnologie e di nuove tecniche di programmazione ha radicalmente modificato la struttura di queste pagine, che hanno perso la loro natura statica per acquisirne una notevolmente più dinamica. Gli utenti del Web hanno così potuto cominciare ad interagire con le pagine visualizzate ed a contribuire in prima persona alla pubblicazione di nuovi contenuti. In questa maniera si sono sviluppate nuove tipologie di siti, come forum, blog, chat, *wiki* e, solo in tempi più recenti, ***social network***. La trasformazione è stata così profonda da indurre all'utilizzo dell'espressione *Web 2.0* per descrivere la nuova forma del più celebre servizio di Internet.

Ciascun individuo ha acquisito primaria importanza nell'era dell'informazione (non a caso la rivista TIME ha scelto *You* come persona dell'anno nel 2006 [36][37]), trasformandosi da semplice fruitore di contenuti in sorgente copiosa di nuovi dati. Diversi fattori hanno, inoltre, contribuito alla crescita della mole di informazioni condivise in rete: lo sviluppo degli *smartphone* e delle reti di comunicazione digitali per telefonia mobile, l'aumento della velocità media di connessione ad Internet, la presenza di sensori in molti dispositivi di uso quotidiano, ecc. Il volume di dati prodotti è così aumentato a dismisura col passare degli anni, avviando quella che viene definita da alcuni come *era dei* ***big data***.

Le informazioni generate si sono concentrate nei *data center* dei colossi dell'*Information Technology* (IT) proprietari dei siti web più visitati, che le hanno utilizzate per creare nuove ed abbondanti fonti di guadagno, accrescendo ricchezza, fama e potere in modo proporzionale al volume dei dati

posseduti. Non stupisce, pertanto, notare come molti degli imprenditori più ricchi del pianeta dirigano proprio queste aziende [12], nè stupisce osservare nelle prime posizioni della classifica dei siti web più popolari al mondo [2] i *social network*, servizi che aggregano milioni di utenti al fine di costituire una rete sociale virtuale. Tra questi spiccano su tutti Facebook e Twitter che sono stati protagonisti di una espansione imponente, superando la complessa fase di *start up* per merito soprattutto della loro semplicità d'uso e dell'innovazione portata nel panorama dei siti web preesistenti. Questa improvvisa ed inaspettata crescita ha però presentato diverse sfide alle aziende proprietarie dei rispettivi siti: tutela della privacy degli utenti, recupero delle spese di gestione, design di interfacce facilmente usabili da individui profondamente diversi tra loro, ecc. La prova più complessa affrontata è stata certamente sviluppare **infrastrutture scalabili e performanti**, adatte a gestire enormi bacini di utenza e di dati. Quest'ultima è stata superata dagli *entrepreneur* per merito di avanzate tecnologie progettate appositamente per **memorizzare** ed **elaborare** *big data*.

Tali tecnologie sono state oggetto di approfondimento per questo caso di studio e, successivamente, sono state impiegate per lo sviluppo di una **piattaforma *cloud*** per l'analisi di grandi moli di dati provenienti da *social network*. La piattaforma è stata progettata per essere di supporto alla Mappa dell'Intolleranza, uno strumento promosso da Vox, Osservatorio Italiano sui Diritti, per monitorare le zone dove omofobia, razzismo, odio verso i disabili, misoginia ed antisemitismo sono maggiormente diffusi nel nostro Paese. La Mappa dell'Intolleranza, infatti, evidenzia i luoghi affetti da questi gravi problemi di discriminazione mediante l'uso di una mappa di calore dell'Italia, ottenuta analizzando i messaggi geolocalizzati inviati dal nostro Paese su Twitter. Quest'ultimo, più di tutti gli altri *social network*, si presta facilmente a veicolare messaggi d'odio, che richiedono, inoltre, accorgimenti particolari in fase di analisi per la loro peculiare natura (*hasthag*, menzioni, *emoticon*, link, ecc.). Una volta completato, lo strumento potrebbe essere usato per agire con interventi mirati sul territorio e prevenire l'insorgere di episodi spiacevoli.

Un sistema che mira ad analizzare velocemente, se non in *real-time*, i *tweet* inviati in Italia necessita di una infrastruttura scalabile, in grado di gestire una quantità enorme di dati. A questa esigenza prova a dare una soluzione la piattaforma sviluppata per il caso di studio, adoperando tecnologie specifiche per memorizzare e processare *big data* in tempi contenuti. Con l'obiettivo di massimizzare l'efficacia degli strumenti utilizzati, è stata progettata una architettura che si avvale di servizi *cloud* per portare a termine i compiti più esosi di risorse. Dopo diverse analisi, Google è stato selezionato come il *cloud provider* ideale per il caso di studio. La piattaforma sviluppata,

pertanto, impiega nel suo *core* alcuni prodotti della Google Cloud Platform, un portfolio di soluzioni *cloud* utilizzanti le stesse infrastrutture del motore di ricerca e degli altri prodotti di Google, come Gmail e YouTube.

Oltre ad essere stato adoperato per la raccolta di dati dal *social network*, il sistema è stato anche impiegato per migliorare il procedimento di individuazione dei *tweet* discriminatori tra quelli nel flusso di messaggi, conducendo indagini di natura statistica su quelli raccolti. L'obiettivo di questa analisi è stato quello di individuare i termini tipicamenti utilizzati nei *tweet* connotati da determinate forme di intolleranza, ma generalmente poco riccorenti in generici testi in lingua italiana. Coi risultati del lavoro si potrebbe rendere più efficace la raccolta di informazioni da mostrare sulla Mappa dell'Intolleranza, che risulterebbe quindi più accurata ed informativa.

Per la parte sperimentale del caso di studio sono stati progettati ed eseguiti due esperimenti. Il primo mira a confrontare le prestazioni ottenute da una macchina tradizionale con quelle fatte registrare da *cluster* di computer in esecuzione sull'infrastruttura di Google, provando configurazioni variabili sia per numero di nodi interconnessi, sia per tipologia di macchine avviate. Analizzando i tempi necessari per completare lo stesso *task* di indicizzazione è stato possibile valutare quando le tecnologie per il calcolo distribuito si rivelano una soluzione consigliabile e quando si dimostrano inefficienti. Il secondo esperimento, invece, si pone come obiettivo quello di evidenziare i termini più peculiari nei messaggi con contenuto potenzialmente intollerante verso il prossimo. Per eseguirlo è stata calcolata la divergenza di Kullback-Leibler tra i *tweet* collezionati ed il corpus itWaC, una grande raccolta di testi in lingua italiana. I risultati delle analisi hanno permesso di comprendere quali contenuti ricorrono con più frequenza nei messaggi ed hanno suggerito dei metodi per migliorare la raccolta futura di nuovi dati.

Sommario

Di seguito viene illustrata per sommi capi la struttura del lavoro.

Per via dell'importanza che rivestono in questo caso di studio, il capitolo 1 è dedicato esclusivamente alla descrizione dei *big data*. Sono analizzati i fattori che hanno permesso il loro sviluppo, le sorgenti contemporanee di nuove informazioni, con particolare riguardo per i *social network*, e le possibilità di profitto originabili dal loro impiego. Vengono, infine, spiegate le due criticità principali che emergono quando si gestiscono *big data*: l'elaborazione e la memorizzazione.

Per risolvere il primo problema, nel capitolo 2 viene illustrato il funzionamento di Hadoop, un *framework* per il calcolo distribuito attraverso *cluster* di computer. Nel corso del capitolo vengono approfonditi il *file system* distribuito condiviso dai nodi, il modello di programmazione MapReduce da seguire nella scrittura di programmi per il *framework*, i principi di funzionamento alla base della tecnologia e le possibilità a disposizione degli sviluppatori per migliorare le prestazioni dei *cluster*.

Il capitolo 3 è destinato alla descrizione dei database NoSQL, una nuova tipologia di database, progettata per memorizzare grandi moli di informazioni. Nel capitolo vengono illustrate le origini di questi database e, successivamente, viene effettuato un confronto con i database relazioni e quelli NewSQL, al fine di evidenziare i punti di forza e di debolezza di ogni tecnologia. Conclude il capitolo la descrizione della tassonomia dei database NoSQL.

Il capitolo 4 descrive le piattaforme di *cloud computing*, le caratteristiche che generalmente presentano ed i modelli di servizi che offrono ad aziende e privati. Particolare attenzione viene dedicata alla Google Cloud Platform, la piattaforma *cloud* utilizzata nel caso di studio, di cui sono descritti nel dettaglio i tre *web service* utilizzati: Compute Engine, Cloud Storage e Datastore.

La piattaforma *cloud* per l'analisi di *big data* sviluppata per il caso di studio utilizzando le tecnologie finora presentate viene descritta nel capitolo 5. I paragrafi che lo compongono illustrano il contesto del progetto, le finalità

del lavoro, l'architettura del sistema ed il funzionamento dei suoi componenti.

Il capitolo 6 riporta, per entrambi gli esperimenti svolti, una descrizione dei *dataset* di input, il protocollo sperimentale seguito, i risultati ottenuti ed una loro discussione.

Completano il lavoro delle sezioni contenenti le conclusioni sul lavoro svolto, i possibili sviluppi futuri del caso di studio ed alcuni sentiti ringraziamenti.

Parte II

Caso di studio

Capitolo 1

Big data

1.1 Definizioni

Con l'espressione *big data* generalmente ci si riferisce ad una grande quantità di dati digitali di varia natura. Tali dati possono essere originati nei modi più diversi: transazioni bancarie, cronologie di acquisti su siti di *e-commerce*, registrazioni di fenomeni atmosferici, sms, email, ecc.

Nel condurre uno studio su questi dati risulta inevitabile fare riferimento alle unità di misura dell'informazione digitale. La tabella 1.1, fedelmente alle indicazioni del Sistema internazionale di unità di misura (SI) [7], riporta i multipli del byte che saranno incontrati in questo e nei successivi capitoli. La sua consultazione è utile per avere una percezione della dimensione di certe sorgenti informative ed è raccomandata ad ogni nuova occorrenza di ordini di grandezza con cui non si ha familiarità.

Unità di misura	Simbolo	Valore (byte)
kilobyte	kB	10^3
megabyte	MB	10^6
gigabyte	GB	10^9
terabyte	TB	10^{12}
petabyte	PB	10^{15}
exabyte	EB	10^{18}
zettabyte	ZB	10^{21}
yottabyte	YB	10^{24}

Tabella 1.1: ordini di grandezza dei dati.

Definire i *big data* costituisce necessariamente il punto di partenza per l'analisi.

Il modo più semplice per formulare una definizione di *big data* è quello di fissare una soglia oltre la quale un insieme di *data* possa ragionevolmente essere considerato *big*. Si potrebbe, ad esempio, affermare che un qualunque insieme di informazioni avente dimensione superiore a 100 PB possa essere etichettato come *big data*. Questo approccio quantitativo, sebbene sembri universalmente valido, nasconde diverse insidie. Il primo problema che insorge è quello di riuscire a quantificare ragionevolmente questa soglia. Il concetto di *big*, infatti, è assolutamente relativo, poiché fortemente vincolato alle capacità di chi si trova a trattare la mole di dati. Un file di *log* contenente 35 milioni di operazioni, infatti, potrebbe risultare difficilmente processabile da un commerciante locale, che probabilmente non venderà tanti oggetti in tutto l'arco della sua attività, ed assolutamente ordinario per Amazon, che è ha venduto oltre 37 milioni di prodotti in un solo giorno [39]. Ma vi è anche un altro problema che deriva da questo approccio: una definizione di *big data* ben formulata non dovrebbe essere soggetta ad obsolescenza. Fissando una precisa quantità, invece, si corre il rischio, un domani più o meno prossimo, di avere una definizione non più valida. Si supponga, infatti, di aver trovato una soglia che oggi risulti adatta ad identificare i *big data*; non serve troppa lungimiranza per comprendere che, qualunque essa sia, col passare del tempo essa risulterà inevitabilmente inadatta. La capacità di memorizzazione dei dispositivi, infatti, crescerà continuamente, trasformando quelli che avevamo definito come *big data* in dei *not-so-big data*.

Risulta chiaro, da quanto detto, che la definizione deve essere inerentemente relativa, per rimanere appropriata, nel corso del tempo, agli strumenti che saranno prodotti dal progresso tecnologico. Questa idea è generalmente condivisa da tutti coloro i quali studiano e trattano *big data*.

Tra le tante definizioni di *big data* che è possibile leggere in letteratura, una afferma che si hanno *big data* quando i dati non possono più essere contenuti su una singola macchina [9]. Tale definizione potrebbe risultare accettabile, poiché è relativa alla capacità di memorizzazione di una macchina, che, verosimilmente, aumenterà nel corso degli anni. In tal modo, quelli che oggi sono considerati *big data* in futuro non lo saranno più, in favore di *dataset* aventi dimensione maggiore. Una definizione migliore, tuttavia, è la seguente [9]:

> Big Data means the data is large enough that you have to think about it in order to gain insights from it.

Questa si sofferma non su un aspetto quantitativo, che, come è stato illustrato, può comportare diversi problemi, ma su un aspetto qualitativo, ben più interessante. I *big data* vengono definiti tali non più in virtù dello spazio fisico che occupano, ma in relazione alle modalità di gestione che essi

richiedono. Si può parlare di *big data* quando le informazioni a disposizione sono così tante da richiedere un accurato ragionamento su come utilizzarle per estrarne qualche forma di conoscenza.

Anche l'IBM sottolinea come l'espressione *big data* alluda ad una realtà quotidiana che non lancia una sfida riducibile esclusivamente alla memorizzazione [19]:

> The term "Big Data" is a bit of a misnomer since it implies that preexisting data is somehow small (it isn't) or that the only challenge is its sheer size (size is one of them, but there are often more).

Vi è generale consenso sul fatto che i *big data* richiedano un modo sostanzialmente diverso di pensare alle modalità di utilizzo delle informazioni a disposizione, soprattutto se l'obiettivo è di trarne profitti.

Gli strumenti adatti all'elaborazione di dati di modeste dimensioni si rivelano del tutto incapaci di trattare *big data*. Apposite tecnologie sono state sviluppate per consentire di effettuare su enormi moli di dati le stesse operazioni che solitamente vengono eseguite su campioni più piccoli (acquisizione, memorizzazione, condivisione, analisi e visualizzazione). Alcuni degli strumenti che permettono di elaborare *big data* saranno analizzati e presentati in questo lavoro.

1.2 Fattori di sviluppo

La disponibilità di una enorme mole di dati che caratterizza i nostri giorni ha origine in una serie di fattori che, sinergicamente, hanno reso possibile la produzione di informazioni a ritmi elevatissimi.

Nessuno riuscirebbe, da solo, a generare *big data* in tempi modesti: essi sono il risultato di piccoli contributi di individui sparsi nel mondo. Risulta evidente, pertanto, che il fattore principale che ha avviato l'era dei *big data* è stato Internet, lo strumento che ha collegato gli abitanti del pianeta ed ha permesso a ciascuno di noi di poter fruire di una quantità enorme di contenuti.

Con oltre 14.000 miliardi di pagine web consultabili, lo 0,35% delle quali è indicizzato da Google, tre miliardi di internauti (circa il 41% della popolazione mondiale) hanno avuto la possibilità di condividere i loro dati ed attualmente si suppone che siano accessibili circa 672 EB di contenuti di natura digitale [18].

Internet, nonostante il suo potenziale illimitato, è stato però solo il più determinante tra i *key-enablers* che hanno avviato l'era del *Web 2.0*.

Un altro fattore è stato, certamente, l'incremento del potere computazionale dei personal computer, che procede incessantemente, seppur con ritmi diversi, sin dalla nascita dei primi calcolatori. L'osservazione empirica di Moore del 1965 [28], secondo la quale il numero di *transistor* in un circuito integrato sarebbe raddoppiato approssimativamente ogni due anni, si è rivelata corretta per quasi mezzo secolo, tanto da divenire legge. Come era stato teorizzato, la crescita ha subito un rallentamento ed ora il numero di *transistor* raddoppia ogni tre anni: questi ritmi sono sufficienti a garantire, comunque, un incremento considerevole delle prestazioni. La figura 1.1 mostra l'aumento del potere computazionale nelle ultime decadi.

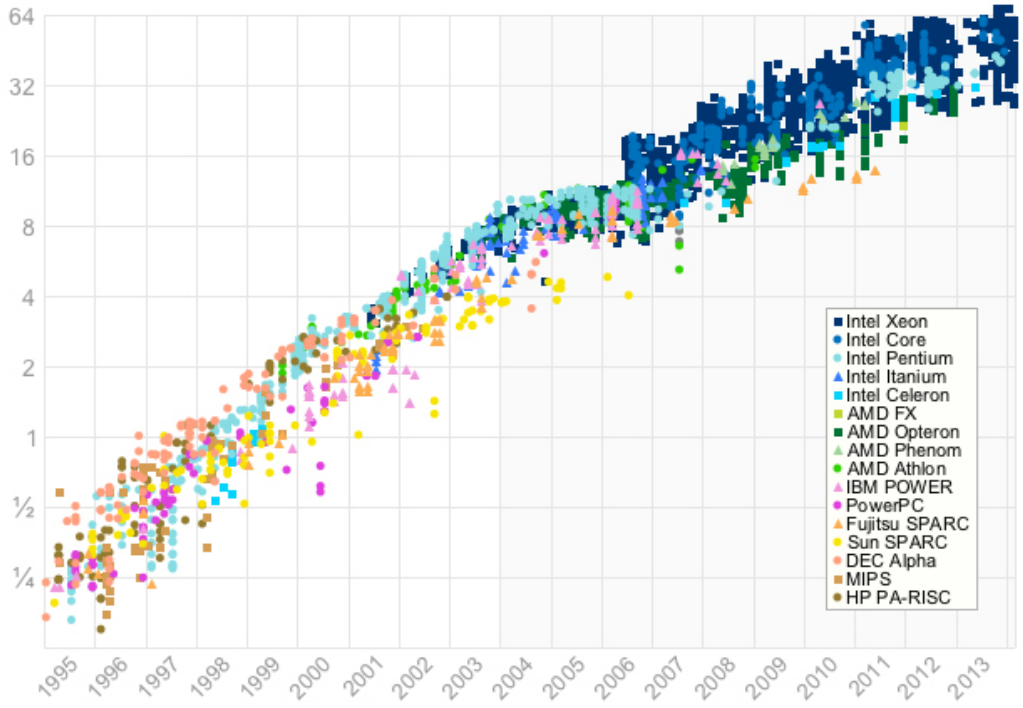

Figura 1.1: incremento nel tempo della capacità di elaborazione [30]. L'asse verticale impiega una scala logaritmica per visualizzare i *benchmark* normalizzati ottenuti nell'elaborazione di numeri interi con singolo *thread*. È possibile notare nella prima parte un incremento costante del 50%, che diminuisce dal 2004 assestandosi al 21%.

Analogo ed essenziale è stato l'aumento della capacità di memorizzazione dei dispositivi in commercio. Negli anni '90 i computer possedevano *hard disk* con dimensione media di 120 MB e tale capacità sembrava ampiamente sufficiente per le esigenze di chi allora li utilizzava; nel 2001 sono arrivati ad avere una capacità media di 40 GB [35]; nel 2014, vengono venduti a

prezzi modici *hard disk* di dimensione pari a 4 TB. Quello che stupisce non è l'economicità di questi prodotti o l'incremento delle capacità dei supporti, facilmente prevedibile osservando quanto accaduto per i processori, ma la facilità con cui questi dispositivi possano essere occupati in breve tempo, proprio a causa della enorme mole di dati a disposizione degli utenti.

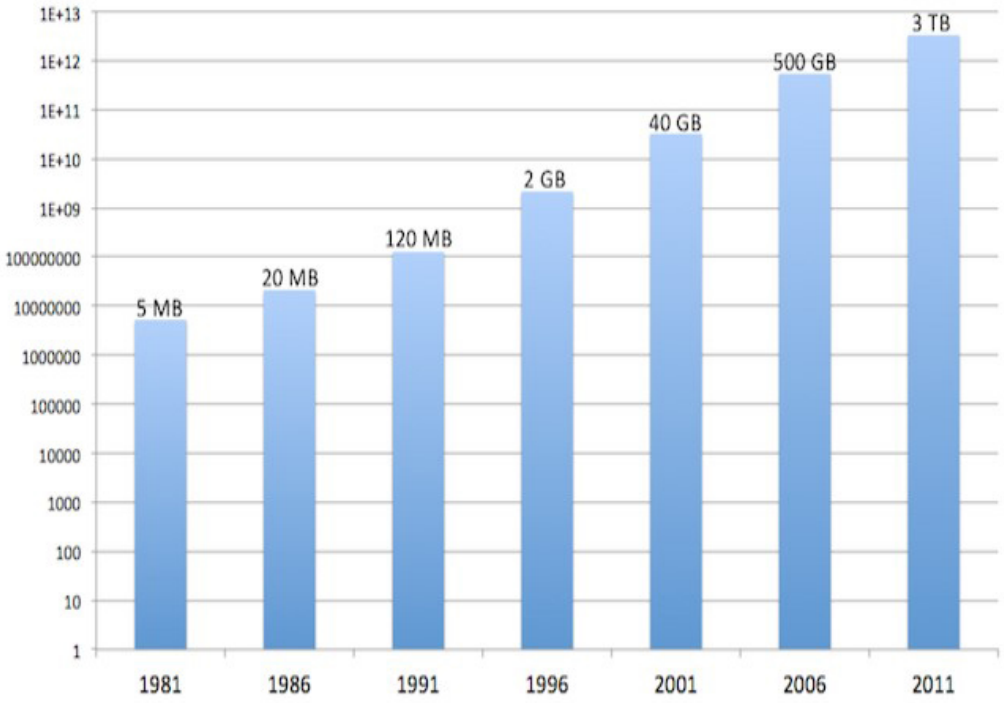

Figura 1.2: incremento nel tempo della capacità di memorizzazione [35].

Per condividere tutte queste informazioni su Internet gli utenti hanno necessitato di una infrastruttura che fornisse loro un'adeguata velocità di connessione. Anche in questo caso lo sviluppo tecnologico ha prodotto degli strumenti adeguati. Uno studio condotto da Jakob Nielsen nel 1998 [29] ha portato alla formulazione di una legge, simile a quella di Moore, che descrivesse bene il tasso di crescita della velocità di connessione, di seguito riportata nella forma sintetica:

> Users' bandwidth grows by 50% per year (10% less than Moore's Law).

I dati raccolti negli ultimi 15 anni confermano la legge e nel 2013 si è raggiunta una velocità di connessione media di 58 Mbps, una larghezza di banda che consente di condividere contenuti a ritmi elevatissimi.

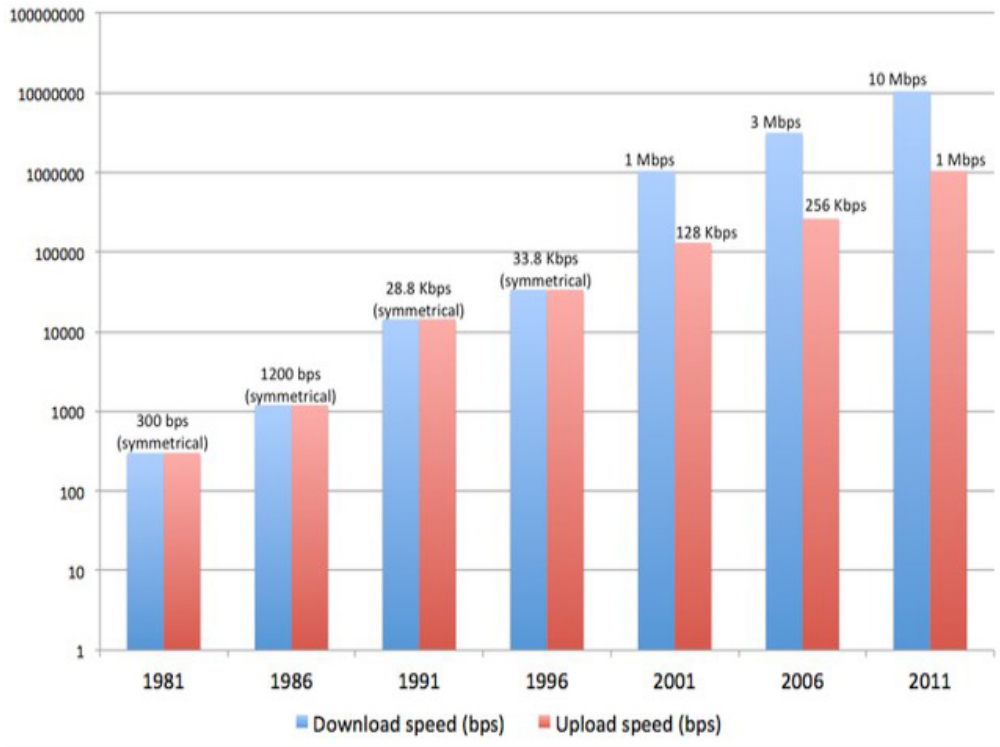

Figura 1.3: incremento nel tempo delle velocità di *download* ed *upload* [35].

Infine, la diffusione di *smartphone* e di altri dispositivi portabili ha dato agli utenti la possibilità di condividere contenuti in ogni momento. Il risultato è stato un aumento dell'utilizzo giornaliero medio di Internet, che è passato dai 46 min/giorno del 2002 alle 4 ore/giorno del 2012 [3].

L'incremento delle capacità dei computer, la diffusione di nuovi dispositivi mobili e la creazione di una rete che li connettesse tutti in modo veloce sono espressioni di un'equazione che ha come unica e scontata soluzione i *big data*.

1.3 Sorgenti di nuovi dati

Le sorgenti di informazioni che producono *big data* sono molte più di quelle facilmente immaginabili. Non siamo pienamente coscienti di quanto succede intorno a noi poiché i dispositivi tecnologici sono diventati parte della nostra quotidianità: li utilizziamo in modo trasparente, senza renderci conto della loro presenza. Il risultato è che generiamo e consumiamo continuamente dati, ma non ce ne accorgiamo.

Gli utenti del web sono raramente consci di produrre nuovi dati con le loro operazioni: il più delle volte, infatti, non sono consapevoli dei processi che vengono avviati in *server* sparsi nel mondo in risposta alle loro azioni. Realizzano di stare generando nuove informazioni, invece, pubblicando video su YouTube (si stima che ogni minuto vengano caricate 100 ore di contenuti [23]), creando nuove pagine web o caricando i loro file su servizi di *cloud storage* come Dropbox o Google Drive. Non lo sono pienamente, al contrario, quando effettuano ricerche su motori di ricerca, comprano oggetti *online*, avviano un *download* o cambiano canale sul televisore. Ogni azione che coinvolge l'utilizzo di Internet produce nuovi dati, che vengono conservati ed analizzati, con insidie latenti per la privacy degli utenti [41] [6].

Poiché i piccoli circuiti integrati sono diventati economici, è stato possibile aggiungerli ad ogni oggetto per dotarlo di una componente di intelligenza. Perfino le reti ferroviarie sono provviste di sensori, che permettono di monitorare traffico, condizioni meteorologiche, stato delle spedizioni e condizioni dei binari, in modo da prevedere ed evitare incidenti. Con questi dispositivi vengono costantemente raccolti voluminosi dati ambientali, finanziari, medici e di sorveglianza in tutto il mondo, ma contribuiscono a produrre molte informazioni anche giornali, riviste, sms, telefonate e sensori integrati nei dispositivi, come gli *smartphone*.

Una delle più grandi fonti di informazioni, tuttavia, si è costituita solo negli ultimi anni ed è di interesse specifico per questo lavoro: i *social network*. L'espansione capillare di questi siti web ha portato ad un aumento sensibile della mole di dati prodotti dall'uomo. A fronte dell'importanza di tali siti web per questo studio, si ritiene opportuno darne una descrizione più approfondita.

1.3.1 Social network

I *social network* sono siti che permettono agli utenti di costituire una rete di connessioni virtuali con altri individui, che siano amici o sconosciuti, e di restare in contatto con loro. Ciò che invoglia le persone ad utilizzarli è la facilità con cui essi potranno condividere i propri contenuti con i membri della loro rete. Tali contenuti sono spesso messaggi pubblici o privati, video, foto e notizie personali: nuove informazioni che, nel complesso, costituiranno *big data*.

I tanti *social network* attualmente *online* si differenziano principalmente per il *target* di utenti cui si rivolgono e per la tipologia di contenuti che tendenzialmente raccolgono. Tutti condividono, però, alcune caratteristiche peculiari. Dinamiche condivise sono la registrazione di un profilo con le

informazioni personali, la condivisione di nuovi elementi, l'instaurazione di nuovi legami e la visualizzazione di contenuti appartenenti ad altri individui.

Elencare i più famosi *social network* e descriverne le caratteristiche principali non è necessario ai fini di questo lavoro. Risulta anche inutile riportare tante statistiche riguardanti questi siti, poiché la maggior parte di esse non sarà più valida nel momento in cui qualcuno le leggerà. Per tale motivo ci si limiterà esclusivamente a descrivere Facebook e Twitter, due dei principali *social network* contemporanei, che sono stati oggetti di studio.

Facebook, fondato nel 2004, rappresenta il più fulgido esempio di *social network* e si può affermare che esso stesso abbia contribuito alla definizione dei canoni propri di questi siti. Ogni iscritto a Facebook possiede un diario personale, con visibilità pubblica o limitata, sul quale può pubblicare aggiornamenti di stato, foto, video, note e link a pagine web. Chi vuole ha facoltà di aggiungere, tra le sue informazioni personali, dettagli inerenti luogo e data di nascita, preferenze musicali, situazione sentimentale, orientamento politico o religioso, ecc. Il lato *social* di Facebook consiste nella possibilità di aggiungere nuovi amici con cui condividere tutti o alcuni dei suddetti contenuti. Negli anni sono state introdotte altre novità che hanno decretato il successo del *social network*, ad esempio la possibilità di utilizzare applicazioni, chattare con gli amici o creare gruppi di interesse, pagine (spesso utilizzate per attività commerciali) ed eventi. Facebook ha conosciuto una rapida espansione fino a diventare il secondo sito più visitato al mondo, secondo solo a Google. Gli utenti attivi mensilmente su Facebook ammontavano ad un milione nel 2004, a 145 milioni nel 2008 ed a 1,23 miliardi nel 2013 [16] (approssimativamente 1/7 degli abitanti del pianeta). Si stima che, nel 2012, ogni minuto venissero inviati 510.000 commenti, 293.000 aggiornamenti di stato e 136.000 foto [32].

Più modesti, ma non per questo meno straordinari, sono i numeri che gravitano attorno a Twitter, nato nel 2006 ed attualmente in settima posizione nella classifica dei siti più visitati al mondo, subito dopo Wikipedia e prima di Amazon [2]. Le dinamiche di interazione con il *social network* sono più limitate rispetto a quelle di Facebook. Gli utenti di Twitter - sono stimati 255 milioni di utenti attivi mensilmente [23] - utilizzano il sito principalmente per inviare messaggi, detti *tweet*, aventi una lunghezza massima di 140 caratteri. Sebbene sia possibile configurare la visibilità dei propri messaggi, Twitter viene usato solitamente senza filtri di sorta, lasciando visibili al mondo intero i propri *tweet*, diversamente da quanto avviene normalmente con Facebook, ove la visibilità è generalmente limitata agli amici. Nei *tweet* spesso vengono menzionati altri utenti o vengono inseriti dei *tag*, detti *hashtag*, che specificano il contenuto del messaggio o lo arricchiscono con informazioni o stati d'animo. Gli *hashtag* vengono usati per permettere al sistema di individuare e mostrare le tendenze che si delineano tra i circa 500 milioni di messaggi

giornalieri [23].

Il risultato di queste incessanti attività è che ogni giorno Facebook produce oltre dieci TB di nuovi dati e Twitter altri sette. Se a questi si aggiungono i dati generati anche sugli altri *social network* allora risulta evidente quanto sia abbondante la sorgente di nuove informazioni. Con questi ritmi non sembrano esagerate le stime del volume di dati memorizzati nel mondo: 800.000 PB nel 2000, 161 EB nel 2006, 35 ZB previsti per il 2020 [19] [20].

1.4 Opportunità di business

Per le sfide che impongono, potrebbe sembrare che i *big data* siano un problema per chi si trova a doverli trattare. In realtà, se sapientemente elaborati, essi costituiscono un'occasione per considerevoli guadagni. Non a caso le aziende che dispongono di più informazioni personali detengono più potere e ricchezza (Google, Amazon, eBay, Facebook, ecc.) [12] e non deve stupire il fatto che sempre più imprese provino a comprare da loro dati con cui profilare gli individui.

Sembra ormai assodata l'uguaglianza tra *big data* e *big power*. In verità *big data* è sinonimo di *big latent power*, poiché questi dati richiedono un processo accurato di elaborazione, al fine di estrarre conoscenza utile per un dominio applicativo o esigenze di mercato.

Il motivo per cui i dati rappresentano una così importante fonte di guadagno si può facilmente immaginare: conoscere abitudini e preferenze delle persone significa poter proporre loro pubblicità mirate oppure prevedere il loro comportamento ed agire di conseguenza. Emblematico è il caso di un *discount* che nel 2002 ha assunto uno statistico per provare a predire quali consumatrici fossero incinte, al fine di pubblicizzare prodotti specifici per i neonati. I proprietari erano convinti, infatti, che identificando una donna nel secondo trimestre della gravidanza avrebbero avuto modo di fidelizzarla per anni, prima vendendole pannolini, poi cereali, libri e DVD [38].

La conoscenza degli utenti, comunque, è utilizzabile in diversi altri modi oltre al marketing personalizzato, ad esempio per dotare i sistemi di interfacce grafiche personalizzate per ogni individuo, per guidare scelte operative (quante scorte avere in magazzino di un determinato prodotto), campagne politiche, ecc.

Ciò che succede è che sempre più organizzazioni hanno accesso ad un gran numero di informazioni (pagine web, file di *log*, *click-stream data*, documenti, sensori), ma non sanno come valorizzarle. Una indagine dell'IBM [19] ha riscontrato come oltre metà dei *business leader* sentono di non avere gli strumenti adeguati per comprendere ciò che è necessario per la loro attività.

Per tali ragioni, sempre più spesso, non sanno se conviene investire denaro e conservare questi dati (ammesso che abbiano i mezzi per farlo), nonostante siano sempre più numerosi gli strumenti che permettono di analizzarli per ottenere una visione più chiara del proprio business, dei clienti e del mercato.

Un altro problema che gli imprenditori devono affrontare è quello della sempre minore longevità dei dati. Ad oggi, infatti, non siamo più in grado di leggere gli *8-track tape*, i *floppy disk* ed i VHS, rispettivamente di 30, 20 e 10 anni fa. La durata media della vita dei dispositivi di memorizzazione è sempre più breve e gli strumenti correlati diventano obsoleti. Un *hard drive* è progettato per durare 5 anni, un nastro magnetico 10 e CD/DVD approssimativamente 20, sebbene già ora inizino ad essere impiegati sempre meno, in favore dei *blu-ray disc*. Ci troviamo di fronte ad un paradosso per il quale le capacità di produrre e memorizzare dati aumentano, ma la capacità di conservarli nel tempo diminuisce. Moltissime aziende, infatti, hanno dovuto trascrivere i loro dati su nuovi supporti con una cadenza di 10-20 anni. Il fenomeno a cui si assiste è la crescita, per dimensione e complessità, dei dati a disposizione di una azienda a discapito della capacità di poterli analizzare. Col passare del tempo questo *gap* sembra aumentare in modo incontrollato: gli imprenditori che si doteranno di strumenti per colmare questo divario sono quelli che controlleranno in futuro il mercato.

Analizzare i *big data* prodotti dai propri utenti, tuttavia, non è l'unico modo che gli *entrepreneur* hanno a disposizione per monetizzare. Diverse sono le aziende, infatti, specializzate nello sviluppo di tecnologie per l'analisi di questi dati che vengono poi noleggiate a terzi per ottenere importanti guadagni. Anche senza possedere *big data*, pertanto, gli imprenditori possono sviluppare dei servizi da vendere a chi ne dispone. Tali servizi possono riguardare infrastrutture per la loro memorizzazione o tecnologie per poterli elaborare efficientemente, finanche in tempo reale. Diverse sono le redditizie piattaforme sviluppate da colossi dell'IT ed offerte alle imprese; alcune delle principali sono di proprietà di Amazon, Google, IBM e Microsoft.

1.5 Criticità nella gestione

La natura dei *big data*, come è stato anticipato, solleva diverse problematiche risolubili solo con particolari tecnologie. Ogni azienda possiede un numero di informazioni variabile ed è interessata ad elaborare i propri dati per conseguire uno specifico obiettivo, pertanto i processi che essi subiranno cambieranno in ogni contesto. È possibile, tuttavia, individuare due criticità comuni a tutti i casi d'uso: l'elaborazione e la memorizzazione dei dati.

Nel momento in cui si deve progettare un sistema avente come obiettivo l'analisi di grandi moli di informazioni bisogna decidere con quali modalità sarà eseguita la computazione e in che modo saranno memorizzati i dati. I criteri che dovrebbero orientare il progettista verso una scelta consona sono due: la natura delle operazioni che si desidera effettuare e la loro frequenza.

In base alle esigenze proprie del dominio bisognerà trovare perciò la soluzione più adatta a risolvere i quattro problemi tipici della gestione di *big data*: come avviene l'elaborazione e come la memorizzazione, dove avviene l'elaborazione e dove la memorizzazione.

Nel corso dei capitoli seguenti saranno presentati nel dettaglio dei *framework* e degli strumenti progettati per risolvere questi problemi e che vengono tuttora frequentemente impiegati in molteplici scenari; adesso ci si limiterà a presentare sommariamente, per ogni problema, i due principali approcci tra cui scegliere.

Il primo problema riguarda come effettuare le operazioni sui dati. Questa scelta ha anche un impatto diretto sul paradigma di programmazione adottato in fase di scrittura del codice del programma preposto all'elaborazione delle informazioni. È possibile seguire un approccio classico ed elaborare i dati in modo centralizzato, su un unico *server*, oppure cercare una soluzione più scalabile, optando per una modalità di elaborazione distribuita tra più nodi di una rete. Questo secondo approccio è quello più adottato e, per tale motivo, i *framework* che consentono una ripartizione automatica del carico di lavoro tra computer interconnessi hanno conosciuto una rapida evoluzione. Un esempio è offerto da Apache Hadoop, che sarà approfondito nel capitolo successivo.

Per quanto concerne le modalità di memorizzazione delle informazioni, è possibile scegliere tra la consolidata soluzione offerta dai database relazionali oppure scegliere sistemi innovativi per conservare i dati. Nel corso del capitolo 3 sarà effettuato uno studio comparativo dei database NoSQL, una nuova tipologia di database che meglio si adatta a gestire *big data*.

Le soluzioni al problema riguardante l'ubicazione del processo di elaborazione sono riconducibili a due tipologie principali: effettuare la computazione in locale su uno o più *server* propri oppure percorrere la via del *cloud computing*, delegando l'elaborazione a macchine remote. Come detto nel paragrafo precedente, diverse aziende come Amazon e Google offrono dei servizi con cui è possibile elaborare efficientemente grandi moli di dati. In seguito verrà dimostrato come essi costituiscano un'opzione valida ed economicamente vantaggiosa rispetto all'effettuare il processo di elaborazione su macchine in locale.

Le opzioni appena illustrate per risolvere il problema dell'elaborazione sono anche valide per quello della memorizzazione. È possibile conservare i

dati su *server* propri oppure optare per il *cloud storage*, trasferendo i propri file in rete. Anche per questo scopo è possibile trovare *online* servizi di noleggio di spazio virtuale ove caricare i propri dati e renderli disponibili in qualunque momento e da qualunque parte del globo.

Tutte le varie possibilità presentate offrono ai progettisti ampia libertà di scelta. Essi potranno propendere per una architettura interamente *cloud* oppure per soluzioni ibride, che impiegano servizi remoti solo per soddisfare alcune necessità, delegando ai propri *server* il resto delle operazioni.

Progettare correttamente la struttura del sistema risulta fondamentale quando vengono elaborati *big data*, poiché è alta la probabilità di incappare in *bottleneck* che potrebbero far degradare le performance del sistema. Ulteriore attenzione va posta se si adoperano soluzioni *cloud*, poiché vanno studiati attentamente i costi di questi servizi al fine di valutare l'effettiva economicità dell'opzione.

Capitolo 2

Elaborazione distribuita: Hadoop

2.1 Descrizione del framework

Un punto di svolta fondamentale nello sviluppo di tecnologie orientate ai *big data* è stata la pubblicazione, ad opera di Google, di alcuni articoli in cui venivano illustrati i principi di funzionamento di alcuni strumenti sviluppati ed utilizzati dal colosso per far fronte alle proprie gravose esigenze. Il *paper* sul Google File System del 2003 [13] e quello su MapReduce del 2004 [8], più degli altri, hanno radicalmente cambiato il modo di pensare ai dati ed hanno avviato lo sviluppo di molte nuove tecnologie, tra cui Hadoop [25].

Hadoop è un progetto *top-level* della Apache Software Foundation, scritto in Java da una community globale di contributori. Lo sviluppo del *framework* è cominciato nel 2004 come sotto-progetto di Lucene, una libreria per l'*information retrieval*, da cui poi si è distaccato per divenire un progetto a sé stante. Hadoop ha avuto un'importante evoluzione dal 2005, sotto la spinta di Yahoo!, interessato a realizzare una tecnologia che potesse competere con quella sviluppata da Google, che, nell'articolo del 2004, descrive MapReduce, un modello di programmazione ed un omonimo *framework* scritto in C++, progettati per eseguire in parallelo programmi su *cluster* di computer. Il *framework* di Google, di cui Hadoop è una riproduzione *open-source*, si preoccupa di distribuire la computazione tra i nodi del *cluster*, gestire i malfunzionamenti delle macchine, bilanciare i carichi di lavoro ed accorpare i risultati prodotti da ciascun nodo prima di restituirli in output all'utente. In questa maniera uno sviluppatore può concentrarsi sulla scrittura del codice del proprio programma, anche senza avere esperienza di computazione parallela, poiché essa sarà gestita automaticamente dal *framework*. Il codice scritto, che deve aderire al modello di programmazione MapReduce (descritto nel dettaglio nel paragrafo 2.3), risulta lineare, poiché non contiene sezioni

orientate a gestire meccanismi di parallelizzazione o recupero da situazioni di errore.

In questa maniera Hadoop diventa lo strumento ideale, talvolta l'unica opzione possibile, per elaborare grandi moli di dati, perché permette di ripartire pesanti carichi di lavoro tra tante macchine di un *cluster*, ciascuna delle quali processa solo una ridotta frazione dei dati di input. Per tal motivo, moltissime aziende che trattano *big data*, quali Yahoo!, Facebook, Twitter, eBay, LinkedIn e Spotify, impiegano *cluster* con Hadoop per effettuare l'elaborazione dei propri dati [40]. Intorno ad Hadoop si sono poi sviluppati numerosi altri progetti, tra cui HBase, Pig, Hive e Spark, che vengono, per convenzione, inclusi nella "piattaforma Hadoop". Questi progetti, generalmente, utilizzano Hadoop nel loro *core* per aggiungergli delle caratteristiche o realizzare tecnologie più complesse, come database o strumenti di analisi.

Nella sua versione iniziale Hadoop era progettato per l'elaborazione *batch* dei dati, ma ora ci sono diversi servizi che consentono di adoperarlo in applicazioni *real-time* (*stream-processing*, *real-time query*, ecc.).

I moduli principali che compongono il *framework* dell'ultima versione (2.5.0) sono: Hadoop Common, una raccolta di strumenti impiegati da diversi moduli, Hadoop Distributed File System (HDFS), il *file system* presentato nel dettaglio in seguito, Hadoop YARN, un *framework* per il *job scheduling* e la gestione delle risorse del *cluster* e Hadoop MapReduce, il sistema basato su YARN per elaborare in parallelo grandi *dataset*.

Nonostante sempre più strumenti vengano sviluppati per elaborare in parallelo i dati, Hadoop rimane una tecnologia largamente impiegata nelle imprese dell'IT. A conferma di quanto detto, diverse sono le importanti aziende che offrono *software* o servizi basati sul *framework*: alcuni esempi sono Cloudera, MapR e Hortonworks, aziende specializzate nello sviluppo di piattaforme e strumenti per l'analisi di *big data*.

Nei paragrafi seguenti saranno descritti l'Hadoop Distributed File System ed il modello di programmazione MapReduce, essenziali per comprendere i principi di funzionamento del *framework*.

2.2 Hadoop Distributed File System

Una componente essenziale del *framework* è il *file system* che tutti i nodi del *cluster* utilizzano: l'Hadoop Distributed File System (HDFS).

La necessità di sviluppare un *file system* specifico per Hadoop derivava da due esigenze avvertite, sin dalle fasi iniziali dello sviluppo, dai contributori del progetto. Il primo problema da risolvere era causato dalla natura distribuita del *framework*: ogni nodo del *cluster*, infatti, per eseguire i compiti

assegnati, doveva poter accedere ai dati di input del programma. La seconda problematica, invece, era legata alla tipologia di questi dati di input: *big data*, non memorizzabili fisicamente su una solo macchina, coerentemente con la definizione data nel paragrafo 1.1. Replicare i *dataset* su ogni nodo, pertanto, risultava impossibile.

L'Hadoop Distributed File System risolve questi ed altri problemi adoperando soluzioni architetturali già presenti in altri *file system* distribuiti ed ignorando alcuni vincoli POSIX per ragioni di efficienza, similmente al Google File System (GFS). Una assunzione che viene fatta, ad esempio, è che i dati del *cluster* siano soggetti a *batch processing* e pertanto il modello adottato è di tipo *write-once-read-many*. Per tale motivo, una volta creati, i file non possono essere modificati (nel GFS, invece, è possibile eseguire operazioni atomiche di *append*): questo, sebbene rappresenti un grande limite, consente di ottenere un elevato *throughput*.

L'HDFS gestisce i file in modo gerarchico, come un *file system* tradizionale, utilizzando cartelle e *path*, con la differenza che i dati non vengono memorizzati fisicamente in un unico luogo, ma vengono ripartiti tra i nodi di un *cluster*. I file, infatti, vengono suddivisi in *chunk* di 64 MB (l'ultimo *chunk* può avere dimensione inferiore) e distribuiti tra i nodi, che, in questo modo, si dividono l'onere di conservare grandi moli di dati. Il *file system* è progettato per essere avviato su grandi *cluster* con *commodity hardware*, dove i malfunzionamenti non sono un caso, quanto la norma (*crash* delle macchine, rottura dei *drive*, errori di rete, ecc.). Per tale motivo, al fine di garantire sempre la disponibilità dei dati, i *chunk* vengono replicati (il fattore di replicazione di *default* è tre, ma è configurabile).

A livello architetturale l'HDFS presenta una struttura *master/slave*, illustrata in figura 2.1. Un unico nodo *master*, il NameNode, si preoccupa di gestire il *file system*, conservando l'elenco dei file memorizzati e tenendo traccia della loro ubicazione nel *cluster* attraverso metadati. Tutti gli altri nodi *slave* del *cluster*, invece, ricoprono il ruolo di DataNode e conservano i *chunk* che sono loro assegnati dal NameNode. Quest'ultimo, conoscendo la locazione dei *chunk*, può indirizzare le richieste del *client* ai DataNode opportuni, che si occuperanno di leggere i dati ed inviarli al programma che li consumerà. Il NameNode si preoccupa, inoltre, di eseguire operazioni quali la rinomina o la cancellazione dei file, ma anche di verificare che i *chunk* siano integri (attraverso il *checksum*) e che siano sufficientemente replicati (in caso negativo avvia la loro copia su nuovi DataNode).

Diversi accorgimenti permettono al NameNode di mantenere un indice dei *chunk* sempre aggiornato per fornire dati consistenti a chi legge dal *file system*. Esso conserva in memoria, innanzitutto, un file contenente le proprietà del sistema ed il *mapping* tra i blocchi dei file ed i nodi del *cluster*. Tale file,

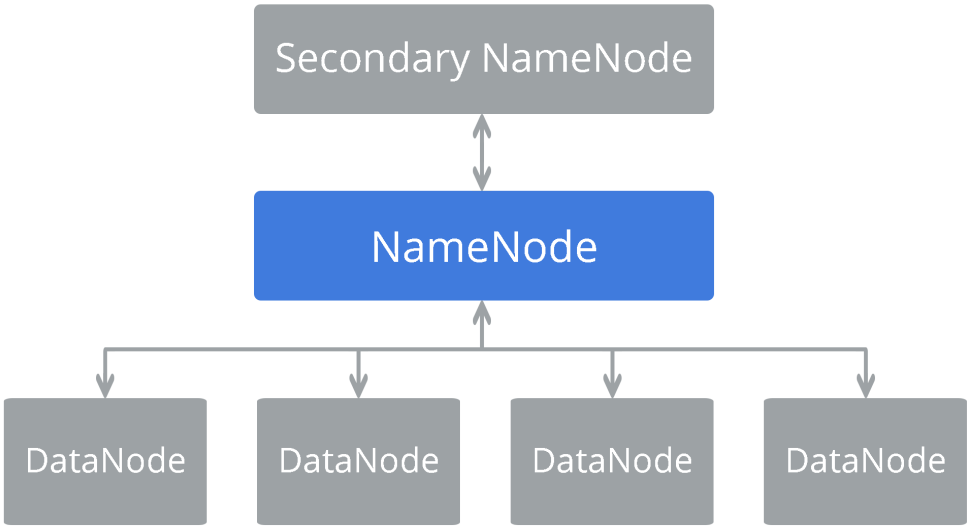

Figura 2.1: architettura dell'HDFS [25].

detto FsImage, viene periodicamente serializzato ed occupa poco spazio (un NameNode con 4 GB di RAM supporta un numero enorme di file e *directory*). Al suo avvio, il NameNode carica l'ultima FsImage ed interroga tutti i DataNode del *cluster* che gli forniscono una risposta, detta BlockReport, contenente l'elenco dei *chunk* da loro attualmente conservati; combinando le informazioni ottenute, il nodo *master* ottiene una rappresentazione generale del sistema e provvede a verificare che ogni file sia aggiornato e sufficientemente replicato. Tutte le azioni eseguite dal NameNode sul *file system*, inoltre, vengono memorizzate e replicate in un file di *log*, detto EditLog, che consente, in caso di fallimento del nodo *master*, di ripristinare al suo riavvio l'ultima configurazione del *file system*. Il nodo *master*, infatti, ricarica l'ultima FsImage e riesegue su di essa le ultime operazioni andate perdute. Per tale motivo, una volta serializzata una nuova FsImage, l'EditLog può essere svuotato, poiché le ultime modifiche sono conservate nell'ultima immagine del sistema.

L'HDFS non supporta gli *snapshot* per memorizzare una copia dei dati ad un certo istante temporale - il GFS, invece, beneficia di questa funzionalità – e non dispone di un meccanismo per riavviare automaticamente il NameNode in caso di malfunzionamenti. Quest'ultimo, inoltre, rappresenta un *single-point-of-failure* del sistema, poiché è l'unico a poter eseguire operazioni sul *file system*. A fronte di questi svantaggi, tuttavia, l'HDFS presenta altri punti di forza, tra cui una indiscutibile semplicità di accesso. Esso può essere utilizzato dalle applicazioni in diverse maniere; espone infatti una API

Java ed un *wrapper* in C di tale API, consente la navigazione tramite browser mediante il protocollo HTTP e la *shell* presenta la maggior parte dei comandi tradizionali riguardanti i file con semantica intuitiva. Un'altra interessante caratteristica del *framework* riguarda la capacità di suddividere in modo ottimale le repliche dei *chunk* tra i DataNode per ottimizzare le performance in fase di lettura. L'HDFS, infatti, in linea col principio secondo cui "*moving computation is cheaper than moving data*", cerca di minimizzare lo spostamento di dati sulla rete, distribuendo in modo omogeneo i blocchi di file tra i vari *rack* del *cluster*, per consentire ai *client* di avere sempre a brevi distanze dei nodi contenenti le informazioni a loro necessarie.

2.3 Modello di programmazione MapReduce

In questo paragrafo vengono descritti i principi fondamentali del modello di programmazione MapReduce, alla base sia del *framework* di Google, che di quello di Apache; per tale motivo, durante la descrizione, ci si riferirà indistintamente ai due prodotti, poiché le differenze che sussistono riguardano principalmente dettagli realizzativi e non strutturali.

Per semplificare il processo di elaborazione dei dati avvantaggiandosi del *framework*, i programmi da eseguire sul *cluster* devono essere costituiti da due operatori: *map* e *reduce*. Nonostante le tipologie di computazione da effettuare sui dati siano di natura diversa, la maggior parte di esse può essere realizzata impiegando queste due sole funzioni, come spiegato anche nell'introduzione di Google su MapReduce:

> We designed a new abstraction that allows us to express the simple computations we were trying to perform but hides the messy details of parallelization, fault-tolerance, data distribution and load balancing in a library. Our abstraction is inspired by the map and reduce primitives present in Lisp and many other functional languages. We realized that most of our computations involved applying a map operation to each logical "record" in our input in order to compute a set of intermediate key/value pairs, and then applying a reduce operation to all the values that shared the same key, in order to combine the derived data appropriately.

Un programma che implementa correttamente le funzioni *map* e *reduce* è in grado di essere eseguito dal *framework* in modo distribuito: il programma-

tore può, perciò, concentrare la sua attenzione esclusivamente sulla scrittura di tali operatori.

La funzione *map* riceve in input una coppia chiave/valore (K, V) e produce in output un insieme di coppie intermedie chiave/valore (I, P). Il *framework* raggruppa automaticamente le coppie intermedie aventi stessa chiave I e inserisce tutti i corrispondenti valori P in una lista. La funzione *reduce* riceve in input una di queste coppie (I, [P1, P2, P3, ...]) generate dal *framework* e produce un insieme, possibilmente ridotto, di tali valori. È possibile schematizzare le funzioni nel modo seguente:

```
map (K, V) -> list(I, P)
reduce (I, list(P)) -> list(P)
```

I tipi delle chiavi e dei valori intermedi sono solitamente quelli supportati dal linguaggio di programmazione del *framework*; generalmente sono sufficienti interi, *double*, *long* e stringhe per la maggior parte dei casi d'uso.

Il *framework* assegna ai nodi del *cluster* l'esecuzione di una o più funzioni *map*/*reduce* e ne raccoglie i risultati (per i dettagli sul funzionamento si rimanda al paragrafo successivo). Ciascun nodo possiede il codice di entrambi gli operatori e, su richiesta del *framework*, esegue la funzione che gli viene richiesta sui dati di input che riceve.

Per chiarire quanto appena spiegato viene presentato lo pseudocodice delle funzioni *map* e *reduce* che si potrebbero scrivere per indicizzare un documento:

```
map(int lineNumber, String line):
  for each word w in line:
    EmitIntermediate(w, 1);
```

```
reduce(String w, List<Integer> values):
  Emit(w, length(values));
```

Il programma conta, per ogni parola, il numero di occorrenze nel documento. La funzione *map* riceve in input una coppia avente per chiave un numero naturale rappresentante la riga i-esima del documento e per valore una stringa uguale alla riga i-esima. L'operatore itera su ogni parola w della riga e produce in output una coppia (w, 1), a significare che la parola w è stata incontrata una volta. Al termine dell'esecuzione di tutte le funzioni *map*, il *framework* si occuperà di accorpare le coppie aventi stessa chiave prodotte dai vari nodi del *cluster*: per ogni parola w del documento verrà creata una coppia avente come chiave w e come valore una lista dei valori intermedi a lei abbinati (in questo caso una lista di numeri 1, provenienti dalle funzioni *map* eseguite su diversi nodi). Le coppie di input per la funzione

reduce saranno pertanto della forma (w, list(1, 1, ..., 1)), che si limiterà a controllare la dimensione della lista per restituire in output coppie del tipo (w, #occorrenze).

Le funzioni potrebbero essere schematizzate così:

```
map (int, String) -> list(String, int)
reduce (String, list(int)) -> (String, int)
```

Come è possibile notare, il modello di programmazione MapReduce permette di scrivere semplici funzioni, delegando al *framework* l'onere di gestire il calcolo in parallelo tra i nodi. Le funzioni *map* e *reduce* illustrate permettono di indicizzare agevolmente *dataset* dell'ordine di PB, a patto di avere *cluster* di dimensione appropriata.

2.4 Principi di funzionamento del framework

Nel corso degli anni Hadoop è stato protagonista di una grande evoluzione, che ha causato, talvolta, modifiche sostanziali al *framework* nel passaggio da una versione all'altra. Sebbene il *core* sia rimasto immutato, diverse altre componenti sono cambiate radicalmente. Per tale motivo, nell'illustrare i principi di funzionamento di Hadoop, sarà scelta una versione di riferimento, la 0.18. I componenti fondamentali di questa *release* sono rimasti pressoché immutati fino alla versione 2 del *framework*, quando sono state introdotte molte novità, tra cui YARN.

Come si è potuto evincere, un *cluster* di Hadoop è composto da diversi nodi, ciascuno dei quali ricopre uno specifico compito. Tali ruoli vengono assegnati alle macchine in fase di configurazione del *cluster* dal manager del sistema, che deve avere pertanto una conoscenza profonda dell'architettura del *framework*: una corretta configurazione, infatti, è essenziale per ottenere elevate prestazioni. Sebbene scrivere programmi secondo il modello MapReduce sia relativamente semplice, configurare correttamente il *cluster* può risultare laborioso per chi si approccia al *framework* per la prima volta. Per questo motivo sono stati sviluppati degli strumenti per effettuare automaticamente il *tuning* dei parametri di configurazione (ad esempio Starfish) e *script* personalizzabili per eseguire il *deploy* del *cluster* (ad esempio quello offerto dalla Google Cloud Platform).

Eseguire un *cluster* di Hadoop significa avviare su ciascun nodo uno o più *daemon*, che specificano il ruolo del nodo nella rete. In generale, è possibile suddividere i *daemon* in due classi: quelli di computazione e quelli di memorizzazione. I *daemon* di computazione sono il JobTracker ed il TaskTracker,

mentre quelli di memorizzazione sono il NameNode, il Secondary NameNode ed il DataNode.

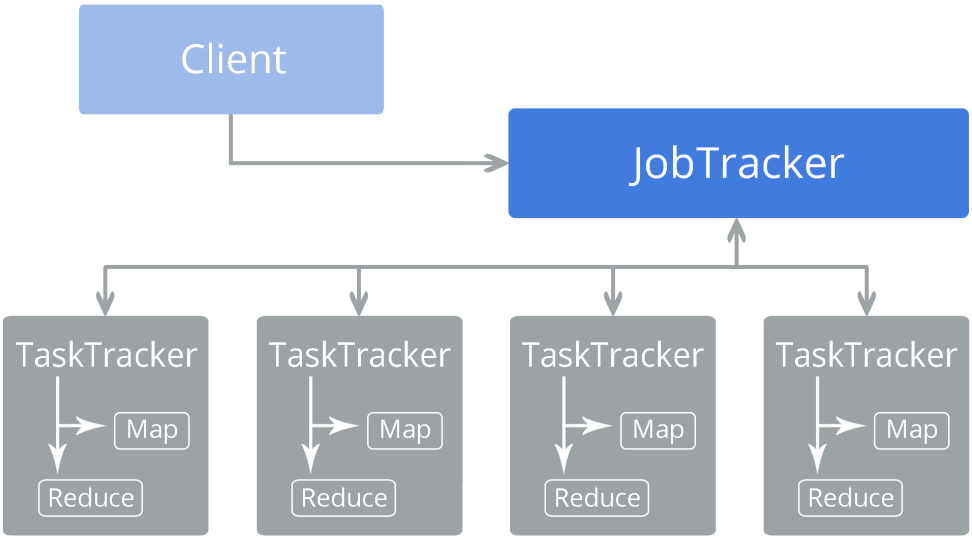

Figura 2.2: architettura del framework [25].

Il JobTracker è il punto di collegamento tra l'applicazione scritta dall'utente ed il *framework* (come si può notare in figura 2.2). Una volta inviato il proprio programma ad Hadoop, il JobTracker fissa un piano di esecuzione, assegna ai nodi disponibili dei *task* da completare e si preoccupa di raccogliere e ridirezionare i risultati intermedi della computazione. Il *daemon*, inoltre, controlla se i nodi lavoratori stiano correttamente funzionando interrogandoli periodicamente; se non riceve risposte da un nodo, il JobTracker assume che sia incappato in una situazione di errore e riassegna il *task* ad un nuovo nodo inattivo. Nel *cluster* vi è solo un JobTracker ed è tipicamente il *master node* del *cluster*. Esso rappresenta un *single-point-of-failure* (con YARN, invece, il ResourceManager non è affetto da questa criticità). Generalmente non si eseguono TaskTracker sul nodo dove è in esecuzione il JobTracker al fine di non sovraccaricare ulteriormente la macchina e non incorrere in degradi di prestazioni.

Il TaskTracker è uno *slave node* e si occupa di eseguire i *task map* e/o *reduce* assegnatigli dal JobTracker, cui spedisce i risultati elaborati. Sebbene ci sia al più un solo *daemon* TaskTracker in esecuzione su ogni nodo, ciascuno di essi può avviare istanze multiple della Java Virtual Machine, in modo da iniziare diversi *task* in parallelo. Il TaskTracker, mentre svolge i compiti, informa periodicamente il JobTracker della sua attività.

Tra i *daemon* di memorizzazione, NameNode e DataNode sono stati già esaustivamente descritti nel paragrafo sull'HDFS. Resta da analizzare il Secondary NameNode, sebbene il suo nome sia eloquente. Il ruolo di questo *daemon*, infatti, è di assistenza al NameNode, che è un *single-point-of-failure*, per monitorare lo stato del *cluster*. Come per il NameNode, ogni *cluster* ha un solo Secondary NameNode, che risiede generalmente su una macchina dedicata, senza altri *daemon* in esecuzione. Esso non riceve in tempo reale aggiornamenti sulla situazione dell'HDFS, ma comunica ad intervalli configurabili con il NameNode per effettuare *snapshot* dei suoi metadati. Le informazioni che il Secondary NameNode memorizza possono aiutare a minimizzare i tempi di *downtime* dovuti a fallimenti del NameNode principale, poiché è possibile, previo intervento manuale, promuovere il Secondary NameNode a NameNode per riattivare il *cluster*. Una rappresentazione di tutta l'architettura è fornita nella figura 2.3.

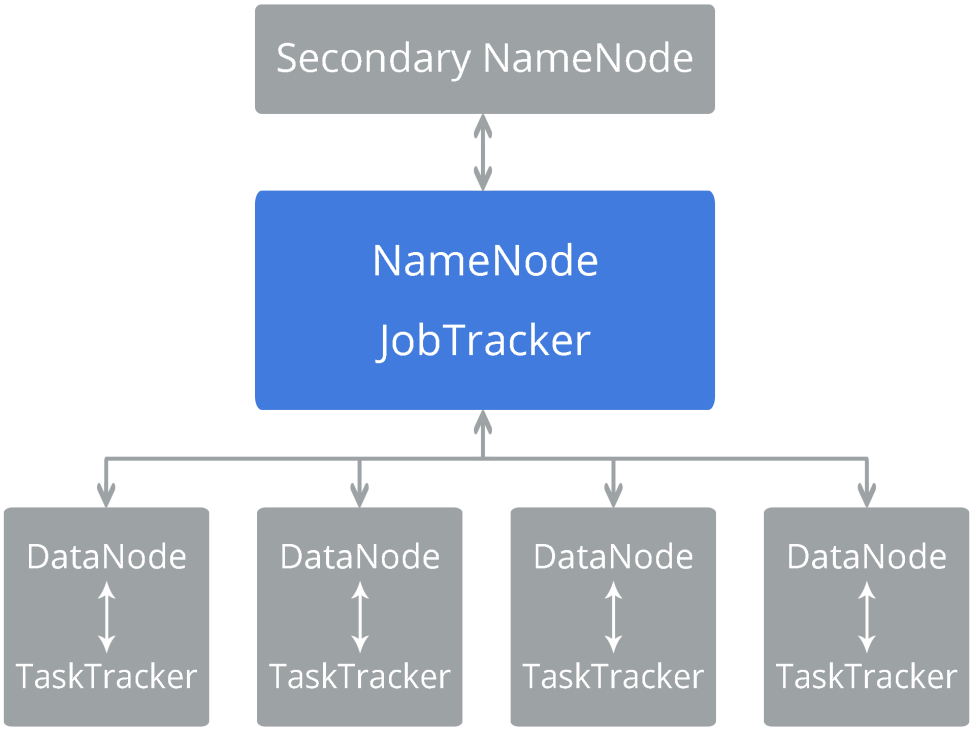

Figura 2.3: ubicazione dei *daemon* nel *cluster* [25].

Su ciascun nodo *slave* sono solitamente in esecuzione sia il DataNode che il TaskTracker per ripartire sia il carico di lavoro che quello di memorizzazione. Per ottenere buone prestazioni, Hadoop è in grado di assegnare i *task* di *map* e *reduce* in modo da sfruttare la località dei dati ed evitare inutili trasferimenti

di informazioni nella rete. Il *framework*, perciò, assegna i *job* ai TaskTracker in esecuzione sulle macchine più vicine ai DataNode contenenti i dati di input necessari.

2.4.1 Modalità di esecuzione

Il *framework* Hadoop può essere eseguito in tre modalità differenti: locale, pseudo-distribuita e distribuita.

La modalità locale (*standalone*), impiegata in *default* da Hadoop, necessita di una sola macchina e non di un *cluster*. In questa modalità non vengono avviati né i *daemon* di Hadoop precedentemente illustrati né l'HDFS, poiché non ci sono forme di comunicazione tra i nodi: i dati ed il codice sono residenti sulla stessa macchina, che esegue un programma in modo tradizionale. Questa modalità è spesso impiegata in fase di sviluppo e di *debug*, poiché permette di verificare la correttezza della logica di un'applicazione MapReduce, senza trattare con la complessità addizionale dovuta all'interazione con i *daemon*.

Anche la modalità pseudo-distribuita richiede una sola macchina ed equivale ad eseguire un *cluster* con un solo nodo. Su questo nodo vengono eseguiti tutti i *daemon*, inclusi quelli dell'HDFS, perciò vi è comunicazione tra le istanze della Java Virtual Machine. Come per la modalità locale, anche quella pseudo-distribuita ha finalità di *debug*. In modo particolare è utile per monitorare l'utilizzo della memoria, i problemi nei flussi di input/output dell'HDFS e, soprattutto, l'interazione tra i *daemon*.

Vi è infine la modalità distribuita, impiegata quando la correttezza del programma è stata precedentemente appurata mediante opportuni test. In questa modalità diversi nodi formano un *cluster* e su ciascuno vengono avviati i *daemon*, che interagiscono tra loro tramite SSH.

2.4.2 Ottimizzazione delle performance

Le informazioni finora fornite dovrebbero permettere una comprensione generale dell'architettura del *framework*, ma ci sono alcuni altri elementi che vale la pena di approfondire.

Un punto cruciale nell'esecuzione di applicazioni MapReduce è il passaggio dal completamento dei *job* di *map* all'avvio di quelli di *reduce*. Da quanto finora detto, sembrerebbe che sia Hadoop a gestire autonomamente questo momento della elaborazione. In effetti questo è il comportamento di *default* del *framework*, che, in modo trasparente all'utente, raccoglie in una lista i valori intermedi appartenenti ad una stessa chiave ed avvia funzioni di *reduce*

sulle coppie generate. In realtà lo sviluppatore può alterare le azioni da eseguire al termine dei *job* di *map* per ottenere significativi miglioramenti nelle performance. Oltre alle classi Mapper e Reducer, che contengono, rispettivamente, le funzioni *map* e *reduce* e che devono tassativamente essere scritte, lo sviluppatore può definire un Combiner ed un Partitioner personalizzati. Se queste due classi non vengono configurate allora il *framework* adopera quelle di *default*, che impiegano funzioni di *hash* per ripartire i dati in modo omogeno tra i nodi.

Il Combiner ha un ruolo simile al Reducer, poiché aggrega i valori delle coppie intermedie. Mentre il Reducer compatta le coppie provenienti da nodi differenti, il Combiner agisce su un singolo TaskTracker, prima che invii nel *cluster* i risultati intermedi prodotti dalla funzione *map*. Il Combiner ed il Reducer, in generale, possono aggregare dati in modo differente, secondo le esigenze specifiche del caso d'uso. Aggregare i dati su ciascun nodo, prima che vengano trasmessi nel *cluster* può ridurre le operazioni di input/output migliorando il *throughput* generale.

A dimostrazione di ciò, viene ora arricchito il precedente esempio per l'indicizzazione di un documento introducendo un Combiner adatto, che sfrutta la proprietà associativa dell'addizione. Il Mapper precedente resta invariato, mentre il Combiner aggiunto si limita a sommare localmente il numero di occorrenze per ciascuna chiave (parola) prima di restituirlo in output. L'output del Mapper, perciò, non saranno più tante coppie (w_n, 1), ma coppie del tipo (w_n, #occorrenze_nella_riga). A fronte di questo cambiamento, il codice del Reducer deve essere lievemente modificato, poiché i valori associati ad una chiave non saranno più liste di 1, ma liste di interi probabilmente diversi tra loro. Il numero di occorrenze di una parola non sarà più dato dalla lunghezza della lista (un 1 per ogni match), ma dalla somma dei valori nella lista:

```
reduce(String w, List<Integer> values):
  int count = 0;
  for each value v in values:
    count = count + v;
  Emit(w, count);
```

Aggregando parzialmente le occorrenze in ogni TaskTracker, il numero di coppie immesse nella rete diminuisce drasticamente. Senza un Combiner, ogni riga del documento data in input ai Mapper produceva tante coppie quante le parole che conteneva; mediante il Combiner, invece, viene emessa una coppia in meno per ogni parola duplicata. Considerando che un documento contiene moltissime parole duplicate, il numero di coppie trasferite diminuisce sensibilmente aumentando le performance generali.

Resta da approfondire il Partitioner, che ha il compito di ripartire le coppie prodotte dai Mapper - ed eventualmente aggregate dai Combiner - tra i nodi del *cluster* che eseguiranno funzioni di *reduce*. Il Partitioner di *default* è l'HashPartitioner ed ha un ruolo fondamentale, poiché serve ad evitare *bottleneck* nella fase conclusiva dell'applicazione: se tutte le coppie intermedie, infatti, confluissero in un unico TaskTracker esso impiegherebbe una grande quantità di tempo per completare la fase di *reduce*, nullificando i benefici guadagnati dell'elaborazione in parallelo nella fase di *map*. Un Partioner preposto alla suddivisione di chiavi di tipo stringa, ad esempio, potrebbe suddividerle in base alla loro lunghezza o al loro primo carattere.

Altre due caratteristiche peculiari di Hadoop meritano un approfondimento: la **Streaming API** e la **DistributedCache**.

Hadoop può interagire con altri linguaggi attraverso una API generica, chiamata Streaming. Questa è particolarmente utile per la scrittura di semplici e brevi programmi MapReduce che possono essere sviluppati più facilmente in linguaggi di *scripting* che non hanno bisogno di librerie Java. Hadoop Streaming interagisce con gli altri programmi usando il paradigma Unix: l'input arriva dallo *STDIN* ed è direzionato in uscita verso lo *STDOUT*. I comandi Unix funzionano con il *framework*, che permette di impiegarli come funzioni *map* o *reduce*. In questo modo è possibile utilizzare, ad esempio, *cut*, *uniq*, *wc* oppure *script* in Python o PHP come parametri nella configurazione dei *task* di Hadoop.

Il seguente esempio mostra un codice in Python della funzione *map* per un *job* di indicizzazione.

```
#!/usr/bin/env python

import sys

for line in sys.stdin:
  line = line.strip()
  keys = line.split()
  for key in keys:
    value = 1
    print( "%s\t%d" % (key, value))
```

Una ultima caratteristica di Hadoop meritevole di essere illustrata è DistributedCache, un semplice ma utile meccanismo che consente di replicare automaticamente alcuni file tra tutti i nodi del *cluster*. DistributedCache viene solitamente impiegato per dotare all'accensione tutte le macchine del *cluster* di file contenenti "dati di *background*", spesso richiesti dai Mapper. Può essere utilizzato, ad esempio, per distribuire file di configurazione, *script*

o documenti contenenti metadati. Nell'ormai noto esempio di indicizzazione del testo, ad esempio, la funzione di *map* potrebbe necessitare di un *tokenizer* per dividere una stringa in parole. Se il *tokenizer* scelto fosse incluso in una libreria esterna, ad esempio, essa potrebbe essere distribuita a tutti i nodi del *cluster* tramite la funzionalità di DistributedCache. Tutte le funzioni *map* potrebbero, pertanto, utilizzare quella libreria, perché certamente presente sul nodo corrente.

Capitolo 3

Memorizzazione: database NoSQL

3.1 Origini delle tecnologie

Memorizzare negligentemente *big data* è il presupposto fondamentale per poterli in seguito elaborare efficientemente. Come detto nel secondo capitolo, il problema che sorge in fase di memorizzazione riguarda non soltanto lo spazio fisico che essi richiedono - col passare del tempo gli *hard disk* diventano sempre più capienti ed economici - ma anche le modalità con cui essi vengono archiviati. Una scelta progettuale sbagliata, infatti, può comportare una incapacità di recuperare questi dati velocemente, causando problemi di entità variabile col contesto d'uso (si pensi alla criticità nei sistemi *real-time* o a *bottleneck* in fase di lettura). Quando bisogna trattate *big data* è richiesto un consistente sforzo ingegneristico per sviluppare *ex novo* soluzioni che ripartiscano l'onere di memorizzazione tra diverse macchine, in un modo simile a quanto illustrato con l'Hadoop File System nel paragrafo 2.2. Infatti, sebbene sia facile aumentare la capacità di memorizzare dati comprando nuovi *drive*, risulta comunque impensabile concentrare tutto questo potenziale su un unico *server*.

L'esigenza di ripartire il carico di lavoro senza degradare nelle performance ha reso inadatti i database tradizionali, che utilizzano il modello relazionale per la strutturazione dei dati. Quando questi database sono stati progettati negli anni '70, infatti, Internet era ancora in uno stato embrionale: non esistevano *social network*, *smartphone* e *big data* e, soprattutto, non si avvertiva l'esigenza di sviluppare sistemi che potessero scalare. Ma il *Web 2.0* ha generato nuovi bisogni che i database NoSQL hanno provato a soddisfare.

Il termine "*NoSQL*" include un'ampia gamma di tecnologie ed architettu-

re per i dati, sviluppate in risposta all'aumento del volume delle informazioni e della frequenza con cui queste sono lette. È stato usato per la prima volta da Carlo Strozzi nel 1998 [26] per definire il suo database che non esponeva la tipica interfaccia di interrogazione tramite Structured Query Language (SQL). Come egli stesso affermò, sarebbe stato più appropriato definire il database "*NoREL*", dal momento che la differenza principale del suo sistema rispetto ai precedenti era nel modello di memorizzazione dei dati e non tanto nel linguaggio di interrogazione, l'SQL. Col passare degli anni, tuttavia, il termine NoSQL si è affermato e viene ora impiegato per etichettare i database che non adoperano il modello relazione, a prescindere dal linguaggio delle *query*. È possibile trovare, pertanto, database NoSQL interrogabili sia tramite API proprie sia tramite SQL; un discorso analogo vale per i database relazionali. Nella grande maggioranza dei casi, tuttavia, i database NoSQL espongono API proprie, mentre quelli relazionali supportano l'SQL.

Per questi motivi l'acronimo viene spesso risolto come "Not Only SQL", ad indicare, più che una tecnologia in particolare, una scuola di pensiero che propone un nuovo approccio alla gestione di grandi dati ed alla progettazione di database.

3.2 Confronto con database relazionali e NewSQL

I database NoSQL non sono stati progettati per sostituire definitivamente quelli relazionali, ma per offrire ai progettisti una valida alternativa da considerare durante la progettazione di sistemi che coinvolgono grandi moli di dati.

I database relazionali, infatti, sono tuttora adatti per molte applicazioni e spesso incontrano le attuali esigenze di business. Inoltre sono semplici da utilizzare e sono supportati da un vasto ecosistema di strumenti creati appositamente per loro, come *tool* di analisi e visualizzazione, di ottimizzazione e di gestione. Esistendo in commercio da molti anni hanno anche potuto beneficiare di una più lunga evoluzione ed il risultato sono sistemi robusti e performanti, a lungo collaudati da milioni di utilizzatori nel corso del tempo. Per lo stesso motivo sono conosciuti da un più ampio gruppo di persone, che ne padroneggia le *best practice*, rendendo semplice, per le aziende, la ricerca di personale qualificato. I database NoSQL, invece, sono ancora poco diffusi, come si può osservare in tabella 3.1, ed il numero di persone con *skill* in questo campo è ancora ridotto, nonostante la domanda del mercato sia in continua crescita [21].

Rank	DBMS	Tipo
1	Oracle	Relazionale
2	MySQL	Relazionale
3	Microsoft SQL Server	Relazionale
4	PostgreSQL	Relazionale
5	MongoDB	Orientato ai documenti
6	DB2	Relazionale
7	Microsoft Access	Relazionale
8	SQLite	Relazionale
9	Cassandra	A colonna
10	Sybase ASE	Relazionale
11	Solr	Motore di ricerca
12	Redis	Coppie chiave-valore
13	Teradata	Relazionale
14	FileMaker	Relazionale
15	HBase	A colonna
16	Elasticsearch	Motore di ricerca
17	Informix	Relazionale
18	Memcached	Coppie chiave-valore
19	Hive	Relazionale
20	Splunk	Motore di ricerca

Tabella 3.1: diffusione dei DBMS aggiornata a settembre 2014 [11].

Confrontare i database relazionali con quelli NoSQL dal punto di vista economico non è rilevante: per entrambi è possibile trovare soluzioni a pagamento ed altre gratuite. Allo stesso modo è possibile individuare opzioni *open-source* o proprietarie.

Tutti i vantaggi dei database relazionali elencati, tuttavia, risultano di poco conto quando l'applicazione richiede di scalare e di maneggiare *dataset* in evoluzione, che non aderiscono a schemi prefissati. I database NoSQL, nonostante si siano cominciati a diffondere solo dagli anni 2000, sono dotati di un architettura che garantisce, oltre a buone performance, flessibilità e scalabilità, a fronte di compromessi accettabili.

3.2.1 Flessibilità del modello dei dati

I database relazionali sono progettati per trattare dati strutturati, ovvero dati aderenti ad uno schema predefinito, in cui le informazioni sono suddivise in campi, ciascuno dotato di un tipo. Questi dati sono perciò facilmente memorizzabili in tabelle, contenenti anche milioni di righe.

I database NoSQL, invece, non richiedono la definizione a priori di uno schema fisso per una tabella, comune a tutti i dati che vi apparteranno; per questo motivo vengono definiti *schema-less*. Questa caratteristica è ideale per uno sviluppo agile, con rapide iterazioni e aggiornamenti frequenti del codice e del sistema. Come sarà spiegato nel dettaglio in seguito, ogni oggetto memorizzato nel database può avere dei propri campi, non necessariamente condivisi con gli altri. Durante il ciclo di vita del database, pertanto, a ciascun oggetto potranno essere aggiunti o rimossi campi all'occorrenza con grande facilità.

A differenza dei database relazionali, inoltre, consentono di memorizzare, oltre ai dati strutturati, anche dati semi-strutturati (come i file XML o JSON) e non strutturati (file multimediali o testi). I database NoSQL, infatti, non utilizzano un modello di dati statico che prevede la memorizzazione di ogni record come una tupla di una tabella, ma modelli dinamici e variabili.

Nei database relazionali la struttura delle tabelle può essere modificata, ma l'operazione è sconsigliata poiché particolarmente esosa, specialmente quando applicata su tabelle con milioni di righe. Questa operazione, inoltre, può essere effettuata solo mentre il database è *offline*, comportando ovvi problemi.

3.2.2 Scalabilità delle architetture

La maggiore differenza tra i database NoSQL e quelli relazionali, però, riguarda la loro capacità di scalare, ovvero la capacità di gestire efficientemente carichi di lavoro crescenti senza degradare nelle performance, adattando dinamicamente la struttura del sistema alla crescita di richieste. Prima di analizzare i due tipi di database in relazione alla loro capacità di scalare, però, risulta conveniente illustrare i due tipi di scalabilità di cui un generico sistema può beneficiare: scalabilità verticale e scalabilità orizzontale.

Un sistema composto da uno o più nodi interconnessi è scalabile verticalmente quando permette l'aggiunta di risorse ad un suo nodo con semplicità. In questa maniera il nodo può sopportare un carico di lavoro maggiore incrementando le performance generali del sistema. Per far scalare verticalmente un sistema è quindi sufficiente dotare un nodo di componenti più potenti, per esempio un processore o la memoria RAM.

Viceversa, un sistema è scalabile orizzontalmente quando è possibile aggiungergli con semplicità nuovi nodi, sui quali sarà ripartizionato il carico di lavoro. Aumentando il numero di nodi interconnessi e bilanciando le richieste tra di essi, infatti, il sistema aumenta la sua capacità di far fronte ad un numero maggiore di richieste. Le macchine che saranno aggiunte come nuovi

nodi della rete possono essere uguali a quelle preesistenti oppure possedere componenti diversi, in base alle caratteristiche del sistema.

I due modelli di scalabilità illustrati presentano dei *tradeoff*. La scalabilità verticale è semplice da ottenere e non richiede la configurazione specifica di nuovi nodi, ma incontra dei limiti fisici e può risultare tutt'altro che economica. Non sempre, infatti, è possibile migliorare i componenti di un nodo e molto spesso farlo richiede l'acquisto di unità molto costose. I componenti sostituiti, inoltre, diventano inutili e risulta necessario cercare di contenere le spese provando ad utilizzarli in altri contesti.

La scalabilità orizzontale, viceversa, richiede uno sforzo iniziale di configurazione, poiché bisogna collegare la nuova macchina al sistema in attività per permetterle di rispondere a nuove richieste (si pensi all'aggiunta di nodi per Hadoop). Risulta però meno dispendiosa perché consente il riutilizzo di molte macchine economiche, non necessariamente di alta fascia. Sistemi scalabili orizzontalmente sono infatti spesso costituiti da *commodity hardware*, macchine di modeste performance, recuperate da precedenti impieghi. La scalabilità orizzontale è un requisito fondamentale per la realizzazione di supercomputer, *cluster* di migliaia di nodi, con capacità di elaborazione pari a decine di petaFLOPS.

I database relazionali scalano verticalmente con grande semplicità, mentre per ottenere scalabilità orizzontalmente è richiesto un significativo sforzo ingegneristico aggiuntivo. I database relazionali, infatti, non sono progettati per effettuare *query* in modo distribuito: coordinare l'esecuzione di interrogazioni tra più macchine e garantire consistenza è perciò un onere dello sviluppatore.

I database NoSQL, invece, possono scalare orizzontalmente con grande facilità, perché quando sono stati sviluppati a questa capacità è stata data molta priorità, in previsione della grande mole di informazioni che essi avrebbero dovuto memorizzare. L'architettura risultante consente di beneficiare automaticamente di nuove macchine (*commodity server*, istanze *cloud*, ecc.) suddividendo i dati tra i nodi e coordinando l'esecuzione di interrogazioni tra di essi, in modo trasparente all'utente. La scalabilità orizzontale permette anche una maggiore replicazione dei dati, utile come protezione da malfunzionamenti e guasti.

3.2.3 Limiti dei database NoSQL

La flessibilità e la scalabilità dei database NoSQL è raggiunta accettando dei compromessi di cui quelli relazionali ne sono scevri.

Una prima fondamentale differenza riguarda le garanzie che la base di dati può dare durante l'esecuzione di transazioni, sequenze di operazioni suc-

cessive che possono concludersi correttamente (se e solo se tutte hanno buon esito) oppure no. In caso di successo (segnalato con l'istruzione di *commit*), le modifiche fatte allo stato del database durante la transazione devono diventare permanenti o persistenti; in caso contrario, invece, il sistema deve essere in grado di ripristinare lo stato in cui era prima dell'avvio della transazione (esecuzione del *rollback*). Un database dovrebbe essere in grado di garantire l'esecuzione di transazioni dotate di un certo insieme di proprietà, comunemente noto come ACID (*Atomicity*, *Consistency*, *Isolation*, *Durability*). In modo particolare le transazioni dovrebbero essere consistenti, ovvero garantire che i vincoli del database non vengano violati durante l'esecuzione delle operazioni e riuscire a vedere correttamente i risultati delle precedenti operazioni conclusesi correttamente. I database tradizionali possono essere configurati per offrire forte consistenza, mentre quelli NoSQL generalmente offrono solo eventuale consistenza. Ogni prodotto, in modo particolare, offre delle proprie parziali garanzie, da studiare con attenzione in fase di scelta del database NoSQL. Ci sono, infatti, contesti in cui il completamento simultaneo di transazioni su tutti i nodi del sistema è fondamentale ed altri in cui è assolutamente irrilevante, come nel caso della applicazioni riguardanti i *social media*, dove le attività degli utenti non sono quasi mai collegate e non impongono particolari vincoli.

Le tecnologie distribuite come i database NoSQL, inoltre, possono soddisfare per loro natura al massimo due delle seguenti proprietà contemporaneamente in virtù del teorema di Brewer (teorema CAP):

- **coerenza** (tutti i nodi vedono gli stessi dati nello stesso momento);
- **disponibilità** (la garanzia che ogni richiesta riceva una risposta su ciò che sia riuscito o fallito);
- **tolleranza di partizione** (il sistema continua a funzionare nonostante arbitrarie perdite di messaggi).

Come nel caso delle garanzie ACID, ogni prodotto ha una propria politica di gestione dei dati e decide di fare a meno di una delle tre proprietà elencate. In generale, però, queste tecnologie offrono le garanzie sintetizzate nell'acronimo BASE (*Basically Available*, *Soft State*, *Eventual consistency*), rilassando i vincoli sulla consistenza. Anche questo è un fattore da considerare in fase di scelta del database NoSQL.

Per i database relazionali *standalone*, invece, non vale il teorema di Brewer. Questo permette loro di garantire contemporaneamente tutte le tre proprietà e fornisce loro un altro punto di forza.

Per superare i limiti illustrati una nuova classe di *database management system* (DBMS) è in corso di sviluppo. Si tratta dei database NewSQL, che

hanno come obiettivo quello di preservare la capacità di scalare dei database NoSQL mantenendo le garanzie ACID dei database relazionali. Questi nuovi sistemi solitamente supportano il modello di dati relazionali ed il linguaggio SQL per le interrogazioni, ma possono essere realizzati in modi molti diversi. Un esempio di database NewSQL è Spanner, progettato da Google e ideato per sopperire alla mancanza di transazioni in BigTable, il suo predecessore. Google F1 è il DBMS sviluppato su Spanner ed è impiegato internamente nell'infrastruttura della Google Cloud Platform (vedasi paragrafo 4.2). Le architetture dei database NewSQL sono molto variegate, infatti è possibile trovare semplici motori di database ottimizzati per scalare o *cluster* di nodi che non condividono alcun dato, con propri meccanismi di interrogazione e flussi di controllo.

3.3 Tassonomia

Come detto nel primo paragrafo del capitolo, i database NoSQL non condividono lo stesso modello di dati, ma è possibile individuare quattro macrocategorie per provare a classificarli: database con coppie chiave-valore, orientati ai documenti, a colonna ed a grafo. Ciascuna categoria verrà analizzata da due prospettive: il modello di memorizzazione impiegato (strutture dati logiche o fisiche) ed il modello di interrogazione offerto all'utilizzatore (come ed a che prezzo i dati sono recuperati). Questo tipo di analisi va condotta attentamente ogni volta che si vuole utilizzare una soluzione di tipo NoSQL nel *core* di un sistema.

3.3.1 Database con coppie chiave-valore

I database con coppie chiave/valore funzionano similmente ad una grande tabella *hash*, in cui ad una chiave viene assegnato un valore. Poiché questi valori abbinati alle chiavi possono essere di diversi tipi (stringhe, JSON, BLOB, ecc.) il database risulta molto flessibile.

Le chiavi possono essere specificate esplicitamente dall'utente oppure generate automaticamente dal sistema. Ad ognuna di esse corrisponde un puntatore ad un particolare oggetto e spesso vengono raccolte in *bucket*, gruppi logici di chiavi, che però non hanno conseguenze sull'ordinamento fisico dei dati.

Le performance di questi sistemi, in virtù della loro semplice architettura, sono facilmente incrementate attraverso l'utilizzo di meccanismi di cache del *mapping* tra le chiavi ed i rispettivi valori. Con riferimento al teorema di Brewer, spesso questi database mancano di consistenza ed il loro mo-

dello estremamente semplice non offre molte delle funzionalità tipiche dei database tradizionali (atomicità della transazioni o consistenza quando transazioni multiple vengono eseguite contemporaneamente) che vanno invece implementate nell'applicazione.

Con la crescita costante del volume dei dati bisogna inoltre sviluppare un meccanismo per poter generare sempre nuove chiavi in modo coerente. Questo requisito non va sottovalutato poiché gli oggetti sono interrogabili unicamente attraverso le loro chiavi: i valori ad esse abbinati sono oscuri al sistema.

Alcuni dei più importanti database di questo tipo sono Riak, DynamoDB e Redis.

Rank	DBMS
1	Redis
2	Memcached
3	Riak
4	DynamoDB
5	Ehcache
6	Hazelcast
7	SimpleDB
8	Berkeley DB
9	Coherence
10	Oracle NoSQL

Tabella 3.2: diffusione dei database con coppie chiave-valore, aggiornata a settembre 2014 [11].

3.3.2 Database orientati ai documenti

I database orientati ai documenti raggruppano le informazioni semanticamente collegate tra loro in documenti, oggetti simili a quelli cui si è abituati nel paradigma di programmazione *object-oriented*. Per certi versi è possibile immaginare i documenti come una raccolta di coppie chiave-valore dei database precedentemente illustrati. I documenti, spesso codificati in file XML, JSON o BSON (codifica binaria di un oggetto JSON), contengono uno o più campi di un certo tipo, come stringhe, date, *array*, sotto-documenti. I record non sono frammentati in tante colonne, ma conservati insieme. Ogni documento può contenere, però, campi diversi, mantenendo la proprietà di schemi dinamici, tipica dei database NoSQL; questa flessibilità è utile per modellare dati non strutturati e polimorfici o per poter far evolvere agevolmente un'applicazione.

Una differenza tra i database con coppie chiave-valore e quelli orientati ai documenti è che quest'ultimi spesso aggiungono dei metadati ai valori inseriti nei campi per permettere *query* sul contenuto degli attributi stessi. In generale queste tecnologie offrono la possibilità di eseguire *query* su un qualunque campo di un documento e di aggiornarne direttamente i valori, ma non di effettuare operazioni di *join*, che dovranno essere gestite dall'applicazione. Alcuni prodotti, infine, offrono funzionalità di indicizzazione per ottimizzare le *query*.

I prodotti più consolidati per questa tipologia di base di dati sono MongoDB e CouchDB.

Rank	DBMS
1	MongoDB
2	CouchDB
3	Couchbase
4	MarkLogic
5	RavenDB
6	Cloudant
7	GemFire
8	OrientDB
9	RethinkDB
10	Datameer

Tabella 3.3: diffusione dei database orientati ai documenti, aggiornata a settembre 2014 [11].

3.3.3 Database a colonna

I database a colonna sono stati sviluppati cercando di emulare l'architettura di BigTable [5] e perciò organizzano le informazioni in un modo simile a quello adoperato nella tecnologia di Google. Il risultato è che il modello di dati assomiglia ad un dizionario multidimensionale, sparso, distribuito e persistente.

Ad una entità del database, individuabile dalla sua chiave, sono abbinati dei valori memorizzati in colonne specifiche, identificate da un proprio qualificatore. Le entità non devono necessariamente possedere un valore per ogni qualificatore e possono liberamente aggiungere nuove colonne. Anche questa tipologia di database risulta molto flessibile, perché i dati non devono aderire necessariamente ad uno stesso modello prefissato e possono modificarlo a *runtime*.

I database a colonna memorizzano in celle adiacenti i valori appartenenti alla stessa colonna, diversamente da quelli relazionali che raggruppano i dati per righe. Più colonne possono essere poi raccolte in una *column family* (da qui la multidimensionalità del dizionario). Generalmente le colonne comprese in una stessa colomun family sono fisicamente memorizzate in celle vicine di memoria, per ottimizzare la scansione dei valori conservati nella base di dati.

La lettura e la scrittura dei dati vengono eseguite iterando su una colonna di una determinata *column family* per un insieme di chiavi, piuttosto che su una riga; per questo motivo i database risultano molto efficienti quando viene richiesto loro di recuperare i valori di uno stesso qualificatore, indipendentemente dal numero di righe nel database, perché questi valori sono adiacenti.

I database di questo tipo più impiegati sono Apache HBase e Apache Cassandra.

Rank	DBMS
1	Oracle
2	HBase
3	Accumulo
4	Hypertable
5	Sqrrl

Tabella 3.4: diffusione dei database a colonna, aggiornata a settembre 2014 [11].

3.3.4 Database a grafo

I database a grafo sono meno diffusi, ma particolarmente utili quando le informazioni da memorizzare sono collegate tra loro in virtù di alcune caratteristiche. Se i dati, infatti, presentano una struttura a grafo, con nodi ed archi, questa tipologia di database risulta adatta a contenerli.

L'utilizzo di una rete di relazioni, sebbene possa sembrare controintuitiva, può essere utile in molte applicazioni. Con un po' di ragionamento, infatti, è possibile impiegare questi database per applicazioni diverse dai *social network*, cui si è soliti pensare a primo acchito. Una conferma viene dalla grande popolarità che stanno raggiungendo (si veda la figura 3.1).

Sia i nodi che le relazioni possono essere arricchite con delle proprietà e gli archi possono essere orientati o bidirezionali.

L'attrattiva di questa soluzione risiede nella semplicità con cui possono essere rappresentate le relazioni tra entità di una applicazione. Per questo

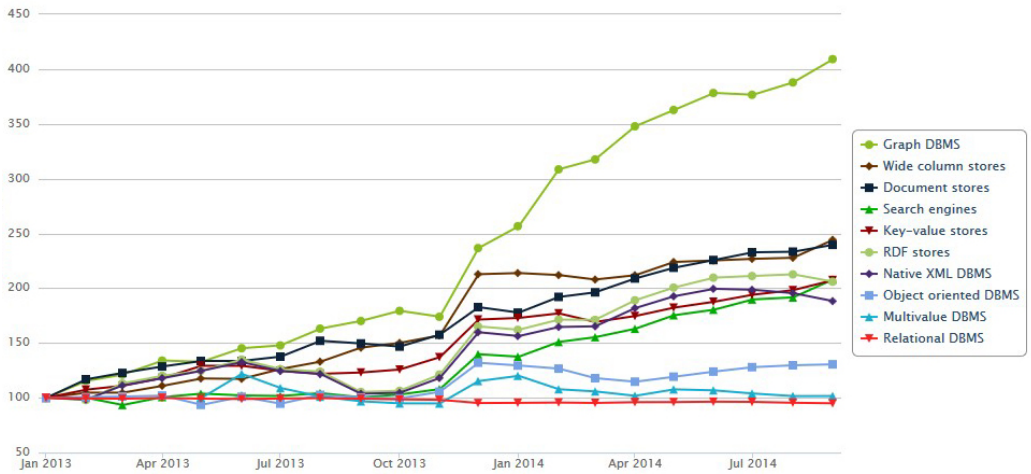

Figura 3.1: *trend* di popolarità delle tecnologie NoSQL [11].

motivo spesso sono impiegati in siti di *e-commerce* per costruire un grafo di utenti e prodotti acquistati. Tramite questo grafo è possibile riuscire a predire, con dei *recommender system* [17], quali nuovi prodotti un utente potrebbe essere interessato a comprare per fornire suggerimenti personalizzati.

I database a grafo possono essere interrogati per effettuare inferenze dirette o indirette sui dati memorizzati: verificare le relazioni tra i nodi, infatti, non richiede il completamento di costose operazioni di *join*. Le analisi sui tipi di relazioni sono molto efficienti, a differenza degli altri tipi, molto meno ottimizzati. Aggiungere o rimuovere relazioni dal grafo risulta banale.

Prodotti attualmente impiegati in molti casi d'uso sono Neo4j, Titan e InfiniteGraph.

Rank	DBMS
1	Neo4j
2	Titan
3	OrientDB
4	Sparksee
5	Giraph
6	ArangoDB
7	InfiniteGraph
8	Sqrrl
9	InfoGrid
10	FlockDB

Tabella 3.5: diffusione dei database a grafo, aggiornata a settembre 2014 [11].

Capitolo 4

Cloud computing

4.1 Caratteristiche e modelli di servizi offerti

Nel paragrafo 1.5 sono stati evidenziati i problemi che si devono solitamente affrontare durante la gestione di *big data*, ovvero quelli riguardanti le modalità e l'ubicazione dei processi di elaborazione e memorizzazione. Nel capitolo 2 è stato presentato Hadoop, un *framework* progettato per elaborare grandi *dataset* in modo distribuito, mentre nel capitolo 3 sono stati approfonditi i database NoSQL, tecnologie scalabili in grado di memorizzare e recuperare efficientemente molte informazioni. Questi due strumenti forniscono modalità adatte per la gestione di grandi moli di dati, ma ancora nessuna soluzione è stata proposta per risolvere il problema dell'ubicazione di questi processi. Nel corso di questo capitolo, pertanto, verranno analizzate le piattaforme di *cloud computing*, degli strumenti per processare dati in remoto, che spesso si rivelano più adeguati dei *server* locali.

Questi servizi si sono sviluppati soprattutto negli ultimi dieci anni, in risposta alle esigenze di molte imprese di decentralizzare la fase di elaborazione dei dati. A fronte del notevole aumento del volume di dati raccolti, infatti, numerose aziende hanno realizzato di possedere infrastrutture inadatte o troppo costose (spese di manutenzione, *upgrade*, personale, energia elettrica, ecc.) per soddisfare le richieste dei propri sistemi. Diversi colossi informatici hanno ritenuto questa situazione una buona occasione per impiegare con maggiore produttività i loro *data center* e per aggiungere una nuova cospicua fonte di guadagno al loro business. Sono nate così le piattaforme di *cloud computing*, gruppi di servizi noleggiabili da aziende o privati secondo dettagliate politiche di prezzi ed utilizzabili nella forma di *web service*. Usufruendo delle potenti infrastrutture di aziende come Amazon, Google, IBM e Microsoft le imprese ricevono la garanzia di elaborare e memorizzare le

informazioni con grande velocità ed in modo sicuro (per via dei processi di criptografia e replicazione dei dati).

Integrare questi servizi nelle applicazioni preesistenti risulta spesso semplice, per via delle API in diversi linguaggi di programmazione che vengono fornite agli sviluppatori. Beneficiando delle infrastrutture di queste imprese, i sistemi risultano scalabili e godono di disponibilità pressoché continua. Le interfacce web, inoltre, rendono la gestione di questi servizi ancora più intuitiva.

Le piattaforme di *cloud computing* differiscono per i servizi contenuti, le politiche dei prezzi ed altre caratteristiche minori, ma, generalmente, condividono i modelli di servizi offerti: *Infrastructure as a Service* (IaaS), *Platform as a Service* (PaaS) e *Software as a Service* (SaaS).

Col modello IaaS il *cloud provider* offre macchine fisiche o, più spesso, virtuali con *hardware* configurabile oppure altre risorse, quali *load balancer*, indirizzi IP, VLAN, *server* per lo *storage* di file, ecc. Queste risorse remote possono essere utilizzate per avviare macchine con sistema operativo personalizzato - tramite immagini di dischi preconfigurate - con cui eseguire qualunque tipo di computazione (ad esempio, è possibile utilizzarle per un *cluster* di Hadoop). In questo modo gli utenti possono disporre di un numero di computer virtualmente illimitato con cui far fronte a qualunque tipo di esigenza. Collegandosi alle proprie macchine, spesso tramite il protocollo di rete SSH, è possibile infatti installare ed avviare su di esse dei propri programmi e raccogliere i risultati delle analisi effettuate nei *data center* del *cloud provider*.

Solitamente i costi di questi servizi sono proporzionali alla quantità di risorse utilizzate nel tempo e, pertanto, è consigliabile condurre analisi sull'effettiva economicità della soluzione. Per tale motivo spesso le macchine vengono avviate, impiegate per qualche elaborazione e subito spente, al fine di non pagare intervalli di inattività: questo approccio risulta praticabile per via dei ridotti tempi di accensione e spegnimento delle istanze remote.

Nel modello di PaaS i fornitori offrono delle risorse di elaborazione già configurate e pronte all'uso. Tipicamente vengono messe a disposizione macchine con un determinato sistema operativo e diversi *software* preinstallati: ambienti d'esecuzione per linguaggi di programmazione (ad esempio la Java Virtual Machine), database, *web server* e strumenti di varia natura. Queste macchine possono essere utilizzate nel *core* di un proprio sistema e, se adeguatamente selezionate, si rivelano molto adatte a sopportare grandi carichi di lavoro. Un servizio PaaS, ad esempio, potrebbe gestire le richieste fatte ad un sito web, ad una applicazione per *smartphone*, ad un servizio di telefonia, ecc.

Diversi *provider*, come Google e Microsoft offrono anche delle funzionalità di *load balancing*: all'aumentare delle richieste il *provider* alloca dinamicamente nuove risorse per non far degradare il sistema nelle performance, senza che l'utilizzatore del servizio debba intervenire manualmente. I sistemi sono quindi capaci di scalare automaticamente e, pertanto, questi servizi si adattano bene ad ambienti *real-time*.

Infine, col modello SaaS, i *provider* noleggiano i propri *software*, nascondendo agli utilizzatori i dettagli sulle infrastrutture sottostanti e sollevandoli dall'onere di configurare ed avviare delle macchine remote. Il modello SaaS viene talvolta definito come "*software on-demand*", proprio perché gli utenti possono fare uso delle applicazioni offerte nella piattaforma in relazione alle loro esigenze. È compito del *cloud provider* assicurarsi che i prodotti siano sempre disponibili e funzionanti, bilanciando i carichi di lavoro e manutenendo ed evolvendo il *software* (i cambiamenti sono trasparenti agli utenti). In questo modo gli investitori possono utilizzare soluzioni già in commercio ed evitare spese per l'acquisto di *software* e per la loro manutenzione.

Tutti i modelli, tuttavia, presentano lo svantaggio di dover memorizzare i dati degli utenti nei *server* del *cloud provider*, esponendoli al rischio di accessi non autorizzati. Per ovviare a questi problemi i fornitori dei servizi solitamente offrono diverse garanzie (i file vengono divisi in più parti, criptati e sparsi in *data center* diversi), ma gli utenti possono anche usufruire di servizi di criptazione di terze parti per assicurarsi maggiore protezione.

La piattaforma per l'analisi di *big data* sviluppata per questo studio necessitava di servizi di *cloud computing* per potere adoperare *framework* come Hadoop e per beneficiare della scalabilità garantita dal *cloud provider*. Per tale motivo essa presenta una architettura ibrida, con diverse componenti *cloud*. È stato comunque dimostrato [1] che le soluzioni *cloud* costituiscono, dal punto di vista economico, una valida alternativa a quelle *bare-metal* per molti casi d'uso e sono pertanto suggerite anche in assenza di necessità simili a quelle di questo caso di studio.

La Google Cloud Platform è stata preferita alle altre piattaforme per la varietà di tecnologie nel suo portfolio, per la superiorità nelle prestazioni rispetto ai concorrenti [22] [34], per la sua semplicità di utilizzo, per la molteplicità di strumenti offerti agli sviluppatori (*plugin* per IDE, *script* di configurazione e *deploy*, ecc.) e per la sua effettiva economicità (per provare i servizi ed eseguire tutti gli esperimenti della piattaforma sono stati consumati in totale circa 12$).

4.2 Google Cloud Platform

La Google Cloud Platform (GCP) è una raccolta di alcune delle tecnologie sviluppate da Google ed attualmente utilizzate internamente dall'azienda per garantire il corretto funzionamento dei suoi prodotti, quali Gmail, YouTube, il motore di ricerca ed altri.

La piattaforma è stata resa disponibile al pubblico a partire dal 2008 con l'obiettivo di noleggiare ad aziende e privati gli stessi strumenti adoperati da Google nella forma di *web service*. Con essa Google è entrata nel panorama dei *cloud provider*, in diretta competizione con Amazon Web Services (AWS), Azure di Microsoft, SoftLayer e BlueMix di IBM, ecc.

Nel corso degli anni la piattaforma di Google è stata evoluta, migliorando i suoi prodotti, introducendo nuovi servizi, fornendo ulteriori API agli sviluppatori ed aggiornando le politiche dei prezzi, conformemente ai competitori.

Secondo le modalità tipiche dei servizi di *cloud computing*, gli interessati alle tecnologie offerte nel portfolio possono utilizzare i prodotti di cui hanno bisogno dietro un opportuno compenso. Noleggiando le proprie infrastrutture ed i propri *software*, pertanto, Google si assicura una cospicua fonte di guadagno (come anticipato nel paragrafo 1.4) e permette agli sviluppatori di adoperare potenti strumenti, adatti ad elaborare efficientemente *big data*.

I servizi inclusi nella piattaforma sono di natura eterogenea e rientrano nei modelli di IaaS, PaaS e SaaS. Tutti beneficiano di diverse qualità come scalabilità e disponibilità, assicurano protezione e replicazione dei dati trattati e garantiscono elevate performance, indipendentemente dal numero di richieste cui devono rispondere. Le tecnologie offerte dalla Google Cloud Platform sono, in generale, orientate all'elaborazione o alla memorizzazione delle informazioni; talvolta costituiscono servizi specifici per le applicazioni. Di seguito viene presentato un elenco di alcuni dei prodotti principali inclusi nella piattaforma:

- Compute Engine: IaaS che permette la creazione di una o più macchine virtuali con *hardware* configurabile dall'utente. Può essere utilizzata per creare *cluster* di computer, che saranno collegati tra loro attraverso la fibra ottica privata di Google;

- App Engine: PaaS utilizzata per sviluppare ed ospitare applicazioni nei *data center* di Google. L'SDK supporta diversi linguaggi di programmazione ed è compatibile con i maggiori *framework*. Google si occupa di gestire le richieste fatte alle applicazioni utilizzanti App Engine, amministrare i database e bilanciare i carichi di lavoro;

- Cloud SQL: database relazionale MySQL che gestisce, in modo trasparente all'utente, la replicazione ed il *backup* dei dati, l'aggiornamento del sistema ed il recupero automatico in caso di errori. È capace di processare efficientemente miliardi di tuple e può essere usato dai tradizionali strumenti di gestione di database relazionali;

- Cloud Storage: servizio per la memorizzazione di dati, simile ai tradizionali servizi di *file hosting*;

- Cloud Datastore: database NoSQL *schema-less*, ideale per memorizzare dati non strutturati. Supporta transazioni ACID e *query SQL-like*. Google provvede allo *sharding* ed alla replicazione dei dati;

- Big Query: IaaS che permette di analizzare grandi *dataset* in pochi secondi eseguendo *query SQL-like*, ideale come strumento per acquisire in tempo reale comprensione su *big data*.

Tra i prodotti presentati alcuni sono stati utilizzati per la piattaforma oggetto di questo lavoro e saranno, pertanto, presentati nel dettaglio.

4.2.1 Google Compute Engine

Google Compute Engine (GCE) è uno dei servizi principali della GCP che permette agli utenti di avviare e controllare remotamente macchine virtuali (VM) in esecuzione nei *data center* di Google.

Gli utenti hanno la possibilità di configurare le macchine che saranno avviate e, una volta attive, utilizzare queste istanze per effettuare qualsiasi tipo di operazione. Nello specifico, essi devono specificare il sistema operativo ed il tipo di macchina. GCE è compatibile con molti sistemi operativi e distribuzioni, alcuni gratuiti, altri a pagamento. Alcuni tra quelli nativamente supportati sono Debian, CentOS, openSUSE, Red Hat, FreeBSD e Windows Server, ma viene anche offerta la possibilità di caricare immagini personalizzate, che meglio rispondono alle esigenze degli utilizzatori. Il tipo di macchina, invece, determina le specifiche fisiche della VM, come il quantitativo di memoria disponibile o il numero di *core* virtuali. Il tipo deve essere scelto tra quelli offerti da Google, che, per comodità, raggruppa l'offerta in quattro categorie di macchine: *standard*, *high CPU*, *high memory*, *small*. Gli utenti possono pertanto scegliere, in relazione all'utilizzo che sarà fatto delle istanze, se avviare macchine con molta memoria o preferire una maggiore potenza di elaborazione oppure propendere per soluzioni economiche e con poche risorse. La tabella 4.1 riassume l'attuale proposta di Google e descrive i tipi di VM.

Tipo macchina	Numero CPU	Memoria (GB)	Prezzo ($/h)
n1-standard-1	1	3,75	0,077
n1-standard-2	2	7,50	0,154
n1-standard-4	4	15	0,308
n1-standard-8	8	30	0,616
n1-standard-16	16	60	1,232
n1-highmem-2	2	13	0,18
n1-highmem-4	4	26	0,36
n1-highmem-8	8	52	0,72
n1-highmem-16	16	104	1,44
n1-highcpu-2	2	1,80	0,096
n1-highcpu-4	4	3,60	0,192
n1-highcpu-8	8	7,20	0,384
n1-highcpu-16	16	14,40	0,768
f1-micro	1 (condivisa)	0.60	0,013
g1-small	1	1,70	0,0347

Tabella 4.1: offerta di Compute Engine. Una CPU virtuale è implementata come un singolo *hyperthread* su un processore Intel Sandy Bridge Xeon o Intel Ivy Bridge Xeon a 2.6GHz (due CPU virtuali corrispondono ad un intero *core* fisico). I prezzi sono relativi all'Europa.

Quando vengono spente dall'utente, le istanze di GCE sono distrutte e le risorse a loro dedicate vengono riutilizzate per nuove VM; con questo processo tutti i dati memorizzati sul disco dall'utente vengono perduti. Per risolvere questo problema, Google offre la possibilità di utilizzare dischi persistenti, risorse assimilabili agli *hard drive* comunemente collegati ai personal computer. Essi si presentano come la soluzione primaria per la memorizzazione di dati in modo persistente. Un disco può essere collegato e scollegato ad una VM secondo le preferenze dell'utente e può essere programmato per distruggersi o rimanere in funzione quando l'istanza cui è collegato viene spenta. Nel secondo caso i dati memorizzati al suo interno non vengono persi e l'utente può collegare il disco ad una nuova istanza. In fase di configurazione di un disco l'utente può scegliere se creare un disco *standard* oppure un SSD: come per i tipi di macchina, anche i dischi hanno performance e prezzi differenti.

Le macchine virtuali possono essere utilizzate indipendentemente oppure possono essere collegate tra loro per formare un *cluster*. Google mette a disposizione, ad esempio, degli *script* per effettuare il *deploy* di un *cluster* Hadoop avente caratteristiche impostate dall'utente (numero e tipologia di nodi, file da copiare sulle macchine all'avvio, *file system*, ecc.). Le macchine sono collegate tra loro attraverso la fibra ottica di Google e, pertanto, le

connessioni registrano basse latenze: il trasferimento di dati tra i nodi del *cluster* risulta molto veloce e GCE rappresenta una buona infrastruttura per il calcolo distribuito.

Tutte le istanze sono collegate ad Internet e possono essere collegate in una rete interna, eventualmente protetta con un *firewall*. Ciascuna di esse, inoltre, possiede un proprio indirizzo IP e può essere identificata tramite una risoluzione DNS.

Le risorse di GCE possono essere facilmente gestite attraverso lo strumento da riga di comando o tramite la *web console* degli sviluppatori della piattaforma (in entrambi i casi il protocollo SSH viene utilizzato per collegarsi alle istanze remote). I *software* possono poi interagire con le macchine utilizzando le RESTful API per diversi linguaggi (Java, Python, Ruby, .NET, Go, PHP, Objective C, ecc.). Esse solitamente utilizzano OAuth 2.0 per completare l'autenticazione e per integrare GCE con altri prodotti della piattaforma come Cloud Storage.

Google non garantisce, ad ogni modo, che le istanze siano in esecuzione per il 100% del tempo, pertanto è compito degli utenti assicurarsi che il servizio possa ripristinare il proprio stato a seguito di un guasto imprevisto; diversi accorgimenti vengono comunque suggeriti per progettare sistemi robusti.

4.2.2 Google Cloud Storage

Google Cloud Storage (GCS) rappresenta una delle tre soluzioni offerte nella piattaforma per la memorizzazione dei dati, insieme a Cloud SQL e Cloud Datastore. Mentre questi ultimi due prodotti costituiscono dei veri e propri database, GCS è un più semplice servizio di file *storage* nel *cloud*. Con esso gli utenti possono arrivare a memorizzare diversi terabyte di dati, pagando in relazione all'ammontare di dati trasferiti ed allo spazio fisico occupato (il prezzo è di circa 0.026$/GB/mese). I dati possono essere memorizzati in uno o più *data center* di Google (Stati Uniti, Unione Europea ed Asia), che garantisce agli utilizzatori elevate velocità di lettura, continua disponibilità e, soprattutto, scalabilità. Il servizio risulta, pertanto, anche adatto a distribuire in *download* diretto file molto grandi o molto richiesti.

GCS è una tecnologia di *storage* rivolta agli sviluppatori, poiché i file memorizzati possono essere utilizzati da altri prodotti della piattaforma di Google. Per questo motivo viene spesso utilizzato insieme ad altri servizi, ad esempio Compute Engine, App Engine e Big Query. Gli utenti che non sono interessati a sviluppare *software* possono, invece, utilizzare il più noto Google Drive, un più semplice servizio di *hosting*. Ad ogni modo, utilizzando

il Google Drive SDK, è possibile integrare i due servizi, per offrire agli utenti finali i file caricati su Cloud Storage.

In GCS tutti i dati vengono contenuti in un progetto, che consiste di un insieme di utenti abilitati a gestire i file, un insieme di API attivate ed impostazioni di fatturazione ed autenticazione. Ogni utente può gestire uno o più progetti e può creare, all'interno di un progetto, un numero arbitrario di *bucket*. Essi rappresentano i contenitori dei file e sono assimilabili a delle cartelle: tutti gli oggetti caricati in GCS devono essere inseriti in un *bucket*. A differenze delle cartelle, però, i *bucket* non possono essere annidati e non possono essere condivisi tra più progetti. Ogni *bucket* è caratterizzato da un identificativo univoco e permette di configurare i privilegi da assegnare agli utenti che vogliono accedervi. Gli oggetti di GCS non possono essere condivisi tra più *bucket* e possiedono due componenti: dati e metadati. I dati rappresentano il file da memorizzare, mentre i metadati sono delle coppie chiave-valore che descrivono delle qualità dell'oggetto.

GCS offre una forte *read-after-write consistency* per tutte le operazioni di *upload* e cancellazione. Questo significa che un oggetto appena caricato può subito essere scaricato, rimosso o ispezionato nei suoi metadati. Allo stesso modo risulta impossibile accedere ad un oggetto che è stato appena cancellato. Dal punto di vista della disponibilità, le operazioni su GCS sono atomiche: gli oggetti caricati sono accessibili solo se perfettamente integri. I file corrotti o parziali non possono essere letti, così come quelli ancora in fase di *upload*.

Come per Compute Engine, anche le risorse di Cloud Storage possono essere gestite in diversi modi: interfaccia da riga di comando, *web console* ed API (XML, JSON, ecc.). I *software* possono interagire con il servizio attraverso una delle librerie offerte da Google, disponibili per tanti linguaggi di programmazione. Anche GCS utilizza OAuth 2.0 per gestire i meccanismi di autenticazione e autorizzazione.

4.2.3 Google Datastore

Datastore è il database NoSQL di Google, progettato per memorizzare dati in forma non relazionale. Il prodotto appartiene al modello di servizi di Database-as-a-Service (DaaS) e può essere immediatamente utilizzato da chi lo noleggia. Il database, infatti, è completamente gestito da Google, che si occupa della manutenzione e degli aggiornamenti, così come della replicazione dei dati attraverso l'algoritmo Paxos e dello *sharding*.

Datastore memorizza i dati nella forma di oggetti, noti come **entità**, ciascuno dei quali è caratterizzato da un **tipo**, necessario per eseguire le *query*, da una **chiave** (o nome) che identifica univocamente l'oggetto e da

una lista di **proprietà**. Le proprietà sono attributi degli oggetti e sono costituite da un nome, un tipo (stringa, intero, booleani, riferimenti ad altre entità, ecc.) ed il valore.

Coerentemente con le caratteristiche dei prodotti NoSQL, Datastore è un database *schemaless*, poichè entità dello stesso tipo possono avere differenti proprietà e differenti entità possono avere proprietà con lo stesso nome ma tipi diversi. L'utente può comunque imporre vincoli di consistenza sull'insieme di proprietà di un tipo attraverso il codice della propria applicazione. Per questa flessibilità il prodotto è molto adatto a sviluppi agili, quando la struttura dei dati cambia ripetutamente.

Il database è stato sviluppato per scalare in relazione alle richieste che gli vengono inoltrate, consentendo alle applicazioni che lo utilizzano di ottenere elevate performance. Esso, inoltre, scala automaticamente anche in base alla quantità di dati memorizzati. La sua infrastruttura, infatti, permette di eseguire nello stesso tempo *query* su centinaia o milioni di entità. Per ottenere questa capacità il database fa uso di indici precostituiti, che rendono impossibili alcune operazioni come *join*, controlli di disuguaglianza su più proprietà o filtraggio dei dati in relazione ai risultati di una *sub-query*.

Tutte le operazioni sui dati vengono eseguite tramite istruzioni *SQL-like*. Più istruzioni possono essere inserite in un'unica transazione. Le transazioni offrono le garanzie ACID, assicurando integrità sui dati. Se una delle operazioni nella transazione fallisce, infatti, Datastore esegue automaticamente il *roll-back*. Se, invece, due transazioni provano a modificare contemporaneamente gli stessi oggetti allora la prima ad eseguire il *commit* avrà successo e tutte le altre falliranno, secondo il meccanismo di *optimistic concurrency*.

Gli utenti hanno la possibilità di instaurare relazioni gerarchiche tra le entità, dando vita a quelli che vengono definiti *ancestor path*. Queste relazioni possono essere usate per delimitare il raggio d'azione delle *query* ed influenzano il modo con cui vengono indicizzate le entità. Gli indici, infatti, sono tabelle contenenti entità in una sequenza specificata dalle proprietà ed, eventualmente, dagli antenati dell'oggetto. Essi vengono aggiornati incrementalmente, per riflettere i cambiamenti fatti alle entità, in modo che i corretti risultati delle *query* siano sempre immediatamente disponibili.

Come tutti i prodotti della Google Cloud Platform, è possibile interagire con Datastore in diversi modi. Il prodotto è utilizzabile tramite HTTP usando documenti JSON oppure tramite API in diversi linguaggi di programmazione. In aggiunta, Datastore offre una interfaccia dal web per gestire il database ed un servizio per lo sviluppo in locale, che non richiede il *deploy* sul servizio *cloud*.

Capitolo 5

Piattaforma cloud per analisi di big data

5.1 Contesto del lavoro

Le tecnologie presentate nei precedenti capitoli sono state impiegate per lo sviluppo di una piattaforma *cloud* per l'analisi di *big data* provenienti da *social network*, i cui componenti saranno illustrati nel corso di questo capitolo.

L'origine di tale strumento risiede nell'esigenza avvertita da Vox, Osservatorio Italiano sui Diritti, di monitorare le zone italiane dove omofobia, razzismo, discriminazione verso i disabili, misoginia ed antisemitismo sono maggiormente diffusi. L'ambizioso obiettivo del progetto, infatti, è lo sviluppo di una Mappa dell'Intolleranza, un mezzo capace di evidenziare i luoghi affetti da questi gravi problemi di discriminazione, attraverso l'analisi di *tweet* in lingua italiana geolocalizzati.

L'idea di questo strumento non è nuova: gli studenti della Humboldt State University in California hanno sviluppato nel 2013 la Hate Map (in figura 5.1), una mappa di calore interattiva in grado di mostrare le principali zone degli Stati Uniti da cui provenivano *tweet* razzisti, omofobi e contro le persone diversamente abili. Il loro lavoro ha consentito importanti osservazioni di carattere sociale: è stato riscontrato, ad esempio, che la maggior parte dei messaggi d'odio avevano origine in piccole città o aree rurali piuttosto che in grandi metropoli.

La Mappa dell'Intolleranza di Vox è pensata per offrire alle amministrazioni locali italiane uno strumento per scoprire le aree più a rischio, al fine di agire concretamente sul territorio per risolvere i problemi di discriminazione. Recenti statistiche dimostrano quanto importante sia operare attivamente in tale direzione: solo nel 2013, il 25% degli omosessuali è stato vittima di

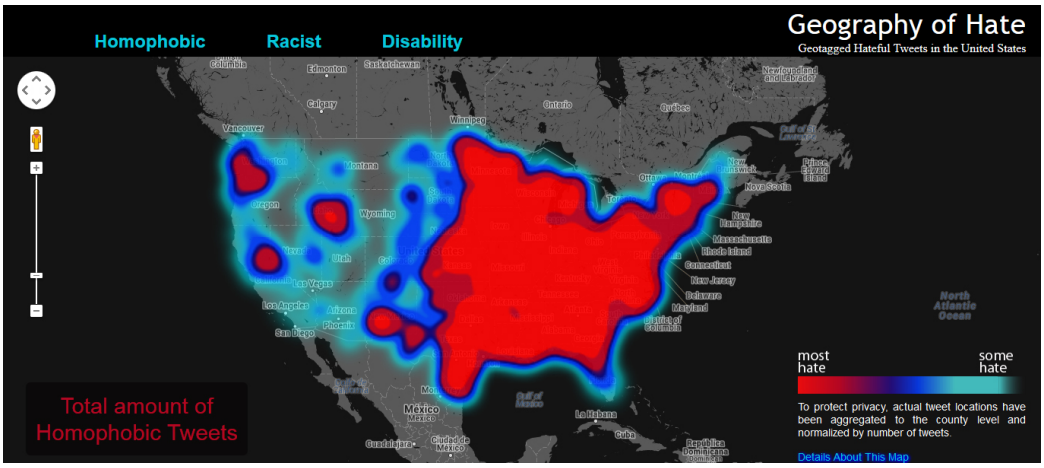

Figura 5.1: Hate Map degli Stati Uniti d'America[1].

violenza, 6.743.000 donne hanno subito abusi fisici o sessuali ed il 45% dei giovani si è dichiarato xenofobo o diffidente degli stranieri [10].

Il ruolo dei *social network* in questo contesto è essenziale, poiché essi permettono di veicolare ed alimentare sentimenti di odio verso il prossimo con grande facilità. Uno studio sistematico dei messaggi discriminatori pubblicati dagli utenti potrebbe, però, permettere di prevenire l'insorgere di episodi violenti e situazioni spiacevoli.

Il progetto avviato da Vox risulta essere tanto importante, quanto complesso. Fa emergere, infatti, diverse problematiche di natura linguistica e tecnologica alle quali diverse facoltà italiane stanno facendo fronte collaborando, al fine di produrre uno strumento affidabile e preciso.

La piattaforma che sarà ora descritta permette un importante passo avanti nella realizzazione del progetto della Mappa dell'Intolleranza, poiché si prefigge di essere l'infrastruttura portante del sistema ed uno strumento scalabile per la raccolta e l'analisi di grandi moli di dati.

5.2 Architettura della piattaforma

Nello sviluppo della piattaforma è stato seguito un approccio modulare, in linea con le *best practice* dell'ingegneria del *software*. Questa scelta progettuale ha portato allo sviluppo di un sistema dotato di moduli indipendenti tra loro, facilmente manutenibili e concepiti per portare a termine determinati compiti in modo autonomo (l'architettura è presentata in figura 5.2). Il *soft-*

[1] `http://users.humboldt.edu/mstephens/hate/hate_map.html`

ware beneficia, nel complesso, di diverse importanti qualità, tra cui spiccano **evolvibilità** e **scalabilità**. La prima risulta essenziale per una piattaforma costituente l'infrastruttura di un sistema complesso, che, verosimilmente, cambierà nel corso del tempo per rispondere a nuove esigenze. Nel suo *core*, infatti, la piattaforma è predisposta ad accettare nuovi moduli, con cui potrà effettuare operazioni attualmente non supportate, anche molto differenti tra loro. Il requisito di scalabilità, d'altronde, è imprescindibile per un *software* finalizzato alla gesione di *big data*, nonostante la dimensione dei *tweet* finora raccolti permettesse l'utilizzo di tecnologie tradizionali per lo *storage* ed il *processing*. In fase di sviluppo, invece, si è supposto di dover elaborare più informazioni di quelle possedute e sono stati presi tutti gli accorgimenti del caso, impiegando tecnologie ed algoritmi adatti per i *big data*. Il sistema è stato, pertanto, contestualizzato in una situazione futura, simulando lunghi periodi di attività ed ipotizzando di aver raccolto molti più dati di quelli attualmente a disposizione.

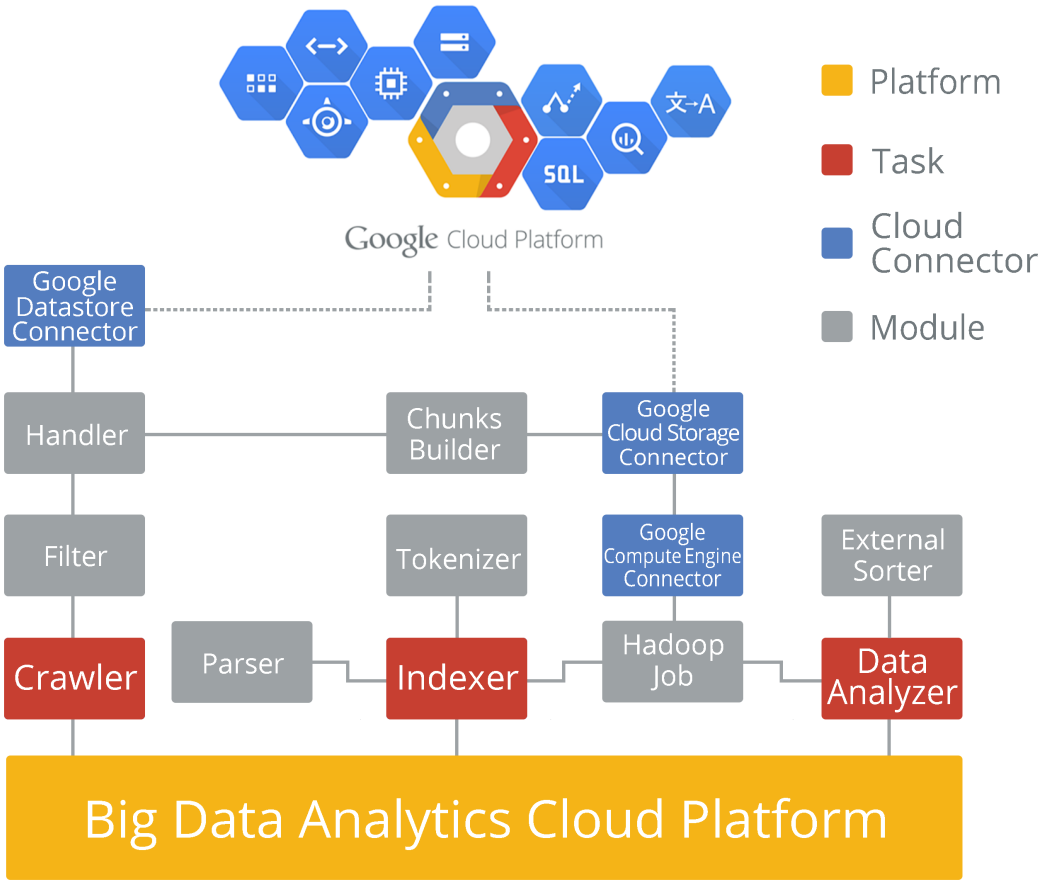

Figura 5.2: architettura della piattaforma sviluppata.

Combinando i moduli che costituiscono il sistema è possibile eseguire delle attività più complesse, dette ***task***. Nella versione iniziale della piattaforma l'attenzione è stata posta sui *task* orientati alla raccolta e all'elaborazione di informazioni, tralasciando quelli di *data visualization*. Tra i *task* implementati, particolare attenzione è stata posta su quelli destinati a trattare in modo diretto grandi moli di dati, detti *big data critic*. Questi *task* hanno richiesto, infatti, l'utilizzo di tecnologie ed algoritmi scalabili e le loro performance sono state oggetto dello studio riportato nel capitolo 6.

Il *software* è stato sviluppato in Java, lo stesso linguaggio utilizzato dalle librerie che esso utilizza come dipendenze. Le tecnologie specifiche per i *big data* impiegate, inoltre, espongono API in questo linguaggio, lasciando, di fatto, poco margine di scelta in fase realizzativa. A fronte del gran numero di *software* di terze parti e librerie necessari per completare i *task* della piattaforma, è stato utilizzato il *framework* Maven per gestire in modo agevole le dipendenze e rendere il programma facilmente evolvibile.

5.3 Obiettivi del lavoro di analisi

La piattaforma realizzata, nonostante sia impiegabile in diversi modi, è stata utilizzata per eseguire due esperimenti specifici, che hanno indirizzato lo sviluppo verso determinati moduli piuttosto che altri.

Il primo obiettivo del lavoro è stato scoprire se l'utilizzo di tecnologie per il calcolo distribuito si rivela una soluzione efficace per risolvere i problemi di elaborazione di *big data*. Per ottenere dei risultati significativi, a diversi *cluster* sono stati fatti eseguire sia *task data intensive*, sia *task* su piccoli *dataset*. Sono stati misurati sia i tempi di avvio che quelli di spegnimento dei *cluster* ed è stato effettuato un confronto con le prestazioni ottenute da macchine tradizionali.

Il secondo obiettivo, invece, è stato quello di individuare delle caratteristiche linguistiche salienti tra i dati raccolti, analizzandoli dal punto di vista statistico. Ignorando la semantica dei *tweet* si è ipotizzato di riuscire comunque a trovare delle peculiarità tra i messaggi di odio indirizzati a determinate categorie di persone. Tali caratteristiche potrebbero essere utilizzate, ad esempio, per scoprire nuovi messaggi, ignorati nelle precedenti sessioni di campionamento.

Per eseguire gli esperimenti, nella fase iniziale del lavoro, è stato raccolto da Twitter un numero significativo di messaggi, che sono stati raggruppati in base alla loro tipologia di intolleranza (contro le donne, contro i disabili, contro gli omosessuali, ecc.). Le categorie più ricche di messaggi sono state indicizzate durante il primo esperimento. Dai risultati ottenuti, durante il

secondo esperimento, sono state calcolate delle distribuzione di probabilità dei termini utilizzati nei *tweet*. Tali distribuzioni saranno più informative all'aumentare dei messaggi raccolti, pertanto i risultati ottenuti potrebbero migliorare a seguito di sessioni più lunghe di raccolta di dati.

Per far emergere dai *tweet* le parole più caratteristiche si è reso necessario confrontare le distribuzioni di probabilità con una di riferimento che approssimasse quanto meglio possibile dei testi generici in lingua italiana. Infatti, i termini frequenti nei *tweet* e non d'uso comune nella lingua scritta appartengono all'insieme di parole più peculiari per la categoria in esame e sono di interesse per lo studio. Ad esempio, il termine "*giorno*" è molto impiegato sia nei testi in italiano, sia nei *tweet* di odio contro le donne e quindi non può essere utile per individuare comportamenti misogini. Al contrario, il termine "*inchiavabile*" - mi venga perdonato l'utilizzo dell'aggettivo dispregiativo - è usato con ricorrenza solo nei *tweet* e può essere d'aiuto ad identificare messaggi contro le donne. Una distribuzione che approssimasse bene la nostra lingua è stata ottenuta indicizzando itWaC, una ricca raccolta di testi in italiano (per dettagli sul corpus si rimanda al paragrafo 6.1.1). I termini più significativi per i *tweet* discriminatori sono stati individuati calcolando la divergenza di Kullback-Leibler tra la distribuzione di probabilità dei messaggi e quella di itWaC (ulteriori dettagli sono forniti nei paragrafi 5.7 e 6.2.2). Ordinando i termini per significatività descrescente nei *tweet* è stato possibile evidenziare nelle prime posizioni i contenuti più peculiari per ogni gruppo.

Di seguito vengono analizzati i componenti della piattaforma che sono stati sviluppati per portare a termine il lavoro.

5.4 Crawler

Uno dei moduli più importanti della piattaforma è il *crawler*. Esso permette di raccogliere dati da una generica sorgente informativa per effettuare delle elaborazioni in tempo reale o in un momento successivo.

Il sistema può eseguire diversi tipi di *crawler*. Uno specifico per Twitter è stato già implementato e costituisce un componente essenziale per il progetto della Mappa dell'Intolleranza. Esso fa utilizzo della **Streaming API** di Twitter per ricevere i *tweet* provenienti dal flusso globale di messaggi che soddisfano dei criteri fissati dallo sviluppatore. Twitter offre diversi *streaming endpoint*, ciascuno adatto per particolari casi d'uso. L'*endpoint* scelto per questo *crawler* è quello pubblico, adatto quando si vogliono raccogliere dati di pubblico dominio su utenti od argomenti specifici, poiché è in grado di filtrare automaticamente i messaggi di interesse per il *client* tra quelli presenti nel

flusso di *tweet*. Gli altri *endpoint* sono quello specifico per gli utenti, che raccoglie tutti i dati (anche quelli privati) riguardanti un determinato utente (utilizzato, ad esempio, per i *client* mobili per Twitter di terze parti) e quello per i siti (la versione multi-utente del precedente).

Per ricevere informazioni da Twitter, il *client* deve autenticarsi come sviluppatore ed effettuare una richiesta al *social network* tramite l'API, specificando i contenuti che è interessato a ricevere. Pervenuta la richiesta, Twitter apre una connessione HTTP permanente con il *client* e gli invia dati in tempo reale, nella forma di documenti JSON, attraverso questo canale. È compito del *client* leggere in modo progressivo le informazioni che riceve da questa sorgente informativa e restare in ascolto di nuovi dati. Questo meccanismo di interazione si contrappone a quello dalla REST API, con cui il *client* può chiedere una particolare informazione al *social network* tramite un'interrogazione che utilizza una connessione temporanea per veicolare il dato. La natura della Streaming API, pertanto, richiede una particolare attenzione in fase di sviluppo del *client*, che dovrà esser capace di consumare dati in tempo reale, senza causare *bottleneck*. Per convertire velocemente i documenti JSON, ricevuti con l'API, in oggetti Java è stato perciò impiegato Jackson, un *JSON-processor* capace di estrarre dai documenti degli attributi specificati dall'utente (corpo del messaggio, autore, identificativo, provenienza, ecc.) ed ignorare tutti gli altri.

Il *crawler* per Twitter è stato progettato per raccogliere messaggi in una determinata lingua contenti specifiche *keyword*. Per realizzare la Mappa dell'Intolleranza si rende necessario possedere *tweet* potenzialmente omofobi, razzisti, discriminanti verso disabili, misogini ed antisemiti. Per comodità questi messaggi sono stati suddivisi in classi numerate progressivamente (1: omofobia, 2: razzismo, 3: disabilità, 4: misoginia, 5: antisemitismo). Per raccogliere questi *tweet*, un gruppo di psicologi della Università degli Studi di Roma La Sapienza ha fornito, per ogni classe, un insieme di termini, detti *seed*, utilizzati con accezione spesso discriminatoria in contesti quotidiani. Un lavoro successivo, inoltre, ha permesso di ampliare questo lessico tramite l'uso di sinonimi, variazioni morfologiche (nel genere e nel numero) e risorse esterne, quali BabelNet e Morph-it!. Tutti i *seed* raccolti sono stati utilizzati per raccogliere i *tweet* appartenenti a ciascuna classe.

Per un corretto funzionamento del *crawler* è richiesto un file di configurazione apposito. Al suo interno l'utente deve specificare alcuni parametri: le credenziali da sviluppatore di Twitter, la lingua dei messaggi che è interessato a raccogliere (per il caso di studio è stata scelta quella italiana), le classi di interesse (coi rispettivi *seed*) e la durata delle sessione di *crawling* per ciascuna classe. Nel file di configurazione, inoltre, l'utente può specificare dei filtri da applicare ai *tweet* in arrivo e degli *handler* per gestire i *tweet* che

frocio	kecca	bocchinari	rottinculo
froci	checche	bokkinari	passiva
frocie	kekke	deviato	passivo
froce	invertito	deviati	passive
ricchione	invertite	deviate	passivi
rikkione	inverite	deviate	leccafica
ricchionazzo	invertiti	camionista	leccafiche
rikkionazzo	pervertito	camioniste	culanda
ricchioni	pervertita	marchettaro	travestito
rikkioni	pervertiti	marchettari	travestiti
finocchio	pervertite	sodomita	travone
finokkio	pompinaro	sodomiti	travoni
finocchi	pompinari	culattone	ciuccia cazzi
finokki	bocchinaro	culattoni	ciuccia cazzo
checca	bokkinaro	pigliarculo	succhia cazzo
culo rotto	culi rotti		

Tabella 5.1: *seed* classe 1.

hanno superato questi controlli.

I filtri sono stati implementati per dare la possibilità all'utente di bloccare determinati messaggi in base a delle loro caratteristiche (contenuto, provenienza, autore, ecc.) e dotare il modulo di un meccanismo di protezione dallo spam. Effettuando lunghe sessioni di *crawling* è emerso, infatti, che diversi *spam bot* impiegavano alcuni dei *seed* delle classi per promuovere siti web di carattere pornografico (cosa consentita dai termini di utilizzo di Twitter) o per campagne pubblicitarie di varia natura, producendo messaggi inutili per il caso di studio. Mediante i filtri sviluppati questi ed altri messaggi sono stati bloccati, al fine di non alterare le distribuzioni di probabilità dei termini delle classi ed i risultati stessi delle analisi. L'utente è libero di non impiegare nessun filtro, utilizzarne alcuni già implementati o scriverne di nuovi.

Gli *handler*, invece, sono i moduli preposti alla gestione dei *tweet* che hanno superato i controlli dei filtri che sono stati attivati. Anche in questo caso l'utente può aggiungere delle classi al sistema per eseguire nuove operazioni, oppure impiegare una già presente nella piattaforma. Gli *handler* possono eseguire azioni di qualsiasi natura: memorizzare le informazioni inerenti un *tweet* in un database (locale o in *cloud*), condurre analisi di vario genere, serializzare il contenuto dei messaggi in un file di testo, ecc. Quest'ultima operazione era quella necessaria al caso di studio e, pertanto, è stato sviluppato un apposito *handler* per serializzare i *tweet*. L'*handler* in questione,

negro	terrone	beduino	zingare
negri	bangla	beduini	muso giallo
negra	rumeno di merda	polentone	musi gialli
negre	rumena di merda	polentoni	muso da scimmia
negretto	rumeni di merda	polentona	musi da scimmia
negretta	rumene di merda	polentone	kebabbaro
negretti	romeno di merda	albanese di merda	kebabbari
negrette	romeni di merda	albanesi di merda	kebabbara
terrone	romena di merda	zingaro	kebabbare
terroni	romene di merda	zingari	crucco
terrona	mangiarane	zingara	crucchi

Tabella 5.2: *seed* classe 2.

handicappato	storpia	nana	quattrocchi
handicappati	storpie	nane	cecato
handicappata	mongoloide	spastico	cecati
handicappate	mongoloidi	spastici	cecata
andicappato	cerebroleso	spastica	cecate
andicappati	cerebrolesi	spastiche	mongoflettico
andicappata	cerebrolesa	zoppo	mongoflettici
andicappate	cerebrolese	zoppi	mongoflettica
storpio	nano	zoppa	mongoflettiche
storpi	nani	zoppe	

Tabella 5.3: *seed* classe 3.

insieme ad altri componenti della piattaforma sviluppata, fa utilizzo di un *chunks builder* (illustrato nel paragrafo 5.8), un modulo utile per scrivere molti dati su più file numerati progressivamente ed aventi stessa dimensione.

Lunghe sessioni di *crawling* sono state avviate in una fase iniziale su tutte le classi. In un momento successivo, però, il *crawler* è stato mantenuto in esecuzione solo su quelle che hanno dimostrato riguardare più messaggi su Twitter: omofobia, disabilità e misoginia. Per ottenere distribuzioni di probabilità sufficientemente significative, infatti, risultava necessario raccogliere quanti più *tweet* possibili per una determinata classe: si è preferito, così, concentrare il *crawler* su specifiche classi ed ottenere campioni più grandi piuttosto che ottenere dati poco significativi su tutte le classi. Le analisi condotte, in ogni caso, potranno essere ripetute in futuro su tutte le classi, qualora il numero di *tweet* raccolto dovesse crescere.

troia	scrofe	zoccoletta	sfasciolacazzi
troie	sgualdrina	zoccolette	culona
troietta	sgualdrine	mignotta	culone
troiette	sciacquetta	mignotte	frigida
troiona	sciacquette	mignottone	frigide
troione	peripatetica	bocchinara	figa di legno
puttana	peripatetiche	bocchinare	fighe di legno
puttane	meretrice	pompinara	battona
puttanella	meretrici	pompinare	battone
puttanelle	cicciona	cagna	ciuccia cazzi
bagascia	ciccione	cagne	cesso
bagasce	vacca	strappona	cessa
baldracca	vacche	strappone	cesse
baldracche	zoccola	smandrappona	sfigata
scrofa	zoccole	smandrappone	sfigate

Tabella 5.4: *seed* classe 4.

ebreo ai forni	ebree di merda
ebrei ai forni	rabbino
ebrea ai forni	rabbini
ebree ai forni	giudeo
ebreo di merda	giudea
ebrei di merda	giudei
ebrea di merda	

Tabella 5.5: *seed* classe 5.

5.5 Parser del corpus itWaC

Come già illustrato nel paragrafo 5.3, per ottenere una distribuzione di probabilità di termini che rappresentasse bene la lingua italiana si è ritenuto opportuno indicizzare il corpus itWaC. Esso, tuttavia, si presenta in una forma non direttamente processabile, poiché è costituito da archivi compressi, contenenti ciascuno una parte dei documenti, per giunta in formato XML. Tali parti, oltretutto, non sono costituite da semplice testo, ma contengono le annotazioni prodotte dalla fase di *part-of-speech tagging* eseguita anni fa dai curatori del progetto.

Per questo motivo la piattaforma è stata dotata di un *parser* capace di elaborare il corpus e produrre dei file di testo indicizzabili. Il modulo sviluppato, pertanto, processa i file XML contenuti negli archivi senza decomprimerli,

ignora i *tag* e serializza progressivamente su file di più piccole dimensioni il testo dei documenti. L'operazione di serializzazione viene delegata al *chunks builder*, illustrato nel paragrafo 5.8.

La grande dimensione del corpus richiede che il *task* venga completato con accorgimenti particolari (non è possibile, ad esempio, caricare tutti i documenti in memoria ed avviare il *parser*). Per effettuare il processo, tuttavia, non è stato necessario ricorrere a tecnologie *cloud*, poiché si è reso sufficiente elaborare i file XML secondo il noto *standard* StAX, che prevede una lettura progressiva dei dati. Con questo approccio i documenti del corpus vengono assimilati a flussi da cui leggere gradualmente le informazioni: in questa maniera risulta indifferente elaborare file di grandi o piccole dimensioni, poiché essi sono comunque processati una riga per volta, senza eccessivi consumi di memoria.

5.6 Indicizzazione con MapReduce

Per indicizzare un generico documento è stato scritto un *job* MapReduce.

Il *job* implementato utilizza un Mapper, un Combiner ed un Reducer per contare le occorrenze di una parola nel *dataset* di input, eseguendo il minor numero di operazioni possibili (piccoli miglioramenti nel codice producono una sensibile diminuzione dei tempi di esecuzione per via dell'elevato numero di chiamate a funzioni da parte dei nodi del *cluster*). Le funzioni scritte condividono i principi di funzionamento alla base dei *job* esemplificativi illustrati nei paragrafi 2.3 e 2.4.2, con particolari accortezze aggiuntive per la natura dei grandi dati di input (ad esempio, l'utilizzo del tipo *long* invece che *int* per tutte le variabili preposte a contare le occorrenze di un termine o il numero di righe di un file).

Dominio e codominio delle funzioni sono riportati di seguito:

```
map (long, String) -> list(String, long)
combine (String, list(long)) -> (String, long)
reduce (String, list(long)) -> (String, long)
```

La funzione *map* riceve in input una coppia costituita dal numero di riga del file di input e dal testo presente su quella riga (potrà essere un *tweet* oppure parte di un documento di itWaC). Successivamente invoca il *tokenizer* opportuno sul testo (inizializzato preliminarmente) per dividerlo nelle sue parole costituenti e ritorna tante coppie (*token*, 1). Due tipi di *tokenizer* sono attualmente presenti nella piattaforma: uno specializzato nell'elaborazione di *tweet* e l'altro più appropriato per semplice testo (per dettagli si rimanda al paragrafo 6.2.2).

```
public void map(LongWritable key, Text value,
  OutputCollector<Text, LongWritable> output,
  Reporter reporter) throws IOException {

  List<String> tokens = tokenizer.tokenize(
    value.toString());
  for(String token : tokens) {
    word.set(token);
    output.collect(word, one);
  }

}
```

La funzione combine, applicata localmente su ogni nodo, riceve in input un *token* ed una lista di 1, avente dimensione pari al numero di occorrenze del *token* in quella riga. Si limita, pertanto, a restituire una coppia (*token*, dimensione_lista).

```
public void combine(Text key, Iterator<LongWritable>
  values, OutputCollector<Text, LongWritable> output,
  Reporter reporter) throws IOException {

  long count = 0;
  while (values.hasNext()) {
    values.next();
    count++;
  }
  output.collect(key, new LongWritable(count));

}
```

La funzione *reduce*, infine, riceve in input un *token* ed una lista delle sue occorrenze nelle righe del *dataset*. Di conseguenza non deve che ritornare una coppia costituita dal *token* stesso e dalla somma dei numeri contenuti nella lista di input, ovvero il numero totale di occorrenze nella collezione.

```
public void reduce(Text key, Iterator<LongWritable>
  values, OutputCollector<Text, LongWritable> output,
  Reporter reporter) throws IOException {

  long sum = 0;
  while (values.hasNext()) {
```

```
    sum += values.next().get();
  }

  output.collect(key, new LongWritable(sum));

}
```

5.7 Divergenza di Kullback-Leibler

La divergenza di Kullback-Leibler (KLD) è una fondamentale equazione della teoria dell'informazione che quantifica la prossimità tra due distribuzioni di probabilità. La sua formula ha origine nella teoria della verosimiglianza e mira a valutare quanta informazione si perde approssimando una distribuzione con un'altra.

Date due distribuzioni discrete di probabilità P e Q, la divergenza di Kullback-Leibler di Q da P è data dalla formula:

$$D_{\mathrm{KL}}(P\|Q) = \sum_i P(i) \ln\left(\frac{P(i)}{Q(i)}\right), \tag{5.1}$$

dove i rappresenta il termine i-esimo delle distribuzioni, P(i) la sua probabilità nella prima distribuzione e Q(i) quella nella seconda distribuzione.

La misura non è simmetrica. La divergenza di Kullback–Leibler di Q da P, indicata con $\mathrm{D_{KL}}(\mathrm{P}||\mathrm{Q})$, è la misura dell'informazione persa quando Q è usata per approssimare P. Tale divergenza non è necessariamente uguale a quella di P da Q, che viene invece indicata con $\mathrm{D_{KL}}(\mathrm{Q}||\mathrm{P})$.

Come si può facilmente osservare, la divergenza viene calcolata sommando l'informazione persa termine per termine. Calcolando l'approssimazione di due distribuzioni su un solo termine si ottiene quella che viene definita come *pointwise Kullback-Leibler divergence* (PKLD).

Fissato un termine i, la PKLD è data dalla formula:

$$\delta_{\mathrm{KL,i}}(P\|Q) = P(i) \ln\left(\frac{P(i)}{Q(i)}\right). \tag{5.2}$$

Tale valore è tanto più alto quanto maggiore è la differenza delle probabilità dello stesso termine nelle due distribuzioni.

I termini con PKLD maggiore, pertanto, risultano essere quelli più significativi quando la prima distribuzione viene confrontata con la seconda.

5.7.1 Calcolo della PKLD con MapReduce

Il calcolo della PKLD tra i termini di due distribuzioni è stato eseguito con un *job* MapReduce, utilizzante un Mapper ed un Reducer.

Dominio e codominio delle funzioni sono riportati di seguito:

```
map (long, String) -> list(String, double)
reduce (String, list(double)) -> (String, double)
```

La funzione *map* riceve in input una coppia costituita dall'indice del file che sta leggendo e dalla sua annessa riga contenente il termine i-esimo ed il suo numero di occorrenze (risultato dell'anteriore *task* di indicizzazione). Calcola, successivamente, la probabilità del termine nel documento (i dettagli sono nel paragrafo 6.1.2 sul protocollo sperimentale). Anteponendo a ciascun termine un determinato prefisso, la funzione *map* è in grado di riconoscere su quale documento calcolare la probabilità. Per motivi di efficienza, le dimensioni dei vocabolari dei due documenti in esame ed il numero totale di occorrenze di parole in essi vengono calcolati dal nodo *master* del *cluster* prima di avviare il *job*. Essi, infatti, rimangono costanti durante l'elaborazione e possono risultare computazionalmente esosi da misurare. Le funzioni *map*, pertanto, possono direttamente leggere questi valori senza doverli determinare ogni volta. I Mapper completano la loro esecuzione ritornando una coppia costituita dal termine e dalla probabilità calcolata. La funzione *map*, quando riconosce di stare calcolando la probabilità di un termine nel secondo documento, moltiplica il valore ottenuto per -1 prima di ritornarlo. In questo modo essa produce coppie del tipo (termine, ±P(termine)). Questo accorgimento è stato ideato per permettere alla funzione *reduce* di riconoscere, successivamente, il documento su cui è stata calcolata una data probabilità (la KLD, infatti, non è simmetrica).

```
public void map(LongWritable key, Text value,
  OutputCollector<Text, DoubleWritable> output,
  Reporter reporter) throws IOException {

  String line = value.toString();
  boolean firstClass = [...]
  String term = [...]
  Long termFreq = Long.parseLong(
    line.substring(separatorIndex+1, line.length()));

  if(firstClass) {
    output.collect(new Text(term), new DoubleWritable(
      (termFreq.doubleValue() + 1) /
```

```
      (firstDistribSize + firstDistribLines)));
  } else {
    output.collect(new Text(term), new DoubleWritable(
    (termFreq.doubleValue() + 1) /
    (secondDistribSize + secondDistribLines) * -1));
  }

}
```

La funzione *reduce* riceve in input un termine ed una lista di probabilità. Nel caso specifico la lista contiene la probabilità del termine nel primo documento e/o la probabilità del termine nel secondo (se un termine compare solo in una distribuzione la lista conterrà un solo elemento). Controllando se la probabilità è positiva o negativa la funzione *reduce* è in grado di determinare la distribuzione d'origine ed applicare correttamente la formula 5.2 per il calcolo della PKLD. Il numero totale di occorrenze di parole nelle collezioni e le cardinalità delle stesse sono già stati calcolati dal nodo *master* e direttamente accessibili dal Reducer.

```
public void reduce(Text key, Iterator<DoubleWritable>
  values, OutputCollector<Text, DoubleWritable> output,
  Reporter reporter) throws IOException {

  while (values.hasNext()) {
    double curr = values.next().get();
    if(curr < 0) {
      secondDistribProbability = curr * -1;
    } else {
      firstDistribProbability = curr;
    }
  }

  if(firstDistribProbability == 0) {
    firstDistribProbability = ((double) 1) /
      (firstDistribSize + firstDistribLines);
  }
  if(secondDistribProbability == 0) {
    secondDistribProbability = ((double) 1) /
      (secondDistribSize + secondDistribLines);
  }

  output.collect(key, new DoubleWritable(
```

```
        (Math.log(firstDistribProbability) -
        Math.log(secondDistribProbability)) *
        firstDistribProbability));

}
```

La funzione *reduce*, quindi, ritorna un insieme di coppie (termine, PKLD) non ordinate, che Hadoop serializza su file.

5.8 Altri moduli della piattaforma

Oltre ai moduli presentati nei paragrafi precedenti, la piattaforma è dotata anche di altri semplici strumenti che offrono delle funzionalità utili per *task* più complessi.

Il primo, già illustrato brevemente, è il *chunks builder*, impiegato in diversi contesti per serializzare dati in file aventi dimensione fissa e nomi fedeli a *pattern* specificati dall'utente. La sua capacità di scrivere dati progressivamente, a partire dall'ultimo *chunk* presente nella cartella di destinazione, lo rende adatto, in modo particolare, a gestire ripetute sessione di *crawling* effettuate ad intervalli di tempo irregolari: i nuovi *tweet*, infatti, vengono automaticamente aggiunti all'ultimo file preesistente che, raggiunto il limite prefissato, sarà chiuso in favore di un nuovo *chunk*. Quest'ultimo presenterà una numerazione consistente con i file già presenti nella cartella, risultati da precedenti attività del *crawler*.

La piattaforma, inoltre, è dotata di un modulo per interfacciare il sistema con Cloud Storage. Esso può essere utilizzato per ispezionare il contenuto di un *bucket*, elencare i file appartenenti ad una cartella, inviare oggetti dal *file system* locale o scaricarli dal *cloud*.

Infine, è presente uno strumento per effettuare l'*external sort*, un algoritmo di ordinamento di *big data* che non richiede il caricamento dell'intero *dataset* in memoria. Utilizzando un approccio che prevede ordinamenti parziali e serializzazioni successive su file i dati di input vengono ordinati con basso consumo di memoria. Questo algoritmo è stato impiegato, ad esempio, per ordinare i valori della PKLD al termine del *job* MapReduce al fine di evidenziare i termini più significativi.

Capitolo 6

Esperimenti con la piattaforma

6.1 Valutazione delle performance dei cluster

Questo esperimento mira a valutare l'efficacia delle tecnologie per il calcolo distribuito attraverso l'esecuzione di *job* MapReduce per indicizzare grandi e piccoli documenti. I *cluster* di Hadoop (versione 1.2.1) di cui si sono misurate le performance sono stati eseguiti su VM di Compute Engine di diverso tipo.

6.1.1 Dataset di input

Il *job* di indicizzazione descritto nel paragrafo 5.6 è stato eseguito su quattro collezioni di dati: il corpus itWaC, i *tweet* contenenti indicatori di odio verso gli omosessuali (classe 1), quelli verso i disabili (classe 3) e quelli verso le donne (classe 4). Queste tre classi, infatti, hanno dimostrato ricevere più messaggi in fase di *crawling*.

itWaC è una ricca raccolta di testi in lingua italiana, ottenuta tramite un processo mirato di *crawling* nel 2006, utilizzante alcuni *seed* scelti *ad hoc*. La collezione fa parte di una raccolta più ampia di corpora annotati (con *part-of-speech tagging* e lemmatizzazione) chiamata WaCky, che comprende anche risorse per altre lingue, come l'inglese (ukWaC) ed il tedesco (deWaC). Con oltre quattro milioni di documenti su contenuti diversi, filtrati ed annotati, itWaC costituisce uno dei più grandi corpus pubblici in lingua italiana e rappresenta una fondamentale e versatile risorsa linguistica. Con il *parser* illustrato nel paragrafo 5.5 sono stati ricavati dai documenti oltre 11 GB di file di testo, suddivisi per comodità in file da 500 MB. La tabella 6.1 riporta alcune caratteristiche del corpus, tratte dalla documentazione ufficiale [4].

Per i *tweet* sono stati usati sia i messaggi in italiano raccolti dal *crawler*, sia altri frutto di precedenti lavori. Essendo questi ultimi in diverse lingue, si è reso necessario applicare un procedimento preliminare di filtraggio. Con

Num. di *seed*	1.000
Dimensione del corpus grezza	379 GB
Dimensione dopo il *filtering*	19 GB
Dimensione con annotazioni	30,6 GB
Num. di *token*	1.585.620.279

Tabella 6.1: caratteristiche del corpus itWaC.

due strumenti, *language-detection*[1] e *language detection with infinity-gram*[2] (precisi oltre il 99.5% per la nostra lingua), sono stati recuperati i *tweet* in italiano. Le collezioni sono state poi ripulite da messaggi duplicati e messaggi di spam.

Un riepilogo delle collezioni è mostrato nella tabella 6.2.

	Dimensione (kB)	Contenuto
itWaC	11.160.944	1.870.000 documenti
classe 1	1.738	22.564 *tweet*
classe 3	2.242	29.793 *tweet*
classe 4	17.935	249.425 *tweet*

Tabella 6.2: dimensioni delle collezioni di input.

6.1.2 Protocollo sperimentale

Le macchine di Compute Engine sono state configurate per utilizzare come *file system* condiviso non il tradizionale Hadoop Distributed File System, ma un *bucket* di Cloud Storage. Questo risultato è stato ottenuto per mezzo del *Google Cloud Storage Connector for Hadoop*, uno strumento sviluppato da Google ed utilizzabile gratuitamente. Il connettore offre una serie di vantaggi, tra i quali la possibilità di gestire facilmente i file consumati da Hadoop (anche quando il *cluster* è spento), migliori performance durante l'esecuzione di *job* MapReduce per via dell'infrastruttura di Google, minori tempi di avvio del *cluster*, assenza delle routine di gestione dell'HDFS, ecc. I *dataset* di input per Hadoop, per questo motivo, sono stati caricati su Cloud Storage, in un momento antecedente all'esecuzione dei *job* MapReduce, in un *bucket* accessibile da tutti i nodi.

Le configurazioni di *cluster* scelte per gli esperimenti sono state limitate da alcuni vincoli che Google impone agli sviluppatori in possesso di account

[1] `https://code.google.com/p/language-detection/`

[2] `https://github.com/shuyo/ldig`

senza determinati privilegi (per ottenere la rimozione di queste restrizioni è necessario compilare una richiesta formale al *cloud provider* e spiegare le motivazioni per cui si vogliono impiegare più istanze di Compute Engine). È stato possibile avviare, per questo motivo, al massimo 23 macchine diverse contemporaneamente e suddividere tra di esse 24 CPU virtuali. Per la natura dei *task* da completare si è preferito scegliere macchine dotate di CPU performanti con un quantitativo di memoria RAM sufficiente per eseguire i *job*, ma comunque molto modesto se paragonato a quello massimo consentito da Compute Engine.

Per poter osservare le variazioni nei tempi di indicizzazione sono state selezionate configurazioni di *cluster* aventi stesso numero di nodi con macchine di tipo diverso ed altre configurazioni con un numero di nodi variabile, ma tutti dello stesso tipo.

Sono stati riportati anche i tempi medi di avvio e di spegnimento dei *cluster* di Hadoop. Questi risultati sono stati calcolati misurando il tempo impiegato da Compute Engine per completare l'esecuzione di alcuni *script*, che automatizzano l'accensione/spegnimento delle macchine, l'interscambio di chiavi per la connessione SSH, la configurazione di Hadoop e l'avvio/terminazione dei suoi *daemon*. Nessun file è stato caricato sulle macchine in fase di *deploy* del *cluster*, presumendo che la piattaforma di analisi dei dati sia presente su dischi persistenti: i risultati, pertanto, non sono affetti da ritardi dovuti a tempi variabili di *upload*.

Le performance dei *cluster* sono, inoltre, messe a confronto con quelle ottenute da una macchina locale eseguente Hadoop in modalità *standalone* sugli stessi dati in input. Essa possiede un processore Intel Core i5-3570K, 3.40GHz con 16GB di memoria RAM. Per le specifiche tecniche delle macchine di Compute Engine, invece, si rimanda alla tabella riassuntiva del paragrafo 4.2.1.

Tutti i *job* di Hadoop sono stati ripetuti tre volte, al fine di minimizzare nei risultati la componente di variabilità dovuta ad errori nelle macchine, problemi di I/O, lentezza della rete, ecc. Quelli che vengono presentati sono i tempi calcolati con media aritmetica.

6.1.3 Risultati

La tabella 6.3 riporta i tempi di avvio e spegnimento dei *cluster*. I tempi di indicizzazione sono invece mostrati nella tabella 6.4.

Le statistiche sugli indici prodotti in output dai *job* non sono rilevanti per i risultati di questa analisi. Vengono, pertanto, mostrate nel paragrafo 6.2.1, contenente la descrizione dei dati di input per l'altro esperimento.

Numero *worker*	Tipo macchine	Accensione	Spegnimento
4	n1-standard-1	185	56
9	n1-standard-1	205	61
22	n1-standard-1	206	66
4	n1-standard-2	181	**52**
9	n1-standard-2	**151**	**52**
4	n1-standard-4	178	57
2	n1-standard-8	143	56

Tabella 6.3: tempi medi (in secondi) di accensione e spegnimento dei *cluster*. In **grassetto** la migliore prestazione.

Numero *worker*	Tipo macchine	itWaC	classe 1	classe 3	classe 4
4	n1-standard-1	1097	164	153	158
9	n1-standard-1	534	137	147	160
22	n1-standard-1	**320**	156	154	167
4	n1-standard-2	822	98	101	110
9	n1-standard-2	409	102	103	110
4	n1-standard-4	457	70	72	70
2	n1-standard-8	459	**66**	**52**	**54**
1	locale	1519	2	2	5

Tabella 6.4: tempi medi (in secondi) di indicizzazione dei documenti. In **grassetto** la migliore prestazione.

6.1.4 Discussione dei risultati

Prima di procedere con l'analisi dei dati raccolti si segnala che in una occasione, durante l'esecuzione di un *job*, è stato ricevuto da Compute Engine un *backend error*, che non ha comunque pregiudicato il completamento del processo di indicizzazione (è stato solo registrato un ritardo di 30 secondi).

Osservando la tabella 6.4 emergono, innanzitutto, alcuni risultati facilmente ipotizzabili già in fase di progettazione dei test. Risulta evidente che, mantenendo costante il numero di nodi nel *cluster*, le prestazioni migliorano con l'aumentare del numero complessivo di CPU a disposizione, ovvero scegliendo macchine progressivamente più potenti. Quanto detto è vero sia per grandi *dataset* (itWaC) che per altri più piccoli (i *tweet*).

Il ragionamento inverso non trova, invece, riscontri empirici. Mantenendo costante il tipo di macchine del *cluster* e facendo variare il numero di nodi si osservano risultati diversi in funzione della dimensione dei dati di input. Per grandi *dataset* si nota un miglioramento delle performance all'aumentare del

numero di nodi interconnessi, verosimilmente perché maggiori risorse computazionali sono a disposizione del *framework*. Quando i dati di input sono di meno, invece, le prestazioni migliorano progressivamente fino ad un certo punto, per cominciare poi a degradare. Il motivo può essere riconducibile al fatto che, oltre una certa soglia, il vantaggio guadagnato con l'aumento di CPU viene perso nelle fasi di interscambio di messaggi e di dati tra i nodi del *cluster*. Questo risultato induce a pensare che, aumentando notevolmente il numero di nodi interconnessi, anche con grandi *dataset* si potrebbe verificare un fenomeno simile: accurati *tuning* del *cluster* sui dati in input sono, pertanto, sempre consigliati.

In generale è possibile osservare come le migliori prestazioni su grandi *dataset* siano state ottenute da *cluster* con moltissimi nodi, seppur costituiti da macchine del tipo formalmente meno potente. Su piccoli dati di input, invece, le prestazioni migliori sono state registrate da *cluster* formati da due soli *worker*, ma del tipo migliore sulla carta.

Effettuando un confronto con la macchina locale eseguente Hadoop in modalità *standalone*, emergono indiscutibilmente i vantaggi di non dover effettuare scambi di dati tra nodi durante l'indicizzazione dei *tweet*: i *task* vengono completati in pochi secondi, con prestazioni nettamente migliori di quelle di qualunque *cluster*. Il risultato opposto si osserva con il *dataset* di itWaC: la macchina locale registra delle prestazioni nettamente peggiori di quelle del *cluster* meno performante. Il confronto, poi, con la configurazione migliore del *cluster* dovrebbe essere sufficiente a dimostrare la potenza del *framework*, poichè i nodi impiegano il 78% di tempo in meno di un computer di fascia medio-alta.

I tempi di accensione e spegnimento delle macchine virtuali sono, tutto sommato, soddisfacenti e poco variabili. I *cluster* con un maggior numero di nodi impiegano, ragionevolmente, più tempo degli altri per accendersi. La stessa osservazione vale per i tempi di spegnimento, anche se qui le differenze si attenuano.

Si fa notare, inoltre, la grande somiglianza nelle performance del *cluster* costituito da quattro *worker*, ciascuno con quattro CPU, e di quello costituito da due *worker* con otto CPU ciascuno (sedici CPU in totale per entrambe le configurazioni). Essendo anche uguali i prezzi complessivi delle VM è difficile affermare quale soluzione sia preferibile.

6.2 Valutazione di peculiarità lessicali dei tweet

Questo esperimento mira ad inviduare i termini più salienti per ciascuna classe di *tweet* utilizzando degli approcci statistici che non richiedono com-

prensione della semantica dei messaggi veicolati sul *social network*. L'ipotesi dell'esperimento è che la divergenza di Kullback-Leibler possa costituire una buona tecnica di *feature selection*.

6.2.1 Dataset di input

Per calcolare la KLD sono stati utilizzati gli indici prodotti dal primo esperimento. La tabella 6.5 riporta, per ciascuna collezione, la dimensione dell'indice generato ed il suo numero di vocaboli.

	Dimensione (kB)	Cardinalità vocabolario
itWaC	91.890	4.431.080
classe 1	1.069	54.507
classe 3	1.430	72.569
classe 4	5.766	280.412

Tabella 6.5: dimensioni degli indici prodotti durante il primo esperimento.

6.2.2 Protocollo sperimentale

Sebbene la fase di indicizzazione dei documenti sia stata effettuata in precedenza, risulta necessario approfondire alcuni dettagli sulle modalità con cui essa è stata eseguita. Le informazioni che seguono non sono state riportare nel protocollo sperimentale dell'altro esperimento perchè irrilevanti per il tipo di analisi che si aveva intenzione di condurre (i *job* da fare eseguire ai *cluster*, infatti, potevano essere di qualunque natura).

Il processo di indicizzazione di itWaC è stato completato invocando semplicemente un *tokenizer* di Apache Lucene, libreria per l'*information retrieval*, sulle righe dei documenti e contando i *token* prodotti. Lo stesso procedimento, però, non è stato applicabile sui *tweet*. La loro particolare natura rende inefficaci gli strumenti tradizionali di analisi del linguaggio, che non sono in grado di trattare appropriatamente menzioni, abbreviazioni, *retweet*, URL, *hashtag* ed *emoticon*. Il *tokenizer* impiegato, per questo motivo, è stato uno progettato appositamente per questi brevi messaggi dal gruppo di ricerca Noah's ARK della Carnegie Mellon University (CMU), in grado di fare anche *part-of-speech tagging*. La libreria *ark-tweet-nlp*[3] è stata così inclusa nella piattaforma ed il suo *tokenizer* invocato sui dati nella collezione. Dovendo calcolare sui termini delle collezioni la PKLD, si è reso necessario

[3] `https://github.com/brendano/ark-tweet-nlp/`

fare in modo che i risultati prodotti dai *tokenizer* fossero quanto più omogenei possibile. Per tale motivo è stato impiegato, durante l'indicizzazione di itWaC, l'UAX29URLEmailAnalyzer di Lucene, che si è rivelato produrre risultati più consistenti con lo strumento della CMU (non frammenta URL ed indirizzi email, ha una simile gestione della punteggiatura, ecc.).

Per via del differente significato che, sui *social network* e sui siti web, una parola assume quando scritta in maiuscolo (per convenzione si assume che stia venendo urlata) non è stato applicato il *case folding*. Nel condurre questo genere di analisi, inoltre, non vengono generalmente eseguite operazioni di *stemming* o altre manipolazioni del testo (come anche la rimozione di "#" dagli *hashtag*, o di "@" dalle menzioni).

Nel corso dell'esperimento è stata calcolata la PKLD tra le parole di ciascuna classe ed il corpus itWaC, secondo costante termine di riferimento. Per dettagli sulla PKLD si rimanda al paragrafo 5.7. Il *job* descritto nel paragrafo 5.7.1 è stato utilizzato per calcolare la significatività delle parole. I valori ottenuti sono stati in seguito ordinati in modo descrente.

Durante il calcolo della probabilità di un termine in una collezione è stato applicato il *Laplace smoothing* [27], poiché alcune parole comparivano nel copus di itWaC ma non nei *tweet* e viceversa (soprattutto nel caso di URL). Pertanto, la formula applicata per il calcolo della probabilità R di un termine i in un documento D è stata:

$$R(i) = \frac{f_{\mathrm{D,i}} + 1}{f_{\mathrm{D}} + |D|}, \tag{6.1}$$

dove $f_{D,i}$ rappresenta il numero di occorrenze del termine i nel documento D, f_D il numero totale di occorrenze di parole in D e |D| la cardinalità del vocabolario, ovvero il numero di parole diverse nel documento D.

6.2.3 Risultati

Il calcolo della KLD e l'ordinamento dei risultati vengono eseguiti in tempi relativamente brevi per via della modesta dimensione degli indici e, per tale motivo, non si è ritenuto significativo riportarli. Vengono, invece, mostrati i primi 30 termini emersi per ciascuna classe nelle tabelle 6.6, 6.7 e 6.8. I valori della PKLD sono stati omessi, atteso che, così come appaiono *de visu*, sono di difficile interpretazione.

Per rendere più leggibili i risultati dai *ranking* sono stati rimossi i *seed* usati in fase di *crawling* e tutte le parole inutili, come congiunzioni, preposizioni, sillabe, segni di interpunzione ed *emoticon*.

Rank 1-10	*Rank* 11-20	*Rank* 21-30
Harry	psiconano	prezzemolo
cazzo	stanzaselvaggia	Justin
culo	foto	@Lory_Bianco
#pechinoexpress	faccia	ragazzo
tisana	#tvoi	Occhio
merda	tweet	@rossmro
isterica	Twitter	Tisana
Louis	@GayOggi	Mussolini
ride	@__Baekhyunnie	SkyTG24
Meglio	#GF13	@Harry_Styles

Tabella 6.6: *ranking* dei termini della classe 1.

Rank 1-10	*Rank* 11-20	*Rank* 21-30
@Ty_il_nano	Apple	merda
@Louis_Tomlinson	iPod	Brunetta
iPhone	ascia	amo
@matteorenzi	cazzo	@NiallOfficial
@justinbieber	giardino	piccolo
Renzi	anni	@beppe_grillo
malefico	#Mistero	Lilli
#mistero	#Amici13	Biancaneve
ottavo	impara	@AlbanoColmo
Louis	Sistema	lanza

Tabella 6.7: *ranking* dei termini della classe 3.

6.2.4 Discussione dei risultati

Le parole emerse con l'esperimento permettono di fare diverse considerazioni sui *tweet* elaborati. Nell'analizzare i risultati bisogna tenere a mente che i messaggi sono stati raccolti dal *crawler* utilizzando delle *keyword* e che, per ovvi motivi, nelle collezioni prodotte una grande quantità di messaggi non risulta contenere alcuna forma di intolleranza (falsi positivi).

Osservando le tabelle risulta evidente la presenza di parole volgari ed ingiuriose, spesso usate per bestemmiare. L'alta significatività di questi termini non può che confermare la prevalenza di *tweet* nel corpus con accezione negativa. Molti messaggi, infatti, sono esternazioni di rabbia o mirano ad offendere altri individui, specialmente politici (Matteo Renzi, Silvio Berlusconi, Beppe Grillo e Renato Brunetta) e cantanti (One Direction e Justin Bieber).

Rank 1-10	*Rank* 11-20	*Rank* 21-30
@justinbieber	capelli	vergine
madonna	whatsapp	ansia
Vaffanculo	mamma	inchiavabile
profilo	Tumblr	Troia
TROIA	Twitter	piango
inferno	Taylor	'rca
Sesso	depressa	Napoli
America	Swift	Veronica
Brutta	bocca	account
Scusa	Jenna	tette

Tabella 6.8: *ranking* dei termini della classe 4.

Numerose sono anche le parole che completano espressioni di uso comune con i *seed*, ad esempio "*iPod*", "*ottavo*", "*giardino*" e "*Biancaneve*" con "*nano*" oppure "*semi*" con "*finocchio*". Tutte queste potrebbero essere utilizzate, ad esempio, per la costruzione di filtri per il *crawler* per raccogliere meno falsi positivi.

Avendo concentrato le sessioni di *crawling* in pochi mesi, i *ranking* evidenziano molte delle parole inerenti le tendenze su Twitter di quei giorni. In modo particolare, balzano agli occhi gli *hashtag* inerenti *reality show*, come Grande Fratello, Amici e The Voice of Italy. La loro presenza può essere giustificata dal grande numero di messaggi di scherno rivolti ai partecipanti, inviati sul *social network* durante la messa in onda dei programmi televisivi.

Nella tabella 6.8 non sono stati deliberatamente rimossi i *seed* "*TROIA*" e "*Troia*", per evidenziare come le differenze tra i termini scritti in maiuscolo ed in minuscolo siano da considerare in questo genere di analisi. Infatti, il primo *seed* risulta più significativo per la classe 4, probabilmente perchè più impiegato del secondo per offendere il prossimo.

Al di là di tutte queste considerazioni, è possibile osservare nelle classifiche alcuni termini che potrebbero avere alta correlazione con sentimenti di intolleranza ("*cazzo*" e "*culo*" per omofobia, "*bocca*", "*inchiavabile*" e "*vergine*" per misoginia). Risulta necessario indagare maggiormente su di essi per cercare di comprendere che utilità possano avere in fase di raccolta di nuovi dati ed in che modo possano essere utilizzati per ricoscere forme di intolleranza. Stessa attenzione deve essere posta sui termini che non possono essere certamente riconducibili a forme di odio e che vanno esclusi in fase di *crawling*. Un esempio palese è offerto da "*tisana*", spesso usato insieme a "*finocchio*" in ambito culinario.

Parte III

Conclusione

Lesson-learnt

I risultati degli esperimenti mostrati nell'ultimo capitolo permettono di formulare alcune conclusioni su questo caso di studio.

La prima fase del lavoro, orientata alla valutazione delle performance dei *cluster* di Hadoop su Compute Engine, ha dimostrato quando è opportuno ricorrere a *framework* per il calcolo distribuito in combinazione con prodotti *cloud*.

La criticità nella gestione dei *big data* è stata del tutto superata per merito del modello di programmazione MapReduce e della scalabile e performante infrastruttura di Google. Hadoop si è rivelata essere una tecnologia adatta per elaborare grandi moli di informazioni ed il suo utilizzo è, pertanto, suggerito sia per il progetto della Mappa dell'Intolleranza, sia per qualsiasi altra applicazione *data intensive*. Nuovi *task* della piattaforma richiederanno, tuttavia, una scrittura *ex novo* delle funzioni *map* e *reduce*: queste, infatti, sono molto specifiche per i compiti da portare a termine e difficilmente riusabili.

I dati raccolti dimostrano anche come gli approcci tradizionali siano migliori per *dataset* di dimensioni ridotte. Per tale motivo bisogna sempre valutare attentamente, in fase di progettazione, se le tecnologie presentate siano effettivamente necessarie.

L'esperimento sull'analisi dei *tweet*, invece, ha permesso di comprendere meglio la natura di questi messaggi e le direzioni verso cui orientare i lavori futuri.

I risultati evidenziano, innanzitutto, come la fase di raccolta dei dati vada raffinata. Effettuare il campionamento limitandosi ad intercettare nel flusso di messaggi i *tweet* contenenti una o più *keyword* non è decisamente sufficiente. Il *crawler* deve essere supportato da filtri capaci di bloccare, oltre allo spam, i falsi positivi. I termini emersi con la KLD possono costituire un buon punto di partenza per discernere i messaggi offensivi da quelli inoffensivi. Diverse *keyword*, inoltre, si sono rivelate poco utili per la raccolta di *tweet* con contenuto intollerante (ad esempio "*nano*"). Raffinamenti dei *seed set* sono consigliati per una raccolta più mirata dei messaggi. Gli stessi benefici si potrebbero anche ottenere utilizzando semplici euristiche basate

sull'analisi di bigrammi o *sliding window* con i *seed*, come già evidenziato in fase di discussione dei risultati dell'esperimento.

Le difficoltà emerse durante il procedimento di analisi dei dati palesano la necessità di sviluppare degli strumenti più efficaci sui *tweet*. La loro natura richiede delle tecniche di elaborazione tarate sul particolare *language model*. Metodi in grado di apprenderlo automaticamente potrebbero rendere più precisa la Mappa dell'Intolleranza e, pertanto, il loro studio è suggerito per nuovi lavori.

Resta da comprendere, infine, se la presenza di molti *tweet* su *trendy topic* costituisca un problema per le analisi e se sia necessario sviluppare degli strumenti per controllare il flusso temporale dei messaggi.

Sviluppi futuri

Il caso di studio presentato si presta ad essere espanso in diverse direzioni oltre a quelle già suggerite nel paragrafo precedente. La piattaforma sviluppata, infatti, può essere impiegata in molti modi ed offre una collezione di strumenti per integrare facilmente nuovi moduli con i prodotti della Google Cloud Platform.

I primi componenti da aggiungere alla piattaforma sono certamente quelli destinati ai *task* di *data visualization*, già sviluppati nel corso di altri lavori. Questi si rendono necessari per visualizzare la Mappa dell'Intolleranza e le zone maggiormente affette dai problemi di discriminazione. È auspicabile che tali moduli facciano uso delle tecnologie approfondite nel caso di studio per beneficiare di grande scalabilità.

Big Query, un altro prodotto della Google Cloud Platform, potrebbe essere impiegato per fornire analisi in tempo reale sulle situazioni di intolleranza in divenire nel nostro Paese. La sua capacità di processare molti terabyte di dati in pochi secondi potrebbe fornire alle amministrazioni locali uno strumento più potente per monitorare nascenti episodi di discriminazione, dal momento che le notizie sui *social network* si diffondono in tempi molto brevi.

L'analisi statistica dei *tweet*, come è stato dimostrato, non fornisce mezzi sufficientemente efficaci per combattere le intolleranze: si rende necessario, dunque, uno strumento capace di condurre indagini sulla semantica dei messaggi. Annotando manualmente dei campioni piuttosto grandi di *tweet*, si potrebbero ottenere corpora di messaggi discriminatori (possibilmente uno per ogni classe) da impiegare per la costruzione di un classificatore in grado di determinare se nuovi *tweet* presentano tracce di intolleranza oppure no.

Una prevenzione più efficace degli episodi di discriminazione si potrebbe ottenere ampliando il raggio d'azione delle analisi alla natura del grafo del *social network*. Se, ad esempio, molti messaggi di carattere omofobo provenissero dallo stesso utente si potrebbe ritenere opportuno analizzare anche gli altri suoi precedenti messaggi, per determinare se la persona, in effetti, prova e manifesta sistematicamente sentimenti di odio verso gli omosessuali. In caso affermativo, si potrebbero anche condurre le stesse analisi sulle persone

a lei correlate (*followers* su Twitter, amici su Facebook, ecc.) per individuare tra di loro altri individui potenzialmente pericolosi. Così facendo potrebbe emergere, ad esempio, una rete di omofobi, le cui attività dovrebbero essere monitorate con maggiore attenzione. I profili di questi utenti potrebbero, inoltre, migliorare il classificatore precedentemente ipotizzato, arricchendo il numero di *feature* nel *training set*.

Diversi studi [15] [14] [31] [24], infine, hanno dimostrato come sia possibile predire, con buona affidabilità, le personalità degli individui analizzando i loro messaggi su Facebook e Twitter. Gli strumenti sviluppati si sono rivelati in grado di quantificare i *Big Five personality traits* (*openness to experience*, *conscientiousness*, *extraversion*, *agreeableness*, *neuroticism*), cinque ampie dimensioni della personalità, solitamente non sovrapposte negli esseri umani e valide per soggetti di sesso, età e culture differenti. Supponendo di aver individuato persone realmente pericolose, questi strumenti potrebbero essere impiegati per indagare sui tratti della loro personalità. Potrebbe così emergere, per esempio, che i razzisti hanno generalmente livelli bassi di estroversione, oppure che l'odio contro le donne è spesso manifestato in individui caratterizzati da assenza di apertura verso nuove esperienze. Queste informazioni potrebbero essere utilizzate per delineare un profilo del razzista, dell'omofobo, ecc.

Ringraziamenti

Il lavoro appena presentato ed il mio percorso di studi universitario non sarebbero stati gli stessi senza il fondamentale aiuto di diverse persone.

Non c'è modo migliore per concludere questa tesi e questo capitolo della mia vita se non ringraziandole.

A tutti i membri del gruppo di ricerca dello SWAP ed, in particolar modo, ai Professori Pasquale Lops e Giovanni Semeraro va la mia gratitudine per il prezioso tempo dedicatomi prima e durante questo lavoro, nonostante i tantissimi impegni. Vi ringrazio, soprattutto, per aver coltivato il mio interesse verso questa area di studio e per avermi consentito di svolgere un lavoro innovativo ed interessante, lasciandomi sufficiente libertà d'azione. Mi auguro di tornare ad avere in futuro il piacere di lavorare nel vostro brioso ambiente, dove la ricerca scientifica viene trattata con la serietà che merita.

Ad Alessandro, mio leale ed inestimabile amico, per aver arricchito questo triennio con la sua infinita disponibilità nei miei confronti, per avere acceso in me quel sano spirito di competizione che mi ha stimolato a dare il massimo ad ogni esame, per essere stato l'interlocutore di tanti accesi e costruttivi dibattiti, per essere stato il mio vero ed unico Prof di programmazione, per aver condiviso quotidianamente con me le sue idee ed il suo sapere e per essere stato il faro nei momenti più bui.

A Federico, Checco, Marta e Lorenzo per avermi aiutato a completare, correggere e rifinire la tesi e la presentazione. Il vostro aiuto è stato davvero essenziale.

Ai membri vicini, più lontani e sempre in viaggio della mia famiglia, nonchè ai miei amici più intimi, per aver dimostrato costanti attenzioni nei miei riguardi: le vostre premure ed il vostro affetto non saranno mai dimenticati.

A mamma, papà e Checco, per la sconfinata pazienza dimostrata nei miei confronti e per avermi donato, giorno dopo giorno, la serenità che mi ha permesso di completare con successo il mio percorso di studi: sapete che vi voglio un bene smisurato e che questo traguardo lo stiamo tagliando tutti insieme.

A tutti voi vanno i miei più sinceri ringraziamenti.

Bibliografia

[1] Accenture. *Cloud-Based Hadoop: The Future of Big Data?* Rapp. tecn. 2014.

[2] Alexa. *The top 500 sites on the web. @ONLINE*. Set. 2014. URL: http://www.alexa.com/topsites.

[3] The Main Street Analyst. *The Growth Of The Internet Over The Past 10 Years – Infographic @ONLINE*. Set. 2014. URL: http://www.themainstreetanalyst.com/the-growth-of-the-internet-over-the-past-10-years-infographic/.

[4] Marco Baroni et al. *The WaCky Wide Web: A Collection of Very Large Linguistically Processed Web-Crawled Corpora*. Rapp. tecn. 2006.

[5] Fay Chang et al. «Bigtable: A Distributed Storage System for Structured Data». In: *ACM Trans. Comput. Syst.* 26.2 (giu. 2008), 4:1–4:26. ISSN: 0734-2071. DOI: 10.1145/1365815.1365816. URL: http://doi.acm.org/10.1145/1365815.1365816.

[6] Andrew Clement. «Re-thinking Networked Privacy, Security, Identity and Access Control in Our Surveillance States». In: *Proceedings of the 19th ACM Symposium on Access Control Models and Technologies*. SACMAT '14. London, Ontario, Canada: ACM, 2014, pp. 185–186. ISBN: 978-1-4503-2939-2. DOI: 10.1145/2613087.2613089. URL: http://doi.acm.org/10.1145/2613087.2613089.

[7] Organisation Intergouvernementale de la Convention du Mètre. *The International System of Units (SI), 8th ed.* Rapp. tecn. 2006.

[8] Jeffrey Dean e Sanjay Ghemawat. «MapReduce: Simplified Data Processing on Large Clusters». In: *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*. OSDI'04. San Francisco, CA: USENIX Association, 2004, pp. 10–10. URL: http://dl.acm.org/citation.cfm?id=1251254.1251264.

[9] Nick Dimiduk e Amandeep KhuranaKorn. «HBase in Action». In: Manning Publications Co., 2012. Cap. 1. ISBN: 9781617290527.

[10] Vox - Osservatorio Italiano sui Diritti. *Vox lancia la "Mappa dell'Intolleranza" @ONLINE*. Set. 2014. URL: http://www.voxdiritti.it/?p=2353.

[11] DB-Engines. *Knowledge Base of Relational and NoSQL Database Management Systems @ONLINE*. Set. 2014. URL: http://db-engines.com/.

[12] Forbes. *Top 10 Entrepreneurs In 2013 @ONLINE*. Set. 2014. URL: http://www.forbes.com/sites/drewhendricks/2013/12/17/top-10-entrepreneurs-in-2013/.

[13] Sanjay Ghemawat, Howard Gobioff e Shun-Tak Leung. «The Google file system». In: *Proceedings of the 19th ACM Symposium on Operating Systems Principles 2003, SOSP 2003, Bolton Landing, NY, USA, October 19-22, 2003*. 2003, pp. 29–43. DOI: 10.1145/945445.945450. URL: http://doi.acm.org/10.1145/945445.945450.

[14] Jennifer Golbeck, Cristina Robles e Karen Turner. «Predicting personality with social media». In: *Proceedings of the International Conference on Human Factors in Computing Systems, CHI 2011, Extended Abstracts Volume, Vancouver, BC, Canada, May 7-12, 2011*. 2011, pp. 253–262. DOI: 10.1145/1979742.1979614. URL: http://doi.acm.org/10.1145/1979742.1979614.

[15] Jennifer Golbeck et al. «Predicting Personality from Twitter». In: *PASSAT/SocialCom 2011, Privacy, Security, Risk and Trust (PASSAT), 2011 IEEE Third International Conference on and 2011 IEEE Third International Conference on Social Computing (SocialCom), Boston, MA, USA, 9-11 Oct., 2011*. 2011, pp. 149–156. DOI: 10.1109/PASSAT/SocialCom.2011.33. URL: http://doi.ieeecomputersociety.org/10.1109/PASSAT/SocialCom.2011.33.

[16] The Guardian. *Facebook: 10 years of social networking, in numbers @ONLINE*. Set. 2014. URL: http://www.theguardian.com/news/datablog/2014/feb/04/facebook-in-numbers-statistics.

[17] Zan Huang, Wingyan Chung e Hsinchun Chen. «A Graph Model for E-commerce Recommender Systems». In: *J. Am. Soc. Inf. Sci. Technol.* 55.3 (feb. 2004), pp. 259–274. ISSN: 1532-2882. DOI: 10.1002/asi.10372. URL: http://dx.doi.org/10.1002/asi.10372.

[18] Facts Hunts. *Tit @ONLINE*. Set. 2014. URL: http://www.factshunt.com/2014/01/total-number-of-websites-size-of.html.

[19] IBM, Paul Zikopoulos e Chris Eaton. *Understanding Big Data: Analytics for Enterprise Class Hadoop and Streaming Data*. 1st. McGraw-Hill Osborne Media, 2011. Cap. 1, pp. 3, 5. ISBN: 0071790535, 9780071790536.

[20] IDC. *The Expanding Digital Universe*. Rapp. tecn. 2007.

[21] Indeed. *NoSQL job trends @ONLINE*. Set. 2014. URL: `http://www.indeed.com/jobtrends/nosql.html`.

[22] InfoWorld. *Ultimate cloud speed tests: Amazon vs. Google vs. Windows Azure @ONLINE*. Set. 2014. URL: `http://www.infoworld.com/article/2610403/cloud-computing/ultimate-cloud-speed-tests--amazon-vs--google-vs--windows-azure.html`.

[23] Digital Insights. *Social Media 2014 Statistics – an interactive Infographic you've been waiting for! @ONLINE*. Set. 2014. URL: `http://blog.digitalinsights.in/social-media-users-2014-stats-numbers/05205287.html`.

[24] Michal Kosinski, David Stillwell e Thore Graepel. «Private traits and attributes are predictable from digital records of human behavior». In: 2013. URL: `https://4f46691c-a-dbcb5f65-s-sites.googlegroups.com/a/michalkosinski.com/michalkosinski/pnas2013.pdf?`.

[25] Chuck Lam. *Hadoop in Action*. Manning Publications Co., 2010. Cap. 1, 2, 3, 4. ISBN: 9781935182191.

[26] Adam Lith e Jakob Mattsson. «Investigating storage solutions for large data - A comparison of well performing and scalable data storage solutions for real time extraction and batch insertion of data». Tesi di laurea mag. 2010.

[27] C.D. Manning, P. Raghavan e M. Schütze. *Introduction to Information Retrieval*. Cambridge University Press, 2008, p. 260.

[28] Gordon E. Moore. «Readings in Computer Architecture». In: a cura di Mark D. Hill, Norman P. Jouppi e Gurindar S. Sohi. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2000. Cap. Cramming More Components Onto Integrated Circuits, pp. 56–59. ISBN: 1-55860-539-8. URL: `http://dl.acm.org/citation.cfm?id=333067.333074`.

[29] Jakob Nielsen. *Nielsen's Law of Internet Bandwidth @ONLINE*. Set. 2014. URL: `http://www.nngroup.com/articles/law-of-bandwidth/`.

[30] Preshing on programming. *A Look Back at Single-Threaded CPU Performance @ONLINE*. Set. 2014. URL: `http://preshing.com/20120208/a-look-back-at-single-threaded-cpu-performance/`.

[31] Daniele Quercia et al. «Our Twitter Profiles, Our Selves: Predicting Personality with Twitter». In: *PASSAT/SocialCom 2011, Privacy, Security, Risk and Trust (PASSAT), 2011 IEEE Third International Conference on and 2011 IEEE Third International Conference on Social Computing (SocialCom), Boston, MA, USA, 9-11 Oct., 2011*. 2011, pp. 180–185. DOI: 10.1109/PASSAT/SocialCom.2011.26. URL: http://doi.ieeecomputersociety.org/10.1109/PASSAT/SocialCom.2011.26.

[32] The Social Skinny. *100 social media statistics for 2012 @ONLINE*. Set. 2014. URL: http://thesocialskinny.com/100-social-media-statistics-for-2012/.

[33] Internet World Stats. *World Internet Users and Population Stats @ONLINE*. Set. 2014. URL: http://www.internetworldstats.com/stats.htm.

[34] Your Story. *Ten Features that make Google Compute Engine (GCE) better than AWS @ONLINE*. Set. 2014. URL: http://yourstory.com/2013/12/google-compute-engine-better-than-aws/.

[35] Ars Technica. *Information explosion: how rapidly expanding storage spurs innovation @ONLINE*. Set. 2014. URL: http://arstechnica.com/business/2011/09/information-explosion-how-rapidly-expanding-storage-spurs-innovation/.

[36] TIME. *Person of the Year: You @ONLINE*. Set. 2014. URL: http://content.time.com/time/covers/0,16641,20061225,00.html.

[37] TIME. *You — Yes, You — Are TIME's Person of the Year @ONLINE*. Set. 2014. URL: http://content.time.com/time/magazine/article/0,9171,1570810,00.html.

[38] The New York Times. *How Companies Learn Your Secrets @ONLINE*. Set. 2014. URL: http://www.nytimes.com/2012/02/19/magazine/shopping-habits.html.

[39] The Verge. *Amazon sold 426 items per second in run-up to Christmas @ONLINE*. Set. 2014. URL: http://www.theverge.com/2013/12/26/5245008/amazon-sees-prime-spike-in-2013-holiday-season.

[40] Hadoop Wiki. *PoweredBy Hadoop @ONLINE*. Set. 2014. URL: http://wiki.apache.org/hadoop/PoweredBy.

[41] Alyson L. Young e Anabel Quan-Haase. «Information Revelation and Internet Privacy Concerns on Social Network Sites: A Case Study of Facebook». In: *Proceedings of the Fourth International Conference on Communities and Technologies*. C&T '09. University Park, PA, USA: ACM, 2009, pp. 265–274. ISBN: 978-1-60558-713-4. DOI: 10.1145/1556460.1556499. URL: http://doi.acm.org/10.1145/1556460.1556499.