Elasticsearch - il motore distribuito di ricerca e analytics

Yuqi Sun
yuqi.sun@studio.unibo.it

February 25, 2024

Il progetto ha lo scopo di studiare e illustrare le principali funzionalità di Elasticsearch [1]. Elasticsearch è un motore di ricerca e di analytics distribuito e *RESTful*, basato sulla libreria Apache Lucene [2] e in grado di lavorare su qualsiasi tipo di dato, che sia numerico, strutturato o non strutturato. Sviluppato da Shay Banon, il primo precursore di Elasticsearch, chiamato *Compass*, risale al 2004, mentre la sua prima versione è stata rilasciata 6 anni dopo, nel 2010; ad oggi, secondo il DB-Engines, Elasticsearch risulta il motore di ricerca più popolare in contesti enterprise [3].

Contents

1 Goals e Requisiti

Come afferma il sito ufficiale, Elasticsearch "memorizza i dati per una ricerca fulminea, un'alta rilevanza dei risultati e potenti analisi che scalano con facilità", dunque il progetto illustrerà il funzionamento generale e le funzionalità di Elasticsearch, ponendo particolare attenzione ai suoi aspetti distribuiti. In particolare, lo scopo è comprendere:

- Apache Lucene, essendo la libreria su cui Elasticsearch è basato;
- la modalità di memorizzazione dei documenti:
  - struttura dati usata per l'indicizzazione;
  - architettura usata per la distribuzione dei documenti attraverso i nodi di un cluster;
  - monitoraggio e affidabilità del cluster (*cross-cluster replication*, *snapshots*);
- come vengono recuperate le informazioni dai dati:
  - tipi di *query* supportate (Query DSL, EQL Search);
  - modalità di ricerca disponibili (*searchable snapshots*, ricerca asincrona);
  - presentazione dei risultati (aggregazioni, Kibana).

Inoltre si effettueranno prove in locale, testando le funzionalità di Elasticsearch nei limiti di hardware e software disponibili, per poi concludere riportando i vantaggi e gli svantaggi di questa tecnologia ed eventualmente confrontandola con le alternative esistenti.

1.1 Work Plan

Il piano è quello di comprendere Elasticsearch dapprima da un punto di vista teorico per capirne appieno il suo funzionamento. Successivamente, si passerà alla parte pratica, con l'installazione e configurazione di Elasticsearch sul proprio pc portatile e lo svolgimento di test che ne provi vantaggi e limitazioni, citando eventuali articoli che ne hanno studiato l'uso in caso i test locali siano risultati non sufficienti.

2 Elasticsearch

2.1 Introduzione a Elastic Stack

L'Elastic Stack è un insieme veloce e altamente scalabile di componenti che permettono di raccogliere in modo sicuro i dati da qualsiasi fonte, in qualsiasi formato, e di cercarli, analizzarli e visualizzarli. L'architettura completa comprende:

- Beat: *data shippers* installati nei server come agenti; inviano dati operazionali a Elasticsearch;
- Logstash: un motore di raccolta dati in grado di effettuate *pipeline* in tempo reale; uniforma e normalizza dati provenienti da sorgenti diverse;
- Kibana: analizza e visualizza i dati memorizzati su Elasticsearch;
- Elasticsearch: un motore di ricerca e di *analytics* ed elemento centrale di tutto lo stack.

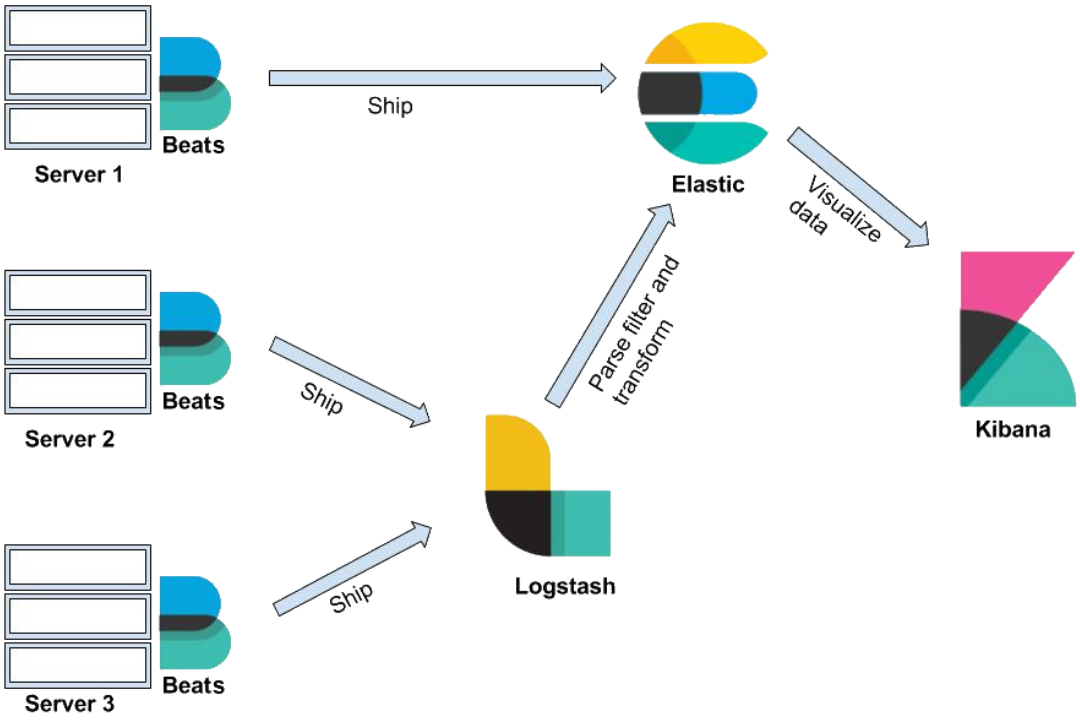


Figure 1: Schema dell'architettura di Elastic Stack.

2.2 Cos'è Elasticsearch - in sintesi

Elasticsearch è un motore di ricerca distribuito, scalabile e open-source, costruito sopra Apache Lucene, una delle librerie di motori di ricerca testuale più avanzata, performante e completa, ma anche molto complessa. Grazie a un'API REST semplice e coerente, Elasticsearch riesce a nascondere questa complessità; in più, fornisce funzionalità aggiuntive come API di monitoraggio dei nodi e del cluster, API di monitoraggio dei dati, gestione dei cluster ecc.

I dati gestiti da Elasticsearch possono essere molto complessi e contenere dati testuali, numeri, coordinate geografiche, array di valori e altri *object*. I dati sono serializzati come documenti JSON e ogni documento:

- è composto da campi e valori;
- ha un identificativo univoco;
- viene indicizzato con strutture dati diverse a seconda del tipo di dato dei campi (*inverted index*).

Documenti correlati tra di loro sono salvati insieme in un **indice**, un *namespace logico* che punta a uno o più *shard* fisici: uno **shard** è sempre un indice, indipendente e completamente funzionale, che contiene solo una parte di tutti i dati dell'indice; può essere **primario** o una **replica** e sono distribuiti tra i vari nodi del cluster.

Ogni documento in un indice appartiene a uno *shard* primario, mentre le repliche sono copie degli *shard* primari. Tutto ciò crea ridondanza: da una parte, permette a Elasticsearch di proteggersi da fallimenti di hardware e perdita di dati; dall'altra, permette di aumentare la capacità di *query*.

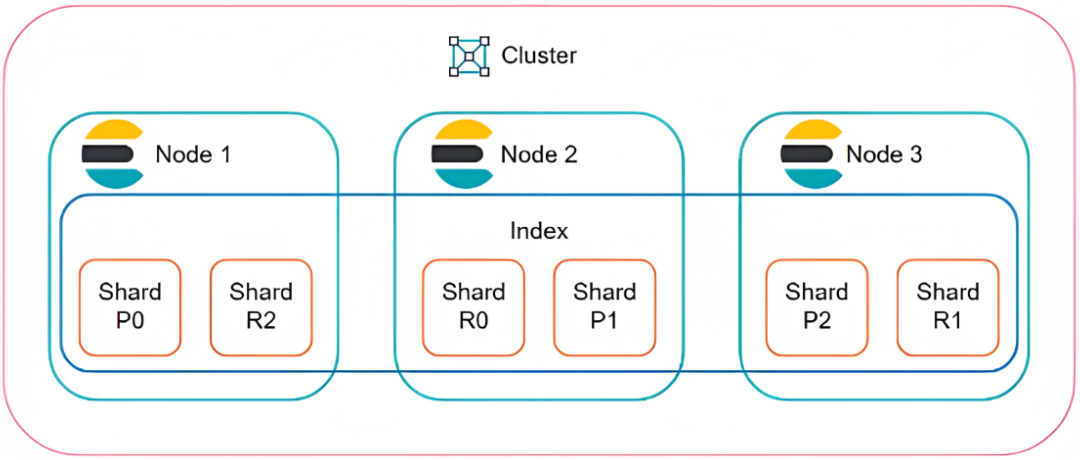


Figure 2: Elasticsearch: indici, shard, repliche.

Volendo fare un confronto con i tradizionali database relazionali, possiamo pensare al cluster come a un database, all'indice come a una tabella, al documento come alla riga e ai campi come alle colonne.

2.3 Apache Lucene

Apache Lucene è la libreria Java open-source sopra la quale è costruito Elasticsearch. Quando Elasticsearch crea uno *shard*, non crea altro che un indice Lucene.

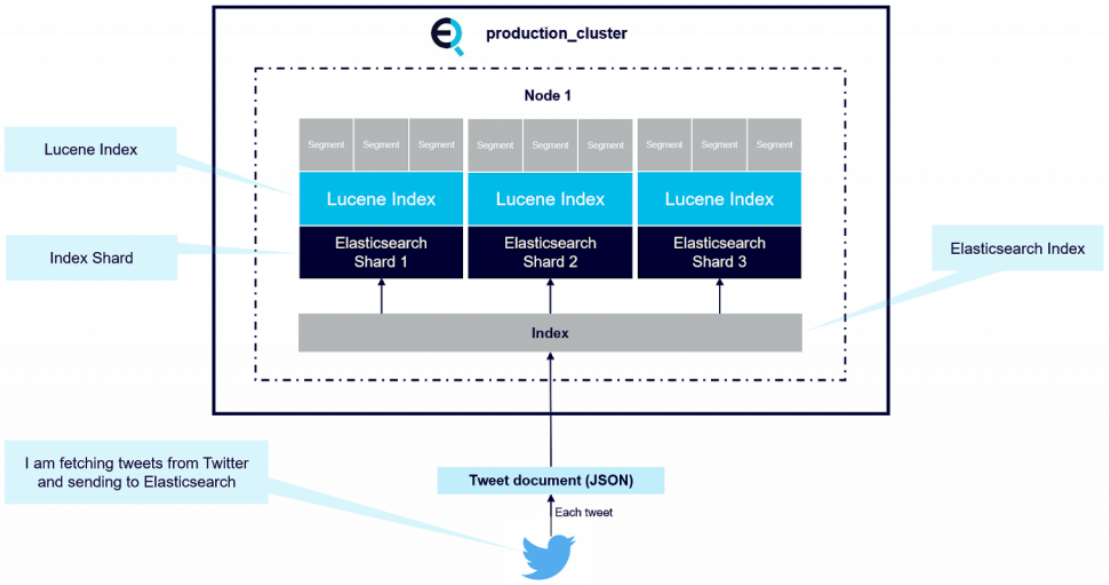


Figure 3: Elasticsearch e Apache Lucene.

2.3.1 Documenti e indici

Abbiamo detto che possiamo pensare all'**indice** come a un insieme ottimizzato di documenti, assimilabile concettualmente a una tabella di un database relazionale. In realtà, esistono delle differenze importanti: mentre le tabelle necessitano di uno schema, definito al momento della sua creazione, e possono avere vincoli sulla definizione della chiave primaria, delle colonne ecc, in un indice Lucene tali vincoli non esistono.

Indipendentemente dal suo contenuto o dalla presenza di uno schema o meno, Lucene è in grado di indicizzare un documento.

Il termine "indice" può avere diversi significati:

- quando si parla di un indice di Elasticsearch, intendiamo un luogo logico dove sono memorizzati i dati; nella pratica, sono un insieme di *shard*, ovvero un'istanza di Lucene;
- un indice Lucene è un motore di ricerca completo a sé stante ed è dove sono fisicamente memorizzati i dati;
- "indicizzare un documento" vuol dire memorizzare un documento in un indice Lucene o di Elasticsearch;
- nella sua forma passiva, come in "il campo viene indicizzato con un un *inverted index*", intendiamo che il campo ha un *inverted index*.

2.3.2 Campo

Ogni documento scritto in un indice Lucene ha una sequenza di numeri (**DocID**) che lo identifica univocamente. Il contenuto di un documento è una lista semplice di coppie chiave-valore di un certo tipo (stringa, numero ecc): mentre Elasticsearch supporta anche campi complessi, di tipo *object*, in Lucene questi campi vengono "appiattiti".

```
"user": {
  "type": "object",
  "properties": {
    "id": { "type": "string" },
    "gender": { "type": "string" },
    "age": { "type": "long" },
    "name":   {
      "type": "object",
      "properties": {
        "full": { "type": "string" },
        "first": { "type": "string" },
        "last": { "type": "string" }
      }
    }
  }
}
```

Listing 1: Esempio di schema di un documento in Elasticsearch

```
{
  "user.id": [@johnsmith],
  "user.gender": [male],
  "user.age": [26],
  "user.name.full": [john,smith],
  "user.name.first": [john],
  "user.name.last": [smith]
}
```

Listing 2: Come lo stesso schema in Elasticsearch viene convertito in Lucene

Un campo ha tre attributi principali:

- il *nome* del campo;
- il *valore* del campo;
- il *tipo* del campo.

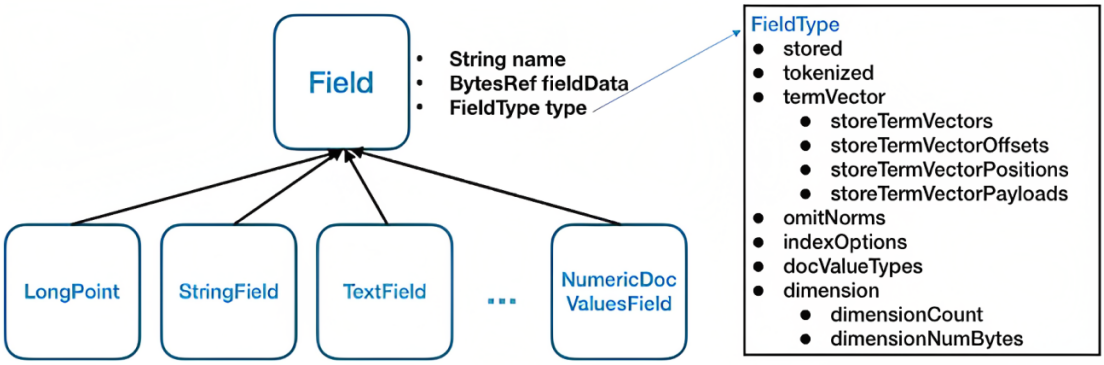


Figure 4: I possibili tipi di campo e i loro attributi in Lucene.

L'attributo *tipo* contiene a sua volta molti attributi i cui valori determinano come il campo viene indicizzato. Alcuni attributi sono:

- *tokenized*: indica se effettuare o meno il processo di *tokenization*; in Lucene, solo i campi testuali devono subire questo processo;
- *docValuesType*: è un indice di tipo *forward* introdotto in Lucene 4.0, migliorando l'efficienza di ordinamento, *faceting* — la suddivisione dei risultati della *query* in categorie in base ai termini indicizzati — e aggregazione dei risultati; essendo una struttura caratterizzata da uno schema forte, tutti i campi con *DocValues* abilitato devono avere esattamente lo stesso tipo;
- dimensione: Lucene supporta l'indicizzazione di dati multidimensionali, impiegando un'indicizzazione speciale per ottimizzarne le *query*; uno scenario tipico sono le coordinate geografiche.

2.3.3 Segmento

Un indice Lucene è una collezione di più indici, chiamati **segmenti**, e un *commit point*, un file che elenca tutti i segmenti. È qui che avviene la memorizzazione vera e propria dei documenti.

- Quando Lucene scrive dati, li scrive prima in un buffer in memoria.

- Quando i documenti nel buffer raggiungono un certo numero, vengono cancellati e scritti su disco in un nuovo segmento (**flushing**). Più precisamente:
  - Il nuovo segmento viene scritto dapprima in cache e solo in un secondo momento su disco.
  - Su disco viene scritto anche un nuovo *commit point*, dove viene incluso l'ultimo segmento creato.
- Il buffer viene svuotato, in attesa di nuovi documenti da scrivere.

Finché i documenti sono ancora nel buffer, non è possibile cercali. È per questo motivo che Lucene viene considerato come un motore di ricerca in tempo *quasi* reale: la ricerca dei dati si basa sull'*inverted index* e altre strutture dati simili, che però viene impostato durante la creazione del segmento e non in tempo reale durante la scrittura stessa dei dati. Proprio per migliorare le performance, la scrittura dei segmenti avviene in due fasi distinte: prima in cache, che è più veloce e viene quindi eseguita più frequentemente, e solo dopo su disco, che è invece un'operazione più costosa ma necessaria per persistere i dati e non perderli in caso di fallimento.

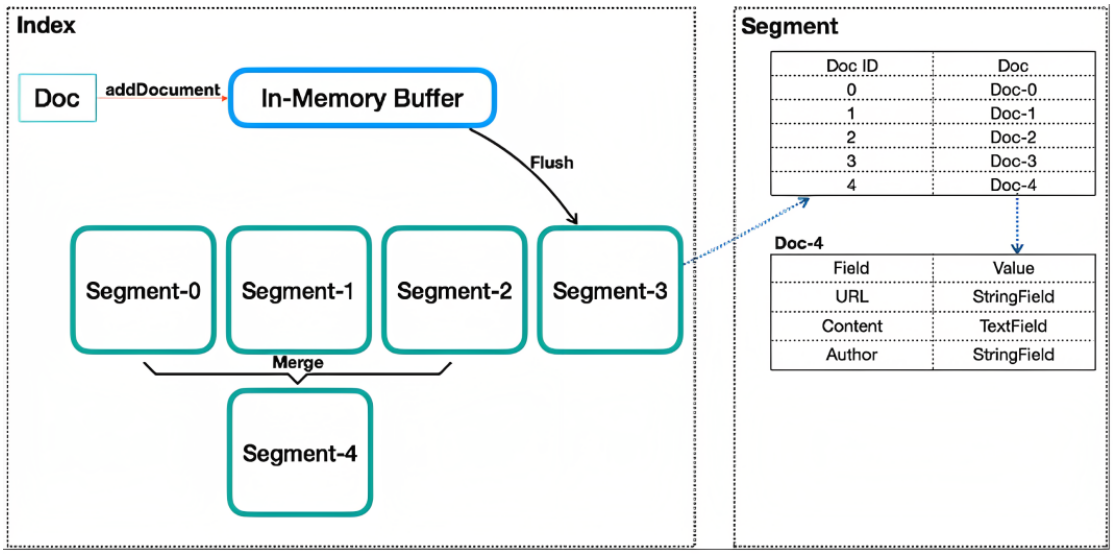


Figure 5: Schema riassuntivo della scrittura dei documenti.

I segmenti e i documenti contenuti in essi sono immutabili. Per effettuare cancellazioni o modifiche, ogni *commit point* include un file *.del* che elenca quali documenti in quali segmenti sono stati cancellati.

- Cancellare un documento vuol dire quindi che viene semplicemente etichettato come "cancellato" nel file *.del*.

- In modo analogo, modificare un documento vuol dire segnare come "cancellato" la versione vecchia, mentre la nuova versione viene indicizzata in un nuovo segmento.

Quando un *client* esegue una *query* su un indice Elasticsearch, devono essere interrogati tutti i segmenti rilevanti. Il numero di segmenti però può essere molto alto, e interrogare ognuno di essi potrebbe rallentare la ricerca. Per migliorare le prestazioni, man mano che avvengono i vari processi di scrittura e ricerca, in background Elasticsearch unisce i segmenti più piccoli in segmenti più grandi (**merge**).

- Durante l'unione dei segmenti i documenti che sono stati precedentemente cancellati vengono ignorati.
- I segmenti più piccoli vengono cancellati, mentre quello nuovo viene scritto su disco, assieme a un nuovo *commit point*.

2.3.4 DocID

Un indice Lucene identifica in modo univoco un documento tramite il suo DocId.

- Un DocId non è in realtà univoco per indice, ma è univoco per segmento.
  - Ciò viene fatto per ottimizzare la scrittura e la compressione.
  - I DocId sono numerati progressivamente a partire da 0, ma non sono necessariamente continui: quando viene cancellato un documento, il suo DocId viene a mancare.
  - Il DocId corrispondente a un documento può cambiare, di solito quando vengono uniti tra loro più segmenti.
- Per identificare in modo univoco un documento a livello di indice, i segmenti vengono ordinati.
  - Per esempio, abbiamo un indice con due segmenti e ogni segmento ha rispettivamente 100 documenti. I DocId del segmento sono compresi tra 0 e 100, ma quando vengono convertiti a livello di indice, l'intervallo dei DocId del secondo segmento viene convertito in 100-200.

2.4 Indicizzazione

Elasticsearch può ritornare risultati diversi a seconda del campo su cui vogliamo fare una *query*: per ciascuno dei tipi principali — stringhe, numeri, booleani e date — esistono piccole differenze nel modo in cui sono indicizzati, ma la differenza maggiore è tra i campi che rappresentano valori esatti, come una data o un ID utente, e i campi che rappresentano testo completo, solitamente in linguaggio umano come il testo di un'e-mail.

I valori esatti sono facili da interrogare, in quanto un valore o corrisponde alla *query* oppure non corrisponde; l'interrogazione dei dati testuali, invece, è meno diretto perché va a cercare all'interno dei campi di testo. Per fare ciò, Elasticsearch analizza prima il testo e poi utilizza il risultato per costruire un *inverted index*.

2.4.1 Inverted index

Un **inverted index** elenca ogni parola univoca che appare in un qualsiasi documento e per ciascuna parola identifica i documenti in cui appare.

Illustriamo un esempio molto semplice di *inverted index*. Supponiamo di avere due documenti contenenti le seguenti frasi:

1. The quick brown fox jumped over the lazy dog.
2. Quick brown foxes leap over lazy dogs in summer.

Dopo aver separato le frasi nelle loro parole distinte, chiamate anche *token* o *termini*, costruiamo il nostro *inverted index*.

```
Terms      Doc_1   Doc_2
-------------------------
Quick    |       |  X
The      |   X   |
brown    |   X   |  X
dog      |   X   |
dogs     |       |  X
fox      |   X   |
foxes    |       |  X
in       |       |  X
jumped   |   X   |
lazy     |   X   |  X
leap     |       |  X
over     |   X   |  X
quick    |   X   |
summer   |       |  X
the      |   X   |
-------------------------
```

2.4.2 Analisi

Attualmente nell'indice appaiono diverse parole che sono considerate come entità separate ma che un utente potrebbe considerare semanticamente simili, come 'Quick' e 'quick', 'fox' e 'foxes' e 'jumped' e 'leap'. Se usassimo un algoritmo *naive* di similarità, che si limita a contare il numero di termini corrispondenti, e un utente eseguisse una *query* per trovare i documenti dove appaiono sia 'Quick' sia 'fox', l'algoritmo risponderebbe che non ne esiste nessuno.

```
Token     Doc_1   Doc_2
-------------------------
Quick    |       |  X
fox      |   X   |
```

Risulta quindi necessario effettuare una fase di *pre-processing*, dove sia i termini che la *query* vengono normalizzati a una forma standard, attraverso operazioni come:

- trasformazione dei termini in *lower case*. 'Quick' diventa 'quick';

- *lemmatization*: riduzione dei termini alla loro forma base. 'fox' e 'foxes' diventano 'fox'.
- equivalenza di sinonimi: 'jumped' e 'leap' possono essere indicizzati in un unico termine.

In Elasticsearch questa fase di *tokenization* e normalizzazione viene chiamata **analisi** ed è una fase necessaria affinché testo non strutturato sia correttamente convertito in testo strutturato e cercabile.

L'analisi viene effettuata durante l'indicizzazione dei documenti, qualora questi contengano campi testuali, e sulla stringa delle *query*.

2.4.3 Mapping

Affinché i campi di un documento siano gestiti nel modo corretto, Elasticsearch esegue il **mapping**, un processo di definizione delle modalità di memorizzazione e indicizzazione di un documento e dei campi in esso contenuti. Durante il *mapping*, viene creata una lista di tutti i campi pertinenti al documento, a cui si aggiungono campi metadata come:

- `\_id`: identificativo univoco, diverso da quello in Lucene;
- `\_index`: indica l'indice di Elasticsearch dove viene memorizzato il documento;
- `\_source`: il corpo del documento originale in JSON.

Il *mapping* può essere dinamico o esplicito. Nel *mapping* dinamico, l'utente è libero dal compito di definire quali sono i campi di un documento: sarà Elasticsearch che in automatico rileva quali sono i campi e il loro tipo. Nel *mapping* esplicito, è l'utente stesso a definire come deve avvenire il *mapping*, specificando, per esempio, quali campi stringa devono essere trattati come campo testuale, quali campi come valore esatto stringa, quali come coordinate geografiche ecc.

È possibile aggiungere uno o più campi a un indice già presente, ma non si può cambiare un campo esistente: in quest'ultimo caso, sarà necessario creare un nuovo indice con il *mapping* aggiornato e re-indicizzare i dati. Bisogna anche fare attenzione a non inserire nello stesso indice documenti che hanno schema completamente diversi perché interferirebbe con l'abilità di Lucene di memorizzare i documenti in modo efficiente.

2.4.4 Altri indici

Oltre all'*inverted index*, vengono usate anche altre strutture dati. Alcuni esempi sono:

- **forward index**: è l'opposto di *inverted index*, ovvero memorizza i documenti e le parole contenute in esse;
- **BKD-tree**: è un albero che supporta dati multidimensionali, mantenendo alte le prestazioni in termini di ricerca e lettura; viene usato, per esempio, per le coordinate geografiche.

3 Dati: distribuzione, affidabilità

Elasticsearch amplia Apache Lucene aggiungendo distribuzione e scalabilità.

3.1 Nodo

Abbiamo detto che un cluster è formato da nodi su cui è in esecuzione un'istanza di Elasticsearch e che lavorano insieme per condividere i dati e il carico di lavoro. Un nodo può avere uno o più ruoli. I ruoli principali sono:

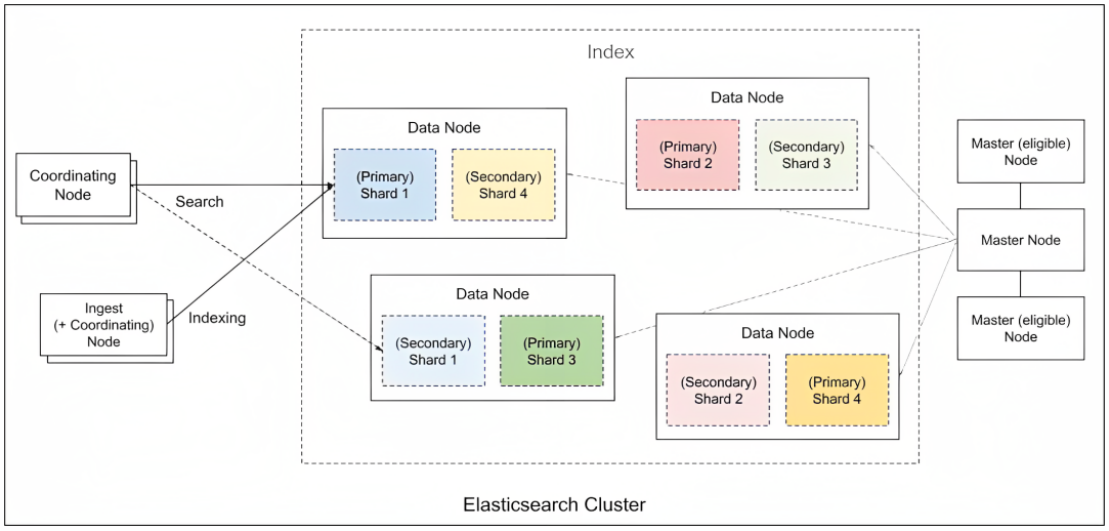
- **data node**: sono nodi che contengono gli *shard* e si occupano di operazioni come CRUD, ricerca e aggregazione.
  - Essendo operazioni costose in termini di I/O, memoria e CPU, è molto importante monitorare queste risorse e aggiungere ulteriori *data node* se necessario.
- **master node**: ha il compito di gestire il cluster, effettuando operazioni come creazione e modifica di un indice, aggiunta o rimozione di un nodo dal cluster, quali *shard* allocare a quali nodi ecc.
- **coordinating node**: è un nodo con il compito di instradare le richieste che il cluster riceve, gestire la fase di ricerca e distribuire l'indicizzazione dei dati.
- *ingest node*: è un nodo che pre-processa i documenti prima che questi vengano indicizzati.

Di default ogni nodo ha tutti questi ruoli. Finché un cluster è di piccole dimensioni e i nodi non hanno un carico di lavoro eccessivo, ogni nodo ha le risorse sufficienti per gestire ogni compito. Quando la dimensione del cluster e il carico di lavoro aumentano, può avere senso avere un *master node* o più nodi *master-eligible* che si occupano soltanto di gestire il cluster, lasciando le restanti responsabilità agli altri nodi: avere un *master node* stabile, infatti, è importante per avere un cluster che funziona bene.

3.1.1 Comunicazione

È possibile parlare con qualsiasi nodo del cluster, compreso il nodo *master*: ogni nodo conosce dove si trova ogni documento e può inoltrare la richiesta di un *client* direttamente ai nodi che possiedono i dati richiesti.

- Il nodo con cui il *client* inizia la comunicazione gestisce l'intero processo di raccolta delle risposte dai nodi e restituire la risposta finale al *client*.
- La comunicazione avviene interamente tra *client* e indice e mai direttamente con gli *shard*, dove invece sono effettivamente archiviati e indicizzati i documenti.



3.2 Repliche

Il modello di replicazione dei dati è basato sul *primary-backup model*:

- una singola copia fa da *shard* primario mentre le altre copie sono repliche;
- la replica è allocata in un nodo diverso dal corrispondente *shard* primario;
- lo *shard* primario sarà il punto di ingresso principale per tutte le operazioni di indicizzazione e si assicurerà che queste siano corrette;
- accettata un'operazione di indicizzazione, lo *shard* primario avrà il compito di replicare l'operazione anche sulle altre copie.

Qualunque operazione di scrittura deve essere completata sullo *shard* primario prima di essere copiata sulle repliche. Di seguito si elencano i passi per effettuare operazioni di scrittura e recupero di un documento su un esempio di cluster formato da tre nodi.

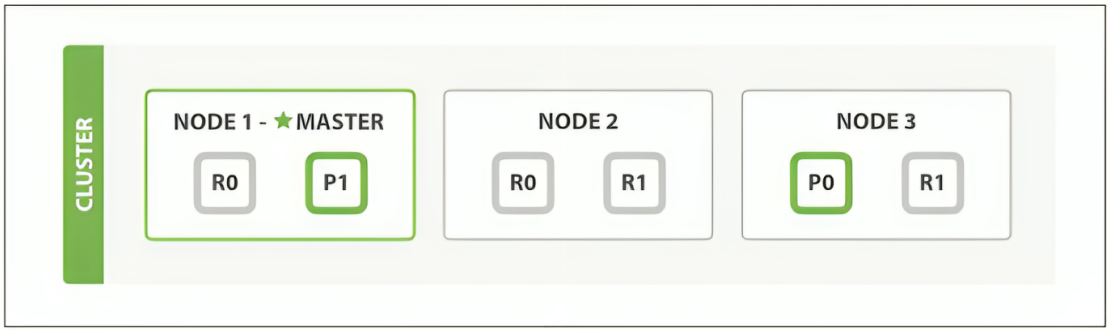
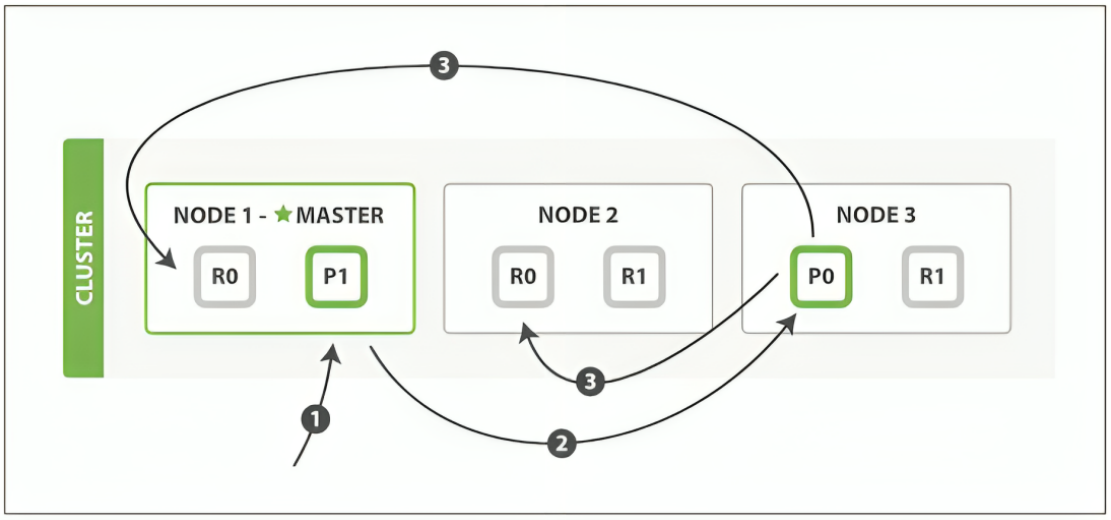


Figure 6: Un semplice cluster composto da un indice e tre nodi.

3.2.1 Creare, indicizzare e cancellare un documento

1. Il *client* invia una richiesta di scrittura al Nodo 1.
2. Il nodo utilizza l'id del documento per determinare che il suo shard primario è P0 e inoltra la richiesta al nodo 3. Se invece è la prima creazione, identifica P0 come *shard* primario dove inserire il nuovo documento.
3. Il nodo 3 esegue la richiesta sullo shard primario. Se l'esito è positivo, inoltra la richiesta in parallelo alle repliche sul nodo 1 e sul nodo 2. Una volta che tutte le repliche riportano un esito positivo, il nodo 3 riporta l'esito positivo al nodo richiedente, che lo riporterà al *client*.



3.2.2 Recuperare un documento

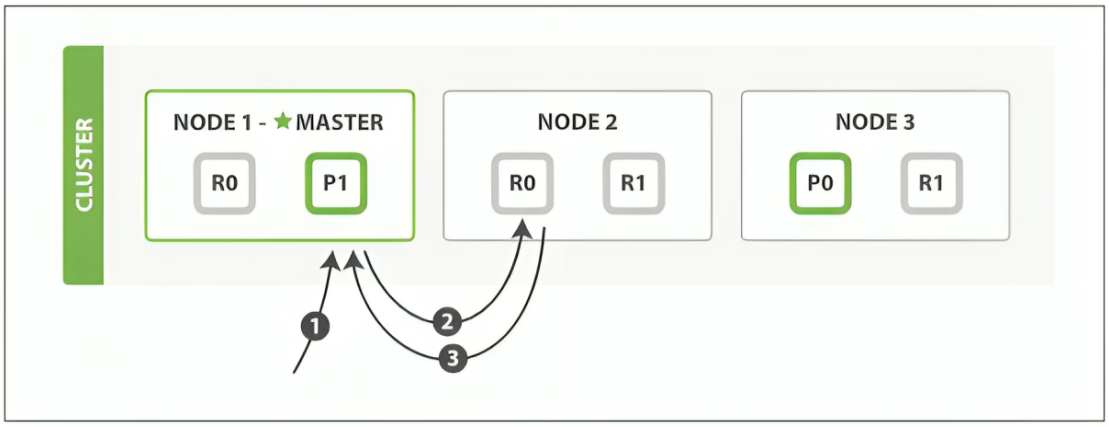
Un documento può essere recuperato da uno *shard* primario o da una qualsiasi delle sue repliche.

1. Il *client* invia una richiesta al nodo 1.
2. Usando l'id del documento, il nodo determina che il documento appartiene allo *shard* P0. Copie dello *shard* P0 esistono su tutti e tre i nodi. In questo caso, inoltra la richiesta al nodo 2.
3. Il nodo 2 restituisce il documento al nodo 1, che lo restituisce al *client*.

Per bilanciare il carico di lavoro, il nodo sceglie uno *shard* diverso per ogni richiesta, con una modalità *round-robin*.

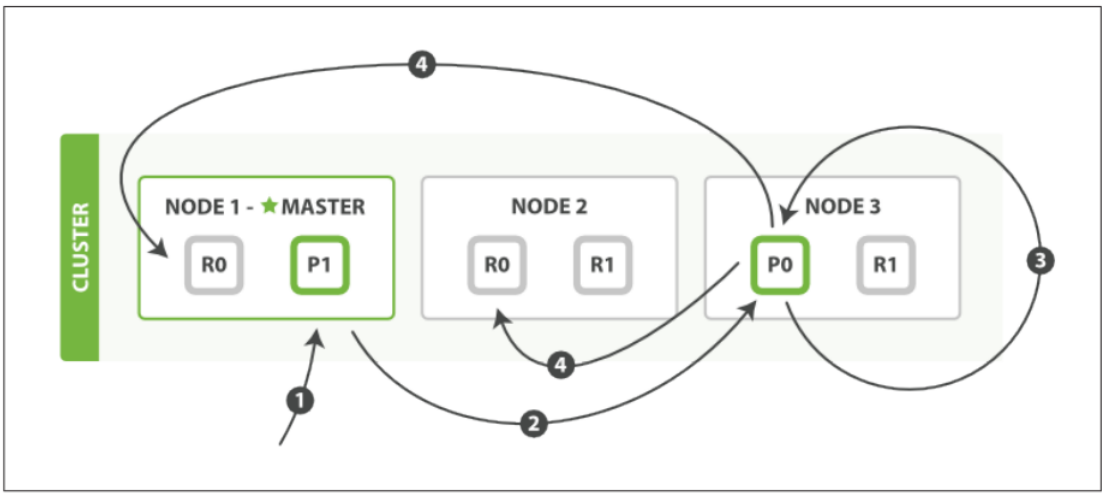
È possibile che, durante l'indicizzazione di un documento, questo sia già presente sullo *shard* primario ma non sia stato ancora copiato sugli *shard* di replica. In questo caso,

una replica potrebbe rispondere al client che il documento non esiste, mentre lo *shard* primario avrebbe restituito il documento con successo. Una volta che la richiesta di indicizzazione ha dato esito positivo all'utente, il documento sarà disponibile sullo *shard* primario e su tutti gli *shard* di replica.



3.2.3 Aggiornare un documento

1. Il *client* invia una richiesta di modifica al nodo 1.
2. Il nodo 1 inoltra la richiesta al nodo 3, dove è collocato lo *shard* primario P0.
3. Il nodo 3 recupera il documento da P0, applica i cambiamenti richiesti e prova a reindicizzare il documento.
   a) Se nel frattempo il documento è già stato cambiato, il nodo 3 riprova a indicizzare il documento fino a un certo numero di tentativi oppure informare il *client* che non è stato possibile applicare la modifica.
4. Se il nodo 3 è riuscito a modificare il documento con successo, invia in parallelo il nuovo documento ai nodi 1 e 2. Se anche il nodo 1 e 2 riportano un esito positivo, il nodo 3 inoltra il risultato al nodo 1, che lo invierà al *client*.
   a) Poiché in Elasticsearch i documenti sono immutabili, aggiornare un documento significa di fatto sostituire quello vecchio, che viene prima etichettato come "cancellato" e rimosso solo in un momento successivo durante un *merge* dei segmenti.



3.2.4 Gestione dei conflitti

Essendo distribuito, quando un documento viene creato, modificato o cancellato, la nuova versione del documento deve essere replicata agli altri nodi del cluster. Elasticsearch è anche asincrono e concorrente: queste richieste di replica sono inviate in parallelo e possono arrivare a destinazione fuori sequenza.

Per garantire che la versione più vecchia di un documento non sovrascriva erroneamente una versione più recente, Elasticsearch aggiunge ai metadati il campo *_version*, un numero che viene incrementato a ogni modifica effettuata su un documento: se una versione più vecchia arriva dopo una versione più nuova, questa viene semplicemente ignorata.

```
{
  "_index" : "website",
  "_type" : "blog",
  "_id" : "123",
  "_version" : 1,
  "_source" : {
    "title": "My first blog entry" ,
    "text": "Just trying this out..."
  }
}
```

Listing 3: Un esempio di documento alla sua prima creazione

```
{
  "_index" : "website",
  "_type" : "blog",
  "_id" : "123",
  "_version" : 2,
  "_source" : {
    "title": "My first blog entry" ,
    "text": "I am starting to get the hang of this..."
  }
```

```
}
```

Listing 4: Lo stesso esempio dopo essere stato aggiornato

Si può sfruttare il campo *_version* anche per garantire che eventuali modifiche in conflitto tra di loro non comportino la perdita di dati:

1. quando si invia la richiesta, si specifica la versione del documento a cui bisogna applicare la modifica;
2. se il documento corrente ha la stessa versione presente nella richiesta, si procede con la modifica e reindicizzazione del nuovo documento;
3. diversamente, se le due versioni non combaciano, Elasticsearch risponde con un messaggio di errore; successivamente si può informare il *client* che non si è potuto applicare la modifica, o recuperare automaticamente l'ultima versione e riprovare di nuovo oppure gestire il conflitto in un altro modo a seconda del caso d'uso.

Questo approccio adottato da Elasticsearch è detto **optimistic locking**. Nei contesti di un sistema ad alto *throughput*, adottare un *locking* tradizionale può risultare un'operazione superflua e troppo pesante; con il *versioning* dei documenti, eventuali conflitti vengono semplicemente segnalati.

3.2.5 Routing di un documento al shard

Per determinare a quale *shard* appartiene un documento si usa la formula:

```
    shard = hash(routing) % number_of_primary_shards
```

Il valore di *routing* è una stringa arbitraria, che per impostazione predefinita corrisponde all'id del documento, ma può anche essere impostata su un valore personalizzato. Questa stringa di *routing* viene passata attraverso una funzione di hashing per generare un numero, che viene diviso per il numero di frammenti primari nell'indice per restituire il resto. Il resto sarà sempre compreso tra 0 e numero di *shard* primari - 1, ovvero il numero dello *shard* in cui risiede un determinato documento.

È per questo motivo che il numero di *shard* primari può essere impostato solo al momento della creazione di un indice e non deve mai essere modificato: se il numero di *shard* primari cambiasse, tutti i valori precedenti non sarebbero validi e i documenti non verrebbero trovati più.

3.3 Dimensionamento dei shard

Un documento appartiene a un solo *shard* primario, di conseguenza il numero di *shard* primari determinano la quantità massima di dati che un indice può contenere: questo numero è fissato al momento della creazione di un indice, mentre il numero delle repliche può essere cambiato in qualsiasi momento.

Ogni *query* viene eseguita in un singolo *thread* per *shard*; tuttavia, più *shard* possono essere elaborati in parallelo, così come più *query* e aggregazioni distinte sullo stesso

shard. Il numero e le dimensioni ottimali di uno *shard* dipendono dal proprio caso d'uso: l'hardware disponibile, la dimensione e la complessità dei documenti, il modo in cui si indicizzano e si interrogano i documenti e i tempi di risposta previsti. In ogni caso, Elasticsearch ribilancia automaticamente gli *shard* dell'indice tra i nodi ogni volta che vengono aggiunti o rimossi dei nodi.

Alcuni aspetti da considerare quando si decide come dimensionare gli *shard* sono:

- maggiore è il numero di *shard*, maggiore è il numero di copie dei dati e, di conseguenza, il *throughput* di ricerca che il cluster è in grado di gestire;
- aumentare *shard* implica meno risorse da allocare per ciascuno di essi e un maggiore overhead di gestione di indici e *shard*, con il rischio di rendere il cluster meno efficiente e stabile;
- avere molti *shard* piccoli rende l'elaborazione per *shard* più veloce, ma comporta avere più task in coda e in esecuzione e ciò può rallentare le perfomance della *query*;
- avere *shard* troppo grandi comporta invece tempi più lunghi di ricovero per copiare i dati su un nuovo nodo.

3.4 Affidabilità e disponibilità

Poiché un cluster può subire guasti di vario tipo, Elasticsearch offre diverse strategie per ottenere elevata disponibilità.

3.4.1 Design del cluster

Sistemi distribuiti come Elasticsearch sono progettati in modo da continuare anche quando falliscono alcuni dei loro componenti, purché ci siano sufficienti nodi ben connessi tra di loro. Come già descritto nelle sezioni precedenti, un cluster è composto da nodi che coprono diversi ruoli, al cui interno si trovano gli *shard*. Affinché un cluster di Elasticsearch più basilare possa essere considerato resiliente, deve avere:

- un nodo eletto a *master node* e almeno tre nodi *master-eligible*;
- almeno due nodi per ruolo;
- almeno due copie per ogni *shard*.

La ridondanza di nodi e *shard* permette al cluster di avere sempre almeno un nodo per ruolo e una copia dei dati.

3.4.2 Replicazione cross-cluster

Quando il cluster è molto grande, ha senso raggruppare nodi che condividono la stessa infrastruttura in *zone* e pianificare come gestire il fallimento di un'intera *zona*. Una possibile soluzione è localizzare tutte le zone in un unico data center, con ciascuna zona dotata con una propria infrastruttura e alimentazione indipendente, oppure separarli in *data center* distinti ma vicini, purché la connessione di rete tra le zone sia di qualità sufficientemente elevata.

Se invece i *data center* disponibili sono lontani o non ben collegati tra di loro, è possibile usare la **replicazione cross-cluster**: questa può essere usata sia per continuare a gestire le richieste di ricerca in caso di interruzione del *data center* principale, ma anche per ridurre la latenza elaborando le richieste di ricerca per prossimità dell'utente.

La replicazione *cross-cluster* usa un modello attivo-passivo:

- i dati vengono indicizzati su un indice *leader* e poi copiati su uno o più indici *follower* di sola lettura;
- un indice *follower* ha gli stessi *shard* dell'indice *leader*;
- quando l'indice *leader* riceve una scrittura su uno *shard*, l'indice *follower* fa pull della modifica e la applica sullo *shard* corrispondente nell'altro cluster;
- solo l'indice *leader* può scrivere ed è quindi detto "attivo", mentre un indice *follower* può solo servire richieste di lettura e ricerca ed è quindi detto "passivo".

È possibile configurare la replicazione *cross-cluster* in due modalità:

- unidirezionale: un cluster contiene solo indici *leader* e l'altro solo indici *follower*;
  - il caso uso più frequente è il recupero dei dati, copiando i dati dal *data center* di produzione a uno o più *data center*;

Figure 7: Configurazione con un cluster di back-up.

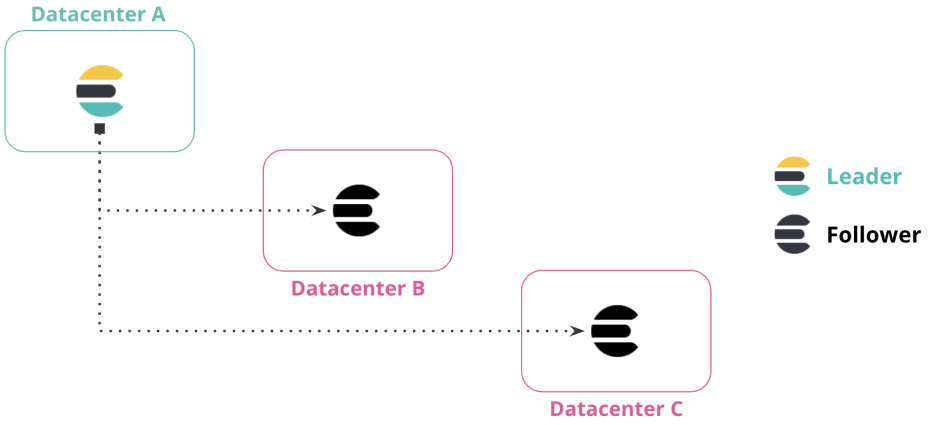


Figure 8: Configurazione con due cluster di back-up.

  - un altro caso uso è la *data locality*, ovvero portare i dati geograficamente più vicino ai *client*;

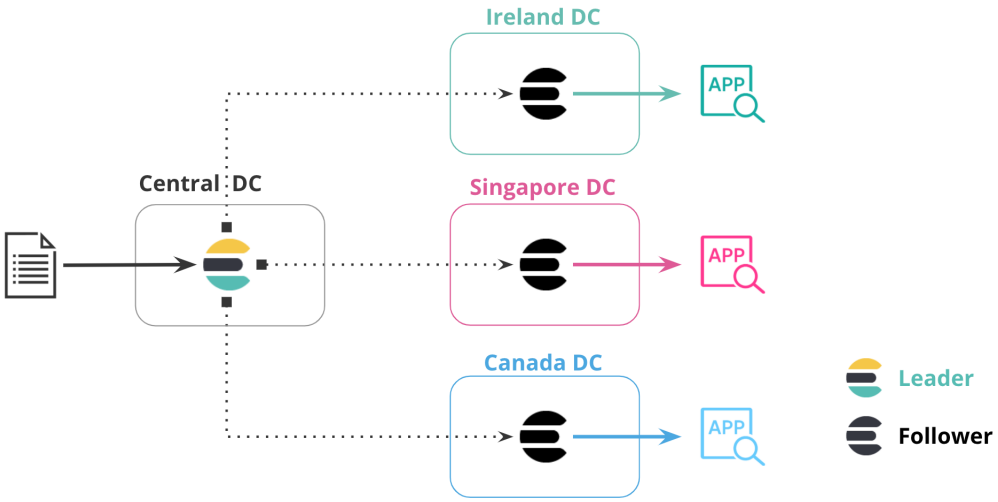


Figure 9: Configurazione per l'uso di *data locality*.

- bidirezionale: ogni cluster contiene entrambi i tipi di indici;
  - ciascun cluster scrive sui propri indici, ma ha accesso anche a tutti i dati grazie agli indici *follower*;
  - questa configurazione è utile quando il carico di lavoro comprende solo l'indicizzazione dei dati, senza che vengano apportati modifiche ai valori, permettendo al cluster di avere una visione intera di tutti i dati.

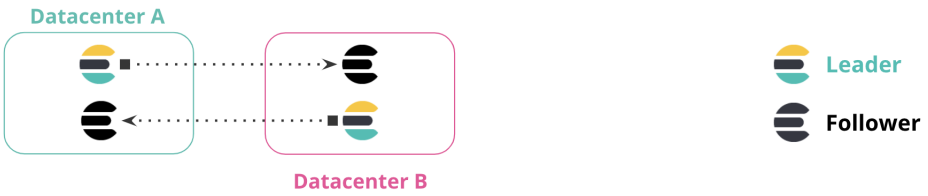


Figure 10: Configurazione bidirezionale.

3.4.3 Snapshot

Uno **snapshot** è un backup del cluster:

- è necessario definire un file, uno *blob storage system* o altra *repository* di archiviazione su cui scrivere gli *snapshot*;
- è possibile copiare tutti gli indici e i metadati del cluster, o solo una parte;
- tutti i dati indicizzati dopo che è stato fatto partire il processo di *backup* non vengono inclusi;
- è possibile eseguire un solo *snapshot* alla volta;
- vengono copiati solo gli *shard* primari.

Poiché gli segmenti che compongono gli *shard* sono immutabili, basta copiare soltanto quelli che non sono stati ancora scritti nella repository. Ciò rende gli *snapshot* di Elasticsearch incrementali.

3.4.4 Vantaggi e svantaggi

Avere uno *snapshot* è molto utile, in quanto rende possibile ripristinare un nuovo cluster in caso il cluster principale e quelli secondari dovessero fallire; d'altra parte, sia creare uno *snapshot* sia ripristinare un cluster da uno di essi richiede tempo.

La replicazione *cross-cluster*, invece, è in tempo reale, ma richiede maggiori configurazioni degli indici e, a differenza degli *snapshot*, richiede il pagamento di una licenza.

3.5 Monitoraggio del cluster

È possibile monitorare molte caratteristiche del cluster. È consigliato salvare queste informazioni in un cluster separato, in modo che fallimenti sul cluster centrale non neghino l'accesso ai dati di monitoraggio e per non impattare le performance con le attività di monitoraggio.

Per collezionare le statistiche del cluster e dei suoi componenti è possibile usare:

- **metricbeat**: una tipologia di Elastic Beat, un *data shipper* che colleziona metriche e le invia a un cluster;

- un **agente Elasticsearch**: un agente che colleziona log, metriche e altri tipi di dati.

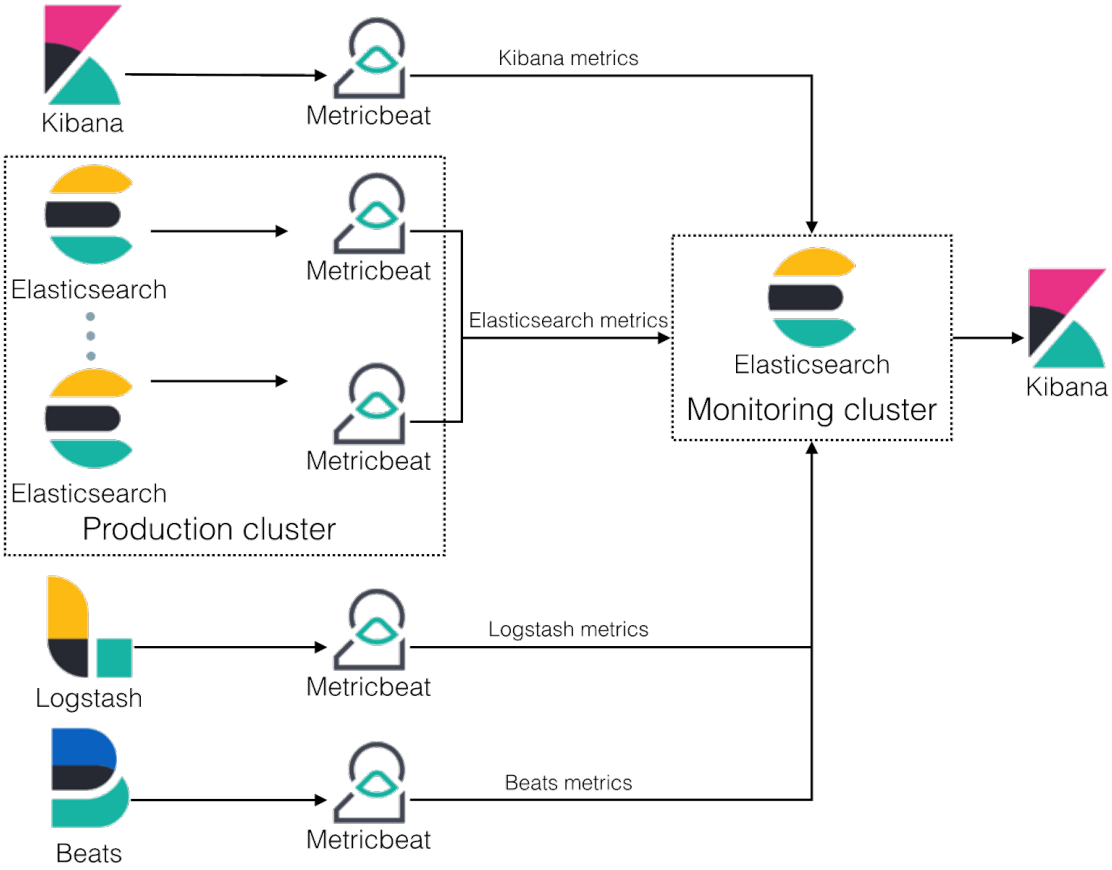


Figure 11: Architettura di monitoraggio usando Metricbeat

Alcuni aspetti che possono essere monitorati sono:

- metriche generali sul cluster come:
  - stato del cluster: può essere "verde" se tutti gli *shard* primari e repliche sono attivi; "giallo" se tutti gli *shard* primari sono attivi ma non tutte le repliche, "rosso" se non tutti gli *shard* primari sono attivi;
  - numero di nodi, indici, *shard* e documenti;
- metriche sui nodi come:
  - statistiche sull'uso di memoria, disco e CPU;
  - numero di richieste in esecuzione o in attesa;
  - statistiche degli indici salvati nel nodo;

- metriche sugli indici come:
  - performance sull'indicizzazione e sulla ricerca;
  - numero di segmenti.

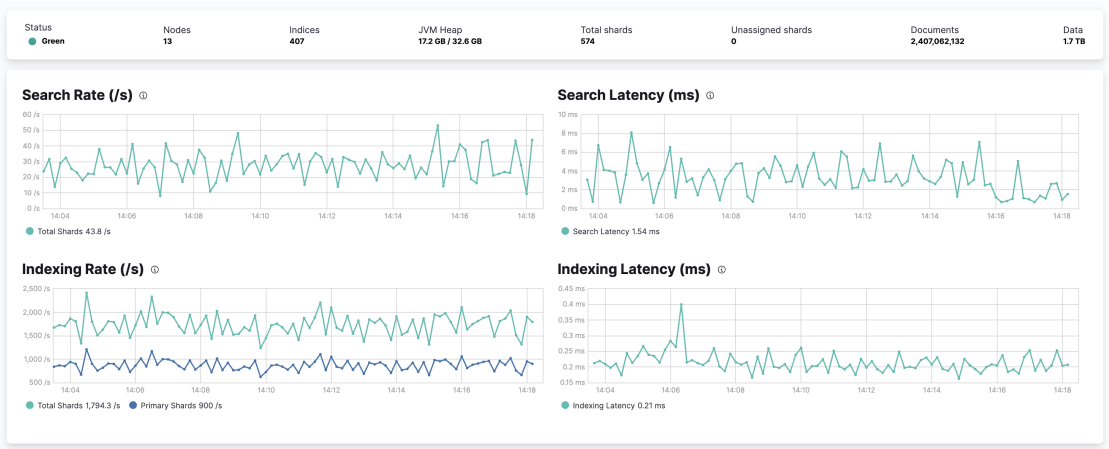


Figure 12: Esempio di alcune metriche mostrate su una dashboard di Kibana.

4 Ricerca e Analisi

Elasticsearch memorizza il documento e indicizza ogni suo campo, rendendolo interamente ricercabile.

Effettuare una ricerca vuol dire eseguire:

- una **query strutturata** su campi concreti come sesso o età, ordinati o raggruppati per un campo, in modo non dissimile da una query SQL;
- una **query testuale** che va a cercare tutti i documenti che corrispondono alle parole chiave della ricerca;
- una combinazione delle due opzioni.

Il risultato della ricerca è un elenco di documenti, nella loro interezza, ordinati per *_score*, un valore che indica quanto un documento è *rilevante*. Se l'utente non specifica nessun indice, la ricerca viene eseguita su tutti gli indici presenti nel cluster.

Nella pratica, eseguire una ricerca vuol dire usare l'API di ricerca offerta da Elasticsearch; esistono due modi per eseguire una query:

- una versione *lite*, dove tutti i parametri vengono passati nella stringa di ricerca;
  - per quanto permettano comunque di effettuare ricerche tutto sommato complesse, sono fragili e proni agli errori;
  - esempio: se volessimo cercare i tweet scritti da Mary John, successivi alla data 2014/09/10 e contenenti le parole "aggregations" o "geo" cercheremmo `GET /_all/tweet/_search?q=+name:(mary john) +date:>2014-09-10 +(aggregations geo)` che, codificata correttamente, diventa `GET /_all/tweet/_search?q=%2Bname%3A(mary+john)+%2Bdate%3A%3E2014-09-10+%2B(aggregations+geo);`
- una versione *full*, dove il corpo della richiesta è in formato JSON e usa un linguaggio di query chiamato DSL.

4.1 Query DSL

Query DSL è un linguaggio di ricerca SQL-Style flessibile ed espressivo che Elasticsearch utilizza per esporre le funzionalità di Lucene attraverso una semplice interfaccia JSON, rendendo le query più precise e più facili da leggere e da debuggare.

Una *query* è composta da **clausole**. Tipicamente, una clausola ha la seguente struttura:

```
{
  QUERY_NAME: {
    ARGUMENT: VALUE,
    ARGUMENT: VALUE,
    ...
  }
}
```

Le clausole sono elementi semplici che possono essere combinati tra loro per creare *query* più complesse e possono essere:

- clausole *foglia* (come la clausola `match`), utilizzate per confrontare un campo (o più campi) con una stringa query;
- clausole *composte*, usate per combinare altre clausole; per esempio, una clausola `bool` consente di combinare altre clausole che devono (`must`) corrispondere, non devono (`must_not`) corrispondere o devono (`should`) corrispondere se possibile.

Alcune clausole importanti sono:

- `term` e `terms`: per filtrare per uno o più valori esatti;

```
{ "term": { "date": "2014-09-01" }}
{ "terms": { "tag": [ "search", "full_text", "nosql" ] }}
```

- `exists` e `missing`: per filtrare documenti dove il campo specificato presenta o non presenta valori;

```
{
  "exists": {
    "field": "title"
  }
}

```

- `bool`: per combinare più clausole tra di loro con logica booleana (`must`, `must_not`, `should`);
- `match`: per cercare su uno o più campi testuali;

```
{
  "bool": {
    "must": { "match": { "email": "business opportunity" }},
    "should": [
      { "match": { "starred": true }},
      { "bool": {
        "must": { "folder": "inbox" },
        "must_not": { "spam": true }
      }}
    ]
 }
}
```

Listing 5: Un esempio di query che cerca email il cui corpo contiene le parole “business opportunity” e che dovrebbero essere segnate come importanti o essere nell’inbox ma non nella cartella spam.

4.2 Altri linguaggi

Oltre a DSL, è possibile interrogare Elasticsearch tramite query SQL e **Event Query Language** (EQL).

EQL è un linguaggio di interrogazione per dati di serie temporali basati su eventi, come log e metriche. EQL consente di esprimere relazioni tra eventi: mentre molti linguaggi di query consentono di abbinare solo singoli eventi, EQL consente di abbinare una sequenza di eventi a diverse categorie di eventi e su diversi intervalli di tempo.

La sintassi di EQL è simile a quella di altri linguaggi di query comuni, come SQL.

4.3 Elaborazione dei risultati

Effettuare una ricerca su un'architettura distribuita richiede un modello di esecuzione più complicato rispetto alle richieste di scrittura e lettura descritte nel capitolo 3: poiché non sappiamo quali documenti soddisferanno la *query*, eseguire una ricerca vuol dire consultare una copia di ogni *shard* negli indici che ci interessano; trovati tutti i documenti, i risultati provenienti dai vari *shard* devono essere combinati in un unico elenco ordinato che verrà poi restituito in una "pagina" di risultati.

Il processo di ricerca può essere diviso in due fasi: *query* e *fetch*.

4.3.1 Fase di interrogazione - query

1. Un nodo riceve una richiesta di ricerca, diventando il nodo coordinatore di tutto il processo.
   a) Il nodo coordinatore crea una coda di priorità di dimensione n.
2. Il nodo coordinatore trasmette la richiesta a ciascun *shard* primario o sua replica.
3. Ogni *shard* recupera gli ID dei documenti che soddisfano la richiesta e li ordina in una propria coda locale; le code vengono poi inviate al nodo coordinatore, che unisce i vari risultati per ordinarli globalmente per *_score*.

Proprio come le richieste GET di documenti, le richieste di ricerca possono essere gestite da uno shard primario o da una qualsiasi delle sue repliche. Di conseguenza, maggiore è il numero delle repliche, ovviamente combinate anche con più hardware, maggiore è il throughput di ricerca: un nodo coordinatore inoltrerà le varie richieste per turno tra tutte le copie dello *shard*, distribuendo il carico di lavoro.

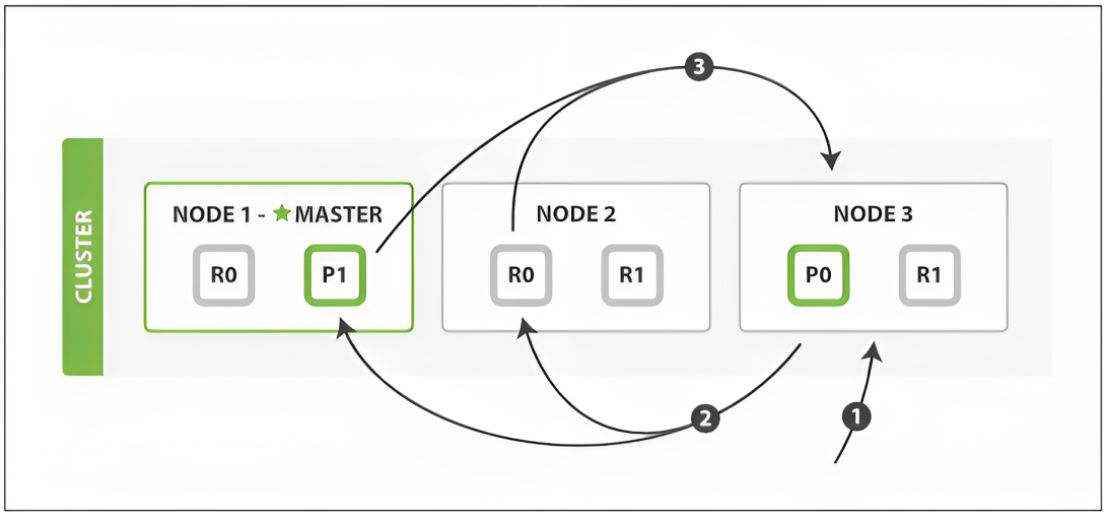


Figure 13: Esempio di interrogazione con nodo 3 come coordinatore.

4.3.2 Paginazione dei risultati

La dimensione della coda è determinata da due parametri che determinano la "pagina" dei risultati, *size* e *from*: *size* è il numero di risultati che si vuole ritornare al client, *from* è il numero di risultati iniziali da saltare; la dimensione n è data data $size + from$.

Bisogna fare attenzione a non paginare troppo profondamente, in quanto ogni *shard* produce i propri n risultati, che vengono poi inviati e ordinati dal nodo coordinatore. Supponiamo di avere un indice da 5 *shard* distinti:

- se chiediamo la prima pagina di risultati, da 1 a 10, il nodo coordinatore ordina i 50 risultati e seleziona i primi 10 ;
- se chiediamo la pagina 1000, corrispondente ai risultati da 10.001 a 10.010, il nodo coordinatore ordina $5 * 10010$ risultati, scartandone i primi 50 040.

In un sistema distribuito, il costo dell'ordinamento dei risultati cresce in modo esponenziale quanto più si va in profondità. A seconda delle dimensioni dei documenti, del numero di *shard* e dell'hardware utilizzato, la paginazione di 10.000-50.000 risultati (da 1.000 a 5.000 pagine) dovrebbe essere perfettamente fattibile.

4.3.3 Fase di recupero - fetch

Se la fase di interrogazione identifica i documenti che soddisfano la richiesta di ricerca, la fase di recupero recupera i documenti stessi.

1. Il nodo coordinatore determina quali documenti devono essere recuperati a seconda della paginazione specificata nella richiesta e invia una richiesta GET agli *shard*.

2. Gli *shard* recuperano i documenti e li invia al nodo coordinatore.

3. Infine il nodo coordinatore ritorna i risultati al *client*.

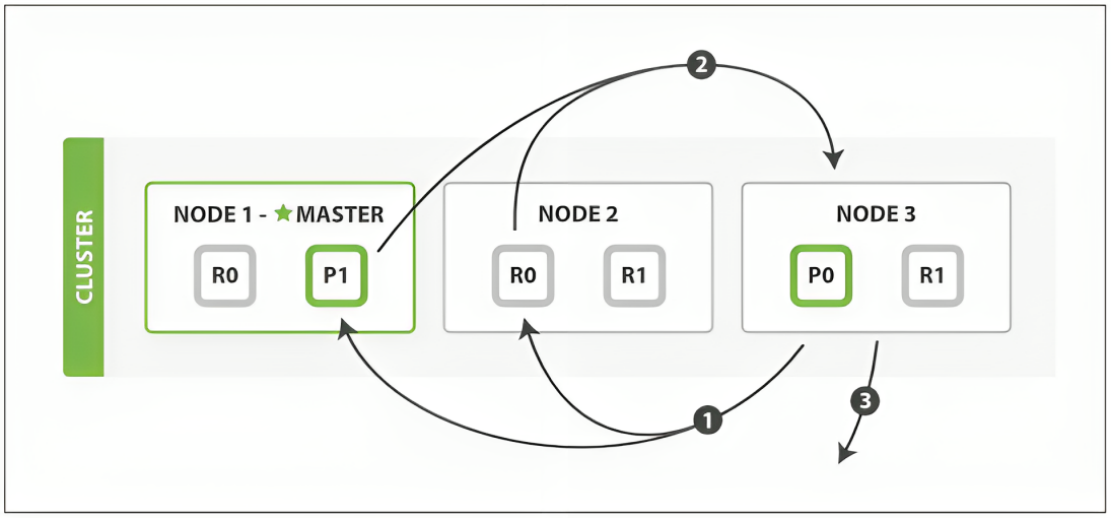


Figure 14: Esempio di recupero con nodo 3 come coordinatore.

4.4 Modalità di ricerca aggiuntive

4.4.1 Ricerca asincrona

Di default le ricerche effettuate su Elasticsearch sono sincrone: la richiesta rimane in attesa dei risultati prima di ritornare i risultati. Solitamente, i risultati, anche su grandi quantità di dati, vengono restituiti in millisecondi.

Quando è invece necessario eseguire ricerche di lunga durata, è possibile eseguirla in modalità asincrona, disponibile dalla versione 7.7, monitorare il suo progresso e recuperare i risultati in una fase successiva. In alternative, è anche possibile recuperare risultati parziali, man mano che diventano disponibili, ma prima che la ricerca sia completata.

4.4.2 Snapshot cercabili

Non tutti i dati sono archiviati e cercati allo stesso modo. Possiamo approssimare affermando che dati recenti vengano ricercati in modo intensivo, mentre altri dati possono essere conservati solo per motivi legali o di conformità, o per un occasionale controllo a scopo di confronto. Gli utenti hanno quindi bisogno di livelli diversi di archiviazione e di potenza di elaborazione per questi diversi livelli di esigenze, in base all'età, alla fonte dei dati o ad altri criteri.

Dalla versione 7.10, Elasticsearch ha reso gli *snapshot* cercabili, rendendoli una possibile alternativa di archiviazione per tutti quei dati che vengono acceduti poco frequente-

mente. Prima era necessario ripristinare manualmente lo *snapshot*, effettuare la richiesta e poi rimuovere nuovamente l'indice per liberare spazio.

4.5 Aggregazioni

A volte l'utente non vuole cercare singoli documenti, ma analizzare e riassumere l'intero set di dati. Ciò è possibile tramite le **aggregazioni**, un tipo particolare di *query* con le quali è possibile effettuare operazioni di *analytics*.

Esistono tre categorie di aggregazioni:

- *bucket*: collezioni di documenti che soddisfano un criterio;
- *metrics*: statistiche calcolate sui documenti di un bucket;
- *pipeline*: prendono in input risultati di altri *bucket*.

4.5.1 Bucket

Un *bucket* è semplicemente una raccolta di documenti che soddisfano un determinato criterio, che può essere per un campo già esistente, per un campo custom, intervalli ecc... I *bucket* possono anche essere annidati all'interno di altri *bucket*, in modo da creare una gerarchia o uno schema di partizione condizionale.

4.5.2 Metriche

Le *metriche* sono operazioni matematiche, come la media, la somma ecc., applicate sopra i *bucket*. Solitamente si usa la seguente sintassi:

```
"aggs": {
  "name_of_aggregation": {
    "type_of_aggregation": {
      "field": "document_field_name"
     }
  }
}
```

Uno degli aspetti più interessanti delle aggregazioni è la facilità con cui possono essere convertite in grafici e diagrammi tramite semplici componenti aggiuntivi di Elasticsearch. Per esempio, dato un dataset sulle vendite automobilistiche, è possibile creare un istogramma che mostra il numero di macchine e il ricavo totale per intervallo di prezzo.

```
GET /cars/transactions/_search?search_type=count
{
  "aggs":{
    "price":{
      "histogram":{
        "field": "price",
        "interval": 20000
      },
      "aggs":{
        "revenue": {
```

```
            "sum": {
              "field" : "price"
            }
          }
        }
      }
    }
}
```

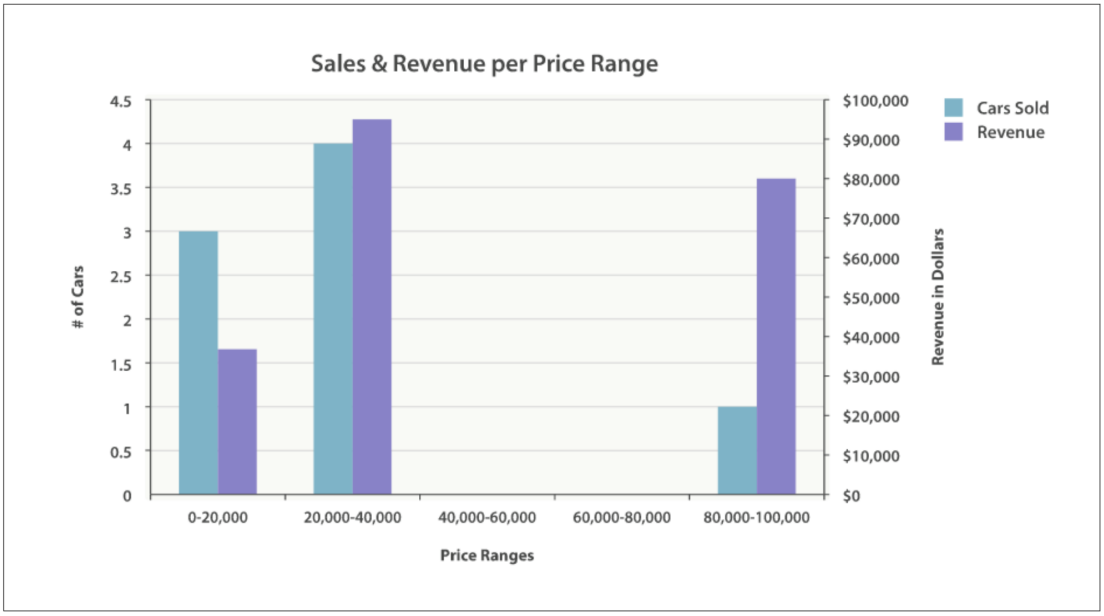


Figure 15: Il grafico ottenuto dalla query precedente.

4.6 Kibana

Molto spesso, assieme a Elasticsearch, viene usato Kibana: se Elasticsearch svolge un ruolo di archiviazione dati e motore di ricerca avanzato, Kibana offre dashboard, diagrammi e grafici intuitivi per esplorare e presentare i dati in modo visivamente accattivante e interattivo.

Kibana sfrutta le capacità di interrogazione di Elasticsearch, esposte attraverso un'API RESTful, offrendo all'utente un'interfaccia grafica di facile utilizzo progettata per la visualizzazione e l'esplorazione dei dati.

Kibana è parte integrante dell'Elastic Stack e può essere usato anche per gestire i propri dati, monitorare il cluster e funzionalità di *access control*. Grazie ai tantissimi tipi di aggregazioni con le quali è possibile costruire dashboard facili da interrogare, manipolare e interagire, rende Elastic Stack uno strumento ideale per utenti e analisti non tecnici che devono analizzare dati ma non hanno le conoscenze necessarie per usare framework più complessi come Hadoop.

Figure 16: Un esempio di dashboard visualizzato su Kibana.

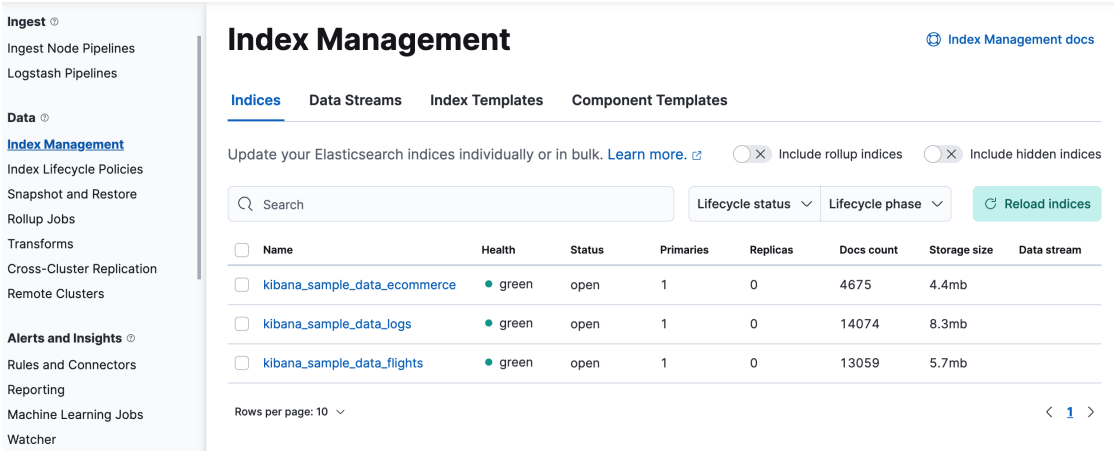


Figure 17: Vista di Kibana per gestire gli indici.

5 Installazione e Test

In questo capitolo si illustreranno i vari passi necessari per installare ed eseguire Elasticsearch.

5.1 Informazioni generali

Elasticsearch può essere installato su qualsiasi computer Linux, MacOS o Windows oppure può essere eseguito tramite un contenitore Docker. In alternativa, si può testare l'intero Elastic Stack su Kubernetes con Elastic Cloud.

Per provare Elasticsearch sul proprio computer, la documentazione consiglia di utilizzare Docker ed eseguire sia Elasticsearch che Kibana. La spiegazione che seguirà è stata testata su Windows.

5.2 Eseguire i container

Dopo aver installato Docker sulla propria macchina, bisogna assicurarsi di aver settato `vm.max_map_count` ad almeno 262 144. Nel seguente link, si possono trovare i comandi necessari a seconda di quale Docker si sta usando su Windows.

Per eseguire Elasticsearch, si lanciano i seguenti comandi

- `docker network create elastic` per creare una nuova docker network;
- `docker pull docker.elastic.co/elasticsearch/elasticsearch:8.10.1` per scaricare l'immagine;
- `docker run --name elasticsearch --net elastic -p 9200:9200 -it -m 1GB docker.elastic.co/elasticsearch/elasticsearch:8.10.1` per eseguire un nuovo container elasticsearch con 1GB di memoria.

Quest'ultimo comando stamperà sul cmd la password e il token necessario per eseguire Kibana.

Per eseguire Kibana:

- `docker pull docker.elastic.co/kibana/kibana:8.10.1` per scaricare l'immagine;
- `docker run --name kibana --net elastic -p 5601:5601 docker.elastic.co/kibana/kibana:8.10.1` per eseguire Kibana.

Quando i comandi hanno avuto successo, si naviga alla pagina `localhost:5601`, dove apparirà una pagina dove inserire il token copiato precedentemente; verrà chiesto di inserire un codice di verifica, che si può recuperare dal cmd dov'è stata eseguita Kibana.

Fatto ciò, verranno installati gli ultimi plugin necessari. L'intera configurazione potrebbe durare qualche minuto. Completato il processo, si può accedere a Kibana con l'user `elastic` e la password stampata dal comando precedente.

5.3 Playground

Per inviare richieste tramite la REST API di Elasticsearch verrà usata la console fornita da Kibana, ma è possibile usare qualunque tipo di client.

5.3.1 PUT e GET di un documento

Per inserire un documento si usa il metodo PUT, nella forma `/[index_name]/_doc/[id]`. Se l'id non viene specificato, ne viene generato uno.

```
PUT /megacorp/_doc/1
{
  "first_name" : "John",
  "last_name" : "Smith",
  "age" : 25,
  "about" : "I love to go rock climbing",
  "interests": [ "sports", "music" ]
}
```

Questa richiesta creerà un indice *megacorp*, con un nuovo documento con id 1.

Con `GET megacorp` si possono richiedere informazioni sull'indice appena creato, tra cui il numero di *shard* primari e replica.

```
"index": {
  "number_of_shards": "1",
  "number_of_replicas": "1",
}
```

Per inserire più documenti alla volta si usa l'API `\_bulk`.

```
PUT megacorp/_bulk
{ "create": { } }
{ "first_name" : "Jane", "last_name" : "Smith", "age" : 32, "about" : "I
    like to collect rock albums", "interests": [ "music" ] }
{ "create": { } }
{ "first_name" : "Douglas", "last_name" : "Fir", "age" : 35, "about": "I
    like to build cabinets", "interests": [ "forestry" ] }
```

Quando si usa l'API `_bulk` ogni riga deve essere separata dal delimitatore `new line \n`

È poi possibile recuperare il documento appena creato tramite una richiesta GET: `GET /megacorp/_doc/1`. Questa ritornerà il documento con il seguente formato:

```
{
  "_index": "megacorp",
  "_id": "1",
  "_version": 1,
  "_seq_no": 0,
  "_primary_term": 1,
  "found": true,
  "_source": {
    "first_name": "John",
    "last_name": "Smith",
    "age": 25,
    "about": "I love to go rock climbing",
```

```
    "interests": [
      "sports",
      "music"
    ]
  }
}
```

5.3.2 Ricerca

Per effettuare una ricerca, si usa sempre una richiesta GET. Una richiesta come `GET / megacorp/_search` ritornerà tutti i documenti:

```
{
  "took": 1,
  "timed_out": false,
  "_shards": {
    "total": 1,
    "successful": 1,
    "skipped": 0,
    "failed": 0
  },
  "hits": {
    "total": {
      "value": 3,
      "relation": "eq"
    },
    "max_score": 1,
    "hits": [
      {
        "_index": "megacorp",
        "_id": "1",
        "_score": 1,
        "_source": {
          "first_name": "John",
          "last_name": "Smith",
          "age": 25,
          "about": "I love to go rock climbing",
          "interests": [
            "sports",
            "music"
          ]
        }
      },
      {
        "_index": "megacorp",
        "_id": "_9gVtIoBfWwo0Uiq2A3l",
        "_score": 1,
        "_source": {
          "first_name": "Jane",
          "last_name": "Smith",
          "age": 32,
          "about": "I like to collect rock albums",
          "interests": [
```

```
            "music"
          ]
        }
      },
      {
        "_index": "megacorp",
        "_id": "ANgVtIoBfWwo0Uiq2A7l",
        "_score": 1,
        "_source": {
          "first_name": "Douglas",
          "last_name": "Fir",
          "age": 35,
          "about": "I like to build cabinets",
          "interests": [
            "forestry"
          ]
        }
      }
    ]
  }
}
```

Si noti come il metadata `_score` sia uguale per tutti i documenti: non avendo specificato una query, ogni documento soddisfa la richiesta.

Una *query lite*, per esempio per cercare il dipendente con il cognome "Smith" con una descrizione che contiene i termini "rock climbing", è:

```
GET /megacorp/_search?q=last_name:Smith +about:(rock climbing)
```

La stessa *query*, in DSL, diventa:

```
GET megacorp/_search
{
  "query": {
    "bool": {
      "must": [
        {"match": { "last_name": "Smith" }},
        {"match": { "about": "rock climbing"}}
      ]
    }
  }
}
```

Entrambe le query ritornano i seguenti documenti:

```
[
  {
    "_index":"megacorp",
    "_id":"1",
    "_score":1.8867437,
    "_source":{
      "first_name":"John",
      "last_name":"Smith",
      "age":25,
      "about":"I love to go rock climbing",
```

```
      "interests":[
      "sports",
        "music"
      ]
    }
  },
  {
    "_index":"megacorp",
    "_id":"_9gVtIoBfWwo0Uiq2A3l",
    "_score":0.9289627,
    "_source":{
      "first_name":"Jane",
      "last_name":"Smith",
      "age":32,
      "about":"I like to collect rock albums",
      "interests":[
        "music"
      ]
    }
  }
]
```

Viene ritornato anche il dipendente Jane Smith in quanto nella sua descrizione viene menzionato il termine "rock" ma avrà uno `_score` inferiore rispetto a John Smith. Se volessimo cercare le esatte parole "rock climbing" bisogna usare la clausola `match_phrase`.

5.3.3 Aggregazione

Un esempio di aggregazione è trovare l'età media dei dipendenti che condividono un certo hobby.

Molti dei nostri campi sono testuali. Elasticsearch li ha automaticamente mappati come un campo testuale con un sottocampo `.keyword`: il campo testuale permette di effettuare ricerca testuali; il campo `.keyword` permette di effettuare aggregazioni.

```
GET /megacorp/_search
{
  "aggs": {
    "all_interests": {
      "terms": { "field": "interests.keyword"},
      "aggs": {
        "avg_age": { "avg" : { "field" : "age" }}
      }
    }
  }
}

Response:
"aggregations": {
  "all_interests": {
    "doc_count_error_upper_bound": 0,
    "sum_other_doc_count": 0,
    "buckets": [
      {
```

```
        "key": "music",
        "doc_count": 2,
        "avg_age": {
          "value": 28.5
        }
      },
      {
        "key": "forestry",
        "doc_count": 1,
        "avg_age": {
          "value": 35
        }
      },
      {
        "key": "sports",
        "doc_count": 1,
        "avg_age": {
          "value": 25
        }
      }
    ]
  }
}
```

5.4 Gestione del cluster

Per ottenere statistiche sul cluster e la sua salute si può usare la richiesta `GET /_cluster/health`.

```
{
  "cluster_name": "docker-cluster",
  "status": "yellow",
  "timed_out": false,
  "number_of_nodes": 1,
  "number_of_data_nodes": 1,
  "active_primary_shards": 30,
  "active_shards": 30,
  "relocating_shards": 0,
  "initializing_shards": 0,
  "unassigned_shards": 1,
  "delayed_unassigned_shards": 0,
  "number_of_pending_tasks": 0,
  "number_of_in_flight_fetch": 0,
  "task_max_waiting_in_queue_millis": 0,
  "active_shards_percent_as_number": 96.875
}
```

Uno stato di salute del cluster di colore giallo significa che tutti gli *shard* primari (29 dei quali shard interni di Elasticsearch) sono attivi e funzionanti, ovvero il cluster è in grado di servire qualsiasi richiesta con successo, ma non tutti gli *shard* di replica sono attivi.

Elasticsearch ha creato uno *shard* di replica per quello primario dell'indice `megacorp`

che però non è stato ancora assegnato ad alcun nodo: poiché al momento c'è un solo nodo, non ha senso memorizzare copie degli stessi dati sullo stesso nodo.

5.4.1 Aggiunta di un nuovo nodo

Per aggiungere un nuovo nodo, basta eseguire il comando: `docker run -e ENROLLMENT_TOKEN ="<token>" --name es02 --net elastic -it -m 1GB docker.elastic.co/elasticsearch/elastic search:8.10.1`. Quando il nuovo nodo sarà attivo, Elasticsearch gestirà automaticamente gli *shard*. Infatti, se rieseguiamo il comando `GET /_cluster/health`, possiamo vedere come lo status del cluster è diventato verde e il numero degli *shard* raddoppiati.

```
{
  "cluster_name": "docker-cluster",
  "status": "green",
  "number_of_nodes": 2,
  "number_of_data_nodes": 2,
  "active_primary_shards": 30,
  "active_shards": 60,
  [...]
}
```

5.5 Altri esempi

Di seguito si elencano diverse query effettuati tramite l'API di Elasticsearch su due dataset distinti: il primo, `goodreads_books_poetry.json`, contiene informazioni riguardanti libri di poesia con circa 25k documenti JSON; il secondo, `goodreads_reviews_poetry.json`, contiene le recensioni dei libri di poesia, con circa 113k documenti.

Query 1: cercare la parola "love" su tutti i documenti: 10k documenti in 36ms.

```
GET /_search
{
  "query": {
    "multi_match" : {
      "query": "love",
      "fields": [ "*" ]
    }
  }
}
```

Query 2: recensioni/libri che contengono le seguenti 5 parole generate casualmente "the underline for consideration in earthwax through weight organisation" a cui sono state aggiunte parole comuni: 831 documenti in 100ms.

```
GET /_search
{
  "query": {
    "multi_match" : {
      "query": "underline consideration earthwax weight organisation",
      "fields": [ "*" ]
    }
```

```
  }
}
```

Query 3: il libro dal titolo "milk and honey": 14 distinti libri (diverse edizioni e lingue) in 13ms.

```
GET /_search
{
  "query": {
    "match_phrase": {
      "title": "milk and honey"
    }
  }
}
```

Query 4: la recensione più votata di "milk and honey", usando l'edizione con book_id 23513349: ritorna 1927 recensioni, la più votata con 1065 voti, in 40ms.

```
GET /_search
{
  "query": {
    "match": {
      "book_id": "23513349"
    }
  },
  "sort": [
    {
      "n_votes": {
        "order": "desc"
      }
    }
  ]
}
```

Query 5: numero di libri per ogni intervallo di rating: 43ms.

```
GET /_search
{
  "aggs": {
    "num_book_by_avg_rating": {
      "histogram": {
        "field": "average_rating",
        "interval": 0.5
      }
    }
  }
}

"buckets": [
  { "key": 0, "doc_count": 11 },
  { "key": 0.5, "doc_count": 0 },
  { "key": 1, "doc_count": 2 },
  { "key": 1.5, "doc_count": 8 },
  { "key": 2, "doc_count": 58 },
  { "key": 2.5, "doc_count": 277 },
  { "key": 3, "doc_count": 1722 },
```

```
  { "key": 3.5, "doc_count": 8926 },
  { "key": 4, "doc_count": 15088 },
  { "key": 4.5, "doc_count": 3066 },
  { "key": 5, "doc_count": 341 }
]
```

5.6 Alternative

Tre alternative a Elasticsearch molto usate sono Splunk, Solr e OpenSearch.

5.6.1 Splunk

Nata nel 2003, Splunk è una piattaforma che permette di cercare, analizzare e visualizzare dati e log generati da siti, sensori ecc.

A differenza di Elasticsearch, è uno strumento commerciale con un costo abbastanza alto, ma altrettanto ricco di funzionalità: è facile da configurare, ha un'interfaccia *user-friendly* e una ricerca veloce e performante, supportata dall'uso di *machine learning* e analisi predittive.

Essendo proprietario, il suo uso non è personalizzabile e flessibile come quello di Elasticsearch.[4]

5.6.2 Solr

Come Elasticsearch, anche Apache Solr è costruito sopra Lucene. Nato nel 2006, è stato il motore di ricerca dominante prima della diffusione di Elasticsearch, grazie alle avanzate funzionalità di ricerca che offriva.

A oggi, i due motori sono abbastanza uguali in termini di performance e scalabilità, ma se i dati sono per la maggior parte statici e c'è la necessita di avere analisi dei dati precise, è più consigliato usare Solr.

Rispetto a Elasticsearch, Solr ha una curva di apprendimento più ripida e richiede maggior sforzo per configurare tutto l'ambiente. Inoltre è meno ricco in termini di funzionalità di monitoraggio del cluster e analytics dei dati, ma è molto più focalizzato per la ricerca full-text e supporta più formati (CSV, XML, JSON), mentre Elasticsearch supporta solo JSON. [5]

5.6.3 OpenSearch

Opensearch è una fork di Elasticsearch avvenuta nel 2021 da parte di Amazon Web Services. Come tale, condivide molte funzionalità con Elastisearch, anche se ci possiamo aspettare che nel futuro i due progetti divergano sempre di più.

Al momento le differenza principali riguardano l'accessibilità di alcune funzionalità, che in Elasticsearch sono a pagamento, mentre Opensearch ha implementato una controparte open-source. [6]

6 Demo

Elasticsearch offre librerie client per molti linguaggi, tra cui Java, JavaScript, .NET, PHP e Python Client [7]. Esistono anche altre librerie non ufficiali, create dalla community, per linguaggi come Scala, Kotlin, C++ [8].

Ogni libreria provvede un mapping uno a uno con la REST API di Elasticsearch. Prendiamo come esempio la libreria client per Javascript per sviluppare una semplice applicazione web con Node.js e React, con una feature di ricerca sul dataset `goodreads_books_poetry.json`.

6.1 Installazione

1. Per eseguire l'applicazione è necessario installare Node.js e il package manager "npm".
2. In un terminale, dirigersi nella cartella "demo" e installare le dipendenze necessarie con `npm install`.
3. Lanciare il server Node con il comando `node index.js`.
4. In un altro terminale, dirigersi nella cartella "frontend" e installare le dipendenze necessarie con `npm install`.
5. Iniziare l'applicazione React con `npm start`.

Eseguiti questi passi, si aprirà la seguente pagina, composta da un semplice campo dove inserire la propria ricerca e un bottone che invia l'effettiva richiesta a elasticsearch.

6.2 Implementazione

Esistono diversi modi per costruire un client Elasticsearch a seconda del tipo di autenticazione che si vuole usare e come è installato Elasticsearch [9]. Nel caso di docker, bisogna copiare il certificato CA dal container alla macchina locale con il seguente comando: `docker cp [container name]:/usr/share/elasticsearch/config/certs/http_ca.crt .`

Durante la configurazione del client, oltre a host, username e password, indicheremo anche il percorso del certificato.

```
const { Client } = require('@elastic/elasticsearch')
const client = new Client({
  node: 'https://localhost:9200',
  auth: {
    username: 'elastic',
    password: 'password'
  },
  tls: {
    ca: fs.readFileSync('./http_ca.crt'),
    rejectUnauthorized: false
  }
})
```

Quando nella pagina web si effettua una ricerca, si invia una richiesta a elasticsearch il cui corpo è analogo a quelle viste nel capitolo precedente. In questo specifico esempio, si cerca:

- nell'indice `poetry`;
- per tutti i campi dei documenti, ma con maggiore enfasi sul titolo del libro;
  - il cursore `^` indica che il punteggio di quel campo deve essere incrementato per il numero specificato;
- con il parametro `fuzziness` per permettere la ricerca non solo per la parola esatta, ma anche per le parole che più si avvicinano in termini di "quanti caratteri dista la parola cercata rispetto a un'altra";
  - il valore "AUTO" indica che è Elasticsearch stesso a gestire la distanza.

```
await client.search({
    index: "poetry",
    query: {
        multi_match: {
            fields: ["title^3", "title_without_series^3", "*"],
            query: req.query.query,
            fuzziness: "AUTO"
        }
    }
});
```

Per esempio, cercare "milk honey" e "mik hony" ritornano gli stessi risultati.

milk honey Search

Milk and Honey

Description:
From Rupi Kaur, the top ten Sunday Times-bestselling author, comes the beautiful audio edition of milk and honey, her debut poetry collection. Read to you by the author, milk and honeyis a book about surivival, love, loss and femininity, milk and honeytakes you on a journey of of hurting, loving, breaking and healing. Listen to Rupi Kaur's incredible poetry, in her own words. this is the journey of suriving through poetry this is the blood sweat tears of twenty-one years this is my heart in your hands this is the hurting the loving the breaking the healing - rupi kaur Praise for Sunday Timesbestseller milk and honey: 'Kaur is at the forefront of a poetry renaissance' Observer 'Kaur made her name with poems about love, life and grief. They resonate hugely' Sunday Times 'Poems tackling feminism, love, trauma and healing in short lines as smooth as pop music' New York Times 'Caught the imagination of a large, atypical poetry audience...Kaur knows the good her poetry does: it saves lives' Evening Standard 'Breathing new life into poetry...It has people reading, and listening' The Pool

Average rating:
4.23

Go to Goodreads

Milk and Honey

Description:

Average rating:
4.23

Go to Goodreads

mik hony Search

Milk and Honey

Description:
From Rupi Kaur, the top ten Sunday Times-bestselling author, comes the beautiful audio edition of milk and honey, her debut poetry collection. Read to you by the author, milk and honeyis a book about surivival, love, loss and femininity, milk and honeytakes you on a journey of of hurting, loving, breaking and healing. Listen to Rupi Kaur's incredible poetry, in her own words. this is the journey of suriving through poetry this is the blood sweat tears of twenty-one years this is my heart in your hands this is the hurting the loving the breaking the healing - rupi kaur Praise for Sunday Timesbestseller milk and honey: 'Kaur is at the forefront of a poetry renaissance' Observer 'Kaur made her name with poems about love, life and grief. They resonate hugely' Sunday Times 'Poems tackling feminism, love, trauma and healing in short lines as smooth as pop music' New York Times 'Caught the imagination of a large, atypical poetry audience...Kaur knows the good her poetry does: it saves lives' Evening Standard 'Breathing new life into poetry...It has people reading, and listening' The Pool

Average rating:
4.23

Go to Goodreads

Milk and Honey

Description:

Average rating:
4.23

Go to Goodreads

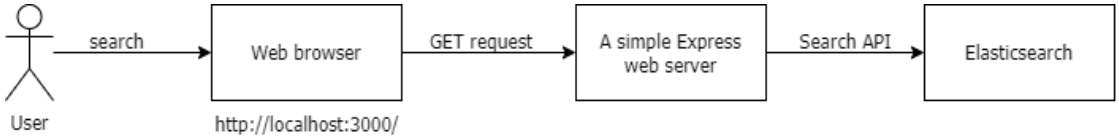


Figure 18: L'applicazione web ha un'architettura molto semplice composta da front end sviluppata in React e una parte di backend sviluppata con Node ed Express.

7 Conclusioni

In generale bisogna non commettere l'errore di usare Elasticsearch come un database tradizionale: non è adatto come storage principale, in quanto manca delle proprietà ACID e presenta un maggiore rischio di perdita di dati.

Benché alcune funzionalità più avanzate siano accessibili solo con una licenza, quando usato correttamente come motore di ricerca e *analytics*, la sua versione base rimane comunque uno strumento molto valido, purché le analisi che si vogliono effettuare non siano troppo complesse: l'API di analytics, infatti, risulta abbastanza limitato, se confrontato a strumenti di business intelligence come Power BI.

References

[1] Elasticsearch. Sito ufficiale.

[2] Apache Lucene. Sito ufficiale.

[3] DB-Engines. Ranking mensile dei motori di ricerca.

[4] Rajesh Tilwani. Splunk vs Elasticsearch.

[5] Rafal Kuć. Solr vs Elasticsearch.

[6] Radu Gheorghe. OpenSearch vs Elasticsearch.

[7] Elasticsearch. Elasticsearch Clients.

[8] Elasticsearch. Community Contributed Clients.

[9] Elasticsearch. Connecting.