HW1 Ingegneria dei Dati – Indicizzazione con Lucene

Link GitHub: `https://github.com/pietroyellow46/Data-Engineering`

Struttura del progetto ed esecuzione

Architettura generale del progetto

Il progetto è stato realizzato in **Java** utilizzando la libreria **Apache Lucene** per l'indicizzazione e la ricerca dei documenti testuali.

I principali file nel progetto sono:

- **Indexer.java:** classe responsabile dell'indicizzazione dei file di testo presenti nella cartella `text/`. Utilizza un `StandardAnalyzer` per il campo `contenuto` e un `StringField` per il campo `nome`, con l'obiettivo di creare un indice testuale consultabile. L'indice viene salvato all'interno della cartella `index/` e viene sovrascritto a ogni nuova esecuzione.
- **Searcher.java:** classe che consente di effettuare ricerche sull'indice creato. Gestisce sia ricerche per termine singolo (es. `contenuto:lucene`) che ricerche per frase (es. `contenuto:"motori di ricerca"`), simulando il comportamento dello `StandardAnalyzer` anche per le query di tipo *phrase*. Inoltre permette di effettuare ricerche esatte sul nome del file (campo nome) tramite `TermQuery`.
- **ConfigLoader.java:** classe di supporto per il caricamento delle variabili di configurazione definite nel file `config.properties`. In tale file vengono specificati i percorsi di `index.path`, `text.path` e altri parametri del sistema, in modo da evitare la modifica diretta del codice sorgente.
- **MainApp.java:** classe orchestratrice che fornisce un menù interattivo a terminale consentendo di scegliere se eseguire l'indicizzazione o la ricerca.

Metodi di esecuzione del progetto

Il progetto può essere eseguito in due modalità alternative: tramite **Maven** oppure manualmente con i comandi `javac` e `java`.

Esecuzione tramite Maven

È necessario avere installato:

- **JDK 17** o superiore
- **Apache Maven 3.8+**

Comandi da eseguire nella cartella principale del progetto:

```
mvn clean compile          # Compilazione del codice
mvn exec:java              # Esecuzione del programma principale
```

Esecuzione manuale con javac e java

Nel caso non si voglia utilizzare Maven, è possibile eseguire manualmente i file sorgenti, avendo cura di:

- Installare JDK 17 o superiore
- Scaricare manualmente le librerie Lucene (ad esempio lucene-core-9.2.0.jar, lucene-analyzers-common-9.2.0.jar, lucene-queryparser-9.2.0.jar, lucene-codecs-9.2.0.jar)
- Inserire tutti i file .jar all'interno della cartella `lib/`

Comandi da eseguire nella cartella principale del progetto:

```
# Compilazione e esecuzione (Windows)
javac -cp "lib/*" -d .class src/main/java/it/uniroma3/*.java
java -cp ".class;lib/*" it.uniroma3.MainApp
```

In alternativa, per eseguire singolarmente i moduli:

```
# Indicizzazione e ricerca (Windows)
java -cp ".class;lib/*" it.uniroma3.Indexer
java -cp ".class;lib/*" it.uniroma3.Searcher
```

Per sistemi Linux o macOS sostituire al punto e virgola ";" i due punti ":".

Analyzer scelti e motivazioni

Per l'indicizzazione dei campi sono stati scelti analyzer differenti in base al tipo di campo. Nel progetto sono stati utilizzati due differenti approcci di analisi testuale per i campi indicizzati, con l'obiettivo di sperimentare le principali tipologie di Analyzer e comprenderne l'impatto sulle query e sui risultati.

1. Campo nome

Per il campo che rappresenta il nome del file (`nome`) è stato utilizzato un `StringField` associato a una `TermQuery`. Questo campo non viene analizzato, cioè non subisce alcuna tokenizzazione, rimozione di stopword o conversione in minuscolo. In questo modo la ricerca è esatta e case-sensitive, garantendo che soltanto i nomi di file perfettamente coincidenti vengano restituiti nei risultati (ad esempio, `nome:lucene.txt` trova il file, mentre `nome:LUCENE.txt` no). Tale scelta permette di distinguere con precisione documenti diversi anche in presenza di maiuscole/minuscole, simulando un comportamento realistico di ricerca di file in un filesystem.

2. Campo contenuto

Per il campo `contenuto`, invece, è stato adottato un `TextField` analizzato tramite lo `StandardAnalyzer` di Lucene. Questo analyzer effettua automaticamente:

- tokenizzazione in base a spazi e punteggiatura
- rimozione delle stopword in lingua inglese
- conversione di tutto il testo in minuscolo

Il medesimo analyzer (`StandardAnalyzer`) è stato usato anche lato ricerca nelle query testuali normali (`QueryParser`), per garantire la coerenza tra l'indicizzazione e la fase di interrogazione. Tuttavia, per le query di tipo frase (`PhraseQuery`) è stata implementata manualmente una simulazione dello `StandardAnalyzer`: le frasi tra virgolette vengono trasformate in minuscolo, private dei caratteri speciali e suddivise in token, così da permettere un confronto sequenziale corretto. Questo approccio è stato necessario perché il costruttore diretto di `PhraseQuery` non esegue alcuna analisi automatica, e senza tale simulazione le ricerche di frasi con maiuscole o punteggiatura non avrebbero prodotto risultati.

Numero di file indicizzati e tempi di indicizzazione

Il sistema indicizza automaticamente tutti i file di testo presenti nella cartella configurata nel file `config.properties`. Per il test sperimentale, oltre ai documenti creati manualmente, è stato utilizzato un subset di 160 righe del dataset pubblico (`https://huggingface.co/datasets/UniqueData/amazon-reviews-dataset`) disponibile su HuggingFace. Durante i test, su macchina locale (Intel i5, SSD, Java 17), il tempo medio di indicizzazione è stato di circa 1.400 ms (1,4 secondi). Il valore è stato calcolato come media su più esecuzioni consecutive. La misurazione è stata eseguita tramite `System.nanoTime()` all'interno del blocco di scrittura dell'indice.

Query di test e risultati

Le seguenti query sono state utilizzate per verificare il corretto funzionamento del motore di ricerca, esplorando tutti i casi d'uso principali: ricerche esatte, errori di maiuscole, frasi tra virgolette e query malformate. Di seguito vengono riportate le query eseguite e i rispettivi risultati.

Query 1 – Ricerca per nome (TermQuery)

Descrizione: La ricerca sul campo `nome` utilizza una `TermQuery` non analizzata; quindi, trova corrispondenza solo se il valore coincide esattamente con quello indicizzato.

Query: `nome:lucene.txt`

Documenti trovati:

1. **lucene.txt** - contiene: Apache Lucene è una libreria Java open-source per la ricerca full-text...

Query 2 – Ricerca per nome con maiuscole

Descrizione: Il campo `nome` non viene elaborato dallo `StandardAnalyzer`, quindi la ricerca è case-sensitive. "LUCENE.txt" non coincide con "lucene.txt" e non produce risultati.

Query: `nome:LUCENE.txt`

Risultato: Nessun documento trovato.

Query 3 – Ricerca per nome con errore

Descrizione: Il nome del file reale è "ciao a tuTTi.txt". Poiché la ricerca richiede una corrispondenza esatta, le differenze nelle maiuscole o nelle lettere impediscono il match.

Query: `nome:ciao a tutti.txt`

Risultato: Nessun documento trovato.

Query 4 – Ricerca per nome corretta

Descrizione: La corrispondenza esatta del nome del documento produce il risultato atteso. È un esempio del funzionamento preciso della `TermQuery` per i nomi dei file.

Query: `nome:ciao a tuTTi.txt`

Documenti trovati:

1. **ciao a tuTTi.txt** - contiene: Ciao a tutti! Oggi è una splendida giornata piena di energia e nuove opportunità...

Query 5 – Ricerca testuale semplice su contenuto

Descrizione: Viene utilizzato lo `StandardAnalyzer`, che converte in minuscolo e rimuove la punteggiatura. Tutti i documenti che contengono la parola "lucene" vengono trovati.

Query: `contenuto:lucene`

Documenti trovati:

1. **elasticsearch.txt** - contiene: Elasticsearch è un motore di ricerca e analisi distribuito basato su Lucene...
2. **index.txt** - contiene: Un inverted index è una struttura dati ... come nel caso di Apache Lucene ...
3. **lucene.txt** - contiene: Apache Lucene è una libreria Java open-source per la ricerca full-text ...

Query 6 – Ricerca testuale con più termini

Descrizione: La query implicitamente mette in OR tutti i termini che formano la query. Tutti i documenti che contengono i termini "analisi" o "lucene" vengono restituiti.

Query: `contenuto:analisi lucene`

Documenti trovati:

1. **elasticsearch.txt** - contiene: Elasticsearch è un motore di ricerca e analisi distribuito basato su Lucene...
2. **index.txt** - contiene: Un inverted index è una struttura dati ... come nel caso di Apache Lucene ...
3. **lucene.txt** - contiene: Apache Lucene è una libreria Java open-source per la ricerca full-text ...

Query 7 – PhraseQuery con variazioni di maiuscole

Descrizione: Grazie al phrase processing attivo, la frase viene normalizzata (minuscole, rimozione di segni). La `PhraseQuery` trova la corrispondenza anche con parole digitate in maiuscolo.

Query: `contenuto:"motori di RICERCA"`

Documenti trovati:

1. **analisi.txt** – contiene: L'analisi del testo estrae ... È fondamentale per motori di ricerca, chatbot e ...
2. **retrival.txt** – contiene: L'information retrieval studia come ... È alla base dei motori di ricerca moderni ...

Query 8 – PhraseQuery con errore di sintassi

Descrizione: Manca la preposizione "di"; L'assenza della sequenza esatta impedisce la corrispondenza, e la query non restituisce risultati.

Query: `contenuto:"motori ricerca"`

Risultato: Nessun documento trovato.

Query 9 – Campo inesistente

Descrizione: Il campo "campoFinto1" non esiste nell'indice. Lucene gestisce il caso restituendo semplicemente nessun risultato, senza errori di esecuzione.

Query: `campoFinto1:campoFinto2`

Risultato: Nessun documento trovato.

Query 10 – Sintassi errata

Descrizione: L'input non rispetta il formato previsto (manca ":" come separatore). Il software riconosce la sintassi errata e mostra un messaggio di errore.

Query: `contenuto-ciao`

Risultato: Sintassi query errata! Usa `nome:term` o `contenuto:phrase`.