ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA

Scuola di Ingegneria
Corso di Laurea Magistrale in Ingegneria Informatica

ANALISI PRESTAZIONALE DELLA RICERCA DI VICINI IN UNO SPAZIO VETTORIALE CON ELASTICSEARCH SU INFRASTRUTTURA CLOUD

Elaborato in
Data Mining M

Relatore
Prof.Ing. Claudio Sartori

Co-relatore
Ing. Riccardo Zanella

Presentata da
Ugo Arcidiacono

Quarta Sessione di Laurea
Anno Accademico 2021/2022

PAROLE CHIAVE

Pipeline

Dataflow

Elasticsearch

Google Cloud Platform

Semantic Similarity search

Indice

Introduzione

Negli ultimi anni, a causa degli enormi progressi dell'informatica e della sempre crescente quantità di dati generati, si è sentito sempre più il bisogno di trovare nuove tecniche, approcci e algoritmi per la ricerca dei dati. Infatti, la quantità di informazioni da memorizzare è diventata tale che ormai si sente sempre più spesso parlare di "Big Data". Nel 2001, Doug Laney, allora vice presidente e Service Director dell'azienda Meta Group, descrisse in un report il Modello delle 3V relativo alle 3V dei Big Data: Volume, Velocità e Varietà. Un modello semplice e sintetico per definire dei nuovi dati, generati dall'aumento delle fonti informative e più in generale dall'evoluzione delle tecnologie. Oggi il paradigma di Laney è stato arricchito dalle variabili di Veridicità e Variabilità e per questo si parla di 5V dei Big Data. Questo nuovo scenario ha reso sempre più inefficaci gli approcci tradizionali alla ricerca di dati. Recentemente sono state quindi proposte nuove tecniche di ricerca, come ad esempio le ricerche Nearest Neighbor. La varietà di queste tecniche le ha rese adatte a diverse applicazioni come il *pattern recognition*, la ricerca nei dati multimediali, la ricerca full-text e il *data mining*.

Questo elaborato è il risultato di un tirocinio svolto presso l'azienda Injenia Srl, durante il quale sono state analizzate le prestazioni della ricerca di vicini in uno spazio vettoriale utilizzando come sistema di data storage Elasticsearch su un'infrastruttura cloud. In particolare, sono stati analizzati e messi a confronto i tempi di ricerca delle ricerche Nearest Neighbor esatte e approssimate, valutando anche la perdita di precisione nel caso di ricerche approssimate, utilizzando due diverse metriche di distanza: la similarità coseno e il prodotto scalare.

La descrizione del lavoro svolto è stata strutturata in cinque capitoli:

- Capitolo 1: viene fatta una panoramica sui principali metodi utilizzati per effettuare ricerche per similarità in uno spazio vettoriale e vengono descritte le ricerche Nearest Neighbor;

- Capitolo 2: viene descritto Elasticsearch e le API Python più utilizzate e successivamente sono riportati i risultati ottenuti da una prima esecuzione locale;

- Capitolo 3: viene descritta l'infrastruttura cloud di sperimentazione realizzata sulla Google Cloud Platform;
- Capitolo 4: sono riportati i risultati ottenuti dall'esecuzione sul cloud;
- Capitolo 5: viene descritto Rally, tool di benchmarking ufficale di Elastic, e sono riportati i risultati ottenuti con alcuni test effettuati in locale.

Capitolo 1

Ricerca di vicini in uno spazio vettoriale

In questo capitolo viene fatta una panoramica sui principali metodi utilizzati per effettuare ricerche per vicinanza (o similarità) in uno spazio vettoriale, ponendo maggiore enfasi nel caso di ricerche in testo non strutturato. Sono quindi descritti i metodi usati per il *word embedding* che permettono di trasformare parole o frasi in vettori n-dimensionali, sui quali poi si effettuano le ricerche k-Nearest Neighbors per trovare i punti più vicini nello spazio ad un punto passato in input come parametro di ricerca.

1.1 Ricerca per similarità

L'operazione di ricerca tradizionale viene usata solitamente con dati strutturati: data una query, vengono recuperati tutti i record di un database che corrispondono esattamente alla ricerca. Ricerche più complicate come le range query su chiavi numeriche o ricerche di prefissi su chiavi alfanumeriche si basano sempre sul confronto di due chiavi, verificando se queste sono uguali o meno, o su un ordinamento delle chiavi. Con il passare degli anni e l'avanzamento della tecnologia, hanno preso sempre più piede i database basati sulla memorizzazione di dati non strutturati, in quanto si è iniziato a memorizzare e ricercare dati sempre più eterogenei come testo libero, immagini, audio e video. Questo scenario richiede algoritmi e modelli di ricerca più generali rispetto a quelli usati tradizionalmente per la ricerca di dati semplici. Un concetto molto importante in questo senso è quello della *ricerca per prossimità* o *ricerca per similarità* che consiste nel cercare in un database elementi che sono "vicini" o "simili" ad un dato elemento di ricerca. La similarità è modellata con una funzione di distanza e l'insieme degli oggetti all'interno dello spazio di ricerca viene mappato su uno spazio vettoriale: quindi ogni oggetto viene salvato come

un insieme di k coordinate. In generale, le ricerche effettuate nei database tradizionali richiedono di recuperare uno o più record che completano un'informazione inserita in ricerca; i nuovi tipi di dato come immagini, audio o video (dati multimediali) non possono essere ricercati efficacemente utilizzando lo stesso approccio. Infatti, non solo non possono essere ordinati, ma non posso nemmeno essere comparati per uguaglianza. Nelle applicazioni multimediali, tutte le query richiedono oggetti che sono simili ad un dato oggetto in ricerca.

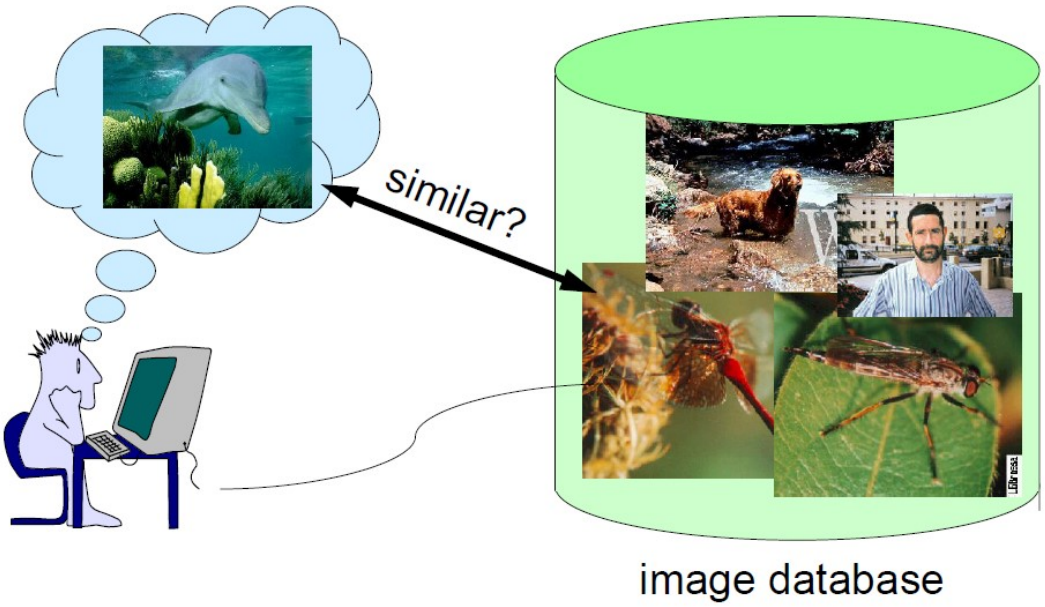


Figura 1.1: Search problem[1]

Questi approcci sono basati sulla definizione di una funzione di similarità tra oggetti. Vengono estratte k *features* dall'immagine (ad esempio da un'immagine può essere estratto il colore predominante) e ogni oggetto viene rappresentato come un punto in uno spazio di dimensione k.

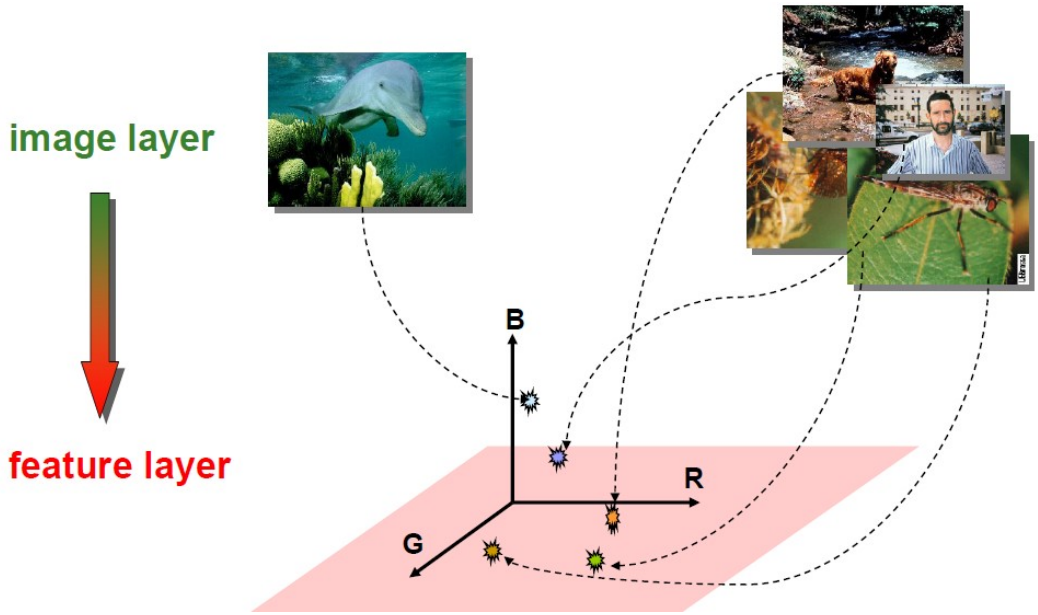


Figura 1.2: Feature-based approach[1]

La ricerca di testo non strutturato richiede soluzioni simili a quelle usate per la ricerca di immagini. I documenti di testo potrebbero essere ricercati utilizzando delle stringhe in input e controllando se queste sono presenti o meno all'interno del testo, ma questo tipo di ricerca è decisamente poco efficiente e quindi spesso si è interessati a ricercare all'interno di un testo a livello di semantica. Ad esempio, cercando "liberare da un obbligo" ci aspettiamo che vengano restituiti anche i documenti in cui è presente la parola "redenzione".[2] Anche in questo caso si utilizza una rappresentazione degli oggetti come punti in uno spazio vettoriale, in cui ogni parola o frase del documento di testo viene mappata in un punto dello spazio. In generale, le tecniche di *word embedding* possono essere divise in due gruppi principali.[3]
Il primo si basa su un approccio di predizione probabilistica: questi metodi sono usati per allenare un modello, sulla base di un contesto composto da parole prese da un corpus, generalizzando i risultati in uno spazio ridotto di dimensione n (scelto arbitrariamente). Così una parola è rappresentata come un vettore nello spazio e preserva le proprietà di contesto, in maniera tale che i vettori delle parole sono più vicini se le parole occorrono negli stessi contesti linguistici, cioè se sono riconosciute come semanticamente più simili. Word2Vec (Word-2-Vector) è uno dei più famosi algoritmi di questa prima categoria che permette di preservare informazioni riguardanti la semantica e la sintassi, uno strumento molto potente sviluppato per mappare parole o

frasi in vettori composti da numeri reali. Tramite l'insieme dei vettori che rappresentano gli elementi di partenza, è possibile sfruttare lo spazio vettoriale costruito su quest'ultimi per svariate operazioni:

- Trovare affinità tra i singoli elementi: più i vettori sono vicini e più potranno essere considerati simili;
- Trovare relazioni semantiche tra coppie di parole in maniera geometrica.

Esistono due versioni diverse dell'algoritmo Word2Vec:

- *Continuous bags of words (CBOW)*: predice una parola dato il contesto di n parole attorno;
- *Skip-gram*: predice le parole che formano il contesto, data la parola centrale.

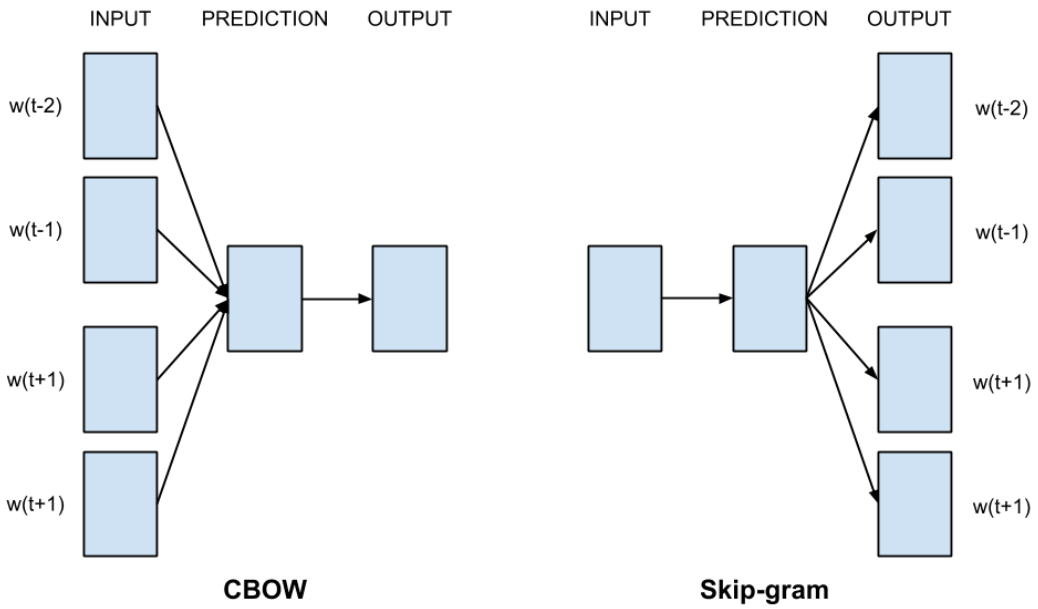


Figura 1.3: Architettura dei modelli CBOW e Skip-gram

Oltre all'architettura dei due modelli, le principali differenze sono la velocità (CBOW è più veloce) e la precisione (Skip-gram predice meglio le parole poco frequenti).

Il secondo gruppo di tecniche di word embedding è il cosiddetto metodo *count-based*. Con questo metodo, la costruzione dei vettori è basata sulla matrice delle co-occorrenze; infatti, la statistica delle occorrenze delle parole in un corpus è la principale fonte di informazione a disposizione di tutti i metodi non supervisionati per l'apprendimento della rappresentazione delle parole. Uno degli algoritmi di count-based word embedding più utilizzati è il *GloVe*[4].
Sia X la matrice delle co-occorrenze, ogni elemento X_{ij} di X indica il numero di volte che la parola j compare nel contesto della parola i.
In Tabella 1.1 è mostrato un esempio di matrice delle co-occorrenze delle due seguenti frasi:

The fast cat wears no hat.
The cat in the hat ran fast.

	cat	fast	hat	in	no	ran	the	wears
cat	0	2	2	1	1	1	3	1
fast	2	0	2	1	1	1	3	1
hat	2	2	0	1	1	1	3	1
in	1	1	1	0	0	1	2	0
no	1	1	1	0	0	0	1	1
ran	1	1	1	1	0	0	2	0
the	3	3	3	2	1	2	1	1
wears	1	1	1	0	1	0	1	0

Tabella 1.1: Esempio di matrice delle co-occorrenze

Da questo esempio possiamo vedere come la matrice delle co-occorrenze è una matrice simmetrica in cui ogni riga e ogni colonna è un vettore di ogni parola presente nel testo in input. Questa tecnica presenta principalmente due problemi:

- Nella applicazioni reali, il numero di parole uniche può essere anche nell'ordine dei milioni;
- La matrice è sparsa e la presenza di molti valori pari a 0 può introdurre qualche problema di gestione a livello applicativo.

Per risolvere questi problemi, solitamente si utilizzano degli algoritmi di riduzione della dimensionalità, come ad esempio il Principal Component Analysis (PCA)[5].

1.2 Ricerche Nearest Neighbor

La ricerca Nearest Neighbor è il problema di ottimizzazione che consiste nel trovare il punto in un insieme più vicino ad un punto dato in input. La vicinanza è solitamente espressa tramite una funzione di dissimilarità: meno i due punti sono "simili", maggiore sarà il valore della funzione. Formalmente il problema del Nearest Neighbor è così definito:

Dato un insieme di punti S in uno spazio n-dimensionale R^n e un punto $q \in S$,
trovare il punto più vicino a q, secondo una data funzione di distanza

È possibile generalizzare la ricerca Nearest Neighbor utilizzando un parametro k per ottenere i k punti più vicini ad un punto di ricerca. In questo caso si parla di ricerca *k-nearest neighbors*.

1.2.1 Metriche di distanza

Per effettuare una ricerca Nearest Neighbor bisogna quindi prima di tutto definire una funzione di distanza. Di seguito alcune delle più utilizzate negli spazi vettoriali:

- *Distanza euclidea*: misura la lunghezza del segmento avente per estremi i due punti

$$d(x, y) = \sqrt{\sum_{k=1}^{n}(x_i - y_i)^2}$$

- *Distanza di Manhattan*: somma il valore assoluto della differenza delle coordinate dei due punti

$$d(x, y) = \sum_{k=1}^{n} |x_i - y_i|$$

- *Distanza di Chebyshev*: la distanza tra due vettori è calcolata come il valore massimo della loro differenza lungo gli assi

$$d(x, y) = max_i|x_i - y_i|$$

- *Distanza di Minkowski*: è una generalizzazione delle distanze precedenti

$$d(x, y) = \sqrt[p]{\sum_{k=1}^{n} |x_i - y_i|^p}$$

Con p = 1 si ottiene la distanza di Manhattan, mentre con p = 2 la distanza euclidea. Nel caso limite in cui p tende ad infinito si ha la Distanza di Chebyshev.

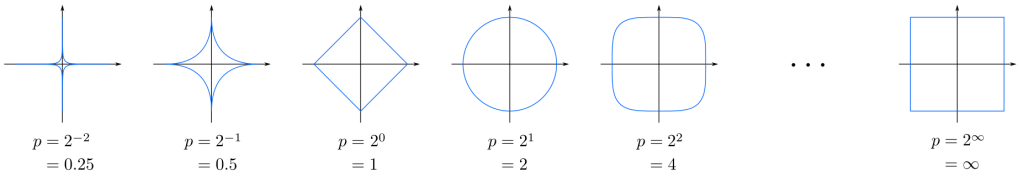


Figura 1.4: Distanza di Minkowski al variare di p [6]

- *Distanza di Hamming*: viene utilizzata tipicamente con vettori booleani e calcola la somma dei punti in cui i vettori non corrispondono

$$d(x, y) = \sum_{k=1}^{n} x_i \oplus y_i$$

- *Similarità coseno*: misura la similitudine tra due vettori calcolando il coseno tra di loro. Ritorna un valore compreso tra -1 e +1, dove -1 indica una corrispondeza esatta ma opposta e +1 indica due vettori uguali.

$$d(x, y) = \frac{x^T y}{\|x\|\|y\|}$$

ricordando che $x^T y = \|x\|\|y\| \cos\Theta$ si ha:

$$d(x, y) = \frac{x^T y}{\|x\|\|y\|} = \cos\Theta$$

dove Θ è l'angolo tra i due vettori.

Effettuare la similarità coseno con vettori normalizzati, la cui norma cioè è pari a 1, equivale ad effettuare un *prodotto scalare*.

1.2.2 Strutture dati

Una delle parti più importanti nell'esecuzione di una ricerca Nearest Neighbor è la scelta dell'algoritmo di ricerca e della struttura dati da utilizzare per effettuarla[7]. Di seguito una panoramica di quelle più utilizzate:

- *k-d-tree*: è un albero binario di ricerca in cui ogni livello è etichettato ciclicamente con una delle n coordinate e in ogni nodo c'è un separatore dato dal valore mediano dell'intervallo che si sta partizionando.

- *Ball-tree*: è un albero binario in cui ogni nodo rappresenta un insieme di punti, chiamato $Pts(N)$. Dato un dataset, il nodo radice del Ball-Tree rappresenta l'insieme completo dei punti del dataset. Possiamo distinguere i nodi foglia dai nodi non-foglia: un nodo foglia contiene una lista di punti, mentre un nodo non-foglia ha due nodi figli ($N.child1$ e $N.child2$), dove:

  $$\begin{cases} pts(N.child1) \cap pts(N.child2) = \varnothing \\ pts(N.child1) \cup pts(N.child2) = pts(N) \end{cases}$$

  Ogni nodo ha un punto particolare chiamato *Pivot*. In base all'implementazione, il Pivot può essere un punto all'interno del dataset oppure il centroide di $Pts(N)$. Ogni nodo memorizza la distanza massima dei suoi punti rispetto al pivot. Definiamo questa distanza come *raggio* del nodo. L'albero è costruito in maniera tale che i nodi più bassi dell'albero hanno un raggio minore.

- *R-tree*: è una struttura ad albero bilanciata e paginata, basata sull'innestamento gerarchico di *overlapping regions*. Ogni nodo corrisponde ad una regione rettangolare, definita come il *Minimum Bounding Box* che contiene tutte le regioni figlie. Il *Minimum Bounding Box* è il più piccolo rettangolo con i lati paralleli agli assi coordinati che contiene tutte le regioni figlie, ed è definito come prodotto di n intervalli.

1.3 Ricerche k-NN approssimate

Le ricerche k-NN classiche calcolano la similarità utilizzando un approccio di forza bruta che calcola la distanza tra il punto di ricerca e i punti presenti nel dataset, in maniera tale da ottenere un risultato esatto. Tuttavia questo metodo non scala bene in caso di dataset di dimensioni importanti, riducendo di molto l'efficienza della ricerca. Per questi casi, può essere utilizzata una

variante della ricerca k-NN, la ricerca k-NN approssimata, che risolve questi problemi utilizzando dei tool di ristrutturazione degli indici in maniera più efficiente e riducendo la dimensionalità dei vettori. Utilizzando questo tipo di approccio, si perde un po' di precisione in fase di ricerca, ma si ottiene un apprezzabile miglioramento in termini di velocità.
Gli algoritmi per la ricerca approssimata possono essere raggruppati in tre diverse categorie, in base alla struttura utilizzata: hash (es. LSH), alberi (es. FAISS, Annoy), grafi (es. HNSW). Di seguito una panoramica sugli algoritmi di ricerca più usati per effettuare le ricerche k-NN approssimate.

1.3.1 Locality Sensitive Hashing

L'idea principale alla base di questa tecnica è quella di applicare diverse funzioni hash ad ogni punto in maniera tale che, per ogni funzione, la probabilità di collisione è molto più alta per punti che sono vicini. In questo modo, per ottenere i punti più vicini ad un dato punto di ricerca q, vengono applicate le funzioni hash a quest'ultimo e si cercano i risultati nel bucket ottenuto. Questa tecnica ha quindi bisogno di una fase di preprocessing in cui a tutti i punti vengono applicate le funzioni hash, per poi effettuare lo stesso procedimento in fase di ricerca.

1.3.2 FAISS

FAISS (Facebook AI Similarity Search)[8] è una libreria per la ricerca per similarità e clustering di vettori. Contiene algoritmi per la ricerca in insiemi di vettori di ogni dimensione, anche nei casi in cui questi non entrino nella memoria RAM.
Dato un insieme di vettori x_i di dimensione d, FAISS costruisce una struttura dati nella RAM. Una volta costruita la struttura, quando viene presentato un nuovo vettore x (sempre di dimensione d), viene eseguita la seguente operazione:

$$j = argmin_i ||x - x_i||$$

dove $|| \cdot ||$ è la distanza euclidea.
La struttura dati creata da FAISS viene usata come indice e il calcolo dell'$argmin$ è l'operazione di ricerca sull'indice.
FAISS implementa anche altre funzioni. È possibile anche:

- ottenere non solo il punto più vicino, ma i k più vicini;
- cercare diversi vettori contemporaneamente (per alcuni tipi di indice, questa operazione è più veloce che cercare un vettore dopo l'altro);

- aumentare la velocità di ricerca a costo di una minore precisione;
- effettuare una ricerca che massimizzi il prodotto scalare di due vettori, piuttosto che minimizzare la distanza euclidea;
- ritornare tutti gli elementi che stanno all'interno di un dato raggio di ricerca (range search);
- salvare l'indice sul disco piuttosto che sulla RAM.

1.3.3 Annoy

Annoy (Approximate Nearest Neighbors Oh Yeah) [9] è una libreria per la ricerca di punti in uno spazio vicini ad un dato punto di ricerca. Utilizza proiezioni casuali per la riduzione della dimensionalità dei vettori e un'albero come struttura di indicizzazione. Per ogni nodo intermedio nell'albero, viene scelto un iperpiano che divide lo spazio in due sottospazi. La scelta dell'iperpiano viene fatta scegliendo due punti e facendo in modo che l'iperpiano sia equidistante da questi. Questo procedimento viene effettuato k volte fino a che non viene generato tutto l'albero. Ci sono due parametri diversi da regolare: il numero di alberi n_trees e il numero di nodi da ispezionare durante la ricerca $search\_k$.

- n_trees viene fornito durante la fase di build e influisce sul tempo di build e sulla dimensione dell'indice. Un valore maggiore darà risultati più precisi, ma genererà indici di dimensione maggiore.
- $search\_k$ viene fornito a runtime e influisce sulle performance di ricerca. Un valore maggiore darà risultati più precisi, ma aumenterà i tempi di ricerca.

Se il parametro $search\_k$ non viene fornito, questo sarà settato di default al valore $n * n\_trees$ dove n è il numero di *nearest neighbors*.

La caratteristica che contraddistingue Annoy dalle altre librerie di ricerca approssimata è la possibilità di utilizzare file statici come indici: questo fa sì che è possibile condividere un indice tra più processi. Annoy inoltre separa il processo di creazione di un indice con quello del suo caricamento, in maniera tale da poter passare gli indici come file e mapparli rapidamente in memoria. Un'altra nota positiva di Annoy è che cerca di minimizzare l'occupazione di memoria degli indici.
Panoramica delle principali caratteristiche:

- supporto a: distanza euclidea, distanza di Manhattan, similarità coseno, distanza di Hamming, prodotto scalare;

- funziona meglio con vettori di dimensioni non troppo grandi (minori di 100) ma sembra lavorare sorprendentemente bene anche con vettori di dimensioni maggiori di 1000;
- utilizzo di memoria ridotto;
- permette di condividere la memoria tra processi;
- la creazione dell'indice è separata dalla consultazione (non è possibile aggiungere altri elementi una volta che l'albero è stato creato);
- possibilità di costruire l'indice su disco per abilitare l'indicizzazione di dataset di grandi dimensioni che non entrano in RAM.

1.3.4 HNSW

Nella maggior parte degli algoritmi per la ricerca k-NN basati su grafi, la ricerca viene effettuata utilizzando un instradamento *greedy* sul grafo. Per un dato grafo di prossimità, la ricerca parte da un *enter point* (può essere scelto casualmente o ricavato da un altro algoritmo) e si percorre iterativamente il grafo. Ad ogni passo, l'algoritmo esamina la distanza del punto di ricerca rispetto ai vicini del nodo corrente e seleziona come prossimo nodo quello adiacente che minimizza la distanza, tenendo sempre traccia del migliore punto trovato fino a quel momento. La ricerca termina quando una *stop condition* viene verificata. Il Navigable Small World (NSW) è un algoritmo basato su grafi che utilizza dei *grafi navigabili* (cioè grafi che scalano il numero di *hops* in maniera logaritmica o polilogaritmica rispetto alla dimensione della rete). Il grafo NSW è costruito attraverso inserimenti consecutivi di elementi con un ordinamento casuale collegati in maniera bidirezionale agli M elementi più vicini tra quelli già inseriti.
L'instradamento lungo il grafo può essere diviso in due fasi: *zoom-out* e *zoom-in*. L'algoritmo greedy inizia nella fase di zoom-out partendo dai nodi a basso grado (il grado di un nodo è il numero di collegamenti con gli altri vertici del grafo) e attraversa il grafo aumentando il grado dei nodi fino a che il raggio caratteristico della lunghezza dei collegamenti del nodo raggiunge la scala della distanza dalla query. Nella fase di zoom-in, invece, si visitano i nodi con un grado maggiore.
L'idea alla base dell'algoritmo Hierarchical NSW (HNSW) [10] è di separare i collegamenti in base alla loro lunghezza in diversi livelli ed eseguire la ricerca in un grafo multilivello. In questo modo è possibile valutare solo una porzione fissa di connessioni indipendentemente dalla dimensione della rete, ottenendo una scalabilità logaritmica. In questa struttura, la ricerca parte dai livelli più alti

(in cui ci sono i collegamenti più lunghi). L'algoritmo attraversa gli elementi in maniera greedy partendo dal livello più alto fino a che non si raggiunge un minimo locale. A quel punto, la ricerca passa al livello sottostante (con dei collegamenti più corti), ricominciando dall'elemento che era il minimo locale nel livello precedente. Il numero massimo di collegamenti per ogni elemento in tutti i livelli può essere reso costante, permettendo così una complessità logaritmica dell'instradamento nella rete.

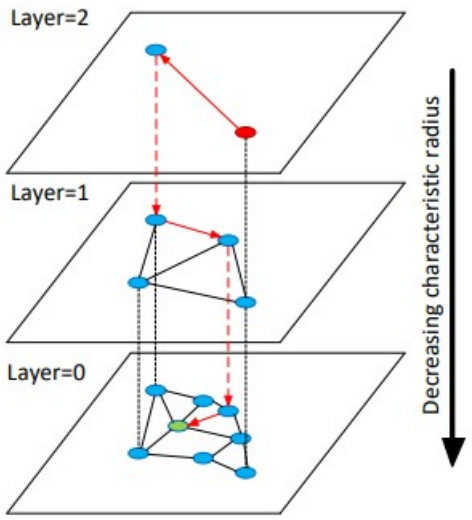


Figura 1.5: Esempio di ricerca in grafo multilivello in HNSW[10]

Capitolo 2

Elasticsearch API per ricerche kNN

In questo capitolo viene fatta come prima cosa una panoramica su Elasticsearch, tool di storage e analisi di dati utilizzato durante tutto il corso di questo elaborato. Sono riportati alcuni esempi delle API più utilizzate e infine sono mostrati i risultati ottenuti effettuando alcuni test su un cluster in locale.

2.1 Introduzione a Elasticsearch

Elasticsearch è un motore di ricerca e analisi dei dati distribuito basato su Apache Lucene. Dal suo rilascio nel 2010, Elasticsearch è diventato rapidamente il motore di ricerca più popolare ed è comunemente utilizzato per l'analisi dei dati di log, la ricerca full-text, l'intelligenza sulla sicurezza, l'analisi dei dati aziendale e i casi d'uso dell'intelligenza operativa. Può anche essere definito dai seguenti tre punti fondamentali[11]:

- un sistema di storage di dati dove ogni campo è indicizzabile e ricercabile;
- un sistema per l'analisi di dati real-time;
- un sistema capace di scalare su centinaia di server e petabytes di dati strutturati e non strutturati.

Elasticsearch è *document-oriented*, nel senso che vengono memorizzati interi oggetti o *document*. Il contenuto dei documenti non viene solo memorizzato, ma anche indicizzato in maniera tale da facilitare le operazioni in fase di ricerca. Elasticsearch usa la JavaScript Object Notation (JSON) come formato di serializzazione per i documenti. JSON è supportato dalla maggior parte dei linguaggi di programmazione ed è di fatto diventato il formato standard usato da tutti i sistemi di storage NoSQL. È un formato semplice, conciso e di facile lettura.

Come già detto, Elasticsearch è costruito per scalare su centinaia (o anche migliaia) di server e gestire petabytes di dati. Tuttavia, la complessità della natura distribuita del sistema viene nascosta all'utente e gestita in maniera automatica. Di seguito alcune delle operazioni che Elasticsearch esegue automaticamente sotto il cofano:

- partizionamento dei documenti in differenti *container* o *shard*, che possono essere salvati in un nodo singolo o multiplo;
- bilanciamento degli shard nei nodi del cluster per distribuire in maniera omogenea il carico di indicizzazione e ricerca;
- duplicazione di ogni shard con una copia dei dati salvati per prevenire la perdita di dati in caso di problemi hardware;
- instradamento delle richieste da qualsiasi nodo del cluster al nodo in cui sono effettivamente memorizzati i dati richiesti;
- inserimento di nuovi nodi man mano che il cluster cresce o ridistribuzione degli shard in caso di perdita di nodi

Elasticsearch è costruito per essere sempre disponibile e per scalare in base alle esigenze dell'utente. La scalibilità può essere ottenuta migliorando le caratteristiche dei server (scalabilità verticale) o aggiungendo nuovi server (scalabilità orizzontale). Nella maggior parte dei database, la scalabilità orizzontale spesso richiede una maggior cura nella gestione dei dati lato applicazione; al contrario, Elasticsearch nasce come sistema distribuito e quindi gestisce in maniera automatica nodi multipli senza che l'utente deve preoccuparsene lato applicazione.

2.1.1 Concetti fondamentali

Un *nodo* è un'istanza di Elasticsearch, mentre un *cluster* è costituito da uno o più nodi con lo stesso attributo *cluster.name* che lavorano insieme condividendo dati e workload. Quando dei nodi vengono aggiunti o rimossi dal cluster, questo si riorganizza per bilanciare il carico sui vari nodi. All'interno di un cluster, un nodo viene contrassegnato come *master*, cioè il nodo che si occupa di gestire le operazioni all'interno del cluster, come l'inserimento o la cancellazione di un indice, o l'aggiunta o la rimozione di un nodo nel cluster. Il nodo master non si occupa dei cambiamenti a livello di documento. Questo significa che avere solo un nodo master in un cluster non crea un collo di bottiglia quando il traffico di dati aumenta. L'utente può comunicare con uno qualsiasi dei nodi all'interno del cluster: ogni nodo sa dove sono memorizzati i dati e la richiesta viene instradata al nodo interessato.

Tra le meta-informazioni memorizzate da Elasticsearch, una delle più importanti è sicuramente il valore della *cluster health*, che può essere:

- *green*: tutti gli shard primari e le repliche sono attivi
- *yellow*: tutti gli shard primari sono attivi, ma non tutte le repliche
- *red*: non tutti gli shard primari sono attivi

Per aggiungere dati in Elasticsearch, dobbiamo prima creare un *indice*. Un indice è semplicemente un nome logico che punta ad uno o più *shard*. Uno shard è una *worker unit* che lavora a basso livello in cui viene memorizzata una parte dei dati dell'indice. I nostri dati sono salvati e indicizzati negli shard, ma la nostra applicazione non comunica direttamente con gli shard, bensì con gli indici. La gestione degli shard è il modo che Elasticsearch utilizza per distribuire i dati all'interno del cluster. I dati sono memorizzati negli shard e gli shard sono allocati ai nodi del cluster. Uno shard può essere uno shard primario o una replica (le repliche sono delle semplici copie degli shard primari). Il numero di shard primari viene definito nel momento di creazione dell'indice ed è fisso, mentre il numero di repliche può essere modificato. In Figura 2.1 è mostrato un cluster Elasticsearch con tre nodi (di cui uno master) e due repliche per ogni shard primario.

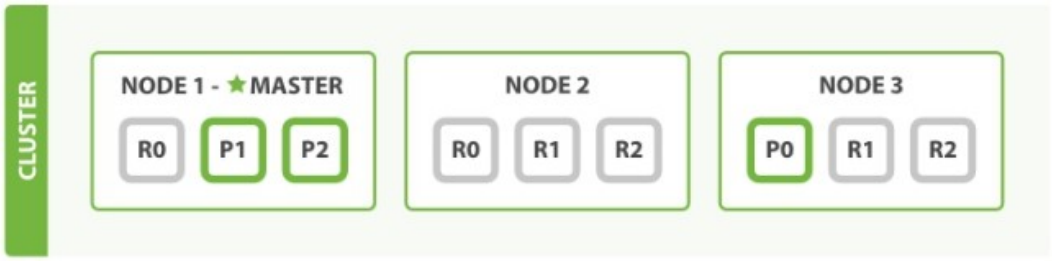


Figura 2.1: Cluster Elasticsearch con tre nodi e due repliche[11]

2.1.2 Dati in Elasticsearch

Nei database tradizionali i dati sono salvati con una struttura tabellare organizzata in righe e colonne, in cui ogni riga rappresenta un elemento e ogni colonna un attributo. Questo tipo di rappresentazione, tuttavia, è poco flessibile e non adatto a gestire dati di grandi dimensioni e soprattutto dati con attributi eterogenei. Per questo motivo, in Elasticsearch i dati sono organizzati come oggetti e rappresentati con la notazione JSON. Un oggetto

salvato in Elasticsearch viene chiamato *document*. Spesso i due termini oggetto e documento sono usati in maniera intercambiabile, anche se non sono esattamente la stessa cosa: un oggetto è una struttura dati JSON, mentre in Elasticsearch con il termine documento si intende il dato memorizzato all'interno di un indice con il suo ID univoco.

Nei documenti Elasticsearch, non vengono memorizzate solamente le informazioni utili all'utente finale, ma anche dei *metadati*, che sono delle informazioni riguardanti il documento salvato su Elasticsearch. Di seguito alcuni dei metadati più importanti:

- *_index*: un indice è come un database nei DBMS relazionali;
- *_type*: indica il tipo di dato che si sta memorizzando e serve ad Elasticsearch per capire come indicizzare il dato;
- *_id*: è una stringa che, combinata con *_index* e *_type*, identifica univocamente un documento in Elasticsearch. Quando viene inserito un nuovo documento, l'ID può essere specificato o assegnato da Elasticsearch.

2.2 Elasticsearch Python API

Per interagire con Elasticsearch, sono disponibili delle REST API. Durante tutto lo svolgimento di questo lavoro, sono state usate le API messe a disposizione da Elasticsearch per lo sviluppo di codice Python[12].
Prima di cominciare ad effettuare le analisi prestazionali delle ricerche, sono stati sviluppati alcuni notebook con Jupyter per capire come comunicare con un cluster Elasticsearch, utilizzando alcune delle API più comuni. Come prima cosa, quindi, è stato avviato un cluster multi-nodo in locale utilizzando Docker, eseguendo il comando *docker-compose*[13].
Di seguito una lista di esempi delle chiamate API più utilizzate.

2.2.1 Connessione al cluster

Per la connessione al cluster Elasticsearch, si crea un oggetto della classe Elasticsearch al quale vengono passati i seguenti parametri:

- Indirizzo del cluster Elasticsearch: in questo caso si effettua una connessione locale. La porta 9200 è quella di default.
- Certificato SSL

- Credenziali di accesso al cluster Elasticsearch. Per motivi di sicurezza, nell'esempio non sono passate in maniera esplicita ma vengono lette da un file salvato in locale.

```
es = Elasticsearch(
    "https://localhost:9200",
    ca_certs="ca.crt",
    basic_auth=(es_credentials.elastic_user,
    es_credentials.elastic_password))
```

2.2.2 Creazione e cancellazione di un indice

Una volta creato l'oggetto Elasticsearch, si utilizzano i metodi disponibili per effettuare le chiamate API.

```
mappings = {
    "properties" : {
        "firstName" : {
            "type" : "text",
            "fields" : {
                "keyword" : {
                    "type" : "keyword",
                    "ignore_above" : 256
                }
            }
        },
        "lastName" : {
            "type" : "keyword"
        }
    }
}
es.indices.create(index="python-client-test", mappings=mappings)
```

In questo caso viene creato un indice di nome *python-client-test* con i due campi $firstName$ e $lastName$ specificati nel mapping. Allo stesso modo, per cancellare un indice, c'è un'API altrettando semplice.

```
es.indices.delete(index="python-client-test")
```

2.2.3 Query

Una volta inseriti i nostri dati nel database, una delle operazioni più interessanti da effettuare è sicuramente quella di ricerca. Di seguito degli esempi di alcune delle tipologie di query più utilizzate, effettuate sull'indice sample fornito da Elasticsearch, *kibana_sample_data_flights*.

```
query = {
    "term" : {
        "DestCityName": "Catania"
    }
}
resp = es.search(index=index_name, query=query)
```

Le *term query* vengono usate per i campi di tipo *keyword*, perché recuperano solo i documenti che fanno un match esatto con il valore passato. In alternativa, si possono usare le *match query*, che effettuano la ricerca nei campi di tipo *text*. Nell'esempio si cercano all'interno dell'indice specificato tutti i documenti che hanno *Catania* come valore del campo *DestCityName*.

```
query = {
    "range": {
        "DistanceMiles": {
            "gt": 12000
        }
    }
}
resp = es.search(index=index_name, query=query)
```

Con le *range query* si possono cercare tutti i documenti che hanno il valore di un determinato campo all'interno di un intervallo specificato. In questo caso vengono recuperati tutti i documenti con un valore di *DistanceMiles* maggiore di 12000.

```
query = {
    "bool": {
        "must": [
            {
                "match": {
                    "OriginCityName": "Catania"
                }
            },
```

```
            {
                "match": {
                    "DestCityName": "Rome"
                }
            }
        ],
        "must_not": [
            {
                "match": {
                    "DestWeather": "Thunder & Lightning"
                }
            }
        ]
    }
}
resp = es.search(index=index_name, query=query)
```

Le *boolean query* sono sicuramente le query più utilizzate perché permettono di combinare assieme diverse clausole. Ci sono quattro tipi di operatori che possono essere usati:

- *must*: la clausola deve essere verificata dai documenti recuperati;
- *filter*: ha la stessa funzione della must, con l'unica differenza che in questo caso non viene influenzato lo *score* dei documenti recuperati;
- *should*: serve per modificare lo score dei documenti recuperati;
- *must_not*: la clausola non deve essere verificata dai documenti recuperati.

Nell'esempio vengono recuperati tutti i documenti che hanno $Catania$ come $OriginCityName$ e $Rome$ come $DestCityName$, ma che non hanno $Thunder\&Lightning$ come $DestWeather$.

2.3 Ricerche kNN in Elasticsearch

Per eseguire una ricerca kNN è necessario convertire i dati in vettori. Questi vettori devono essere costruiti fuori da Elasticsearch e aggiunti ai documenti in un campo di tipo *dense_vector*. La stessa trasformazione deve essere effettuata per le query. I vettori devono essere costruiti in maniera tale che più il vettore di un documento è vicino al vettore di una query, in base a una metrica di distanza, migliore sarà il suo score.

Elasticsearch supporta due metodi per la ricerca kNN[14]:

- Ricerca kNN approssimata utilizzando l'API *kNNsearch*
- Ricerca kNN esatta usando una *script_score* query con una funzione vettoriale

La ricerca approssimata offre dei tempi di risposta minori al costo di un'indicizzazione più lenta e un margine di errore nei documenti restituiti. La ricerca esatta garantisce i risultati precisi ma non scala bene con dataset di dimensioni importanti.

2.3.1 Ricerca approssimata

Per effettuare una ricerca kNN approssimata in Elasticsearch si usa l'API *kNNsearch* per ricercare in un campo di tipo *dense_vector* con l'indicizzazione attivata. Nella definizione dell'indice si deve quindi definire almeno un campo *dense_vector* e sono inoltre richieste le seguenti opzioni di mapping:

- il campo *index* impostato a *true*;
- un valore di *similarity*. Questo valore determina la metrica di distanza usata per calcolare la distanza tra i vettori dei documenti e quello di ricerca. Le metriche supportate sono:
  - Distanza euclidea
  - Prodotto scalare
  - Similarità coseno

Oltre ai campi *index* e *similarity* (entrambi obbligatori per effettuare una ricerca kNN approssimata), gli altri parametri di un *dense_vector* sono:

- *dims* (Required, integer): dimensione del vettore. Può assumere un valore massimo di 1024 per i vettori con il campo *index* a *true*, o 2048 in caso contrario;
- *index_options* (Optional, object): una sezione opzionale che configura l'algoritmo di indicizzazione. Quando viene passato questo campo, devono essere definite tutte le sue proprietà:
  - *type* (Required, string): il tipo di algoritmo da usare. Attualmente è supportato solo HNSW;
  - *m* (Required, integer): il numero di vicini a cui ogni nodo è connesso nel grafo HNSW. Il valore di default è 16;

– $ef\_construction$ (Required, integer): il numero di candidati da tracciare durante la costruzione della lista dei nearest neighbors per ogni nuovo nodo. Il valore di default è 16.

Per ottenere i risultati, l'API di ricerca approssimata trova un numero pari a $num\_candidates$ di nearest neighbors approssimati per ogni shard. Viene calcolata la distanza dei vettori di questi candidati dal vettore della query e vengono calcolati i k risultati più vicini per ogni shard. Alla fine i risultati ottenuti da ogni shard vengono messi insieme e vengono restituiti i k nearest neighbor globali. Il valore di $num\_candidates$ può essere aumentato per migliorare la precisione dei risultati al costo di tempi di ricerca maggiori: infatti aumentando questo valore verranno presi in considerazione più candidati per ogni shard, di conseguenza per la ricerca servirà più tempo ma ci saranno maggiori probabilità di trovare gli effettivi k nearest neighbors.

2.3.2 Ricerca esatta

Per esegure una ricerca kNN esatta si utilizza una $script\_score$ query con una funzione vettoriale. Anche in questo caso è necessario un campo di tipo $dense\_vector$. A differenza di quanto succede nelle ricerche approssimate, però, è consigliabile impostare il valore di $index$ a $false$ se non si intende fare ricerche approssimate. Così facendo si otterranno notevoli miglioramenti in termini di velocità di indicizzazione.
Una volta indicizzati i nostri dati, si può effettuare una ricerca kNN esatta attraverso l'API $search$ usando una $script\_score$ query contenente una funzione vettoriale tra quelle supportate:

- $cosineSimilarity$: calcola la similarità coseno
- $dotProduct$: calcola il prodotto scalare
- $l1norm$: calcola la distanza di Manhattan
- $l2norm$: calcola la distanza euclidea

2.4 Risultati esecuzione in locale

Prima di eseguire i test finali sull'infrastruttura cloud, sono stati effettuati gli stessi test, con una mole di dati decisamente minore, in locale. Per fare ciò, è stato sviluppato del codice che crea un dataset con la stessa struttura dei dati sui quali sono stati effettuati i test finali, assegnando dei valori casuali. In particolare, gli elementi salvati su Elasticsearch avranno i seguenti attributi:

- *sentence_key*: stringa contenente identificativo univoco della frase;
- *sentence*: stringa contenente la frase nel formato originale;
- *embedding*: vettore di dimensione 512 in cui è stata mappata la frase.

2.4.1 Tempi di inserimento

Per poter effettuare una ricerca, prima di tutto servono dei dati su cui effettuarla. Quindi, come prima cosa sono stati calcolati i tempi di riempimento di un indice nei due casi: nel caso di indicizzazione del campo *embedding* (per la ricerca approssimata) e nel caso in cui non viene effettuata l'indicizzazione di questo campo, per vedere quanto questo impatta sui tempi di inserimento. L'inserimento dei dati è stato inoltre diviso in due parti per capire se e quanto impatta a livello di tempo la presenza di documenti all'interno dell'indice rispetto all'inserimento di documenti in un indice vuoto. Sono quindi inseriti in un primo momento N documenti in un indice vuoto e successivamente D documenti nell'indice precedentemente riempito. Nello specifico questi test sono stati effettuati con N pari a 8000 e D pari a 4000 ed ogni inserimento è stato eseguito 3 volte consecutive per cercare di ottenere dei risultati più stabili. Di seguito i risultati dei test effettuati.

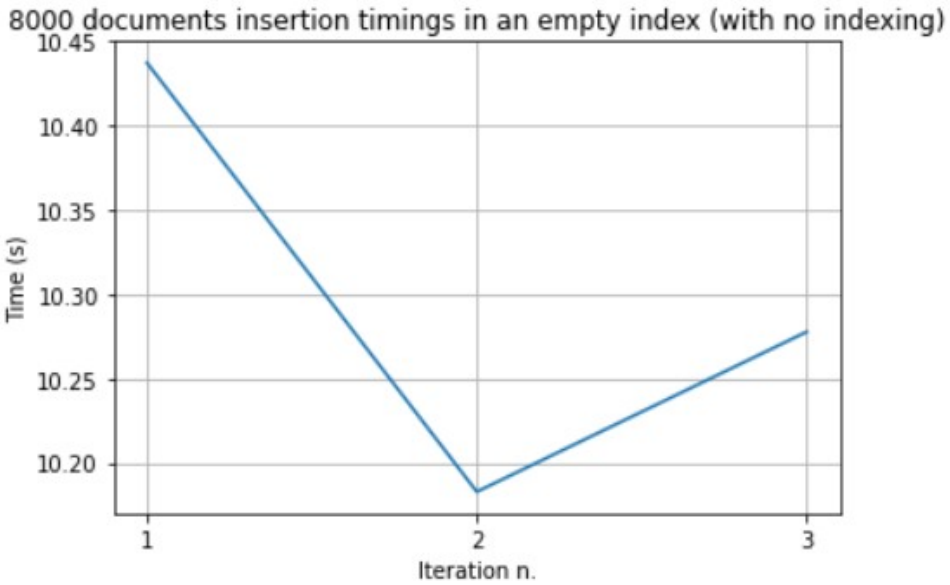


Figura 2.2: Inserimento di 8000 documenti in un indice vuoto con l'indicizzazione del vettore disattivata

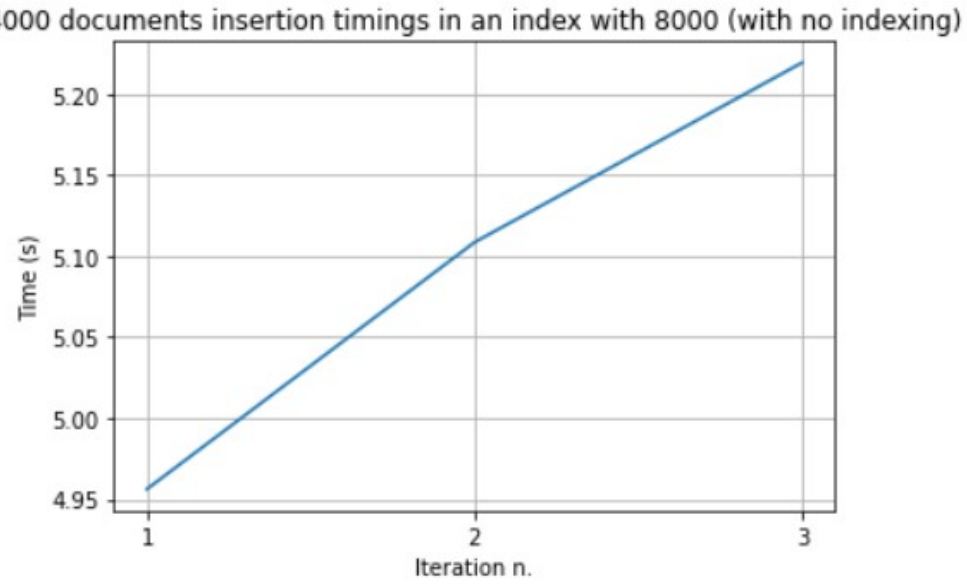


Figura 2.3: Inserimento di 4000 documenti in un indice pieno con l'indicizzazione del vettore disattivata

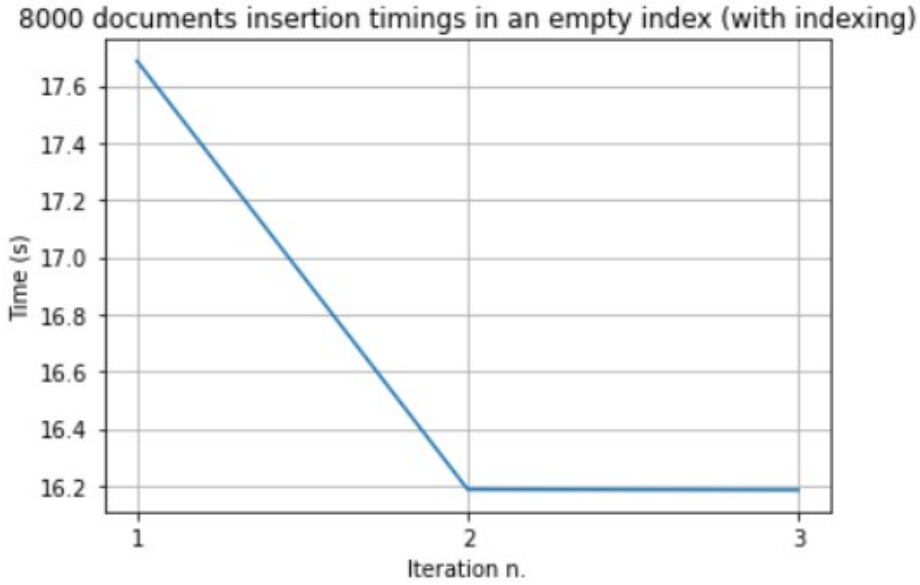


Figura 2.4: Inserimento di 8000 documenti in un indice vuoto con l'indicizzazione del vettore attivata

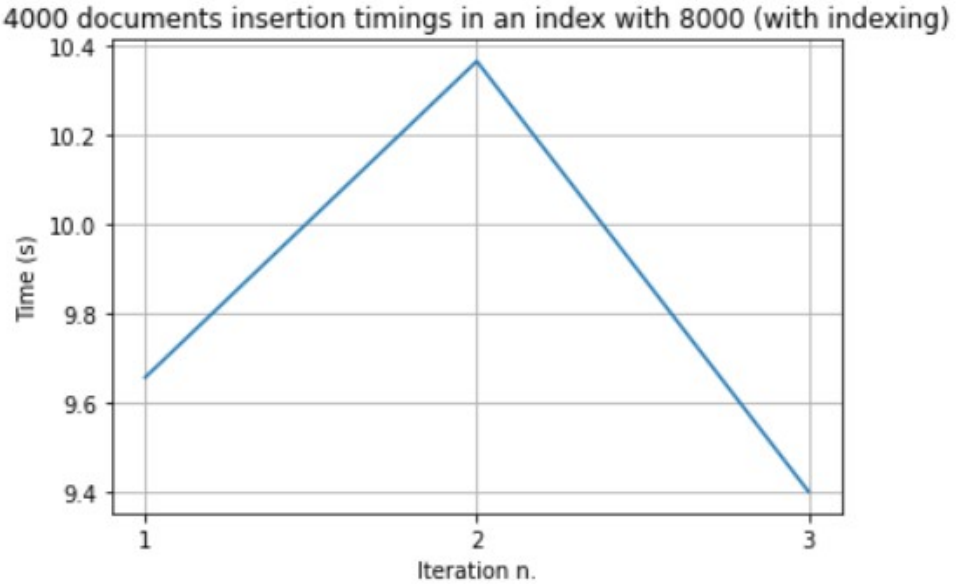


Figura 2.5: Inserimento di 4000 documenti in un indice pieno con l'indicizzazione del vettore attivata

Dai risultati ottenuti possiamo vedere come effettivamente c'è un netto aumento dei tempi di inserimento nel caso di indicizzazione del vettore di embedding (passiamo dai circa 10 secondi nel primo caso ai circa 17 secondi nel secondo), mentre la presenza di documenti nell'indice al momento dell'inserimento di nuovi dati non sembra impattare particolarmente, almeno con questa mole di dati.

2.4.2 Tempi di ricerca esatta

Tutti i test effettuati di seguito, sono stati fatti sugli indici precedentemente riempiti con 12000 documenti di prova. Gli indici su cui sono stati effettuati i test presentano la stessa struttura su cui sono stati successivamente effettuati i test in remoto: 3 shards e 1 replica.

Per il calcolo dei tempi di ricerca esatta, sono stati presi 5 vettori di prova e, per ogni valore di k, sono state effettuate 5 ricerche esatte, di cui poi è stata fatta la media. Come possiamo notare dal grafico e come ci potevamo aspettare, i tempi di risposta crescono all'aumentare del valore di k.

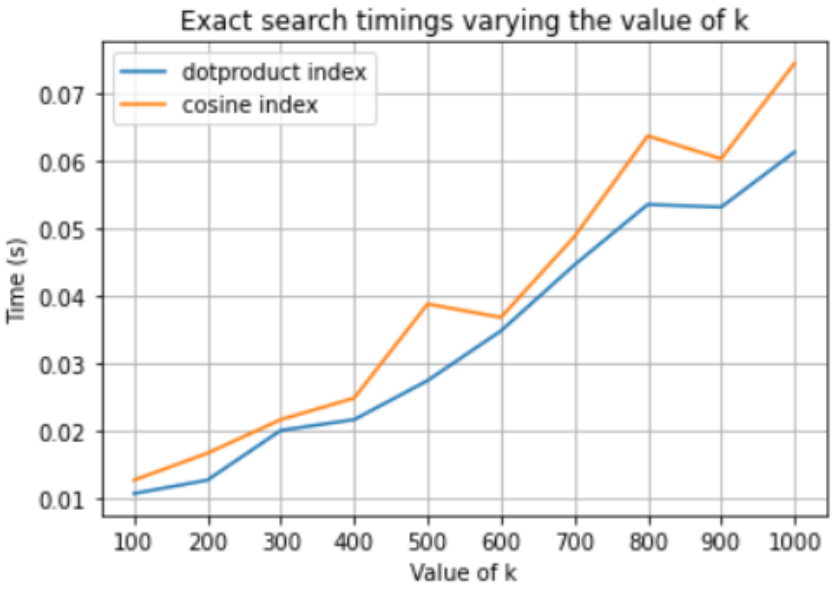


Figura 2.6: Tempi di ricerca esatta al variare di k (esecuzione in locale)

2.4.3 Tempi di ricerca approssimata

Per quanto riguarda i tempi di ricerca approssimata, sono stati effettuati diversi test in quanto i parametri da far variare questa volta erano 2: k e num_candidates. Ad ogni iterazione, facendo variare il valore di k, sono state effettuate 5 ricerche approssimate con 5 vettori di ricerca diversi ed è stata calcolata la media. La stessa operazione è stata fatta con valori di num_candidates crescenti (1, 3, 5 e 10). Anche in questo caso, possiamo notare come i tempi di ricerca aumentano al crescere di k, mentre con dataset di queste dimensioni non sembrano ancora apprezzabili le differenze di tempo aumentando il valore di num_candidates e nemmeno l'incremento dei tempi rispetto alla ricerca esatta. Confrontando i tempi ottenuti con i due tipi di indice, invece, sembra che l'indice con similarità coseno ci metta leggermente più tempo.

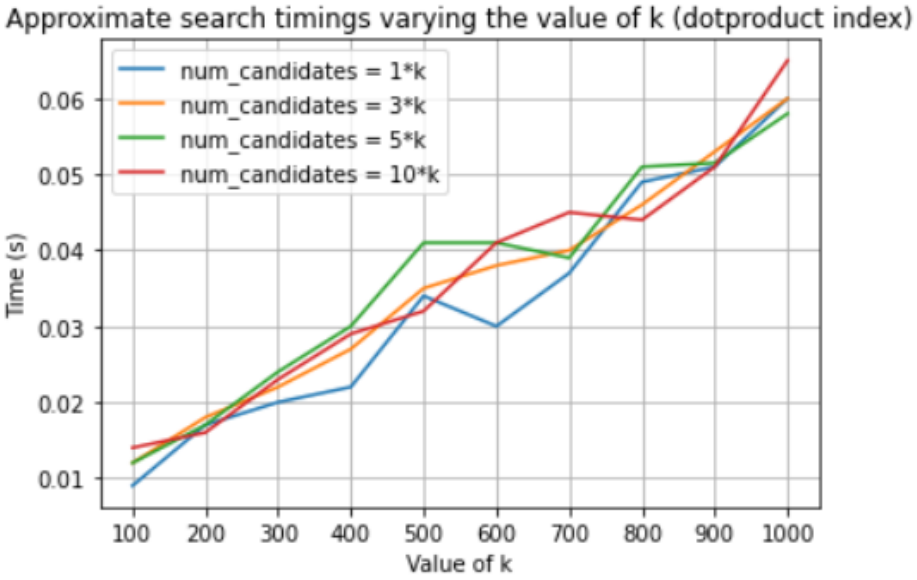


Figura 2.7: Tempi di ricerca approssimata al variare di k (indice dotproduct, esecuzione in locale)

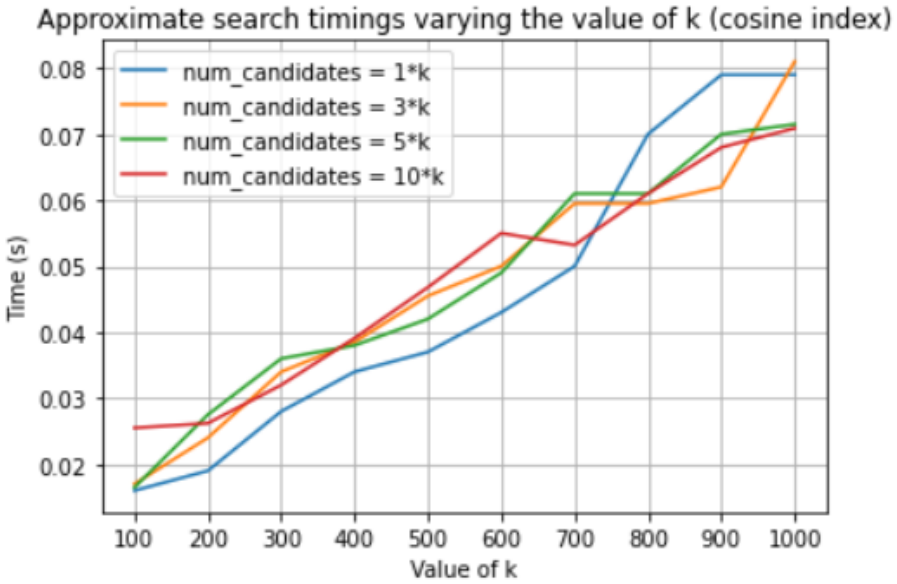


Figura 2.8: Tempi di ricerca approssimata al variare di k (indice cosine, esecuzione in locale)

2.4.4 Recall

Per capire quanto precisa sia effettivamente la ricerca approssimata, è stato calcolato il valore di recall. Prendendo come riferimento i documenti ritornati dalla ricerca esatta (positivi), viene definita come il numero di veri positivi (documenti ritornati dalla ricerca approssimata che son in effetti positivi), diviso la totalità dei positivi. Il calcolo è stato effettuato anche in questo caso variando i due parametri della ricerca approssimata. Per ogni iterazione sono state effettuate 5 ricerche esatte e 5 ricerche approssimate, calcolato il valore di recall e salvato il valore medio. La stessa operazione è stata effettuata con valori crescenti di num_candidates. Dai risultati ottenuti possiamo notare come, tenendo costante il valore di num_candidates, il valore della recall cresce all'aumentare del valore di k; mentre, tenendo costante il valore di k, il valore della recall cresce all'aumentare del valore di num_candidates. Mentre confrontando i due tipi di indici, sembra essere leggermente più precisa la ricerca effettuata sull'indice con similarità coseno.

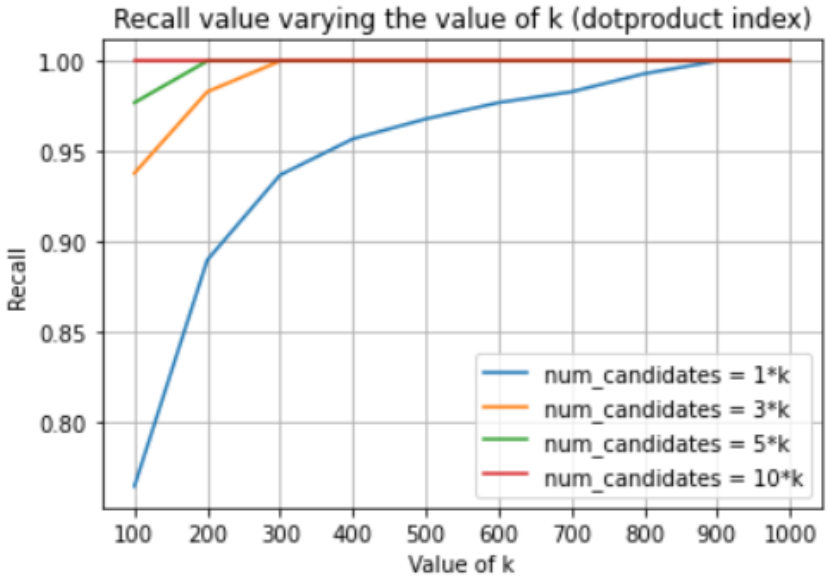


Figura 2.9: Valore della recall al variare di k (indice dotproduct, esecuzione locale)

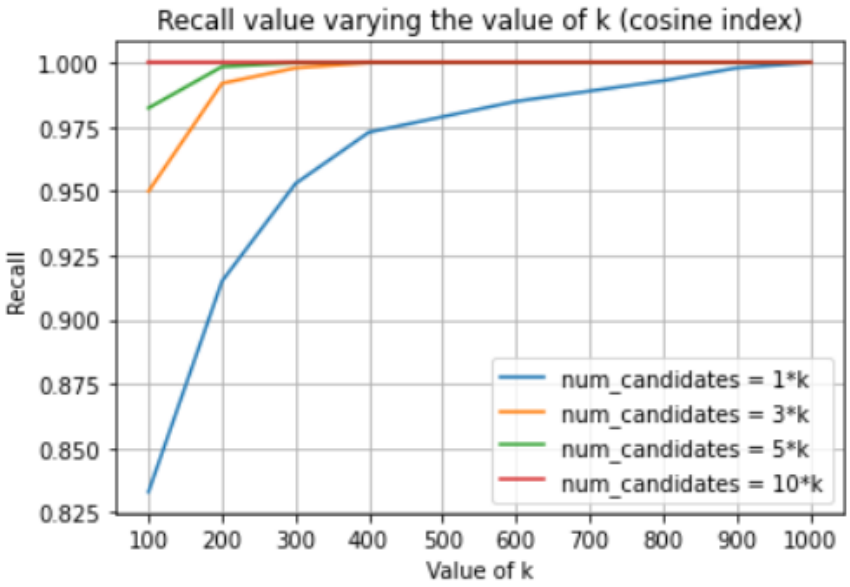


Figura 2.10: Valore della recall al variare di k (indice cosine, esecuzione locale)

2.4.5 Errore relativo nello score

L'ultimo test effettuato è stato il calcolo dell'errore relativo sullo score dei documenti restituiti dalla ricerca approssimata confrontati con lo score dei documenti in comune con la ricerca esatta. L'errore relativo è calcolato come:

$$E_r = \frac{|S_E - S_A|}{S_E}$$

dove S_E è il valore dello score ottenuto dalla ricerca esatta, mentre S_A è lo score ottenuto dalla ricerca approssimata.

Come possiamo notare dai risultati ottenuti, in tutti i casi l'errore relativo medio è nell'ordine di 10^{-7}, quindi un errore relativo sicuramente trascurabile perchè compatibile con la precisione di macchina relativa a dati in singola precisione, che è attorno a 10^{-7}.

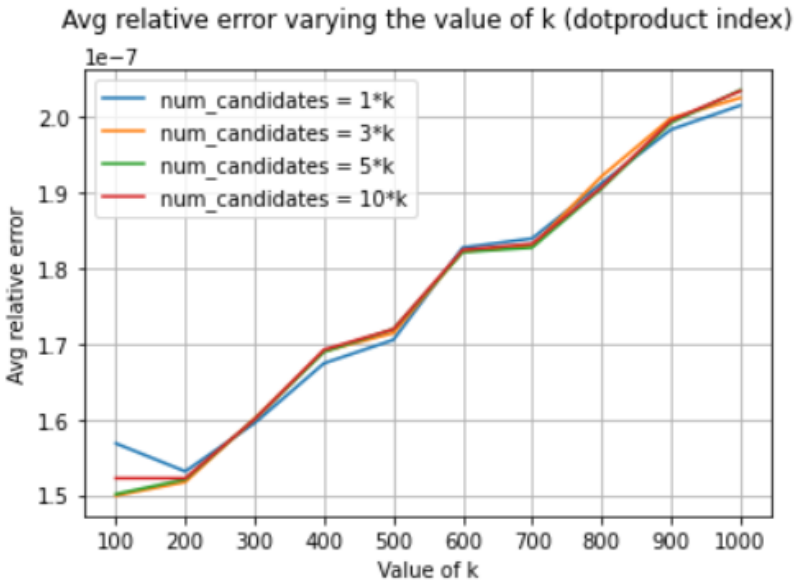


Figura 2.11: Errore relativo medio dello score al variare di k (indice dotproduct, esecuzione locale)

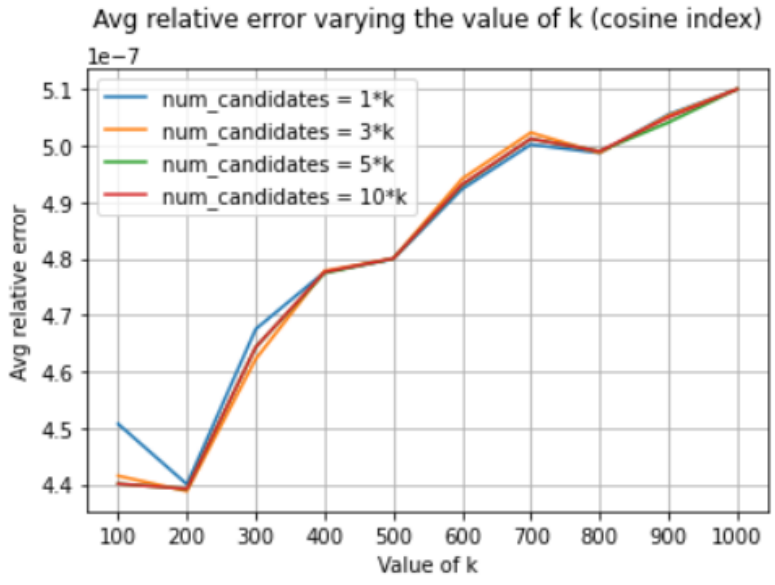


Figura 2.12: Errore relativo medio dello score al variare di k (indice cosine, esecuzione locale)

Capitolo 3

Infrastruttura cloud di sperimentazione

In questo capitolo vengono descritte tutte le operazioni finalizzate all'inserimento dei dati sul cluster Elasticsearch in remoto. Prima di tutto viene presentata l'architettura su cui sono stati eseguiti i test, realizzata utilizzando la Google Cloud Platform. Successivamente viene fatta una panoramica sugli strumenti utilizzati per la realizzazione della pipeline: Apache Beam e Dataflow. Infine viene descritta nel dettaglio la pipeline implementata per l'ingestion dei dati.

3.1 Descrizione dell'architettura

Dopo aver completato la parte di sperimentazione in locale, sono stati eseguiti gli stessi test sul cloud, utilizzando Google Cloud Platform (GCP)[15], una suite di servizi di cloud computing che gira sulla stessa infrastruttura che Google utilizza internamente per i suoi prodotti per gli utenti finali, come Ricerca Google, Gmail, archiviazione di file e YouTube. Tra i servizi messi a disposizione da Google, quelli utilizzati per questo progetto sono principalmente tre: Google Cloud Storage per la memorizzazione dei dati, Compute Engine per la creazione delle macchine virtuali e Dataflow per l'implementazione delle pipeline ETL[16].

Google Cloud Storage (GCS) è una piattaforma di storage in cloud basata sul filesystem *Colossus*[17]. È possibile realizzare *bucket* GCS, che rappresentano l'astrazione di un singolo disco. Ogni bucket è accessibile dall'URI *gs://nome-bucket/* attraverso le API messe a disposizione. Come il filesystem di Hadoop[18], è ottimizzato per le operazioni write-once, read-many.

Compute Engine rappresenta una piattaforma di virtualizzazione in cloud. Si possono creare macchine virtuali, configurando la potenza delle singole macchine,

capacità di memoria, presenza o meno di scheda video, tipi e dimensioni dei dischi e altri parametri. Si dispone anche di una pratica interfaccia di collegamento diretto in SSH a ciascuna macchina.

Per l'installazione di Elasticsearch in remoto, sono state create quattro macchine virtuali tramite il tool di Compute Engine della GCP: tre per i nodi del cluster e una per Kibana, l'interfaccia grafica messa a disposizione da Elastic Stack per un'interazione più intuitiva con Elasticsearch. Come tipologia di macchina, sono state utilizzate quelle messe a disposizione dalla GCP[19]; in particolare, per i tre nodi è stata utilizzata la tipologia *n2-standard-2*, mentre per Kibana è stata utilizzata la tipologia *e2-micro*. Le quattro macchine sono connesse ad una rete Virtual Private Cloud (VPC) e comunicano tra di loro utilizzando un indirizzo IP interno. Per la comunicazione con l'esterno invece, non avendo asssegnato indirizzi IP esterni, viene utilizzata una cloud NAT. La comunicazione con la macchina virtuale su cui è in esecuzione Kibana è stata abilitata utilizzando il forwarding TCP di Identity-Aware Proxy (IAP), che ci permette di stabilire una connessione SSH. Con queste due accortezze si ottiene sia la connettività in uscita, attraverso il cloud NAT, che l'accesso tramite autenticazione e autorizzazione a livello di rete. In aggiunta, viene comunque utilizzato uno schema di autenticazione a livello applicativo (username e pasword) gestito da Elasticsearch. In Figura 3.1 è mostrata l'architettura utilizzata sulla GCP.

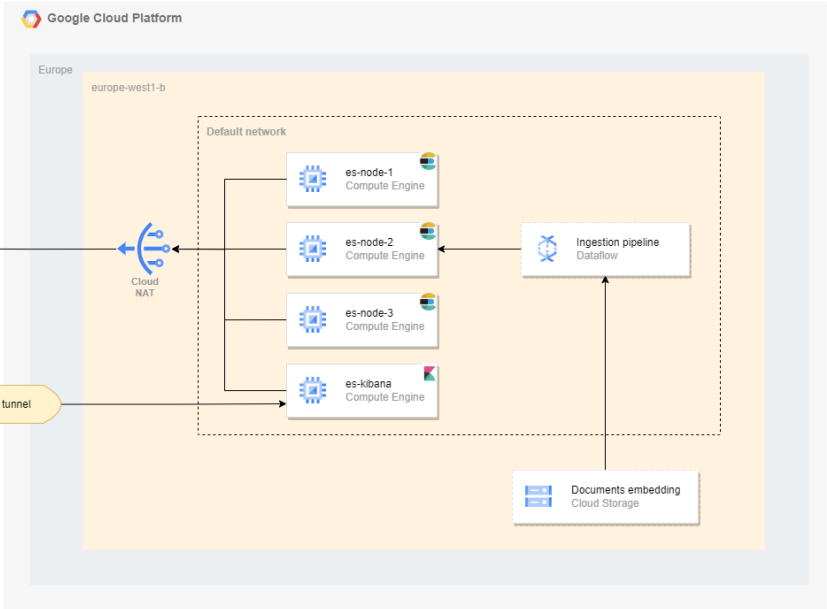


Figura 3.1: Architettura cluster Elasticsearch su GCP

3.2 Dataflow e Apache Beam

Dataflow è un servizio di Google Cloud di tipo serverless, cioè completamente gestito, per l'elaborazione di flussi di dati sia real-time che batch. Riduce al minimo la latenza, scala automaticamente e ottimizza i costi dando la possibilità di pagare solo per le risorse realmente utilizzate, attraverso uno schema pay-per-use. Le caratteristiche principali di Google Dataflow sono:

- elaborazione rapida, unificata ed illimitata di dati in modalità flusso e batch;
- provisioning automatico delle risorse computazionali;
- scalabilità automatica ed orizzontale;
- affidabilità e coerenza dei dati trattati;
- gestione delle pipeline;
- gestione dei workload per abbattere i costi grazie alla flessibilità delle risorse;
- partizionamento e ridistribuzione automatica dei workload.

Dataflow supporta una completa integrazione con Apache Beam. Apache Beam è un framework open source per la definizione ed esecuzione delle pipeline per il processamento dei dati sia in batch che in streaming[20]. Una pipeline in batch è caratterizzata dal fatto di avere tutti i dati disponibili al momento in cui la pipeline viene lanciata in esecuzione. Al contrario, una pipeline in streaming offre la possibilità di gestire flussi virtualmente illimitati, come nel caso di flussi dati che vengono generati continuamente. Utilizzando uno degli SDK di Apache Beam viene definita la pipeline e successivamente questa viene eseguita da un framework di esecuzione, chiamato *runner*, come Dataflow. Questo modello consente di concentrarsi sulla composizione logica della pipeline piuttosto che sulla gestione dell'elaborazione parallela. È possibile, cioè, concertrarsi sul *cosa* fare piuttosto che sul *come* farlo. In Figura 3.2 è mostrato il modello di programmazione Beam, composto da quattro blocchi. Per lo svolgimento di questo lavoro, sono stati utilizzati:

- *Processing type*: batch. Tutti i dati da processare sono disponibili al momento in cui viene lanciata la pipeline;
- *Runner*: Dataflow;
- *SDK*: Python;

- *I/O Connector*: FileIO ed ElasticsearchIO (è stato implementato manualmente in quanto non disponibile per l'SDK Python).

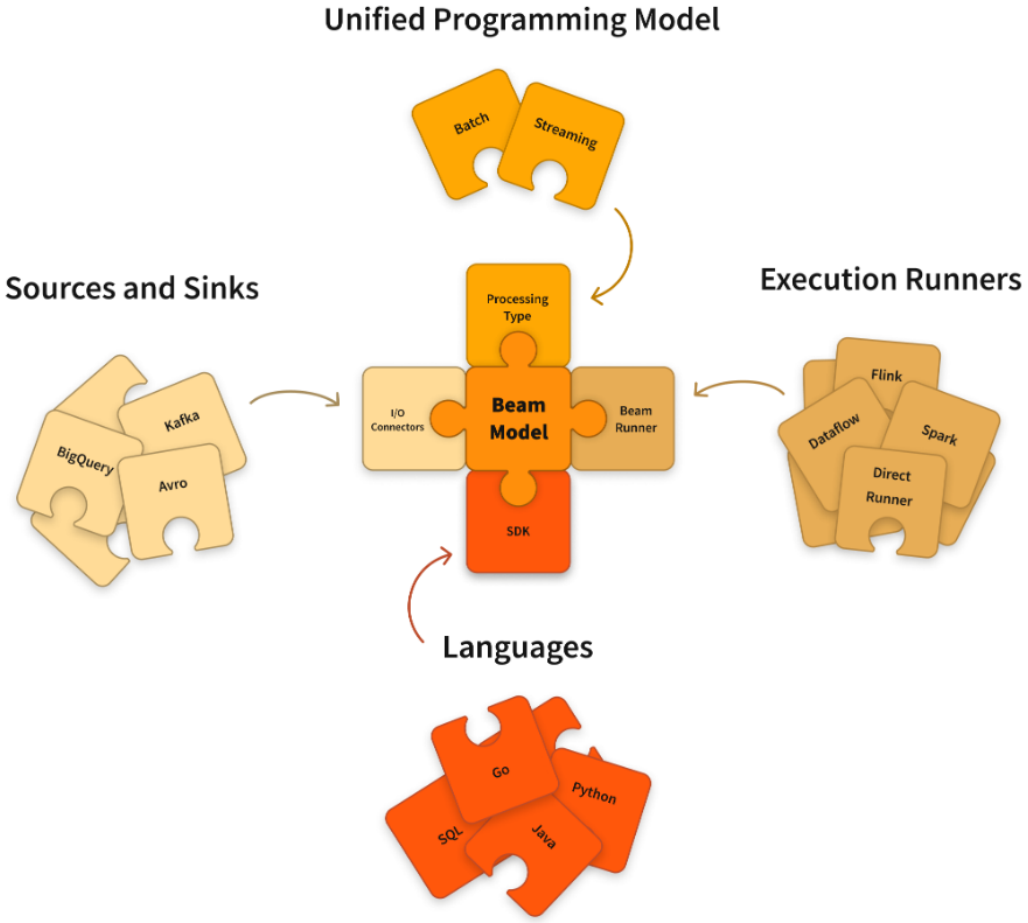


Figura 3.2: Modello di programmazione Beam[21]

I concetti alla base del modello di programmazione Beam sono principalmente tre: *Pipeline*, *PCollection* e *PTransform*.
Una pipeline Beam è un grafo aciclico diretto[22] di tutti i processi di elaborazione, partendo quindi dalla lettura dei dati in input, alle trasformazioni eseguite sui dati fino alla scrittura dei dati di output.
Una PCollection è un insieme di elementi non ordinato e rappresenta l'unità elementare del dato in Beam. Ogni PCollection appartiene ad una specifica pipeline e non può essere condivisa tra più pipeline. Le caratteristiche di una PCollection sono:

- *Tipo*: gli elementi di una PCollection possono essere di qualsiasi tipo, ma all'interno della stessa PCollection devono essere tutti dello stesso tipo;

- *Immutabilità*: una PCollection è immutabile. Una volta creata, non è possibile aggiungere, rimuovere o modificare elementi. Nelle fasi di processamento della pipeline, le PTransform processano gli elementi di una PCollection e ne generano una nuova, senza modificare quella di partenza;

- *Accesso*: non è supportato l'accesso casuale agli elementi della PCollection. Gli elementi all'interno di una PCollection vengono iterati in un ordine prestabilito che non può essere variato;

- *Dimensione*: una PCollection è un insieme di elementi di dimensione non determinabile a priori. Può infatti contenere un insieme finito ma anche infinito di elementi.

Una PTransform rappresenta l'operazione di processamento del dato nella pipeline. Viene applicata ad una o più PCollection in input (con l'unica eccezione di quella che legge i dati in input) e genera una o più PCollection in output. Le operazioni di elaborazione vengono definite dall'utente tramite gli SDK di Beam e vengono applicate ad ogni elemento della PCollection di input. La PTransform più utilizzata è la *ParDo*. Una ParDo è una trasformazione generica, simile alla fase di *Map* nel paradigma *MapReduce*[23]: riceve tutti gli elementi della PCollection in input, in porzioni dette bundle, esegue una funzione definita dall'utente e genera zero, uno o più elementi nella PCollection di output. È molto utile per una varietà di operazioni di processamento del dato abbastanza comuni, come:

- filtraggio di un dataset;

- formattazione o conversione del tipo di ogni elemento di un dataset;

- estrazione di alcune parti degli elementi di un dataset;

- esecuzione di operazioni sui singoli elementi di un dataset.

La definizione della logica di elaborazione di una ParDo viene fatta dall'utente sotto forma di una classe derivata da ParDo, dove viene implementato il metodo *process*.

3.3 Descrizione della pipeline

Una volta definita l'architettura cloud di sperimentazione, è stata sviluppata la pipeline per l'inserimento dei dati su Elasticsearch. I dati su cui si è lavorato sono dei documenti di testo. Ogni frase presente all'interno di questi documenti è stata mappata in un vettore di dimensione 512 e memorizzata con i seguenti attributi:

- *sentence_key*: stringa contenente identificativo univoco della frase;
- *sentence*: stringa contenente la frase nel formato originale;
- *embedding*: vettore di dimensione 512 in cui è stata mappata la frase.

per un totale di 10.126.704 frasi organizzate in una lista di file *parquet* salvati in dei bucket GCS.

La pipeline di *ingestion*, la cui struttura è mostrata in Figura 3.3, è stata implementata con l'SDK Python di Apache Beam ed eseguita con Dataflow.

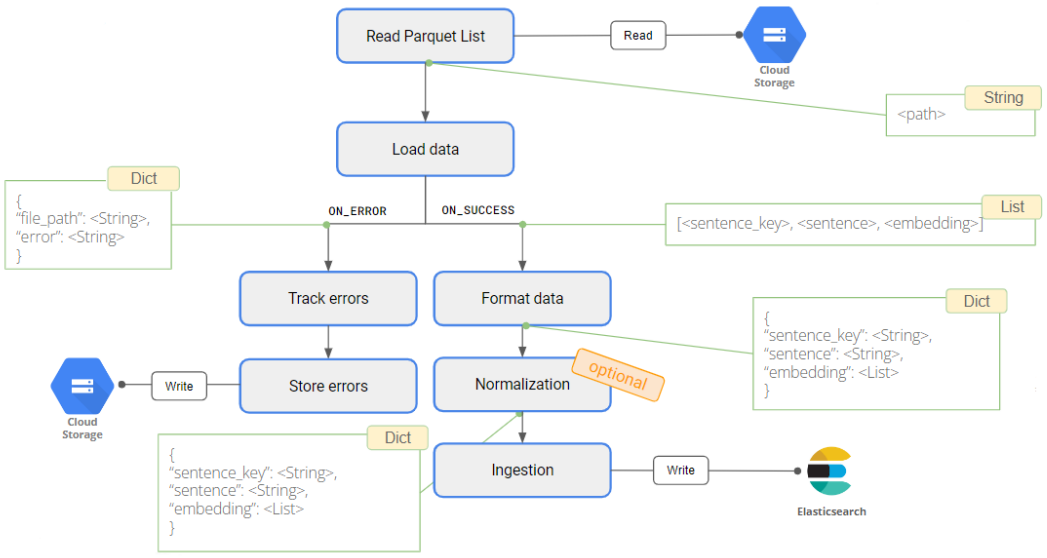


Figura 3.3: Pipeline in fase di design

Il file di input della pipeline è un file *csv*, salvato su GCS, contenente in ogni riga l'URI di un file parquet. La prima operazione della pipeline legge quindi il file csv e genera una PCollection con una lista di URI.

```
parquet_list = pipeline | "Read parquet list" >> beam.io.ReadFromText(
    known_args.input, skip_header_lines=True
)
```

Una volta letto il file di input e ottenuta la lista degli URI, si procede andando ad estrarre i dati contenuti all'interno dei file parquet.

```
data = (
    parquet_list | "Load data" >>
    beam.ParDo(FromCsvToParquet()).with_outputs()
)
```

L'output della trasformazione *Load data* può seguire due strade diverse in base all'esito della lettura del file parquet: in caso di errori in lettura (ad esempio nel caso di URI non valido) la pipeline continuerà sul ramo *ON_ERROR* e si andrà a creare un file csv salvato su un bucket GCS contenente gli URI che hanno dato problemi.

```
_ = (
    data["on_error"]
    | "Track errors" >> beam.ParDo(ErrorTracking())
    | "Store errors" >> WriteToText(known_args.errors_log_output)
)
```

Da notare come la PCollection di output in questo caso non viene salvata in nessuna variabile, in quanto non si devono più effettuare operazioni su di essa. In caso di esito positivo, invece, si procede con il ramo *ON_SUCCESS*. La PCollection creata con i parquet correttamente letti, viene data in input alla PTransform *Format data*, per una formattazione dei dati che permette l'inserimento su Elasticsearch.

```
formatted_data = (
    data["on_success"] | "Format data" >> beam.Map(format_result)
)
```

Una volta formattati i dati, si può procedere con l'inserimento su Elasticsearch. Prima però c'è un'operazione opzionale che la pipeline esegue solo in caso si stesse provando a riempire un indice con funzione vettoriale prodotto scalare: in questo caso infatti si deve prima effettuare la normalizzazione di tutti i vettori.

```
if known_args.normalize:
    final_data = formatted_data
    | "Normalization" >> beam.Map(normalize_vector)
else:
    final_data = formatted_data
```

L'ultima operazione eseguita dalla pipeline è la scrittura su Elasticsearch.

```
final_data | "Ingestion" >> beam.ParDo(
    WriteToES(
        known_args.es_url,
        known_args.es_user,
        known_args.es_password,
        known_args.index_name,
        known_args.ca_cert,
    )
)
```

Tutti gli argomenti utilizzati dalla pipeline sono passati da linea di comando e gestiti utilizzando la libreria Python *argparse*[24]. Le credenziali di accesso al cluster Elasticsearch sono gestite tramite il Secret Manager, sistema di archiviazione sicuro messo a disposizione dalla GCP.

Capitolo 4

Risultati ottenuti sul cloud

In questo capitolo vengono riportati tutti i risultati e i grafici ottenuti effettuando i test sull'infrastruttura cloud. Nella prima parte vengono messi a confronto i risultati ottenuti con le due tipologie di indice dall'esecuzione della pipeline per l'inserimento dei dati. Successivamente vengono analizzate le differenze in termini di tempo di ricerca, valore della recall ed errore relativo nello score.

4.1 Grafici e tempistiche di ingestion

Come descritto nel capitolo precedente, per l'inserimento dei dati sul cluster Elasticsearch remoto è stata implementata una pipeline Apache Beam eseguita da Dataflow. Il numero totale di documenti in input è di 10.126.704. Per la creazione degli indici sul cluster Elasticsearch remoto, sono stati creati degli *index template*[25], funzionalità di Elasticsearch che permette di automatizzare la struttura degli indici al momento della creazione. In particolare, gli indici su cui sono stati effettuati i test sono indici con numero di repliche pari a 1 e 3 shards. Tuttavia, per cercare di ridurre quanto più possibile i tempi di indicizzazione, è stato effettuato un *tuning* degli indici[26], impostando a 0 il numero di repliche e il tempo di refresh a 1 minuto. Una volta completato l'inserimento dei dati, prima di effettuare le ricerche, sono stati ripristinati i valori dell'indice, impostando di nuovo a 1 il numero di repliche e a 1 minuto il tempo di refresh.
Prima di lanciare la pipeline con la totalità dei dati, sono stati effettuati degli inserimenti con una porzione crescente di dati: prima dando in input 1000 parquet, poi 10000 e infine 100000. Aumentando la quantità di dati in input, si sono presentati alcuni problemi: la RAM dei worker di Dataflow cominciava a saturarsi e il cluster Elasticsearch cominciava a rispondere con alcuni *ConnectionTimeout*. Per risolvere il primo problema sono state utilizzate

delle macchine con più RAM (passando dalla *n1-standard-1*, con 3.75GB di RAM alla *n2-highmem-2*, con 16GB di RAM); mentre per cercare di limitare il numero di ConnectionTimeout, è stato implementato un sistema di *retry* di tipo exponential backoff: ogni volta che si presenta un ConnectionTimeout, il sistema si mette in pausa per un periodo di tempo (partendo da 2 minuti, raddioppiando ogni volta in caso di errori ripetuti fino ad un massimo di 16 minuti, cioè provando ad inserirli per un massimo di 5 volte) per poi riprovare ad inserire i dati.
Di seguito i risultati ottenuti.

4.1.1 Indice prodotto scalare

Il primo tentativo è stato effettuato impostando il numero di worker massimo a 5, ottenendo i seguenti risultati:

- Tempo di esecuzione della pipeline: 3 ore e 7 minuti
- Numero di documenti inseriti: 4743789
- Dimensione dei dati iseriti: circa 50GB (solo shards primari)
- Numero di bulk insert fallite: 15

Successivamente si è provato ad abbassare il numero di worker massimo per cercare di abbassare la frequenza di richieste inoltrate ad Elasticsearch, ottenendo i seguenti risultati:

- Tempo di esecuzione della pipeline: 4 ore e 30 minuti
- Numero di documenti inseriti: 6123399
- Dimensione dei dati inseriti: circa 62GB (solo shards primari)
- Numero di bulk insert fallite: 13

Diminuendo il numero di workers si sono ottenuti dei miglioramenti in termini di documenti inseriti, passando da 4743789 a 6123399. Il numero totale di documenti in input tuttavia era di 10.126.704. Per indagare il motivo dietro alla differenza tra il numero di documenti in input e quelli inseriti su Elasticsearch, è stato utilizzato il sistema di *logging* della GCP e si è trovato che i documenti "persi" possono essere divisi in due categorie:

1. Documenti che hanno dato ConnectionTimeout dopo 5 tentativi di retry

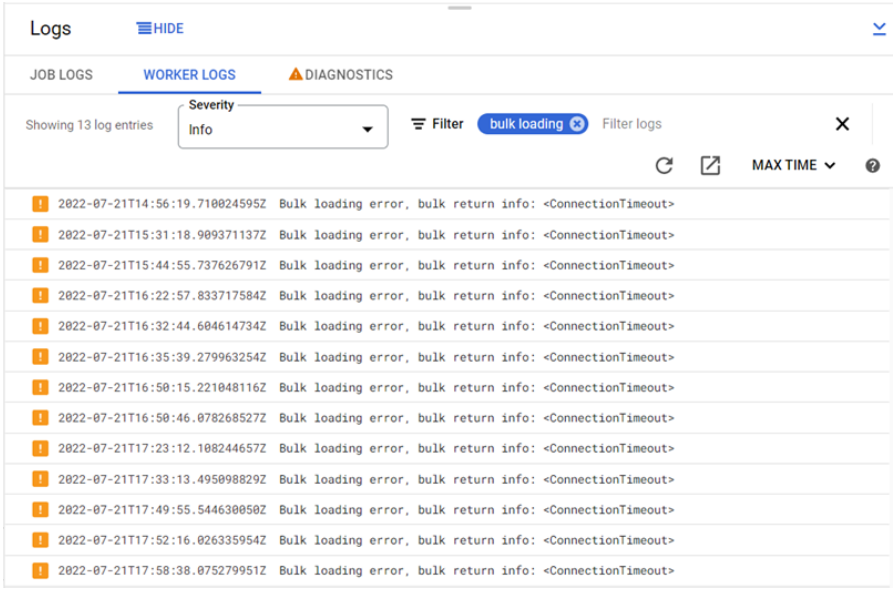


Figura 4.1: Documenti non inseriti per ConnectionTimeout

2. Documenti che l'embedder ha mappato come vettori pieni di zeri

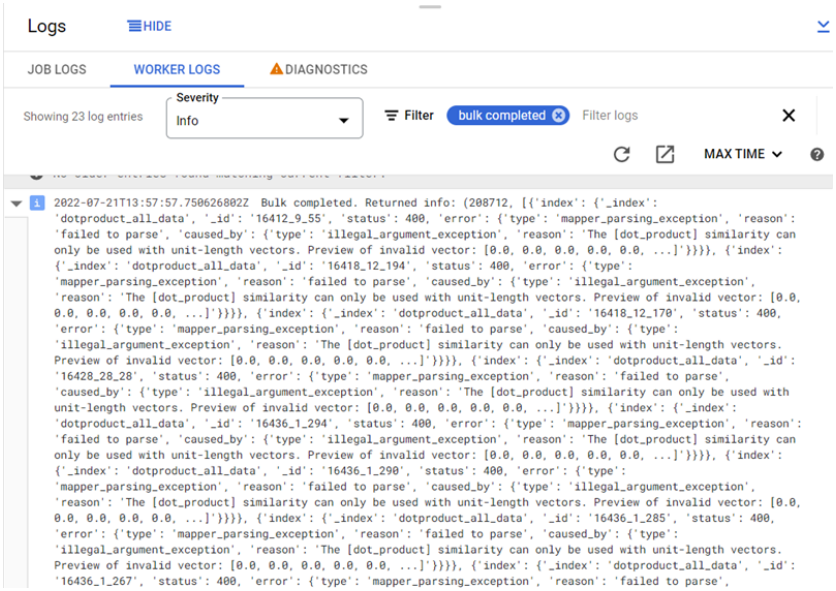


Figura 4.2: Documenti con vettore pieno di zeri

Il primo caso è un limite dell'infrastruttura su cui stiamo lavorando: nonostante sia stato ridotto il numero di workers da 5 a 3, il numero di *bulk insert* (cioè l'inserimento di un numero di documenti per mezzo di una singola chiamata a Elasticsearch) fallite rimane consistente. Il secondo caso, invece, è dovuto al comportamento dell'embedder utilizzato: ha un primo layer che recepisce solo parole che sono state utilizzate durante il training. Di fatto, delle parole che costituiscono una frase, vengono passate ai livelli interni solo quelle "conosciute" dal modello. Se questo layer non passa nulla, in output si ha un vettore pieno di zeri. La pipeline implementata nel corso di questo lavoro non controlla i vettori dei documenti da inserire; una possibile soluzione potrebbe essere quella di implementare un blocco della pipeline che controlli i vettori prima di effettuare una bulk insert, andando a scartare quelli con tutti zeri.

4.1.2 Indice similarità coseno

Provando ad eseguire la pipeline per l'inserimento di tutti i documenti nell'indice con similarità coseno con le stesse impostazioni usate in precedenza, sono stati riscontrati dei problemi: la pipeline infatti sembrava bloccarsi dopo aver inserito poco più di mezzo milione di documenti. Controllando il grafico dell'utilizzo della RAM nelle macchine dei worker, si è notato che tendeva a saturarsi, come mostrato in Figura 4.3.

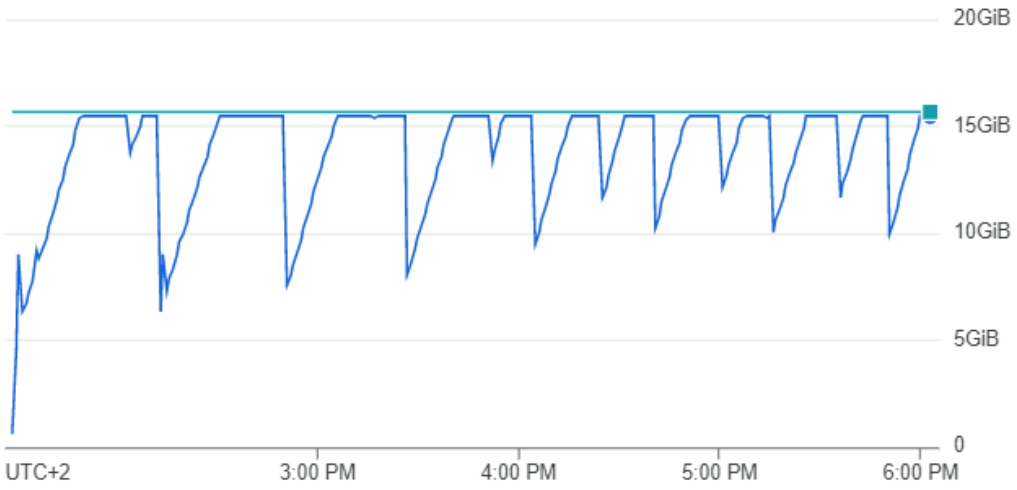


Figura 4.3: Utilizzo della RAM con la macchina n2-highmem-2

È stato quindi fatto un secondo tentativo utilizzando una macchina con il doppio della RAM (*n2-highmem-4*, con 32GB di RAM), ottenendo dei

miglioramenti, ma anche in questo caso la pipeline non è riuscita a completare e quindi è stata interrotta dopo essere stata in esecuzione per più di 12 ore. In Figura 4.4 è mostrato l'andamento della RAM con la nuova tipologia di macchina.

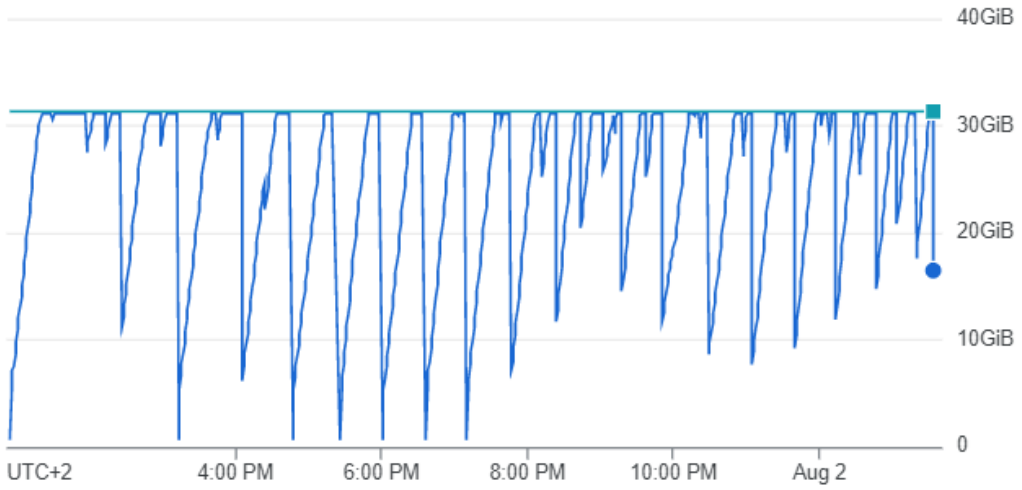


Figura 4.4: Utilizzo della RAM con la macchina n2-highmem-4

Alla fine si è deciso di evitare di utilizzare macchine con ancora più RAM e quindi i test effettuati in seguito sull'indice con similarità coseno sono stati effettuati con un numero minore di documenti. La pipeline, prima di essere stata interrotta, ha dato i seguenti risultati:

- Tempo di esecuzione della pipeline: 12 ore e 46 minuti
- Numero di documenti inseriti: 3213297
- Dimensione dei dati inseriti: circa 38GB (solo shards primari)
- Numero di bulk insert fallite: 3

4.2 Grafici di tempi di ricerca, recall ed errore relativo

I test effettuati di seguito per il calcolo dei tempi di ricerca, recall ed errore relativo sullo score, sono gli stessi effettuati nel Paragrafo 2.4, con l'unica differenza che questa volta il cluster Elasticsearch è in esecuzione

sull'infrastruttura cloud e la mole di dati su cui vengono effettuate le ricerche è decisamente maggiore.

4.2.1 Indice prodotto scalare

Di seguito sono riportati i risultati ottenuti effettuando le ricerche sull'indice con il prodotto scalare, con 6123399 documenti.

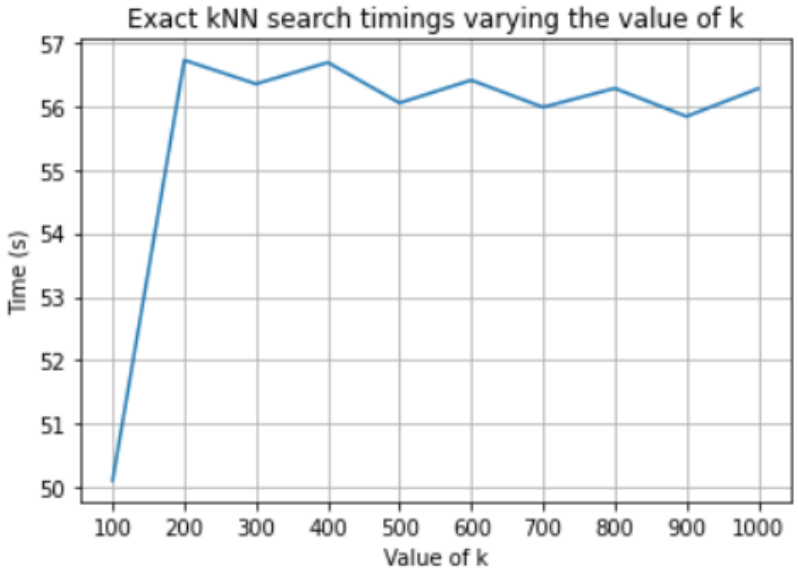


Figura 4.5: Tempi di ricerca esatta al variare di k (indice dotproduct, esecuzione in cloud)

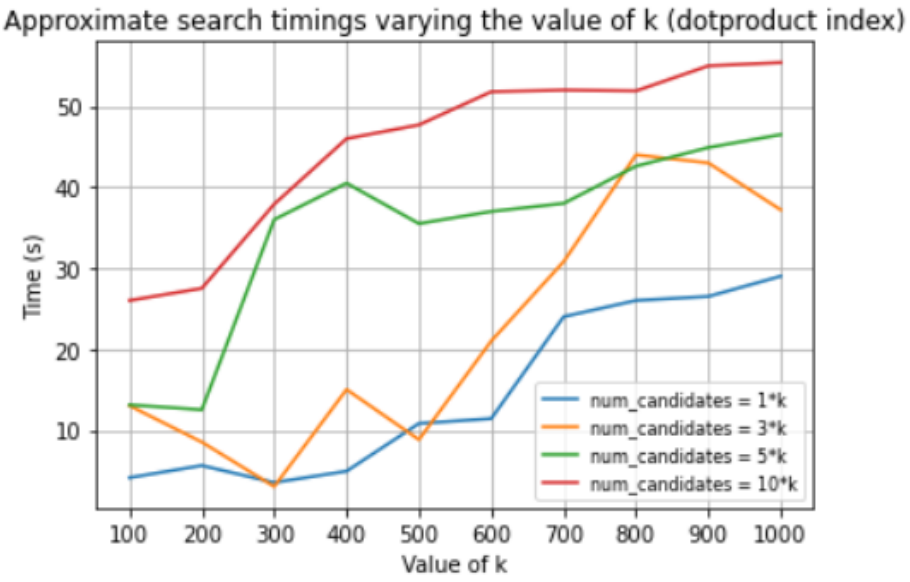


Figura 4.6: Tempi di ricerca approssimata al variare di k (indice dotproduct, esecuzione in cloud)

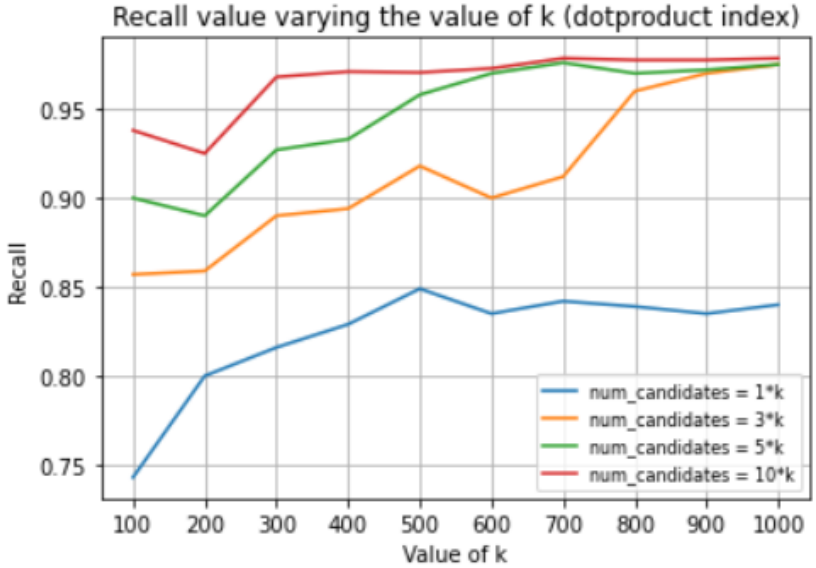


Figura 4.7: Valore della recall al variare di k (indice dotproduct, esecuzione in cloud)

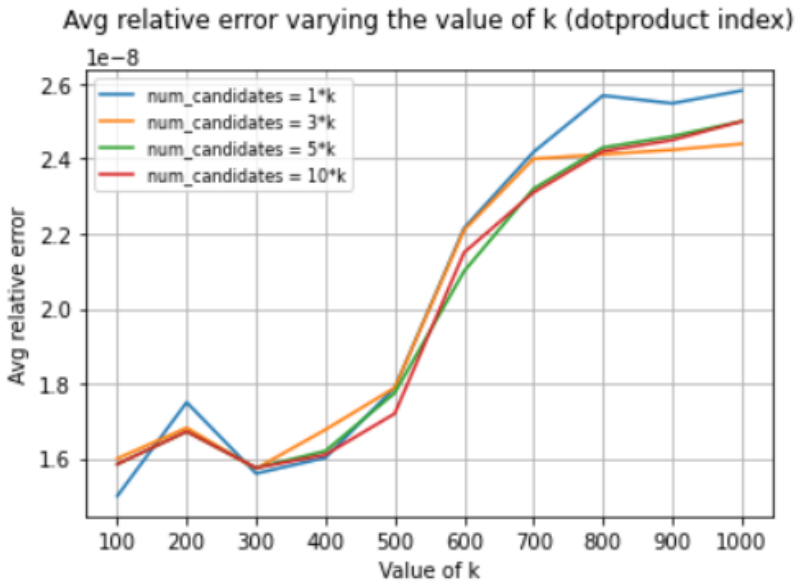


Figura 4.8: Errore relativo medio dello score al variare di k(indice dotproduct, esecuzione in cloud)

4.2.2 Indice similarità coseno

Di seguito sono riportati i risultati ottenuti effettuando le ricerche sull'indice con similarità coseno, con 3213297 documenti.

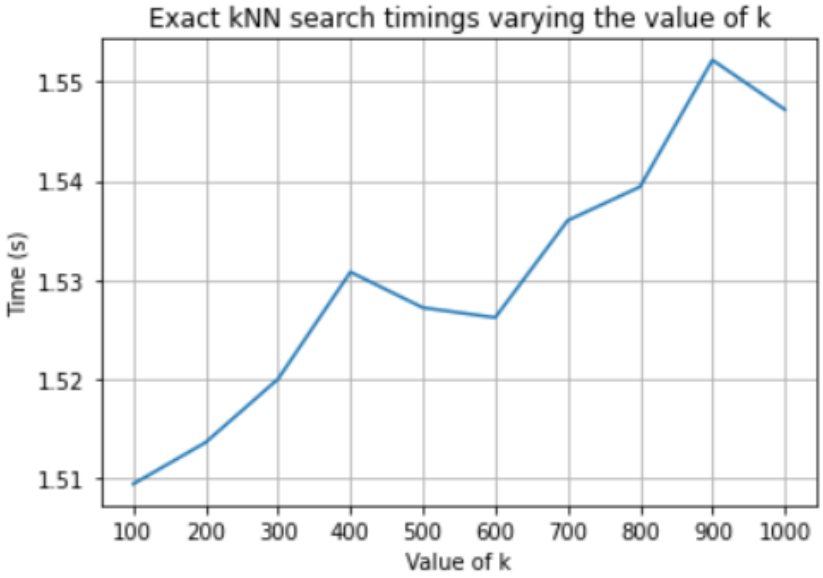


Figura 4.9: Tempi di ricerca esatta al variare di k (indice cosine, esecuzione in cloud)

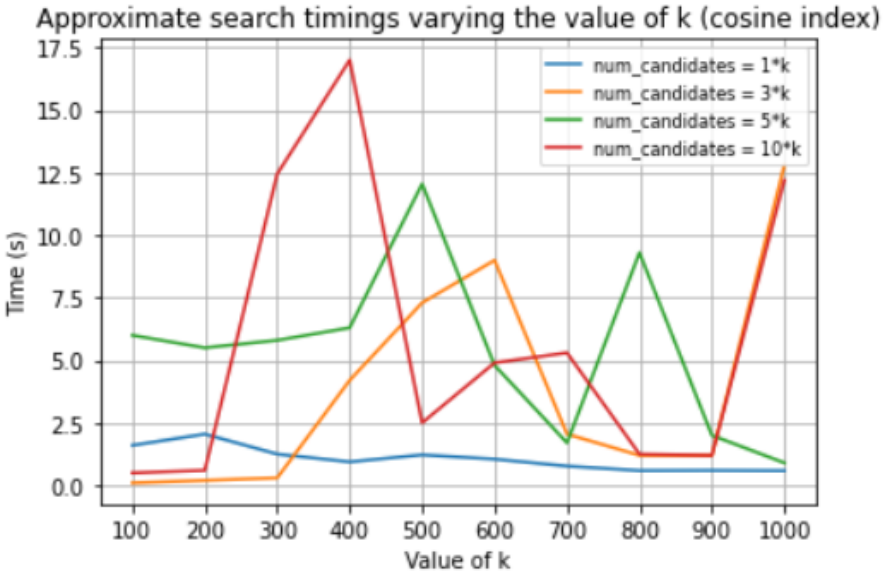


Figura 4.10: Tempi di ricerca approssimata al variare di k(indice cosine, esecuzione in cloud)

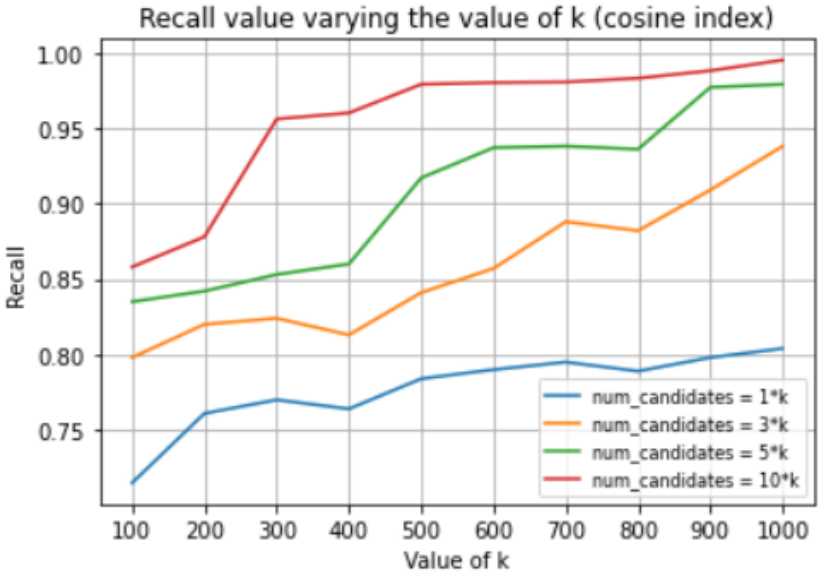


Figura 4.11: Valore della recall al variare di k(indice cosine, esecuzione in cloud)

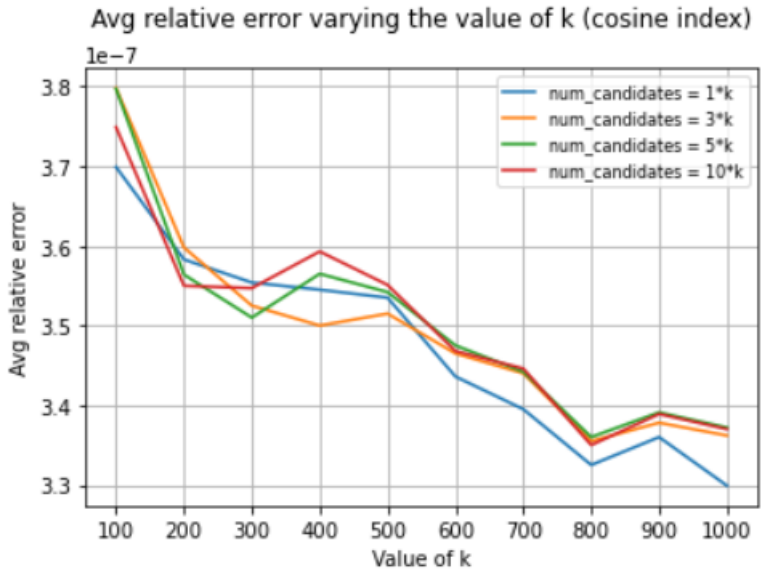


Figura 4.12: Errore relativo medio dello score al variare di k(indice cosine, esecuzione in cloud)

4.3 Considerazioni sui risultati

Per quanto riguarda l'inserimento dei dati nelle due diverse tipologie di indice, dai test effettuati si è fatta decisamente più fatica nel caso di indice con similarità coseno: infatti, nonostante siano state usate macchine con il doppio della RAM (nel caso di indice coseno 32GB, mentre nel caso di indice con prodotto scalare 16GB), la pipeline di riempimento non è riuscita a completare, riuscendo ad inserire solo 3213297 di documenti contro i 6123399 inseriti dalla pipeline con l'indice con prodotto scalare. Rimane da indagare perchè, nel caso di indice coseno, ci sia una elevata richiesta di RAM da parte dei client che inseriscono dati.
Analizzando invece i risultati ottenuti effettuando le ricerche, possiamo notare come le ricerche approssimate effettuate sull'indice con prodotto scalare hanno introdotto dei notevoli miglioramenti in termini di tempi di ricerca, con tutti i valori di num_candidates provati. Questi miglioramenti sono meno apprezzabili nell'indice con similarità coseno, probabilmente per due principali motivi: i tempi della ricerca esatta in questo caso erano già molto bassi (ricordiamo che il numero di documenti caricati in questo indice era minore) e probabilmente in questo caso non ci sono abbastanza dati da far convenire l'utilizzo della ricerca approssimata. La ricerca approssimata, infatti, permette di guardare meno dati, ma introduce un overhead dovuto al fatto che viene creata una struttura interna di ricerca.
Confrontando invece i tempi di ricerca dei due indici, tenendo sempre in considerazione la differenza di documenti, la ricerca sembra avere tempi di risposta minori nell'indice con similarità coseno, in tutti i casi.
Per quanto riguarda la precisione delle ricerche, invece, nel caso di indice con prodotto scalare sembrano esserci dei risultati leggermente migliori.
L'errore relativo è trascurabile in entrambi i casi.

Capitolo 5

Benchmark avanzati con Rally

In questo capitolo viene fatta prima un'introduzione a Rally, il tool di benchmarking rilasciato ufficialmente da Elastic. Successivamente vengono descritti tutti i passaggi eseguiti per effettuare alcune semplici race sul cluster Elasticsearch locale con gli indici precedentemente creati e sono riportati i risultati ottenuti.

5.1 Introduzione a Rally

Annunciato per la prima volta nel 2016 e rilasciato nel luglio del 2018, Rally è un tool open source per il benchmarking di Elasticsearch, sviluppato da Elastic e utilizzato inizialmente dal team di sviluppo per i loro test interni[27]. Può essere utilizzato per effettuare le seguenti operazioni:

- setup e teardown di un cluster Elasticsearch per il benchmarking;
- gestione delle specifiche e dei dati di benchmark anche tra diverse versioni di Elasticsearch;
- esecuzione di benchmark e salvataggio dei risultati;
- trovare problemi di prestazioni utilizzando i cosiddetti dispositivi di telemetria[28];
- confrontare i risultati delle prestazioni

Come mostrato in Figura 5.1, i concetti alla base di Rally, i cui nomi sono tutti ispirati al mondo delle corse, sono:

- *Race*: l'istanza del benchmark;

- *Track*: definisce tutti i parametri sui quali effettuare la race, come ad esempio l'indice Elasticsearch, i dati, le *challenges* (le operazioni da effettuare) e l'ordine di esecuzione. È anche possibile condividere le track utilizzando un *track repository*.
- *Car*: configurazione del cluster Elasticsearch su cui eseguire i test. Può essere creato da Rally prima dell'esecuzione di una race o si può indicare un cluster già in esecuzione;
- *Pipeline*: definisce tutte le operazioni da effettuare prima e dopo l'esecuzione di una race (ad esempio è possibile istanziare un cluster Elasticsearch su cui eseguire i benchmark);
- *Metric store*: alla fine di ogni race, Rally salva tutti i risultati in un metric store. Possono essere salvati in-memory o su un cluster Elasticsearch.

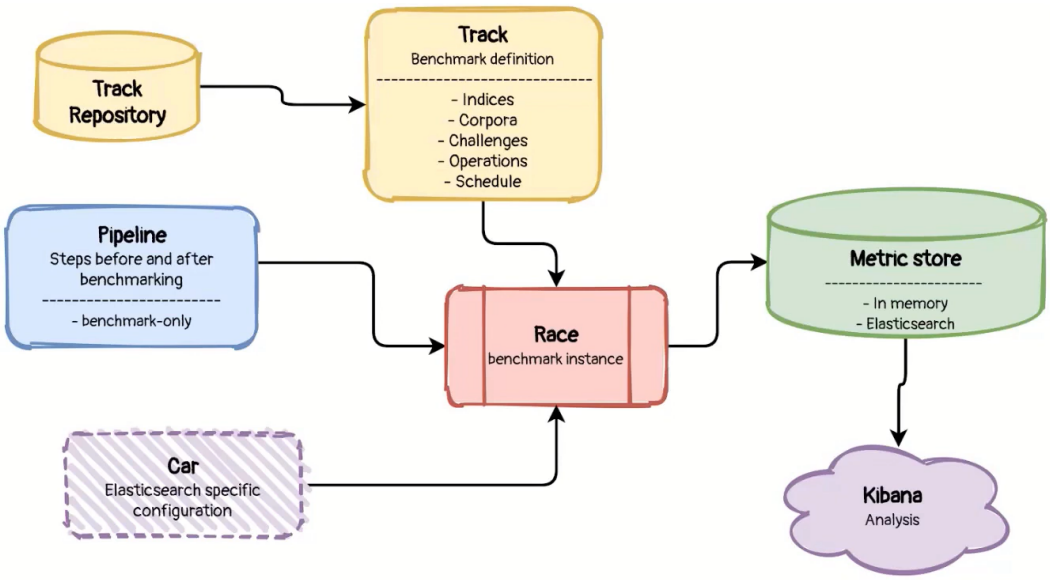


Figura 5.1: Rally overview[29]

5.2 Race eseguite e risultati

Con Rally è possibile creare dei cluster Elasticsearch su cui eseguire le race, ma nei test eseguiti di seguito, viene utilizzato un cluster Elasticsearch già in esecuzione in locale.

Prima di lanciare una race, bisogna definire una track. Rally ne mette alcune predefinite a disposizione, consultabili con il comando *esrally list tracks*, ma è possibile anche crearne una partendo dai dati memorizzati in un indice[30]. Per i test effettuati di seguito, sono state create delle track partendo dagli indici creati in precedenza nel Paragrafo 2.4. In particolare, sono state create due track diverse per le due tipologie di indice:

```
esrally create-track --track=<TRACK-NAME>
--target-hosts=https://localhost:9200
--client-options="use_ssl:false,verify_certs:false,
basic_auth_user:<USERNAME>,basic_auth_password:<PASSWORD>"
--indices=<INDEX-NAME> --output-path=<TRACK-PATH>
```

Una volta eseguito il comando, il *track generator* crea una cartella nella directory specificata contenente una lista di file:

- *track.json*: contiene le informazioni della track;
- *cosine-index.json* e *dotproduct-index.json*: contengono i mappings e i settings degli indici estratti;
- **-documents.json*: contiene tutti i documenti estratti dall'indice. Vengono generati anche dei file con il suffisso $-1k$ che contengono una versione ridotta dei dati estratti, utilizzabili per fare delle prove preliminari.

Utilizzando il comando:

```
esrally info --track-path=<TRACK-PATH>
```

è possibile visualizzare tutte le informazioni relative ad una track. In Figura 5.2 sono mostrati i dettagli della track precedentemente generata.

```
Showing details for track [cosine-track]:

* Description: Tracker-generated track for cosine-track
* Documents: 12,000
* Compressed Size: 46.0 MB
* Uncompressed Size: 116.6 MB


Schedule:
----------

1. delete-index
2. create-index
3. cluster-health
4. bulk (8 clients)
```

Figura 5.2: Esempio di track generata automaticamente

Tutte le informazioni mostrate sono presenti nel file *track.json*, che è possibile modificare per personalizzare la lista di *challenges*. Nella track generata automaticamente possiamo vedere che ci sono 4 challenges abbastanza autoesplicative:

- *delete-index*: se presente, viene cancellato l'indice;
- *create-index*: viene ricreato l'indice;
- *cluster-health*: prima di procedere all'inserimento dei dati, si controlla lo stato del cluster;
- *bulk*: viene riempito l'indice. Come mostrato in Figura 5.2, di default sono impostati 8 clients per la bulk; nei test che seguono è stato modificato questo parametro a 1 solo client, in quanto utilizzando più di 1 client si sono riscontrate perdite nell'inserimento (con 2 client è stata registrata una perdita del 50% delle bulk).

Successivamente è stata modificata la lista di challenges presente nel file *track.json*, andando ad aggiungere una challenge per la ricerca knn. Per fare ciò, è stato aggiunto all'interno del vettore contenente tutte le challenges, il seguente elemento:

```
{
  "operation": {
    "name": "knn-search",
    "operation-type": "search",
    "body": {
      "query": {
        "script_score": {
          "script": {
            "source":
                "cosineSimilarity(params.queryVector, 'embedding') + 1.0",
            "params": {
              "queryVector":
              [512 DIMS ARRAY]
            }
          },
          "query": {
            "match_all": {}
          }
        }
      },
      "size": 10
    }
  },
  "clients": 1,
  "warmup-iterations": 100,
  "iterations": 100,
  "target-throughput": 100
}
```

Per motivi di leggibilità, non è riportato il vettore utilizzato per la ricerca. È stato utilizzato un vettore estratto casualmente dall'indice. Una volta modificato il file *track.json*, andando ad eseguire il comando per vedere i dettagli della track, possiamo vedere come la challenge è stata correttamente aggiunta.

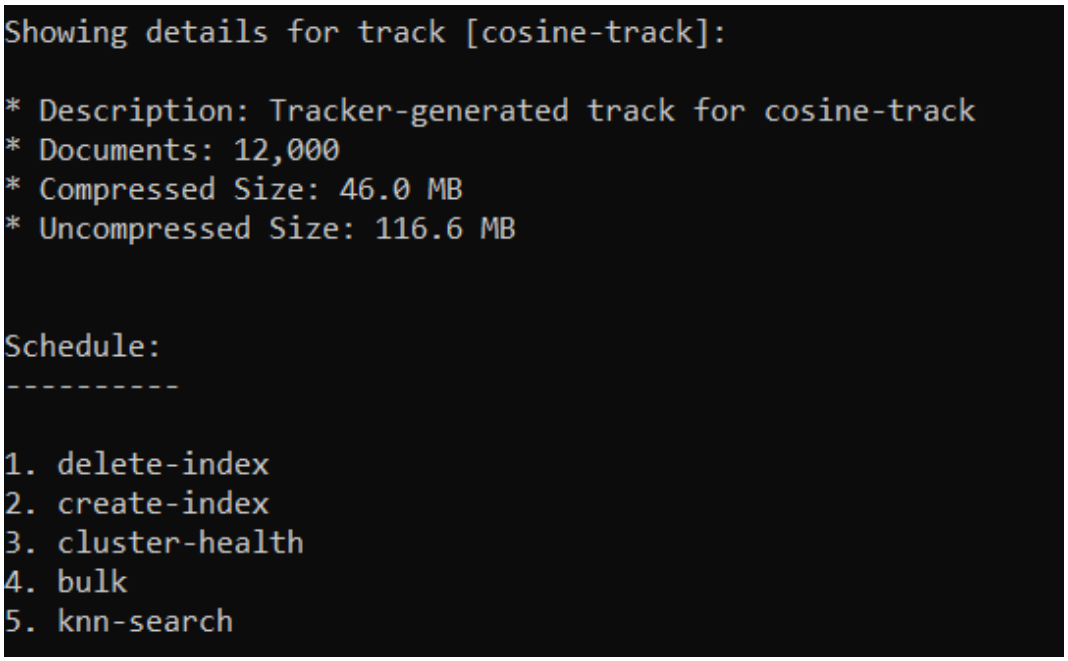


Figura 5.3: Track con la challenge knn-search

Le stesse operazioni sono state effettuate anche per l'indice con prodotto scalare. Con la challenge così definita, le statistiche collezionate da Rally si basano su quelle ottenute effettuato 100 iterazioni, dopo aver effettuato 100 *warmup-iterations*.
Una volta definite le track, è possibile eseguire le race, con il comando:

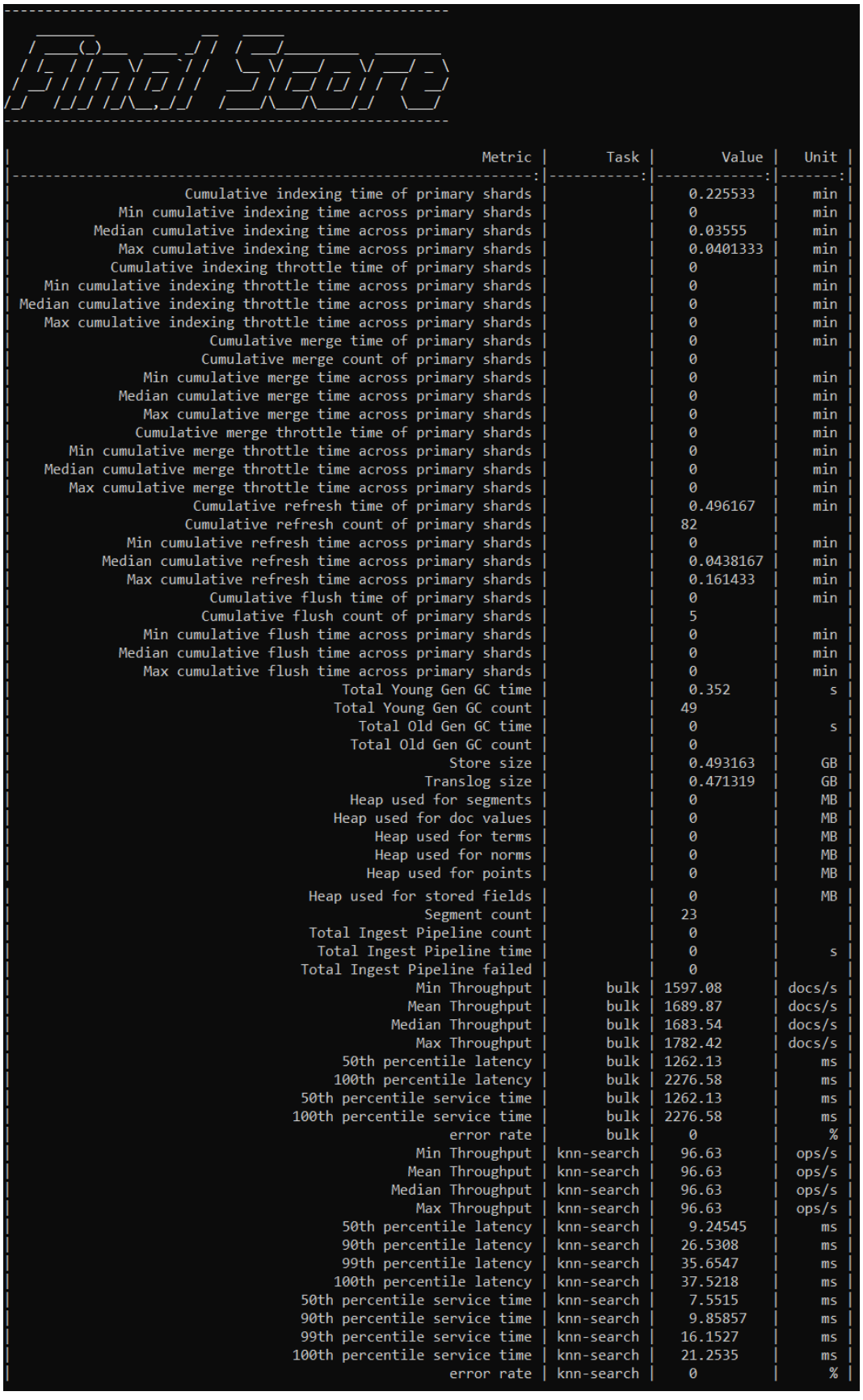
```
esrally race --track-path=<TRACK-PATH>
--target-hosts=https://localhost:9200 --pipeline=benchmark-only
--client-options="use_ssl:false,verify_certs:false,
basic_auth_user:<USERNAME>,basic_auth_password:<PASSWORD>"
```

Di seguito i risultati ottenuti dall'esecuzione della race nei due indici.

```
------------------------------------------------------
    _______             __   ____
   / ____(_)___  ____ _/ /  / __/_______  ________
  / /_  / / __ \/ __ `/ /  / /_/ ___/ _ \/ ___/ _ \
 / __/ / / / / / /_/ / /  / __/ /__/ /_/ / /  /  __/
/_/   /_/_/ /_/\__,_/_/  /_/  \___/\___/_/   \___/
------------------------------------------------------
```

Metric	Task	Value	Unit
Cumulative indexing time of primary shards		0.213033	min
Min cumulative indexing time across primary shards		0	min
Median cumulative indexing time across primary shards		0.0324167	min
Max cumulative indexing time across primary shards		0.03795	min
Cumulative indexing throttle time of primary shards		0	min
Min cumulative indexing throttle time across primary shards		0	min
Median cumulative indexing throttle time across primary shards		0	min
Max cumulative indexing throttle time across primary shards		0	min
Cumulative merge time of primary shards		0	min
Cumulative merge count of primary shards		0	
Min cumulative merge time across primary shards		0	min
Median cumulative merge time across primary shards		0	min
Max cumulative merge time across primary shards		0	min
Cumulative merge throttle time of primary shards		0	min
Min cumulative merge throttle time across primary shards		0	min
Median cumulative merge throttle time across primary shards		0	min
Max cumulative merge throttle time across primary shards		0	min
Cumulative refresh time of primary shards		0.674667	min
Cumulative refresh count of primary shards		86	
Min cumulative refresh time across primary shards		0	min
Median cumulative refresh time across primary shards		0.06565	min
Max cumulative refresh time across primary shards		0.167483	min
Cumulative flush time of primary shards		0	min
Cumulative flush count of primary shards		5	
Min cumulative flush time across primary shards		0	min
Median cumulative flush time across primary shards		0	min
Max cumulative flush time across primary shards		0	min
Total Young Gen GC time		0.365	s
Total Young Gen GC count		52	
Total Old Gen GC time		0	s
Total Old Gen GC count		0	
Store size		0.486182	GB
Translog size		0.471319	GB
Heap used for segments		0	MB
Heap used for doc values		0	MB
Heap used for terms		0	MB
Heap used for norms		0	MB
Heap used for points		0	MB
Heap used for stored fields		0	MB
Segment count		27	
Total Ingest Pipeline count		0	
Total Ingest Pipeline time		0	s
Total Ingest Pipeline failed		0	
Min Throughput	bulk	1659.75	docs/s
Mean Throughput	bulk	1711.03	docs/s
Median Throughput	bulk	1701.4	docs/s
Max Throughput	bulk	1777.72	docs/s
50th percentile latency	bulk	1229.29	ms
100th percentile latency	bulk	1793.56	ms
50th percentile service time	bulk	1229.29	ms
100th percentile service time	bulk	1793.56	ms
error rate	bulk	0	%
Min Throughput	knn-search	87.83	ops/s
Mean Throughput	knn-search	87.83	ops/s
Median Throughput	knn-search	87.83	ops/s
Max Throughput	knn-search	87.83	ops/s
50th percentile latency	knn-search	166.494	ms
90th percentile latency	knn-search	177.606	ms
99th percentile latency	knn-search	180.841	ms
100th percentile latency	knn-search	181.9	ms
50th percentile service time	knn-search	8.3293	ms
90th percentile service time	knn-search	9.52042	ms
99th percentile service time	knn-search	10.7558	ms
100th percentile service time	knn-search	10.9717	ms
error rate	knn-search	0	%

Figura 5.4: Risultati race su indice dotproduct



```
------------------------------------------------------
    ______                 __  ____
   / ____(_)___  ____ _/ /  / __/_____  ______
  / /_  / / __ \/ __ `/ /  \__ \/ ___/ __ \/ ___/ _ \
 / __/ / / / / / /_/ / /  ___/ / /__/ /_/ / /  /  __/
/_/   /_/_/ /_/\__,_/_/  /____/\___/\____/_/   \___/
------------------------------------------------------

|                                                         Metric |       Task |        Value |   Unit |
|---------------------------------------------------------------:|-----------:|-------------:|-------:|
|                     Cumulative indexing time of primary shards |            |     0.225533 |    min |
|             Min cumulative indexing time across primary shards |            |            0 |    min |
|          Median cumulative indexing time across primary shards |            |      0.03555 |    min |
|             Max cumulative indexing time across primary shards |            |    0.0401333 |    min |
|            Cumulative indexing throttle time of primary shards |            |            0 |    min |
|    Min cumulative indexing throttle time across primary shards |            |            0 |    min |
| Median cumulative indexing throttle time across primary shards |            |            0 |    min |
|    Max cumulative indexing throttle time across primary shards |            |            0 |    min |
|                        Cumulative merge time of primary shards |            |            0 |    min |
|                      Cumulative merge count of primary shards |            |            0 |        |
|              Min cumulative merge time across primary shards |            |            0 |    min |
|           Median cumulative merge time across primary shards |            |            0 |    min |
|              Max cumulative merge time across primary shards |            |            0 |    min |
|             Cumulative merge throttle time of primary shards |            |            0 |    min |
|      Min cumulative merge throttle time across primary shards |            |            0 |    min |
|   Median cumulative merge throttle time across primary shards |            |            0 |    min |
|      Max cumulative merge throttle time across primary shards |            |            0 |    min |
|                    Cumulative refresh time of primary shards |            |     0.496167 |    min |
|                  Cumulative refresh count of primary shards |            |           82 |        |
|             Min cumulative refresh time across primary shards |            |            0 |    min |
|          Median cumulative refresh time across primary shards |            |    0.0438167 |    min |
|             Max cumulative refresh time across primary shards |            |     0.161433 |    min |
|                      Cumulative flush time of primary shards |            |            0 |    min |
|                    Cumulative flush count of primary shards |            |            5 |        |
|               Min cumulative flush time across primary shards |            |            0 |    min |
|            Median cumulative flush time across primary shards |            |            0 |    min |
|               Max cumulative flush time across primary shards |            |            0 |    min |
|                                     Total Young Gen GC time |            |        0.352 |      s |
|                                    Total Young Gen GC count |            |           49 |        |
|                                       Total Old Gen GC time |            |            0 |      s |
|                                      Total Old Gen GC count |            |            0 |        |
|                                                 Store size |            |     0.493163 |     GB |
|                                              Translog size |            |     0.471319 |     GB |
|                                   Heap used for segments |            |            0 |     MB |
|                                 Heap used for doc values |            |            0 |     MB |
|                                      Heap used for terms |            |            0 |     MB |
|                                      Heap used for norms |            |            0 |     MB |
|                                     Heap used for points |            |            0 |     MB |
|                                  Heap used for stored fields |            |            0 |     MB |
|                                                Segment count |            |           23 |        |
|                                 Total Ingest Pipeline count |            |            0 |        |
|                                  Total Ingest Pipeline time |            |            0 |      s |
|                                Total Ingest Pipeline failed |            |            0 |        |
|                                                Min Throughput |       bulk |      1597.08 | docs/s |
|                                               Mean Throughput |       bulk |      1689.87 | docs/s |
|                                             Median Throughput |       bulk |      1683.54 | docs/s |
|                                                Max Throughput |       bulk |      1782.42 | docs/s |
|                                      50th percentile latency |       bulk |      1262.13 |     ms |
|                                     100th percentile latency |       bulk |      2276.58 |     ms |
|                                 50th percentile service time |       bulk |      1262.13 |     ms |
|                                100th percentile service time |       bulk |      2276.58 |     ms |
|                                                    error rate |       bulk |            0 |      % |
|                                                Min Throughput | knn-search |        96.63 |  ops/s |
|                                               Mean Throughput | knn-search |        96.63 |  ops/s |
|                                             Median Throughput | knn-search |        96.63 |  ops/s |
|                                                Max Throughput | knn-search |        96.63 |  ops/s |
|                                      50th percentile latency | knn-search |      9.24545 |     ms |
|                                      90th percentile latency | knn-search |      26.5308 |     ms |
|                                      99th percentile latency | knn-search |      35.6547 |     ms |
|                                     100th percentile latency | knn-search |      37.5218 |     ms |
|                                 50th percentile service time | knn-search |       7.5515 |     ms |
|                                 90th percentile service time | knn-search |       9.85857 |     ms |
|                                 99th percentile service time | knn-search |      16.1527 |     ms |
|                                100th percentile service time | knn-search |      21.2535 |     ms |
|                                                    error rate | knn-search |            0 |      % |
```

Figura 5.5: Risultati race su indice cosine

5.3 Considerazioni sui risultati

Come possiamo vedere dai risultati ottenuti, Rally esegue dei benchmark molto avanzati. I test eseguiti in locale sono stati fatti su una mole di dati probabilmente troppo ridotta per notare grosse differenze di performance tra i due tipi di indice. Andando ad analizzare i risultati, infatti, possiamo notare come i tempi di indicizzazione di tutti i documenti nei due indici sia praticamente uguale (indicato dal valore di *cumulative indexing time of primary shards*). Anche a livello di memoria occupata siamo più o meno sugli stessi livelli: 493 MB per l'indice coseno, 486 MB per l'indice con prodotto scalare. Le operazioni di bulk hanno inserito una media di 1689 documenti al secondo nell'indice coseno, mentre nell'indice con prodotto scalare 1711 documenti al secondo. La ricerca kNN sembra essere leggermente più veloce nell'indice coseno: una media di 96 operazioni al secondo contro le 87 dell'indice con prodotto scalare.

Conclusioni e sviluppi futuri

In questo elaborato è stato richiamato il concetto di ricerca per similarità semantica, ponendo particolare enfasi al caso della ricerca full-text. In un contesto in cui la ricerca di informazioni rilevanti diventa sempre più complessa a causa dell'enorme mole di dati a disposizione e delle diverse tipologie di dato, gli approcci di ricerca tradizionale sono diventati sempre meno efficaci.

Si è quindi condotta un'analisi prestazionale delle ricerche Nearest Neighbor, utilizzando come sistema di data storage Elasticsearch in esecuzione su un'infrastruttura cloud realizzata sulla Google Cloud Platform. Per l'implementazione della pipeline per l'inserimento dei dati è stato utilizzato il modello Apache Beam, con Dataflow come runner. Sono state valutate le prestazioni delle ricerche kNN su un dataset di documenti di testo, in cui ogni frase è stata mappata in un vettore di dimensione 512, lavorando con milioni di frasi. Sono stati analizzati i tempi di risposta sia della ricerca kNN esatta che di quella approssimata per capire quanto e quando fosse conveniente utilizzare l'una o l'altra, e sono state messi a confronto anche i valori della recall, per capire quanta precisione si può perdere utilizzando la ricerca approssimata. Questi test sono stati effettuati su due diversi indici Elasticsearch, ognuno dei quali utilizzava una metrica di distanza diversa: uno il prodotto scalare e l'altro la similarità coseno.

Infine sono stati effettuati alcuni test di benchmarking utilizzando Rally, il tool rilasciato ufficialmente da Elastic, con il quale si possono ottenere delle informazioni decisamente più articolare sui risultati dei test. Quest'ultima parte tuttavia non è stata approfondita fino in fondo.

Tra i possibili sviluppi futuri, quindi, si possono condurre dei test più approfonditi con Rally, che è in una fase di continuo supporto, così come Elasticsearch, e quindi rilascia continuamente aggiornamenti che potrebbero essere interessanti per ottenere ulteriori informazioni in questo senso. Un altro aspetto che può essere migliorato è l'implementazione di un blocco della pipeline che controlli i vettori prima di inoltrare la richiesta di inserimento al cluster Elasticsearch, scartando i vettori che non possono essere inseriti (come ad esempio i vettori pieni di zeri).

Ringraziamenti

Eccomi giunto alla fine di questo elaborato e di questi due anni di Università che mi hanno fatto maturare sotto tutti i punti di vista e durante i quali ho acquisito numerose conoscenze. Vorrei dedicare queste ultime righe per ringraziare tutte le persone che hanno sempre creduto in me e mi hanno sostenuto nei momenti di difficoltà.
In primis, un ringraziamento speciale al mio relatore Claudio Sartori per la sua infinita disponibilità e tempestività ad ogni mia richiesta. Grazie per avermi guidato e supportato nella fase più importante del mio percorso accademico.
Grazie al mio correlatore Riccardo Zanella per il supporto costante, le dritte indispensabili e la sua complicità nella realizzazione di ogni capitolo della mia tesi.
Ringrazio tutto il team di Machine Learning di Injenia Srl per l'ospitalità e per avermi accompagnato durante questi mesi di tirocinio; in particolar modo grazie a Simone Guardati per tutti i suoi consigli e la sua disponibilità.
Ringrazio di cuore la mia famiglia perché mi è sempre stata accanto e non mi ha mai fatto mancare il suo sostegno e il suo aiuto. Senza di loro non sarei mai diventato quello che sono.
Un caloroso grazie alla mia ragazza che con amore, pazienza e fiducia mi ha sostenuto per tutti questi anni.
Infine, grazie a tutti i miei amici che mi sono sempre stati accanto, sopportando i miei sfoghi nei momenti più difficili e festeggiando insieme a me i miei successi.

Elenco delle figure

Bibliografia

[1] Pavel Zezula, Giuseppe Amato, Vlastislav Dohnal, and Michal Batko. *Similarity Search - The Metric Space Approach*, volume 32 of *Advances in Database Systems*. Kluwer, 2006.

[2] Edgar Chávez, Gonzalo Navarro, Ricardo A. Baeza-Yates, and José L. Marroquín. Searching in metric spaces. *ACM Comput. Surv.*, 33(3):273–321, 2001.

[3] Roman Bartusiak, Lukasz Augustyniak, Tomasz Kajdanowicz, Przemyslaw Kazienko, and Maciej Piasecki. Wordnet2vec: Corpora agnostic word vectorization method. *Neurocomputing*, 326-327:141–150, 2019.

[4] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. Glove: Global vectors for word representation. In Alessandro Moschitti, Bo Pang, and Walter Daelemans, editors, *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing, EMNLP 2014, October 25-29, 2014, Doha, Qatar, A meeting of SIGDAT, a Special Interest Group of the ACL*, pages 1532–1543. ACL, 2014.

[5] Jean Jacques Denimal and Sergio Camiz. Complex principal component analysis: Theory and geometrical aspects. *J. Classif.*, 39(2):376–408, 2022.

[6] Wikipedia. Distanza di minkowski — wikipedia, l'enciclopedia libera, 2021. [Online; in data 23-agosto-2022].

[7] Mohammad Reza Abbasifard, Bijan Ghahremani, and Hassan Naderi. A survey on nearest neighbor search methods. *International Journal of Computer Applications*, 95(25), 2014.

[8] Jeff Johnson, Matthijs Douze, and Hervé Jégou. Billion-scale similarity search with GPUs. *IEEE Transactions on Big Data*, 7(3):535–547, 2019.

[9] erikbern. Annoy. `https://github.com/spotify/annoy`, 2022.

[10] Yury A. Malkov and Dmitry A. Yashunin. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *IEEE Trans. Pattern Anal. Mach. Intell.*, 42(4):824–836, 2020.

[11] C. Gormley and Z. Tong. *Elasticsearch: The Definitive Guide: A Distributed Real-Time Search and Analytics Engine*. O'Reilly Media, 2015.

[12] Elasticsearch python api reference. `https://elasticsearch-py.readthedocs.io/en/v8.2.0/api.html`.

[13] Documentazione Elasticsearch. Install elasticsearch with docker. `https://www.elastic.co/guide/en/elasticsearch/reference/current/docker.html#docker-compose-file`.

[14] Documentazione Elasticsearch. k-nearest neighbor search. `https://www.elastic.co/guide/en/elasticsearch/reference/8.2/knn-search.html`.

[15] Google cloud platform. `https://cloud.google.com/`.

[16] Hana Mallek, Faiza Ghozzi, and Faïez Gargouri. Towards extract-transform-load operations in a big data context. *Int. J. Sociotechnology Knowl. Dev.*, 12(2):77–95, 2020.

[17] Google Cloud. Colossus under the hood: a peek into google's scalable storage system. `https://cloud.google.com/blog/products/storage-data-transfer/a-peek-behind-colossus-googles-file-system`.

[18] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In Mohammed G. Khatib, Xubin He, and Michael Factor, editors, *IEEE 26th Symposium on Mass Storage Systems and Technologies, MSST 2012, Lake Tahoe, Nevada, USA, May 3-7, 2010*, pages 1–10. IEEE Computer Society, 2010.

[19] Google Cloud Platform. About machine families. `https://cloud.google.com/compute/docs/machine-types`.

[20] Apache beam programming guide. `https://beam.apache.org/documentation/programming-guide/`.

[21] Apache beam overview. `https://beam.apache.org/get-started/beam-overview/`.

[22] Wikipedia. Grafo aciclico diretto — wikipedia, l'enciclopedia libera, 2022. [Online; in data 6-settembre-2022].

[23] S. Rajeswari, K. Suthendran, K. Rajakumar, and S. Arumugam. An overview of the mapreduce model. In S. Arumugam, Jay S. Bagga, Lowell W. Beineke, and B. S. Panda, editors, *Theoretical Computer Science and Discrete Mathematics - First International Conference, ICTCSDM 2016, Krishnankoil, India, December 19-21, 2016, Revised Selected Papers*, volume 10398 of *Lecture Notes in Computer Science*, pages 312–317. Springer, 2016.

[24] Python documentation - argparse. `https://docs.python.org/3/library/argparse.html`.

[25] Elasticsearch documentation. Index template. `https://www.elastic.co/guide/en/elasticsearch/reference/8.2/index-templates.html`.

[26] Elasticsearch documentation. Tune for indexing speed. `https://www.elastic.co/guide/en/elasticsearch/reference/current/tune-for-indexing-speed.html`.

[27] logz.io. Benchmarking elasticsearch with rally. `https://logz.io/blog/rally/`.

[28] Rally documentation. Telemetry devices. `https://esrally.readthedocs.io/en/stable/telemetry.html`.

[29] ElasticCC. Creating a custom track in rally to benchmark your production cluster.

[30] Rally documentation. Define custom workloads: Tracks. `https://esrally.readthedocs.io/en/stable/adding_tracks.html`.