Università degli Studi di Roma Tre

Ingegneria dei Dati

Homework 3

Data Discovery tramite Indicizzazione di file JSON

Antonio Lanza, Daniel Luca

Project repository: https://github.com/AntonioSouls/Data-Engineering-Projects/tree/master/homework3

1. Introduzione

Avendo, con il precedente lavoro sulla "*Creazione di un Motore di Ricerca per documenti HTML*", iniziato a muovere i primi passi verso la scoperta di nuove tecnologie e nuovi strumenti utili per la risoluzione di problematiche in ambito Data Science, si è deciso di proseguire il caso di studio approfondendo la comprensione e l'applicazione di tali tecnologie (si veda immagine 1) in un ambito più complesso.
Infatti, il progetto proposto in questo frangente richiama nuovamente le tecnologie già utilizzate, ma richiede esplicitamente di applicarle per l'indicizzazione non più di comuni file HTML, ma di tabelle all'interno di file JSON. L'obiettivo è dunque sfruttare il suddetto indice per sviluppare un motore di ricerca di dati scientifici all'interno delle tabelle indicizzate.
Tale richiesta, apparentemente molto simile a ciò di cui discusso nello scorso elaborato, nasconde in realtà non poche insidie ed è tesoro di sfide più complesse rispetto a quelle affrontate precedentemente. Tali sfide hanno portato a maturare una conoscenza più completa degli strumenti di indicizzazione come **Lucene** ed hanno consentito un miglioramento della capacità di ragionamento. Inoltre hanno fornito un notevole spunto per migliorare nella gestione dei problemi.

Compito di questo elaborato, dunque, è quello di illustrare e spiegare brevemente la soluzione che si è deciso di implementare per sopperire alla richieste effettuate da questo terzo homework, sottolineando anche il valore aggiunto e le conoscenze raggiunte attraverso lo svolgimento di tale progetto.

Figura 1: Loghi relativi alle tecnologie utilizzate. Da sinistra a destra: JSON, Java, IntelliJ ed Apache Lucene.

2. Svolgimento

Il progetto proposto risulta suddiviso in quattro momenti fondamentali: la **fase di indicizzazione**, la **fase di test**, lo **sviluppo del motore di ricerca** e lo **sviluppo delle metriche**. Questa suddivisione è stata scelta per garantire un approccio strutturato e granulare, che consente di affrontare ogni aspetto del progetto in modo sistematico e progressivo.

Scegliendo dunque di adottare questa impostazione del progetto, si è favorito uno sviluppo logico delle componenti, progettando in primo luogo quelle basilari (necessarie per costruire le fondamenta del codice) e, solo in un secondo momento, implementando le funzionalità più complesse che costituivano il core del progetto, assicurando, in tutto ciò, anche un controllo efficace durante l'avanzamento del lavoro. Questo ha garantito una crescita solida del progetto in quanto si è costantemente testato il corretto funzionamento delle componenti alla base prima di progredire con le logiche più complesse. Ogni fase, dunque, risulta strettamente interconnessa alle altre, contribuendo in modo specifico e complementare al raggiungimento degli obiettivi finali del progetto.

Nel corso del capitolo, saranno esaminati i dettagli operativi e metodologici di ciascuna fase, evidenziandone le finalità, i passaggi principali e i risultati ottenuti, al fine di offrire una visione completa e organica dello sviluppo

2.1 Fase di Indicizzazione

La *Fase di Indicizzazione*, seppur rappresenta il primissimo step per la costruzione dell'intero progetto, è risultata essere il primo grande scoglio da superare. Essendo il momento in cui si struttura la logica per una corretta ricerca delle informazioni nei file, è senz'altro una fase fondamentale da gestire con cura.
Il contesto è quello in cui, da un folder contenente un ammontare di all'incirca 10k documenti JSON, bisogna estrarre tutte le tabelle in essi contenute ed inserirle dentro un indice Lucene. L'estrazione dai file JSON e l'inserimento nell'indice non vanno sottovalutate, poiché un'estrazione errata delle tabelle o una logica di indicizzazione mal concepita potrebbero comportare la mancata inclusione di alcune informazioni rilevanti o la restituzione di dati non coerenti con le richieste dell'utente del motore

di ricerca. Dunque, in maniera minuziosa, si è proceduto in primis con l'estrazione delle tabelle dai file dove, per ogni file, si è prestata particolare attenzione ad analizzare ogni suo elemento, verificando metodicamente che si trattasse di reali tabelle ed escludendo quelle che non corrispondevano ad un corretto formato tabellare (si veda 2.2). Degli elementi che superavano questo filtraggio, si estraeva l'informazione relativa all'*id della tabella*, la *caption della tabella*, il *contenuto della tabella* stessa e, inoltre, dal contenuto si estraevano i titoli delle colonne e delle righe della tabella in un formato stringato. Tali informazioni estratte sono poi state indicizzate attraverso le **funzioni Lucene** diventando così parte di un indice capace di cercare dettagliatamente il contenuto in ciascuno dei campi estratti e per ognuna delle tabelle estratte.

Chiaramente, una corretta indicizzazione è sempre accompagnata dalla definizione di alcuni Analyzer che possano consentire il parsing della query dell'utente ed il suo confronto con tutti i campi indicizzati delle varie tabelle al fine di trovare la massima similarità tra la query stessa e alcune delle tabelle, restituendo così quelle risultate più pertinenti.
Si è deciso, così, di creare alcuni Analyzer personalizzati che consentissero:

- Di realizzare un Natural Language Processing sul campo caption;
- La tokenizzazione come termine singolo di tutti i simboli che costituissero l'indice della tabella;
- Il processamento in linguaggio naturale anche delle stringhe relative alle righe e colonne della tabella, ricercando su queste anche sinonimi rispetto alle parole della query invece che ricercare solo strettamente i termini della query stessa;

Di seguito viene mostrato il codice relativo ai vari Analyzer personalizzati che si è deciso di implementare:

```
01 | public static Analyzer getTableCaptionAnalyzer() {
02 |         try {
03 |             return CustomAnalyzer.builder()
04 |                     .withTokenizer(WhitespaceTokenizerFactory.class
       )
05 |                     .addTokenFilter(LowerCaseFilterFactory.class)
06 |                     .addTokenFilter(WordDelimiterGraphFilterFactory
       .class)
07 |                     .addTokenFilter(EnglishPossessiveFilterFactory.
       class)
08 |                     .addTokenFilter(PorterStemFilterFactory.class)
09 |                     .build();
10 |         }
11 |         catch (IOException e) {
12 |             throw new RuntimeException(e);
13 |         }
14 |     }
```

```
01 | public static Analyzer getTableIdAnalyzer() {
02 |         try {
03 |             return CustomAnalyzer.builder()
04 |                     .withTokenizer(KeywordTokenizerFactory.class)
05 |                     .addCharFilter(PatternReplaceCharFilterFactory.
       class,"pattern", "_", "replacement", "")
06 |                     .addTokenFilter(LowerCaseFilterFactory.class)
07 |                     .build();
08 |         }
09 |         catch (IOException e) {
10 |             throw new RuntimeException(e);
11 |         }
12 |     }
```

```
01 | public static Analyzer getTableColumnsAnalyzer(){
02 |         try {
03 |             return CustomAnalyzer.builder()
04 |                     .withTokenizer(KeywordTokenizerFactory.class)
05 |                     .addTokenFilter(LowerCaseFilterFactory.class)
06 |                     .build();
07 |         }
08 |         catch (IOException e) {
09 |             throw new RuntimeException(e);
10 |         }
11 |     }
```

```
01 | public static Analyzer getTableRowsAnalyzer(){
02 |         try {
03 |             return CustomAnalyzer.builder()
04 |                     .withTokenizer(KeywordTokenizerFactory.class)
05 |                     .addTokenFilter(LowerCaseFilterFactory.class)
06 |                     .build();
07 |         }
08 |         catch (IOException e) {
09 |             throw new RuntimeException(e);
10 |         }
11 |     }
```

Chiaramente, per verificare un comportamento pulito di questa fase di Indicizzazione, sono state effettuate delle misurazioni di performance, valutando il tempo totale speso per la realizzazione dell'indice. Questo tempo si aggira attorno ai 2 minuti e 30 secondi, tempo che sembra essere sostenibile per questa tipologia di task.

2.2 Fase di Test

Avendo sottolineato la crucialità della *Fase di Indicizzazione*, risulta chiaro come ogni singolo step descritto nel precedente paragrafo sia stato accompagnato da test solidi che ne verificassero il corretto funzionamento sintattico e semantico.
Scopo dei test, infatti, era quello di garantire che ogni procedura cruciale dell'estrazione delle tabelle e dell'indicizzazione dei file avvenisse in maniera fluida e corretta, senza errori semantici che portassero il programma a funzionare in maniera diversa dalla logica pensata.
Sono state create, dunque, due classi di test:

- **TableExtractorTest** = Classe volta a verificare la corretta estrazione di tutte le tabelle dai file JSON e giustificasse le motivazioni della mancata estrazione di alcune delle tabelle esistenti;
- **JSONIndexerTest** = Classe volta a verificare la corretta indicizzazione di tutte le tabelle estratte e testasse il funzionamento della ricerca sull'indice attraverso qualche *TermQuery*;

La prima delle due classi elencate è stata ragionata seguendo la logica del contare, per ogni file JSON, il numero di tabelle totali presenti nel file e il numero di tabelle non estratte, memorizzando anche il contenuto delle tabelle non estratte al fine di consentire una comprensione delle motivazioni per cui tali tabelle venivano escluse. Tale logica ha permesso di ricavare l'esatto numero totale di tabelle presenti in tutti i file JSON ed il numero complessivo di tabelle non estratte. Il risultato ottenuto è stato di 5.068 tabelle non estratte su un ammontare complessivo di 49.538 tabelle, circa il 10%.
Attraverso il salvataggio delle informazioni relative a tutte le tabelle non estratte, si è potuto osservare come le motivazioni di tale mancata estrazione fossero legate al fatto che le tabelle risultavano o vuote oppure non erano delle reali tabelle, ma piuttosto informazioni testuali in formato tabellare. Dunque, la loro mancata estrazione era perfettamente giustificata.

La classe **JSONIndexerTest**, invece, è stata progettata semplicemente implementando una *AllDocsQuery* che restituisse il numero totale di tabelle indicizzate. Questo numero è risultato essere pari a 44.470 tabelle, il che ci sorprende in positivo in quanto:

$$\text{TOTALE\_TABELLE} - \text{TABELLE\_NON\_ESTRATTE} = \text{TABELLE\_ESTRATTE}$$

$$49.538 - 5.068 = 44.470$$

Il che significa che tutte le tabelle estratte sono state realmente indicizzate.
La classe **JSONIndexerTest**, inoltre, è stata usata anche per testare il funzionamento della ricerca sull'indice. Test che ha portato comunque a buoni risultati in quanto, la ricerca di un termine specifico di una specifica tabella portava la restituzione (tra i risultati ritenuti pertinenti da Lucene) anche della tabella in questione, sottolineando il corretto funzionamento della ricerca per delle *OneTermQuery*.

2.3 Sviluppo del Motore di Ricerca

Quanto descritto fino ad ora rappresenta la logica con cui si è deciso di indicizzare le varie tabelle. Questo è necessario per uno scopo preciso: Creare un Motore di Ricerca.
L'obiettivo che il progetto ci chiedeva di raggiungere, infatti, era proprio quello di simulare un Browser che permettesse di ricercare informazioni significative all'interno delle tabelle e, come visto nei precedenti paragrafi, è fondamentale la costruzione di un indice previa realizzazione del Motore di Ricerca.
Questo spiega il motivo per cui si è deciso di soddisfare la richiesta del progetto solamente in questa fase, trascurandola negli step precedenti.

Il Motore di Ricerca da noi proposto permette di effettuare:

- **Una ricerca sul Table_ID** = Ossia, si è pensato che (in una situazione reale) potrebbe accadere di ricordare l'ID di una tabella, ma non ricordare in quale file fosse collocata. Dunque, si è pensato di soddisfare tale esigenza consentendo una ricerca esclusivamente sul campo ID, in modo che l'utente possa inserire direttamente l'ID della tabella cercata così che gli venga restituita. Chiaramente, essendoci una leggera replicazione degli ID (che si sono dimostrati non univoci per tutto il corpus di documenti), può succedere che ad una

query di questo tipo corrispondano più tabelle come risultato, ma tale problematica non crea disagi. Questo perché, nonostante gli ID replicati, la tabella cercata viene sempre restituita tra le 10 tabelle maggiormente pertinenti;

- **Una ricerca complessa** = E' la ricerca effettiva di un contenuto, di una *PhraseQuery* su tutte le tabelle. Rappresenta la situazione più probabile in cui un utente ricerca delle informazioni specifiche all'interno di tutto il corpus di Tabelle. Per implementare questa logica, si è deciso di leggere la query da tastiera, farne il parsing utilizzando gli stessi Analyzer descritti nella sezione 2.1, e poi cercare il contenuto parsato all'interno della *Caption della Tabella* e all'interno del*Contenuto della Tabella*. In particolar modo, però, la ricerca sul contenuto è stata pensata in modo che le informazioni venissero ricercate sulle intestazioni delle righe e delle colonne. Di fatto, si è ritenuto che, in uno scenario reale, le query possibili non contengano informazioni relative a dati interni alla tabella, ma che piuttosto siano più generiche e ricerchino informazioni descrittive della tabella come, appunto, le informazioni contenute nell'intestazione di righe e colonne o informazioni contenute nella *Caption*;

Il Motore di Ricerca, poi, mostra i risultati per una specifica query in ordine di pertinenza, la quale è rappresentata tramite uno score.

2.4 Sviluppo delle Metriche

Per valutare il motore di ricerca realizzato, come da richiesta, sono state appositamente implementate delle metriche di valutazione.
Nello specifico è stata realzizata una classe in java che racchiudesse tali metriche dove, per il momento, sono state implementate la **Mean reciprocal rank (MRR)** e la **Normalized discounted cumulative gain (NDCG)**.
Per valutare correttamente il sistema, bisognava confrontare i risultati ottenuti inserendo una determinata query nel motore di ricerca, con una "grand truth" che fornisse un giusto ordine dei risultati piu rilevanti che dovrebbero essere restituiti eseguendo quella stessa query.
La soluzione che è stata pensata, è consistita nel considerare un piccolo insieme di query molto specifiche (10 per l'esattezza) e stilare manualmente un ordine delle prime 10 tabelle che dovrebbe restituire una specifica query. L'ordine dei risultati nei due diversi sistemi di ranking sono risultati quasi identici per tali 10 query specifiche perciò il valore delle due metriche è risultato piuttosto alto.