Alma Mater Studiorum · Università di Bologna

CAMPUS DI CESENA
SCUOLA DI SCIENZE
Corso di Laurea in Ingegneria e Scienze Informatiche

Progettazione e prototipazione di un sistema di Social Business Intelligence con ElasticSearch

Relatore:
Prof.
Matteo Golfarelli

Correlatore:
Dott.
Enrico Gallinucci

Presentata da:
Luca Longobardi

Sessione II
Anno Accademico 2014/2015

PAROLE CHIAVE

Elasticsearch

NoSQL

Business Intelligence

Introduzione

Il presente elaborato ha come oggetto la progettazione e lo sviluppo di una soluzione Elasticsearch come piattaforma di analisi in un contesto di Social Business Intelligence. L'elaborato si inserisce all'interno di un progetto del Business Intelligence Group dell'Università di Bologna, incentrato sul monitoraggio delle discussioni online sul tema politico nel periodo delle elezioni europee del 2014. Le difficoltà riscontrate nella capacità di analizzare il grande volume di dati raccolto su una base dati relazionale Oracle sono alla base della ricerca di uno strumento pià adatto all'elaborazione dei cosiddetti Big Data. L'obiettivo del presente elaborato consiste nel valutare e confrontare le performance ottenute da una base dati relazionale Oracle, la quale modella il cubo relazionale, e da un sistema Elasticsearch imlementato su un cluster multinodo.
Nel primo capitolo viene svolta una panoramica sul movimento NoSQL, le esigenze dei moderni sistemi e come queste tecnologie possono risolvere il problema dei Big Data. In particolare vengono illustrate le proprietà fondamentali dei database NoSQL in relazione alle moderne basi di dati relazionali, e quali sono i principi fondamentali che guidano questi tipi di tecnologie. In seguito viene effettuata una presentazione dei quattro tipi principali di basi di dati NoSQL: i Key/Value store, i Columnar database, i Document Oriented database e i Graph database. Infine per ogni tipo di base di dati viene fornita una descrizione dettagliata sull'origine della tecnologia, sulle sue caratteristiche fondamentali in relazione alle altre basi di dati, e sulle sue principali applicazioni, fornendo un esempio di implementazione. Il secondo capitolo

si concentra principalmente sulla descrizione della piattaforma Elasticsearch. In primo luogo vengono esaminate le catatteristiche principali di scalabilità, orientamento agli oggetti e protocollo di comunicazione del sistema.
In seguito viene analizzata l'architettura logica di un database Elasticsearch, illustrando i concetti principali di indice, sistema master-slave e distribuzione dei nodi. Nella sezione successiva viene introdotta l'architettura fisica del sistema e in che modo i dati vengono distribuiti all'interno di un cluster, entrando nel dettaglio del concetto di affidabilità del sistema. Relativamente alla memorizzazione dei dati, viene descritto il concetto di orientamento ai documenti di Elasticsearch, con particolare attenzione ai tipi di dato supportati e ai concetti di campo full-text e mapping. Dopodichè viene introdotto il linguaggio di comunicazione di Elasticsearch, il JSON, e quali sono le API principali con cui è possibile interagire con il sistema, soffermandosi sulle operazioni di manipolazione dei documenti e sulle interrogazioni. Infine vengono illustrati i quattro pattern fondamentali che Elasticsearch fornisce per modellare un qualsiasi domino applcativo: Application-side join, Data denormalization, Nested Objects e Parent-Child relationship.
Il terzo e ultimo capitolo si focalizza interamente sul caso di studio trattato. Inizialmente è stata effettuata una introduzione alla Social Business Intelligence, con una descrizione del domino applicativo della Politica e di come è stata pensata l'architettura del sistema. In seguito sono stati analizzati i requisiti di dominio, e illustrate le motivazioni principali che hanno portato alla scelta della tecnologia Elasticsearch. Dunque è stato analizzato il sistema databasistico corrente, illustrando la sua architettura e la struttura dei dati in esso contenuti, per poi proporre una serie di metodologie di modellazione di dati per il sistema Elasticsearch. Scelta la metodologia, viene illustrata l'installazione del sistema su un cluster di sette nodi, e la procedura con cui i dati sono stati trasportati da un database Oracle ad Elasticsearch. Infine, attraverso un adeguato set di interrogazioni, sono state valutate le performance del sistema a fronte di un grande carico di lavoro, in modo da evidenziarne pregi e difetti.

Indice

Capitolo 1

I DBMS NoSQL

Accade sempre più spesso che le aziende, e non solo, abbiano la necessità di analizzare dati, da sempre prodotti in grande quantità, per gestire strutture decisionali di estrema importanza. Nel corso degli anni si sono resi disponibili dati che, per tipologie e numerosità, hanno contribuito a far nascere il fenomeno dei Big Data. Il termine Big Data viene applicato a tutti quei dati e informazioni che possono essere disponibili in enormi volumi, possono presentarsi in forma destrutturata e possono essere prodotti a velocità incredibili. In particolare i Big Data possono essere descritti dalle seguenti caratteristiche:

- Il **Volume** è una caratteristica fondamentale dei big data; una quantità incredibile di dati viene generata ogni giorno, sono Twitter e Facebook generano 7 TeraByte ogni giorno. Anche se è possibile memorizzare questi dati su database relazionali, sarebbe necessario investire sia per lo storage, sia per la capacità di calcolo necessaria ad analizzare una tale mole di dati.

- La **Varietà** rappresenta un'altra caratteristica fondamentale dei big data; con l'utilizzo di sensori, smartphone e social network i dati si sono complicati, ovvero non presentano più una struttura predefinita,

e dunque non sono più riconducibili ad un formato tabellare, ma possono presentarsi in formato semistrutturato o, nel peggiore dei casi, in formato completamente destrutturato.

- La **Velocità** un ulteriore caratteristica fondamentale dei big data; enormi quantità di dati vengono prodotti e resi disponibili in pochissimo tempo. Si rivela dunque necessario utilizzare tecnologie diverse, in grado di tenere il passo ad un tale flusso di informazioni.

- La **Veracità** rappresenta l'ultima caratteristica fondamentale dei big data; avere molti dati in volumi differenti e grandi velocità è inutile se i dati sono incorretti. Dati errati possono causare grandi problemi sia per le aziende che per i consumatori. Dunque un'organizzazione deve accertarsi non solo che i dati siano corretti, ma che anche l'analisi su questi ultimi sia efficace.

Negli ultimi anni l'esigenza di gestire enormi quantità di dati, caratterizzati dalle quattro V (*Volume, Varietà, Velocità e Veracità*) ha portato alla nascita di tecnologie che fanno capo ad un nuovo modello di progettazione: **NoSQL**. Il termine NoSQL, letteralmente Not only Sql o Not Relational, non si oppone all'utilizzo del modello relazionale, ma fa riferimento ad una grande varietà di tecnologie databasistiche che si discostano dal mondo relazionale, basato sul concetto di relazione e tabella.
Queste tecnologie sono state sviluppate come risposta sia al grande incremento del volume di dati memorizzati dagli utenti, sia alla frequenza con cui questi dati vengono acceduti, sia alle performance e alle esigenze di processazione. Dall'altro lato i database relazionali non sono stati progettati per cooperare con le grandi sfide di scalabilità e agilità che le moderne applicazioni mettono in campo, inoltre non sono costruiti in modo da sfruttare i bassi costi delle memorie e la grande potenza dei processori.
A causa della loro struttura, generalmente i database relazionali scalano verticalmente; un singolo server deve memorizzare l'intero database per assicurare affidabilità e continua disponibilità dei dati. Questo porta ad un grande

aumento dei costi, pone un limite allo scaling e crea un numero limitato di failure-points per l'infrastruttura del database. La soluzione più ovvia consiste nello scalare orizzontalmente aggiungendo server invece di concentrarsi sull'aumento delle capacità del singolo nodo, dividendo il contenuto del database tra più nodi; questo processo viene comunemente chiamato sharding di un database.
Per effettuare Sharding di un database relazionale tra più nodi è spesso necessario utilizzare techiche specifiche per fare in modo che tutto l'hardware agisca come un singolo server. Dato che i moderni RDBMS non forniscono questa funzionalità nativamente, spesso vengono sviluppate più basi di dati relazionali distribuite su un certo numero di nodi; i dati vengono memorizzati su ogni database in modo autonomo. In questo modo le applicazioni devono essere specificatamente sviluppate per distribuire dati e interrogazioni, e per effettuare aggregazioni cross-database. Inoltre alcuni dei benefici di una base di dati relazionale, come il supporto per le transazioni, vengono persi o eliminati quando si implementa lo Sharding manualmente. Dall'altro lato i database NoSQL supportano nativamente lo Sharding, ovvero effettuano automaticamente la distribuzione dei dati attraverso un numero arbitrario di server; in questo caso l'applicazione non si deve preoccupare nemmeno della struttura fisica della base dati. Dati e interrogazioni vengono automaticamente bilanciati tra i nodi; inoltre quando un nodo fallisce può essere sostituito in modo semplice e trasparente.
Al crescere della quantità di dati i problemi di scalabilità e di costi legati all'infrastruttura dei database relazionali sono solo una parte degli svantaggi; molto spesso, avendo a che fare con i big data, la variabilità, ovvero la mancanza di una struttura fissa, rappresenta un grande problema.
I database NoSQL, al contrario dei moderni RDBMS, permettono di gestire dati schema-less, ovvero che non possiedono una particolare struttura. I database NoSQL puntano sulla flessibilità e sulla capacità di gestire dati con strutture difficilmente rappresentabili in formato tabellare. La definizione di database NoSQL, riportata sul sito ufficiale, presenta una serie di

caratteristiche che contraddistinguono questi sitemi:

- Distributed
- Open-source
- Horizontally scalable
- Schema free
- Easy replication support
- Simple API
- Eventually consistent/ BASE model
- Not ACID property
- Huge amount of data

Generalmente non tutte le caratteristiche vengono rispettate dalle singole implementazioni delle basi di dati NoSQL.
La natura distribuita di questi sistemi fa si che le proprietà ACID (Atomicity, Consistency, Isolation e Durability) che caratterizzano i database relazionali non siano applicabili in questo contesto; questa è una conseguenza inevitabile del **teorema CAP**(Consistency, Availability, Partition tolerance), il quale afferma l'impossibilità per un sistema distribuito di garantire simultaneamente consistenza, disponibilità e tolleranza di partizione, ma è in grado di soddisfarne al massimo due.
Le tre proprietà sopracitate vengono definite in questo modo:

- Consistenza: tutti i nodi del sistema vedono gli stessi dati nello stesso istante
- Disponibilità: tutti i nodi del sistema devono essere in grado di garantire che ogni richiesta riceva una risposta che indichi il suo successo o il suo fallimento.

- Tolleranza di partizione: il sistema deve essere in grado di operare a fronte di una qualsiasi partizione dovuta sia a fallimenti di nodi sia all'aggiunta o rimozione degli stessi.

I database NoSQL, non potendo garantire per definizione le proprietà ACID, utilizzano un modello meno restrittivo, chiamato BASE model. Questo modello concilia la flessibilità offerta dai sistemi NoSQL con l'amministrazione di dati non strutturati. Il modello BASE consiste di tre principi:

- Basic Availability:l'approccio NoSQL si concentra sulla disponibilità dei dati anche in presenza di fallimenti multipli; raggiunge questo obiettivo utilizzando un approccio altamente distribuito. Invece di mantenere un singolo grande data-store e garantirne la resistenza ai guasti, i database NoSQL distribuiscono i dati tra molti nodi con un grande grado di replicazione. Nel caso in cui un fallimento distrugga l'accesso ad un particolare segmento di dati, il database sarà comunque in grado di ripristinare il guasto.

- Soft State: le basi di dati BASE abbandonano completamente i principi di consistenza dettati dal modello ACID. Uno dei concetti base del modello BASE asserisce che il problema della consitenza deve essere risolto dallo sviluppatore: la base di dati non si assume questo tipo di responsabilità.

- Eventual Consistency: l'unica garanzia che le basi di dati NoSQL forniscono sulla consistenza consiste nell'assicurarsi che, al momento di un fallimento, i dati convergeranno in uno stato consistente. Questa è una completa separazione con il modello ACID, il quale proibisce alle transazioni di terminare prima che le transazioni precedenti abbiano completato l'esecuzione e la base dati abbia raggiunto uno stato consistente.

Il modello BASE non è appropriato in ogni contesto, ma è certamente un'alternativa flessibile al modello ACID per tutti quei database che non

aderiscono strettamente al modello relazionale. Fino ad ora abbiamo parlato dei vantaggi del modello NoSQL; esistono però due svantaggi molto importanti che bisogna tenere in considerazione al momento della scelta tecnologica da intraprendere per un determinato progetto.

Figura 1.1: Database relazionali e NoSQL a confronto

	Database Relazionali	**Database NoSQL**
Storia	Sviluppati a partire dagli anni '70 per amministrare la prima ondata di dati.	Sviluppati a partire dagli anni 2000 per affrontare le limiazioni dei database relazionali, comprendenti in particolare lo scaling, la replicazione e i dati non strutturati.
Esempi	MySQL, Postgres e Oracle DB	MongoDB, Cassandra, HBase e Elasticsearch.
Modello di memorizzazione dati	Classico modello relazionale; i dati vengono memorizzati in tabelle e le gerarchie vengono ricostruite mediante operazioni di "Join"	L'approccio varia fortemente in base al tipo di base di dati: Column-Oriented, Key/value, i Document-Oriented e i Graph.
Schemi	La struttura e i tipi di dato sono fissati in anticipo. Per memorizzare informazioni riguardanti un nuovo tipo di dato, spesso deve essere modificata una parte consistente della base dati.	La struttura e i tipi di dato sono fortemente dinamici. I records possono aggiungere nuove informazioni in qualsiasi momento, e al contrario dei database relazionali, dati di tipo diverso possono essere memorizzati insieme se necessario.
Scaling	Verticale, ovvero ogni server deve essere aumentato di potenza per far fronte ad una domanda crescente. E' possibile dividere una base relazionale in più nodi, ma è necessario un grande livello di ingegnerizzazione.	Orizzontale, ovvero per aggiungere capacità un dbadmin può semplicemente incrementare il numero di nodi. Il database distribuisce automaticamente i dati se necessario.
Modello di sviluppo	Mix di Open Source (Postgres, MySQL) e Closed Source (Oracle).	Principalmente Open Source.
Supporto di transazioni	Si, i database relazionali offrono completo supporto per le transazioni.	Solamente in alcune circostanze (per esempio Elasticsearch fornisce supporto solo a livello di documento).
Manipolazioni di dati	Linguaggio specifico e standard: SQL.	Generalmente attraverso API Object-Oriented, altrimenti attraverso linguaggi specifici per ogni prodotto.
Consistenza	Può essere configurato per garantire un livello di consistenza elevatissimo.	Dipende dal prodotto. Alcuni garantiscono un forte livello di consistenza (MongoDB), mentre altri offrono l'eventual consistency (Cassandra).

Se da un lato i tempi di risposta di una base di dati non relazionale superano ampiamente quelli di una base di dati relazionale, dall'altro queste prestazioni vanno a scapito della **replicazione dei dati**. In realtà grazie ai costi sempre meno proibitivi dei dispositivi di storage questo problema è e diventerà sempre meno importante con il passare del tempo. Il vero problema è costituito dalla mancanza di uno **standard universale**, come per esempio il linguaggio SQL che caratterizza le basi di dati relazionali: ogni database appartenente al mondo NoSQL mette a disposizione un set molto specifico di API e di strumenti di storing che dipendono dalla specifica implementazione che si prende in considerazione. Come si evince dalla figura 1.1, non esiste un'unica tipologia di implementazione per i database NoSQL, ma vengono classificati in base a come i dati vengono memorizzati. Anche se esistono molte implementazioni diverse, è possibile individuare quattro categorie principali:

- **Column-Oriented database**
- **Key/value store**
- **Document Oriented Database**
- **Graph Database**

Di seguito forniamo una breve descrizione di ognuna di queste categorie insieme alle principali implementazioni disponibili sul mercato.

1.1 Column Oriented Database

Un database Column Oriented è un DBMS che memorizza tabelle dati come sezioni di colonne invece che di righe. Per effettuare un paragone, gran parte dei DB relazionali memorizzano dati per righe. Questi database column oriented portano grandi vantaggi per i data warehouse, l'analisi di dati clinici e tutti quei sistemi in cui le aggregazioni sono computate su una grande quantità di dati simili. E' in effetti possibile fare uso dei benefici sia del

modello column oriented sia del modello row oriented con qualsiasi DBMS; la notazione column-oriented database indica tutti quei sistemi ottimizzati per memorizzare ed effettuare operazioni su colonne di dati. Per esempio se volessimo memorizzare i seguenti dati:

Figura 1.2: Dati relazionali

Nome	Cognome	Occupazione	Salario
Luca	Longobardi	Ingegnere	40000
Marina	Londei	Grafico	50000
Rossi	Rossini	Panettiere	20000

In una base di dati relazionale i dati verrebbero memorizzati nel seguente modo:

Luca, Longobardi, Ingegnere, 40000
Marina, Londei, Grafico, 50000
Rossi, Rossini, Panettiere, 20000

Invece in un database di tipo Column Oriented i dati verrebbero memorizzati come:

Luca, Marina, Rossi
Longobardi, Londei, Rossini
Ingegnere, Grafico, Panettiere
40000, 50000, 20000

Le organizzazione row oriented sono più efficienti quando molte colonne di una singola riga sono richieste nello stesso momento, e quando la lunghezza di riga è piccola, in modo da essere letta con un solo accesso a disco. Inoltre questo tipo di organizzazione è molto efficace per inserire nuove righe quando i dati per riga sono forniti contemporaneamente. Al contrario le organizzazioni column oriented sono molto efficienti quando un'aggregazione necessita

di essere computata su più righe e su un subset piccolo di colonne, perchè leggere quel piccolo subset di dati è più performante rispetto a leggere tutti i dati. Inoltre questo tipo di organizzazione è più efficiente quando nuovi valori di colonna devono essere aggiunti per tutte le righe, poichè una colonna può essere scritta efficacemente e rimpiazzare la vecchia colonna senza toccare la restante porzione di dati per ogni riga.
Nella pratica la memorizzazione di dati per riga viene utilizzata per workloads di tipo OLTP, poichè la complessità risiede su transazioni spesso molto interattive. Dall'altro lato la memorizzazione di dati per colonna è estremamente efficace per workloads di tipo OLAP, caratterizzati da un numero limitato di query molto complesse su una grande quantità di dati (spesso con ordine di grandezza di terabytes).
HBase è uno dei più popolari column oriented database, basato su Google BigTable e scritto in Java. Come riportato dalla documentazione, Big Table è stato creato per il supporto di applicazioni che necessitano grande scalabilità; questa tecnologia viene utilizzata per dati con grandezza dell'ordine dei petabytes. Questo database è progettato per operare su cluster di grandi dimensioni, utilizzando un modello di dati molto semplice che Google descrive come una mappa ordinata sparsa, distribuita e multidimensionale. Nonostante HBase si basi appunto sul modello di BigTable, è un progetto Apache e fa parte dell'ecosistema Hadoop. Tra i più famosi utilizzatori di HBase ricordiamo la piattaforma di messaging istantanea di Facebook, Linkedin e Netflix.
Un altro componente facente parte dell'ecosistema Hadoop è Cassandra, un column-oriented database tra i più utilizzati, anche da grandi aziende come eBay, Github e Reddit. Cassandra è un database altamente distribuito che si basa sui principi di scalabilità lineare e high availability: le performance di questo sistema aumentano linearmente all'aumentare del numero di nodi. E' importante notare che in questo tipo di sistema (al contrario di Elasticsearch, come vedremo nel prossimo capitolo) è di tipo masterless, ovvero non esiste un singolo punto di fallimento oppure un collo di bottiglia a livello di rete:

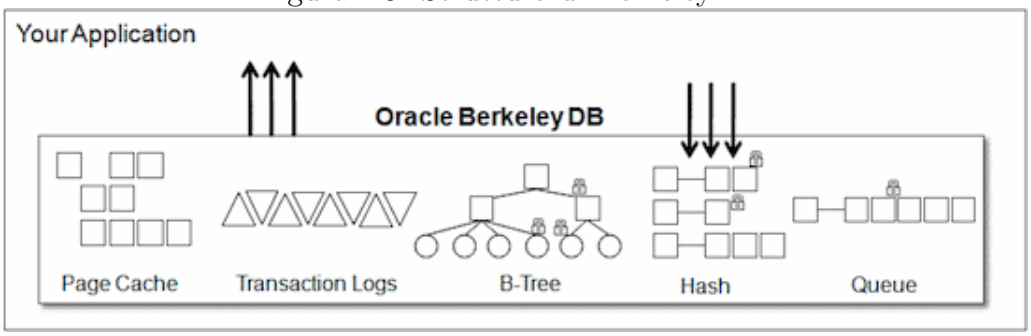
tutti i nodi all'interno del cluster sono identici.

1.2 Key/Value store

Un Key/value store, o Key/value database, è un paradigma di memorizzazione costruito per immagazzinare, ricercare e amministrare array associativi, una struttura dati comunemente conosciuta come dizionario o hash map. I dizionari possono contenere una collezione di oggetti, o di records, i quali possono avere strutture diverse. Questi records vengono memorizzati e ricercati utilizzando una chiave identificativa unica, in modo effettuare ricerche estremamente rapide a livello dell'intero database. I Key/valye database funzionano in modo molto diverso dai classici database relazionali. Gli RDB definiscono a priori la struttira del database attraverso una serie di tabelle contenenti un certo numero di campi con un tipo di dato ben definito. Esporre i tipi di dato esplicitamente permette di effettuare un certo numero di ottimizzazioni. Al contrario i Key/value store trattano i dati come una singola e opaca collezione, la quale può avere campi diversi per record diversi. Generalmente questi tipi di sistemi vengono utilizzati come base per l'implementazione sia di database document-oriented sia di graph database. Un esempio di Key/value store è **Berkeley DB**, attualmente gestito e aggiornato da Oracle e scritto in C. Questo sistema è in grado di memorizzare dati come opachi array di byte attraverso il paradigma chiave/valore, indicizzandoli in tre possibili modi: B-Tree, Hash o queue. Inoltre è possibile configurare il sistema per il supporto di transazioni ACID, permettendo di gestire gruppi di operazioni per essere considerate come un unico blocco che può o interamente fallire oppure completare con successo. In caso di fallimento del sistema, Berkeley DB è in grado di recuperare automaticamente uno stato consistente, eseguendo il rollback di tutte le transazioni parzialmente completate.Berkeley DB supporta inoltre grande scalabilità e availability, permettendo di effettuare replicazione a livello del singolo nodo. Tutti gli aspetti di integrità dei dati, garanzie transazionali, throughtput e trasporto

di rete sono completamente configurabili, rendendo di fatto il sistema altamente adattabile a seconda delle necessità. Inoltre questo sistema, a differenza della maggior parte dei database NoSQL, offre la possibilità di utilizzare il linguaggio SQL (SQLlite) sia per l'amministrazione che per l'interrogazione.

Figura 1.3: Struttura di Berkeley DB



1.3 Document Oriented Database

Un document oriented database è una base dati progettata per ordinare, ricercare e amministrare informazioni document-oriented, anche conosciute come dati semistrutturati. I database document oriented formano una sottoclasse dei Key/value store. La differenza tra i due risiede nel modo in cui i dati sono processati; in un key/value store il dato viene considerato opaco dalla base dati, mentre un sistema document-oriented utilizza la struttura interna del documento per estrapolare i metadati necessari ad ottimizzare il sistema di ricerca. I document vengono rappresentanti come un insieme strutturato di coppie chiave/valore, spesso organizzti in formato JSON o XML. La struttura delle coppie chiave/valore non pone vincoli allo schema dei documenti garantendo così una grande flessibilità in situazione in cui, per loro natura, i dati hanno una struttura variabile. I document oriented database sono in forte contrasto con i database relazionali tradizionali. I DBM necessitano di definire le strutture dei dati a priori attraverso la creazione di tabelle, mentre i document oriented database inferiscono le proprie strutture dai dati stessi, memorizzando automaticamente i tipi di dato simili insieme

per aumentare il livello di performance. Questa caratteristica rende questa classe di sistemi molto flessibili quando si tratta di amministrare l'aspetto della variabilità, caratteristica cruciale dei Big Data. Tra i sistemi document-oriented più famosi ricordiamo MongoDB e Elasticsearch.
Elasticsearch è uno dei principali database document-oriented; scritto in Java, il sistema rispetta la maggior parte dei criteri del paradigma NoSQL. Ogni record in Elasticsearch corrisponde ad un documento, ovvero una struttura dati composta da coppie key/value. I valori dei campi hanno una grande varietà di tipi, inclusi altri documenti, array oppure array di documenti. Ogni documento possiede una serie di metadati, tra cui il campo ?id?, che può essere assegnato in fase di inserimento, oppure, in mancanza di un valore, assegnato automaticamente dal sistema. Elasticsearch, per rappresentare i documenti, utilizza il formato JSON. Come molti degli strumenti NoSql, Elasticsearch gestisce autonomamente lo sharding e la replicazione della base dati, garantendo un alto livello di fault-tolerance e di availability, nonchè performance davvero impressionanti.

Figura 1.4: Struttura di un documento

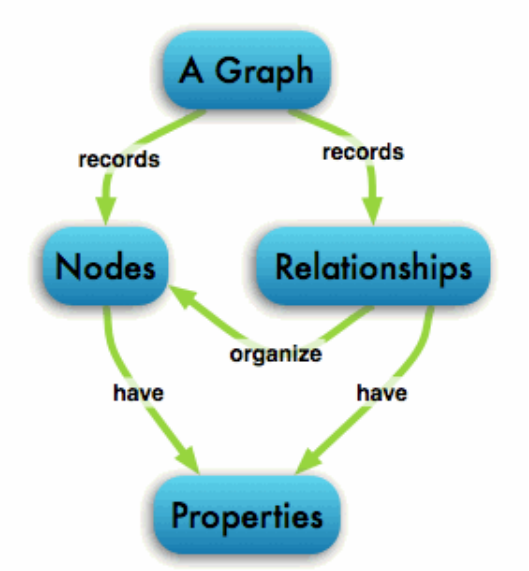
Elasticsearch contiene un potente motore di aggregazione che permette di raggiungere livelli di complessità di molto superiori al Group by relazionale. Per garantire un livello di performance estremamente alto delle operazioni

di aggregazione Elasticsearch fa uso di strutture dati molto complesse; lo spazio occupato da queste strutture cresce esponenzialmente con il numero di livelli dell'aggregazione, pertanto si rivela necessario un hardware sempre più potente al crescere della complessità.

1.4 Graph databases

I ghaph database sono una classe particolare di database NoSQL che utilizzano la struttura a *grafo* per mappare le relazioni tra i vari oggetti. Questi sistemi sono basati sulla teoria dei grafi, pertanto la struttura è composta da un insieme di *nodi* aventi un certo insieme di *proprietà* e collegati tra loro attraverso *archi*.

Figura 1.5: Graph database



La particolare struttura a grafo si presta molto bene per la rappresentazione di dati semistrutturati e altamente interconnessi come, ad esempio, i dati provenienti da Social Network o Web. Tutte le categorie di database NoSQL esaminati fino ad ora possiedono dei meccanismi per modellare relazioni, che però in generale portano ad ottenere interrogazioni potenzial-

mente molto complesse e poco efficienti, oppure ad anomalie di inserimento, cancellazione e modifica. I graph database sono stati pensati appositamente per modellare e navigare attraverso le relazioni tra oggetti, adattandosi facilmente ai cambiamenti delle strutture dati.

Tra le implementazioni di Graph database troviamo Neo4J; scritto in Java, questo sistema implementa il *Property Graph Model* (figura 1.5). Ogni record in Neo4j corrisponde ad una coppia chiave/valore, dove il valore può essere un nodo oppure un arco, i quali possono a loro volta avere un certo numero di proprietà che li descrivono. Come tutti i database NoSQL, questo sistema supporta grande scalabilità su cluster, un ottimo livello di fault-tolerance e ottime prestazioni. Inoltre Neo4j fornisce il supporto per transazioni ACID, rendendolo di fatto uno strumento davvero interessante in uno scenario di production.

Capitolo 2

Introduzione ad Elasticsearch

Elasticsearch è un database server, scritto in Java, che si occupa di memorizzare grandi quantità di dati in un formato sofisticato ottimizzato per ricerche basate sul linguaggio. Elasticsearch nasce come una evoluzione di Compass, applicazione realizzata dal fondatore del progetto Shay Banon nel 2004, basata sulla libreria Apache Lucene.Questo potente strumento è ottimizzato per la ricerca full-text, e contiene la maggior parte degli algoritmi di ricerca e memorizzazione di indici. Negli anni successivi Elasticsearch viene arricchito con funzionalità sempre più potenti, fino ad arrivare alla versione attuale 2.0. Di fatto, Elasticsearch è la prima tecnologia a proporre un sistema che permetta di eseguire ricerche full-text in modo perfettamente parallelo, scalabile e real-time. Per questa caratteristica e per il suo primato nel settore, Elasticsearch è il secondo motore di ricerca più popolare nel mondo. Tra i suoi utilizzatori più famosi ricordiamo Mozilla, GitHub, CERN, StackExchange e Netflix.
Mostriamo ora alcune caratteristiche fondamentali che costituiscono il nucleo di Elasticsearch:

- protocollo HTTP/JSON:Elasticsearch comunica utilizzando un linguaggio JSON particolare abbinato alle REST API. REST, ovvero REpresentational State Transfer è un particolare approccio architetturale che sfrutta il protocollo HTTP per il trasferimento di rete, rendendo di fat-

to standard l'interfaccia di comunicazione. Insieme ad HTTP troviamo JSON, utilizzato da Elasticsearch per costruire il proprio linguaggio di comunicazione. JSON, ovvero JavaScript Object Notation, è un modo per memorizzare informazioni in maniera organizzata e facilmente accedibile, garantendo che la collezione di dati sia semplice sia da leggere che da accedere.

- Scalabilità e affidabilità: Elasticsearch nasce come sistema di ricerca parallelo, perciò la capacità del database può essere aumentata semplicemente aggiungendo nodi al cluster. Inoltre Elasticsearch gestisce autonomamente il problema dell'affidabilità, migrando dati o promuovendo dati replica a dati primari al momento del malfunzionamento di un qualsiasi nodo.

- Analisi del linguaggio: Cuore delle funzionalità di Elasticsearch è senza dubbio il sistema automatizzato di analisi del linguaggio. Il database è in grado di analizzare porzioni di testo in qualsiasi lingua e formato, indicizzando in modo efficiente termini chiave e persino sinonimi, garantendo un sistema di ricerca dalle potenzialità incredibili.

- Query DSL: Elasticsearch utilizza un particolare linguaggio basato su JSON per codificare la struttura di una ricerca. A differenza del linguaggio SQL, il query DSL è componibile a piacere in modo da garantire la massima libertà di ricerca. Tuttavia se da una parte questa libertà permette di effettuare ricerche a qualsiasi livello, dall'altra spesso il modo di interrogare il database risulta piuttosto complesso, come vedremo nel capitolo dedicato al linguaggio.

2.1 Architettura del sistema

Procediamo analizzando l'architettura di Elasticsearch eseguendo un parallelo con le basi di dati relazionali.

Figura 2.1: Parallelo tra Oracle ed Elasticsearch

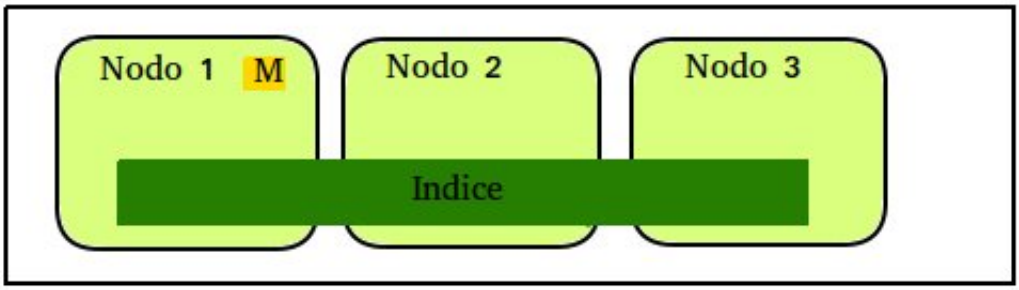
Elasticsearch	Indice	Tipo	Documento
Oracle	Database	Tabella	Tupla

Un indice è un meccanismo di organizzazione dati, che ci permette di partizionarli a piacere. Come descritto nella tabella 2.1, possiamo pensare ad un indice come l'equivalente di una base dati relazionale, ovvero un complesso contenitore di dati.
Ogni oggetto indicizzato in Elasticsearch appartiene ad una classe che ne definisce le proprietà e i dati ad esso associati. In un database relazionale solitamente memorizziamo oggetti della stessa classe nella stessa tabella, perchè condividono la stessa struttura. Di fatto in Elasticsearch il Type rappresenta la struttura di una classe di oggetti, dunque nella figura 2.1 il Type corrisponde ad una tabella in accezione relazionale. In parallelo con una base dati relazionale, così come un database contiene delle tabelle, un indice Elasticsearch può contenere più Type.
Molte entità o oggetti nella maggior parte delle applicazioni può essere serializzata in un oggetto JSON, utilizzando coppie chiave/valore. Spesso utilizziamo i termini oggetto e documento in modo intercambiabile. Esiste però una distinzione: un oggetto è semplicemente un oggetto JSON, in quale può contenere altri oggetti. In Elasticsearch, il termine documento ha un significato specifico, ovvero si riferisce all'oggetto radice serializzato in JSON e memorizzato in Elasticsearch con un ID univoco. Come possiamo notare in figura 2.1, facendo un parallelo con le basi di dati relazionali, un documento Elasticsearch corrisponde alla riga di una tabella in accezione relazionale.

Data la natura distribuita di Elasticsearch, la sua architettura pone al centro della complessità i Nodi e la loro organizzazione. Elasticsearch utilizza il paradigma Master-Slave per gestire la comunicazione tra i nodi. Quando una nuova istanza di Elasticsearch viene avviata, un nodo master viene scelto tra quelli disponibili attraverso una politica di elezione. Il nodo master è responsabile sia dell'organizzazione dei dati all'interno del cluster sia dell'ag-

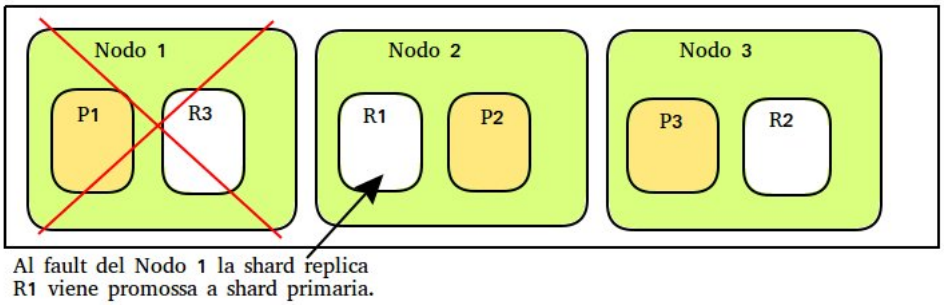
giunta e rimozione di un nodo attivo o malfunzionante. Se il nodo master smette di funzionare, Elasticsearch ne eleggerà uno nuovo.

Figura 2.2: Architettura master-slave



Fino ad ora abbiamo parlato di indici come semplici contenitori di dati che in qualche modo sono legati tra di loro. In realtà un indice è soltanto un namespace logico che punta ad una o più shard. Una shard è una unità di lavoro di basso livello che contiene una porzione di dati dell'indice. Ogni shard è una singola istanza Lucene, dunque è un sistema di ricerca autonomo. Le shard costituiscono lo strumento con cui Elasticsearch distribuisce i dati all'interno del cluster, ed è per questo che, come possiamo notare in figura, un indice può essere distribuito su più nodi.

Figura 2.3: Failure di un nodo



L'affidabilità è in generale il punto critico di tutte le basi di dati, sia relazionali che non. Per garantire un alto tasso di affidabilità nel mondo relazionale si tende a ridondare l'intero database, spesso locando le copie in luoghi differenti. Di fatto questo può causare numerosi problemi, dato che

al momento del fault bisognerà sostituire l'intera base di dati. Al contrario Elasticsearch utilizza un approccio completamente differente. Abbiamo detto che le shard sono lo strumento principale con cui Elasticsearch garantisce l'affidabilità. Un indice può contenere un qualsiasi numero di shard, le quali saranno distribuite all'interno del cluster. Possiamo distinguere due tipi di shard: primarie e replica; le replica sono delle semplici copie di shard primarie che permettono al cluster di continuare a funzionare anche in caso di fault o sostituzione di un nodo. Se un nodo qualsiasi smette di funzionare, le shard replica vengono promosse a primarie, e all'inserimento di un nuovo nodo verranno rigenerate le replica. Attraverso un particolare algoritmo Elasticsearch è in grado di garantire grande affidabilità distribuendo i dati in modo da garantire la maggior fault tolerance possibile. Dato che questo processo è completamente automatizzato, se da una parte l'amministratore non ha necessità di interagire con il sistema, dall'altra non esiste alcun modo per modificare questa configurazione. In figura 2.3 viene mostrata la promozione di shard replica a primarie.
Come effetto di questo meccanismo, all'interno di un cluster è possibile aggiungere e rimuovere nodi a piacimento senza alterarne in alcun modo il funzionamento. Questo rende di fatto Elasticsearch uno strumento dalle potenzialità enormi, permettendo al sistema di scalare orizzontalmente in maniera quasi automatica abbattendo sia i costi di manutenzione sia i costi che sarebbero necessari per mantenere personale competente.

Figura 2.4: Confronto proprietà tra Elasticsearch e RDBMS

	RDBMS	Elasticsearch
Scalabilità	Bassa: è generalmente difficile scalare orizzontalmente in un Database Relazionale a causa delle prestazioni proibitive di join cross-nodo.	Molto alta: Elasticsearch è in grado di effettuare uno scaling orizzontale elevato; l'unico limite è la disponibilità di shard, che va decisa al momento della creazione degli indici.
Manutenibilità	Media: sono spesso necessarie competenze specifiche per effettuare manutenzione su un DB relazionale.	Bassa: se un qualsiasi nodo fallisce, è sufficiente sostituirlo. Non sono necessarie competenze specifiche.
Affidabilità	Alta: i dati contenuti in un Database relazionale sono altamente affidabili. Questi sistemi sono costruiti proprio per questa ragione.	Media: Elasticsearch non effettua politiche di locking sui dati, e spesso l'ultimo update su un documento è quello rilevante. Non è un sistema adatto ad amministrare dati sensibili.

Fino ad ora abbiamo parlato dei vantaggi di avere un sistema perfettamente distribuito e scalabile, e quanto questo aspetto possa incidere sia sulle performance che sulla manutenibilità. Questo però pone un problema, già esistente e largamente trattato nei database relazionali: la concorrenza. Nel mondo relazionale questo problema viene risolto adottando il paradigma *Pessimistic concurrency control*. Questo approccio assume che i conflitti siano molto comuni e perciò utilizza politiche di lock complesse per prevenirli. Un tipico esempio è il blocco di una tupla prima della lettura dei suoi dati, in modo da garantire che soltanto il thread che ha generato il lock sia in grado di modificarla. Al contrario Elasticsearch utilizza il paradigma *Optimistic concurrency control*. Questo approccio assume che i conflitti accadano molto di rado e dunque non utilizza politiche di lock. In ogni caso, se il dato viene modificato tra una lettura e una scrittura, l'update fallirà. Sarà poi l'applicazione a decidere come risolvere il conflitto. Quando un documento viene creato, modificato o cancellato le nuove versioni devono essere propagate in tutti i nodi del cluster. Elasticsearch è anche asincrono e concorrente, per cui le modifiche vengono eseguite in parallelo e dunque potrebbero arrivare a destinazione fuori sequenza. Per risolvere questo problema Elasticsearch utilizza un sistema di versioning dei documenti, evitando che vecchie versioni

di un documento sovrascrivano quelle nuove arrivate appunto fuori sequenza. La diversità nel modo di operare delle due tecnologie trova le sue ragioni nel diverso scopo per cui sono state progettate: una base di dati relazionale nasce con l'obiettivo di fornire non solo un metodo sicuro per gestire la consistenza dei dati, ma anche di garantire che le operazioni generate da più sessioni in concorrenza equivalgano all'esecuzione seriale delle stesse. Elasticsearch al contrario nasce come sistema di ricerca ed analisi su una grande quantità di dati; è perciò importante mantenere elevate prestazioni a fronte di analisi complesse ed in tempo reale. Spesso non è necessario mantenere una stretta consistenza dei dati, poichè sarà molto raro incorrere in una modifica concorrente sulla stessa informazione. La scelta della tecnologia da utilizzare dipenderà dalle esigenze specifiche del dominio applicativo.

2.2 Descrizione dei componenti

In questa sezione effettueremo una descrizione dettagliata dei componenti principali di Elasticsearch.

Formato JSON e documenti

JSON (JavaScript Object Notation) è un linguaggio di scambio di dati, molto semplice sia da leggere che da scrivere per le persone. E' basato su un subset del linguaggio JavaScript, ed è completamente indipendente dai linguaggi di programmazione. JSON è costituito da due strutture: una collezione di coppie chiave/valore e una lista ordinata di valori. Queste strutture sono universali, e dunque supportate virtualmente da qualsiasi linguaggio di programmazione. Questa caratteristica rende JSON uno standard perfetto per la comunicazione tra applicazioni.
Elasticsearch è *document oriented*, ovvero il documento costituisce l'unità di elaborazione della base di dati. Di fatto Elasticsearch non si limita a memorizzare documenti, ma rende ricercabili i loro attributi attraverso un partico-

lare sistema di indicizzazione. A differenza delle basi di dati relazionali, dove le operazioni vengono effettuate sulle colonne delle tabelle, in Elasticsearch si indicizza, ordina, ricerca e filtra per documenti. Questa fondamentale caratteristica costituisce la ragione per cui Elasticsearch è in grado di eseguire complesse ricerche full-text. Un documento in Elasticsearch è rappresentato da un oggetto JSON, il quale contiene una collezione di coppie chiave/valore. La chiave rappresenta il nome del campo, mentre il valore può essere uno qualsiasi dei valori accettati da Elasticsearch:

Figura 2.5: Tipi di dato supportati a confronto

	String	Numeric	Date	Boolean	Binary	Array	Object	Geo-point
Elasticsearch	Si	Si	Si	Si	Si	Si	Si	Si
RDBMS	Si	Si	Si	Si	Si	No	No	Si

La tabella mostra anche la principale differenza sul trattamento dei dati tra Elasticsearch e modello relazionale: Elasticseach dà la possibilità di memorizzare come valore di un campo un qualsiasi altro oggetto, creando un sistema di gerarchia. Questi oggetti appartengono alla categoria dei Complex field types, di cui parleremo più avanti nel capitolo. Il vantaggio di possedere una struttura basata sui documenti risiede nel fatto di poter memorizzare interi oggetti come attributi del singolo oggetto radice, aumentando notevolmente la libertà di espressione rispetto alla piatta riga di una tabella. Nel modello relazionale questo violerebbe un vincolo della prima forma normale. Lo svantaggio principale di questo modello e che in realtà riguarda non solo Elasticsearch, ma l'intero mondo no-sql, risiede nell'estrema inefficienza della realizzazione del join. Dato che in Elasticsearch la granularità più fine non è il singolo attributo, ma l'intero documento, risulta praticamente impossibile emulare il join relazionale. Per ovviare a questo problema spesso nella pratica, come vedremo, si ricorre a tecniche di modellazione che dipendono in maniera specifica dal dominio applicativo di interesse, senza fornire una metodologia generica per la risoluzione del problema.

Un documento, oltre ai dati in esso contenuti, è caratterizzato dai metadata. I metadata contengono informazioni specifiche riguardo il documento. In generale tutte le basi di dati non relazionali document-oriented utilizzano i metadati per implementare funzioni specifiche per la ricerca di documenti. In particolare Elasticsearch utilizza tre metadati fondamentali:

- Index: questo metadata indica l'indice di appartenenza del documento. Dato che un indice, come abbimo detto, è solo un namespace logico che identifica un particolare gruppo di shard, index indica a quale gruppo di shard appartiene il documento.
- Type: il type rappresenta la classe a cui il documento appartiene. Ogni type avrà una sorta di schema definition, chiamato mapping, che descrive la struttura e le caratteristiche della classe di documenti. Possiamo paragonare il type alla tabella in accezione relazionale.
- ID: l'id è una semplice stringa che combinata con i metadati Index e Type permettono di identificare univocamente il documento.

Mapping, analisi e analizzatori

Uno degli strumenti fondamentali di Elasticsearch è il mapping. Il mapping corrisponde ad una sorta di schema definition per un particolare tipo. Possiamo paragonare il mapping alla definizione di una tabella nel mondo relazionale (uno statement del tipo create table...). In particolare il mapping definisce tutti i campi contenuti in un tipo, a quale tipo di dato corrisponde ogni campo e come ognuno di essi deve essere trattato:

Blogpost
title : { type : string, index:analyzed},
body : { type : string, index: analyzed},
comments : {
type : object ,
properties : {
name : { type : string, index : analzed},
comment : { type: string , index : analyzed}
age :{ type : long}
}
}

In generale Elasticsearch è in grado di generare il mapping autonomamente, seguendo un suo particolare standard o inferendo il tipo di dato dal dato stesso. Spesso però è necessario modificare o creare un mapping ad-hoc per modellare alcuni tipi di dato complessi oppure per precisare un tipo di dato ambiguo. Per esempio, di default elasticsearch mappa tutti i campi di tipo stringa come full-text; ciò significa che il testo sarà spezzato e reso ricercabile anche per sue parti. Se vogliamo invece che il nostro testo sia trattato come valore esatto, sarà necessario specificarlo nel mapping. Se al contrario desideramo che il nostro campo venga trattato come full-text, possiamo ottimizzare la ricerca specificando come il contenuto di quel campo debba rispondere a ricerche di rilevanza. Esamineremo questo caso nel prossimo capitolo descrivendo gli analyzers. Un altro caso nel quale spesso è necessario specificare il mapping riguarda i complex field types. Di fatto la possibilità di innestare campi multivalore (come array di valori) o interi oggetti all'interno di un documento rende Elasticsearch uno strumento tanto potente quanto complesso. Mostriamo in un esempio di nesting:

Blogpost
title : Nest eggs,
body: Making your money work...,

comments: {
name : John Smith,
comment : Great article,
age : 28,
},
{
name : Alice White,
comment : More like this please,
age : 31,
}

I campi di questo tipo sono accessibili scendendo la gerarchia attraverso l'oggetto padre. In ogni caso Lucene non conosce il tipo di dato object. Un documento Lucene consiste in una semplice lista di coppie chiave/valore. Il documento verrà appiattito per essere indicizzato, e si presenterà in questo modo:

title : [eggs, nest],
body : [making, money, work, your],
comments.name : [alice, john, smith,
white], comments.comment : [article, great, like, more, please, this],
comments.age : [28, 31]

La correlazione tra Jhon e la sua età, 28 anni, viene persa poichè ogni campo multivalore rappresenta soltanto un gruppo di valori, non un array ordinato. Ci sono casi in cui non è interessante mantenere la correlazione tra i valori, per cui si demanda la produzione di un mapping al sistema. Tuttavia nella maggior parte dei domini applicativi mantenere la correlazione è un requisito molto importante. Gli oggetti correlati vengono chiamati Nested Objects, e verranno esaminati nei prossimi capitoli.

I dati in Elasticsearch possono essere divisi in due tipi fondamentali: va-

lori esatti e full-text. I valori esatti rappresentano ciò che sono, come un identificatore o una data. Al contrario i valori full-text si riferiscono a dati di tipo testuale, spesso scritti in un certo linguaggio, come ad esempio un tweet oppure una e-mail. I valori esatti sono semplici da ricercare. La decisione è binaria; o il documento corrisponde ai criteri di ricerca oppure no. Ricercare per full-text invece è un problema molto complesso. Non ci stiamo soltanto chiedendo se il nostro documento corrisponde oppure no, ma anche quanto il nostro documento si avvicina al criterio di ricerca. Per facilitare questi tipi di ricerca Elasticsearch analizza il testo e utilizza il risultato per creare un Inverted index. Un inverted index consiste in una lista di termini unici che appaiono in ogni documento, e per ogni termine, la lista di documenti in cui compare. Per generare un inverted index Elasticsearch utilizza gli Analizzatori. L'analisi di un testo si compone di due fasi: tokenizzazione del blocco di testo per individuare le parole da inserire nell'inverted index e normalizzazione delle parole per aumentare la loro ricercabilità. Il compito di analisi viene svolto dagli analizzatori. Gli analizzatori svolgono tre funzioni fondamentali:

- Filtro di caratteri: il testo viene analizzato e tutti i caratteri ridondanti vengono cancellati o normalizzati (per esempio il simbolo & può essere sostituito dalla parola and).
- Tokenizzazione: la stringa viene divisa in parole singole attraverso uno specifico criterio. Per esempio il testo potrebbe essere diviso per spazi vuoti.
- Token filter: il testo viene ri-analizzato per rimuovere parole inutili (come congiunzion ecc.) e aggiungere sinonimi.

Elasticsearch fornisce una serie di analizzatori ad-hoc per la maggior parte dei linguaggi, ma è anche in grado di accettare un analizzatore costruito appositamente dallo sviluppatore. Quando progettiamo un mapping per un tipo è necessario specificare quale analizzatore utilizzare per ognuno dei campi di tipo full-text presenti. La specifica dell'analizzatore è molto importante

poichè determina in che modo il testo sarà ricercabile. Un analizzatore non adatto al caso spesso produrrà un risultato di ricerca inaspettato. Abbiamo detto che quando indicizziamo un documento, i suoi campi full-text vengono analizzati per produrre una serie di termini da inserire nell'inverted index per renderli ricercabili. In realtà anche quando effettuiamo una ricerca su un campo full text dobbiamo effettuare lo stesso processo sulla stringa di ricerca fornita, in modo da assicurarci che la ricerca sia effettuata su termini della stessa forma.

2.3 Macrofunzionalità

Fino ad ora abbiamo parlato di come Elasticsearch è costruito a livello architetturale e come questi componenti interagiscono per produrre un determinato risultato. In questa sezione volteremo pagina, incominciando a descrivere il funzionamento e le caratteristiche delle principali funzionalità che il sistema mette a disposizione dell'utente.

HTTP over JSON e operazioni sui documenti

Abbiamo detto che Elasticsearch utilizza il formato JSON sia per le comunicazione tra i nodi sia per le richieste provenienti dagli utenti. La potenza del formato JSON risiede nella semplicità sia di lettura che di scrittura, ma non solo; JSON è anche standard, ovvero tutti i moderni linguaggi di programmazione e sistemi possiedono un modo per comunicare informazioni in questo formato. Come mezzo per veicolare le informazioni Elastic ha scelto HTTP; questo protocollo presenta le stesse caratteristiche di JSON, ovvero semplicità d'uso e standardizzazione. Tutti i moderni sistemi possiedono i mezzi per comunicare in HTTP. In definitiva Elasticsearch espone un'interfaccia di comunicazione standard e semplice da utilizzare, rendendo il sistema facile da integrare indipendentemente dalle tecnologie che ne faranno uso. Il problema principale di questo approccio consiste nella sicurezza; non esiste

infatti un sistema integrato per garantire un livello di sicurezza equiparabile alle moderne basi di dati relazionali. Ad esempio oracle, attraverso il meccanismo degli users e dei privilegi, è in grado di limitare la visione che un utente ha sul contenuto della base di dati e le risorse a cui può accedere. Al contrario Elasticsearch non fornisce un servizio di questo tipo: tutti quelli che hanno accesso alla base di dati possono accedere a qualsiasi dato. Esiste tuttavia un plugin, chiamato Sheld, che fornisce ad Elasticsearch tutti gli strumenti necessari alla sicurezza.

Dato che Elasticsearch utilizza HTTP per la comunicazione, essenzialmente è possibile eseguire tutte le operazioni del protocollo. In questo capitolo esamineremo le operazioni caso per caso, ponendo particolare attenzione alle operazioni più critiche e al modo in cui queste vengono eseguite dal cluster. L'operazione di inserimento (PUT/POST) permette di inserire un documento all'interno della base di dati. Abbiamo detto che un documento è identificato dalla tripla Index-Type-Id, per cui quando si inserisce un documento è necessario specificare queste tre caratteristiche:

PUT /index/type/id {*Body del documento*}

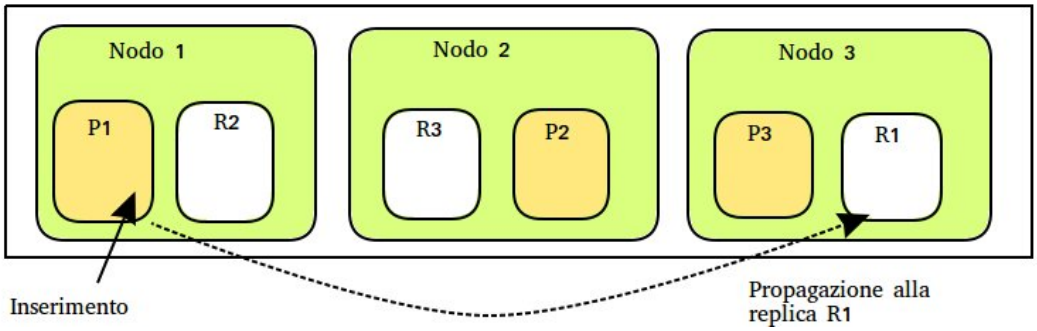
Il documento verrà memorizzato all'interno di una singola shard primaria. Per determinare lalocazione esatta di un documento Elasticsearch utilizza la seguente formula:

$$shard = hash(routing) \ \% \ number\ of\ primary\ shards$$

Dove il routing generalmente si riferisce all'identificatore del documento, mentre la funzione hash viene assegnata dal sistema. Il risultato di questa operazione produrrà sempre un numero compreso tra zero e il numero di shard primarie, corrispondente all'effettiva locazione del documento. Questo è il motivo principale per cui il numero di shard primarie di un indice deve essere determinato a priori; cambiando il numero di shard primarie si rende-

rebbero invalidi tutti i valori di routing dei documenti, rendendoli di fatto non più ricercabili. In alcuni casi può succedere che il documento non abbia un identificatore. In genere nel mondo relazionale per porre rimedio a questo problema si crea una chiave detta surrogata in modo da fornire un identificatore alla tupla. Elasticsearch similmente risolve il problema generando automaticamente una chiave e assicurandosi che questa sia unica all'interno di una coppia indice- tipo. Come tutte le operazioni di scrittura, l'operazione di inserimento deve essere prima completata sulla shard primaria per poi essere propagata alle replica corrispondenti.

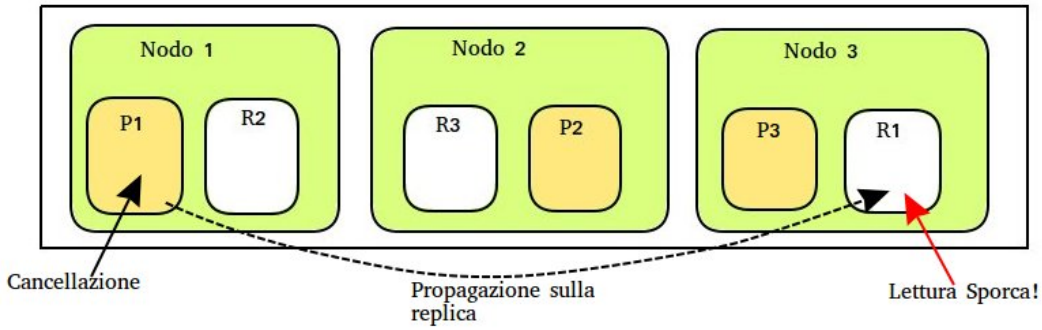
Figura 2.6: Propagazione delle modifiche alle replica



Tutte le operazioni di indicizzazione devono essere eseguite dal nodo master; questo invierà la richiesta al nodo corretto, il quale effettuerà la modifica sulla shard primaria, per poi propagare la richiesta alle replica. L'operazione di ricerca (GET) permette di recuperare uno o più documenti dal database. Similmente all'operazione di PUT possiamo specificare una o più triple index-type-id per recuperare un qualsiasi numero di documenti. Inoltre, attraverso un linguaggio specifico è possibile eseguire query e aggregazioni con una potenza espressiva maggiore dell'SQL. Analizzeremo nello specifico il linguaggio di interrogazione di Elasticsearch, chiamato Query DSL, nel prossimo capitolo. L'operazione di cancellazione(DELETE) permette di cancellare uno o più documenti dal database. Quando si effettaua una richiesta di cancellazione il documento in realtà non viene immediatamente rimosso, ma marchiato come da eliminare. In questo caso un garbage collector si occuperà di effettuare

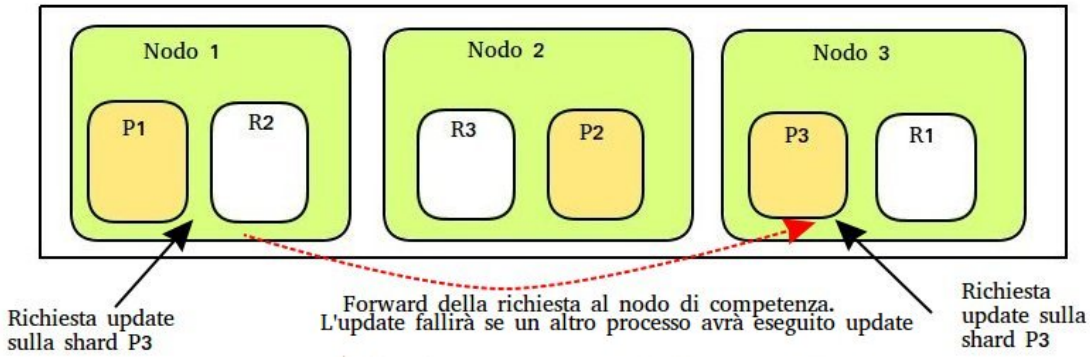
tutte le operazioni di cancellazione. Data la natura distribuita di Elasticsearch, l'operazione potrebbe essere propagata in modo asincrono tra i nodi, provocando in alcuni casi un problema di consistenza. Immaginiamo di effettuare, da due nodi diversi, un'operazione di cancellazione e un'operazione di lettura sullo stesso documento:

Figura 2.7: Propagazione e dirty read



Un'operazione di cancellazione deve essere prima completata sulla shard primaria per poi essere propagata alle replica. Se l'operazione di richiesta viene effettuata sulla shard secondaria mentre il documento sulla shard primaria è già stato cancellato effettueremo una lettura sporca. Elasticsearch di fatto garantisce soltanto la consistenza tra una o più operazioni di scrittura attraverso il sistema del versioning dei documenti. Al contrario la consistenza tra un'operazione di scrittura e una o più operazioni di lettura, a causa della natura distribuita del sistema, non viene garantita. L'operazione di update permette di modificare il contenuto di un documento, ma aggiunge una nuova complessità. In Elasticsearch i documenti sono immutabili, perciò ogni modifica corrisponde sempre ad una operazione di cancellazione, inserimento e reindicizzazione. Durante questo processo, analogamente all'operazione di cancellazione, è possibile incorrere in problemi di consistenza. Esaminiamo un caso reale di update:

Figura 2.8: Propagazione e dirty read



In questo caso il client effettua una richiesta di update al nodo 1. Il nodo propaga la richiesta al nodo 3, il quale contiene la shard primaria relativa al documento. In seguito il nodo 3 effettua la modifica del documento, per poi propagare l'operazione alle shard secondarie. In alcuni casi può accadere che tra il momento in cui viene effettuata la richiesta e il momento in cui il nodo 3 effettua l'operazione di modifica, il documento sia già stato modificato da un altro processo. In questo caso l'operazione fallirà, demandando all'applicazione le scelte di reazione all'evento.

Fino ad ora abbiamo guardato le operazioni soffermandoci in particolare sulle loro caratteristiche e criticità dal lato della consistenza. Un altro aspetto estremamente importante riguarda l'efficienza di esecuzione di gruppi di operazioni. Se da un lato, con le basi di dati relazionali classiche, non emerge quasi mai un problema del genere poichè spesso la base dati è locata su un solo nodo, dall'altro l'architettura distribuita di Elasticsearch fa si che una grande quantità di richieste possa generare un traffico di rete tale da porre un nuovo problema. Infatti ogni singola operazione deve essere inviata ad un nodo per poi essere propagata. Elasticsearch mette a disposizione uno strumento per effettuare grandi gruppi di richieste in sequenza: le operazioni Bulk.

Dato che tutti i nodi conoscono la posizione esatta di tutti i documenti locati sul database, ogni richiesta bulk inviata ad un nodo verrà partizionata per shard, generando nuovi gruppi di richieste che verrando inoltrate alle shard di

competenza. Ogni shard eseguirà il suo gruppo di operazioni in sequenza per poi inviare il risultato al nodo richiente. Di fatto questa macro operazione, come dovremmo aspettarci, non è atomica e dunque non può essere utilizzata per costruire trasazioni. Ogni richiesta viene processata separatamente, per cui il successo o il fallimento di una singola operazione non bloccherà le altre. I moderni database relazionali mettono a disposizione dei metodi per creare transazioni che rispettano quattro proprietà fondamentali, dette ACID (Atomicity, Consistency, Isolation, Durability). Se una o più operazioni all'interno della transazione fallisce, l'intera operazione fallirà causando un abort e spesso un rollback. Elasticsearch al contrario fornisce queste quattro proprietà soltanto sulle singole operazioni di scrittura del documento.

2.4 Interrogare il database

Tutte le moderne basi di dati relazionali mettono a disposizione un linguaggio standard per interrogare il database, oppure forniscono un linguaggio proprietario che integra alcune componenti per aumentare l'espressività. Al contrario Elasticsearch fornisce un linguaggio proprietario molto diverso dall'SQL: il Query DSL. Questo linguaggio, molto flessibile ed espressivo, è usato dal sistema per esporre la maggior parte delle capacità di Lucene attraverso la semplice interfaccia JSON. La potenza del query DSL risiede nella possibilità di effettuare ricerche di tipo full-text, ovvero il risultato di una ricerca non conterrà soltanto i documenti che soddisfano determinati criteri, ma anche quelli che si avvicinano di una certa percentuale al risultato.

Contrariamente all'SQL una interrogazione in Elasticsearch può essere di due tipi: query e filtro. Un filtro viene utilizzato per controllare se un documento soddisfa i criteri di ricerca oppure no, quindi può essere utilizzato solamente per campi che contengono valori esatti per cui è immediato stabilire se il documento appartiene oppure no all'insieme del risultato. Al contrario una query viene utilizzata per effettuare una ricerca su campi di tipo full-text, ovvero che non contengono valori esatti. Una query non re-

stituisce un risultato booleano (se il documento corrisponde oppure no) ma fornirà un'indicazione quantitativa sulla rilevanza del documento. Una volta selezionato il tipo di interrogazione, è possibile comporre la richiesta utilizzando un nuovo componente: le clausole. Le clausole sono dei building blocks che possono essere combinati per ottenere interrogazioni di complessità arbitraria. Esistono due tipi di clausole : le clausole query, che possono essere utilizzate soltanto in un contesto di tipo query, e le clausole filter, che analogamente possono essere usate solo in un contesto di tipo filter. A loro volta essi si dividono in due sottotipi: Leaf e Compound. Le clausole leaf vengono utilizzate esplicitamente per comparare uno o più campi con un determinato valore.

query : { *match:* { *tweet: elasticsearch* } }

Le clausole compound vengono utilizzate per combinare clausole leaf e altre clausole compound. Anche in questo caso esistono vari tipi di clausole, ognuna con il suo compito specifico. Nell'esempio possiamo notare la clausola bool, che compone le tre leaf clauses must, must not e should.

bool : {
    must : { *match :* { *tweet : elasticsearch* }},
    must not : { *match :* { *name : mary* }},
    should : { *match :* { *tweet : full text* }}
}

Talvolta anche le query e i filtri possono svolgere il ruolo di clausole compound. Tuttavia non è possibile inserire le clausole query e filter a qualsiasi livello, ed è necessario introdurre un nuovo concetto: il context. Le clausole query possono essere usate in un query context, mentre le clausole filter possono essere usate in un filter context. Abbiamo detto che il tipo di interrogazione deve essere necessariamente o di tipo query o di tipo filter, dunque

Elasticsearch inferisce il contesto distinguendo tra questi due tipi. Le clausole compound di tipo query fungono da wrapper per altre clausole di tipo query, e analogamente le clausole compound di tipo filter fungono da wrapper per altre clausole di tipo filter. Spesso però vorremmo applicare un filtro ad una query, o viceversa utilizzare una ricerca full-text come un filtro. In questo caso, utilizzando un descrittore, è possibile indicare, ad esempio, che la nostra query sarà di tipo filtered, e dunque conterrà anche clausole filter.

query: {
    filtered: {
        query: { *match:* { *email: business opportunity* }},
        filter: { *term:* { *folder: inbox* } }
    }
}

L'output di molti filtri, una semplice lista di documenti che corrispondono ad alcuni termini di ricerca, è veloce da calcolare e semplice da mantenere in cache. Questi filtri sono semplici da riutilizzare per richieste seguenti. Al contrario le query non solo devono trovare i documenti corrispondenti, ma anche calcolare la rilevanza per ogni documento, che spesso rende le query più complesse da calcolare rispetto ai filter; inoltre le query non possono essere mantenute in cache. Grazie all'inverted index le performance di una query possono essere davvero elevate, ma mai quanto un filtro mantenuto in cache. Il compito dei filter è quello di *ridurre il numero di documenti che devono essere esaminati dalla query.* In generale è buona norma utilizzare le query soltanto per campi full-text o interrogazioni in cui è interessante la rilevanza, e usare i filtri per tutto il resto.

Le aggregazioni sono lo strumento fondamentale con cui è possibile avere una visione complessiva e riassuntiva dei dati presenti all'interno del Database senza ricercare interi documenti; all'interno di strutture decisionali è molto importante possedere strumenti potenti per eseguire questo tipo di

interrogazioni. Le aggregazioni permettono di rispondere a domande molto sofisticate che riguardano i nostri dati. Inoltre, nonostante la funzionalità sia completamente diversa dalla ricerca, utilizza lo stesso tipo di linguaggio e le stesse strutture dati. Infine le aggregazioni possono operare insieme alle richieste di ricerca; ciò significa che è possibile sia ricercare/filtrare documenti sia effettuare analisi sugli stessi dati in una sola richiesta.
Allo stesso modo del query DSL, le aggregazioni hanno una sintassi componibile: funzionalità indipendenti possono essere combinate per generare il comportamento desiderato. Ciò significa che esistono pochi concetti base da imparare, ma un numero di combinazioni illimitate di risultati possibili. Per comprendere le aggregazioni è necessario introdurre due componenti: bucket e metriche. Un bucket è semplicemente una collezione di documenti che soddisfano un determinatocriterio; durante l'esecizione di un'aggregazione, i valori all'interno di ogni documento vengono valutati per determinare se corrispondono ai criteri di un bucket; se c'è corrispondenza, il documento viene inserito nel bucket e l'aggregazione continua. I bucket possono anche essere innestati all'interno di altri bucket, permettendo di creare gerarchie di aggregazione. Elasticsearch fornisce una grande varietà di bucket, che permettono di partizionare i documenti in molti modi (per ora, per popolarità, per range di età, per luogo geografico...). Fondamentalmente però operano con lo stesso principio: partizionare i documenti basandosi su un criterio.
Una metrica è un'operazione matematica (min, max,sum...) calcolata sui valori dei documenti che ricadono all'interno di uno o più bucket.

aggs: {
    colors: {
        terms: { *field: color* },
        aggs: {
            avg price: {
                avg: { *field: price* } } } } }

Nell'esempio proposto vediamo come i documenti (corrispondenti a delle automobili) vengono partizionati in base al colore (bucket), e su ogni colore viene calcolata la media del prezzo (metrica).
In definitiva un'aggregazione è una combinazione di bucket e metriche. Essa può avere un singolo bucket, o una singola metrica, o entrambi. Può persino avere più bucket organizzati gerarchicamente (innestati). Attraverso l'usi di bucket innestati è possibile comporre aggregazioni di grande complessità ed espressività, che spesso superano la potenza del GROUP BY relazionale.

2.5 Pattern di modellazione

Nel mondo reale, le relazioni hanno grande importanza. Le basi di dati relazionali sono progettate appositamente per gestire le relazioni:

- Ogni entity (o riga di una tabella) viene identificata univocamente dalla sua chiave primaria.
- Le entity sono normalizzate, ovvero i suoi dati sono memorizzati soltanto una volta, e entity correlate mantengono solo la chiave primaria come riferimento. Il cambiamento di una entity deve accadere soltanto una volta.
- E' possibile effettuare join, per eseguire ricerche cross-entity.
- I cambiamenti sulle entity godono delle proprietà ACID.
- Supporto di transazioni ACID.

Ma i database relazionali hanno i loro limiti, oltre allo scarso supporto per le ricerche full-text. Effettuare join tra entity è molto costoso; più join eseguiamo, più inefficiente sarà la nostra query. Inoltre eseguire Join tra entity locate su nodi diversi è talmente costoso che non è possibile nella pratica. Questo limita fortemente la quantità di dati che possono essere memorizzati all'interno di un singolo server. Elasticsearch, come molti database NoSQL,

tratta il mondo come se fosse piatto. Un indice è semplicemente una piatta collezione di documenti indipendenti. Un singolo documento dovrebbe contenere tutte le informazioni richieste per decidere se corrisponde ai criteri di una ricerca. Abbiamo detto che in Elasticsearch tutti i cambiamenti su un documento sono ACID, ma le transazioni che coinvolgono più documenti non lo sono. Non esiste un modo per eseguire il rollback di un indice se parte di una transazione fallisce. Il mondo piatto ha ovviamente i suoi vantaggi:

- Indicizzare è veloce e lock-free.
- Ricercare è veloce e lock-free.
- Grandi quantità di dati possono essere distribuite su molti nodi, poichè ogni documento è indipendente dagli altri.

Ma le relazioni hanno grande importanza. E' necessario utilizzare dei pattern di modellazione per riempire il gap tra il mondo reale e il mondo piatto. In questo capitolo analizzeremo le quattro tecnche utilizzate per gestire i dati relazionali in Elasticsearch: Application side joins, Data denormalization, Nested objects e Parent-child relationship. Molto spesso le soluzioni pratiche utlizzano più di una tecnica per raggiungere il corretto trade-off tra performance e ridondanza.

Application-side Joins

E' possibile emulare parzialmente un database relazionale implementando i join nella nostra applicazione. Immaginiamo di indicizzare degli utenti insieme ai loro blog posts. Nel mondo relazionale modelleremo il dominio applicativo in questo modo:

User
name : Jhon,
email : jhon@gmail.com

Blogpost
title : Relationships,
body : It's complicated...
user : 1

Memorizzando l'identificatore del documento user all'interno di Blogpost abbiamo creato una correlazione. Se volessimo ricercare tutti i blogpost redatti dagli utenti chiamati jhon, dovremmo lanciare due query: la prima ricerca tutti gli identificatori degli utenti di nome jhon, la seconda sfrutta questi identificatori per restituire la lista di blogposts. Il vantaggio principale degli application side joins risiede nel fatto che tutti i dati sono normalizzati. I cambiamenti che riguardano un utente avvengono solamente in un documento. Lo svantaggio principale è ovvio: sarà necessario eseguire più query a tempo di esecuzione per effettuare join tra i documenti. Nel mondo reale la prima query potrebbe ritornare milioni di record, incrementando molto lo spazio di ricerca della seconda, la quale potrebbe ritornare miliardi di record. Questo tipo di approccio nella pratica è applicabile solamente quando la cardinalità del risultato della prima query è basso (esattamente come accade per il Nested Loops), altrimenti la complessità computazionale diventerebbe tale da non permettere l'esecuzione della ricerca.

Data denormalization

La data denormalization, specialmente in ambito NoSQL, è una delle tecniche più utilizzate ed apprezzate. Questo approccio consiste nel denormalizzare parte di un' entità normalmente separata all'interno di un'altra, in modo da avere copie ridondanti dello stesso dato, evitando costosi joins. Riprendiamo l'esempio degli utenti e dei blogpost:

User

name : jhon,
email : jhon@gmail.com

Blogpost
title : Relationships,
body : It's complicated...
user : { *id : 1, name : John Smith* }

In questo caso il nome dell'utente è stato denormalizzato all'interno del blogpost. Sarà così possibile ricercare e aggregare i Blogpost per nome dell' user senza effettuare un costoso application-side join.
In generale il (grande) vantaggio della Data denormalization risiede nella velocità, poichè ogni documento contiene tutte le informazioni necessarie per determinare se corrisponde oppure no ai termini di ricerca. Esistono però anche degli svantaggi: il primo consiste nell'uso di spazio: replicare dati, specialmente per entità di grande cardinalità, comporta un significativo aumento dello spazio su disco occupato. Spesso questo non è un problema poichè i dati scritti su disco sono ampiamente compressi e lo spazio ha un costo relativamente basso. Il vero problema risiede nella gestione degli update: cambiare il contenuto di una entità denormalizzata significa effettuare un' operazione di update su potenzialmente una grandissima quantità di documenti. In uno scenario in cui i cambiamenti sulle entità sono molto frequenti l'approccio di denormalizzazione si dimostra molto inefficiente sia in termini di prestazioni sia in termini di sicurezza per modifiche concorrenti sugli stessi dati. Viceversa, in situazioni in cui i cambiamenti accadono raramente, la data denormalization permette di ottenere prestazioni davvero interessanti, come vedremo nel caso di studio analizzato nei prossimi capitoli.

Nested Objects

Dato che in Elasticsearch la creazione, cancellazione e modifica di un do-

cumento sono operazioni atomiche, può essere utile memorizzare entità correlate all'interno dello stesso documento. Vediamo l'esempio dei blogpost:

Blogpost
title : Nest eggs,
body : Making your money work...
comments : [
{ *name : John Smith,*
comment : Great article,
date : 2014-09-01
}, { *name : Alice*
...
]

In questo caso tutte le entità comment vengono innestate nel corrispettivo blogpost di appartenenza. Abbiamo già analizzato questo tipo di approccio quando abbiamo introdotto i Multilevel Objects durante la trattazione dei tipi di dato accettati. Nel caso dei Multilevel Objects la correlazione tra i comments viene persa: non è più possibile stabilire, in questo esempio, la relazione tra Jhon Smith e il commento Great article. I Nested Objects risolvono questo problema indicizzando le entità come documenti nascosti separati:

User1
comments.name : [*john, smith*],
comments.comment : [*article, great*],
comments.date : [*2014-09-01*]
User2
comments.name : [alice],
...
Indicizzando ogni nested object separatamente, i campi all'interno dell'ogget-

to manterranno le loro relazioni. Inoltre, tenendo in considerazione il modo in cui questi oggetti sono indicizzati, effettuare join tra il nested document e il documento padre durante l'esecuzione di una query è estremamente veloce. Questi documenti sono nascosti, e dunque non accedibili direttamente. Per effettuare un'operazione di update, aggiunta o cancellazione è necessario reindicizzare tutto il documento; analogamente i risultati ottenuti da una richiesta di ricerca non conterranno solo i nested objects, ma l'intero documento. I Nested Objects sono utili quando esiste una sola entità principale, come i blogpost, con un numero limitato di entità correlate e meno importanti, come i commenti. Spesso è importante poter ricercare un'entità basandosi sul contenuto delle sue entità innestate. Anche questo pattern di modellazione presenta alcuni svantaggi: per aggiungere, modificare o cancellare un nested document è necessario reindicizzare tutto il documento, rendendo l'operazione costosa se sono presenti molti nested documents; inoltre tutte le ricerche restituiscono l'intero documento, non solo un eventuale nested document. In alcuni casi può essere necessario una completa separazione tra il documento principale e le sue entità associate; é possibile ottenere questo tipo di separazione utilizzando la *parent-child relationship*.

Parent-child relationship

La parent-child relationship è per sua natura simile al pattern dei nested objects: entrambi i metodi permettono di associare una o più entità ad un'altra. La differenza è che, nel caso dei nested objects, tutte le entità correlate vivono sullo stesso documento mentre, nel caso della parent-child relationship, i documenti padre e quelli figlio sono completamente separati. La funzionalità parent-child permette di associare un tipo di documento con un altro, in una relazione one-to-many, ovvero un padre ha molti figli. I vantaggi della parent-child modeling rispetto al pattern dei nested objecs sono i seguenti:

- E' possibile effettuare un'operazione di update sui padri senza reindicizzare i figli.

- I documenti figlio possono essere aggiunti, modificati e cancellati senza che nè i padri nè gli altri figli ne siano affetti.
- I documenti figlio possono essere restituiti da una query di ricerca.

Elasticsearch mantiene un mapping di tutte le associazioni padre-figlio, ed è per questo motivo che le ricerche che effettuano joins sulla gerarchia hanno un grado di velocità apprezzabile. Esiste però una limitazione: *i documenti padre e tutti i loro figli devono vivere obbligatoriamente sulla stessa shard.* Quando abbiamo spiegato il modo in cui Elasticsearch assegna i documenti alle shard, abbiamo introdotto il concetto di routing; generalmente il valore di routing corrisponde all'identificatore del documento da indicizzare. Per questo motivo, quando si indicizza un documento figlio, è necessario fornire anche l'identificatore del padre per fare si che vivano sulla stessa shard.
In conclusione la Parent-child relationship è una tecnica di modellazione davvero interessante quando è necessario ottenere un livello alto di performance a fronte di costanti modifiche sui dati del database. In generale però le richieste parent-child possono essere da 5 a 10 volte più lente dell'equivalente nested, compromettendo i tempi di risposta del sistema; toccherà al progettista scegliere la modellazione adatta per ottenere il corretto trade-off tra performance di richiesta e performance di indicizzazione.

Capitolo 3

Caso di studio

Il progetto che si è sviluppato, nel presente elaborato, è stato svolto all'interno del contesto di un sistema di Social Business Intelligence(SBI). Un' enorme quantità di *user-generated-content*(UGC) legata ai gusti delle persone, ai loro pensieri e azioni è oggi disponibile grazie alll'incredibile diffusione dei social network e dei dispositivi portatili. Questa grande quantità di informazioni sta catturando l'attenzione di molte strutture decisionali(sia in ambito aziendale e non) poichè permette di avere una percezione istantanea della situazione di mercato e può in qualche modo aiutare a comprendere i fenomeni di buisness e della società. Nel presente capitolo introdurremo e spiegheremo il concetto di SBI, per poi spostare la nostra attenzione sul caso di studio, descrivendo in maniera dettagliata l'architettura del progetto e i punti chiave della metodologia con cui è stato sviluppato; infine analizzaremo l'elaborato prodotto, spiegando quali sono i punti chiave dell'architettura su cui siamo intervenuti, quale idea ha guidato la scelta della tecnologia e come è stato modellato il dominio applicativo per soddisfare i reqirements di progetto.

3.1 Introduzione alla Social Business Intelligence

La *Social Business Intelligence* è una disciplina che mira a combinare dati aziendali con dati UGC per permettere a strutture decisionali di analizzare e reingegnerizzare il loro buisness basandosi sul trend di mercato dell'ambiente circostante. Parallelamente alla classica buisness intelligence, l'obiettivo della SBI è di fornire strumenti potenti e flessibili alle strutture decisionali (che chiameremo *utenti*) riducendo al minimo i costi in termini di database e ICT. Nel contesto della SBI, la categoria più diffusa di UGC proviene da sorgenti di testo chiamate *clip*. Le clip possono essere messaggi apparsi su social media, oppure articoli diffusi sui giornali o magazine on-line. Estrapolare dati utili agli utenti da testi UGC richiede in primo luogo un metodo per eseguire *crawling* e recupeare tutte le clip appartenenti ad una certa *area di interesse*, per poi arricchirle e far emergere tutte le informazioni interessanti dal semplice testo. L'area di interesse definisce i confini e l'estensione del progetto, e per esempio può essere un particolare brand oppure un'intera area di mercato. L'attività di arricchimento consiste nell'identificare le parti strutturate della clip, come l'autore, e spesso include tecniche di *sentiment analisys* per interpretare ogni frase, recuperare tutti i concetti menzionati ed eventualmente assegnare un *sentiment* (chiamato anche *polarità*, per esempio positivo, negativo o neutrale). Di fatto, un elemento fondamentale dell'analisi degli UGC è l'individuazione dei topic, ossia quei termini che individuano i concetti di interesse per l'utente. A seconda del dominio, un topic può essere il nome di un prodotto, di una persona, oppure un termine di uso comune che ha un significato particolare per l'utente. Attraverso la definizione dei topi e di una gerarchia di aggregazione degli stessi (ad esempio, prodotti che si aggregano in categoria) si aggiunge un'importante dimensione di analisi degli UGC focalizzata sul dominio dell'utente. Infine, tutte le fasi che vanno dal web crawling all'analisi dei risultati da parte degli utenti faranno parte di un unico processo di Social Business Intelligence.

Analizziamo ora l'architettura generale di un progetto SBI in ogni sua parte, il quale mette a disposizione degli utenti le informazioni necessarie attraverso l'uso cubi multidimensionali e repository testuali:

- **L' Operational Data Store (ODS)** memorizza le informazioni rilevanti delle clip, degli autori e dei canali sorgente; l'ODS rappresenta tutti i topic all'interno di un'area di interesse e le loro relazioni.

- **Il Data Mart (DM)** memorizza dati integrati in forma di set di cubi multidimensionali che supportano tutti i vari processi decisionali.

- **Il modulo di Crawling** utilizza un set di query, basate su un certo numero di parole chiave, per reperire le clip (insieme ai loro possibili metadati) che sono nel dominio dell'area di interesse. L'obiettivo del crawler può essere sia l'intero web che un set di sorgenti definite dall'utente (come blog, forum, siti web oppure social network).

- **Il modulo di Semantic Enrichment** effettua le sue operazioni sull'ODS in modo da estrarre la semantica nascosta all'interno dei testi delle clip. In base alla tecnologia adottata queste informazioni possono includere le singole frasi all'interno delle clip, i topic, la sintassi e la relazione semantica tra parole, oppure il sentiment di una intera frase o di un singolo topic che contiene.

- **Il modulo ETL** trasforma periodicamente l'output semi strutturato del crawler in una forma strutturata, che verrà poi caricata sull' ODS, possibilmente combinando i dati con quelli estratti dal CRM. In seguito estrae i dati relativi alle clip e ai topic dall'ODS, per poi integrarli con i dati di buisness estratti dall' Enterprise Data Warehouse, caricandoli all'interno del DM.

- **L' Analisi OLAP** permette all'utente di esplorare l'UGC da prospettive diverse e nell'effettivo di controllare nel complesso il social feeling.

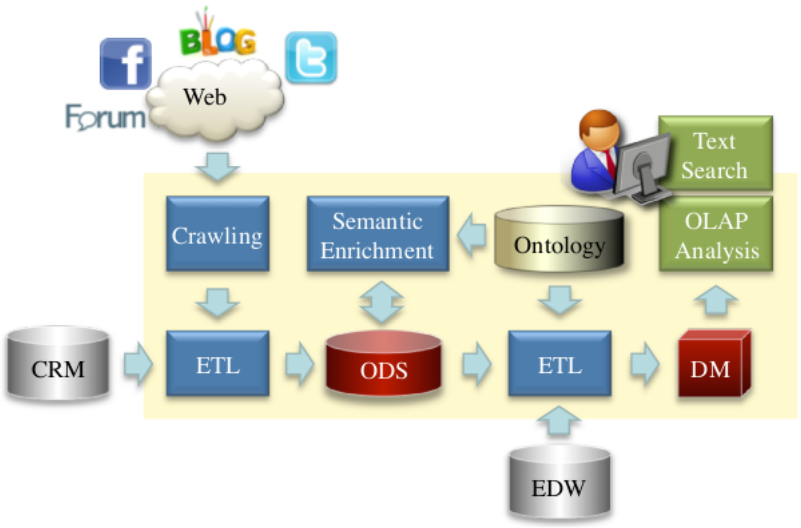
Usando strumenti OLAP per l'analisi di dati UGC in ottica multidimensionale permette di ottenere grande flessibilità da questo tipo di architettura.

Da un punto di vista metodologico, è importante osservare che generalmente in un progetto SBI mancano una vista organica e strutturata del processo di design. Questo accade sia a causa del grande dinamismo degli UGC, sia perchè spesso è necessario analizzare dati in tempo reale, in modo da reagire tempestivamente ai cambiamenti del dominio. La metodologia iterativa utilizzata per la realizzazione di questo progetto (Figura 3.1) è stata concepita sia per far fronte alla necessità di velocizzare il processo di SBI, sia per massimizzare l'efficacia dell'analisi da parte dell'utente ottimizzando e raffinando tutte le sue fasi. Ovviamente in ogni progetto SBI è necessario effettuare una continua manutenzione a causa della continua (e spesso rapida) variabilità dell'ambiente, causata dalla grande volatilità dei dati provenienti da sorgenti web. La variabilità ha un impatto forte in ogni attività, dall'operazione di crawling all'arricchimento semantico dei testi, richiedendo a designer e sviluppatori di amministrare efficacemente i cambiamenti dei requisiti.

3.2 Progetto WebPolEu

Il progetto che analizzeremo in questa sezione, WebPolEu, si pone come obiettivo lo studio delle connessioni tra la politica e i social media in una prospettiva comparativa sia dal punto di vista dei cittadini sia dal punto di vista degli attori della politica. Analizzando testi digitali, partecipazioni politiche on-line e discussioni politiche su social media, la ricerca valuta il livello di inclusione, rappresentazione e qualità delle discussioni a tema politico on-line in Germania, Italia e United Kingdom. In particolare, studiando il modo in cui gli attori politici comunicano on-line e come i cittadini si relazionano ai leader di partito e ai sindaci cittadini sui social media, il progetto ricerca

Figura 3.1: Architettura del progetto



fino a che punto la comunicazione politica on-line incontra la domanda dei cittadini e se i media digitali forniscono effettivamente opportunità di assicurarsi il controllo dei governati rispetto ai governanti. Inoltre effettuando un paragone tra Germania, Italia e UK, la ricerca sarà anche in grado di testare alcune ipotesi in relazione a fattori istituzionali e sistematici in tre sistemi politici rilevanti per la definizione del ruolo globale dell'Europa.
Il progetto ha raccolto i dati da Aprile a Maggio 2014 nell'ambito delle elezioni europee(figura 3.2).

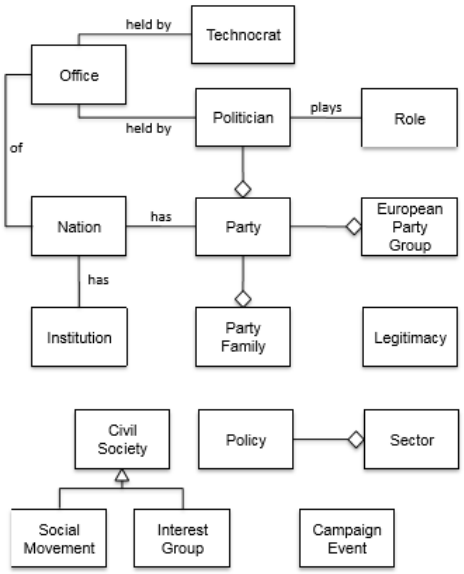
Figura 3.2: Dati raccolti durante le elezioni europee 2014

Clip	Topic	Entity	Occorrenze Topic	Occorrenze Entity	Relazioni semantiche
3.275.193	432	2.902.942	23.398.601	519.446.526	23.837.474

Dal dominio mostrato in figura 3.3 sono stati estrapolati tutti i topic di interesse (ad esempio, i nomi dei politici coinvolti nelle elezioni, gli argomenti di maggior discussione, ecc.) è stata costruita una gerarchia di topic per raffinare l'aggregazione degli UGC raccolti (ad esempio, per poter aggregare

le clip in cui compare un partito considerando anche le clip in cui compaiono i politici di tale partito).

Figura 3.3: Rappresentazione UML dell'ontologia dei topic per il progetto WebPolEu



3.3 Caso di studio: Elasticsearch

Fino ad oggi la parte di analisi dei dati front-end è stata realizzata e implementata, come abbiamo visto, attraverso un cubo relazionale. Di fatto la grande quantità di dati UGC, unito alla loro variabilità, pone un serio problema di Big Data, compromettendo ampiamente le performance di una classica base di dati relazionale. A questo punto si rivela necessario scegliere una tecnologia che unisca la necessità di ottenere performance apprezzabili alla possibilità di costrurire una base dati che garantisca grande scalabilità, affidabilità e disponibilità dei dati; per questo motivo è stata scelta la tecnologia di Elasticsearch per far fronte a questi problemi. Elasticsearch fornisce cinque funzionalità fondamentali:

- **Scalabilità.** Elasticsearch, come la maggior parte dei database NoSQl, permette di scalare orizzontalmente in modo molto semplice ed intuitivo. Di fatto è possibile aggiungere e rimuovere nodi a caldo senza danneggiare in alcun modo l'integrità del sistema. Al contrario una base dati relazionale, per godere del privilegio di scalabilità, necessita un grande livello di ingegnerizzazione, nonchè una forte necessità di investire in analisi e competenze.

- **Affidabilità.** Elasticsearch, attraverso il sistema di sharding e replicazione descritto nei capitoli precedenti, garantisce e gestisce automaticamente un'alta affidabilità del sistema a fronte di un minimo investimento in strutture. In una base dati relazionale, quando si rende necessario effettuare un'operazione di scaling, l'affidabilità tra nodi non è più garantita. Di fatto è necessario ancora una volta reingegnerizzare la logica applicativa del sistema e reinvestire in competenze specifiche.

- **Ricerca full-text.** Al contrario dei classici database relazionali e di molti database NoSQL, Elasticsearch mette a disposizione un potente strumento di ricerca full-text. Data la natura del dominio applicativo può essere interessante eseguire ricerche basate sul contenuto di testo delle clip o di singole frasi. Elasticsearch è al momento il secondo motore di ricerca full-text più usato al mondo, e ricordiamo tra i suoi utilizzatori Wordpress, GitHub e Soundcloud.

- **Aggregazioni.** Elasticsearch mette a disposizione un set di API apposite estremamente potenti per eseguire ricerche (non necessariamente full text), ma soprattutto aggregazioni, raggiungendo livelli di complessità di molto superiori a quelle di un comune database relazionale.

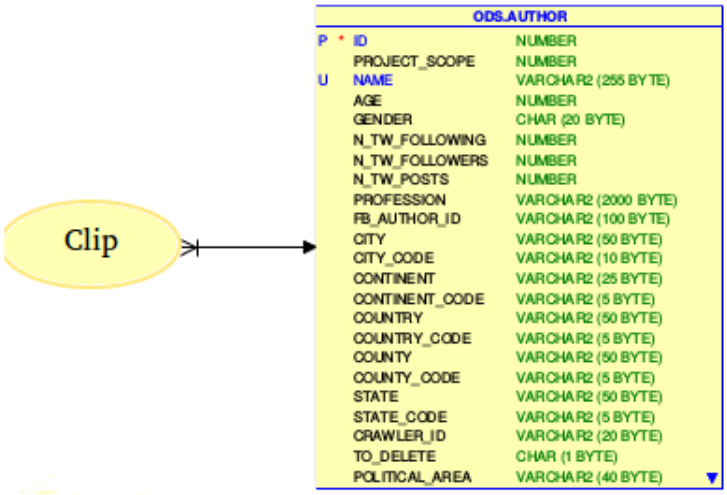
- **Performance.** Elasticsearch, come tutte le basi di dati NoSQL distribuite, garantisce un livello di performance molto alto a fronte della problematica di Big Data descritta pocanzi.

Analisi del corrente sistema databasistico

Durante la realizzazione del progetto è stato necessario interfacciarsi con due basi di dati: un ODS e un DM. In questa sezione analizzeremo la struttura dell' ODS, dal quale sono state prese la maggior parte delle informazioni, senza soffermarci sul DM, da cui sono stati reperiti solo alcuni dati per ragioni di completezza.

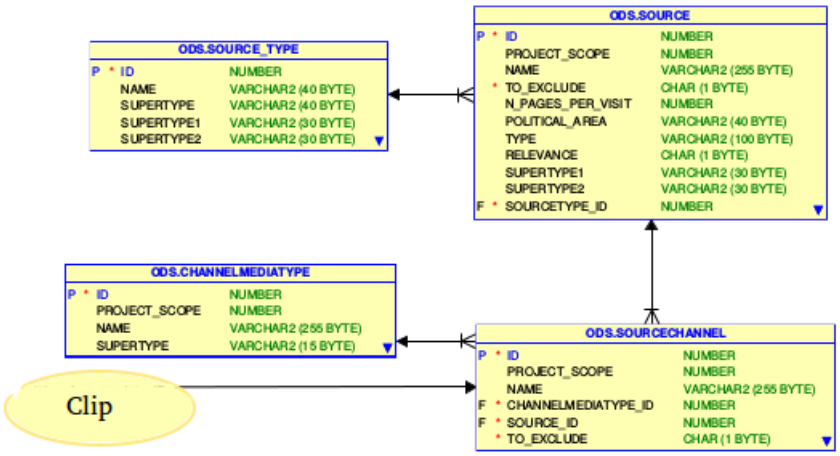
Le clip contengono una serie di metadati quali la geolocalizzazione della clip, la data di pubblicazione, il sentiment estrapolato dal servizio di crawling (positivo, negativo o neutro) e tutta una serie di misure numeriche (ad esempio, il numero di retweet in un tweet). Inoltre, tra i campi di ogni clip, ci sono il Title e il Content, che contengono rispettivamente il titolo della clip e il suo contenuto (quest'ultimo in formato CLOB)

Figura 3.4: Gerarchia clip-autore



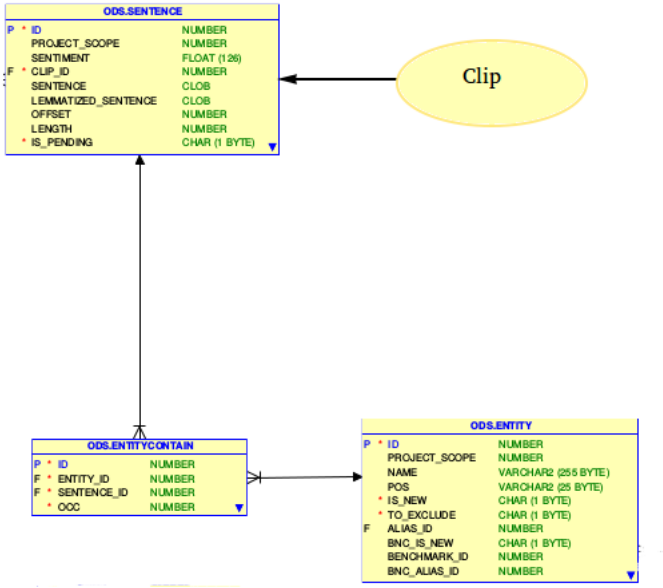
A ogni clip è associato un solo autore (figura 3.4), il quale contiene una serie di metadati che lo caratterizzano.
Ogni clip proviene da un source channel (figura 3.5), un metadato che identifica la sorgente della clip (ossia il sito web) ed il tipo di canale mediatico utilizzato (ad esempio, news, blog o forum); il sourcechannel si rende necessario dal momento in cui una sorgente può disporre di più canali mediatici (ad esempio, la sezione di news e di blog su theguardian.com).

Figura 3.5: Gerarchia clip-source channel



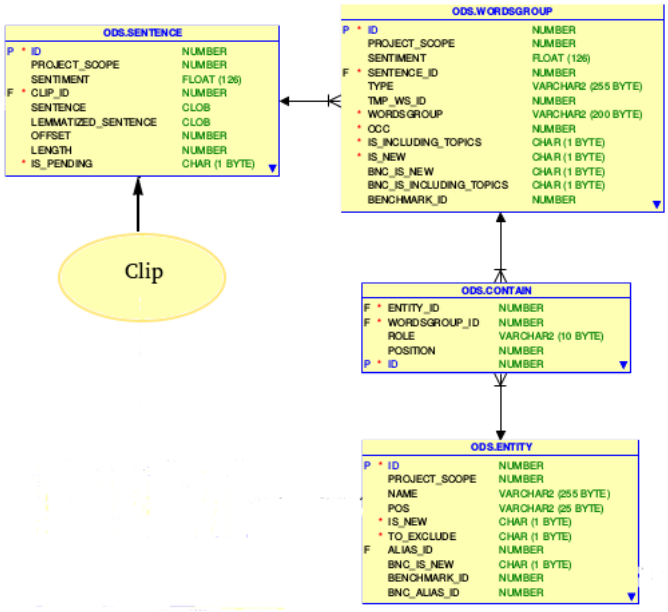
Lo schema in figura 3.6 illustra la gerarchia delle entities contenute all'interno di una clip. Ogni clip viene divisa in un certo numero di sentences, le quali contengono a loro volta un certo numero di entities. Ogni sentence possiede il campo Sentence,che contiene il testo della frase; anche questo campo è di grande importanza, poichè sarà soggetto a ricerche di testo. Inoltre le sentence contengono il sentiment, recuperato invece dal motore semantico, che ha un valore numerico variabile. Questo sentiment viene poi riportato sulle entity e sulle relationship pesandolo rispetto al numero di occorrenze delle stesse.

Figura 3.6: Gerarchia clip-sentence-entity



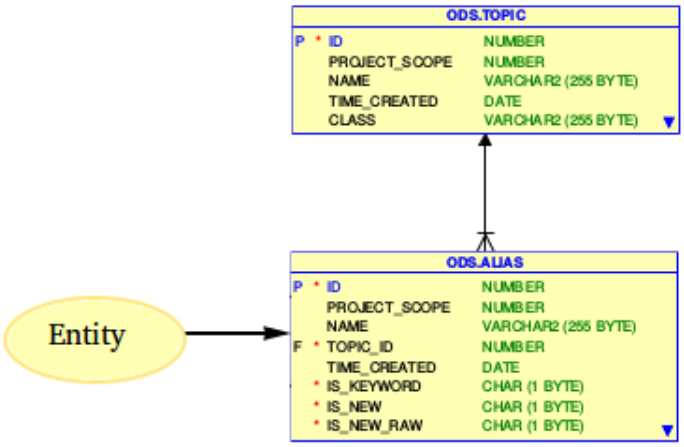
Lo schema in figura 3.7 illustra la gerarchia delle relationships (rappresentate dalla tabella wordsgroup) contenute all'interno di una clip. Le relationships sono associate a tre entity: due entity legate tra loro ed un'eventuale terza entity che funge da qualificatore della relazione (ad esempio, in una frase Renzi a cena con la Merkel possono essere indivituati Renzi e Merkel cone entity legate tra loro da una terza entity, cenare).

Figura 3.7: Gerarchia clip-sentence-relationship



Infine ogni entità del contesto è associata ad un particolare topic (figura 3.8). Il collegamento tra topic ed entity è effettuato attraverso la tabella alias, in cui sono memorizzate tutte le declinazioni ed i sinonimi di ciascun topic; ad esempio, il topic university è associato agli alias university, universities,-academy e academies, i quali sono a loro volta associati all'entity sulla base di una corrispondenza esatta del nome. Siccome gli alias hanno l'unico scopo di facilitare il collegamento tra tipic ed entity, essi non verranno considerati nella modellazione su Elasticsearch.

Figura 3.8: Gerarchia entity-topic



Proposte di modellazione e risultato finale

Per la realizzazione di una corretta traduzione del sistema dal mondo relazionale quello di Elasticsearch è stato necessario esaminare e considerare alcuni fattori:

- **Dinamicità dei dati.** In un contesto in cui la maggior parte dei dati è formata da informazioni testuali, e dunque di scarsa dinamicità, è ragionevole pensare di voler inserire all'interno di una sola struttura dati tutte le informazioni relative alle singole clip, comprendendo i relativi metadati (autore,sorgente e canale mediatico) e la scomposizione in frasi, parole e relazioni. Ciò è dovuto al fatto che, una volta recuperate dai servizi di crawling e di arricchimento semantico, tali informazioni non subiscano più alcuna modifica. La componente dinamica della base dati è costituita fondamentalmente dalle gerarchie che comprendono i topic. Questi di fatto, essendo in continua mutazione durante il ciclo di vita del sistema, necessitano attenzione particolare poichè, come abbiamo spiegato nei capitoli precedenti, per compere un'operazione di modifica su un documento è necessario reindicizzarlo. Sarà necessario tenere in considerazione questo aspetto durante la modellazione, valu-

tando il corretto trade-off tra performance di lettura e performance di scrittura.

- **Orientamento ai documenti di Elasticsearch.** Elasticsearch è una base di dati NoSQL orientata ai documenti. Sarà necessario tenere in considerazione la struttura e valutare la corretta indicizzazione dei dati all'interno dei documenti indicizzati dal sistema.

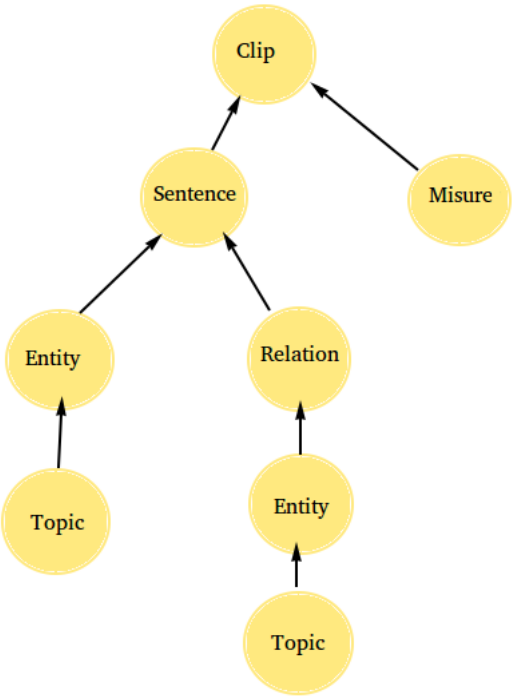
- **Espressività delle interrogazioni.** Al contrario delle basi di dati relazionali, Elasticsearch non possiede un metodo diretto per effettuare join tra diversi documenti; è tuttavia molto più espressivo di un DB relazionale quando le interrogazioni riguardano una sola classe di documenti, attraverso l'uso delle aggregazioni. Questo aspetto, come vedremo, sarà fondamentale durante la decisione della modellazione.

- **Performance.** Sono due gli aspetti da tenere in considerazione al momento della valutazione della modellazione: performance in scrittura e performance in lettura. Eseguendo una completa denormalizzazione e nesting dei dati si ottengono performance estremamente interessanti in termini di ricerca e aggregazione, causando però un'anomalia di aggiornamento. Viceversa eseguendo una scissione in termini di modellazione tra elementi statici (o correlati) ed elementi dinamici si ottiene una grande ottimizzazione in scrittura, cedendo però parte della velocità di ricerca.

Sono dunque state proposte due idee per la modellazione della base dati Elasticsearch:

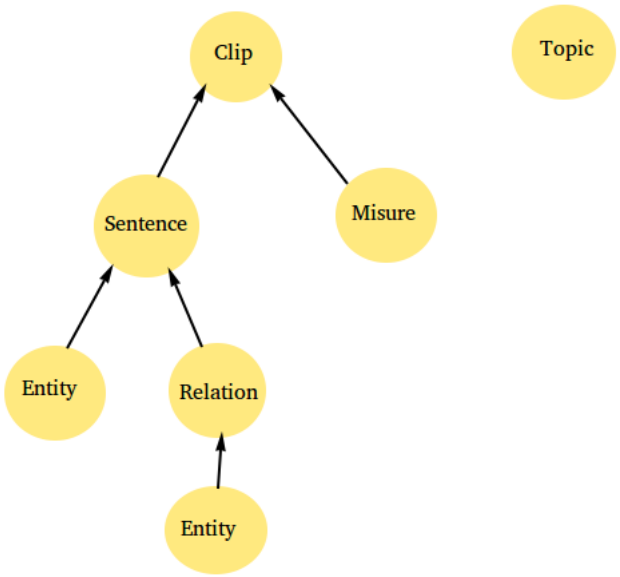
- Completa denormalizzazione. Tutti i dati dei database relazionali vengono condensati all'interno di un solo documento. In questo modo viene privilegiata molto la componente delle performance in termini di ricerca e aggregazione. In questo caso però a fronte dell'inserimento di nuovi topic, sarà necessario reindicizzare tutti i documenti correlati.

Figura 3.9: Denormalizzazione completa



- Scissione delle gerarchie dinamiche. La parte relativa ai topic, per la sua dinamicità, viene separata dal resto della gerarchia, consentendo di evitare anomalie di scrittura al momento dell'inserimento di nuovi topic, peggiorando però l'user-experience al momento dell'esecuzione di interrogazioni.

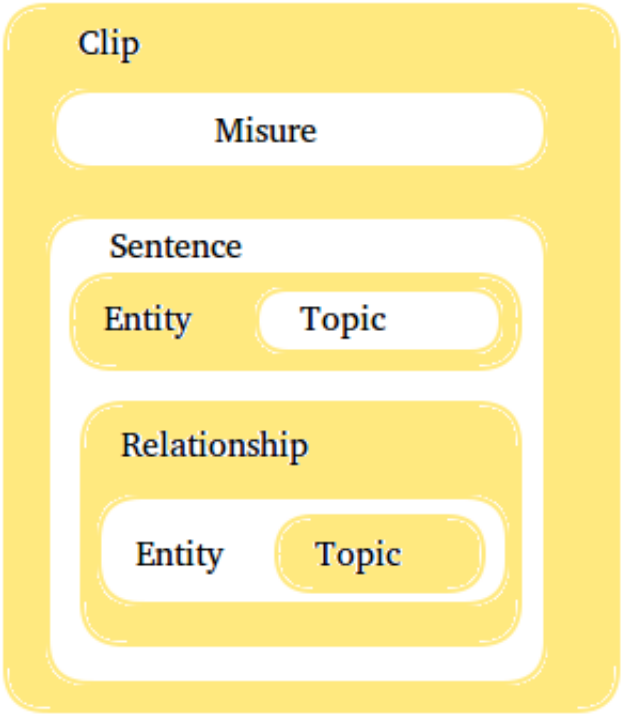
Figura 3.10: Separazione gerarchie dinamiche



In conclusione è stato deciso di procedere operando una modellazione full-nesting per le seguenti ragioni:è' stata data grande importanza alla user-experience al momendo della definizione dei requisiti; dunque una modellazione di tipo full-nesting aumenta drasticamente le performance delle interrogazioni.La complessità delle interrogazioni in termini di ingegnerizzazione diminuisce operando una modellazione full-nesting; in una modellazione di scissione infatti, per ricostruire le gerarchie entity-topic sarebbe stato necessario eseguire un application-side join, che genera un ulteriore livello di complessità al momento dell'implementazione del sistema.E' stato deciso che, a fronte di un cospicuo aumento dell performance, non è rilevante l'anomalia di inserimento causata dalla replicazione dei topic al momento dell'aggiornamento.

In figura 3.11 possiamo notare il risultato finale della modellazione di tipo full-nesting.

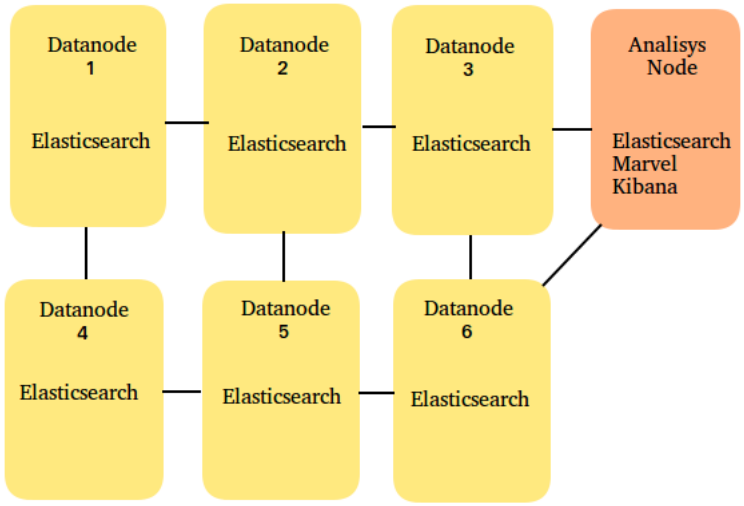
Figura 3.11: Documento finale Elasticsearch



3.4 Installazione e porting dei dati

Dal punto di vista sia dell'installazione che della manutenzione, come abbiamo visto, Elasticsearch non richiede alcun tipo di competenza specifica in merito. Il sistema è stato installato su un cluster di sette macchine, corrispondenti ai sette nodi di Elasticsearch. Sei nodi sono utilizzati per esegiure ricerche, mentre l'ultimo contiene tutte le strutture dati e i plugin necessari (Marvel e Kibana) per eseguire analisi e manutenzione attraverso un'interfaccia utente.

Figura 3.12: Struttura del cluster



Ogni nodo del cluster presenta queste specifiche:

CPU: Intel i7-4790,4 core, 8 threads, 3.6 Ghz
Ram: 32 GB
HDD: 2 x 2 TB HDD, 7200 RpM
Sistema operativo: CentOS 6.6 (Linux)

Mentre il Nodo su cui è installato Oracle presenta queste specifiche:

CPU: AMD Opteron 6128, 8 core, 2 GHz
Ram: 64 GB
HDD: 1.2 TB, 15000 RpM
Sistema operativo: Windows Server 2008 R2 Standard SP1

Durante la fase di progettazione, è stato deciso di memorizzare tutti i documenti all'interno di un singolo indice. L'indice contiene cinque shard primare e una replica per ogni primaria, per un totale di dieci shard distribuite sui sei datanode. Per l'operazione di popolamento è stato necessario produrre uno script java apposito, dato che Elasticsearch, al contrario di alcuni databa-

se NoSQL, non fornisce alcun supporto alla migrazione dei dati provenienti da database relazionali; ciò è perfettamente comprensibile, considerando che la logica di implementazione della base dati è completamente diversa da un classico database relazionale, sia in termini di struttura concettuale (tabelle vs. documenti) sia in termini di struttura fisica (nodo singolo vs. Cluster). Per costruire il documento finale a partire dai dati in forma strutturata è stato necessario effettuare una serie di interrogazione SQL per riuscire a recuperare i dati di ogni clip e metterli insieme in un unico oggetto. Tale processo è stato svolto in maniera bottom-up ed è stato ottimizzato per effettuare un bulk loading di clip a gruppi di mille.
Descriviamo ora nel dettaglio la procedura utilizzata: in pimo luogo, vista la bassissima cardinalità dell'insieme, sono stati recuperati tutti i sector raggruppati per identificatore del topic di appartenenza. In seguito sono state recuperate tutte le relationships (e relativa gerarchia), raggruppate per l'identificatore della sentence di appartenenza. Analogamente alle relationships sono state recuperate tutte le entities (e relativa gerarchia), raggruppate per l'identificatore della sentence di appartenenza. Dunque sono state recuperate tutte le sentences, raggruppate per l'identificatore della clip di appartenenza e, attraverso le precedenti strutture calcolate, è stato possibile riempire ogni sentece con le entities e le relationships corrispondenti. Infine sono state recuperate tutte le clip, conseguentemente riempite con sentences calcolate nel passaggio precedente, per poi essere caricate sul database Elasticsearch a blocchi di mille.

3.5 Test delle performance

L' analisi delle performance sono state effettuate con Oracle, in quanto strumento originariamente utilizzato nel progetto, con Elasticsearch, e con Impala, un database analitico per Apache Hadoop; Impala fornisce un layer SQL-like sul popolare file system di Hadoop. Le performance sono state messe a confronto utilizzando sei classi di interrogazioni e aggregazioni. Per

quanto riguarda Elasticsearch, le query sono state eseguite sia utilizzando un cluster a sette nodi, sia un cluster al minimo delle proprie capacità, ovvero utilizzando due nodi. Per quanto riguarda Impala, le performance sono state valutate sullo stesso cluster di Elasticsearch. Notiamo che in questo caso, una colonna della tabella è stata dedicata alle query filtrando tutti i topic Unclassified; di fatto, non tutte le entità del contesto potrebbero essere collegate ad un topic specifico, e dunque sono state assegnate ad un topic di tipo Unclassified. Questo problema non si pone su Elasticsearch, poichè i topic di questo tipo non sono stati assegnati alle entity, ma sono stati direttamente scartati.

Figura 3.13: Tempi di esecuzione dei tre sistemi

	Impala	Impala con filtro Unclassified	Elasticsearch	Elasticsearch Dual node	Oracle
SELETT1	33.7	9.4	16.2	26.1	39.0
SELETT2	20.6	3.3	16.4	23.1	31.4
SELETT3	5.6	4.8	16.28	23.8	15.4
SELETT4	4.2	5.0	16.2	14.3	35.2
N_PRED1	18.6	3.1	19.8	17.6	25.7
N_PRED2	7.8	3.1	19.8	19.1	19.1
N_PRED3	5.8	2.4	18.9	17.6	15.6
AGGREG1	29.5	9.3	16.7	17.1	38.4
AGGREG2	29.8	4.1	15.7	15.1	40.8
AGGREG3	29.9	4.5	21.8	21.1	42.8
AGGREG4	30.3	4.5	19.1	18.6	43.1
AGGREG5	32.4	4.9	20.4	19.3	45.3
N_COL1	21.5	3.6	19.1	18.4	32.4
N_COL2	22.0	3.6	21.4	27.1	32.0
N_COL3	24.0	4.2	25.5	33.1	40.2
ORDER_1	35.3	100.8	38.3	-	245.8
ORDER_2	3.4	12.2	-	-	84.0
COOCC	154.7	82.5	-	-	654.3

Tutte le query eseguite sono di tipo OLAP, in cui i dati vengono raggruppati per una o più dimensioni in cui, per ogni gruppo, viene mostrato il numero di occorrenze rilevate.

- Nel primo blocco di interrogazioni (SELETT) i dati sono stati raggruppati per classe e channel mediatype, e le query sono caratterizzate da una variazione via via crescente della selettività (da 1 a 0.004).

- Nel secondo blocco di interrogazioni (N_PRED) i dati sono stati raggruppati per nome del topic, e le query sono caratterizzate da una variazione via via crescente sul numero di predicati (variando la selettività da 0.54 a 0.012).

- Nel terzo blocco di interrogazioni (AGGREG) i dati sono stati raggruppati per party family, e le query sono caratterizzate da una variazione della seconda dimensione utilizzata per costruire il risultato,producendo un risultato che varia da 18 a 49721 tuple. Per la seconda dimensione sono stati utilizzati in ordine: continent, channel mediatype, county, city e source.

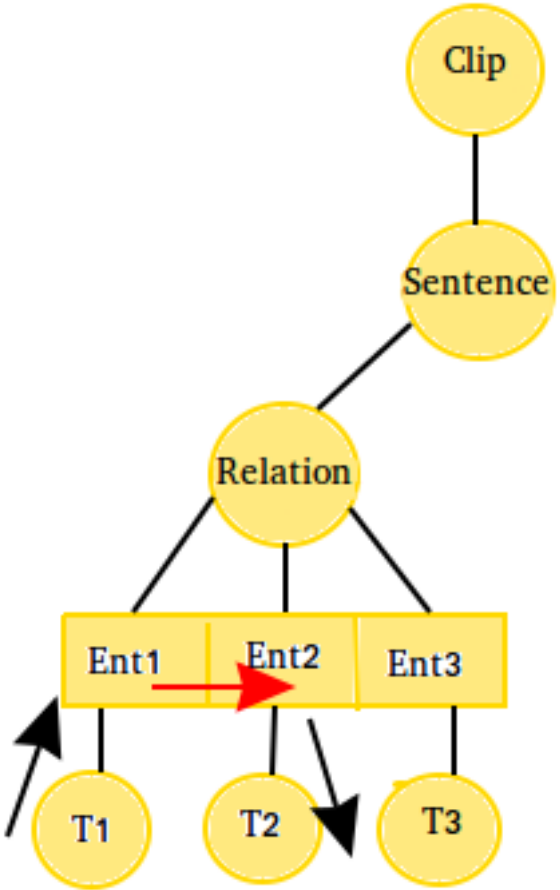
- Nel quarto blocco di interrogazioni (N_COL) è stato variato l'insieme di raggruppamento del risultato. In particolare, nella prima query i dati sono stati raggruppati per politician e city, mentre nella seconda query è stato effettuato un raggruppamento per politician, city, party, party family e epg, e infine nell'ultima query i dati sono stati raggruppati per politician, city, county, country e continent.

- Nel quinto blocco di interrogazioni (ORDER) i dati sono stati raggruppati per nome dell'entità, e le query sono caratterizzate dall'ordinamento del risultato.

- Nel sesto blocco di interrogazioni (COOCC) i dati sono stati raggruppati per nome dell'entità, e la query consiste nel calcolare le co-occorrenze delle delle entità. Anche se la query risulta estremamente complessa, Impala mostra un tempo di risposta tutto sommato accettabile, mentre Oracle fornisce un risultato, ma non in tempo utile.
  Per quanto riguarda Elasticsearch, non è possibile eseguire questo tipo di query a causa di una lacuna nelle API. In Elasticsearch, alla versione

1.7, non è possibile navigare all'interno della gerarchia come mostrato in figura. Elasticsearch, come indicato nella freccia rossa, non permette di navigare tra due entità appartenenti alla stessa relation (rendendo di fatto impossibile calcolare le cooccorrenze tra topic). Questa mancanza è stata colmata nella versione 2.0 di Elasticsearch attraverso l'introduzione delle aggregazioni in pipeline.

Figura 3.14: Lacuna nella navigazione delle gerarchie



Infine, effettiamo una serie di considerazioni riguardo alle performance:

- I tempi di Elasticsearch mostrano pochissime variazioni, a differenza di Oracle e Impala che sono più sensibili alle variazioni di selettività. Questo è dovuto alla differenza di modellazione dei dati; i tempi di ricerca di Elasticsearch sono molto bassi, mentre la complessità maggiore (che determina la maggior parte del costo) risiede nella composizione dell'aggregazione.

- I tempi variano sensibilmente rispetto alla granularità del risultato (AGGREG) e al numero di colonne (N_COL).

- Per quanto riguarda l'ordinamento, Elasticsearch deve invertire i suoi inverted index (che abbiamo introdotto della descrizione del sitema), ed essendo un'operazione estremamente complessa, nella maggior parte dei casi il sistema, anche con un cluster di sette nodi, non è stato capace di eseguire l'interrogazione.

- Elasticsearch dual node, rispetto alla versione full, in alcuni casi impiega meno tempo ad eseguire le query, ma si dimostra più sensibile a vari tipi di variazioni, quali che siano di aggregazione o di selettività.

- E' importante notare come i tempi siano inferiori rispetto ad Oracle non solo nella versione full cluster, ma -molto spesso- anche nella versione dual node.

- Sebbene Impala, effettuando il filtro sui topic Unclassified, sia molto più veloce di Elasticsearch, quest'ultimo riesce a competere con la versione senza filtro, considerando il fatto che la modellazione dei dati è completamente differente e che Elasticsearch non nasce per analisi di questo tipo.

Oltre alle query di tipo OLAP, comuni a tutte e tre le piattaforme, sono stati effettuati dei test sulle funzionalità di ricerca full-text di Elasticsearch, che Impala e Oracle non possiedono.

Figura 3.15: Tempi di esecuzione query Elasticsearch

	Elasticsearch	Elasticsearch Dual Node
Term_occ1	32.3	43.3
Term_occ2	33.6	48.3
Sig_term1	31.3	52.1
Sig_term2	31.9	46.9
Co_occ_ft	1.3	1.8

Nel primo blocco di interrogazioni è stata testata una funzionalità di Elasticsearch: fornire l'occorrenza dei termini, in ordine discendente, di un blocco di testo. La query è stata effettuata applicando filtri per la variazione della selettività dei documenti, tenendo in considerazione una particolare data e il content (filtro sulla presenza della parola ukip, corrispondente ad uno dei partiti inglesi). Come possiamo notare dalla tabella, diminuire la selettività consente al sistema di effettuare meno analisi su grandi blocchi di testo, e come ci aspettavamo il tempo di risposta diminuisce. Il risultato dell'interrogazione consiste in una coppia termine-valore, che indica il numero di occorrenze del termine all'interno di tutte le clip considerate dopo l'esecuzione del filtro.

Nel secondo blocco di interrogazioni è stata testata ancora una volta una funzionalità di Elasticsearch: fornire una lista dei termini più significativi presenti all'interno di un blocco di testo. Al variare della selettività in questo caso, i tempi rimangono abbastanza stabili, aumentando in maniera apprezzabile su un cluster di soli due nodi. Anche in questo caso a query è stata effettuata applicando filtri per la variazione della selettività dei documenti, tenendo in considerazione una particolare data e il content (filtro sulla presenza della parola ukip). Il risultato dell'interrogazione consiste in una coppia termine-valore, che indica la rilevanza di un certo termine (calcolata da Elasticsearch con un particolare algoritmo), valutata all'interno delle clip che soddisfano le condizioni di filtro.

Nel terzo blocco di interrogazioni è stata testata una delle funzionalità princi-

pali di Elasticsearch: la correlazione di termini all'interno di blocchi di testo. In questo caso, lo scopo della query era di valutare ed indicare, in ordine di rilevanza, le clip contenenti i termini Farage e Cameron. Questo tipo di query restituisce i documenti, in ordine di rilevanza, in cui i termini selezionati sono correlati, secondo un particolare algoritmo proprio di Elasticsearch. Essendo il sistema pensato principalmente per questo tipo di query, il tempo di risposta risulta incredibilmente basso in entrambe le configurazioni del cluster. A titolo di esempio, mostriamo il codice dell'interrogazione:

```
GET firb_eng_single_doc/clip/_search
{
    query:{
        match:{
            content:{
                query: farage cameron,
                operator: and
            }
        }
    }
}
```

Come possiamo notare dall'esempio, la clausola responsabile della ricerca full-text è match. L'incredibile semplicità con cui è possibile eseguire una interrogazione tanto potente è uno dei pregi più importanti e apprezzati di Elasticsearch.

Conclusioni

L'avvento del fenomeno dei Big Data ha portato con se un innumerevole quantità di aspetti e implicazioni che si sono percepite in diverse aree di business, specialmente nell'ambito della Social Business Intelligence. Sia chi deve trarre informazioni da questa tipologia di dati, sia chi li deve gestire e manipolare, si trova si fronte a un mondo che, nonostante sia molto discusso e noto a tutti, presenta molti aspetti sconosciuti. Le molte aziende che, fino ad oggi, hanno fatto del dato il loro business, si trovano di fronte a un grosso cambio generazionale. Le sole tecniche tradizionali risultano non essere più sufficienti a soddisfare i requisiti di elaborazione e gestione dei Big Data. Le tecnologie legate a questo fenomeno sono molte e molte altre nasceranno. Le nuove tecnologie non sono state sviluppare per sostituire le tecniche fino ad oggi utilizzate, ma per affiancare gli strumenti già in produzione al fine di estra polare valore da questa nuova tipologia di dati. I due mondi dovranno coesistere e cooperare, cercando di sfruttare e fare emergere il meglio da ognuna. Non vi è un unico strumento al quale affidarsi, non vi sono regole o standard particolari che permettono di gestire questa moltitudine di tecnologie. In un contesto di Social Business Intelligence, nel quale è stata analizzata una grande quantità di dati relativi alla Politica, abbiamo scelto di utilizzare la tecnologia di Elasticsearch. Questo sistema, ancora in fase di sviluppo, è stato utilizzato come soluzione enterprise da molte aziende, tra cui Ebay e Wordpress, ed è al secondo posto tra i motori di ricerca più utilizzati al mondo. Le sue caratteristiche di scalabilità e affidabilità sono cruciali per realizzare moderni sistemi di analisi del buisness, dove un singolo

nodo relazionale estremamente potente risulta essere inadeguato alla grande quantità di dati caratteristica dei Big Data. Inoltre Elasticsearch fornisce strumenti molto potenti per eseguire ricerche di tipo full-text e per l'analistica in tempo reale, caratteristiche piuttosto rare anche nei moderni sistemi NoSQL. Nell'ambito del progetto realizzato in questa tesi, la soluzione sviluppata si è dimostrata più efficiente e performante rispetto al tradizionale cubo relazionale; all'aumentare dei dati infatti, le tradizionali basi di dati relazionali risultano estremamente inefficienti, impiegando tempi di elaborazione non compatibili in un contesto in cui l'User Experience e la rapidità con cui vengono prese delle decisioni è molto importante. I risultati operativi derivanti dai test hanno mostrato che l'utilizzo delle capacità di aggregazione e ricerca offerte da Elasticsearch, permette di ridurre sensibilmente i tempi di risposta. Alla luce di tutto ciò, si può concludere che in presenza di Big Data i tradizionali sistemi di analisi devono essere affiancati a tecnologie in grado di gestire enormi quantità di dati, garantendo affidabilità e rapidi tempi di risposta. La mancanza però di uno standard e di adeguate metodologie di modellazione crea problemi in termini di progettazione e sviluppo, rendendo queste tecnologie ancora non pervasive. In particolare, essendo ancora in fase di sviluppo, spesso alcune funzionalità importanti risultano essere in fase di sperimentazione o, come nel caso di Elasticsearch, non ancora disponibili. Nonostante tutto ciò le due tipologie di strumenti (Big Data e Tradizionali) continuano coesistere e collaborare. Questo aspetto continuerà a presentarsi anche in altri contesti progettuali.

Bibliografia

[1]*Elastic - Revealing Insights from Data, http://www.elastic.co*

[2]*Elasticsearch: The Definitive Guide, Clinton Gormley & Zachary Tong*

[3]*No SQL databases, http://nosql-database.org*

[4]*Apache HBase, http://hbase.apache.org*

[5]*Cassandra, https://cassandra.apache.org/*

[6]*MongoDB Manual, http://docs.mongodb.org/manual/*

[7]*Base model of data, http://databases.about.com/Abandoning-Acid-In-Favor-Of-Base.htm*

[8]*NoSQL, https://en.wikipedia.org/wiki/NoSQL*

[9]*Impala, http://impala.io/index.html*

[10]*NoSQL databases explained, https://www.mongodb.com/nosql-explained*

[11] *What is Big Data?, http://www.ibm.com/big-data/us/en/*

[12] *Marvel: Monitor and Manage Elasticsearch, https://www.elastic.co/products/marvel*

[13] *Kibana: Explore - Visualize - Discover Data, https://www.elastic.co/products/kibana*

[14] *Progettezione e sviluppo di una soluzione Hadoop per il calcolo di big data analytics, Tesi di Laurea Magistrale di Francesca Marchi, Anno accademico 2013-2014*

[15] *Gallinucci E., Golfarelli M., Rizzi S., Advanced topic modeling for social business intelligence. Information Systems (2015).*

[16] *Francia, M., Golfarelli, M., & Rizzi, S.. A methodology for social BI. In Proceedings of the 18th International Database Engineering & Applications Symposium (pp. 207-216). ACM, 2014.*

Ringraziamenti

Innanzitutto desidero ringraziare i miei genitori, Elisabetta e Vincenzo, e mia sorella Camilla, per il loro amore e per il grande supporto che mi hanno dato durante tutto il corso dei miei studi. Vorrei ringraziare la mia ragazza, Marina, per il suo grande amore e per il supporto che mi ha dato in tutti questi anni; senza di lei sarebbe stato tutto molto più difficile. Vorrei ringraziare tutti i miei amici di corso, che nonostante la mia indole un po' scontrosa, mi hanno sempre sostenuto e apprezzato per quello che sono. Vorrei ringraziare i miei nonni, per l'amore, la sicurezza e il sostegno che mi hanno dato in tutti questi anni. Ringrazio il professor Matteo Golfarelli per la grande opportunità di crescita chie mi ha dato, nonchè per la sua disponibilità e professionalità. Ringrazio di cuore il mio co-relatore, Enrico Gallinucci, per l'abilità, l'incredibile pazienza e disponibilità che ha mostrato durante lo sviluppo del progetto. Ringrazio infine il mio caro nonno Gigi, che ora non è più tra noi, per la passione e la forza che mi ha trasmesso; senza il suo consiglio non sarei la persona che sono.