Università degli Studi di Modena e Reggio Emilia

Dipartimento di Ingegneria “Enzo Ferrari”

Corso di Laurea Triennale in Ingegneria Informatica

Full-Text Search e applicazioni:

Confronto tra Microsoft Sql Server ed Elasticsearch

Relatore:

Prof. Sonia Bergamaschi

Correlatore:

PhD Luca Gagliardelli

Candidato:

Davide Corradi

Anno Accademico 2019-2020

Prima di procedere con la trattazione, vorrei dedicare qualche riga a tutti coloro che mi sono stati vicini in questo percorso di crescita personale e professionale.

Ringrazio i miei genitori che mi hanno sempre sostenuto, appoggiando ogni mia scelta, dalla scelta del mio percorso di studi all'incoraggiamento e supporto quotidiano.

Grazie al mio relatore Sonia Bergamaschi per avermi fornito materiale utile alla stesura dell'elaborato.

Ringrazio le mie sorelle Aurora e Sara per avermi reso le giornate di studio e i momenti di difficoltà meno complicati e vuoti.

Un ringraziamento speciale è dedicato a Giulia, che mi ha supportato, accompagnato e aiutato in questo periodo intenso e poi poter gioire insieme dei traguardi raggiunti.

Ringrazio i miei amici e colleghi per le idee e l'originalità che rendono il periodo universitario speciale e indimenticabile.

Infine, dedico questa tesi a me stesso, che dopo sacrifici lavorativi e di studio mi hanno permesso di raggiungere felicemente questo traguardo.

INDICE

Sommario

1. Introduzione

1.1 Importanza del text-search

Al giorno d'oggi risulta impossibile non avere contatti quotidianamente con sistemi per la gestione di basi di dati. Accedendo a internet visitiamo siti di vendita online, prenotiamo voli o camere d'hotel, controlliamo il nostro conto bancario, leggiamo le notizie giornaliere, ecc. Tutte queste azioni prevedono l'accesso e l'aggiornamento ad una base di dati. Con lo sviluppo di internet e dei sistemi cloud si è vista l'esigenza di pubblicare e condividere una quantità enorme di dati e documenti.

La ricerca di contenuti all'interno di una mole così ingente di dati è stata ed è ancora una sfida per gli sviluppatori di tutto il mondo. C'è la necessità di avere a disposizione moltissimi dati, nel minor tempo possibile.

In un contesto in cui siamo circondati da grandi sorgenti con flussi di dati che presentano irregolarità, diversità di formato e di locazione bisogna avere database molto flessibili, scalabili e affidabili. Il modello relazionale ha rappresentato per decenni un esempio e un efficace metodo per la gestione di una base di dati. Si sta cercando di andare oltre per rispondere a esigenze di mercato che cercano sempre più nella velocità di ricerca dei dati vicino al real-time il fine a cui puntare. Sono nati per questo motivo modelli con molta più dinamicità sia nella gestione dei dati ma anche nella loro modellazione e visualizzazione come quelli basati su sistemi NoSQL, caratterizzati anche da una alta scalabilità. Le performance anche in presenza di dataset di notevoli dimensioni sono in alcuni casi ottime e le varie funzionalità, anche non banali e complesse, non inficiano sulla semplicità e comodità di utilizzo di questi DBMS.

Dal NoSQL si sono evoluti modelli dedicati alla search engine (alla ricerca dei dati) e document-oriented (caratterizzato da una organizzazione con schema libero dei dati). La ricerca di documenti all'interno delle nostre basi di dati sarà il tema che accompagnerà l'elaborato.

Questo progetto di tesi è scaturito da una analisi di quelle che sono le varie tipologie di ricerca full-text search nei vari modelli proposti da sistemi relazionali, NoSQL e il suo sviluppo orientato alla creazione dei search engine.

Nel primo capitolo, definiremo cos'è la ricerca full-text, mentre il secondo capitolo tratterà dei sistemi relazionali classici prendendone in esame uno dei più famosi: MS SQL Server. Analizzeremo come viene implementata la full-text search e i pro e i contro di questo modello e quali siano le difficoltà che possono nascere nel tentativo di conformarsi alle richieste ed esigenze odierne.

Il terzo capitolo parlerà dei modelli NoSQL, delle novità apportate alla concezione di database. Il quarto tratterà di Apache Lucene, come è costruita questa libreria e della sua importanza. Entrambe le tecnologie sono alla base dello sviluppo di Elasticsearch che sarà il punto cardine del quarto capitolo.

Guarderemo come il tema della ricerca di testo sia centrale nell'ecosistema odierno e come questo, nello specifico grazie ad Elasticsearch, ci permette di visualizzare e modellare i nostri dati.

Confronteremo risultati e specifiche tecniche per capire come lo sviluppo delle tecnologie e i sistemi hanno migliorato la full-text search per comprendere al meglio come adattarsi al futuro sempre più accompagnato da questo genere di ricerca.

Nell'appendice troveremo una trattazione di MongoDB, per capire come siano sviluppati e concepiti i DBMS NoSQL, ma non utile al tema della full-text search.

Sebbene MongoDB implementi una ricerca di testo, esso viene comunque accompagnato da motori di ricerca full-text, tra i quali anche Elasticsearch.

1.2 Full text-search

1.2.1 Ricerca e recupero delle informazioni

Prima di definire e analizzare la tecnica full text search, fulcro di questo elaborato, è doveroso dare uno sguardo più ampio, purtroppo non sufficientemente esaustivo, di quello che è il mondo del reperimento e recupero delle informazioni.

Una delle tecniche più importanti per il recupero delle informazioni è la *Document Retrieval*[1] (recupero dei documenti) che è definita come la corrispondenza (matching) di alcune query dichiarate dall'utente con un insieme di free-text records. Questi record potrebbero essere qualsiasi tipo di testo non strutturato, come articoli di giornale, documenti immobiliari o paragrafi di un manuale. Le query degli utenti possono variare da descrizioni complete di informazioni di cui si ha bisogno, a più frasi, a poche parole.

La document retrieval è talvolta denominata *text retrieval*[2]. Essa, in particolare, è una branca del recupero delle informazioni in cui le informazioni vengono memorizzate principalmente sotto forma di testo. Il text retrieval è oggi un'area critica di studio, poiché è la base fondamentale di tutti i motori di ricerca in Internet.

[1] [30] Document Retrieval. https://en.wikipedia.org/wiki/Document_retrieval

[2] Spesso usata come sinonimo della Document Retrieval. Spiegazione specifica in: [30] Document Retrieval. https://en.wikipedia.org/wiki/Document_retrieval

Un sistema di document retrieval è costituito da un database di documenti, un algoritmo di classificazione per creare un indice di testo completo e un'interfaccia utente per accedere al database. La Text Retrieval ha due compiti principali:

1. Trovare documenti pertinenti alle domande degli utenti.
2. Valutare i risultati del matching e li ordina in base alla rilevanza, utilizzando algoritmi specifici.

I motori di ricerca su Internet sono applicazioni classiche di document/text retrieval. La stragrande maggioranza dei sistemi di retrieval attualmente in uso va dai semplici sistemi booleani a sistemi che utilizzano tecniche di elaborazione statistica o del linguaggio naturale.

Nel mondo vasto e complesso della text retrieval, la full text search si riferisce a delle tecniche per cercare un unico documento archiviato su computer o una collezione in un full-text database. Un *full-text database*[3] è un database che contiene i testi completi di libri, saggi, giornali, riviste e moltissimi altri tipi di documenti testuali.

Full-text search è distinta dalle ricerche basate sui *metadati*[4] o su parti di testi originali rappresentati nei database (come titoli, astrazioni, sezioni selezionate o riferimenti bibliografici).

In una ricerca full-text, un motore di ricerca esamina tutte le parole in ogni documento memorizzato mentre cerca di soddisfare i criteri di ricerca (ad esempio, il testo specificato da un utente). È facile comprendere come una tecnica di ricerca full-text sia comoda e utile in un sistema software che è stato progettato per eseguire ricerche web. La particolarità di questo genere di ricerca è che i risultati ottenuti sono generalmente rappresentati in righe e queste informazioni

[3] Full-Text Database. https://en.wikipedia.org/wiki/Full-text_database
[4] Metadata: https://en.wikipedia.org/wiki/Metadata

sono un insieme di link a web pages, immagini, video, informazioni geografiche, articoli di giornali, ecc.

Le tecniche di ricerca full-text sono diventate comuni nei database bibliografici online negli anni '90 (anche se questo tipo di database era formato da metadata più che testi completi, c'è la presenza di molti riferimenti e quindi non si sfrutta a pieno la potenzialità di questa tecnica).

I metadata sono dati che forniscono informazioni su altri dati. In altre parole, sono dati sui dati. Esistono molti tipi di metadati come metadati descrittivi, metadati strutturali, metadati amministrativi, metadati di riferimento e metadati statistici. Le informazioni gestite nell'ambito dei metadati non possono descrivere il documento nei suoi dettagli. Se fosse necessario rintracciare i documenti in archivio attraverso keyword non prevedibili in fase di archiviazione, l'analisi con metadati perde di utilità. Di solito la full text search, come detto precedentemente, evita l'utilizzo di metadati e la potenza di questa tecnica risiede nel fatto che la ricerca avviene in modo "full", completo, oggi con prestazioni elevatissime, senza il bisogno di effettuare query solo su porzioni strutturate di testo come titoli e sommari.

Molti siti web e programmi applicativi (come il word processing software, programma per computer che fornisce editing, formattazione con output di testo, con funzionalità aggiuntive) forniscono funzionalità di ricerca full-text.

Alcuni motori di ricerca web, come faceva *AltaVista*[5], impiegano tecniche di ricerca full-text, mentre altri indicizzano automaticamente solo una parte delle pagine web esaminate dai loro sistemi di indicizzazione.

AltaVista era un motore di ricerca fondato nel 1995, acquistato da Yahoo! nel 2003 e chiuso nel 2013 a causa della enorme diffusione di Google che lo rese di fatto

[5] [31] Il Giornale.it, Maddalena Camera, 2013.

inutilizzato. Proprio quest'ultimo ha reso disponibile al pubblico, da aprile 2018, una importante e nuova tecnologia chiamata *Talk to Books*[6], che consente di porre delle domande per poi avere informazioni estratte da più di 100.000 libri, per un totale di oltre 600 milioni di frasi. È un esempio perfetto di quella che è la potenzialità della full-text search, in particolare della *ricerca semantica*[7], cioè la ricerca basata sul significato di ciò che chiediamo e non basato su parole chiave, frasi, parole esatte.

1.2.2 Indicizzazione

Quando si tratta di un piccolo numero di documenti, è possibile per il motore di ricerca full-text scansionare direttamente il contenuto dei documenti con ogni query, una strategia chiamata "serial scanning".

Tuttavia, quando il numero di documenti da cercare è potenzialmente elevato o la quantità di query di ricerca da eseguire è notevole, il problema della ricerca full-text è spesso suddiviso in due attività: *indicizzazione e ricerca*[8]. La fase di indicizzazione eseguirà la scansione del testo di tutti i documenti e creerà un elenco di termini di ricerca (chiamato indice). Nella fase di ricerca, quando si esegue una query specifica, viene fatto riferimento solo all'indice, anziché al testo dei documenti originali.

L'indicizzatore inserirà una voce nell'indice per ogni termine o parola trovati in un documento e, eventualmente, annoterà la sua posizione relativa all'interno del documento. Di solito l'indicizzatore ignorerà le "stopword" (come "il" o "e") che sono comuni e non sufficientemente significative per essere utili nella ricerca. Alcuni indicizzatori utilizzano anche la radice specifica della lingua utilizzata

[6] [32] Talk to books. *https://books.google.com/talktobooks/*
[7] [29] Google annuncia la sua nuova funzionalità semantica per interagire con i libri, medium.com, 2018.
[8] [5] Full-Text search. https://en.wikipedia.org/wiki/Full-text_search

sulle parole indicizzate (*stemming*). Ad esempio, le parole "drive", "drove" e "driven" verranno registrate nell'indice sotto la singola parola "drive".

1.2.3 Il compromesso tra precision e recall

Recall misura la quantità di risultati rilevanti restituiti da una ricerca rispetto a quelli effettivamente rilevanti, mentre la precision è la misura della qualità dei risultati restituiti rispetto al numero totale di risultati restituiti. In generale, alta precision significa che vengono presentati meno risultati irrilevanti (pochi falsi positivi), mentre una alta recall significa che meno risultati rilevanti sono stati persi (falsi negativi).

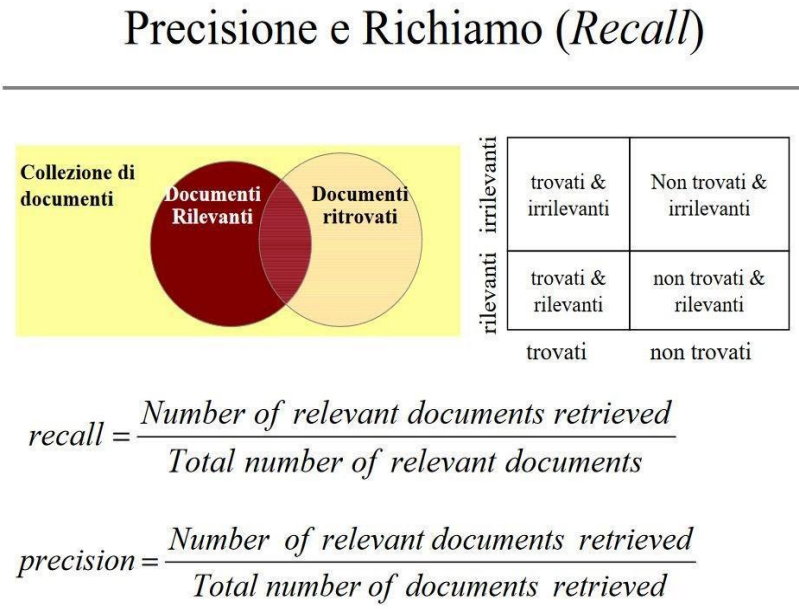

Figura 1.1 *Sistemi di Information Retrieval: Performance Evaluation [25]*

La full-text search offre molta flessibilità, ti permette di ridurre la precision a favore della Recall. Entrando nel dettaglio, la maggior parte delle implementazioni di ricerca full-text usano un "*inverted index*" (indice invertito). Si tratta di un indice in cui le chiavi sono i termini individuali e i valori associati sono set di record (o documenti) che contengono il termine. Il tutto è ottimizzato per calcolare l'intersezione, l'unione, ecc. di questi set di record e solitamente

fornisce un algoritmo di classificazione per quantificare la forza con cui un dato record corrisponde alle parole chiave di ricerca.

A causa delle ambiguità del linguaggio naturale, i sistemi di ricerca full-text in genere includono opzioni di stopwords per aumentare la precision e lo *stemming* per aumentare la Recall. La ricerca con controllo su vocabolario, chiamata ricerca semantica (per sinonimi o *significato*) aiuta anche ad alleviare i problemi di bassa precisione contrassegnando i documenti in modo tale da eliminare le ambiguità. Il compromesso tra precision e recall è semplice: un aumento della precision riduce la recall complessiva, mentre un aumento del recall riduce la precision[9].

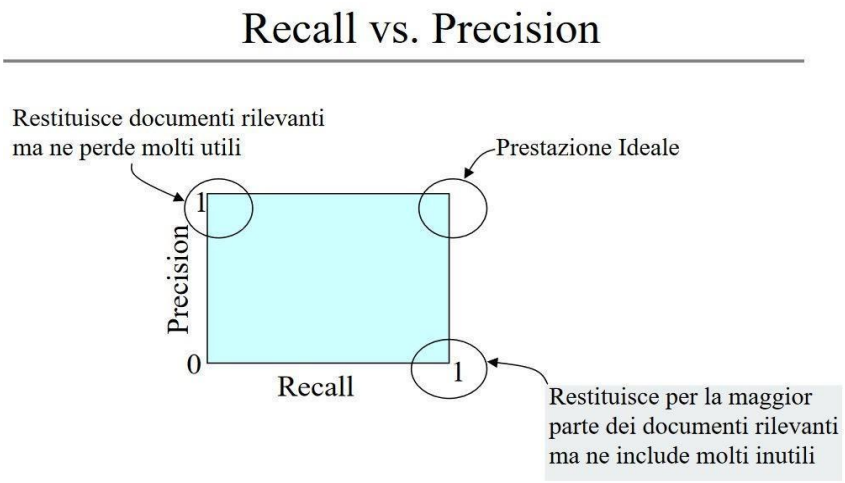

Figura 1.2 *Sistemi di Information Retrieval: Performance Evaluation [25]*

1.2.4 Problema dei falsi positivi

È probabile che la ricerca full-text recuperi molti documenti non pertinenti alla domanda di ricerca prevista. Tali documenti sono chiamati falsi positivi. Il recupero di documenti irrilevanti è spesso causato dalla ambiguità intrinseca del linguaggio naturale. Falsi positivi sono i risultati irrilevanti della analisi di

[9] Si ritrovano maggiori informazioni riguardanti i concetti trattati in: [25] Sistemi di Information Retrieval: Performance Evaluation, R. Basili, a.a 2004-5.

precision e recall. Ci sono varie tecniche di clustering (ad esempio basate su algoritmi bayesiani) che permettono di ridurre i falsi positivi, riuscendo a classificare quello che è l'universo dei documenti e dei dati inserendoli in categorie.

1.2.5 Miglioramenti delle prestazioni

Si può quindi intuire che a livello prestazionale e di efficacia la full-text search può avere molte problematiche e queste carenze sono state affrontate in due principali criteri: fornendo agli utenti strumenti che consentono loro di esprimere le loro domande di ricerca in modo più preciso e sviluppando nuovi algoritmi di ricerca che migliorano la precisione del recupero[10].

Ecco alcune delle soluzioni per migliorare le query:

- Parole chiave. Ai creatori di documenti (o indicizzatori) viene chiesto di fornire un elenco di parole che descrivono l'oggetto o argomento, inclusi i sinonimi. Le parole chiave migliorano il recall, soprattutto se queste non sono presenti nel documento di testo.
- Ricerca limitata al campo. Alcuni motori di ricerca consentono agli utenti di limitare le ricerche a un campo particolare di un record di dati archiviato, come Autore o Titolo.
- Query booleane. Le ricerche che utilizzano operatori booleani possono aumentare notevolmente la precisione di una ricerca di testo libero. L'operatore AND dice "Non recuperare alcun documento a meno che non contenga entrambi questi termini". L'operatore NOT dice "Non recuperare alcun documento che contiene questa parola". Se l'elenco di recupero recupera un numero insufficiente di documenti, l'operatore OR può essere utilizzato per aumentare il recall; si

[10] Per ulteriori informazioni sulle tipologie di ricerca elencate, visitare le pagine dedicate in: Performance improvements, [5] Full-Text Search, https://en.wikipedia.org/wiki/Full-text_search

consideri, ad esempio, "enciclopedia" AND "online" OR "Internet" NON "Encarta". Questa ricerca recupererà documenti sulle enciclopedie online che utilizzano il termine "Internet" invece di "online". Questo aumento della precisione è molto comunemente controproducente poiché di solito si accompagna a una drammatica perdita di recall.

- Ricerca per frase. Una ricerca per frase trova solo i documenti che contengono una frase specifica.
- Ricerca di concetti. Una ricerca basata su concetti composti da più parole, come la ricerca semantica.
- Ricerca di concordanza. Una ricerca di concordanza produce un elenco alfabetico di tutte le parole principali che si trovano in un testo con il loro contesto immediato.
- Ricerca di prossimità. Una ricerca per frase che trova solo quei documenti che contengono due o più parole separate da un numero specificato di parole; una ricerca di "Youtube" WITHIN2 "Ulisse" recupererebbe solo quei documenti in cui le parole "Youtube" e "Ulisse" si trovano a due parole l'una dall'altra.
- Espressione regolare. Un'espressione regolare utilizza una sintassi di query complessa ma potente che può essere utilizzata per specificare le condizioni di recupero con precisione.
- La ricerca fuzzy cercherà il documento che corrisponde ai termini dati e qualche variazione intorno a essi.
- Ricerca con caratteri jolly. Una ricerca che sostituisce uno o più caratteri in una query di ricerca con un carattere jolly come un asterisco. Ad esempio, utilizzando l'asterisco in una query di ricerca "s * n" si troverà "sin", "son", "sun" e così via in un testo.

Oppure si deve puntare sul miglioramento degli algoritmi di ricerca:

> Un esempio è l'algoritmo *PageRank*[11] sviluppato da Google dà maggiore risalto ai documenti a cui sono collegate altre pagine Web.

Non è facile costruire una buona ricerca full-text e avere a disposizione una buona tecnologia può tornare molto utile. I maggiori sistemi per la ricerca full-text hanno caratteristiche in comune e possono essere riassunte come:

- Risultati di ricerca quasi real-time che indicano quali documenti su milioni potrebbero contenere uno o più termini (una parola, un numero, ecc.). Capacità di includere la suddivisione in categorie e classificazioni efficaci del contenuto della ricerca in base a valori e campi specifici predeterminati.
- La possibilità di creare delle query ricche, flessibili, sofisticate per migliorare le proprie ricerche di documenti e record.
- Oltre alla ricerca e l'indicizzazione, molti prodotti software permettono anche la memorizzazione dei dati (come *Lucene* e *Solr*[12], che vedremo in seguito).
- Si hanno però delle limitate capacità per cercare e manipolare dei dati che nei fatti rappresentano differenti tipi di record (da qui la sfida dei software odierni per la ricerca).

Prodotti che si basano su sistemi software come Lucene o Solr per la ricerca full-text hanno caratteristiche sorprendenti come velocità, buona classificazione di rilevanza fuori dagli schemi comuni, funzionalità di query complete, elaborazione completa dei risultati, portabilità, scalabilità, bassi indici di overhead e rapido incremento degli indici.

[11] Per maggiori informazioni visitare il sito: https://sitechecker.pro/it/page-rank/
[12] Sito proprietario di Apache: https://lucene.apache.org/index.html

2. Full-text search, modello relazionale

In questo capitolo analizzeremo come viene implementata la full text search in Microsoft SQL Server, software preso in considerazione per esaminare l'implementazione di ricerca di testo di un classico DBMS relazionale. Vedremo quali sono i limiti e anche i punti di forza di questo DBMS e perché col tempo è stato sostituito o accompagnato da altre tipologie di sistemi che permettono una ricerca di dati differente.

2.1 Il modello relazionale

I Relational Database Management System hanno dominato tutto il settore informativo per anni, tutti basati su un modello logico di rappresentazione e gestione dei dati di tipo relazionale, come ad esempio Oracle, MySQL e Microsoft SQL Server.

Le basi di questo modello furono dettate inizialmente da Edgar F. Codd prima nel 1970 e successivamente ampliate nel 1985 con i rispettivi articoli: "A *Relational Model of Data of Large Shared Data Banks*", "*Is your DBMS really Relational*" e "Does *your DBMS run by the Rules?*"[13]. Per determinare coerenza e integrità a un modello di dati in ascesa, vengono stabilite tredici regole da rispettare per considerare un DBMS effettivamente relazionale. Ad esempio, un RDBMS deve operare esclusivamente attraverso capacità relazionali e che ogni informazione deve, a livello logico, essere rappresentata per mezzo di un valore di una tabella. Fondamentale, inoltre, è l'indipendenza dei dati sia a livello logico che fisico garantita attraverso anche un modello semplice con basi teoriche ben salde. Con

[13] [24] Analisi e sperimentazione del DBMS NoSQL MongoDB: il caso di studio della Social Business Intelligence, Tesi di Laurea di Alice Gambella, Anno Accademico 2013/2014.

la possibilità di elaborare dati tramite tabelle e avere una struttura fissa dei dati dà l'opportunità di lavorare su più informazioni contemporaneamente e con un linguaggio di alto livello. Pratiche non scontate per l'epoca tecnologica in cui furono concepite, ma sono dettagli che porteranno a considerare troppo rigida questa gestione di dati per compiere una ricerca veloce, con un conseguente allontanamento verso altri sistemi e search engine differenti.

Negli anni i DBMS relazionali si sono sviluppati adeguandosi a quelle che erano e che sono tutt'oggi richieste di mercato: motori di ricerca veloci, precisi e intuitivi. Diamo ora un ampio e non completamente esaustivo sguardo su cosa Microsoft SQL Server ci mette a disposizione per la ricerca full-text, riassumendo ciò che viene descritto sul sito proprietario *Microsoft*[14].

2.2 Come avviene la full text search in SQL Server

La full-text Search è una componente opzionale di SQL Server e deve essere selezionata al momento della installazione del programma.

2.2.1 Indicizzazione

La prima cosa da compiere è quella di definire un indice, fondamentale per la ricerca. L'indice full-text include una o più colonne basate su caratteri in una tabella qualsiasi del nostro database. Queste colonne possono avere uno dei tipi di dati seguenti: char, varchar, nchar, nvarchar, text, ntext, image, xml o varbinary(max) e FILESTREAM. Ogni indice full-text consente di indicizzare una o più colonne della tabella e ciascuna colonna può essere utilizzata con una lingua specifica.

[14] [10] Ricerca full-text, Documentazione di SQL, 2018, https://docs.microsoft.com/it-it/sql/relational-databases/search/full-text-search?view=sql-server-ver15

Un qualsiasi indice full-text viene archiviato in quello che Microsoft chiama "*catalogo full-text*"[15]. È un contenitore per gli indici full-text che serve per gestirli e visualizzarli in modo semplice e chiaro, dato che spesso sono suddivisi in più tabelle interne denominate frammenti di indice full-text. È possibile definire un solo indice per tabella e l'aggiunta di dati ad esso è definita *popolazione*. Se noi inseriamo dati in una tabella indicizzata, non necessariamente l'indice full-text viene aggiornato perché il suo popolamento avviene in modo asincrono. In realtà questa modalità può essere cambiata e il rilevamento delle modifiche, con il conseguente aggiornamento dell'indice, può essere di tre tipi: *auto*, *manual*, *off*.

AUTO: chiede a SQL Server di tenere traccia delle modifiche e di compiere il popolamento automaticamente.

MANUAL: tiene traccia delle modifiche ai dati per una tabella e consente all'utente di richiedere il popolamento dell'indice.

OFF: SQL Server non tiene traccia delle modifiche che vengono apportate alla tabella e di conseguenza anche il popolamento dell'indice deve avvenire completamente manualmente.

L'aggiornamento dell'indice non è automatico anche in altri sistemi, come in Elasticsearch, il quale ha anche le problematiche aggiunte della *replicazione* e dello *sharding*[16], descritti in seguito.

Per creare un indice nella tabella bisogna innanzitutto creare un *catalog* che ci permette di memorizzare i nostri indici. l'azione è la seguente:

```
CREATE FULLTEXT CATALOG FullTextCatalog AS DEFAULT;
```

[15] [28] Hands on Full-Text Search in SQL Server, Jefferson Elias, 2017
[16] [3] ElasticSearch Architecture Overview, how Elasticsearch organizes data.

Questo è il caso di un catalog come da default, ma ci sono varie impostazioni dettagliate da poter utilizzare.

A questo punto si può creare l'indice voluto e per farlo ci servono alcune informazioni: qual è la chiave che vogliamo utilizzare per identificare univocamente il nostro record nell'indice? Quali colonne fanno parte dell'indice? Che tipo rappresenta la colonna e in quale colonna archiviare le informazioni? Quale lingua utilizzare nelle colonne dell'indice o nel caso, si preferisce essere neutrali per l'interpretazione linguistica? Lasciare il rilevamento delle modifiche auto, manual o off?

Vediamo un esempio:

```
CREATE FULLTEXT INDEX ON dbo.DM_OBJECT_FILE (
    FILE_TXT TYPE COLUMN OBJ_FILE_IDX_DOCTYPE LANGUAGE 0
) KEY INDEX PK_DM_OBJECT_FILE
WITH
    CHANGE_TRACKING = AUTO,
    STOPLIST=OFF
;
```

In questo caso si crea un indice sulla colonna FILE_TXT della tabella dbo.DM_OBJECT_FILE, con le informazioni inserite in OBJ_FILE_IDX_DOCTYPE, con interpretazione di linguaggio neutrale e l'update automatico e nessun stop list.

Possiamo, una volta creato l'indice, andare a modificare alcune sue impostazioni o operazioni attraverso il comando *ALTER FULLTEXT INDEX*. Possiamo per esempio abilitare o disabilitare l'indice; abilitare o disabilitare il rilevamento automatico delle modifiche con il suo aggiornamento (change tracking); aggiungere, modificare o editare le colonne che fanno parte del full-text index; controllare in modo specifico il popolamento dell'indice.

Una volta che il nostro indice è stato creato, possiamo compiere ricerche full-text usando funzioni che lo hanno incorporato nella loro struttura, come quelle descritte in seguito.

2.2.2 Query full-text in SQL Server

Attraverso le query full-text è possibile eseguire ricerche in linguaggio naturale sui dati di testo contenuti negli indici full-text, usando parole e frasi in base alle regole di una determinata lingua, come ad esempio l'inglese o il giapponese. Le query full-text possono contenere semplici parole e frasi oppure più forme di una parola o frase e restituiscono qualsiasi valore contenente almeno una corrispondenza. Si ottiene un matching quando un documento di destinazione contiene tutti i termini specificati nella query full-text e soddisfa qualsiasi altra condizione di ricerca, come ad esempio la distanza entro i termini corrispondenti.

Analizziamo ora quella che è la tipologia di query full text implementata da SQL Server.

Dopo l'aggiunta delle colonne a un indice full-text, gli utenti e le applicazioni possono eseguire query full-text sul testo contenuto all'interno delle colonne. Queste query possono consentire la ricerca degli elementi seguenti:

- Una o più parole o frasi specifiche (*termine semplice*)
- Parola o frase in cui le parole iniziano con il testo specificato (*termine di prefisso*)
- Forme flessive di una parola specifica (*termine di generazione*)
- Una parola o frase vicina a un'altra parola o frase (*termine di prossimità*)
- Sinonimi di una parola specifica (*thesaurus*)
- Parole o frasi che usano valori ponderati (*termine ponderato*)

Per le query di ricerca full-text non viene fatta distinzione tra maiuscole e minuscole.

Nelle query full-text viene utilizzato un set ridotto di predicati (CONTAINS e FREETEXT) e funzioni (CONTAINSTABLE e FREETEXTTABLE) Transact-SQL (una estensione proprietaria del linguaggio SQL classico, sviluppata da Microsoft e Sybase).

2.2.3 Predicati e funzioni

Per scrivere query full-text, usare i predicati **CONTAINS** e **FREETEXT** e le funzioni valutate a livello dei set di righe **CONTAINSTABLE** e **FREETEXTTABLE** con l'istruzione **SELECT**.

In seguito, vengono forniti alcuni esempi per comprendere meglio quali di questi sono i più adatti per quale utilizzo.

- Per trovare i valori corrispondenti a parole e frasi, usare **CONTAINS** e **CONTAINSTABLE**.
- Per trovare i valori corrispondenti al significato, ma non all'esatta formulazione delle parole, usare **FREETEXT** e **FREETEXTTABLE**.

I frammenti di codice seguenti usano il database di esempio *AdventureWorks2012*[17], trovabile nel sito ufficiale Microsoft qui sotto. Per eseguire le query, è necessario impostare anche la ricerca full-text sul nostro programma.

CONTAINS

L'esempio seguente trova tutti i prodotti con un prezzo di $80.99 che contengono la parola "Mountain":

[17]Sito Microsoft dove poter scaricare AdventureWorks2012: ttps://docs.microsoft.com/en-us/sql/samples/adventureworks-install-configure?view=sql-server-ver15&tabs=ssms

```
USE AdventureWorks2012
GO
SELECT Name, ListPrice
FROM Production.Product
WHERE ListPrice = 80.99
  AND CONTAINS (Name, 'Mountain')
GO
```

FREETEXT

L'esempio seguente cerca tutti i documenti che contengono parole correlate a vital safety components:

```
USE AdventureWorks2012
GO
SELECT Title
FROM Production.Document
WHERE FREETEXT (Document, 'vital safety components')
GO
```

CONTAINSTABLE

In questo caso vengono restituiti l'ID della descrizione e la descrizione di tutti i prodotti per i quali la colonna **Description** contiene la parola "aluminum" accanto alla parola "light" o "lightweight". Vengono restituite solo le righe con un valore di pertinenza (RANK, descritto in seguito) maggiore o uguale a 2.

```
USE AdventureWorks2012
GO
SELECT FT_TBL.ProductDescriptionID,
  FT_TBL.Description,
```

```
  KEY_TBL.RANK
FROM Production.ProductDescription AS FT_TBL INNER JOIN
  CONTAINSTABLE (Production.ProductDescription,
    Description,
    '(light NEAR aluminum) OR
    (lightweight NEAR aluminum)'
  ) AS KEY_TBL
  ON FT_TBL.ProductDescriptionID = KEY_TBL.[KEY]
WHERE KEY_TBL.RANK > 2
ORDER BY KEY_TBL.RANK DESC;
GO
```

FREETEXTTABLE

Nell'esempio seguente viene estesa una query FREETEXTTABLE in modo che vengano restituite per prime le righe con valore di pertinenza maggiore e che la classificazione di ogni riga venga aggiunta all'elenco di selezione. Per scrivere una query simile, è necessario sapere che *ProductDescriptionID* è la colonna chiave univoca per la tabella *ProductDescription*.

```
USE AdventureWorks2012
GO
SELECT KEY_TBL.RANK, FT_TBL.Description
FROM Production.ProductDescription AS FT_TBL
   INNER JOIN
   FREETEXTTABLE(Production.ProductDescription, Description,
            'perfect all-around bike') AS KEY_TBL
   ON FT_TBL.ProductDescriptionID = KEY_TBL.[KEY]
ORDER BY KEY_TBL.RANK DESC
GO
```

2.2.4 Quale tipologia di matching usare?

CONTAINS/CONTAINSTABLE e FREETEXT/FREETEXTTABLE sono utili per vari tipi di matching. Le informazioni seguenti consentono di scegliere il predicato o la funzione più idonea per la query:

CONTAINS/CONTAINSTABLE

- Trova i valori corrispondenti di parole e frasi singole con un livello di matching esatto o fuzzy (meno preciso).
- È anche possibile eseguire le operazioni seguenti:
    - Specificare la prossimità delle parole all'interno di una certa distanza l'una dall'altra.
    - Restituire corrispondenze ponderate.
    - Combinare le condizioni di ricerca con gli operatori logici. Come l'uso di operatori booleani (and, or, not).

FREETEXT/FREETEXTTABLE

- Trova i valori corrispondenti al significato, ma non all'esatta formulazione, di parole, frasi o periodi specificati (*stringa free-text*).
- Vengono generate corrispondenze se nell'indice full-text di una colonna specificata viene trovato un qualsiasi termine o formato di un qualsiasi termine.

Confrontare predicati e funzioni

I predicati CONTAINS/FREETEXT e le funzioni valutate a livello dei set di righe CONTAINSTABLE/FREETEXTTABLE hanno sintassi e opzioni diverse. Le informazioni seguenti consentono di scegliere il predicato o la funzione più idonea per la query:

Predicati CONTAINS e FREETEXT

Utilizzo. Usare i *predicati* full-text CONTAINS e FREETEXT nella clausola WHERE o HAVING di un'istruzione SELECT.

Risultati. I predicati CONTAINS e FREETEXT restituiscono un valore TRUE o FALSE che indica se una determinata riga corrisponde alla query full-text. Le righe corrispondenti vengono restituite nel set di risultati.

Altre opzioni. È possibile combinare i predicati con altri predicati Transact-SQL, ad esempio LIKE e BETWEEN.

È possibile specificare una sola colonna, un elenco di colonne o tutte le colonne della tabella in cui eseguire la ricerca.

Facoltativamente, è possibile specificare la lingua le cui risorse vengono usate dalla query full-text specificata per il word breaking e lo stemming, le ricerche nel thesaurus e la rimozione di parole non significative.

Funzioni valutate a livello dei set di righe CONTAINSTABLE e FREETEXTTABLE

Utilizzo. Usare le *funzioni* full-text CONTAINSTABLE e FREETEXTTABLE come un normale nome di tabella nella clausola FROM di un'istruzione SELECT.

È necessario specificare la tabella di base per la ricerca quando si usa una di queste funzioni. Come con i predicati, è possibile specificare una singola colonna, un elenco di colonne o tutte le colonne della tabella in cui eseguire la ricerca e, facoltativamente, la lingua le cui risorse vengono usate dalla query full-text specificata.

In genere, i risultati del predicato CONTAINSTABLE o FREETEXTTABLE devono essere uniti in join alla tabella di base. Per creare una join tra le tabelle, è necessario conoscere il nome della colonna chiave univoca.

Risultati. La tabella restituita contiene solo righe della tabella di base che corrispondono ai criteri di selezione specificati nella condizione della ricerca full-text della funzione.

Le query che usano una di queste funzioni restituiscono un valore di classificazione per pertinenza (RANK) e una chiave full-text (KEY) per ogni riga restituita, come illustrato di seguito:

- Colonna **KEY**. La colonna KEY restituisce valori univoci delle righe restituite. Può essere utilizzata per specificare i criteri di selezione.
- Colonna **RANK**. La colonna RANK restituisce un *valore di pertinenza* per ogni riga che indica il livello di corrispondenza tra la riga e i criteri di selezione. Maggiore è il valore di pertinenza del testo o del documento in una riga, maggiore sarà la pertinenza della riga per una determinata query full-text. Sono possibili molti modi per limitare le ricerche anche secondo RANK, si consiglia la lettura delle limitazioni su base di RANK su sito microsoft.

2.2.5 Quando conviene usare gli indici Full-Text

Abbiamo citato in precedenza l'operatore LIKE, che può essere utilizzato al di fuori dell'ambito della ricerca full-text per avere sostanzialmente lo stesso tipo di risultato. Allora perché utilizzare gli indici full-text invece che il semplice LIKE? Ad esempio, una delle interrogazioni che abbiamo mostrato in precedenza:

```
SELECT Autore, Titolo
FROM Libri
WHERE CONTAINS(Riassunto, '"innamor*"');
```

Può essere riscritta con l'operatore LIKE nel modo seguente:

SELECT Autore, Titolo

FROM Libri

WHERE Riassunto LIKE '%innamor%'

La differenza sta nel tempo di esecuzione: la ricerca in una colonna di tipo VARCHAR con l'operatore LIKE può richiedere parecchio tempo. È più efficiente utilizzare le funzionalità di Full-Text Search di SQL Server, che si comportano decisamente meglio perché utilizzano un indice costruito proprio sui campi su cui si effettua la ricerca.

Il vantaggio in termini di prestazioni è visibile soprattutto quando si eseguono query su grandi quantità di dati non strutturati (cioè conservati senza alcuno schema o tipologia precisa): una query che fa uso dell'operatore LIKE eseguita su milioni di righe può richiedere alcuni minuti, mentre per un'interrogazione Full-Text sugli stessi dati possono essere necessari solo pochi secondi.

Bisogna comunque evidenziare che la manutenzione del catalogo ha un costo non trascurabile, soprattutto se si modificano spesso i dati contenuti nelle colonne indicizzate. Nel caso in cui le informazioni siano prevalentemente statiche (come i riassunti di libri del nostro esempio), l'utilizzo di indici Full-Text sui campi testuali è sicuramente consigliabile.

Contrariamente alla ricerca full-text, il predicato LIKE di Transact-SQL funziona unicamente con i modelli di caratteri. Non è inoltre possibile utilizzare il predicato LIKE per eseguire query su dati binari formattati. C'è quindi molta limitazione data da LIKE che può essere risolta con gli indici full-text (furono creati proprio perché LIKE non era prestante).

Considerazioni riportate da html.it, "Il Full-Text Search di SQL Server"[18].

[18] [40] Il Full-Text Search di SQL Server, Marco Minerva, 2008, Il Full-Text Search di SQL Server | HTML.it

3. Nuovi DBMS NOSQL

Abbiamo visto come avvengono le ricerche in SQL Server e le potenzialità che queste possiedono. Malgrado ciò negli anni è nata la necessità di plasmare la realtà con sistemi più flessibili rispetto al modello relazionale e di fornire tecnologie di gestione con caratteristiche diverse. Il primo problema che fu riscontrato è che il modello relazionale era stato concepito per utilizzare, elaborare e amministrare dati dalla struttura omogenea, lasciando poco spazio alla flessibilità di memorizzazione e schematizzazione per avere una consistenza e una capacità di interrogazione più semplice. Il panorama però, con la nascita del web, è notevolmente cambiato e la quantità di dati da trasmettere, scambiarsi, analizzare è aumentato a dismisura. Non solo, sorgenti differenti, da luoghi diversi, condividono immagini, video, e-mail, articoli di giornali, pdf, segnali e molto altro.

Le persone condividono anche i loro gusti, preferenze, passioni attraverso continui feedback e informazioni raccolte da applicazioni e siti web, disposti a pagare ingenti somme di denaro per averle. Esaminare questi dati in tempo reale e capire quello che un utente desidera porta ad annunci mirati e ad attuare un piano di mercato il più competitivo possibile. Come organizzare e progettare una ricerca specifica, full-text, in DBMS così rigidi e strutturati? Ricordiamo che sono ancora diffusissimi e sono tutt'ora i DBMS più utilizzati.

Compresa l'importanza che l'informazione ha al giorno d'oggi è facile capire che la mole di dati da trattare è enorme e la loro complessità aumenta sempre più. La scalabilità diventa l'unico modo per gestire i dati in modo efficiente, lo scaling verticale, cioè acquistare macchine sempre più potenti e con memorie maggiori non è più un'opzione valida, meglio puntare sullo scaling orizzontale. La

creazione di cluster è più economica e aumenta anche la resilienza, riuscendo a gestire un guasto attraverso la replicazione dei dati su più macchine del cluster.

I sistemi basati su modelli relazionali non sono stati concepiti per questo scopo, anche se si sono trovate soluzioni basate sul Shared Disk Subsystem (ma permane il single point of failure) o porzioni diverse di dati su server separati, le applicazioni devono comunque sapere su quale server operare e magari le query su server differenti hanno risultati errati o con tempistiche diverse. Un'allocazione diversa dei dati comporterebbe una riprogrammazione delle applicazioni stesse. Troppo complesso e oneroso questo tipo di approccio, soprattutto se sul mercato nel tempo si sono affacciate applicazioni che sono state create per soddisfare questo genere di richiesta.

Serviva un modello dinamico che si adattasse alla dinamicità della rete, veloce, che permettesse di interagire con molti dati in real-time in qualsiasi luogo e momento e che potesse affiancarsi ai numerosissimi DBMS relazionali come applicazione di supporto. Aumentare le performance, la suddivisione del carico di lavoro e meno rigidità di gestione furono le sfide prese in carico dagli sviluppatori nel primo decennio degli anni 2000. Nasce proprio per questo motivo il mondo NoSQL con molte nuove tecnologie e prodotti software legati principalmente ai grandi players di Internet: Amazon, Google, nonché ai social networks (Facebook, Instagram, Linkedin, etc.). Un DBMS NOSQL sviluppato per la gestione di documenti che è ormai diffusissimo è *MongoDB*[19].

Ci stiamo rendendo conto che la velocità con cui le tecnologie consolidate vengono rimpiazzate è velocissima. Apparentemente da un giorno all'altro, tecnologie affermate sono messe in discussione da un repentino cambiamento di attenzione da parte dei programmatori. Il fenomeno è evidente con le tecnologie

[19] [13] MongoDB, https://www.mongodb.com/. Approfondito in Appendice.

NoSQL (Not only SQL) a scapito dei ben consolidati database relazionali. Dal web guidato da pochi RDBMS, in qualche anno quattro o cinque soluzioni NoSQL si erano già affermate come attendibili alternative. Anche se sembra che queste transizioni avvengano nel corso di una notte, la realtà è che possono passare anni prima che una nuova tecnologia divenga pratica comune e, in certi casi, il passaggio è dubbio e può non verificarsi mai. L'entusiasmo iniziale è guidato da un gruppo relativamente piccolo di sviluppatori e aziende. I prodotti migliorano con l'esperienza e, quando ci si rende conto che una tecnologia è destinata a rimanere, altri cominciano a sperimentarla. Ciò è particolarmente vero nel caso NoSQL poiché spesso queste soluzioni non vengono progettate come alternative a modelli di storage più tradizionali, ma intendono piuttosto far fronte a nuove necessità, come quelle descritte nel capitolo precedente. NoSQL è la convinzione che lo strato di persistenza non è necessariamente responsabilità di un solo sistema. Laddove storicamente i fornitori di database relazionali hanno sempre tentato di posizionare i loro software come soluzione universale per qualunque problema, NoSQL tende a individuare piccole unità di responsabilità per ognuna delle quali scegliere lo strumento ideale. In parole povere NoSQL entra in un mondo dove esistono modelli e strumenti alternativi per la gestione dei dati. Questa nuova tecnologia nasce quindi con lo scopo di essere affiancata ai più comuni RDBMS e non come antitesi.

Le peculiarità dei sistemi NoSQL è che sono solitamente schemaless, permettendo l'aggiunta di nuovi campi o la loro modifica e rimozione senza che ci siano cambiamenti alla struttura del database. Si può intuire come la gestione di dati estremamente diversi tra loro sia molto facilitata e quindi con maggiori prestazioni grazie alla linear scalability, cioè con l'aggiunta di tanti server nel cluster, distribuendo dati e lavoro. Con la diffusione del web odierno diventa fondamentale la velocità di lettura e scrittura con un susseguirsi di operazioni facili, ma che richiedono uno scambio di dati massiccio, che trova riscontro su

questo modello che permette un aumento di letture e scritture suddivise tra più server.

Sembra che si vanno a perdere con questa concezione di database quelle che sono le proprietà fondamentali nel modello relazionale: dette *ACID*[20], cioè Atomicità (una transazione è una unità logica e quindi tutte le operazioni riguardanti questa transazione devono andare a buon fine o altrimenti non avviene nessuna modifica), Consistenza (dopo ogni transazione il database deve essere consistente), Isolamento (se più transazioni avvengono in modo concorrente, queste devono avvenire sequenzialmente e in modo indipendente) e Durabilità (quando una modifica viene confermata questa deve essere persistente). Queste proprietà con l'avvento dei sistemi NoSQL furono messe in discussione perché puntavano tutto sulla disponibilità dei dati (*availability*) e che questi potessero essere partizionati in più nodi del cluster con la conseguente resilienza (attraverso anche la *replicazione* dei dati in più server) e aumento delle performance e un continuo funzionamento (dovuto alla partition *tolerance*). La consistenza dei dati, come tutte le proprietà ACID, sono state considerate imprescindibili e vitali da sempre e i database di tipo NoSQL si sono sviluppati in questo senso, garantendole come nei RDBMS (assicurate da questi anche dal fatto che il dataset è comunemente mantenuto sullo stesso server). Per questo è importante ribadire come affiancare spesso i due sistemi in determinati campi può essere una scelta vincente. Un'altra importante innovazione dei prodotti NoSQL è che solitamente non prevedono operazioni di join, fondamentali nei sistemi relazionali, ma onerose e dispendiose dal punto di vista prestazionale.

La rappresentazione dei dati, come vedremo nello specifico, in seguito, avviene raccogliendo insieme dati correlati tra loro e memorizzati in determinati campi

[20] [34] Cosa si intende per ACID nei sistemi di database?, 2016

(rappresentazione denormalizzata). Nel modello relazionale l'informazione deve essere unica, atomica, con i valori memorizzati in determinati attributi dei record.

Le idee e i prodotti che nascono sono funzionali e ben strutturati, ad esempio MongoDB, valida alternativa per un database relazionale, ma che non hanno ancora sostituito i prodotti software ben funzionanti e stabili basati sul modello SQL.

La concezione NoSQL ha gettato le basi per la nascita di quello che è Elasticsearch, un motore di ricerca e non un vero e proprio database. Esso, infatti, garantisce un ottimo supporto per la ricerca full-text a quelli che sono i sistemi e modelli di DBMS presenti già sul mercato. Dopo uno sguardo ampio di quelle che sono le sue funzionalità principali di Lucene vedremo come esso opera nella ricerca di testo e come viene sfruttato da Elasticsearch per capire come esso è divenuto così popolare.

4. Apache Lucene

Elasticsearch non ha inventato nuove tecnologie ma si è basato e adattato su prodotti e concetti ben consolidati e funzionanti. Il più importante tra questi è *Apache Lucene*[21], ideato da *Apache Software Foundation*, che si descrive così: "Apache Software Foundation fornisce supporto alla comunità Apache di progetti software open source. I progetti Apache sono definiti da processi collaborativi basati sul consenso, una licenza software aperta e pragmatica e il desiderio di creare software di alta qualità che sia alla guida del suo campo". Apache Lucene è progetto che sviluppa un software di ricerca open source che

[21] [33] Welcome to Apache Lucene, https://lucene.apache.org/index.html

rilascia una libreria di ricerca di base, denominata Lucene Core. Fornisce potenti funzionalità di indicizzazione e ricerca, oltre che al controllo ortografico, hit highlighting e funzionalità di analisi e tokenizzazione. Permette di creare motori di ricerca full-text ad alte prestazioni ed è scritta interamente in Java (sono disponibili implementazioni che permettono altri linguaggi di programmazione per la indicizzazione). Si adatta a quasi tutte le applicazioni che richiedono la ricerca full-text, soprattutto multipiattaforma.

Apache Lucene offre potenti funzionalità attraverso una semplice API. Ha una indicizzazione scalabile e ad alte prestazioni ed è questo che ha reso Elasticsearch così performante:

- con un hardware moderno permette oltre i 150 GB/ora
- utilizza poca RAM e solo 1 MB di heap
- la dimensione degli indici che è circa il 20-30% della dimensione del testo che viene indicizzato

Le caratteristiche appena citate non sono banali, hanno inciso sul successo di Elasticsearch ai danni di altri software e database come MongoDB, la cui ricerca di testo proprietaria occupava molta più memoria RAM e una dimensione di dati maggiore (vedi Appendice).

Apache Lucene offre anche moltissimi algoritmi di ricerca accurati ed efficienti, ad esempio:

- ranked searching, il miglior risultato viene restituito per primo;
- molti tipi di query potenti: query per frase, query con caratteri jolly, query di prossimità, query di intervallo e altro ancora;
- ricerca sul campo (field searching), ad es. titolo, autore, contenuto;
- sorting in base a qualsiasi campo;

- ricerca su più indici con risultati uniti (operazione similare alla join, punto critico di una ricerca con modello relazionale);
- faceting flessibile, highlighting, joins and grouping dei risultati;
- possibilità di suggerimenti veloci efficienti (in termini di memoria) e tolleranti agli errori di battitura
- e molto altro

Apache Lucene ha reso Elasticsearch un ottimo prodotto (come molti altri motori di ricerca, vedi *Solr*[22] di Apache) e insieme alle caratteristiche in seguito riportate è diventato una delle opzioni più scelte degli ultimi anni.

Lucene presenta un vasto insieme di interfacce che ci definiscono tutti i classici passaggi della ricerca. Non si tratta di un vero motore di ricerca che noi chiamiamo ogni volta chiedendogli di cercare qualcosa e lui lo fa in automatico. Il fatto che sia "semplicemente" una libreria permette molta flessibilità delle API che vengono messe a disposizione da Lucene.

Una base di dati testuale Lucene viene chiamata indice (*index*), che corrisponde al corrispettivo di una tabella in un database relazionale e le righe sono i documenti da noi inseriti. L'indice viene conservato solitamente in una cartella del filesystem, ma esistono anche altre possibilità, come la creazione di indici in RAM, usato per aumentare le prestazioni (ma con un limite imposto dalla dimensione della memoria). All'interno dell'indice vengono inseriti i documenti (istanze di *Document*), a loro volta divisi in campi (*Fields*), che corrispondono alle colonne nei database relazionali. Alcuni campi verranno utilizzati per l'indicizzazione mentre altri, come per esempio l'*id* del documento (*identificatore*) saranno solo salvati all'interno del database. Non si potrà eseguire quindi la ricerca su tutti i campi ma in certi casi si potrà solo accedere ad essi grazie al documento ritornato dalla query di ricerca eseguita però su un altro campo. Lo

[22] Apache Sorl: https://lucene.apache.org/solr/

scenario di utilizzo dei motori di ricerca prevede nella quasi totalità dei casi l'affiancamento della base di dati full-text ad una relazionale e questa tecnologia lo permette senza grosse problematiche. Si utilizzerà Lucene per fornire un sistema di ricerca, con i relativi risultati e quando l'utente vorrà visualizzare l'intero documento analizzato si accederà alla specifica pagina che sarà popolata invece da database spesso relazionali.

L'inserimento di un documento all'interno dell'indice comporta una scansione del testo, per trovare tutte le parole presenti e questa operazione prende il nome di "analisi". Lucene fornisce analizzatori basati sulle lingue più diffuse e una volta che l'indicizzazione è avvenuta si può procedere con la ricerca. Il tutto è ottimizzato da Lucene con risultati ottimi e prestazioni elevate senza che le queste operazioni interferiscano con le ricerche, anche multiple.

La ricerca può avvenire direttamente tramite le API di ricerca oppure utilizzando uno specifico linguaggio di interrogazione. Nel primo caso, utilizzato anche tramite Elasticsearch si possono eseguire query molto complesse. Si usano termini, che possono essere singoli, multipli o frasi, uniti insieme ad operatori e impostazioni di ricerca particolari. Molto utilizzati sono gli operatori booleani (AND, OR, NOT) e i Wildcard (?, *, ~, ^). Con quest'ultimi si possono compiere ricerche con caratteri jolly, ricerche Fuzzy, il boosting di un termine e molto altro[23].

Proprietà molto importante implementata da Lucene è lo *scoring*. Utilizza una combinazione del *Vector Space Model*[24] (VSM) dell'*Information Retrieval* e del

[23] Tutte queste funzionalità sono spiegate nel dettaglio sul sito: [35] Apache Lucene, https://www.appuntisoftware.it/apache-lucene/

[24] Dettagli su VSM su questo articolo di Kurtis Pykes, 2020: https://towardsdatascience.com/vector-space-models-48b42a15d86d

Boolean model[25] , per determinare quanto un documento sia rilevante rispetto ad una query dell'utente. La logica di VSM è che se un termine compare più volte in un documento rispetto a tutti gli altri documenti della collezione, allora tale documento sarà più rilevante. Viene usato il Boolean Model per scansionare quei documenti che devono essere sottoposti allo scoring su base logica booleana di una specifica query. Lucene utilizza ulteriori modelli e livelli di precisione per varie tipologie di ricerca ma il concetto è sempre il medesimo. Diventa di fondamentale importanza il come i documenti vengono indicizzati, di come sono stati salvati e il contenuto dei campi con la loro propria semantica, anche perché i risultati della ricerca e quindi dello scoring sono sempre i documenti. Lo scoring lavora sui *Fields*, combina i risultati e ritorna i *Documents* (documenti veri e propri). Lucene permette anche di modificare lo score attraverso incrementi di importanza su determinati concetti, il *boosting*, che può essere a livello di documento (si indica la sua rilevanza prima che venga aggiunto a un indice), a livello di campo (si indica la sua rilevanza al momento della creazione del documento), a livello di query (mentre si effettua una ricerca si dà più importanza a una clausola rispetto ad un'altra).

Abbiamo detto che Lucene fornisce la possibilità di implementare motori di ricerca full-text mediante l'implementazione di meccanismi di indicizzazione e di ricerca. Le API facilitano tutti questi compiti, che unite a prodotti software più complessi rendono l'esperienza utente semplice, veloce e intuiva. È il caso di Elasticsearch, basato proprio su Apache Lucene, che descriveremo nel dettaglio nel capitolo seguente.

[25] Maggiori informazioni sul modello booleano al sito: https://mathworld.wolfram.com/BooleanModel.html#:~:text=In%20most%20modern%20literature%2C%20a%20Boolean%20model%20is,the%20model%20is%20said%20to%20be%20driven%20by

5. Elasticsearch e full-text

5.1 Cos'è Elasticsearch?

Elasticsearch è un motore di ricerca e di analisi distribuito, memorizza grandi quantità di dati in formati sofisticati ottimizzati per una ricerca basata sul linguaggio. Nasce prima come Compass e poi evoluto con la libreria di Apache Lucene diventando Elasticsearch. Creato nel 2004 da *Shay Banon*[26] e reso disponibile gratuitamente (open source) agli altri utenti. Verrà successivamente arricchito di funzionalità e prodotti come Logstash, Beats e Kibana, che insieme a Elasticsearch formano l'Elastic stack. Intorno al 2015 l'azienda ha iniziato ad offrire una soluzione a stack integrato tramite Amazon Web Services, ma ora offre un proprio sistema basato su cloud: Elastic Cloud Enterprise. La compagnia di Banon e i suoi collaboratori oggi prende il nome di Elastic e dal 2018 è diventata una società per azioni quotata alla Borsa di New York. Il grafico rappresentato nella figura 5.1 ci mostra il successo e un aumento formidabile del ranking che ha avuto dal 2013 al 2017 (undicesimo nel ranking mondiale come database, pur non essendo nato per quello). Attualmente si trova al settimo posto secondo db-engine.com e molte aziende e società stanno cominciando a migrare verso questo motore di ricerca.

[26] [4] Our story, https://www.elastic.co/about/history-of-elasticsearch

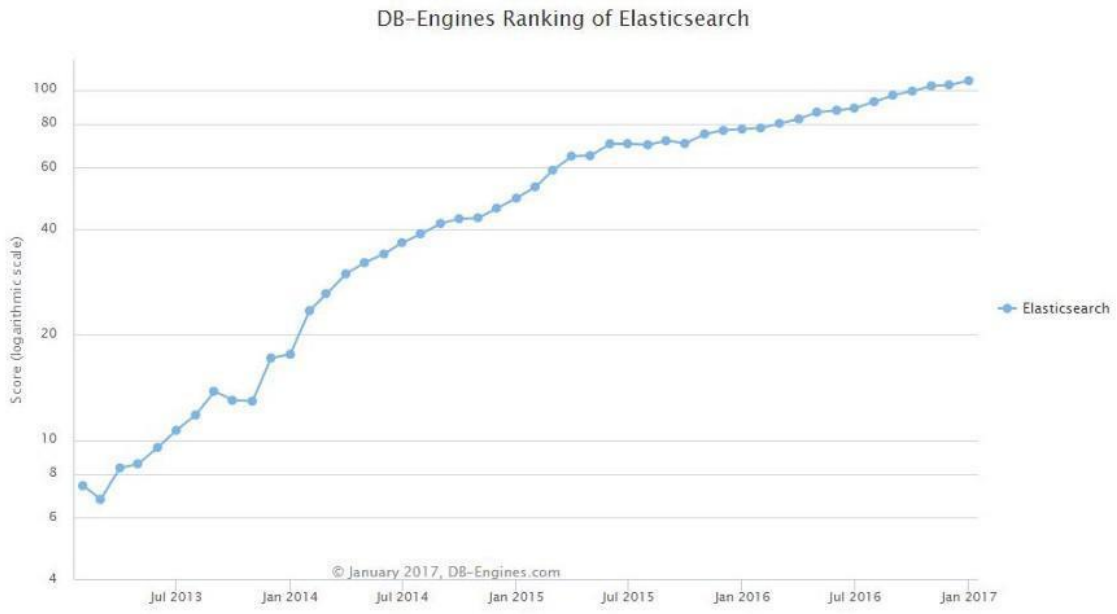

Figura 5.1 *Elasticsearch: How to Add Full-Text Search to Your Database* [20]

Prendendo in considerazione il grafico appena illustrato, aggiungiamo il ranking degli altri motori di ricerca più diffusi e notiamo come sia diventato il migliore e al giorno d'oggi rimane il re indiscusso della ricerca di testo.

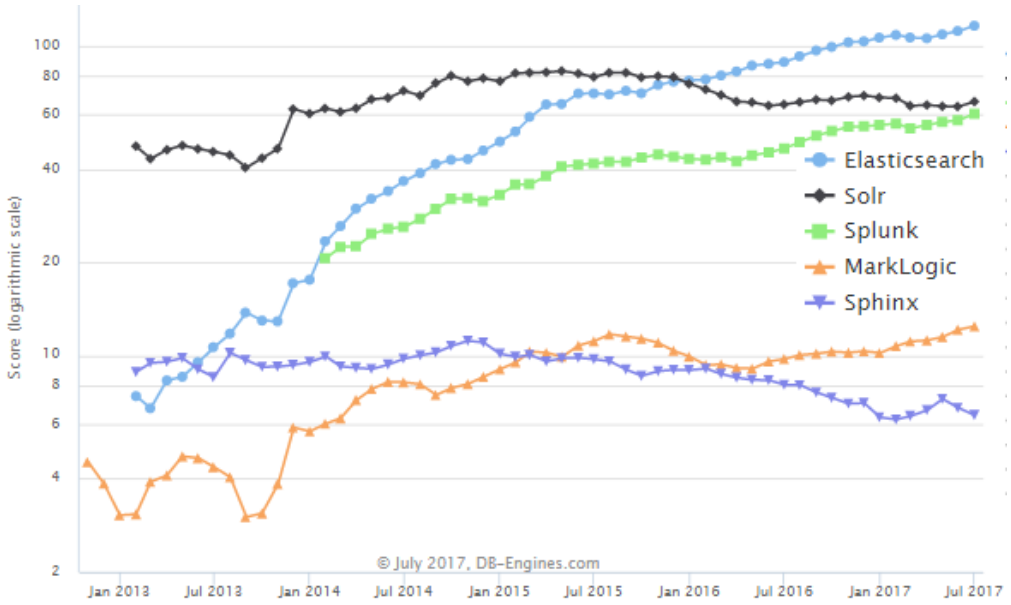

Figura 5.2 *Elasticsearch è oggi il più diffuso motore di ricerca aziendale* [26]

È uno strumento recente, ma potente, ottimizzato proprio per la full-text search, con molteplici algoritmi di ricerca e indicizzazione. Ha una tecnologia che lo rende un sistema parallelo e scalabile.

Elasticsearch riesce con tempistiche vicine al real-time a cercare e analizzare ogni tipo di dato: testo strutturato, non strutturato, dati numerici, geospaziali. Riesce a memorizzarli e a indicizzarli efficacemente in modo che venga supportata una ricerca veloce e dalle prestazioni costanti. Se col tempo la mole di dati aumenta, Elasticsearch sarà sempre in grado di adattarsi perfettamente e a mantenere sempre delle alte performance, grazie alla sua natura distribuita.

In questo capitolo analizzeremo i punti chiave che caratterizzano questo motore di ricerca per capirne i punti di forza e perché sta diventando la prima scelta in questo settore dell'informatica.

5.2 Dati in documenti e indici

Elasticsearch è un archivio distribuito di documenti. Memorizza le informazioni in strutture dati complesse, serializzate come documenti JSON. Queste strutture dati vengono chiamate a inverted index (indice invertito) grazie alla quale viene listata ogni parola unica che compare in un documento e identifica tutti i documenti in cui ricorre questa parola.

Un indice può essere considerato come una raccolta ottimizzata di documenti e ogni documento è una raccolta di campi, che sono coppie chiave-valore che contengono i dati. Elasticsearch si affida completamente ad Apache Lucene per la sua struttura dell'indice. Un indice invertito, in Lucene, contiene diversi dati: un dizionario di termini, dati sulla frequenza del termine, informazioni ausiliarie per un dato documento, fattori di normalizzazione (per tenere conto della lunghezza del documento), ecc.

Questi campi e dati dell'indice vengono memorizzati in più file: file di segmenti, file di informazioni sui campi o di testo, file di frequenza o di posizione. La capacità di utilizzare strutture dati basate sui campi lo rende molto veloce.

Elasticsearch è anche schema-less, quindi i documenti possono essere indicizzati senza sapere poi come effettivamente saranno gestiti. Il mapping dinamico, se abilitato, troverà e aggiungerà in automatico nuovi campi agli indici esistenti. Questa funzionalità potrebbe fornire risultati totalmente errati e per questo si possono settare regole di controllo del mapping dinamico, che lo definiscono esplicitamente per aver maggior controllo sulla indicizzazione e archiviazione, ad esempio: distinguere un campo stringa per full-text search o per valore esatto, una lingua specifica, per matching parziali, usare formati dei dati particolari, ecc.

5.3 Search e Analisi

Elasticsearch comunica utilizzando le REST API, ovvero REpresentational State Transfer, che è un particolare approccio architetturale che sfrutta il protocollo HTTP per il trasferimento di rete, rendendo di fatto standard l'interfaccia di comunicazione. Insieme ad HTTP troviamo JSON (JavaScript Object Notation), che serve per memorizzare le informazioni in modo organizzato, garantendo la semplice lettura e accesso alle collection di dati. Grazie a questi strumenti sono supportate query strutturate, full-text, alcune molto complesse e l'unione di queste, permette anche di compiere aggregazioni complesse per ottenere informazioni su metriche particolari, modelli e trend. Queste aggregazioni operano a fianco delle richieste di ricerca per filtrare risultati e documenti ed eseguire analisi mentre compiamo una richiesta. Con l'uso delle funzioni di machine learning si possono anche automatizzare i processi di analisi su una

serie di dati, molto utile per identificare anomalie, comportamenti inusuali o rari valori statistici[27].

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.

5.4 Scalabilità, affidabilità e resilienza

L'unità base di archiviazione dei dati in Elasticsearch sono i frammenti (*shard*) e ognuno di essi risiede su un nodo del sistema. I frammenti contengono istanze di Apache Lucene e hanno due ruoli principali: primario (Primary *shard*) che è la copia principale dei dati, orchestra la lettura e la scrittura e coordina le repliche; replica di frammento (*shard replica*) che è una copia di un frammento primario ed esiste per consentire la ridondanza. Capita che solo alcune repliche siano aggiornate con il primario, queste vengono chiamate repliche sincronizzate (*in-sync replicas*). Un gruppo di frammenti che trattano gli stessi dati è chiamato gruppo di replica (*replication group*).

Esistono anche vari tipi di nodi all'interno di un cluster Elasticsearch e servono a diversi scopi: il più importante è il *Master node*, che contiene i metadati dell'indice per il cluster e i dati di configurazione del cluster. Alcuni nodi possono essere idonei per il Master (*Master-eligible node*), che possono cioè essere eletti come master node di un cluster. Un nodo così diventa responsabile delle azioni sul cluster (es. creazione o eliminazione di un indice, allocazione di frammenti a nodi) e in generale non funge da archiviazione dati, al contrario del voting-only master-eligible node che può anche memorizzare i dati.

[27] [2] An Overview on Elasticsearch and its usage, Giovanni Pagano Dritto, 2019

Il data node contiene i frammenti ed è responsabile della gestione delle operazioni *CRUD* (create, read o retrieve, update, delete o destroy), ricerca e aggregazione dei dati. Contiene anch'esso i metadati di indice per i frammenti e i dati di configurazione. Ci sono molte altre tipologie di nodi nel cluster[28].

Figura 5.3 *ElasticSearch architecture Overview, how Elasticsearch organizes data* [3]

Il nodo master controlla periodicamente gli altri nodi del cluster con una comunicazione intra-cluster. Se viene rilevato un errore persistente del nodo, questo viene rimosso dal cluster e ci si adeguerà di conseguenza. Una politica chiamata consapevolezza dell'allocazione (*allocation awareness*[29]) può essere utilizzata per etichettare i nodi con la loro posizione fisica, consentendo al master di tenere conto delle informazioni ed evitare la perdita dei dati di quel nodo in caso di errore (si archiviano repliche dei frammenti primari in altre località, vedi figura 5.3).

Elasticsearch nasce come sistema di ricerca parallelo, la sua capacità può aumentare aggiungendo nodi al cluster e abbiamo visto come autonomamente

[28] [3] ElasticSearch Architecture Overview, how Elasticsearch organizes data.
[29] Cluster-level shard allocation and routing settings: https://www.elastic.co/guide/en/elasticsearch/reference/master/modules-cluster.html#shard-allocation-awareness

gestisca l'affidabilità facendo migrare i dati o replica di essi al momento di un malfunzionamento.

5.5 Lettura e scrittura

Alla ricezione di una richiesta di scrittura, un nodo diventa responsabile della gestione di questa richiesta[30]:

- inoltra tale richiesta ai nodi che contengono frammenti pertinenti
- Raccoglie le risposte dei nodi
- Risponde all'utente

Procedura in breve:

1. Inoltrare la richiesta di lettura agli indici giusti, che possono essere molteplici dato che i dati spesso sono divisi tra vari frammenti avviene utilizzando le funzioni di hash se l'id del documento è noto o eseguendo una ricerca round robin tra i frammenti del cluster.
2. Selezionare una copia attiva di ogni frammento rilevante dai gruppi di replica e di default Elasticsearch esegue il round robin tra le copie
3. Invio delle richieste a livello di frammento alle copie selezionate
4. Combina i risultati se necessario e risponde

Quando viene identificato un gruppo di replica e la richiesta viene inoltrata al suo primary shard, questo:

- Convalida la richiesta;
- Esegue le operazioni in locale e controlla l'infrastruttura sottostante;
- Inoltra le modifiche alle repliche sincronizzate in parallelo

[30] Procedura descritta da "An introduction to Elasticsearch" di Enrico Ghidoni, Big Data Management & Governance, a.y. 2019/2020

- Le repliche eseguono operazioni e confermano il riconoscimento al frammento principale
- Il frammento principale riconosce il successo al client richiedente

Nel caso di errori della scrittura delle operazioni:

- Se il frammento principale non riesce:
  - Il nodo su cui risiede invia un messaggio al nodo master del gruppo di replica
  - Il nodo principale mette la richiesta in attesa, designa un nuovo frammento primario e inoltra la richiesta per essere nuovamente elaborata
- Se una replica del frammento non riesce:
  - Lo shard primario chiede al nodo master del cluster di rimuoverlo dal file del sottoinsieme sincronizzato
  - Una volta che il master riconosce la rimozione, l'operazione è completa
  - Successivamente, il nodo master designa un altro nodo per iniziare a creare un frammento replica al posto di quello rimosso
- Se il frammento principale viene isolato:
  - Viene eletto un nuovo frammento primario
  - Le shard replicas rifiuteranno le richieste di commit dell'operazione dal vecchio primary shard
  - Il primo primary contatta il nodo master per richiedere il nuovo primary

5.6 Elastic Stack

Sebbene il motore di ricerca sia al centro della concezione di Elasticsearch, gli utenti che iniziavano a interagire con questo nuovo prodotto avevano bisogno di poter facilmente visualizzare e controllare i dati che archiviavano. Per questo motivo insieme ad Elasticsearch vengono progettati e distribuiti anche Logstash e Kibana (che sono i principali componenti, ne sono presenti altri[31]), che insieme formano la cosiddetta "*ELK*"[32].

KIBANA

Kibana è un software che permette di visualizzare i propri dati e navigare l'elastic stack. Dà forma ai dati con una interattiva interfaccia utente che permette di compiere domande sui dati e decidere come rappresentarli, dai classici grafici a visualizzazioni più complesse come mappe per i geodati. Può visualizzare anche i dati rilevati tramite machine learning, trovare relazioni tra le informazioni e compiere analisi profonde e complesse.

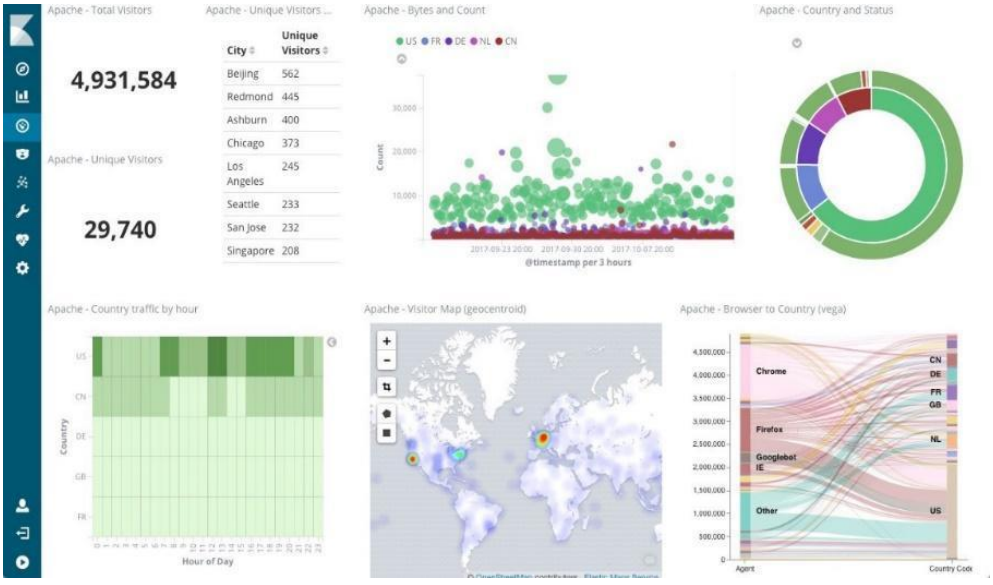

Figura 5.4 "Kibana console", *An Overview on Elasticsearch and its usage* [2]

[31] Consultabili al sito ufficiale di Elastic: https://www.elastic.co/elastic-stack

[32] [2] An Overview on Elasticsearch and its usage, Giovanni Pagano Dritto, 2019

LOGSTASH

Logstash è una pipeline di elaborazione dati lato server, open source che acquisisce dati da varie origini contemporaneamente, li trasforma e li invia per poterli poi raccogliere.

Sembra un'operazione banale, ma in realtà i dati sono spesso sparsi tra vari sistemi e in molti formati. Su Logstash si può importare log, metriche, applicazioni web, archivi, ecc. con uno streaming continuo. Trasforma questi dati, li prepara dinamicamente indipendentemente dal formato che essi hanno, costruisce i campi per una struttura e formato comune. Permette di monitorare le varie operazioni di Logstash come gli eventi ricevuti o inviati, come viene utilizzata la memoria, i tempi di attività, compiere statistiche e controllare lo stato dei tuoi nodi.

Figura 5.5 “Logstash node stats”, *An Overview on Elasticsearch and its usage* [2]

5.7 Indici e ricerca su Elasticsearch

5.7.1 Funzionalità principali di Elasticsearch e Kibana

Vediamo ora come avviene una ricerca full-text su Elasticsearch[33]. Ci avvaliamo dell'uso di Kibana combinato a Elasticsearch per poter gestire più semplicemente i nostri dati. Le stesse operazioni però si potranno effettuare senza Kibana tramite linea di comando nel nostro computer. In Kibana, come in un qualsiasi altro client, si possono utilizzare i metodi HTTP PUT, GET, POST e DELETE per inviare comandi alla vostra ricerca full text.

Come prima cosa bisogna creare un indice, che alimentiamo con i nostri dati. Si possono usare due metodi HTTP, si impiega PUT se si desidera specificare un particolare ID per la voce in esame, con POST Elasticsearch crea da sé un ID.

Per esempio, se noi volessimo creare una libreria dove ogni voce debba contenere il nome dell'autore, il titolo del libro e l'anno di pubblicazione scriveremmo:

Figura 5.6

Elasticsearch dovrebbe restituire le informazioni di conferma, nel caso sia stato inserito tutto correttamente, attraverso questo messaggio:

[33] Funzionalità descritte anche su [19] Elasticsearch: il motore di ricerca flessibile, 2019, prestare però attenzione che la modalità di ricerca è stata aggiornata.

Figura 5.7

Elasticsearch avrà inserito un indice con il nome *libri* e il tipo *_doc*. Abbiamo usato il metodo POST e di conseguenza il tool di Kibana ha generato in automatico un ID univoco per il nostro libro. Tutto ciò è confermato dalla voce *result* uguale a *created*. Usando PUT diamo al nostro libro un ID specifico, lo si ottiene mettendolo nella prima riga del codice. Se si vuole modificare una versione esistente è necessario usare sempre PUT. Inseriamo un altro libro nei nostri dati:

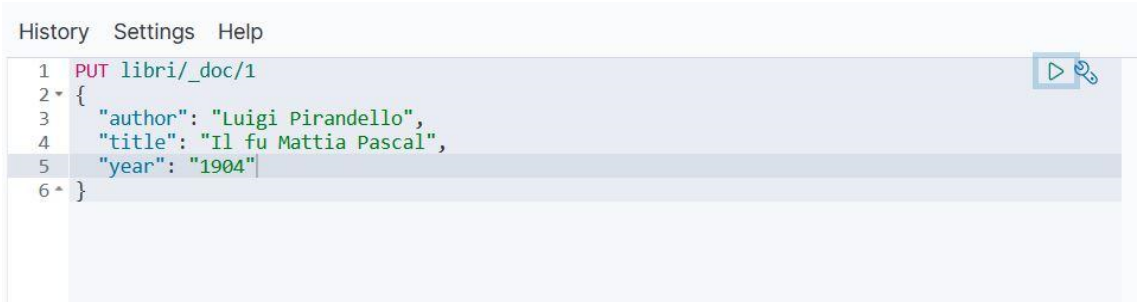

Figura 5.8

L'output che otteniamo ci indica l'ID che abbiamo fornito nella prima riga. L'ordina è sempre questo: *_index*/*_type*/*_id*.

Figura 5.9

Il PUT permette anche di modificare le voci, basta inserire l'_id da voler modificare. Il libro sarà modificato e la voce *version* sarà incrementata di 1 e il *result* sarà *updated*. Elasticsearch va a sostituire solo quelle che sono le voci e nel caso si confondessero gli _id, per evitare modifiche accidentali, se si vuole creare un nuovo libro basta inserire l'endpoint *_create*. Ad esempio: *libri/_doc/1/_create* o nella versione recente però c'è la rimozione del campo type e al suo posto possiamo scrivere l'endpoint; di conseguenza il codice sarà il seguente: *libri/_create /1*. In questo caso verrà segnalato un errore perché l'id è già presente.

Per modificare invece un libro, per non sbagliare, si potrà usare l'endpoint *_update*. Quindi: *libri/_doc/1/_update*, o con la versione recente è *libri/_update/1*. Vediamo questo esempio:

Figura 5.10

Abbiamo aggiunto in questo caso il campo *genre*. Sembra che sia stato solo aggiornato il nostro libro, ma in realtà Elasticsearch ha ricreato l'intero libro inserendo autonomamente i contenuti già esistenti. Questa tipologia di scrittura si basa su Lucene ma è dispendiosa a livello di prestazioni.

Una volta inserito il nostro dato lo possiamo richiamare attraverso il metodo GET:

GET libri/_doc/1

Elasticsearch ci mostrerà in output i dettagli del nostro documento:

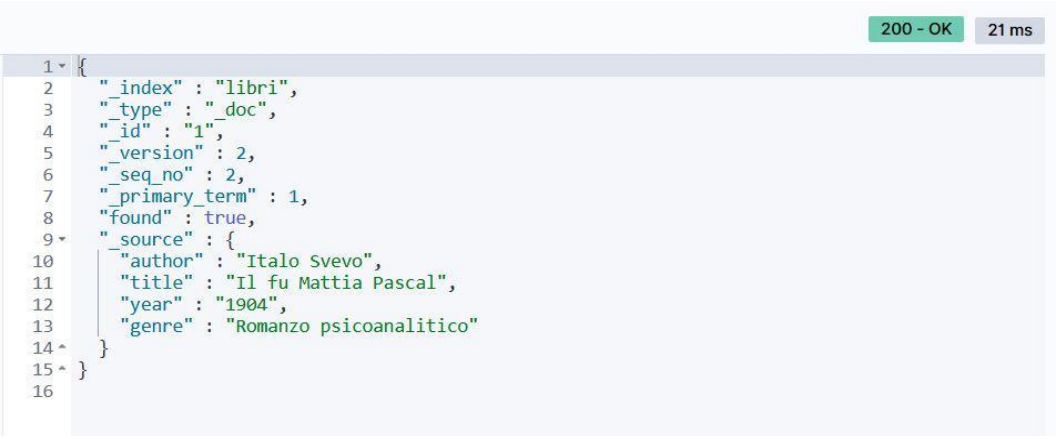

Figura 5.11

Con *found* ci viene detto se il libro è stato trovato (*true*) o no (*false*). Nella voce *_source* ci vengono elencati tutti i nostri campi e nel caso lo avessimo fatto, anche il testo completo del romanzo registrato. Per cercare determinati campi basta inserirli insieme a *_source* in questo modo:

GET bibliography/novels/1?_source=author,title

Qui ci verrà restituito solo l'*author* e il *title*. Per restituire il contenuto basta utilizzare solo l'endpoint *_source* nel medesimo modo visto in precedenza.

Per restituire diversi documenti basta indicare i loro indici con l'endpoint *_mget* (multi-get):

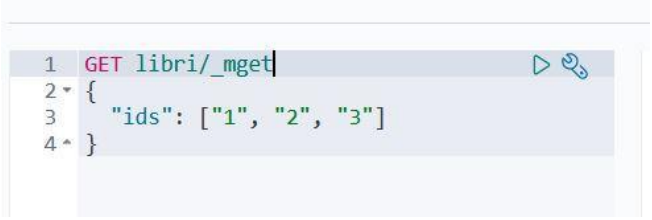

Figura 5.12

Nel caso gli _id siano sbagliati o inesistenti, la query avverrà lo stesso, ma restituirà *found* : *false*;

Per cancellare un libro basta scrivere:

DELETE /libri/_doc/1

e il libro con quell'indice sarà rimosso. Ma con una particolarità, vediamo l'output:

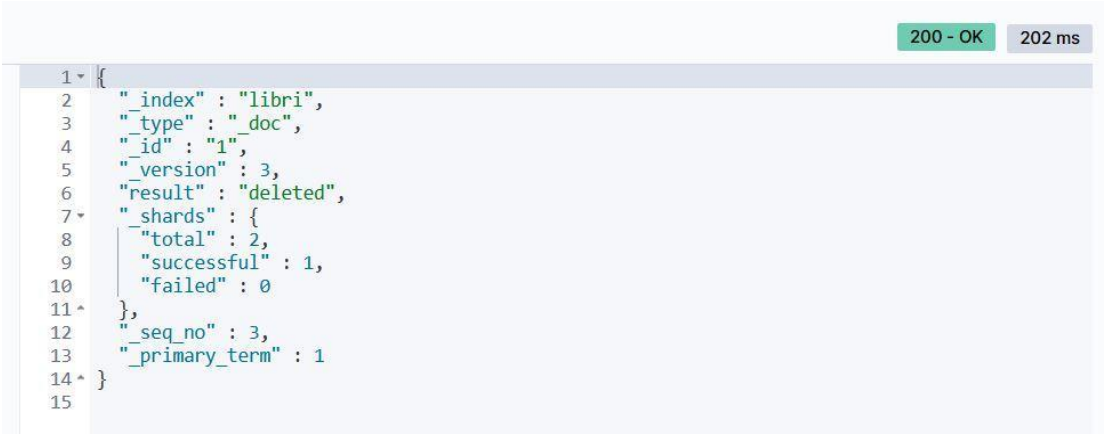

Figura 5.13

Possiamo notare che il numero della versione è stato incrementato e il nostro documento eliminato, questo per due motivi:

1. Elasticsearch contrassegna il documento solo come *deleted* (cancellato) e non lo rimuove direttamente dal disco fino a quando non ci sarà una nuova indicizzazione.

2. Lavorando con indici che sono distribuiti su più nodi, le versioni e il loro controllo è fondamentale. Elasticsearch segna ogni cambiamento con le versioni, compresa la cancellazione.

Noi possiamo apportare modifiche a informazioni solo in base al numero di versione conosciuto. Se nel cluster esiste già una versione più aggiornata rispetto alla nostra, il tentativo di modifica genererà un errore e non andrà a buon fine.

Possiamo infine richiamare più voci contemporaneamente, con il comando *_bulk* possiamo crearne ed eliminarne più d'una, ecco un esempio:

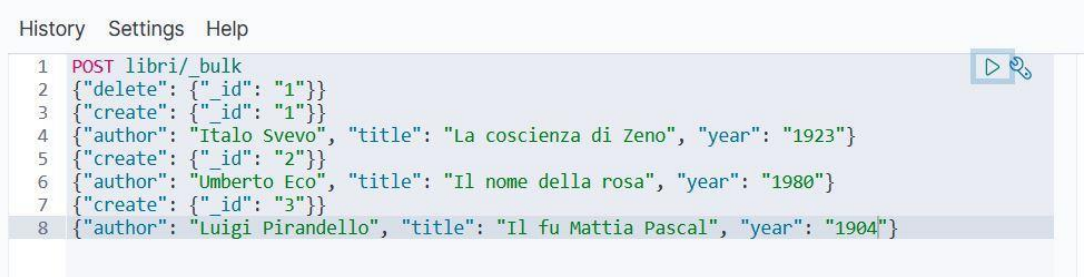

Figura 5.14

Per ogni documento bisogna specificare quale azione compiere (*create, index, update, delete*), poi si specifica qual voce creare e in quale posizione. Se utilizziamo più indici basta non indicare l'indice nella prima riga di codice ma inserirlo come in figura 5.14. È necessario inserire un *request body* in una nuova riga dove sarà inserito il contenuto del documento. Ovviamente con il metodo DELETE non sarà necessario inserirlo, dato che il documento deve essere cancellato.

Queste sono solo alcune delle funzionalità principali di come gestire i propri dati e informazioni grazie ad Elasticsearch e Kibana. Le stesse operazioni possono essere scritte anche senza Kibana, tramite la shell del proprio sistema operativo, tramite i comandi cURL.

5.7.2 Mapping di Elasticsearch

Il mapping in Elasticsearch è fondamentale e spesso è causa di errori nella ricerca dei propri dati. Per questo comprenderlo e renderlo adatto alle nostre esigenze è molto importante. Esso determina come gli algoritmi debbano interpretare le stringhe di input che noi inseriamo nella nostra console di Kibana o di un qualsiasi altro strumento con il medesimo scopo. Per vedere quale mapping è in uso nel vostro indice bisogna inserire il seguente comando:

GET libri/_mapping

Tutti i campi di un documento sono assegnati a tipi di testo e a parole chiave. Elasticsearch ha tanti *type* che mette a disposizione, i principali sono i seguenti:

- **string**: include sia *text* che *keyword*. Mentre le parole chiave sono considerate corrispondenze esatte, Elasticsearch presume che il testo debba essere analizzato, come vedremo tra poco, prima di poter essere utilizzato.
- **numeric**: Elasticsearch riconosce diversi valori di tipo numerico. Ad esempio, il tipo *byte* può assumere valori tra -128 e 127, *long* si ha un margine compreso tra -2^{63} bis $2^{63}-1$.
- **date**: una data può essere specificata giornalmente o annualmente o anche con un orario preciso. Inoltre, si ha la possibilità di specificare una data nel formato temporale di secondi o millisecondi.
- **boolean**: i campi formattati come booleani possono avere valore vero (*true*) o falso (*false*).
- **binary**: in tali campi è possibile memorizzare dati binari.
- **range**: in questo modo specificate dei limiti di competenza entro i quali è necessario rimanere. Può essere tra due valori numerici, due date o anche tra due indirizzi IP.

Questi *type* (e molti altri descritti nella documentazione di Elastic presente in bibliografia) sono o *exact-value* o *full text*. Nel primo caso Elasticsearch considera il campo come esatto e gestirà quel dato come tale, oppure, nel caso della ricerca full text l'informazione in input dovrà essere elaborata, come descritto nel corso dell'elaborato. Nella mappatura può diventare importante quindi come analizzare un dato, cioè l'*analyzing*. Come abbiamo già visto può essere:

- Tokenizing: avviene la procedura della tokenizzazione, da un testo vengono raccolti i vari token individuali.
- Normalizing: i token trovati sono normalizzati, vengono ridotti tutti in minuscolo, alle forme base, eliminando spesso termini ripetitivi.

Una problematica di Elasticsearch è quella di non poter cambiare la propria mappatura. Per questo è molto importante essere capaci e consapevoli del mapping che si crea, perché l'unico modo per utilizzare una mappatura è creare un nuovo indice. Elastic, nella sua documentazione spiega e viene incontro con varie tipologie di soluzioni per ovviare a questo problema, ma la tecnica più comoda e veloce è quella di creare un nuovo indice "template", che serve da modello e raggiunto il mapping voluto considerare questo il nostro indice eliminando il precedente. Guardiamo un esempio:

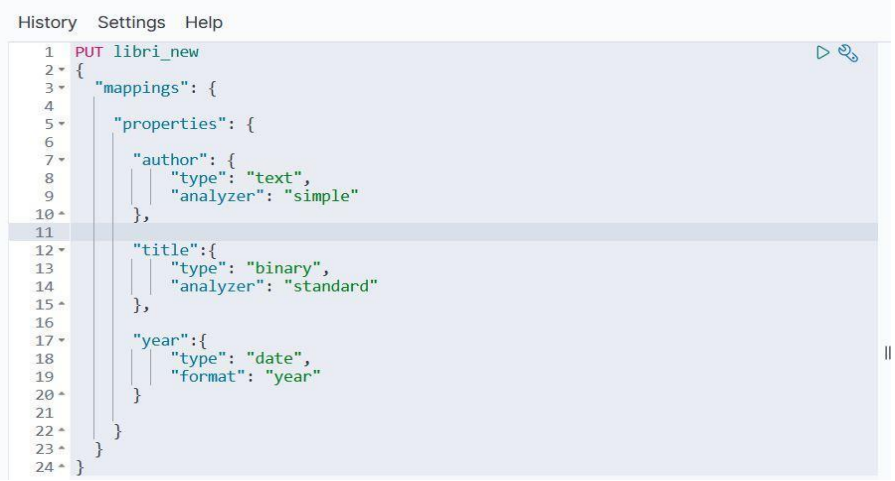

Figura 5.15

Author e title sono *text* e quindi adatti alla ricerca full text, quindi serve un giusto analyzer, uno *simple* per l'autore del libro e standard per il titolo. Definendo l'anno come *date* esso sarà un *exact-value* Elasticsearch limiterà la sua ricerca sull'anno esatto.

5.7.3 Ricerca full-text, Elasticsearch e la sua sintassi

Dopo aver guardato il mapping e l'indicizzazione di Elasticsearch è doveroso affrontare ciò per cui essi sono stati concepiti: la full-text search. Abbiamo visto come cercare i documenti all'interno dell'indice, ma la vera potenza di Elasticsearch è che ci permette di cercare contenuto specifico tra milioni di dati e informazioni velocemente attraverso un semplice motore di ricerca. Per poterlo utilizzare ci viene fornito l'endpoint *_search*. Unendolo al metodo GET possiamo richiedere per esempio di visualizzare tutte le voci nel nostro indice (se non indichiamo nessun indice ci troverà tutti gli indici e i rispettivi documenti sul nostro elasticsearch):

GET libri/_search

L'output (non verranno riportati tutti i libri presenti nell'indice in Figura 5.15) è mostrato di seguito:

Figura 5.15

Le informazioni che sono di nostro interesse sono sotto la voce *hits*. Sono stati trovati libri per un *total* (numero) pari a 4, con un massimo di score pari a 1. Poi sono inserite altre informazioni che ci fanno comprendere la ricerca full-text maggiormente. Un ulteriore campo hits, entro la quale vengono elencati tutti i documenti che hanno fatto match con i criteri di richiesta compiuti nella nostra query iniziale. Importante è anche lo score che viene assegnato ad ogni documento trovato che indica quanto il nostro risultato sia rilevante in base alla richiesta che è stata fatta. Ovviamente nel nostro caso saranno tutti pari a 1 perché abbiamo richiesto solo una loro visualizzazione ed elencazione, senza particolari specifiche.

Vengono anche indicati su quanti shard è stata compiuta la ricerca, se si è verificato un timeout, quanti ms sono stati impiegati per la ricerca e se l'operazione ha effettivamente avuto successo.

Elasticsearch mostra di default solo i primi dieci risultati della nostra ricerca. Tuttavia, si possono modificare tali impostazioni cambiano o inserendo i parametri:

- **size**: quanti risultati dovrebbe mostrare Elasticsearch?
- **from**: quante voci dovrebbe saltare il programma prima di visualizzarle?

Ad esempio: GET *libri/_search?size=15&from=10* (vedere 15 risultati dopo i primi 10).

Posso cercare anche attraverso un campo specifico come l'autore:

GET libri/_search?q=author:Alessandro (mi troverà Alessandro Manzoni nel nostro caso).

Questo è un esempio di query semplice e questo genere di ricerca è la versione lite di quella che è la potenza di Elasticsearch. Possiamo usare più operatori, aggiungere e togliere criteri attraverso rispettivamente + e -, utilizzando anche codifiche percentuali. Si consiglia di controllare tutta la documentazione fornita dagli sviluppatori sulla pagina di Elastic, che è completa e dettagliata.

Per ricerche più complesse, per evitare errori di scrittura o impostazione si può utilizzare un metodo più pratico: le Query DSL (Domain Specific Language). Permettono di creare query basate su JSON con semplicità ma dalle capacità sorprendenti, per niente scontate, date le considerazioni fatte durante la stesura dell'elaborato. Queste ricerche hanno due contesti principali: il query context e il filter context. Vediamoli con degli esempi.

Importante è sempre iniziare con la parola chiave *query* alla quale aggiungeremo una leaf query clauses (cioè un particolare una clausola che ci permette di cercare particolari valori in particolari campi) come ad esempio match, term o range (possono essere usati anche da soli). Vediamo la Figura 5.16:

Figura 5.16

Il risultato mostrerà i documenti che avranno il campo dell'autore che farà match con "Alessandro". Per visualizzare tutte le voci si può utilizzare

"match_all": {} al posto di del semplice *match*. Il contrario della ricerca fatta invece viene indicato con *"match_none"*.

Possiamo anche cercare un solo termine in più campi contemporaneamente in questo modo:

Figura 4.17

Possiamo anche utilizzare, oltre alle leaf query clauses, delle compound query clauses, che possono racchiudere altre leaf query o compound query, per utilizzarle in modo combinato. Come rappresentato nella seguente immagine:

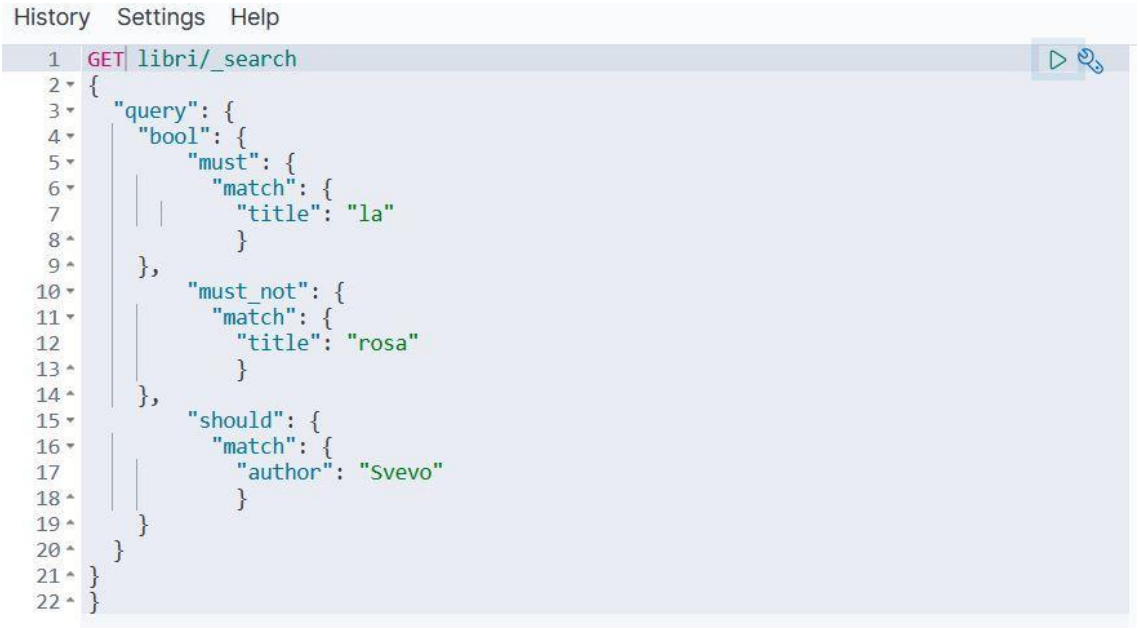

Figura 4.18

Qui abbiamo usato il compound query clause *bool* unito con alcune modalità di vincolo: *must*, la clausola deve verificarsi, *must_not*, la clausola non deve verificarsi, *should*, se si verifica questo match allora si aumenta lo score e quindi la pertinenza del risultato della ricerca.

Oltre al query context appena descritto aggiungiamo ora il filter context. Nel primo caso una query clause rispondeva alla domanda "Quanto bene questo documento corrisponde alla clausola della ricerca?", al quale viene attribuito uno score; nel secondo caso la clausola risponde alla domanda "Questo documento corrisponde alla clausola?" e la risposta è Sì o No, senza nessun calcolo dello score. Può essere molto utile il filter context nel calcolo di valori esatti come l'anno di pubblicazione di un libro. Ad esempio:

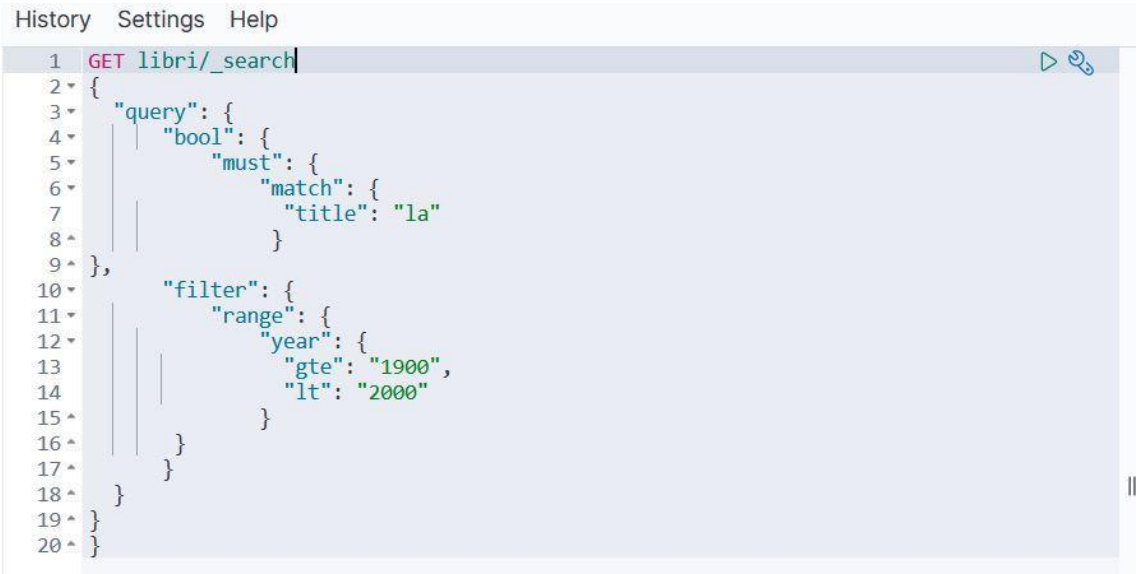

Figura 4.19

In questo esempio viene aggiunto un filtro che permette di elencare i libri che sono stati pubblicati in un range di anni tra il 1900 e il 2000, insieme alla clausola che il titolo debba contenere la parola "la".

Sul sito ufficiale di Elastic si possono trovare query molto più complesse e possibilità di ricerca sempre più complessa. Possiamo notare anche da questi semplici esempi la potenza che può avere un prodotto software del genere. Mantenendo sempre prestazioni di alto livello anche con un numero di dati e documenti notevoli lo rendono un sistema capace di dare numerosi vantaggi rispetto a tutti gli altri concorrenti. Vediamo nel prossimo capitolo alcune delle query importanti per la complessa full-text search, con un conseguente confronto prestazionale con MongoDB.

5.8 Full-text query

Le *query full text*[34] consentono di cercare campi di testo analizzati e indicizzati. La stringa di query viene elaborata allo stesso modo di quando si compie l'indicizzazione su quel campo.

Le query sono:

- *Intervals query*: query full text che consente un controllo dettagliato dell'ordinamento e della prossimità dei termini trovati.
- *Match query*: è la query standard per eseguire una query full text, incluso il fuzzy matching o per frase o prossimità.
- *match_bool_prefix query*: crea una query *bool* che fa match con ogni termine come terms query, ad eccezione dell'ultimo termine, che è corrisposto come prefix query.

[34] [36] Full-Text queries, https://www.elastic.co/guide/en/elasticsearch/reference/current/full-text-queries.html#full-text-queries

- *match_phrase query*: come la match query, ma utilizzata per il match frasi esatte o la prossimità di parole.
- *Match_phrase_prefix query*: come il match_phrase query, ma esegue una ricerca con caratteri jolly sull'ultima parola.
- *Multi_match query*: la versione multi-campo della match query.
- *Common terms query*: È una query molto più specializzata, che dà preferenza a parole non comuni.
- *Query_string query*: È dedicata agli utenti esperti, permette l'uso della sintassi compatta della stringa per le query di Lucene. Consente di specificare le condizioni AND, OR, NOT e la ricerca multi-campo all'interno di una singola stringa di una query.
- *Simple_query_string query*: versione più semplice e robusta della sintassi query_string adatta per essere diretta agli utenti.

6. Confronto prestazionale

6.1 Differenze, limiti e punti di forza

Nel corso dell'elaborato abbiamo compreso che un'opzione non relazionale potrebbe migliorare significativamente la velocità di ricerca semplicemente a causa della lentezza di *join* su set di grandi dimensioni. Elasticsearch crea documenti unici con le informazioni rilevanti che poi cercheremo. Le informazioni a cui si accede insieme vengono archiviate insieme, è questa la logica del modello. Elasticsearch ha funzionalità di ricerca più complete e più facilmente utilizzabili, ma non dovrebbe essere utilizzato come primary data source (origine dei dati primaria); è meno affidabile di SQL Server, la sua natura completamente distribuita non permette una integrità dei dati ai livelli del database appena citato. L'indice, per rendere la ricerca così veloce, deve essere costruito in anticipo per cercare in modo efficiente ma questo aggiunge complessità e si può ritardare l'aggiornamento dei dati in "real time"; è un buon compromesso avere scritture lente se la velocità di ricerca su milioni di file è di appena qualche secondo. Bisogna quindi capire le proprie esigenze, essere consapevoli quali siano i piani di produzione o la mission aziendale nel caso si voglia creare un prodotto software. Un database SQL sarà sempre più semplice a livello di configurazione e infrastruttura. Elastic e i database NoSQL non hanno costo di licenza molto spesso (MS SQL Server sì) ma si avrà il bisogno delle replication e nel caso si scelga di usare una hosted solution (soluzione ospitata, utilizzando server e servizio di terzi) questa può diventare anche molto costosa. Ogni framework di database ha i suoi pro e contro, Elasticsearch è valido perché è facile installarlo e farlo funzionare, scala orizzontalmente con l'aggiunta di nodi nel cluster senza problemi, ha un mapping dei dati e si può modificare in ogni

momento la porzione di RAM e di disco che si vuole utilizzare; ha un supporto per vari text analysers, ha controllo preciso su quali parti di documenti cercare e ha varie API disponibili in più linguaggi di programmazione. Un'aggiunta recente allo stack di Elastic è il supporto di X-Pack che permette l'uso di più funzionalità come monitoraggio, avviso, sicurezza, ecc.

Un'altra differenza importante che bisogna notare è che Elasticsearch è stato costruito su Lucene e utilizza i Lucene segments per scrivere i dati all'interno di inverted indexes. Tutte le informazioni sui metadati come il mapping dell'indice, le impostazioni e gli stati del cluster sono tutti file basati sempre su Lucene. Il problema di questi segmenti è che sono di natura immutabile e ogni commit crea un nuovo segmento. Questi segmenti si uniscono insieme in base alle impostazioni di merge (unione). Quando ogni documento viene aggiornato, un nuovo documento viene generato e fa l'override del precedente, ciò rende gli update dei dati pesanti. Per evitare di generare troppi segmenti e scambio di dati eccessivo, Elasticsearch mantiene un transactional log (log delle transazioni) per ogni index, evitando in questo modo un commit di basso livello su ogni operazione di indicizzazione. I registri con i transactional log sono utili per il ripristino dei dati in caso di guasto, arresto anomalo o danneggiamento dei dati.

La scrittura e gli aggiornamenti sono quindi lenti, ai quali occorre anche elevato uso di risorse. Per questo la scelta di accompagnare elasticsearch a un database relazionale (o NoSQL, come MongoDB, veloce nella scrittura, ma lento nella lettura di molti dati) è una scelta che può portare a tempi di ricerca veloci e alla scrittura e aggiornamenti dei dati stabili, sicure e solo all'occorrenza poterli utilizzare nelle ricerche full-text. È bene ribadire che non necessariamente l'utilizzo di un database ne escluda un altro.

Si potrebbe pensare di archiviare i dati attraverso MongoDB (o SQL Server) e utilizzare Elasticsearch esclusivamente per le sue capacità di ricerca full-text. In

poche parole, in questo caso si inviano sottoinsiemi dei campi dei dati Mongo sui quali poi avverrà una ricerca da parte di Elasticsearch. Se i dati cambiano continuamente, più volte al giorno con molte modifiche, può richiedere la reindicizzazione di quel record in Elastic e abbiamo visto che non è immediata questa operazione. Aggiornare determinati campi comporta la sua completa indicizzazione tutte le volte. Prendere in considerazione Elasticsearch come database unico di archiviazione dati potrebbe essere rischioso. La velocità di ricerca varia anche in base alle query che si attuano, senza dubbio se si vuole avere piena libertà con domande che sono spesso diverse, con campi non costanti e filtri differenti, allora conviene usare Elasticsearch. Essendo un motore di ricerca e nato per quello e non come database, non sempre garantisce risultati esatti, ma restituisce quelli che si avvicinano maggiormente alla query fatta. Per avere buone prestazioni con SQL Server devi creare con cura gli indici del *catalog* per farli corrispondere il più possibile con le query eseguite. In particolare, se disponi di più colonne in base ai quali interrogare, devi creare attentamente i tuoi indici in modo che riducano il set di dati che verrà interrogato il più velocemente possibile. Se ciò non avviene e il matching con i valori delle tabelle non è adeguato le prestazioni calano drasticamente.

Confrontiamo ora con qualche tabella presa da DB-BEST technologies che ci mostra bene le differenze tra Lucene e SQL Server Full Text Search.

Comparing MS SQL Full Text Search and Lucene

	Lucene	MS SQL FTS
Index auto update	No	Yes
Store data in index	Yes	No
Location in RAM	Yes	No
Interface	API	SQL
Queering multiple columns	Yes	Yes
Stop words, synonyms, sounds-like	Yes	Yes
Custom Index Documents Structure	Yes	No
Wildcards	Yes	With restrictions
Spellchecking, hit-highlighting and other extensions	Provided in "contrib" extensions library	No

Figura 6.1 *COMPARING MICROSOFT SQL SERVER FTS AND APACHE LUCENE* [37]

Elasticsearch ha la possibilità di aggiornare in modo automatico gli indici ma è una procedura che richiede attenzione e un mapping adeguato. Possiamo riscontrare nella tabella le caratteristiche precedentemente esposte, Lucene supporta le wildcard all'inizio o in mezzo o alla fine delle parole, mentre SQL Server solo alla fine. La vera differenza risiede nella velocità di computazione di query complesse, se supportate da SQL Server.

Le seguenti tabelle (Figura 6.2 e Figura 6.3) fanno riferimento a una ricerca avvenuta su un Wikipedia dump (Il dump, in informatica, è un elemento di un database contenente un riepilogo della struttura delle tabelle del database medesimo e/o i relativi dati, ed è normalmente nella forma di una lista di dichiarazioni SQL. Tale *dump* è usato per lo più per fare il backup del database[35]). Vogliamo visualizzare gli articoli più rilevanti che sono stati modificati di recente tra i più ricercati dagli utenti. La computazione della ricerca è avvenuta su un

[35] Significato di *dump*: https://it.wikipedia.org/wiki/Dump

computer Intel i5 -3330, 3.0 GHz, RAM: 8 GB. Le tabelle seguenti riportano le velocità di esecuzione di Lucene e SQL Server:

Indexing speed, size and single query execution time

	Lucene	MS SQL FTS
Indexing Speed	3 MB/sec	1 MB/sec
Index Size	10-25%	25-30%
Simple query	<20 ms	< 20 ms
Query With Custom Score	< 4 sec	>20 sec

Figura 6.2 *COMPARING MICROSOFT SQL SERVER FTS AND APACHE LUCENE* [37]

Parallel Query Executions

We used 10 threads, average execution time per query in ms.

		MS SQL FTS	Lucene (File System)	Lucene (RAM)
Cold System	Simple Query	56	643	21
	Boost Query	19669*	859	27
Second executions	Simple Query	14	8	<5
	Boost Query	465	17	9

*average time, the very first query could be executed up to 2 min(!)

Figura 6.3 *COMPARING MICROSOFT SQL SERVER FTS AND APACHE LUCENE* [37]

Si nota come SQL Server sia leggermente più veloce nelle query più semplici eseguite la prima volta, ma nel caso di query molto complesse Lucene è nettamente superiore.

Dopo aver capito le principali differenze tra le ricerche che avvengono nei software, database e search engine appena analizzati, vediamo un caso di studio particolare per comprendere meglio come le prestazioni e le velocità nella ricerca di testo siano effettivamente differenti.

6.2 Caso di studio: US Fights

Microsoft SQL Server può utilizzare come motore di ricerca full-text Elasticsearch e la condivisione dei dati dal DBMS relazionale alla piattaforma di Elastic può avvenire attraverso diverse modalità. Come citato in precedenza l'Elastic Stack è messo a disposizione degli utenti e uno dei prodotti utile allo scopo è Logstash. "Logstash ingerisce, trasforma e spedisce dinamicamente i dati indipendentemente dal formato o dalla complessità", descrive il sito ufficiale[36] di Elastic.

Logstash si interpone tra i due software per garantire che i database di SQL Server arrivino con il formato e gli indici corretti. Permette attraverso opportuni driver (SQL *JDBC Driver*[37] nel nostro caso) di poter sincronizzare e tenere aggiornati i dati tra un database di un DBMS ed Elasticsearch. Come mostrato in figura 6.4 Logstash importa una moltitudine di dati differenti, da piattaforme diverse e li trasferisce ad Elasticsearch con la conseguente possibilità di analizzare, archiviare e monitorare questi dati. Kibana, con tutte le sue funzionalità e potenzialità, può lavorare su tutto ciò che viene introdotto sulla piattaforma da Logstash.

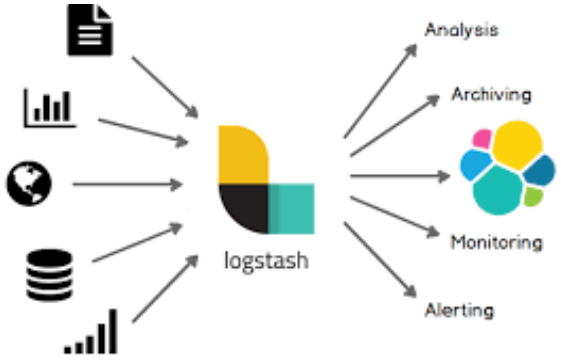

Figura 6.4 *Controlla la tua applicazione Spring Cloud con Elastic Stack* [38]

[36] Logstash: https://www.elastic.co/logstash
[37] Sito Microsoft dove poter trovare JDBC Driver per SQL Server: https://docs.microsoft.com/it-it/sql/connect/jdbc/download-microsoft-jdbc-driver-for-sql-server?view=sql-server-ver15

Una volta installato Logstash, dovrà essere configurato, tramite un file apposito .conf, nel quale vengono definiti input, filtri e output del servizio (dettagli spiegati sul *sito ufficiale Elastic*[38]) e una volta avviato, il database selezionato di MS SQL Server sarà importato su Elasticsearch.

Prendiamo in esame un database di 71.4 MB riguardante più di 500.000 voli compiuti negli Stati Uniti nel mese di febbraio 2020. Verifichiamo attraverso alcune query di ricerca full-text su SQL Server ed Elasticsearch se la differenza di velocità effettivamente si presenta anche su dati di media o piccola dimensione come quelli considerati. Il test è stato eseguito su un computer con processore Intel(R) Core(TM) i5-8250, CPU 1.60 GHz -1.80 GHz, RAM 8 GB.

QUERY 1

La più grande differenza si nota nel recupero dei dati completo:

Elasticsearch per recuperare la totalità completa dei documenti risulta essere più performante. (Figura 6.5):

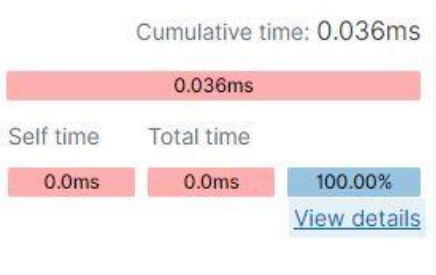

Figura 6.5

Mentre MS SQL Server ha impiegato oltre i 10 secondi:

Stato connessione	Aperto
Tempo trascorso connessione	00:00:10.3446763
Versione server	15.0.2000

Figura 6.6

[38] [39] Configuring Logstash: https://www.elastic.co/guide/en/logstash/current/configuration.html

QUERY 2

Selezionando i voli effettuati nei giorni dal 20 al 28 febbraio 2020, le differenze di prestazioni dei due software sono rilevanti:

Query di MS SQL Server:

```
SELECT *
FROM dbo.Feb_2020_ontime
WHERE CONTAINS(["DAY_OF_MONTH"], '"2*"');
```

Figura 6.7

Il tempo impiegato per la ricerca è il seguente:

Stato connessione	Aperto
Tempo trascorso connessione	00:00:04.1552761
Versione server	15.0.2000

Figura 6.8

Query di Elasticsearch:

```
GET flights/_search
{
  "query":{
    "bool": {
    "filter": {
        "range":{
        "DAY_OF_MONTH": {
          "gte": "20",
          "lt": "28"
        }
        }
    }
  }
  }
}

```

Figura 6.9

Il tempo di esecuzione è di 22 ms rispetto ai 4,155 secondi di SQL Server:

Figura 6.10

QUERY 3

L'esempio che segue è eseguito con i sistemi "caldi", cioè con gli indici caricati e pronti all'utilizzo. Ciò rende i motori di ricerca più prestanti.

La query è effettuata attraverso FREETEXT, che lavora sul significato delle parole, in SQL Server è la seguente:

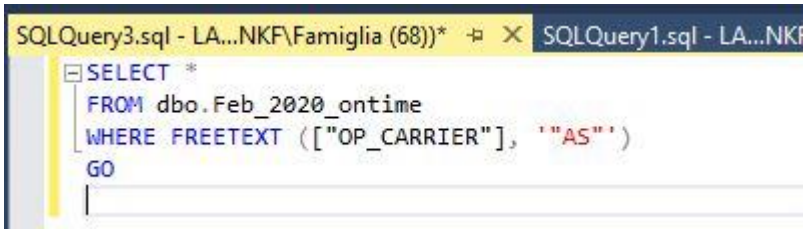

Figura 6.11

Il tempo trascorso nella ricerca è di 890 ms indicato in figura 6.12:

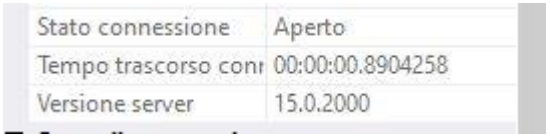

Figura 6.12

Elasticsearch, con l'uso di query per la ricerca con stemming presentata in Figura 6.13, ha una velocità di 20 ms:

```
GET flights/_search
{
  "query": {
    "simple_query_string": {
      "fields": [ "OP_CARRIER" ],
      "query": "AS"
    }
  }
}
```

Figura 6.13

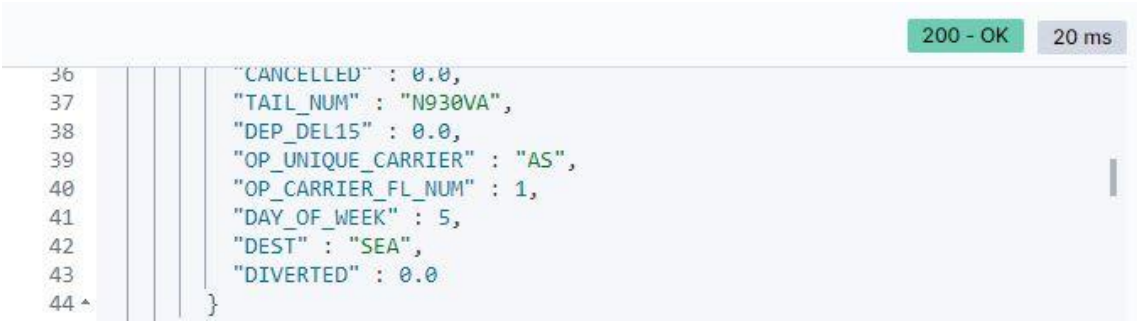

Figura 6.14

Con l'aumentare della complessità delle query e della complessità e computazione le performance di SQL Server tendono a peggiorare.

Il problema maggiore si presenta quando SQL Server deve creare e restituire grandi tabelle con i risultati della ricerca. Il tempo di recupero dei dati aumenta in modo direttamente proporzionale alla quantità di righe (cioè corrispondenze) da ritornare.

Il grafico in figura 6.15 mostra gli esempi eseguiti precedentemente (*Query 1, 2 e 3*) e i risultati derivanti da altre due ricerche effettuate come conferma.

	Elasticsearch	MS SQL Server
Query 1: tutti i voli presenti nel database	36	10344
Query 2: voli effettuati dal 20 al 28 Feb (*CONTAINS*)	22	4155
Query 3: voli che hanno "AS" nel campo/colonna "OP_CARRIER" (*FREETEXT*)	20	890
Query 4: voli effettuati in diversi giorni che hanno peso diverso (*CONTAINSTABLE*)	58	823

Figura 6.15

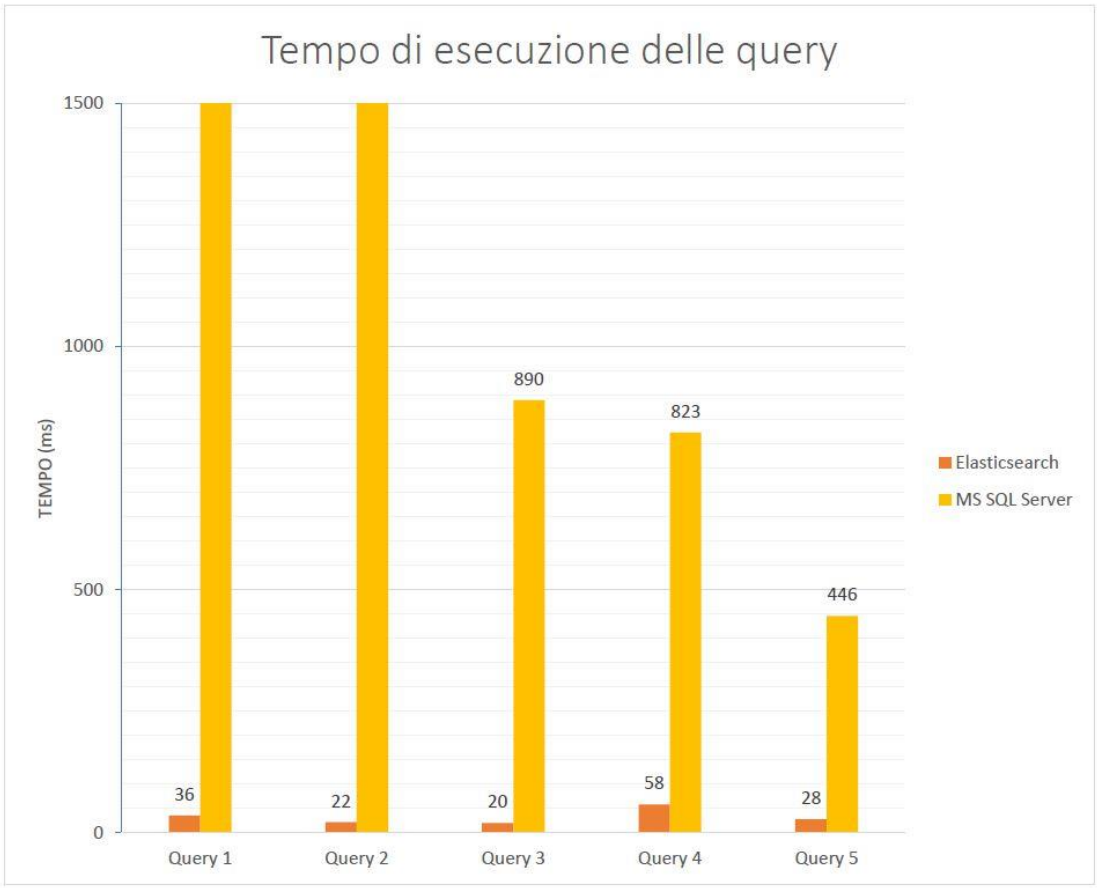

Figura 6.16

Nelle prime due ricerche il risultato corrispondeva a centinaia di migliaia di righe/documenti, negli altri casi a decine di migliaia. Il tempo di costruzione delle tabelle nei primi due casi rende la ricerca lenta e inadeguata alle richieste di elevate performance odierne.

MS SQL Server ha prestazioni altalenanti, con differenza di velocità e tempo troppo ampie tra una ricerca e l'altra per essere accettabili e non competitive rispetto ad altri software sul mercato, come appunto Elasticsearch.

Elasticsearch mantiene sempre le prestazioni stabili. Con pochi o molti dati, con la complessità o meno delle query, la velocità di calcolo e di ricerca è sempre pressoché la medesima.

7. Conclusioni

Lo scopo di questo elaborato è stato quello di approfondire, seppur in modo non del tutto esaustivo e completo, quella che è la ricerca full-text, l'importanza del suo ruolo al giorno d'oggi e come poterla eseguire al meglio per i nostri fini e obiettivi.

La diffusione dei *Big Data*[39], con dati strutturati ma soprattutto non strutturati, semi-strutturati, provenienti da differenti fonti e sorgenti, dai social network alla domotica, alla sensoristica e il mondo riguardante l'*Internet of Things*[40], con formati completamenti diversi tra loro. Da questi dati abbiamo visto nel corso dell'elaborato, quale ricchezza può derivare dal loro perfetto controllo e gestione.

[39] Cosa sono i Big Data: https://www.cloudtalk.it/big-data-esempi/
[40] Che cos'è l'Internet of Things (IoT): https://www.redhat.com/it/topics/internet-of-things/what-is-iot

I sistemi SQL sono diventati lo standard per i database ormai dagli anni '70 e col tempo hanno avuto un uso sempre più efficiente delle risorse. L'integrità dei dati è forte e ben compresa (attraverso ACID) così come l'accesso ai dati, anch'esso standard. Sono modelli di dati rigidi, che richiedono una attenta progettazione iniziale per garantire prestazioni adeguate e resistere all'evoluzione; la modifica di schemi può comportare tempi lunghi e spesso di inattività. Altro problema abbiamo visto può essere il ridimensionamento orizzontale, non completamente supportato o con tecnologie ancora troppo immature. Non esiste praticamente scalabilità a causa single point of failure, talvolta mitigato con tecniche di replica e failover.

L'aumentare dei dati e in particolare la ricerca di essi in un ambiente così strutturato porta a tempi di elaborazione delle ricerche full-text non compatibili con un contesto dove la User Experience e la rapidità di scelte e decisioni è importante.

MongoDB, citato in precedenza, è un database NoSQL molto popolare e scalabile che è leader tra i database orientati al documento. Scelto soprattutto quando la use case richiede un database altamente scalabile con transazioni ad alto rendimento, ma rimane comunque al quinto posto nel ranking del database più popolare (secondo *DB-Engine*[41]). Davanti a lui sono ancora ora diffusissimi i database relazionali. MongoDB, come tutti i database NoSQL, non sono in grado di compiere ricerche su milioni di dati con la velocità di Elasticsearch. Il motore di ricerca è performante nel campo della ricerca full-text, l'analisi dei log, la ricerca di anomalie e il rilevamento di cause particolari o principali (anche su il diretto competitor Sorl di Apache).

Bisogna quindi capire ciò per cui una tecnologia viene utilizzata e qual è il fine della produzione e progettazione, che sia aziendale, a scopo di lucro o meno. Il

[41] DB-Engine Ranking popularity: https://db-engines.com/en/ranking

mondo è pervaso da una infinità di dati nelle loro forme più disparate e con essi si sono sviluppati anche molteplici strumenti per trattarli. Elasticsearch è ormai diventato la miglior soluzione di ricerca full-text, grazie al suo continuo sviluppo e integrazione dell'Elastic Stack; è usato dalle migliori compagnie di gestioni dati al mondo come Netflix, GitHub, Foursquare, SoundCloud, ecc.

Affiancare tecnologie tradizionali a sistemi in grado di gestire e cercare enormi quantità di dati in rapidissimi tempi di risposta può essere ad oggi la soluzione migliore.

Alla luce di quanto detto è importante sempre di più guardare al futuro, come si prospetta la ricerca negli anni a venire, che è strettamente legata ai database e alla loro gestione. L'amministrazione, lo studio, l'indagine dei dati è e sarà fondamentale e bisogna prenderne atto in tutti i contesti progettuali.

APPENDICE

A. MongoDB, funzionalità principali

MongoDB [42]è un DBMS orientato al documento scritto in C++, uscito in pianta stabile nel 2009. È un sistema open-source e può operare con i maggiori sistemi operativi. Il documento è un insieme di coppie campo-valore, in cui il campo è costituito da una stringa e il valore è un tipo di dato in formato BSON.

All'interno di una istanza MongoDB potete avere zero o più database, ognuno dei quali agisce come un contenitore di alto livello per tutto il resto. Un database può avere zero o più collezioni. Le collezioni sono composte da zero o più documenti. Un documento è a sua volta composto da uno o più campi. Gli indici in MongoDB funzionano in gran parte come le loro controparti RDBMS. È importante sapere che quando si chiedono dati a MongoDB questi restituisce un puntatore al set di risultati chiamato cursore, col quale possiamo già compiere operazioni come contare i documenti o spostarci in avanti, prima ancora di scaricare i dati.

Abbiamo detto che MongoDB prevede una suddivisione e organizzazione dei documenti in collezioni (*collection*). Ciascuna collection è identificata dal suo nome e raggruppa documenti logicamente e strutturalmente simili ma non necessariamente identici. Importante è che il campo _id di un documento debba identificarlo univocamente all'interno della collection. Per creare una collection basta inserire un nuovo documento in una collection qualsiasi e se questa non esiste, allora verrà creata (posso anche crearla esplicitamente con *db.createCollection()*).

[42] [13] MongoDB, https://www.mongodb.com/

MongoDB consente la modifica dei dati attualmente presenti in una specifica collection attraverso inserimenti, aggiornamenti e cancellazioni. Gli inserimenti possono essere effettuati attraverso il metodo *db.collectionName.insert(),* che permette di aggiungere a *collectionName* un documento o un array di documenti.

Per gli aggiornamenti e le cancellazioni, invece, è possibile utilizzare, rispettivamente, i metodi *db.collectionName.update()* e *db.collectionName.remove()*. Entrambi consentono di specificare delle condizioni da utilizzare come filtro per determinare su quali documenti, fra tutti quelli esistenti nella collection a cui si applica l'operazione di aggiornamento o di eliminazione, è necessario agire.

MongoDB mette a disposizione un comando (*$isolated*) che consente di isolare, durante la sua esecuzione, un'operazione di scrittura che coinvolge più documenti. In questo modo nessuno potrà avere accesso ai prodotti interessati all'aggiornamento finché quest'ultimo non sarà stato portato a termine. Anche utilizzando *$isolated*, comunque, non si rende un aggiornamento complesso atomico: se un errore dovesse verificarsi quando l'esecuzione dell'operazione di scrittura ha già avuto inizio il sistema non si occupa automaticamente di riportare i documenti già aggiornati ai valori precedenti. Il solo modo per avere atomicità è racchiudere tutte le informazioni da modificare in un solo documento, strutturandolo in modo complesso[43].

MongoDB permette di scegliere fondamentalmente tra due differenti modalità di organizzazione dei dati: è possibile optare per una strutturazione "*normalizzata*" dei dati, in cui informazioni correlate ma logicamente distinte sono mantenute mediante documenti separati, eventualmente raccolti in collection e database differenti, allo stesso tempo però il sistema ammette anche un'organizzazione differente, "*denormalizzata*", in cui informazioni fra loro strettamente connesse

[43] Procedura nel dettaglio in: [24] Analisi e sperimentazione del DBMS NoSQL MongoDB: il caso di studio della Social Business Intelligence, Tesi di Laurea di Alice Gambella, Anno Accademico 2013/2014.

possono essere concentrate in un unico documento, sfruttando la possibilità di gestire documenti innestati.

Lo strumento più semplice di aggregazione offerto da MongoDB è costituito da alcuni comandi che consentono di mettere in atto operazioni specifiche e comuni, quali: il calcolo del numero di documenti di una collection che soddisfano determinati criteri di selezione, la restituzione di tutti e soli i valori distinti assunti da un certo campo fra i document di una data collection che rispettano specifiche condizioni, il raggruppamento dei document di una particolare collection che soddisfano i criteri fissati dall'utente. Il raggruppamento può essere definito su uno o più campi, sia realmente esistenti che calcolati, e può prevedere anche la definizione, e dunque l'applicazione, di semplici funzioni aggregate, come il conteggio del numero di elementi in ogni gruppo o la somma (o anche la media) dei valori assunti da un certo campo fra i membri dello stesso gruppo. Per raggiungere questi semplici obiettivi è possibile far uso rispettivamente dei comandi count, distinct e group o, in alternativa, dei metodi corrispondenti (ovvero count(), db.collection.distinct() e db.collection.group()).

A.1 Aggregation Pipeline e MapReduce

All'estremo opposto rispetto alla semplice metodologia già presentata si pone il *MapReduce*[44], un meccanismo facilmente parallelizzabile che offre possibilità decisamente più ampie rispetto all'opzione precedente. Map-reduce è certamente lo strumento più potente offerto da MongoDB per quanto riguarda le possibilità di aggregazione dei dati.

[44] Disponibile su sito ufficiale: [13] MongoDB, https://www.mongodb.com/

Il MapReduce è un paradigma di data processing che condensa grandi volumi di dati in utili risultati aggregati. Viene fornito il comando mapReduce. L'esempio qui sotto mostra come avviene il mapReduce:

In questa operazione in particolare (Figura A.1), MongoDB applica la map phase a ciascun documento nella collezione che fa match con la condizione della query e la funzione di map emette coppie chiave-valore. Alle chiavi che hanno più di un valore verrà applicata reduce phase, che raccoglie e condensa i dati aggregati. Verranno poi memorizzati i risultati in una raccolta. Si possono poi ulteriormente condensare o elaborare i risultati con una funzione di finalizzazione.

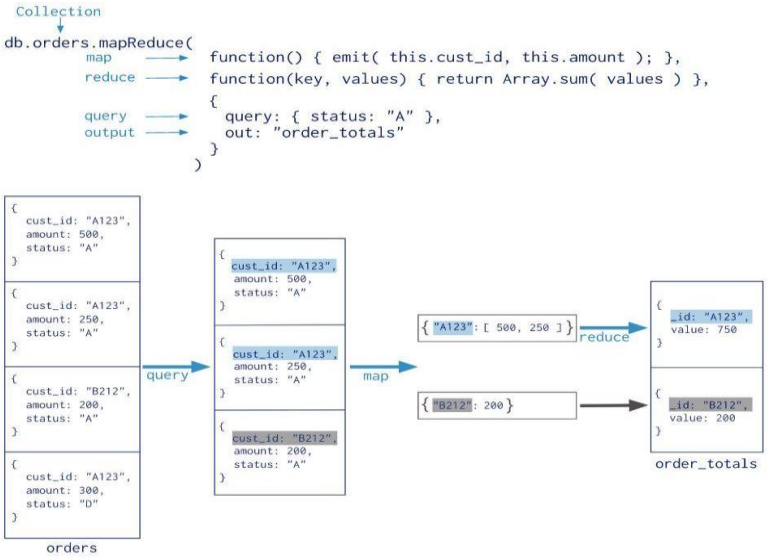

Figura A.1 *MongoDB* [13]

Un trade-off apprezzabile fra le due realtà già presentate è costituito dalla *Aggregation Pipeline*[45], che permette anche una ricerca di testo al suo interno, campo di nostro interesse.

L'aggregation pipeline è un framework per l'aggregazione dei dati modellato sul concetto di di data processing pipelines (pipeline di elaborazione dei dati). I

[45] Approfondimento su sito ufficiale: [13] MongoDB, https://www.mongodb.com/

documenti entrano in una pipeline a più fasi (stage) che trasforma i documenti in risultati aggregati. Le fasi della pipeline non devono necessariamente produrre un documento di output per ogni documento di input; ad esempio, alcune fasi possono generare nuovi documenti o filtrare i documenti. MongoDB fornisce il metodo *db.collection.aggregate ()* nella shell di mongo e il comando aggregate per eseguire la pipeline di aggregazione. Il comando aggregate opera su una singola collection, logicamente passando l'intera collection nella aggregation pipeline e per ottimizzare questo processo è possibile usare filtri che evitano di leggere e scansionare tutta la collection: $match, che usa un indice per filtrare i documenti all'inizio della pipeline, serve per richiedere un sottoinsieme di una collection; $sort, questa fase usa un indice fino a quando non è preceduta da una fase di $project, $unwind, $group; $group, può usare un indice per trovare il primo documento in ogni gruppo se sono soddisfatti questi criteri: se è preceduto da una $sort stage che ordina in base a un campo, c'è un indice del campo che effettivamente corrisponde all'ordinamento e deve essere aggiunto $first; $geoNear, prende in considerazione subito i geospatial index (va usato come primo operatore).

Ecco un semplice esempio:

```
db.orders.aggregate([
  { $match: { status: "A" } },
  { $group: { _id: "$cust_id", total: { $sum: "$amount" } } }
])
```

Il primo Stage: $match filtra i documenti in base al campo dello stato e passa alla fase successiva quei documenti che hanno lo stato uguale ad "A".

Il secondo Stage: $group raggruppa i documenti in base al campo cust_id per calcolare la somma dell'importo per ogni cust_id univoco.

Nell'aggregation pipeline è disponibile la ricerca di testo tramite l'uso dell'operatore di query $text (descritto meglio in seguito) nella fase di $match.

Ci sono alcune limitazioni:

- il $match che include $text deve essere la prima fase della pipeline;
- $text può comparire una sola volta nella fase e non può apparire in espressioni con $or o $not;
- Questo tipo di text search non restituisce documenti in ordine in base al matching score, ma per ordinare i risultati bisogna utilizzare l'espressione $meta nella fase di $sort;

L'esempio seguente ha una collection *articles* che ha un *text index* sul campo *subject*:

```
db.articles.createIndex( { subject: "text" } )
```

Vogliamo che il matching sia fatto o su *cake* o su *tea*, ordinati secondo *textScore* in ordine decrescente e ritorna solo il *title* nel set di risultati:

```
db.articles.aggregate(
  [
        { $match: { $text: { $search: "cake tea" } } },
        { $sort: { score: { $meta: "textScore" } } },
        { $project: { title: 1, _id: 0 } }
  ]
)
```

A.2 Selettori

Un selettore di query in MongoDB assomiglia alla clausola *WHERE* di un comando SQL. Viene usato per trovare, contare, aggiornare e rimuovere documenti dalle collezioni.

Un selettore è un oggetto JSON la cui forma più semplice è {}, che rintraccia tutti i documenti. Usiamo {campo: *valore*} per trovare documenti il cui campo sia uguale a *valore*. Usiamo {campo1: *valore1*, campo2: *valore2*} per indicare l'operatore *and*.

Usiamo gli operatori *$lt*, *$lte*, *$gt*, *$gte* e *$ne* rispettivamente per minore di (*less than*), minore o uguale (*less than or equal*), maggiore di (*greater than*), maggiore o uguale (*greater then or equal*) e diverso da (*not equal*).

I Selettori possono essere usati in abbinamento al comando *find*, o anche con *remove, count*, e col comando update.

Quando in MongoDB vogliamo cambiare il valore di uno o più campi è necessario usare (insieme ad update) l'operatore di aggiornamento *$set* con in seguito i vari campi con i valori da aggiornare. Senza *$set* il documento viene completamente sostituito dai campi dell'update. l'operatore *$inc* consente di aumentare o diminuire il valore di un campo. Possiamo aggiungere a un array specifico un valore grazie all'operatore *$push*.

Una delle sorprese più piacevoli che update ci riserva è senz'altro il supporto per gli upsert. Se upsert trova il documento cercato lo aggiorna, altrimenti lo crea. Per attivare l'upsert passiamo un terzo parametro di aggiornamento: {upsert: true}.

Per default update aggiorna solo il primo documento trovato, quindi va usata l'opzione {multi: true} quando si desidera aggiornare tutti i documenti rintracciati[46].

[46] Funzionalità descritte in: [15] Il piccolo libro di MongoDB, Karl Seguin, 2012

A.3 Come avviene la full text search su MongoDB

Guardiamo ora come avviene una ricerca di testo semplice ma chiara per comprendere il funzionamento di MongoDB. Tutte le funzionalità che verranno descritte sono rese molto più facili da costruire su MongoDB Atlas, descritto successivamente.

MongoDB (come descritto sull'omonimo sito proprietario[47]) supporta operazioni di query che eseguono una ricerca di testo nel contenuto di una stringa. Per eseguire la ricerca di testo utilizza un indice di testo e l'operatore *$text*.

Prendiamo in considerazione un esempio posto sul sito di MongoDB: come costruire un indice di testo e usarlo per trovare dei coffee shops, dandogli solo campi di testo.

Creiamo di principio la nostra collection *stores* con i documenti:

```
db.stores.insert(
  [ { _id: 1, name: "Java Hut", description: "Coffee and cakes" },
    { _id: 2, name: "Burger Buns", description: "Gourmet hamburgers" },
    { _id: 3, name: "Coffee Shop", description: "Just coffee" },
    { _id: 4, name: "Clothes Clothes Clothes", description: "Discount clothing" },
    { _id: 5, name: "Java Shopping", description: "Indonesian goods" }
  ])
```

MongoDb fornisce indici di testo per supportare le query di ricerca di testo sul contenuto della stringa. Gli indici di testo possono includere qualsiasi campo il cui valore è una stringa o un array di elementi stringa.

[47] Text Search MongoDB: https://docs.mongodb.com/manual/text-search/

Per eseguire query text search, è necessario disporre di un indice di testo nella collection. Una collection può avere un solo text search index, ma quell'indice può coprire più campi.

Facciamo un esempio. Nella nostra mongo shell possiamo eseguire la ricerca qui sotto:

db.stores.createIndex({ name: "text", description: "text" })

abbiamo così creato un indice che permette la ricerca sui campi name e description.

Una volta definito l'indice possiamo eseguire ricerche su una collection attraverso l'operatore di query *$text.*

$text tokenizzerà la stringa passata utilizzando spazi bianchi o segni di punteggiatura come delimitatori e poi eseguirà un OR dei token risultanti da questo processo. Ad esempio, usando la stringa seguente potremmo trovare tutti i negozi che hanno i termini nella lista coffee, shop e java:

db.stores.find({ $text: { $search: "java coffee shop" } })

Possiamo cercare anche documenti che includono una frase esatta, basta porla tra le virgolette doppie. In questo caso il match con solo i documenti che includono questa frase. Ad esempio, se vogliamo trovare i documenti che contengono la frase "coffee shop" useremo:

db.stores.find({ $text: { $search: "\"coffee shop\"" } })

Per escludere una parola dalla ricerca si può anteporre il carattere "-". Se vogliamo trovare gli store che contengono "java" o "shop" ma non "coffee" useremo:

db.stores.find({ $text: { $search: "java shop -coffee" } })

Molto importante è il concetto di sorting. Di default MongoDB restituisce i risultati in ordine casuale. Tuttavia, le query del text-search calcoleranno un punteggio di pertinenza per ogni documento che specifica quanto un documento abbia un buon match con la query appena eseguita.

Per ordinare i risultati basta porre esplicitamente il campo *$meta textScore* come segue:

```
db.stores.find(
   { $text: { $search: "java coffee shop" } },
   { score: { $meta: "textScore" } }
).sort( { score: { $meta: "textScore" } } )
```

La ricerca di testo all'interno di MongoDB avviene quindi grazie all'operatore $text, dopo avere compiuto una opportuna indicizzazione. La text indexes supporta le query su contenuti di stringhe e l'indice può includere un qualsiasi campo che ha valore nella stringa o array degli elementi di una stringa. Abbiamo appena mostrato come avviene una indicizzazione nel caso più semplice, ma ci sono molti casi particolari, con la possibilità di specificare il peso che può avere un campo dell'indice rispetto agli altri nel medesimo indice. Si possono creare Wildcard Text Indexes, utili con dati altamente strutturati o quando non è chiaro quali campi includere nell'indice di testo per l'esecuzione delle query. Gli indici supportano lingue e stop words, il case sensitive, il diacritic sensitive e molto altro. Una collection può avere al massimo un indice. Per avere il nome dell'indice, usare il metodo *db.collection.getIndexes()* e per cancellare l'indice bisogna passare il nome dell'indice nel metodo *db.collection.dropIndex()*.

Abbiamo compreso che $text esegue una text search sul contenuto dei campi indicizzati con il text index. Il $search Field è fondamentale, specifica la stringa

di parole sulla quale $text dovrà compiere il parse usando il text index per la query.

Ci possono essere varie match operation: le stopwords, dove l'operatore $text ignora parole poco significative come the o and; le stemmed words, di default la corrispondenza avviene con un completo stemmed word e quindi vengono messe a disposizione delle funzionalità aggiuntive case sensitive option e diacritic sensitive option per avere più precisione. Il primo attivato se si desidera che il suffisso di stem contenga lettere maiuscole e quindi l'operatore $text dovrà corrispondere con l'esatta parola (per usarlo: *$caseSensitive: true*). Il secondo si attiva nel caso in cui sia importante che il suffisso di stem contenga i simboli diacritici (come la *h* o la *i* in italiano, oppure *é* o *è*, che non cambiano il senso alla frase o parola, ma la pronuncia). Per usarlo impostare *$diacriticSensitive: true*.

Si possono cercare i risultati in una determinata lingua attraverso *$language: "nomelingua"*. Abbiamo anche visto come aggiungere lo score e ordinare i risultati attraverso lo score. Si possono limitare il numero di risultati attraverso *limit()*.

A.4 MongoDB Atlas

MongoDB Atlas[48] è un servizio di Cloud Database che consente di ospitare un database su una piattaforma di Cloud Computing. È comodo e scelto come testing grazie alla sua semplicità e non ha bisogno di installazione e gestione a livello di infrastruttura. MongoDB Atlas Search semplifica la creazione di funzionalità di ricerca full-text rapide e pertinenti sui dati nel cloud ed è disponibile esclusivamente con MongoDB Atlas. Si possono creare delle ricerche direttamente su questa piattaforma e quindi si elimina la necessità di replicare dati altrove. Non è necessario impostare, gestire e ridimensionare una

[48] [16] MongoDB Atlas Search, https://www.mongodb.com/atlas/search

piattaforma di ricerca separata. Si può creare un indice di ricerca sulle collection velocemente o con una chiamata API. Si riescono a testare query di ricerca e visualizzare i risultati nella Aggregation Pipeline Builder per poi poterli utilizzare poi come codice nelle applicazioni. È integrato con il framework di aggregazione MongoDB ed è costruito su Apache Lucene, ormai la libreria standard del settore per la ricerca full-text.

Atlas Search è nata proprio per adattarsi alle nuove esigenze di mercato che hanno portato moltissime società a preferire altri prodotti software come ElasticSearch, Apache Solr o Algolia.

Con MongoDB Atlas vengono forniti molti più modi per ottimizzare la rilevanza dei risultati della ricerca e supportare i risultati di query più rapidi perché è basato su Apache Lucene, il motore di ricerca utilizzato anche su Elasticsearch e Solr. Viene aumentato anche il set di funzionalità con oltre 35 lingue, più tipi di dati, autocomplete, fuzzy search e molto altro. MongoDb ha intenzione di ampliare tutte queste funzionalità e tutti i casi d'uso che la piattaforma può supportare. Sembra che non siano previsti lavori aggiuntivi invece per migliorare la ricerca full-text di MongoDB e osservando l'andamento di mercato attuale, questo lascia il posto a implementazioni differenti di MongoDB Atlas o di altri prodotti software proprietari come Elasticsearch.

Elasticsearch ha prestazioni e funzionalità maggiori di MongoDB, che presenta una ricerca di testo meno complessa di Elastic, per questo nel capitolo successivo lo analizzeremo e cercheremo di confrontare le due tipologie di Text Search offerte per capire effettivamente se ElasticSearch è migliore e dove.

A.5 Caso di studio: Ivy

Prendiamo in esame un test compiuto da QuarksLab (compagnia che si occupa di sicurezza di servizi, applicazioni e dati) attraverso Ivy, un enorme software di

ricognizione della rete, progettato per rilevare potenziali difetti su reti di grandi dimensioni, formate da anche milioni di indirizzi IP. Può eseguire molte operazioni, come la raccolta degli http-header o testare le vulnerabilità di servizi su queste reti, adattandosi a funzionalità più specifiche e con più flessibilità.

Ivy fornisce agli utenti l'accesso a un motore di ricerca per l'estrazione dei dati e delle informazioni in modo veloce, preciso.

Ivy è nata con MongoDB. Questo perché la principale problematica di Ivy consisteva nel memorizzare dati non strutturati e un dato di un singolo indirizzo IP differiva notevolmente da un altro: alcune porte potrebbero essere aperte e altre no, alcune informazioni potrebbero non essere proprio raccolte, oppure in modo incompleto, potrebbero esserci dei plug-in, ecc.

MongoDB era la scelta più giusta per avere questo genere di dato, vista la sua natura schemaless, con aggregazioni potenti, è intuitivo e anche ben documentato.

Ogni indirizzo IP di una scansione è un documento, che appartiene a una collection. I campi più importanti sono: quali porte sono state aperte e quali plug-in mandano una risposta indietro.

Un esempio di documento può essere:

```
{ "_id" : ObjectId("54c921e3902ed0377abffcab"),
  "country" : "US",
  "hostIpnum" : "1234567890",
  "tlds" : ["zetld.com"],
  "ports" : [{
     "service" : "http",
     "server" : "Microsoft IIS",
     "version" : "",
     "port" : {
        "value" : 443,
        "protocol" : "tcp"
     }}],
  "pluginResults" : {
     "sc:nse:http-headers" : [{
        "relatedPorts" : [{
```

```
            "value" : 443,
            "protocol" : "tcp"
        }],
        "data" : {
            "outputRaw" : "\n  Content-Length: 3151  \n  \n  (Request type: GET)\n"
        }}],
    "sc:nse:sslv2" : [{
        "relatedPorts" : [{
            "value" : 443,
            "protocol" : "tcp"
        }],
        "data" : {
            "outputRaw" : "\n  SSLv2 supported\n    SSL2_RC4_128_WITH_MD5\n
SSL2_DES_192_EDE3_CBC_WITH_MD5"
        }}]}
}
```


Documento che può facilmente essere memorizzato con MongoDB.

Sono stati implementati anche filtri e aggregazioni. Filtrare gli indirizzi IP irrilevanti, quando si trattano milioni di indirizzi può essere fondamentale e molto efficiente (ad esempio considerare solo indirizzi IP con quattro porte aperte). Le aggregazioni invece evidenziano una particolarità di una scansione, dato che analizzare gli indirizzi uno a uno non avrebbe senso, allora si mostra magari una rete in base a particolari dati come la distribuzione di porte aperte, o servizi sulla rete, o i paesi che hanno determinati indirizzi IP, adattandosi ai filtri che stiamo utilizzando.

Lo scopo di questo prodotto era quello di essere fruibile, disponibile e portatile, utilizzabile anche su comuni computer (Intel Core 7 4700MQ processor, 8GB RAM con una regolare SSD) Fu fatto un primo test su una scansione di indirizzi IP della Malesia, circa 7 milioni, con molte porte e plug-in. Le prestazioni furono pessime, le aggregazioni richiedevano tra i 30 e i 40 secondi. Il problema fu risolto cambiando il meccanismo di cache, suddividendo l'unica collection in più collection dedicate a determinate scansioni e aumentando la memoria RAM.

Si era trovato un rimedio, ma si era consapevoli che con un aumento ingente di dati le prestazioni si sarebbero abbassate ulteriormente.
Bisognava adattarsi a MongoDB, invece che avere un prodotto che si adattasse facilmente alle richieste di Ivy.
Le dedicated collection di indirizzi IP per ogni scansione è quindi una soluzione scadente ma che permette di migliorare la velocità avendo scansioni medio-piccole rispetto a quelle più grandi.
Le aggregazioni su Ivy sono dinamiche, perché vengono calcolate in base al filtro scelto in quel momento. Quando questo avviene devono essere calcolati risultati in tempo reale su un'ampia mole di dati e non solo sulle piccole aggregazioni basate sulle scansioni senza filtro (calcolate magari precedentemente alla richiesta).
Lo scopo degli sviluppatori di Ivy era quello di dare massima libertà agli utenti, anche con implementazione su laptop, ma tutto ciò non fu reso possibile a causa delle scarse prestazioni.
Non solo, MongoDB mantiene ampi intervalli di spazio inutilizzato tra i vari document durante l'inserimento di dati per ordinare in modo efficiente i successivi inserimenti. Gli indici occupano molto spazio su disco e se si pensa che per avere alte prestazioni bisogna lavorare con dati memorizzati su RAM (che non ha grandi dimensioni, soprattutto su laptop) questo potrebbe essere un problema. In più in tutto questo la frammentazione di MongoDB non porta a miglioramenti, ma a un rallentamento del reperimento dati e all'utilizzo di remove e update.
Per questo motivo gli sviluppatori hanno speso le restanti risorse più che a un miglioramento della gestione di MongoDB alla sperimentazione e testing di tecnologie di supporto, tra cui il motore di ricerca Elasticsearch.

Non esisteranno mai delle tabelle di confronto che sottolineano le differenze in modo corretto MongoDB ed Elasticsearch perché sono uno un database e l'altro un motore di ricerca, sono nati per due motivi differenti, il primo puntava alla flessibilità dei dati, l'altro ad avere una ricerca veloce.

Oltre ad avere una scrittura dei filtri più intuitiva, su un laptop con un filtro applicato su 6.357.427 di documenti di una collection, i documenti corrispondenti alla stessa richiesta di seguito sono stati trovati con:

- 14 649 secondi per MongoDB;
- 0.715 secondi per Elasticsearch;

MongoDB:

```
{ $or: [
  {"nbOpenedPorts": {$not: {$gt: 4}}},
  {"ports.port.value": {$ne: 80}}
]}
```

Elasticsearch:

```
{"filter": {"not": {"and": [
  {"term":
    {"ports.port.value": 80}},
  {"range":
    {"nbOpenedPorts": {"gt": 4}}}
]}}}
```

(negazione della "port.port.value": 80)

Differenza notevole tra le due ricerche.

Lo stesso vale per le aggregazioni. In seguito, viene riportata una semplice aggregazione per paese. Cioè si vuole il numero di indirizzi IP contati per paese nella scansione in esame.

MongoDB pipeline:

```
[{"$group": {"_id": "$country", "total": {"$sum": 1}}}, # Sum of IP addresses by "country"
 {"$sort": {"_id": 1}}]                    # Sort on the country names
```

Elasticsearch:

```
{"aggs": {<agg_name>:
        {"terms": {"field": "country",        # Aggregation on field "country"
                "size": 0              # Return every buckets
                "order": {"_term": "asc"}}  # Sort on the country names
}}
```

Anche qui la performance di Elasticsearch è notevolmente superiore. Sullo stesso laptop e sulla stessa collection della prova precedente i risultati sono:

- 42,116 secondi per MongoDB (primo tentativo);
- 3,799 secondi per MongoDB (secondo tentativo, il primo aggregato ha inserito i dati sulla RAM);
- 0,902 secondi per Elasticsearch (primo tentativo);

Si è inoltre visto che l'inserimento di dati ha risultati similari. Interessante invece è l'analisi di come avviene il reperimento dei documenti con MongoDB ed Elasticsearch (testato con una semplice operazione di scrolling dei dati). Le figure seguenti illustrano come i risultati con blocchi da 100 000 documenti (su oltre 23 milioni) siano differenti:

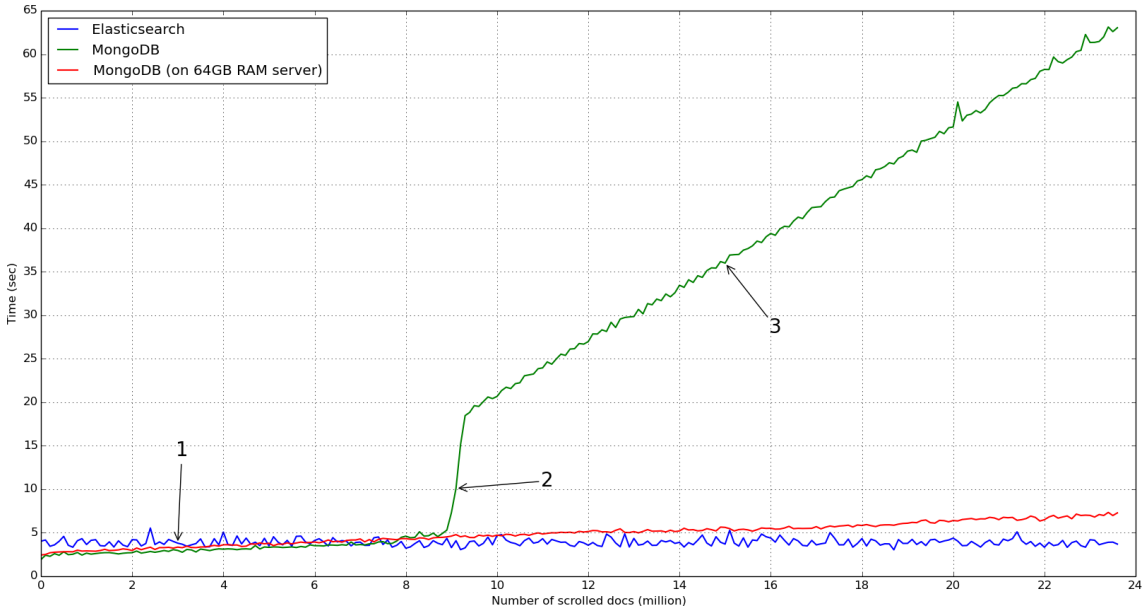

Figura A.2 *MongoDB vs. Elasticsearch: The Quest of the Holy Performances* [21]

Riportando le considerazioni fatte su questo test dagli sviluppatori possiamo dire:

1) Con i primi blocchi, MongoDB è più reattivo di Elasticsearch, ma il primo tende poi a rallentare mentre il secondo rimane costante.
2) Vi è un repentino innalzamento del tempo in secondi (in corrispondenza di nove milioni di indirizzi IP) dove la cache passa alle letture da RAM a quelle su disco. Infatti, ci dicono gli sviluppatori, la velocità di lettura su disco da 20MB/s a 150MB/s.

Se questo calcolo viene fatto su un server con memoria da 64GB, il sistema ha sempre spazio RAM a disposizione continua l'aumento molto lentamente, ma con prestazioni ancora accettabili.

3) MongoDB su laptop continua ad avere un tempo di risposta con un continuo aumento sempre maggiore, mentre Elasticsearch è ancora sui quattro secondi di tempo di risposta.

Dettaglio sui primi blocchi:

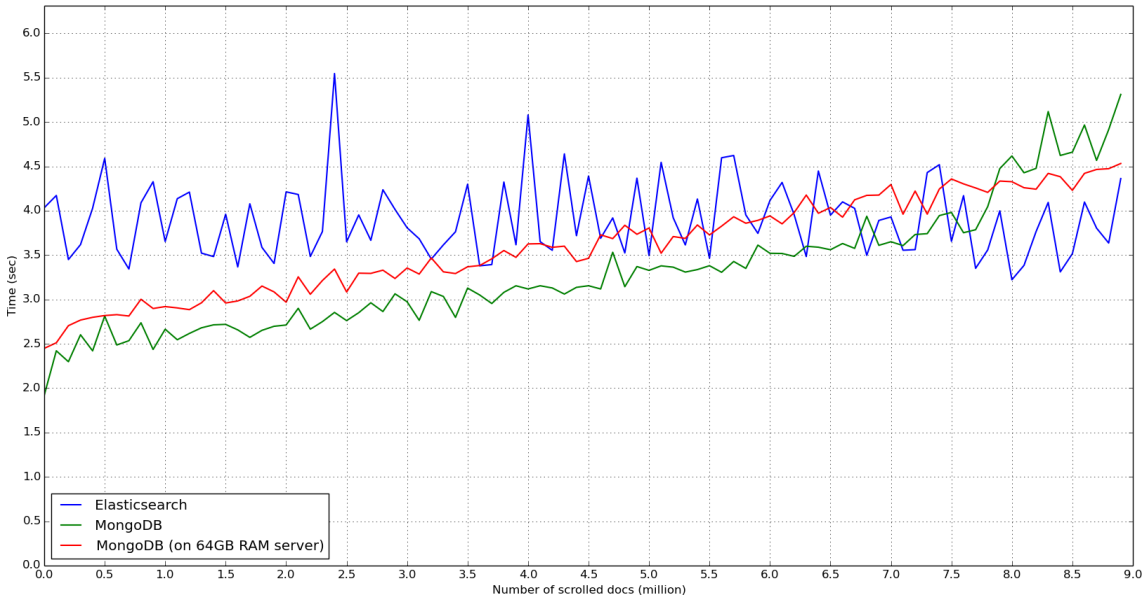

Figura A.3 *MongoDB vs. Elasticsearch: The Quest of the Holy Performances* [21]

Elasticsearch ad ogni chiamata raccoglie i documenti seguenti per la chiamata successiva se questa è conosciuta. MongoDB attraversa la collection per la sua interezza come se avesse un cursore che punta sempre al documento che viene richiesto. Questo è un problema perché il metodo di ordinamento è gravoso sulle prestazioni e ad ogni chiamata la collection dovrà essere riordinata.

MongoDB quindi si destreggia discretamente su dati di dimensioni medio piccole, con prestazioni anche lievemente superiori a Elasticsearch, ma ha notevoli difficoltà a soddisfare esigenze più ampie.

Grazie ai dati forniti dagli sviluppatori, notiamo la medesima situazione riprendendo anche l'esempio precedente su l'aggregation sul numero di indirizzi IP per paese e anche qui possiamo notare ancora di più il divario tra i due prodotti:

"by country" aggregation results

Document number	MongoDB (sec)	Elasticsearch (sec)	Ratio (MongoDB/Elasticsearch)
29,103,278	19.435	0.758	25.647
24,504,269	16.795	0.774	21.693
4,213,248	3.972	0.133	29.891
336,832	0.299	0.056	5.325
28,672	0.024	0.005	4.863

Figura A.4 *MongoDB vs. Elasticsearch: The Quest of the Holy Performances* [21]

Elasticsearch non è privo però di problemi. Come detto nel capitolo precedente, esso detta dei vincoli di mappatura a cui prestare molta attenzione quando si aggiungono funzionalità aggiuntive perché potrebbero essere errate a causa di un ordinamento interno che potrebbe sembrare del tutto insensato. La gestione dei nodi Elasticsearch può essere più complicata che gestire una base di MongoDB. Quest'ultimo ha migliori prestazioni di inserimento ed è più flessibile, ma Elasticsearch, con tutto lo stack di Elastic, diventa un completo e promettente motore di ricerca, con tantissime funzionalità.

È opportuno ribadire quanto, anche con sistemi NoSQL, sia importante affiancare motori di ricerca Full-Text qualora la dimensione dei dati lo richiedano, per avere maggiori funzionalità più specifiche e selettive, da rendere l'ecosistema completo. Tutt'ora MongoDB utilizza per la Full-Text Search l'implementazione di Elasticsearch.

8. Bibliografia

[1] Elastic Reference, https://www.elastic.co/guide/en/elasticsearch/reference/current/index.html

[2] An Overview on Elasticsearch and its usage, Giovanni Pagano Dritto, 2019, https://towardsdatascience.com/n-overview-on-elasticsearch-and-its-usage-e26df1d1d24a

[3] ElasticSearch Architecture Overview, how Elasticsearch organizes data, http://solutionhacker.com/elasticsearch-architecture-overview/

[4] Our story, https://www.elastic.co/about/history-of-elasticsearch

[5] Full-text search, https://en.wikipedia.org/wiki/Full-text_search

[6] Ricerca documenti full-text, trovare il documento a partire dal testo, https://www.bucap.it/news/approfondimenti-tematici/software-gestione-documentale-aziendale/ricerca-documenti-full-text-trovare-il-documento-a-partire-dal-testo.htm

[7] Full Text Search, Best Full Text Search System Capabilities, https://web.archive.org/web/20101223192214/http://www.lucidimagination.com/full-text-search

[8] Ricerche full text, Giuseppe Maggi, 2015 https://www.html.it/pag/54194/ricerche-full-text/

[9] Esecuzione della query con ricerca Full-Text, Documentazione di SQL, 2017, https://docs.microsoft.com/it-it/sql/relational-databases/search/query-with-full-text-search?view=sql-server-ver15

[10] Ricerca full-text, Documentazione di SQL, 2018, https://docs.microsoft.com/it-it/sql/relational-databases/search/full-text-search?view=sql-server-ver15

[11] Full Text Search — How to install, configure and use it with SQL Server (and Outsystems), Joao Duro, 2019, https://itnext.io/full-text-search-how-to-install-configure-and-use-it-with-sql-server-and-outsystems-23fcf316e870

[12] Hands on Full-Text Search in SQL Server, Jefferson Elias, 2017, https://www.sqlshack.com/hands-full-text-search-sql-server/

[13] MongoDB, https://www.mongodb.com/

[14] MongoDB Atlas: un servizio di Cloud Database, 2020, https://www.appuntisoftware.it/mongodb-atlas-un-servizio-di-cloud-database/

[15] Il piccolo libro di MongoDB, Karl Seguin, 2012, https://nicolaiarocci.com/mongodb/il-piccolo-libro-di-mongodb.pdf

[16] MongoDB Atlas Search, https://www.mongodb.com/atlas/search

[17] Ricerca Full-Text in MongoDB, https://it.accentsconagua.com/articles/code/full-text-search-in-mongodb.html

[18] MongoDB Atlas Search - Product Demo, 2020, https://www.youtube.com/watch?v=kZ77X67GUfk

[19] Elasticsearch: il motore di ricerca flessibile, 2019, https://www.ionos.it/digitalguide/server/configurazione/elastic-search/

[20] Elasticsearch: How to Add Full-Text Search to Your Database, 2017, https://medium.com/@MentorMate/elasticsearch-how-to-add-full-text-search-to-your-database-ee2f3ea4d3f3

[21] MongoDB vs. Elasticsearch: The Quest of the Holy Performances, 2015, https://blog.quarkslab.com/mongodb-vs-elasticsearch-the-quest-of-the-holy-performances.html

[22] Database SQL e NoSQL: differenze, pro e contro, come scegliere, 2020,

https://www.01net.it/database-sql-nosql-differenze-pro-contro-come-scegliere/

[23] Progettazione e prototipazione di un sistema di Social Business Intelligence con ElasticSearch, Tesi di Laurea di Luca Longobardi, Anno Accademico 2014/2015.

[24] Analisi e sperimentazione del DBMS NoSQL MongoDB: il caso di studio della Social Business Intelligence, Tesi di Laurea di Alice Gambella, Anno Accademico 2013/2014.

[25] Sistemi di Information Retrieval: Performance Evaluation, R. Basili, a.a 2004-5

https://didattica-2000.archived.uniroma2.it/BdDD/deposito/004_evaluationrbas.pdf

[26] Elasticsearch è oggi il più diffuso motore di ricerca aziendale, Marco Bizzantino, 2017 https://www.kiratech.it/news/elasticsearch-%C3%A8-il-pi%C3%B9-diffuso-motore-di-ricerca-aziendale-machine-learning

[27] Elasticsearch vs. MongoDB, Gedalyah Reback, 2020, https://logz.io/blog/elasticsearch-vs-mongodb/

[28] Hands on Full-Text Search in SQL Server, Jefferson Elias, 2017

https://www.sqlshack.com/hands-full-text-search-sql-server/

[29] Google annuncia la sua nuova funzionalità semantica per interagire con i libri, 2019, https://medium.com/visionari/google-annuncia-la-sua-nuova-funzionalit%C3%A0-semantica-per-interagire-con-i-libri-9be22003bd9e

[30] Document Retrieval. https://en.wikipedia.org/wiki/Document_retrieval

[31] Il Giornale.it, Maddalena Camera, 2013. https://www.ilgiornale.it/news/chiude-altavista-stato-primo-motore-ricerca-veloce-web-932099.html

[32] SQL LIKE Operator, https://www.w3schools.com/SQL/sql_like.asp#:~:text=The%20SQL%20LIKE%20Operator%20The%20LIKE%20operator%20is,percent%20sign%20represents%20zero%2C%20one%2C%20or%20multiple%20characters

[33] Welcome to Apache Lucene, https://lucene.apache.org/index.html

[34] Cosa si intende per ACID nei sistemi di database?, 2016 https://database.guide/what-is-acid-in-databases/

[35] Apache Lucene, https://www.appuntisoftware.it/apache-lucene/

[35] Procedura descritta da "An introduction to Elasticsearch" di Enrico Ghidoni, Big Data Management & Governance, a.y. 2019/2020

[36] Full-Text queries, https://www.elastic.co/guide/en/elasticsearch/reference/current/full-text-queries.html#full-text-queries

[37] SO YOU THINK YOU CAN SEARCH – COMPARING MICROSOFT SQL SERVER FTS AND APACHE LUCENE, Yan Roginevich, 2013, https://www.dbbest.com/blog/lucene-vs-sql-server-fts/

[38] Controlla la tua applicazione Spring Cloud con Elastic Stack (a.k.a ELK)

https://codingjam.it/controlla-la-tua-applicazione-spring-cloud-con-elastic-stack-a-k-a-elk-parte-1/

[39] Configuring Logstash: https://www.elastic.co/guide/en/logstash/current/configuration.html

[40] Il Full-Text Search di SQL Server, Marco Minerva, 2008, Il Full-Text Search di SQL Server | HTML.it