Ingegneria dei Dati:
Motore di Ricerca con Lucene

Mattia Micaloni, Carlo Proserpio, Alessandro Di Girolamo

Project repository: https://github.com/Xhst/de-projects/tree/main/project-2

Keywords: Ingegneria dei Dati, Apache Lucene, Motore di Ricerca, Indicizzazione, Tokenizzazione, Ricerca Booleana, Arxiv

1. Introduzione

La seguente relazione intende presentare in dettaglio la soluzione implementata per la creazione di un motore di ricerca, basato su un insieme di articoli scientifici raccolti precedentemente dal portale *arXiv*[1]. Questo motore di ricerca è stato sviluppato utilizzando la libreria *Apache Lucene*, che fornisce una vasta gamma di API pensate per supportare ogni fase della costruzione di un *Search Engine* personalizzato, configurabile in base ai requisiti specifici del progetto.
Apache Lucene offre all'utente grande flessibilità nella scelta del linguaggio di programmazione, consentendo l'integrazione in vari ambienti e linguaggi, tra cui *Python*, *.NET*, *C++*, *Ruby* e *Java*. In questo progetto abbiamo scelto di utilizzare *Java* come linguaggio principale per l'implementazione poiché *Apache Lucene* è stato originariamente sviluppato come libreria *Java* ed è tuttora la versione più completa, ottimizzata e collaudata rispetto ai wrapper in altri linguaggi.
L'obiettivo principale del progetto è stato la creazione di un motore di ricerca efficiente per la ricerca su vari campi di oltre 9000 articoli. A questo scopo, è stata posta particolare attenzione alle fasi di progettazione e implementazione delle logiche di indicizzazione e ricerca.

2. Implementazione e struttura del progetto

Per implementare il motore di ricerca sugli articoli abbiamo realizzato una semplice architettura client-server.
Il server è un'applicazione *Spring Boot* con un *Rest Controller* che riceve le query dal

[1] https://arxiv.org/

client, le quali vengono poi passate al *SearchService* che si occupa del parsing della query, che viene poi elaborata dall'*IndexSearcher* presente in Lucene. All'avvio del server, se è abilitata la re-indicizzazione nella configurazione, un apposito *Component* chiamato *Indexer* si occupa della creazione dell'indice.
Il client è una semplice applicazione web responsiva con un input per inserire la query. É presente anche una ricerca avanzata che semplifica la costruzione di query booleane, permettendo di inserire un numero arbitrario di vincoli, di manipolare il modo in cui sono visualizzati i risultati (se mostrare o meno i campi abstract, autori e keywords) e per specificare il numero massimo di risultati che il server può ritornare. Abbiamo aggiunto anche la possibilità di cambiare il tema del client potendo scegliere tra quello chiaro (figura 1) e quello scuro (figura 2).

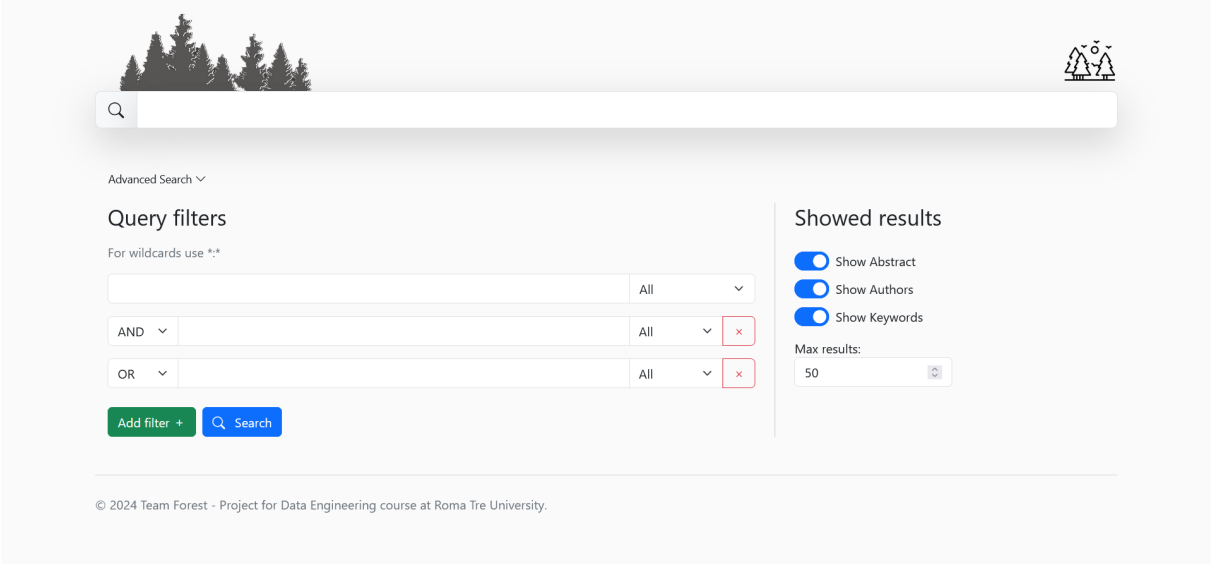

Figura 1: Tema chiaro del client

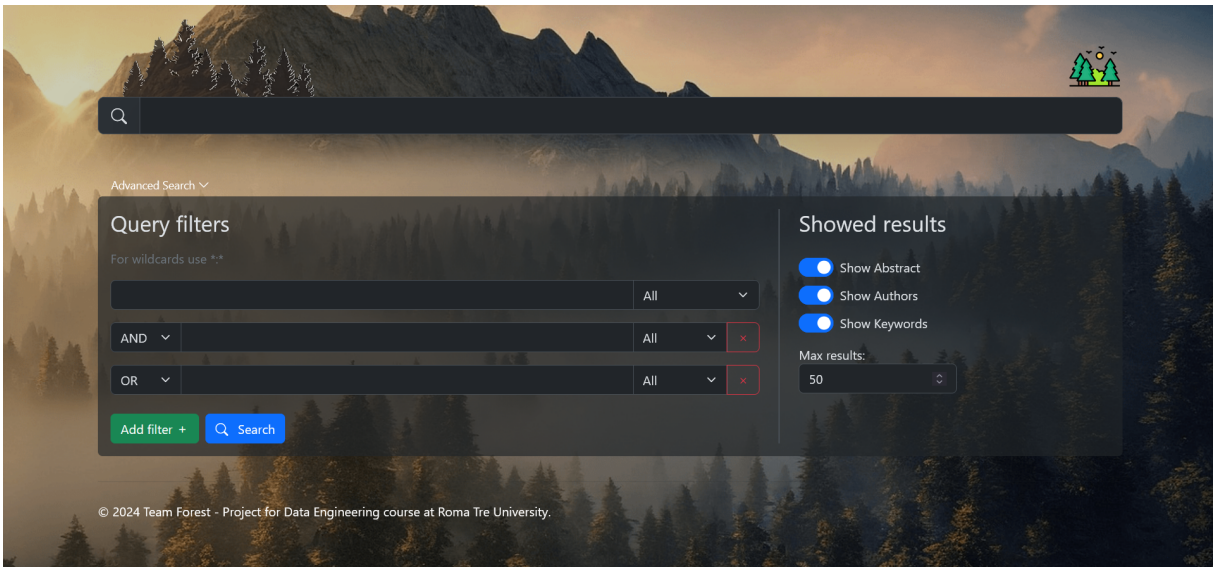

Figura 2: Tema scuro del client

2.1 Estrazione dei campi

Prima di parlare dell'indicizzazione è opportuno specificare il modo in cui sono stati estratti i vari campi dagli articoli.
I documenti sono stati analizzati uno alla volta tramite la libreria Java *Jsoup*, la quale permette di utilizzare espressioni *xPath* con cui abbiamo estratto i campi: **codice paper**, **titolo**, **autori**, **keyword**, **abstract**, **contenuto**. I campi sono stati poi passati all'*Indexer* e gestiti tramite i vari analizzatori, approfonditi nel successivo capitolo.

2.2 Analizzatori

Particolare importanza va data agli analizzatori, in quaunto utilizzati per l'indicizzazione, quindi determinanti per l'efficacia delle query. L'analizzazione è stata implementata nel file *Indexer*, mediante i metodi offerti dalla libreria *Apache Lucene*, i quali permettono di definire tokenizzatori e filtri (es.: stemmer) al fine di indicizzare i paper secondo alcune regole. Inoltre, c'è la possibilità di costruire degli analizzatori personalizzati a seconda delle esigenze del progetto. L'indicizzazione è stata effettuata su diverse sezioni dei paper e su ciascuna sono stati definiti gli opportuni analizzatori personalizzati. Più precisamente, per ogni paper sono stati indicizzati: **codice paper**, **titolo**, **autori**, **keyword**, **abstract**, **contenuto**.
Di seguito sono sono elencati e approfonditi i vari analizzatori.

2.2.1 Analizzatore per il Codice Paper

Questo analizzatore è stato progettato per indicizzare il codice univoco di ciascun paper.

```
CustomAnalyzer.Builder filenameanalyzerBuilder = CustomAnalyzer.builder()
        .withTokenizer(KeywordTokenizerFactory.class)
        .addTokenFilter(TrimFilterFactory.class);
```

KeywordTokenizerFactory tokenizza il testo come un unico token, utile in quanto il codice del paper è univoco, mentre *TrimFilterFactory* filtra i token rimuovendo gli spazi bianchi superflui alla fine e all'inizio di quest'ultimo

2.2.2 Analizzatore per il Titolo

L'analizzatore del titolo utilizza un tokenizzatore e dei filtri utili a migliorare l'indicizzazione.

```
CustomAnalyzer.Builder titleAnalyzerBuilder = CustomAnalyzer.builder()
         .withTokenizer(StandardTokenizerFactory.class)
         .addTokenFilter(LowerCaseFilterFactory.class)
         .addTokenFilter(TrimFilterFactory.class);
```

StandardTokenizerFactory tokenizza rimuovendo spazi bianchi e punteggiatura, *LowerCaseFilterFactory* converte tutto il testo in minuscolo al fine di rendere la ricerca nell'indice

più flessibile e *TrimFilterFactory* per un ulteriore controllo su eventuali spazi bianchi extra nei token.

2.2.3 Analizzatore per le Keyword

Per indicizzare le keyword è stato utilizzato un analizzatore analogo a quello del titolo.

2.2.4 Analizzatore per gli Autori

L'analizzatore per gli autori è ottimizzato per indicizzare nomi propri, facilitando la ricerca per autore. Anche in questo caso, per motivi analoghi, è stata utilizzata la stessa configurazione dell'analizzatore del titolo.

```
CustomAnalyzer.Builder titleAnalyzerBuilder = CustomAnalyzer.builder()
        .withTokenizer(StandardTokenizerFactory.class)
        .addTokenFilter(LowerCaseFilterFactory.class)
        .addTokenFilter(TrimFilterFactory.class);
```

2.2.5 Analizzatore per l'Abstract

L'analizzatore dell'abstract applica una serie di filtri per ridurre il testo da indicizzare, concentrandosi sui concetti principali espressi.

```
CustomAnalyzer.Builder titleAnalyzerBuilder = CustomAnalyzer.builder()
        .withTokenizer(StandardTokenizerFactory.class)
        .addTokenFilter(LowerCaseFilterFactory.class)
        .addTokenFilter(TrimFilterFactory.class);
        .addTokenFilter(PorterStemFilterFactory.class)
        .addTokenFilter(StopFilterFactory.class)
        .addTokenFilter(RemoveDuplicatesTokenFilterFactory.class)
        .addTokenFilter(ASCIIFoldingFilterFactory.class)
```

StandardTokenizerFactory, I filtri *LowerCaseFilterFactory*, *TrimFilterFactory* sono stati discussi precedentemente, per quanto riguarda gli altri abbiamo: *PorterStemFilterFactory* che applica lo stemmer di Porter ai token, un algoritmo che riduce le parole alla loro radice; *StopFilterFactory* che rimuove le stop-word (es.: "the", "and", "in"), le quali non sarebbero rilevanti ai fini della ricerca; *RemoveDuplicatesTokenFilterFactory* che rimuove le parole duplicate consecutive al fine di prevedere e gestire, ad esempio, eventuali errori commessi dall'autore durante la scrittura e *ASCIIFoldingFilterFactory* utilizzato per rimuovere gli accenti dai token.

2.2.6 Analizzatore per il Contenuto

Progettato per l'intero contenuto del paper, questo analizzatore utilizza la stessa configurazione dell'analizzatore dell'abstract in quanto deve indicizzare su un testo con la forma molto simile. In effetti, l'abstract è un sunto del contenuto del paper quindi l'utilizzo dello stesso analizzatore è una scelta valida.

2.3 Query

Le query vengono passate dal client al server, come descritto nella sezione 2, in un formato interpretabile dal parser di Lucene, mostrato nella figura 3.

```
<campo>:<termine o "frase"> (AND, NOT, OR) (<campo>:<termine o "frase">)
                                (AND, OR, NOT) ...
```

Figura 3: Formato delle query assemblate sul client. *Campo* e *termine* possono essere sostituiti con una wildcard (*:*).

Una volta ricevuta la query sul server, basterà usare le API di Lucene per interrogare il sistema, e i risultati verranno inviati al client, che si occuperà di mostrarli a schermo. Infine, segue una serie di query che abbiamo ritenuto interessanti ai fini di valutare il sistema. I risultati ottenuti verranno discussi nella sezione 3.3.

1. Query generale: `record`
2. Query generale (max 10000+ risultati): `record`
3. Query su titolo: `title:data`
4. Query contradditoria: `title:data NOT title:data`
5. Phrase query: `title:"monte carlo"`
6. Phrase query su topic progetto 1 (max 10000+ risultati): `title:"synthetic data"`
7. Query su più autori: `machine AND authors:Zheng AND authors:Ming`
8. Query su keyword: `machine AND keywords:federated`
9. Query su abstract: `"federated learning" AND abstract:medicine`
10. Wildcard Query (max 10000+ risultati): `*:*`
11. Query negativa (max 10000+ risultati): `*:* NOT data`

3. Valutazione

In questa sezione sono riportate le valutazioni che abbiamo effettuato su indicizzazione e sull'esecuzione delle query, l'hardware utilizzato per effettuare le misurazioni e alcune delle accortezze che abbiamo seguito per ottimizzare i tempi.

3.1 Hardware utilizzato

CPU: Intel® Core™ i7-12650H 2.30 GHz di 12° generazione.

DISCO: SSD NVMe Intel® SSDPEKNW512GZL.

RAM: uno slot da 16GB DDR4 SODIMM.

3.2 Ottimizzazioni

Per l'indicizzazione abbiamo deciso di aggiungere nella configurazione la possibilità di scegliere la grandezza del buffer (RAM) utilizzato in scrittura, nel nostro caso l'abbiamo impostato a 2048 MB. Abbiamo salvato nell'indice i campi *filename*, *title*, *authors*, *keywords* e *abstract* poiché vengono restituiti come risposta al client e generalmente non sono di grandi dimensioni. Il contenuto non è stato salvato nell'indice poiché di grande dimensione e avrebbe quindi appesantito troppo l'indice, degradando sia i tempi di indicizzazione che di query. Il tempo di indicizzazione richiesto per tutti i documenti è stato di 189 secondi.
Per quanto riguarda l'ottimizzazione dell'esecuzione delle query utilizzato dei *Bean* per *IndexSearcher* e *IndexParser* in modo da crearli una singola volta e iniettarli come dipendenze al *SearchService*. Facendo riuso delle stesse istanze di questi oggetti ad ogni ricerca si riducono notevolmente i tempi di query.

3.3 Valutazione query

In questa sezione riportiamo i risultati dell'esecuzione delle query descritte nella sezione 2.3, analizzando per ciascuna i tempi di ricerca (calcolati sul server) e il numero di paper restituiti. Per ogni interrogazione, se non specificato diversamente, viene impostato un numero massimo di articoli restituiti pari a 50. In generale, se una query viene ripetuta, i meccanismi di caching di Lucene permettono di ottenere uno speedup medio di 5.81 volte rispetto al tempo di ricerca della prima volta che si esegue l'interrogazione. Questo però è vero soltanto se manteniamo un massimo di articoli restituiti pari a 50. Se aumentiamo il numero di articoli restituiti e la query trova effettivamente un numero elevato di articoli, allora lo speedup tende a 1.
Un riassunto dei risultati ottenuti è riportato nella tabella 1, con tempi di ricerca e numero di articoli restituiti per ogni query.

Per quanto riguarda la query 1, eseguiamo una ricerca generale su "record". In questo caso il motore di ricerca controlla tutto il documento. Otteniamo i 50 risultati richiesti in 32ms. Se invece ripetiamo la query, ma impostiamo il numero massimo di articoli restituiti a 10.000 (query 2), otteniamo 3.362 risultati in 717ms. Possiamo osservare come chiedere al motore di ricerca di restituire tutti gli articoli che trova aumenta di molto (10x) il tempo di ricerca.

La query 3 serve per testare l'indicizzazione sul titolo. Otteniamo 50 risultati in 23ms. Da questa query possiamo osservare come fare una ricerca su un campo specifico, che contiene quindi un testo più breve rispetto all'intero contenuto dell'articolo possa diminuire i tempi di ricerca.
La query 4 vuole testare i vincoli booleani. In questo caso l'interrogazione è volutamente contraddittoria e dovrebbe ritornare 0 articoli. Eseguendola, effettivamente non si trova nulla e il tempo di ricerca è meno di 1ms, indicando probabilmente il fatto che Lucene ha inferito dalla query che non c'era bisogno di fare nulla.

La query 5 è una semplice phrase query sul titolo. Otteniamo solo 4 risultati in 16ms. Controllando manualmente abbiamo verificato che tutti e 4 contengono correttamente "Monte Carlo" nel titolo. Un'interrogazione più interessante è la phrase query 6, che effettua una ricerca sul titolo "synthetic data". Nel precedente progetto[2], abbiamo estratto le tabelle da articoli su Arxiv relativi all'argomento *synthetic data*. Per trovare gli articoli, in quel caso abbiamo usato il motore di ricerca avanzato di Arxiv, eseguendo una ricerca sul titolo come nella query 6 e scegliendo infine 649 articoli da cui estrarre tabelle. Il nostro motore di ricerca ne trova 616 in 125ms, ovvero circa il 95%. Non siamo certi del motivo, ma potrebbe essere una combinazione di vari fattori. Per esempio, il modo in cui estraiamo il titolo per l'indice, oppure semplicemente il funzionamento diverso del motore di ricerca di Arxiv, che sicuramente riesce a gestire in modo più flessibile le interrogazioni. Un'ulteriore ipotesi potrebbe essere che gli articoli mancanti siano stati convertiti erroneamente dal formato PDF al formato HTML su Ar5iv/Arxiv.

Dopodiché, testiamo con la query 7 una ricerca su più autori. Cerchiamo articoli che siano stati scritti da Zheng e Ming in cui appaia la parola "machine". In questo caso otteniamo 3 risultati in 16ms. Lucene riesce a sfruttare bene l'indice sul campo per ottimizzare i tempi di ricerca. Controllando a mano i risultati ne confermiamo la correttezza. La query 8 effettua una ricerca sulla keyword "federated" insieme a una ricerca generale su "machine". Otteniamo, in 16ms, 50 risultati. Anche qui, oltre a verificare la correttezza dei risultati, notiamo come la ricerca su un campo con un contenuto breve riesce a mitigare i tempi di ricerca. La query 9 invece cerca tutti gli articoli con "federated learning" dove nell'abstract appare "medicine". Otteniamo 14 risultati in 33ms. Il tempo di ricerca continua a confermare l'efficienza degli indici sui campi, anche se l'abstract in questo caso, avendo un testo più lungo rispetto agli altri campi, causa uno speedup minore nella ricerca.

[2] https://github.com/Xhst/de-projects/tree/main/project-1

	Tempi di ricerca (ms)		Numero di articoli restituiti su numero massimo richiesti
	Prima esecuzione	**Terza esecuzione**	
Query 1	32	16	50 / 50
Query 2	717	703	3362 / 9235
Query 3	23	16	50 / 50
Query 4	0	0	0 / 50
Query 5	16	0	4 / 50
Query 6	125	143	616 / 9235
Query 7	16	6	3 / 50
Query 8	16	7	3 / 50
Query 9	33	0	14 / 50
Query 10	1606	1771	9235/9235
Query 11	95	34	336 / 9235

Tabella 1: Tempi di ricerca dopo una e tre esecuzioni delle varie query e il numero di articoli ritornati da ciascuna.

Le restati query mirano a testare la funzionalità delle *wildcard* in Lucene. La query 10 esegue una ricerca senza nessun vincolo, con l'obiettivo di restituire tutti gli articoli presenti. A questo punto, va specificato che tra i 9372 articoli disponibili, abbiamo eliminato degli articoli vuoti o corrotti. Il nuovo totale è quindi 9235 articoli. L'interrogazione ritorna correttamente tutti gli articoli impiegando 1606ms. L'ultima query analizzata (n.11) vuole testare il caso in cui vogliamo ritornare tutti gli articoli tranne alcuni con delle caratteristiche. Per esempio cerchiamo tutti gli articoli tranne quelli dove è presente la parola "data". Otteniamo 336 articoli in 95ms. Possiamo testare la correttezza della query eseguendone un'altra su "data" e aggiungendo il numero di articoli restituiti dalla query 11, ovvero $336 + 8899 = 9235$.