Università degli Studi di Milano - Bicocca
Scuola di Scienze
Dipartimento di Informatica, Sistemistica e Comunicazione
Corso di Laurea in Informatica

Enterprise Search con Apache Solr: un caso di studio per il settore delle Risorse Umane

Relatore: Prof.ssa Gabriella Pasi

Relazione della prova finale di:
Marco Ferri
Matricola 807130

Anno Accademico 2017-2018

Ai miei genitori.

Abstract

Lo studio nasce dalla curiosità di comprendere i meccanismi che regolano il funzionamento di un motore di ricerca. Lo scopo finale consiste nella progettazione e realizzazione di un sistema che, a partire da un'importante quantità di informazioni estratte da fonti eterogenee, permetta agli utenti di effettuare, in maniera intuitiva, ricerche rapide ed efficienti per ottenere risultati in linea con le aspettative.

La tesi si sviluppa secondo un approccio di indagine progressivo.

Nella prima parte dello studio viene analizzato il contesto all'interno del quale nasce l'esigenza di effettuare una ricerca, quali sono le problematiche che ne scaturiscono e le soluzioni usualmente adottate. Particolare attenzione è dedicata al concetto di analisi del testo, presentato attraverso modelli e algoritmi che formalizzano il problema.

L'approccio teorico è seguito da un capitolo dedicato ad Apache Solr, piattaforma di *enterprise search* che svincola l'utilizzatore dai modelli matematici e implementa ad alto livello i concetti precedentemente affrontati. Dopo una breve occhiata al funzionamento di Apache Solr, vengono presentate le soluzioni che esso propone per affrontare lo sviluppo di un sistema basato sulla ricerca *full text*.

La terza parte dell'elaborato costituisce quella più corposa. Viene preso in considerazione un caso di studio reale, un software di tipo *Customer Relationship Management* (CRM) con un orientamento *Applicant Tracking System* (ATS) per la Ricerca e la Selezione del Personale. All'analisi dei requisiti segue la progettazione del motore di ricerca, successivamente sviluppato con il supporto di Solr ed un approccio di miglioramento incrementale. In questa fase vengono discussi i problemi riscontrati e le scelte intraprese per la loro risoluzione.

Come risultato dello stage viene infine presentato il prototipo del motore di ricerca da me sviluppato a supporto ed integrazione del software già esistente. Particolare attenzione è dedicata all'esperienza utente, parte integrante della progettazione del sistema sia da un punto di vista funzionale che prestazionale.

Indice

Introduzione

Si è recentemente diffuso il concetto di "big data" e al giorno d'oggi sono costantemente a disposizione in rete un'enorme quantità di dati, i quali rappresentano tutto ciò che l'uomo conosce, produce e di cui fa uso nella propria quotidianità. Fino al secolo scorso era difficile immaginare che la rivoluzione tecnologica a cui si è assistito negli ultimi anni avrebbe generato un incremento così massiccio dei dati che vengono perennemente scambiati da una parte all'altra del mondo. Tale quantità è destinata ad aumentare ed è di fondamentale importanza che si impari a gestire il fenomeno, affinché questi dati non siano solo un insieme di lettere e numeri ma vengano interpretati perché possano costituire un elemento informativo.

Il primo responsabile dell'aumento esponenziale dei dati è stato il World Wide Web, grazie al quale si è vista nascere la possibilità di condividere con chiunque il proprio sapere e creare così un'immensa rete di conoscenza e di informazioni interconnesse. Insieme alla nascita del WWW, negli anni '90, si diffonde quindi la necessità di doversi orientare nella miriade di pagine e documenti che hanno popolato la rete di anno in anno, ed è a partire da questa esigenza che i motori di ricerca hanno assunto un ruolo fondamentale. Se in precedenza erano utilizzati solo in ambiti aziendali ed istituzionali, i motori di ricerca sul Web si sono progressivamente fatti conoscere ad un pubblico decisamente più ampio e tutt'oggi sono costantemente utilizzati da chiunque.

Con il passare del tempo, i motori di ricerca sono diventati sempre più precisi e hanno iniziato ad essere utilizzati per consultare e ricercare oltre che informazioni testuali, anche contenuti multimediali: foto, audio e video. Se la ricerca di foto e video è stata in prima battuta accompagnata dalla catalogazione manuale - cioè umana - delle informazioni, negli ultimi anni si sta assistendo all'automatizzazione del processo, per merito di sistemi di riconoscimento sempre più precisi e diffusi in grado di determinare automaticamente quale possa essere il contenuto di un'immagine o di un video. Infine, nonostante i motori di ricerca di tracce audio siano sicuramente meno conosciuti, chiunque di noi ne ha probabilmente utilizzato uno senza sapere che si trattasse proprio di questo: Shazam. Il medesimo tipo di ricerca consente inoltre di rilevare in maniera automatica contenuti multimediali sul Web in violazione dei diritti d'autore.

Appena un decennio dopo la sua comparsa, il WWW ha portato con sé anche l'avvento dei *social media*, che hanno determinato un cambiamento radicale nelle vite di tutti. Tale cambiamento è causa dell'elevata diffusione di nuovi contenuti digitali e

informazioni personali che, ogni giorno, vengono numerosamente condivisi con la propria rete di contatti. Il 2018 è stato e continua ad essere un anno di fervore proprio su questo argomento ed in particolare sulla condivisione, detenzione e utilizzo dei dati personali.

Se quel che facciamo, come ci comportiamo, viene reso disponibile online da noi stessi attraverso un *social network* o indirettamente attraverso il nostri spostamenti (si pensi all'utilizzo sempre più diffuso dei *wearable device* o delle automobili *smart*), è chiaro che tali informazioni devono essere raccolte ed elaborate da qualcuno. Questo "qualcuno" non sono altro che le aziende proprietarie delle tecnologie e delle piattaforme che quotidianamente utilizziamo per condividere le nostre informazioni, ed è essenziale che tali aziende sappiano gestire questi dati affinché possano costituire un elemento informativo, che generi un profitto o quantomeno si riveli di una certa utilità.

In un mondo nel quale il "dato" sembra essere l'elemento sotto l'attenzione di tutti, che le grandi aziende dei *social* sanno opportunamente gestire per ottenere dei risultati profittevoli, si assiste contemporaneamente all'esistenza di numerose realtà all'interno delle quali non si riesce a valorizzare a dovere i dati che si detengono, che talvolta risultano sprecati o di difficile consultazione. Accade spesso, infatti, che molte aziende o istituzioni siano responsabili di enormi archivi documentali ricchi di informazioni potenzialmente utili, che possono però finire per rivelarsi più un peso che un'opportunità.

Questo elaborato nasce proprio dall'esigenza di fare ordine all'interno di un archivio documentale e, in particolare, considera un caso di studio reale per il settore delle risorse umane. È importante notare, infatti, che la stragrande maggioranza dei processi di ricerca e selezione del personale si svolgono attraverso il solo scambio di un documento di testo: il *curriculum vitae* (CV). Nonostante si assista a molte situazioni che prevedono l'utilizzo di appositi moduli di candidatura, atti a strutturare precisamente le informazioni fornite dal candidato, anche questi si concludono sempre e comunque con la richiesta di un CV.

Essendo il *curriculum* un file testuale, è chiaro che un archivio composto da questo tipo di documenti appare complicato da consultare e ciò determina una seria difficoltà nel reperire in esso informazioni rilevanti in maniera rapida ed efficace. Per questo motivo, si è cercato di comprendere in che modo sia possibile elaborare archivi costituiti da documenti testuali affinché vi si possano facilmente reperire delle informazioni che si rivelino effettivamente utili agli occhi di un *recruiter*.

Lo studio condotto sui temi di reperimento delle informazioni ha prodotto come risultato il prototipo di un motore di ricerca, il cui funzionamento è stato integrato all'interno di Arkivium Recruiting, software adibito alla gestione dei processi di ricerca e selezione del personale. Questo progetto di laurea documenta le fasi che si sono susseguite per la progettazione e lo sviluppo del motore di ricerca, che ha l'obiettivo di consentire all'utente di cercare - attraverso un'unica interfaccia Web - all'interno della totalità delle informazioni a propria disposizione, cioè in maniera congiunta all'interno di conoscenze strutturate, provenienti da un database, ed in quelle non strutturate (o semi-strutturate) contenute nei *curriculum* o in altri tipi di file testuali.

Capitolo 1

Reperimento delle Informazioni per Ricerche Testuali

Parlando di "Reperimento delle Informazioni" (in inglese *Information Retrieval*, abbreviato IR) si fa riferimento a problemi ed esigenze che non caratterizzano la sola età moderna: l'essere umano ha da sempre avuto la necessità di reperire informazioni in maniera semplice ed efficace. Fino a qualche decennio fa, l'unico modo per poter cercare e ritrovare informazioni ben precise, contenute all'interno di una base documentale, consisteva nel suddividere e catalogare le fonti a propria disposizione, quanto più precisamente possibile, in biblioteche e archivi.

L'avvento dei computer ha aperto nuove opportunità. Se precedentemente era impensabile poter cercare una specifica frase o immagine all'interno di una collezione di libri, ciò è stato reso possibile grazie al trattamento dell'informazione in formato digitale. La rappresentazione di una crescente mole di documenti in digitale e la necessità di reperire velocemente informazioni in grado di soddisfare le esigenze informative dell'utente sono i motivi che hanno contribuito alla nascita dell'Information Retrieval (IR), che vede la luce nei primi anni '60 e trova il suo massimo punto di espansione dopo gli anni '90, con l'avvento del World Wide Web.

L'**Information Retrieval** è la disciplina informatica che si occupa della rappresentazione e dell'organizzazione di ingenti quantità di risorse digitali, al fine di consentire il reperimento di informazioni che si rivelino utili a soddisfare specifiche necessità informative espresse dall'utente mediante la formulazione di una *query*. Il risultato principale degli studi e della ricerca nel campo dell'IR sono i **motori di ricerca**, *Information Retrieval System* (abbreviato in IRS) che svolgono il ruolo di intermediari fra sorgenti dei dati e utente finale. In questa tesi si sono considerati motori di ricerca che gestiscono esclusivamente collezioni di testi.

1.1 Struttura di un Information Retrieval System

Un sistema di reperimento delle informazioni è costituito da molteplici componenti che cooperano per

- rappresentare formalmente i documenti che costituiscono l'archivio dei dati
- formalizzare le esigenze informative dell'utente, espresse attraverso la formulazione di una query
- confrontare query e documenti
- presentare all'utente i documenti che vengono stimati rilevanti rispetto alla query

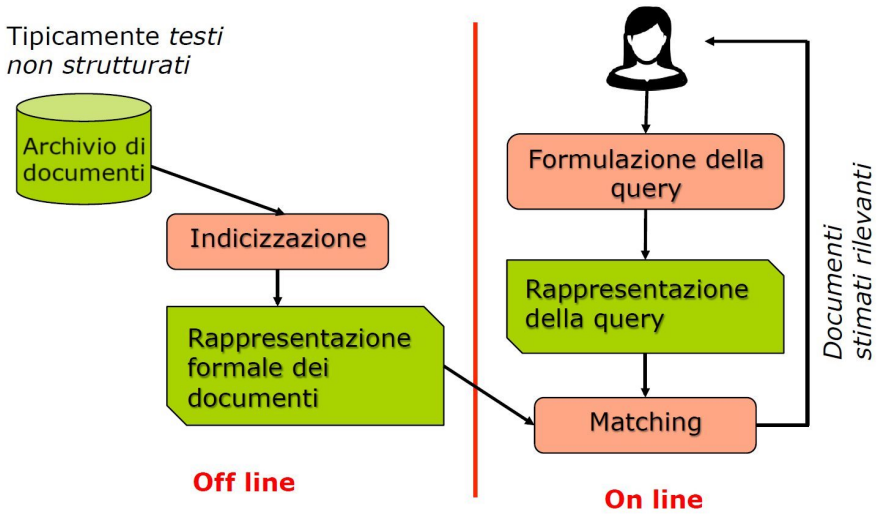

Figura 1.1: Struttura di un IRS

L'obiettivo principale dell'IRS consiste nel reperire, a seguito di una query, tutti i documenti rilevanti per l'utente, minimizzando al contempo il reperimento di documenti non rilevanti. Il concetto di **rilevanza**, difficile sia da modellare che da valutare, esprime l'utilità di un documento nel soddisfare una specifica esigenza informativa e costituisce uno degli elementi chiave attorno al quale si sviluppa un IRS.

Un sistema per il reperimento delle informazioni è basato su un **modello matematico** che consente di produrre una rappresentazione formale del problema e di definire opportune metodologie di confronto, in grado di stimare la rilevanza di un documento rispetto ad una query. Si rivela difficile valutare la correttezza del risultato prodotto dal confronto, poiché influenzato da numerosi fattori di incertezza e soggettività.

Seppur l'unica parte visibile di un motore di ricerca sia costituita dall'interfaccia che consente all'utente di formulare delle query e consultarne i risultati, la ricerca è frutto di una catena di operazioni complessa, approfondita gradualmente nel corso della tesi fino allo sviluppo di un motore di ricerca vero e proprio.

1.2 Indicizzazione del testo

Un sistema di reperimento delle informazioni opera con archivi costituiti come agglomerato di documenti che provengono da sorgenti di diversa natura. Il primo compito del sistema consiste nell'elaborazione di ciascun documento per ricavarne il contenuto informativo, affinché possa essere catalogato e memorizzato nella maniera opportuna.

Questo processo è detto **indicizzazione** e consiste nell'estrazione, da un documento, di opportuni "descrittori" in grado di rappresentarne formalmente ed esaustivamente il contenuto. Il risultato di questa elaborazione sono gli **indici**, strutture dati che costituiscono l'elemento base della rappresentazione formale di un archivio documentale e consentono di reperire informazioni in maniera molto rapida.

Gli indici adibiti alla rappresentazione di archivi testuali devono essere:

- **esaustivi**, cioè rappresentare tutti gli argomenti trattati nel testo;
- **specifici**, cioè in grado di discriminare il contenuto di due documenti diversi.

Fra le possibili soluzioni per l'analisi e l'indicizzazione del testo, una delle tecniche principali per la sua rappresentazione consiste nell'indicizzarne il contenuto per singoli **termini**. Ciò consiste in un processo di trasformazione del testo originario in un flusso di *token*, ovvero sequenze di caratteri (come le parole), che vengono successivamente elaborati per ricavare i termini da inserire nell'indice. I termini indice sono l'elemento base di un sistema di reperimento delle informazioni, in quanto sia la ricerca che le strutture di memorizzazione sono pensate per operare su insiemi di termini.

L'indicizzazione si rivela di fondamentale importanza, poiché è durante questa fase che un IRS decide cosa sia opportuno memorizzare, cosa invece scartare ed eventualmente come far sì che determinate informazioni appaiano più "importanti" di altre, cioè più adatte a descrivere efficacemente il contenuto di un documento. A tale scopo, è possibile catalogare gli elementi estratti da un documento per differenziare il trattamento che sarà riservato a ciascuno di essi. Ad esempio, è possibile determinare in che modo sia necessario elaborare e memorizzare l'*oggetto* piuttosto che il *corpo* o il *mittente* di una email.

1.2.1 Estrazione dei termini

L'analisi del testo, che consente di ricavarne i termini da indicizzare, è un processo meticoloso che si compone di molteplici passaggi, i quali costituiscono il primo passo per la rappresentazione formale dei documenti. Un IRS non è tenuto ad implementare l'intera catena di analisi del testo, ma è fondamentale che tale analisi sia effettuata in maniera compatibile anche durante la fase di interrogazione, affinché il meccanismo di confronto fra documenti e query sia in grado di produrre risultati attendibili.

Suddivisione in Token

Tutto comincia con la suddivisione del flusso di caratteri del testo originale in *token*, ricavati dalla separazione tramite spazi o segni di punteggiatura. In linea di massima, ogni parola è un *token*, ma può capitare che per la separazione dei termini vengano sfruttati per esempio anche numeri (*peer2peer*), caratteri speciali (*wi-fi*) o maiuscole (*WiFi*).

Input: Friends, Romans, Countrymen, lend me your ears;
Output: | Friends | | Romans | | Countrymen | | lend | | me | | your | | ears |

Figura 1.2: Esempio di suddivisione in *token*

Ogni *token* viene successivamente "normalizzato" tramite l'applicazione di un insieme di regole scelte arbitrariamente. Ad esempio, spesso accade che le lettere maiuscole vengano sostituite con le corrispondenti minuscole e che la medesima sorte sia riservata alle lettere accentate, rimpiazzate dalla propria forma base (*è* diventa *e*, *Ü* diventa *u*).

Rimozione delle Stop Word

Ogni lingua è caratterizzata da un insieme di parole che sono molto frequenti all'interno di un testo: articoli, preposizioni, pronomi, avverbi, verbi ausiliari, ... Considerata l'elevata frequenza con la quale vengono utilizzate, esse risultano poco adatte a descrivere il contenuto di un documento e influiscono negativamente sulla dimensione degli indici, motivi per cui è consuetudine ignorarle o rimuoverle durante il processo di indicizzazione. Queste prendono il nome di *stop word* e sono caratteristiche di ciascuna lingua.

Stemming

Anche nel caso dello *stemming* ci si trova di fronte ad un caso di analisi linguistica. Consiste nella "contrazione" di alcune parole alla loro forma base o alla radice di appartenenza, operazione che può essere effettuata nel caso ci si trovi ad esempio in presenza di plurali, verbi o parole che derivano da altre. Come la rimozione delle *stop word*, non è una fase obbligatoria dell'analisi testuale ma porta dei benefici sia in termini di dimensione degli indici che dal punto di vista delle interrogazioni, che grazie allo *stemming* possono godere di una maggiore flessibilità (ad esempio, facendo sì che cercare *studente* o *studenti* produca i medesimi risultati).

Per essere precisi, sarebbe opportuno specificare che lo *stemming*, seppur dipendente dalla lingua, consiste semplicemente nell'applicazione di alcuni algoritmi che troncano le parole secondo determinate regole prestabilite. Se si vuole ottenere un risultato più accurato e ridurre davvero ciascuna parola alla propria forma canonica (lemma) - ad esempio, in italiano, i verbi al proprio infinito - si dovrebbe parlare di **lemmatizzazione**, un'operazione più complessa che sfrutta opportuni dizionari linguistici.

Espansione dei Termini

Spesso accade che, sia in fase di indicizzazione che durante un'interrogazione, alcuni termini possano rivelarsi troppo specifici o, al contrario, troppo generici. Ne è un esempio il caso in cui cercando la parola *animale* un utente si aspetta che vengano reperiti anche tutti i documenti in cui compaiono le parole *cane* e *labrador*. Inoltre, si verifica spesso che una stessa parola possa assumere significati diversi a seconda del contesto all'interno del quale viene utilizzata. Per questi motivi è efficace poter definire delle correlazioni fra alcuni termini, assegnandovi significati e sinonimi specifici del contesto di appartenenza. Ciò è possibile attraverso l'utilizzo di appositi dizionari e tesauri (tematici o linguistici) che consentano di effettuare l'espansione di ciascun termine con altri ad esso correlati. Il vantaggio consiste nell'esecuzione di ricerche più utili per l'utente.

Questa è probabilmente la fase più complessa dell'analisi testuale poiché altamente dipendente sia dalla lingua che dal contesto applicativo. Un'opportuna implementazione richiede l'utilizzo di risorse costruite *ad hoc* per la realtà all'interno del quale si inserisce il motore di ricerca.

1.2.2 Pesatura dei termini

Ottenuta la lista dei termini che compongono la base di dati testuale, è fondamentale decidere il criterio secondo il quale l'occorrenza di un termine in un certo documento influisca sui risultati della ricerca in atto da parte dell'utente. È chiaro che non tutte le parole di un documento siano ugualmente utili a descriverne il contenuto informativo, pertanto si rivela fondamentale la possibilità di assegnare a ciascun termine un **peso**, che esprima l'importanza del termine stesso come descrittore del documento.

Una pesatura dei termini di tipo binario (`1` se il termine compare in un certo documento, `0` se non compare) non è sufficiente per ottenere dei risultati di ricerca abbastanza significativi. Per ogni termine è estremamente importante valutare anche quale sia la **frequenza** con cui appare sia all'interno di uno specifico documento, sia nell'intera collezione. Per questo motivo si è sviluppato il modello di pesatura dei termini conosciuto come *TF*IDF*. Secondo quest'ultimo, il peso w_{ij} di un certo termine t_j nel documento d_j prende il nome di **significatività**, che può essere calcolata con la seguente funzione matematica:

$$w_{ij} = f_{ij} \times log\frac{N}{d_{fi}}$$

Il primo fattore f_{ij} rappresenta la frequenza (il numero di occorrenze) del termine t_i nel documento d_j.
Il secondo fattore è detto *inverse document frequency (IDF)* ed è tanto più alto quanto minore è il numero di documenti in cui il termine t_i compare (N è il numero totale di documenti nella collezione e d_{fi} è il numero di documenti che contengono il termine t_i).

Poiché la frequenza di un termine in un documento cresce naturalmente con la lunghezza del documento stesso, è opportuno che il parametro f_{ij} venga normalizzato come segue:

$$w_{ij} = \frac{tf_i}{\max tf_j} \times log\frac{N}{d_{fi}}$$

Dividendo la *frequenza assoluta* tf_{ij} (del termine t_i nel documento d_j) per la *frequenza massima* di tutti i termini in d_j si ottiene come primo fattore la *frequenza relativa (TF)* del termine t_i nel documento d_j. Così facendo, si è in grado di pesare ogni termine sulla base della frequenza cui compare all'interno dell'intera collezione e di ogni specifico documento, di cui per ognuno se ne considera anche la lunghezza.

1.3 Strutture dati per la ricerca: gli indici inversi

Si è visto che il processo di indicizzazione ha lo scopo di analizzare l'archivio documentale per estrarre gli opportuni descrittori di ciascun documento. Questo processo produce ciò che vengono definiti indici, apposite strutture dati in grado di garantire risposte molto rapide alle richieste dell'utente. Gli indici sono costruiti *offline*, cioè prima che venga messa a disposizione la funzionalità di ricerca, perché la loro costruzione è un processo che può richiedere diverso tempo e risorse.

Di tutte le metodologie atte a memorizzare fisicamente indici *full text* verrà qui presentata solamente quella che sfrutta i cosiddetti **indici inversi**, molto veloci in termini di reperimento delle informazioni e largamente utilizzati su scala commerciale.

La dicitura "inverso" deriva dall'idea di base che costituisce questo tipo di indici. Si potrebbe ingenuamente pensare di memorizzare, per ogni documento, una lista di termini che vi compaiono, magari all'interno di una matrice statica di natura sparsa. Oltre che uno spreco di spazio, questa è anche una soluzione poco adatta alle nostre esigenze; **ogni ricerca è di fatto incentrata sui termini** ed in quanto tale è molto più facile reperire i documenti in cui compare un certo termine se la matrice precedentemente accennata viene memorizzata in maniera inversa: **per ogni termine viene salvata una lista (dinamica) di documenti in cui esso compare**.

In questo modo l'insieme di tutti i termini costituisce il **dizionario** dell'indice. Ogni termine del dizionario è associato alla propria frequenza globale e possiede un puntatore al *posting file*, che a sua volta contiene la lista di documenti all'interno dei quali il termine compare. Per ogni elemento della *posting list* viene memorizzato l'*identificatore* univoco del documento, la *frequenza* del termine al suo interno e tutte le relative *occorrenze*; per ciascuna occorrenza è possibile memorizzare anche la *posizione* del termine nel documento, per agevolare interrogazioni in cui compaiano operatori di prossimità fra termini.

Questo tipo di memorizzazione consente, in fase di ricerca, di accedere molto velocemente al dizionario e, successivamente, alla *lista di posting* associata al termine trovato nel dizionario. Qualora la query sia composta da termini multipli, ognuno di questi è sottoposto al medesimo trattamento e il risultato finale è costituito da un opportuno *merge* dei risultati parziali.

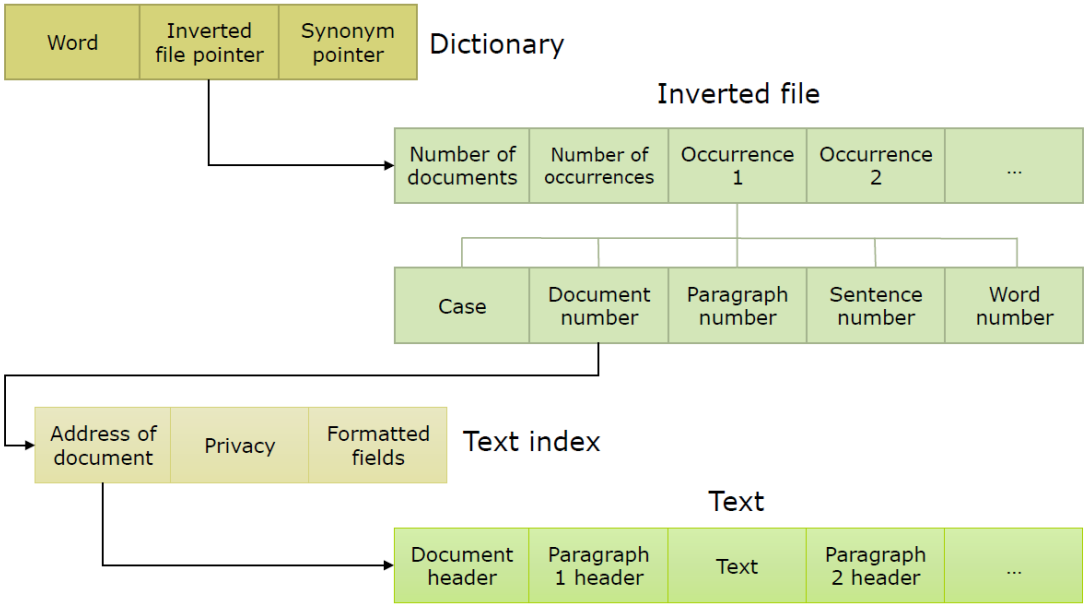

Figura 1.3: Struttura a File Inverted

1.4 Modalità di interrogazione

La fase d'interrogazione è l'unica parte di un IRS visibile all'utente, composta da un'interfaccia che gli consenta di esprimere delle **query**, elaborate dal sistema per produrre risultati coerenti con quanto richiesto. È importante che il linguaggio di interrogazione sia semplice e intuitivo per l'utente, ma abbastanza completo per garantire un certo grado di flessibilità.

Per quanto riguarda l'utilizzo dei motori di ricerca sul Web, una query è mediamente composta da 2-3 parole[5] e spesso alla prima ricerca ne seguono altre per raffinare il risultato; molto difficilmente si arriva a sottoporre interrogazioni complesse, che ad esempio si avvalgano di operatori logici[1].

Approfondendo le modalità con le quali le interrogazioni vengono gestite ed elaborate ci si accorge in maniera evidente che un sistema di Information Retrievial sia in realtà basato su un **modello matematico**, il quale fornisce innanzitutto una **descrizione formale dei documenti e delle query**, per consentire in seguito di mettere a confronto i due elementi. Il meccanismo di confronto è detto ***matching*** e ha lo scopo di **stimare la rilevanza** di ogni documento rispetto alla query; fornisce

[1] Alcuni esempi di interrogazioni complesse con Google e Bing: `http://advangle.com/`

pertanto solamente un'approssimazione dell'insieme di documenti che sono davvero rilevanti, producendo un **risultato incerto**, influenzato negativamente dall'ambiguità del contenuto dei documenti e delle richieste, nonché dalla soggettività di ciascun utente rispetto al concetto stesso di rilevanza.

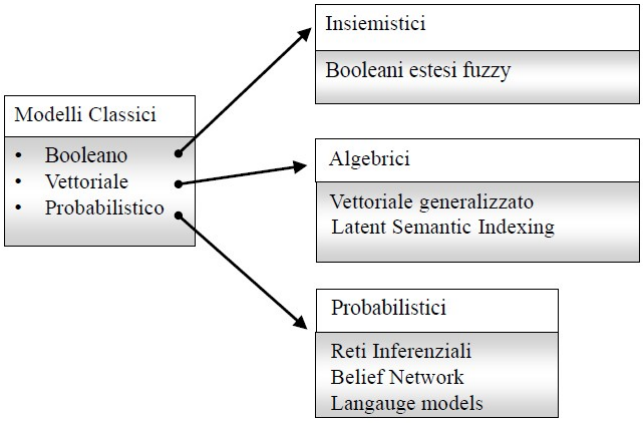

Figura 1.4: Modelli matematici per la rappresentazione di IRS

I modelli matematici disponibili per la rappresentazione di un IRS sono molteplici e in questa sede vengono brevemente descritti solamente quelli booleano e vettoriale, da cui si sviluppano modelli più complessi e comunemente utilizzati.

1.4.1 Modello booleano

Nel modello booleano la significatività dei termini è rappresentata da pesi binari w_{ij}, pari a `1` se il termine t_i compare nel documento d_j e pari a `0` se invece non compare. Il documento j-esimo è rappresentato dall'insieme dei termini tale per cui $w_{ij} = 1$.

Una query è formalmente definita come **espressione booleana sui termini** e definisce l'insieme di elementi da selezionare tramite l'utilizzo degli operatori logici, che vengono tradotti in operazioni insiemistiche sui documenti: gli operatori `AND`, `OR` e `NOT` corrispondono rispettivamente alle operazioni di intersezione, unione e differenza.

Con questo modello, per ogni termine che compare nella query viene recuperata nell'indice inverso la lista dei documenti associata a tale termine; poi sulle liste dei documenti vengono effettuate le opportune operazioni insiemistiche che danno origine ad un risultato composto dai soli documenti considerati rilevanti.

Nonostante il linguaggio di interrogazione booleano si rilevi particolarmente intuitivo e adatto a modellare sufficientemente le esigenze dell'utente, considerare la rilevanza come proprietà binaria (un documento è rilevante oppure non rilevante) non favorisce ricerche particolarmente utili. Infatti, questo tipo di modellazione ignora completamente la frequenza dei termini all'interno dei documenti e ciò rende impossibile presentare la lista dei risultati in ordine di rilevanza rispetto alla query.

1.4.2 Modello vettoriale

Il modello vettoriale nasce dall'esigenza di poter modellare la rilevanza come proprietà graduale e permettere l'ordinamento dei risultati di un'interrogazione. Questo modello poggia le propria fondamenta sull'algebra lineare: **documenti e query sono rappresentati come vettori** all'interno di uno spazio vettoriale n-dimensionale, dove n corrisponde al numero di termini che appartengono al dizionario degli indici. Si assume che i termini del dizionario siano linearmente indipendenti fra loro e si rappresenta il termine i-esimo come un versore: un vettore di `zeri` con il peso di valore `1` alla posizione i-esima.

$$t_i = [0, 0, ..., 1, ..., 0, 0]$$

Analogamente, anche documenti e query sono rappresentati come vettori di pesi, cui ogni posizione è associata all'i-esimo termine del dizionario. Ciò consente di effettuare un calcolo di **similarità** fra il vettore della query e ciascun vettore documento, calcolando di fatto il coseno dell'angolo α che disegnano i rispettivi vettori.

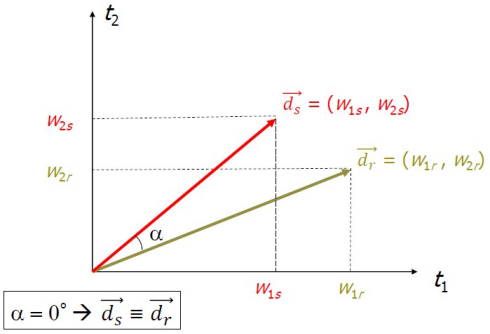

Figura 1.5: Rappresentazione di due documenti nello spazio vettoriale

In questo modo è possibile specificare, per ogni termine che appartiene a un documento, un peso arbitrario positivo che non sia necessariamente binario, per consentire al sistema di tenere in considerazione le frequenze di ogni termine nel documento e nella collezione.

La pesatura dei termini migliora la qualità del risultato è il confronto parziale fra vettori consente di reperire anche documenti che approssimino le condizioni della query senza soddisfarle completamente, garantendo la visualizzazione dei risultati in ordine decrescente di similarità, e quindi di rilevanza.

I limiti principali di questo modello consistono nel fatto che l'assunzione di indipendenza fra i termini è priva di fondamento e che il linguaggio di interrogazione si rivela decisamente poco intuitivo ed espressivo. Per sopperire alle mancanze del linguaggio è possibile combinare i modelli booleano e quello vettoriale affinché il primo si occupi di reperire i documenti rilevanti e il secondo abbia il solo compito di ordinarli, pur sapendo che in questo modo si perde la componente di parzialità sul confronto fra

query e documenti. Per operazioni di *matching* più complesse e complete è necessario approcciare modelli matematici più avanzati.

1.5 Valutazione di un sistema IR

"L'Information Retrieval è una disciplina che deve cercare di soddisfare **necessità informative espresse in modo vago e impreciso**, mediante le ambiguità del linguaggio naturale, e deve **confrontarle in modo approssimativo con le informazioni contenute in un documento**, espresse mediante il medesimo linguaggio naturale."
[Alan Smeaton, 1997]

In funzione dei fattori di incertezza con i quali un IRS è tenuto ad operare, si può facilmente dedurre che non sia affatto facile appurare l'affidabilità dei risultati prodotti dal sistema a seguito di una ricerca. Risulta infatti piuttosto complicato riuscire a discriminare se un documento reperito sia effettivamente utile per le esigenze informative dell'utente o se rappresenti invece un elemento di poco conto.

Il concetto di rilevanza è fondamentale per misurare l'efficacia di un sistema di Information Retrieval. Tale misura è principalmente soggetta a due valutazioni:

- **Precisione** (precision), la percentuale di documenti reperiti che risultano effettivamente rilevanti per una certa query, rispetto alla totalità dei documenti reperiti come risultato della query stessa.
- **Richiamo** (recall), la percentuale di documenti rilevanti che vengono effettivamente reperiti in seguito a una query, rispetto a tutti quelli rilevanti (per tale query) presenti nella collezione.

$$Precisione = \frac{|Rilevanti \cap Reperiti|}{|Reperiti|}$$

$$Richiamo = \frac{|Rilevanti \cap Reperiti|}{|Rilevanti|}$$

È immediato notare che precisione e richiamo siano proprietà effettivamente valutabili solo a patto che si conosca a priori quali sono i documenti a propria disposizione, ed in particolare quali sono quelli rilevanti rispetto alla query che si sta cercando di valutare. Tale valutazione è altamente soggettiva e influenzata anche dall'ambiguità delle richieste sottoposte al sistema dall'utente. Un ulteriore elemento che influisce su questo tipo di valutazione può essere l'incompletezza nella rappresentazione dei documenti, che potrebbe rivelarsi poco adatta per rispondere ad alcune specifiche esigenze.

Oltre alla valutazione d'efficacia, che si basa principalmente sulla bontà della rappresentazione formale dell'archivio e dell'algoritmo di matching utilizzato, un IRS è soggetto ad ulteriori criteri di valutazione più facilmente misurabili:

- Interazione con il sistema, cioè quali sono le **funzionalità** che mette disposizione dell'utente e il livello di facilità ed **intuitività** con le quali siano utilizzabili.
- Efficienza, cioè i **tempi** di risposta alle richieste dell'utente e lo **spazio** occupato dagli indici su disco.

Sulla base di quanto detto, emerge chiaramente che la valutazione di un sistema di reperimento delle informazioni sia un compito molto complesso che richiede un certo livello di esperienza.

Capitolo 2

Introduzione ad Apache Solr

Come visto nel capitolo precedente, realizzare un sistema di reperimento delle informazioni richiede che vangano affrontati numerosi problemi, talvolta anche molto complessi. Per questo motivo, qualora vi sia la necessità di progettare e sviluppare un proprio motore di ricerca, è possibile contare su alcune soluzioni già esistenti. Fra le più conosciute vi è **Apache Lucene**[1] una libreria open source scritta in Java che offre funzioni di ricerca *full text*. Lucene si occupa personalmente della gestione fisica degli indici inversi e mette a disposizione apposite funzionalità per schematizzare le informazioni a disposizione, sulle quali condurre un'ampia gamma di interrogazioni.

Nonostante esistano porting della libreria Java in numerosi altri linguaggi di programmazione, può rivelarsi tediante sviluppare motori di ricerca complessi con il solo utilizzo di Lucene. Per questa ragione nasce **Apache Solr**[2], una piattaforma di ***enterprise search*** costruita su Lucene stesso che offre funzionalità di alto livello, il cui utilizzo si rivela adatto all'interno di sistemi dalle dimensioni importanti con ampie esigenze informative. Solr si occupa di organizzare e gestire alcuni aspetti che non sono di facile gestione con Lucene e permette di configurare il sistema in maniera più rapida ed efficace.

2.1 Enterprise Search

Il termine *enterprise search* è utilizzato per descrivere finalità di ricerca progettate per il reperimento delle informazioni all'interno di contesti specifici e di natura aziendale. Si contrappone al concetto di *web search*, che opera con le risorse pubblicamente disponibili nel World Wide Web.

Nonostante la vera e propria funzionalità di ricerca possa di fatto essere resa disponibile anche al di fuori delle mura aziendali, la base documentale con cui ha a che fare un software di *enterprise search* appartiene al dominio dell'azienda ed è frutto dell'aggregazione di molteplici sorgenti. In funzione di questa varietà delle fonti, è ricorrente la necessità di gestire documenti con significati e rappresentazioni molto

[1] `http://lucene.apache.org/`
[2] `https://lucene.apache.org/solr/`

differenti fra loro. L'obiettivo consiste nel fornire l'opportunità all'utente di effettuare ricerche efficaci ed intuitive sulla base documentale, senza che egli si renda conto dell'eterogeneità delle informazioni a propria disposizione.

Un software di *enterprise search* elabora e mette a disposizione per la ricerca dati che prevengono non solo da uno o più database, quindi ben strutturati, ma anche memorizzati e scambiati tramite qualsiasi altro mezzo aziendale che fornisca l'accesso a molteplici tipi di documenti, comunicazioni, file di log, calendari, mappe, etc., le quali costituiscono informazioni non strutturate o semi-strutturate all'interno delle quali è difficile orientarsi se non le si elabora nella maniera adeguata.

L'utente che deve utilizzare un software aziendale per finalità di ricerca:

- Conosce il dominio applicativo d'utilizzo;
- Vuole reperire velocemente informazioni dall'enorme quantità di dati;
- Vuole poter ritrovare facilmente elementi di cui conosce già l'esistenza;
- Vuole che la ricerca sia proficua ed eventualmente raffinarne i risultati;
- Vuole che il sistema conosca il contesto applicativo e sia quindi in grado di effettuare le dovute inferenze;
- Vuole minimizzare i propri sforzi.

Tali esigenze corrispondo agli obiettivi da tenere a mente in fase di progettazione di un motore di ricerca e costituiscono i punti di riferimento che hanno accompagnato la realizzazione del mio progetto di *enterprise search* e, pertanto, guidano la stesura di questa tesi. Prima di affrontare il caso di studio analizzato vorrei dedicare un capitolo ad Apache Solr, per spiegarne il funzionamento di base e comprendere quali sono le soluzioni che propone per affrontare problemi ricorrenti di Information Retrieval.

2.2 Cos'è Apache Solr

Apache Solr è una piattaforma di *enterprise search* che nasce nel 2004, il cui progetto si sviluppa nel corso degli anni al fianco di Lucene, una libreria Java rilasciata sotto licenza Apache che implementa soluzioni di Information Retrieval, ed in particolare l'indicizzazione e la ricerca *full text*.

Come Lucene, anche Solr è open source, scritto in Java e può vantare un'ampia comunità di persone che lo utilizzano e contribuiscono al suo sviluppo. Nasce con lo scopo di ampliare le funzionalità di Lucene per renderlo più scalabile e versatile, ma anche più facilmente utilizzabile grazie ad interazioni ad alto livello. Utilizzare una piattaforma del calibro di Solr consente di gestire il processo di ricerca a 360 gradi, totalmente configurabile sulla base delle proprie esigenze funzionali e architetturali.

Ciò ne giustifica l'utilizzo rispetto ad altri sistemi che pur offrendo funzionalità di *full text search*, come ad esempio alcuni DBMS[3], non mettono disposizione una così ampia gamma di personalizzazioni e funzionalità.

2.2.1 Architettura

Così come alcuni DBMS NoSQL, anche Solr sfrutta una memorizzazione logica di tipo *document-based*, che consiste nel considerare l'unità informativa come un **documento** composto da più **campi** e rappresentato sotto forma di JSON. Tale rappresentazione appare molto comoda e adatta alla modellazione di qualsiasi tipo di informazione, in maniera versatile e facilmente gestibile.

Solr offre ed espande tutte le funzionalità di Lucene ma è di fatto un applicativo **server**, in grado di lavorare sia in modalità *stand-alone* sia come sistema distribuito in rete, prendendo in tal caso il nome di **SolrCloud**. Questa caratteristica lo rende particolarmente adatto ad applicazioni che necessitano di un'architettura ampiamente scalabile e di una forte tolleranza agli errori. Se un normale server Solr lavora con indici inversi Lucene, ognuno dei quali prende il nome di ***core***, con SolrCloud si lavora in ***collection*** che costituiscono lo stesso tipo di struttura dati, ma con la possibilità di effettuare repliche e shard su molteplici istanze di Solr. Una **replica** è un'intera copia degli indici, che vengono duplicati su diversi server; per **shard** si intende invece la possibilità di suddividere un unico indice su molteplici server per distribuire il carico di lavoro. Le due tecniche di scalabilità dei dati sono utilizzabili contemporaneamente, per favorire al contempo alte prestazioni (shard) e resistenza ai guasti (replica).

2.2.2 Utilizzo

Solr è un applicativo Java da eseguire in modalità server. Fornisce una comoda interfaccia di amministrazione accessibile tramite browser, che mette a disposizione gran parte delle principali funzionalità della piattaforma, altrimenti configurabile tramite XML e interrogabile tramite API REST, per loro natura utilizzabili da qualsiasi programma e linguaggio di programmazione in grado di inviare, ricevere ed interpretare richieste HTTP. In questo risiede sostanzialmente la portabilità di Solr rispetto a Lucene, dato che risulta facilmente configurabile ed interrogabile da qualsiasi client connesso ad internet, senza che siano necessari particolari requisiti di installazione.

Ad esempio, per chiedere a Solr di indicizzare un documento è sufficiente inviarlo al server con una richiesta POST, includendolo nel body in formato JSON/XML/CSV o come file da cui estrarre il contenuto. In alternativa è possibile affidarsi a sistemi di

[3]Ho personalmente effettuato un confronto fra la funzionalità di ricerca testuale di Solr e del DBMS NoSQL document-based MongoDB ed è emerso che quest'ultimo, nonostante riesca ad equiparare Solr riguardo molte opportunità di ricerca e tempi di esecuzione, si rivela carente nel momento in cui si hanno esigenze di ricerca più avanzate come ad esempio un'analisi linguistica completa di sinonimi e tesauri oppure funzionalità fortemente tipiche di enterprise search systems: estrazione del testo da un file, Highlighting, Spellcheck, MoreLikeThis, etc.

importazione da database relazionali o sorgenti HTTP di varia natura (RSS, e-mail repository, etc.).

2.2.3 Query

Per sottoporre una query al sistema è necessario inviare una richiesta GET, specificando i parametri di ricerca nella *querystring*. Solr gode di tutte le funzionalità di Lucene, proponendone di nuove. Il risultato di una ricerca, per default, presenta come risultato i documenti reperiti in ordine di rilevanza decrescente, che in Solr prende il nome di `score` (punteggio).

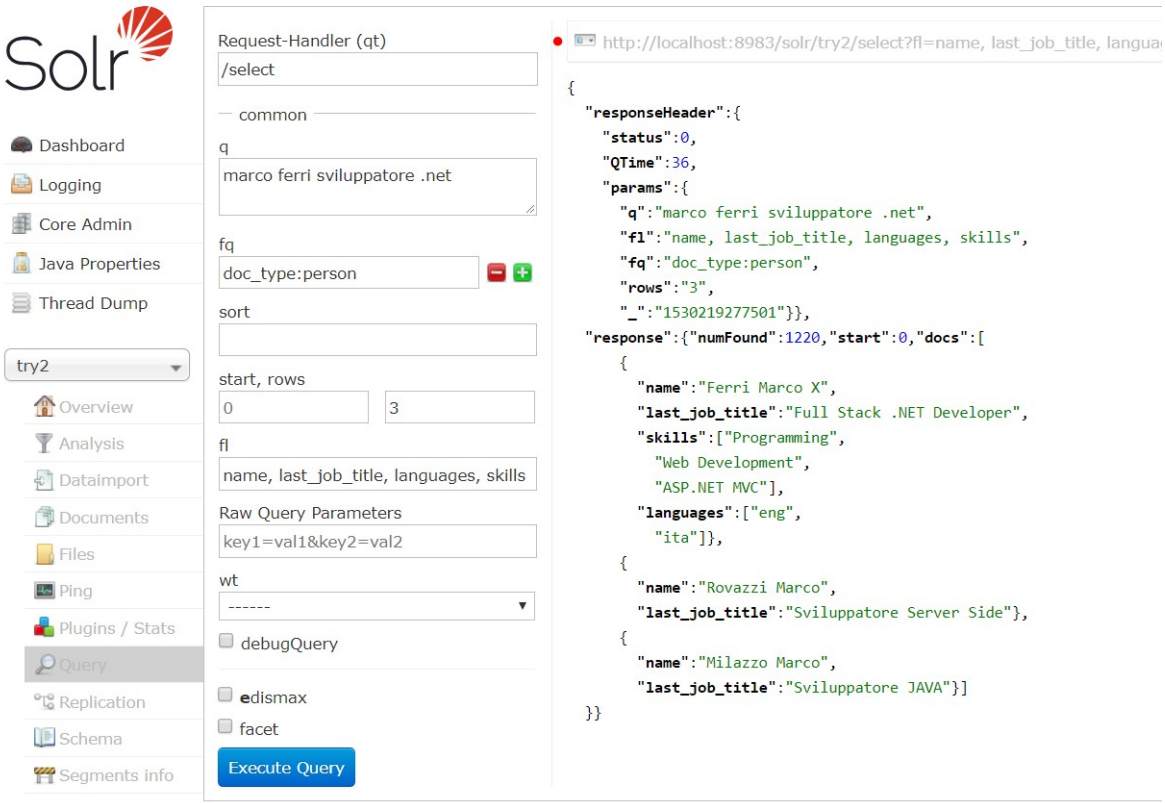

Figura 2.1: Esempio di ricerca con Solr

Nell'immagine 2.1 è possibile osservare un esempio di interrogazione, effettuata attraverso l'interfaccia web di amministrazione[4], in cui vengono mostrati un limitato numero di parametri con cui è possibile effettuare la ricerca[5]. Dallo screenshot si evince che il *core* su cui si esegue la query prende il nome di *try2* (lo si può notare sulla sinistra) e che i parametri di ricerca utilizzati sono i seguenti:

[4] `https://lucene.apache.org/solr/guide/7_4/overview-of-the-solr-admin-ui.html`
[5] `https://lucene.apache.org/solr/guide/7_4/common-query-parameters.html`

- "`q`" indica a Solr di cercare le parole *marco*, *ferri*, *sviluppatore* e *.net*. Questo parametro, se non diversamente specificato, utilizza il *query parser* standard di Lucene che cerca ciascun termine in `OR` e considera per la ricerca un **campo di default** (definito nei file di configurazione). Con gli appositi parametri è possibile sfruttare altri *parser*, ognuno dei quali fornisce funzionalità specifiche, e indicare diversi campi su cui effettuare la ricerca.
- "`fq`", letteralmente *filter query*, consiste in un filtro che non influisce sullo `score` dei documenti e pertanto sul calcolo della rilevanza. In questo caso viene specificato `doc_type`:*person*, che chiede a Solr di reperire solamente i documenti con il valore *person* nel campo `doc_type`.
- "`rows`", pari a `3`, chiede di restituire soltanto i primi tre risultati.
- "`fl`", che significa *field list*, specifica quali campi (o **field**) dei documenti trovati devono essere inclusi nel risultato.

La query dell'esempio è il risultato di una richiesta al seguente url `http://localhost:8983/solr/try2/select?fl=name,%20last_job_title,languages,skills&fq=doc_type:person&q=marco%20ferri%20sviluppatore%20.net&rows=3` e produce una risposta che vede `1220` risultati trovati (`numFounds`), ordinati per rilevanza decrescente, di cui vengono mostrati in formato JSON solamente i primi tre. È importante notare che non tutti i documenti trovati contengono la totalità delle parole cercate, vista la ricerca in `OR`, ma che il primo risultato le contiene tutte, seppur il termine *sviluppatore* appaia in realtà con il suo sinonimo *developer*.

Fra le altre funzionalità disponibili in fase di interrogazione vi è la possibilità di effettuare dei ***boosting***, cioè di specificare che alcuni campi o termini della query vengano considerati più o meno importanti di altri per il calcolo della rilevanza, influenzando così l'ordine di presentazione dei risultati.

Un ulteriore strumento molto utile consiste nelle **ricerche geo-spaziali**: Solr è in grado, dato un punto geografico rappresentato da latitudine e longitudine, di reperire tutti i documenti che specificano una posizione geografica (espressa anch'essa in coordinate) entro una certa distanza dal punto di partenza.

2.2.4 Funzionalità

Oltre alle classiche funzionalità di ricerca *full text*, Solr ne mette a disposizione molte altre pensate per applicazioni che fanno della ricerca il proprio punto cardine. Alcune di queste vengono ora presentate in maniera molto rapida e verranno approfondite meglio nel capitolo successivo, per comprendere in che modo siano state utilizzate all'interno del motore di ricerca da me progettato e realizzato.

Faceting e Grouping

Il *faceting*, difficilmente traducibile in italiano se non con il termine "ricerca sfaccettata", è un tipo di ricerca ampiamente conosciuta ed utilizzata sui principali portali di compra-vendita e non solo. Consiste in un insieme di **filtri su specifici campi**, che dispongono solitamente di una sorta di "**conteggio**" atto ad indicare quanti dei documenti risultanti dalla query soddisfano uno specifico valore del filtro. Nell'immagine 2.2, un esempio - riguardante la ricerca di un hard disk - preso da un noto sito di e-commerce potrà sicuramente chiarire qualsiasi dubbio circa il significato del termine *faceting*.

Figura 2.2: Faceting

Questa funzionalità è, insieme al **raggruppamento per campo**, una delle principali caratteristiche di un sistema di *enterprise search*, perché estremamente utile all'utente per raffinare i risultati della ricerca in essere o semplicemente visualizzare in maniera rapida quali sono le "categorie" cui i risultati reperiti appartengono.

Highlighting

L'*highlighting* è la tecnica che consente di **mettere in evidenza i termini** cercati (e trovati) all'interno di un documento, solitamente mettendoli graficamente in risalto. È tipico degli snippet di Google ed è uno strumento utilissimo per gli utenti, specie per quando si sta effettuando una ricerca all'interno di testi molto lunghi o non si riesce a comprendere il motivo per cui un determinato documento sia stato giudicato come rilevante da parte del sistema.

Spellcheck & Suggester

Chiunque si sarà imbattuto nella scritta *"Forse cercavi..."*, tipica dei più comuni motori di ricerca online. In Solr questa funzione prende il nome di *spellcheck* e può essere utile per **correggere** (anche in maniera automatica) le parole specificate dall'utente nella query. In combinazione con il *suggester*, che fornisce **autocompletamento** delle query, ha lo scopo di ridurre ricerche fallimentari causate da errori di battitura.

MoreLikeThis

Ancora una volta scenario tipico dei motori di ricerca sul Web, che si tratti di siti di compravendita o ricerca per immagini, questa funzionalità permette all'utente di dare in pasto a Solr un certo documento affinché il sistema si occupi di cercare nell'indice tutti i **documenti simili** ad esso. Funzione molto usata per fini commerciali.

2.3 Modellazione dei dati con Solr

Come precedentemente accennato, l'entità base che costituisce informazione in Solr è il documento, rappresentato sotto forma di JSON. Chiunque sia familiare a questo tipo di formato comprende istantaneamente l'importanza che ha assunto nell'ultimo ventennio per lo scambio di informazioni online e più recentemente anche come modello di memorizzazione nei più recenti DBMS. Un JSON è perfettamente adatto a rappresentare il *record* di un database relazionale o l'*oggetto* di un linguaggio di programmazione *object-oriented* e data la sua versatilità si rivela adatto a modellare le informazioni eterogenee con cui una piattaforma di *enterprise search* ha costantemente a che fare.

```
{
    "name": "Mario",
    "surname": "Rossi",
    "active": true,
    "favoriteNumber": 42,
    "birthday": {
        "day": 1,
        "month": 1,
        "year": 2000
    },
    "languages": [ "it", "en" ]
}
```

Figura 2.3: JSON

2.3.1 Schema e tipizzazione

Essendo la rappresentazione JSON costituita da un insieme di proprietà `nome:"valore"` (figura 2.3), è chiaro che tali proprietà devono essere opportunamente memorizzate da Solr all'interno degli indici inversi, che per ereditarietà da Lucene sono in grado di suddividere le informazioni di ogni documento in specifici **campi** (o *field*).

È possibile, per ogni *core* di Solr, definire uno **schema** che descrive la struttura dei documenti all'interno dell'indice, oppure affidarsi ad un approccio ***schemaless***. Nel primo caso, il server sarà in grado di indicizzare esclusivamente documenti che contengono i soli campi dichiarati nello schema (o alcuni di essi) e rifiuterà tutto ciò che non è conforme. Al contrario, se si sceglierà di sfruttare la tecnica *schemaless*, qualsiasi tipo di documento potrà essere *postato*, delegando a Solr la creazione di campi finora sconosciuti. Questo secondo approccio è molto utile se non si conosce la composizione dei documenti che faranno parte dell'indice o se lo schema è frequentemente soggetto ad aggiornamenti, ma è generalmente sconsigliato per software in ambiente di produzione.

Ciascun campo che appartiene ad un documento Solr è **fortemente tipizzato**, che esso sia esplicitamente dichiarato nello schema o inferito automaticamente dal sistema quando ne crea di nuovi. Tutti i tipi di campo descritti nello schema sono mappati con opportune classi Java che ne definiscono il comportamento di base. Poi, per ogni tipo, è possibile specificare una **catena di analisi testuale** personalizzata, che indica quali devono essere le operazioni da eseguire sul contenuto dei campi di un certo tipo, prima che il testo dei documenti (e più precisamente i termini) vengano indicizzati da Solr.

La tipizzazione dei campi consente di definire comportamenti altamente specifici per ognuno di essi e permette all'amministratore Solr di scegliere quale dev'essere la granularità di ricerca per ciascun tipo definito nello schema. Per esempio, può capitare che alcune informazioni debbano essere trattate come un unico termine seppur appaiono

intervallate da caratteri separatori (e.g., IBAN bancari, codici fiscali, etc.), o che alcuni campi necessitino di analisi linguistiche personalizzate.

Con Solr l'analisi testuale dei processi di indicizzazione e interrogazione è ben distinta, ma è essenziale che i due trattamenti siano compatibili affinché l'algoritmo di *matching* sia in grado di confrontare i termini dei documenti e quelli della query in maniera consistente.

2.3.2 Content Extraction Library

Si è precedentemente accennato all'eterogeneità delle fonti che costituiscono la base documentale di un sistema di reperimento delle informazioni, in particolare per quanto riguarda software di enterprise search.

Se le tabelle di un database sono facilmente esportabili in JSON, non è altrettanto vero per quanto riguarda i file memorizzati sul disco fisso. Come è possibile immaginare, non tutti i file sono adatti ad un'estrazione diretta del testo, perché la maggior parte di essi sono in un formato binario (e.g., doc, pdf, xls, ppt, etc.) oppure contengono dati "sporchi", come nel caso dei file XML, HTML o delle email, di cui solitamente non si vuole indicizzare direttamente il testo che compone tag o intestazioni. Solr ci viene in aiuto, mettendo a disposizione la propria *Content Extraction Library* (Solr CELL), che sfrutta il progetto **Apache Tika**[6] per estrarre testo e metadati da un qualsiasi file.

Il contenuto testuale estratto da un file sarà assegnato ad un campo scelto arbitrariamente, soggetto alla catena di analisi testuale prevista per quel campo. L'analisi dei file rappresenta una delle operazioni più stressanti per il sistema, per via dell'enorme quantità di *token* solitamente estratti dal file stesso, che dovranno essere singolarmente analizzati affinché vadano a costituire i termini indice.

Proprio a causa di questa numerosità, è opportuno decidere in fase di progettazione se è opportuno, oltre che indicizzare, anche "memorizzare" il contenuto estratto dai file. Un campo indicizzato (`indexed`) ma non memorizzato (`not stored`) è disponibile per la ricerca e il meccanismo di *matching* fra query e documenti ne terrà conto, ma non verrà mostrato fra i risultati della ricerca stessa. In altre parole, se un certo file contiene un termine della query, il documento che rappresenta tale file verrà mostrato fra i risultati della ricerca, ma privo del campo `not stored` su cui è stata realmente effettuata la ricerca. Scegliere di non memorizzare il contenuto estratto da un file può risparmiare moltissimo spazio sulla memoria di massa del server Solr, ma introduce alcune limitazioni che si vedranno più avanti.

[6] `https://tika.apache.org/`

2.4 Algoritmo di matching basato su Okapi BM25

A conclusione del capitolo, ci si ricollega ai modelli matematici che costituiscono un sistema di reperimento delle informazioni cui si faceva riferimento nel capitolo 1.4, per fornire una breve spiegazione di quello che è l'attuale[7] algoritmo di *matching* di default utilizzato da parte di Solr.

Il suo nome è Okapi BM25 e vede la luce nel lontano 1994.
Si tratta di un modello di tipo **probabilistico**, che considera il concetto di rilevanza come la probabilità che un certo documento possa o meno essere considerato rilevante da parte dell'utente. Nello specifico, Okapi BM25 basa il proprio funzionamento su un'evoluzione del modello *TF*IDF (term frequency * inverse document frequency)*, funzione di pesatura dei termini presentata nella sezione 1.2.2.

All'algoritmo di base vengono apportate diverse modifiche e miglioramenti, il cui elemento chiave è la nuova modalità di calcolo della *inverse document frequency*, che ora costituisce la componente probabilistica di BM25 . Anche la trattazione della *term frequency* e della *lunghezza del documento* sono state riviste, affinché la prima si riveli meno incisiva nel calcolo della rilevanza mentre la seconda possa essere soggetta ad un maggiore controllo. Approfondimenti ulteriori vanno oltre lo scopo di questa tesi e per maggiori dettagli si rimanda ad un articolo in bibliografia[6] che spiega questi concetti in maniera molto chiara ed evidenzia che, seppur BM25 sia un modello con molte potenzialità, non è detto si riveli adatto a qualsiasi situazione.

$$score(D,Q) = \sum_{i=1}^{n} IDF(q_i) \times \frac{f(q_i,D) \times (k_1+1)}{f(q_i,D) + k_1 \times (1 - b + b \times \frac{|D|}{avgdl})}$$

Il risultato consiste di questa formula abbastanza complessa - all'interno della quale compaiono alcuni parametri di normalizzazione - che va sostanzialmente a considerare l'insieme delle caratteristiche presentate già nel primo capitolo di questa tesi. Ciò va a dimostrazione del fatto che un buon algoritmo di *matching*, per quanto complesso possa rivelarsi, consiste principalmente nel trovare la giusta combinazione dei concetti di base dell'Information Retrieval precedentemente presentati.

[7] Nel momento in cui scrivo questa tesi, e quindi con riferimento alla versione 7.4 di Solr

Capitolo 3

Definizione di un sistema di Enterprise Search per le Risorse Umane

Lo studio condotto sui temi di Information Retrieval ha trovato finalizzazione nello sviluppo di un motore di ricerca che potesse integrare il proprio funzionamento all'interno di un prodotto aziendale già ben funzionante ed utilizzato. Lo stage di durata trimestrale si è svolto presso l'azienda Capitolo Quinto srl [1] e il software oggetto del mio studio è Arkivium Recruiting[2], un applicativo di natura gestionale che rientra nella categoria di *Customer Relationship Management* (CRM). Più specificatamente, il software si occupa della gestione del processo di ricerca e selezione del personale e assume il ruolo di *Applicant Tracking System* (ATS), indicato per il settore delle risorse umane.

Il lavoro di progettazione e realizzazione del sistema è stato condotto con una metodologia di sviluppo incrementale, che a partire dall'analisi delle esigenze è stato in grado di produrre di volta in volta dei risultati intermedi, che affrontassero e risolvessero i problemi riscontrati in maniera evolutiva. È seguendo questo tipo di approccio che vorrei accompagnare il lettore attraverso l'intero processo decisionale che ha dato luce ad un prototipo del motore di ricerca, il quale sarà soggetto nei mesi seguenti ad ulteriori miglioramenti che si concluderanno con il rilascio del prodotto sul mercato.

3.1 Analisi dei requisiti

Arkivium Recruiting è utilizzato da diversi tipi di organizzazioni (agenzie per il lavoro, società di ricerca e selezione, reparti aziendali delle risorse umane) per coordinare e agevolare i processi di ricerca e selezione del personale. Il sistema gestisce numerosi dati, tipici di questo ambito, che vedono la "figura professionale" al centro dell'attenzione, con le proprie competenze ed esperienze pregresse che la rendono parte di una rete costituita da numerose aziende, persone ed enti di formazione. Tali informazioni possono essere a disposizione dei *recruiter* in forma strutturata o, come molto più spesso accade in questo mondo, nascondersi all'interno del *curriculum vitae* di una persona insieme ad

[1] `http://www.capitoloquinto.com/`
[2] `http://www.arkivium.com/`

una miriade di informazioni che di volta in volta devono essere manualmente consultate ed interpretate dai selezionatori.

In questo contesto nasce l'esigenza di poter effettuare delle ricerche in maniera molto efficiente all'interno di informazioni solo parzialmente strutturate, che provengono in parte da un database ed in parte dai diversi tipi di file associati ad una persona o azienda, fra cui appunto i *curriculum vitae*.

Tale esigenza era in Arkivium già in parte soddisfatta dal precedente motore di ricerca, implementato con Apache Lucene, che permetteva di effettuare ricerche *full text* abbastanza basilari. La necessità di sottoporre interrogazioni più intuitive e di presentare risultati di maggiore utilità per l'utente, nonché di contare su di un sistema maggiormente scalabile, ha inevitabilmente richiesto il passaggio da Lucene a qualcosa di più versatile e facilmente utilizzabile. La scelta è ricaduta su Apache Solr, in grado di offrire numerose opportunità ad elevate prestazioni senza la necessità di stravolgere in alcun modo l'ecosistema nel quale Arkivium abitualmente opera.

Nello specifico, quest'ultimo consiste di un'applicazione web realizzata attraverso il paradigma MVC del Microsoft .NET Framework[3], all'interno della quale si è inserito il codice per la gestione delle informazioni da indicizzare ed il loro reperimento in fase di ricerca; a tal proposito, per l'utilizzo di Solr si sono utilizzate direttamente le Web API esposte dal server.

3.2 Costruzione dell'archivio documentale

Il primo quesito da affrontare, quando si progetta un motore di ricerca, è **ciò che si vuole cercare**. Una prima soluzione a cui si potrebbe pensare consiste nella volontà di cercare fin da subito nella totalità delle fonti d'informazione a propria disposizione; tuttavia, dal momento che si ha a che fare con molti dati dai significati più disparati, questo tipo di approccio non è immediato e non è scontato possa rivelarsi una soluzione adatta, perché ciascuna informazione deve influire in maniera adeguata sui risultati della ricerca e deve successivamente essere presentata all'utente nel modo corretto.

Nel caso in esame, i dati trattati dall'applicazione sono vari e tipici di un *Customer Relationship Management*. Non tutti sono oggetto di questo studio, che si concentra solamente sui pilastri fondamentali attorno ai quali è costruita l'intera base informativa: **Persone**, **Aziende**, **Progetti** e **Documenti**. L'obiettivo è quello di costruire un motore di ricerca in grado di cercare contemporaneamente documenti che rappresentano molteplici tipi di informazioni, senza che l'utente si accorga di alcuna limitazione tecnica che possa derivarne ed in modo che i risultati gli vengano presentati in maniera intuitiva e funzionale.

Il principale sistema di memorizzazione di Arkivium è un database di tipo relazionale, che sfrutta numerose tabelle e associazioni per rappresentare le informazioni a disposi-

[3] `https://www.microsoft.com/net/apps/web`

zione dei *recruiter*. Le principali entità coinvolte sono `Persone` e `Aziende`, a propria volta legate a molteplici `Progetti`. Ciascun elemento può essere inoltre collegato ad uno o più `Documenti`, veri e propri file salvati su memoria di massa e il cui riferimento lo si ottiene consultando il database stesso (`NOTA`: d'ora in poi, il tipo di entità del database che rappresenta un file sarà sempre scritta come `Documento`, per evitare si possa confondere con il concetto di "documento Solr" precedentemente descritto, che invece rappresenta la metodologia di rappresentazione delle informazioni in Solr).

Affinché le informazioni strutturate presenti nel database (`Persone`, `Aziende` e `Progetti`) e quelle non strutturate costituite dai file (`Documenti`) possano essere gestite tramite Solr, è necessario convertirle in un formato a questo comprensibile; ciascuna entità deve essere adeguatamente rimodellata sfruttando le informazioni presenti nel database, spesso attingendo a molteplici tabelle o al file system.

Prima di modellare le conoscenze a disposizione partendo dallo schema *Entity-Relationship* che rappresenta il database, è necessario decidere in che modo si vogliono utilizzare gli indici Solr per memorizzare e reperire i documenti a disposizione.

3.2.1 Utilizzo degli indici

Lavorando con Solr, e più precisamente con Lucene che ne sta alla base, è fondamentale decidere in che modo le informazioni devono essere suddivise e memorizzate negli opportuni indici. I fattori da tenere in considerazione sono:

- necessità informative, in particolare ciò che si vuole ottenere da una singola query;
- accessibilità delle informazioni, cioè chi deve avere accesso ai dati degli indici.

Suddivisione delle informazioni

Il precedente motore di ricerca integrato in Arkivium sfruttava due indici, di cui uno per indicizzare le `Persone` e uno per i `Documenti`. Questa soluzione prevedeva la necessità di effettuare operazioni differenti sui due, in particolare per quanto riguarda il processo di interrogazione, che poteva agire solamente su un indice per volta con la necessità di sottoporre due query distinte nel caso in cui si volesse ottenere un risultato misto.

Progettando il nuovo sistema ci si è accorti che il modello multi-indice non fosse particolarmente comodo e, in funzione delle necessità informative dell'utente - che potrebbe avere la necessità di cercare, con una singola query, sull'intero contenuto indicizzato -, si è deciso di optare per una soluzione a singolo indice. La natura malleabile di un JSON e quindi di un documento Solr consentono infatti di rappresentare oggetti ben distinti all'interno del medesimo indice, offrendo la possibilità di sfruttare proprietà comuni a più di un oggetto pur personalizzando ciascuno di essi con campi specifici. Esemplificando, è chiaro che una `Persona` e un `Documento` siano entrambi caratterizzati da un *nome*, ma il primo abbia bisogno di memorizzare anche una *data di nascita* mentre del secondo si predilige salvare l'*estensione*.

Questo tipo di approccio consente una gestione semplificata dell'intero processo di indicizzazione e ricerca, ma costringe ad aggiungere per ciascun documento un campo che ne identifichi il *tipo*.

Gestione degli accessi

La seconda considerazione cui si è andati incontro durante il processo di progettazione è legata alla separazione degli accessi fra clienti. Per "cliente" si intende l'azienda che acquista Arkivium per la gestione dei propri processi di ricerca e selezione. É chiaro che ogni cliente detiene un database con i propri dati ed è essenziale che questi rimangano di natura confidenziale.

Anche questa volta, grazie all'adattabilità di un documento Solr, sarebbe possibile decidere di mettere a disposizione un singolo indice per tutti i clienti di Capitolo Quinto. Ciò prevedrebbe l'inserimento di un ulteriore campo che, per ogni documento, identifichi il cliente a cui appartiene. In fase di interrogazione, l'applicazione dovrebbe fare in modo che ogni cliente possa accedere solamente ai propri dati.

Nonostante la soluzione sia attuabile, è inevitabilmente poco sicura se non adeguatamente gestita, perché potrebbe facilmente portare ad accidentali falle di sicurezza che causerebbero una spiacevole diffusione di informazioni private. Inoltre, così facendo l'indice risultante assumerebbe dimensioni considerevoli e potrebbe essere causa di complicazioni sia a livello prestazionale sia nel caso si presenti la necessità di ricostruire l'indice, un processo di cui si parlerà nella sezione 3.3 dedicata all'indicizzazione.

Soluzione adottata

Ricapitolando, si è scelto di sfruttare un singolo indice per ciascuna azienda che utilizza Arkivium. Ciò consente di soddisfare contemporaneamente i requisiti utente e la necessità di interfacciarsi facilmente con un sistema sicuro. In Solr, ogni indice corrisponde a un *core* del server (o *collection*, se si sta utilizzando SolrCloud), che è caratterizzato da un proprio schema e risponde ad un certo *url*. Ogni cliente utilizzerà l'*url* dedicatogli, il cui accesso in futuro potrebbe essere protetto da autenticazione.

3.2.2 Dal modello Entity-Relationship al documento JSON

Riprendendo quanto detto poc'anzi, Arkivium memorizza gran parte delle proprie informazioni all'interno di un database relazionale, che consente di trattare dati ben strutturati e interconnessi. Affinché possano essere utilizzati da Solr, devono essere convertiti in un formato che quest'ultimo sia in grado di interpretare: CSV, XML o JSON. Per semplicità e leggibilità si è scelto di usare il JSON, che consente una modellazione intuitiva e, per certi versi, anche più adatta rispetto ad un approccio relazionale: alcune relazioni `N:N` sono infatti più facilmente rappresentabili con un JSON piuttosto che tramite il formato tabellare di un database SQL.

Gli elementi più importanti che fanno parte della base di dati informativa di Arkivium sono rappresentati nella figura 3.1, che è solamente un piccolo estratto del complesso diagramma ER che descrive il database. Pur essendo focalizzato solamente sulle tabelle principali `Person`, `Company`, `Attachment` e `Project` (in azzurro nel modello), lo schema concettuale appare piuttosto intricato a causa delle numerose relazioni che intercorrono fra le diverse entità.

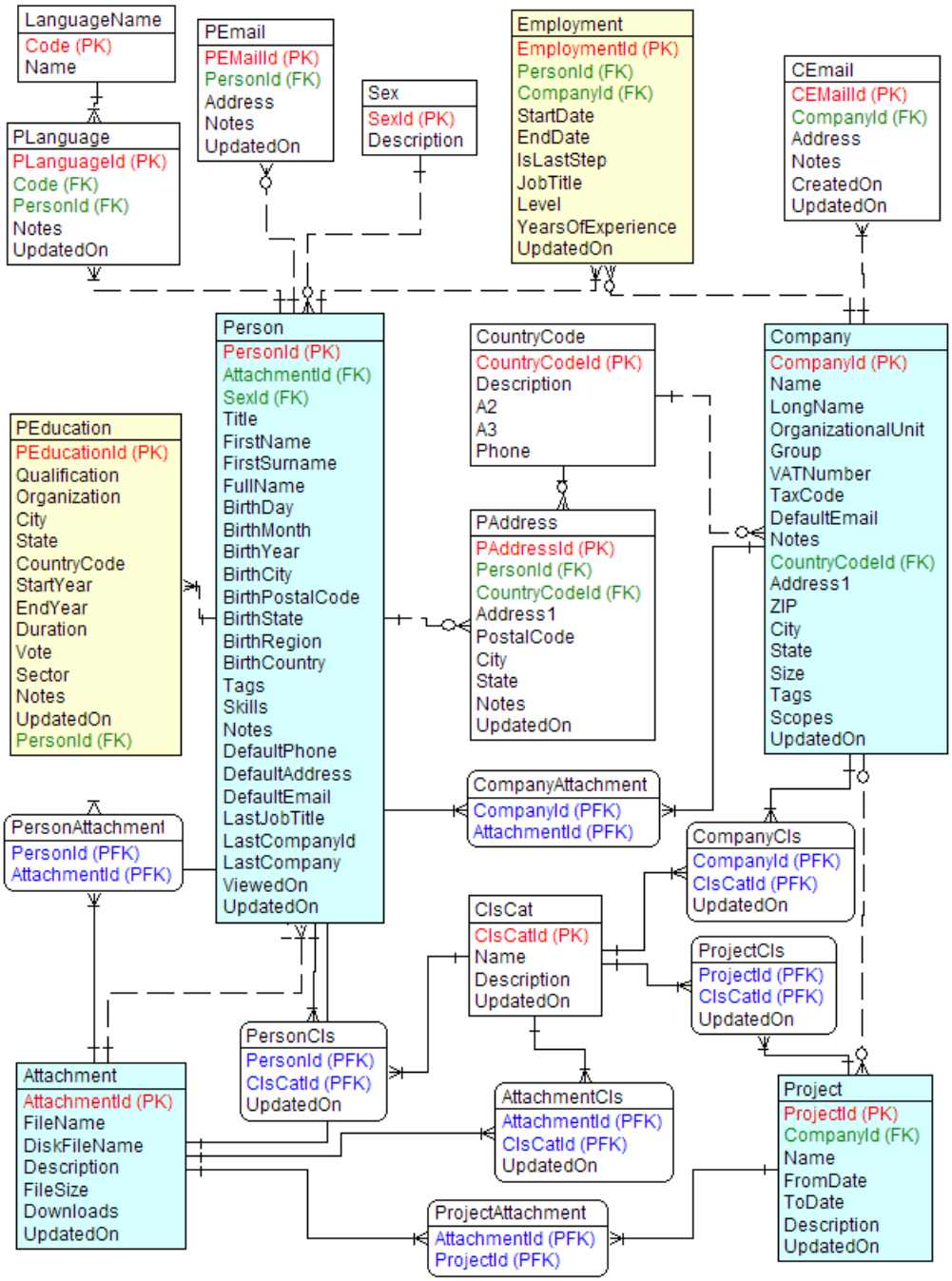

Figura 3.1: Modello ER del database di Arkivium

Nel modellare le tabelle in formato JSON sono state effettuate delle scelte atte a descrivere opportunamente le entità coinvolte agendo sulla struttura dei documenti che rappresentano ciascuna tabella, ognuna delle quali caratterizzata da molteplici campi personalizzati. Tutti i documenti dispongono di alcuni campi comuni, necessari per identificarli e descriverli brevemente; quest'ultimi campi costituiscono la struttura di un **documento base**:

- `doc_type`, un campo *enum* che identifica il *tipo* del documento (e.g, `Persona`, `Azienda`, `Progetto` o `Documento`).
- `arkivium_id`, che corrisponde all'*id* originale del *record* nel database di Arkivium.
- `id`, un campo speciale che Solr utilizza per identificare univocamente i propri documenti all'interno del *core* di appartenenza (questo capo è fondamentale perché usato anche per gestire l'aggiornamento dei documenti in fasi successive alla costruzione dell'indice); nel nostro specifico caso sarà costituito da una stringa, formata dalla concatenazione dei campi `doc_type` e `arkivium_id`.
- `heading`, un campo di tipo testuale, calcolato da Arkivium tramite concatenazione di molteplici informazioni, atto a rappresentare sinteticamente un documento; dovrebbe permettere all'utente di distinguere facilmente un documento dagli altri.

Se quelli appena descritti sono i campi che accumunano tutti i documenti da indicizzare, di seguito vengono sinteticamente esposte le caratteristiche specifiche di ogni tipologia di documento. La spiegazione è corredata di eventuali scelte e tecniche utilizzate per la modellazione, facendo sempre riferimento allo schema ER del database nonché tenendo in considerazione i requisiti di ricerca e aggiornamento dei documenti nell'indice. Quanto viene mostrato è il risultato della sequenza di modifiche apportate di volta in volta nel corso della progettazione funzionale del sistema.

Persone

Dal modello ER in figura 3.1 si evince che per ogni `Persona` vengono registrati i dati anagrafici (`Person`) e di contatto (`PAddress` e `PEmail`), le lingue conosciute (`PLanguage`), il sesso (`Sex`), le classificazioni (`PersonCls`), le esperienze lavorative (`PEmployment`) e la formazione (`PEducation`). Quest'ultime due (in giallo nello schema 3.1) sono descritte da numerosi campi, motivo per cui si è deciso di modellarle come *nested documents*. Ciò significa che `PEmployment` e `PEducation` diventeranno dei documenti JSON veri e propri, innestati all'interno del documento padre di *tipo* `Person`. Ogni `Esperienza lavorativa` è a propria volta connessa con un'`Azienda` e tale relazione viene opportunamente memorizzata all'interno del campo `relations`.

Le restanti proprietà, più semplici, verranno totalmente accorpate all'interno del JSON che rappresenta una `Persona`, in forma di stringhe, numeri oppure come array. Per finire, si noti che ciascuna `Persona` può essere connessa ad un arbitrario numero di `Documenti`, relazione che sarà modellata solamente nel documento di tipo `Attachment`, per motivi spiegati in seguito.

```
{
    "id": "person-15341",
    "doc_type": "person",
    "arkivium_id": 15341,
    "name": "Ferri Marco",
    "heading": "Ferri Marco Full Stack .NET Developer Capitolo Quinto",
    "location": "Viale Giotto, 1 21052 Busto Arsizio  (VA)  - Italia ",
    "default_email": "96marco@libero.it",
    "last_job_title": "Full Stack .NET Developer",
    "last_company": "Capitolo Quinto",
    "skills": [ "Programming", "Web Development", "ASP.NET MVC" ],
    "classifications": [ "Clienti", "Fornitori" ],
    "_childDocuments_": [
      {
        "start_year": 2015,
        "end_year": 2018,
        "sector": "Informatica",
        "title": "Laurea Triennale - Informatica",
        "organization": "Universita' degli Studi di Milano-Bicocca",
        "relations": [ "person-15341" ],
        "doc_type": "education",
        "arkivium_id": 87,
        "id": "education-87",
        "updated_on": "2018-05-14T10:48:54.4127Z"
      },
      {
        "start_date": "2016-07-01T00:00:00Z",
        "title": "Full Stack .NET Developer",
        "organization": "Capitolo Quinto",
        "location": "via Italia, 10 - 20121 Milano  (MI)  - Italia ",
        "relations": [ "person-15341", "company-16289" ],
        "doc_type": "employment",
        "arkivium_id": 16394,
        "id": "employment-16394",
        "heading": "Full Stack .NET Developer Capitolo Quinto",
        "updated_on": "2018-06-01T09:42:08.6814Z"
      }
    ]
}
```


Listato 3.1: JSON che rappresenta un documento Solr di tipo `Persona`

Il codice 3.1 mostra il documento che rappresenta una `Persona`, dal quale si evince che molte informazioni sono state semplificate e il risultato sia decisamente più leggibile rispetto a quanto possa accadere con un database relazionale. Il campo `heading` è composto dalla concatenazione dei campi più significativi, `location` rappresenta sinteticamente le informazioni di un indirizzo e i documenti innestati relativi a `Educazione` e `Professione` sono stati rappresentati nell'array `_childDocuments_`. Tale array assume un ruolo speciale in Solr ed è l'unico modo per indicizzare documenti innestati.

Ovviamente la costruzione di un JSON di questo tipo richiede numerose operazioni di *join* sul database durante la fase di indicizzazione di una `Persona`.

Aziende e Progetti

Entrambi condividono i ragionamenti validi per le Persone, pertanto non ci si soffermerà ad analizzarli approfonditamente né si mostreranno i corrispettivi documenti in formato JSON. I risultati sono molto semplici: sono state opportunamente modellate solamente le proprietà utili per la ricerca e il campo `heading` è ancora una volta ottenuto dalla concatenazione dei campi più rappresentativi del documento.

Documenti

I `Documenti` sono rappresentati nel database tramite la tabella `Attachment`. La modellazione JSON è molto simile alle precedenti, ma ciascun `Attachment` è dotato del campo `relations`, un array di stringhe atto a contenere gli `id` di tutti i documenti con i quali è in relazione, che siano essi `Persone`, `Aziende` o `Progetti`. Un array è il modo migliore per rappresentare l'associazione di tipo `N:N` fra documenti per mezzo di opportuni *reference* (gli `id`), che in un contesto relazionale sarebbero rappresentati dalle chiavi esterne. Su questo campo è possibile effettuare opportune *join query*[4].

```
{
    "id":"attachment-18789",
    "doc_type":"attachment",
    "arkivium_id":18789,
    "heading":"dropbox cover letter.txt",
    "name":"d83e97dd-6b55-45e0-892d-6fdb0dceb361.txt",
    "description":"CV Marco",
    "filename":"dropbox cover letter.txt",
    "extension":"txt",
    "relations":[
      "person-15341",
      "company-12345",
      "project-12344"
    ],
    "classifications":["Curriculum"],
    "updated_on":"2018-05-16T13:10:59Z"
}
```

Listato 3.2: JSON che rappresenta un documento Solr di tipo `Documento`

Rappresentare le relazioni **solo** nel documento di tipo `Attachment` (collegandolo a `Person/Company/Project`), evitando la rappresentazione speculare da `Person/Company/Project` ad `Attachment`, è una scelta atta ad ottimizzare l'indicizzazione *real time*.

[4] `https://wiki.apache.org/solr/Join`

La pratica verrà spiegata in seguito, nella sezione 3.3.3; per ora è sufficiente ragionare sul fatto che l'aggiornamento (o inserimento) di un `Attachment` nel database di Arkivium avviene in concomitanza all'assegnazione dello stesso ad una o più `Persone/Aziende/Progetti`. È normale quindi che la modifica di un `Attachment` porti con sé le relazioni ad esso associate, senza però intaccare nessuna delle altre tabelle. Alla luce di questa considerazione, si rivela triviale aggiornare il campo `relations` di un `Attachment` nel momento in cui si modificano le sue relazioni, operazione che non sarebbe altrettanto banale se si dovessero contemporaneamente aggiornare degli ipotetici campi `attachments` all'interno dei documenti (`Persone`, `Aziende` e `Progetti`) associati all'`Attachment` stesso.

È importante notare che, nonostante ogni `Attachment` si riferisca di fatto ad un vero e proprio file su disco, in questa fase della modellazione non si parla ancora del suo contenuto. Sarà in seguito necessario conoscere il *path* di memorizzazione del file, per poterlo recuperare dal file system e inviarlo a Solr affinché Tika ne estragga il contenuto, che sarà quindi indicizzato (ed eventualmente memorizzato) nel campo `content`.

3.2.3 Schema Solr - prima versione

La modellazione dei dati in formato JSON prevede la definizione dei campi atti a contenere le informazioni sulle quali è necessario effettuare le ricerche. Ciascun campo in Solr deve essere **tipizzato**, in maniera analoga a quanto accade in un linguaggio di programmazione *object-oriented*. Campi e tipi sono descritti all'interno dello **schema**[5], che definisce quale deve essere la struttura degli indici Lucene sottostanti e come trattare il contenuto di ciascun campo.

Lo schema viene descritto in XML tramite un apposito file, a scelta fra i seguenti:

- `managed-schema`, l'impostazione di default, consente di effettuare modifiche allo schema solo tramite apposite API e permette di usufruire dell'approccio *schemaless*;
- `schema.xml` è il file tradizionale, che consente solo modifiche manuali.

Per il caso in esame si è voluto esercitare un forte controllo sullo schema, motivo per cui è stata accantonata la modalità *schemaless*. Dunque, per comodità di modifica del file si è scelto di utilizzare il file `schema.xml`.

[5] `https://lucene.apache.org/solr/guide/7_4/documents-fields-and-schema-design`

FieldTypes

All'interno dello schema si definiscono innanzitutto i tipi (`fieldType`) che verranno utilizzati per la dichiarazione dei campi (`field`). Ciascun tipo è caratterizzato da un nome univoco e associato ad una classe Java che ne definisce il comportamento primario; successivamente, tramite appositi attributi, si specificano ulteriori proprietà:

- `required`, determina se un campo è obbligatorio;
- `indexed`, determina se su un campo è concessa la ricerca;
- `stored`, determina se il valore del campo è memorizzato e quindi restituito fra i risultati di una query;
- `multiValued`, determina se un campo è multivalore;
- `docValues`, definisce apposite strutture per ottimizzare alcune interrogazioni.

Se il `fieldType` è di natura testuale è necessario descrivere anche l'**analisi del testo** prescelta per ciò che esso contiene, costituita innanzitutto da un `Tokenizer`[6] che si occupa di suddividere il flusso di caratteri in *token*, poi seguito da una catena di `Filter`[7] che analizzano, scartano o trasformano i *token* prima dell'indicizzazione.

La prima versione dello schema pensata per Arkivium nasce senza particolari pretese, per ottenere dei risultati semplici ma efficaci. Lo schema di default, che dichiara i principali tipi Solr, viene leggermente modificato per rimuovere i contenuti superflui e adattarlo alle esigenze dell'applicazione. Per i campi testuali si è scelto di usare un'analisi molto minimale, mostrata nel seguente listato.

```
<fieldType name="text_general" class="solr.TextField">
  <analyzer type="index">
    <tokenizer class="solr.StandardTokenizerFactory"/>
    <filter class="solr.StopFilterFactory" words="stopwords.txt" ignoreCase="true"/>
    <filter class="solr.LowerCaseFilterFactory"/>
  </analyzer>
  <analyzer type="query">
    <tokenizer class="solr.StandardTokenizerFactory"/>
    <filter class="solr.StopFilterFactory" words="stopwords.txt" ignoreCase="true"/>
    <filter class="solr.SynonymGraphFilter" expand="true" synonyms="synonyms.txt"/>
    <filter class="solr.LowerCaseFilterFactory"/>
  </analyzer>
</fieldType>
```

Listato 3.3: `schema.xml` - analisi testuale per i campi di tipo `text_general`

[6] https://lucene.apache.org/solr/guide/7_4/tokenizers.html
[7] https://lucene.apache.org/solr/guide/7_4/filter-descriptions.html

Tale processo di analisi testuale si occupa di:

- Suddividere il testo in *token* (`StandardTokenizerFactory`);
- Rimuovere le *stop word*, contenute nel file `stopwords.txt` (`StopFilterFactory`);
- Solo in fase di interrogazione, espandere la query con i sinonimi specificati nel file `synonyms.txt` (`SynonymGraphFilterFactory`);
- Convertire i *token* in lettere minuscole (`LowerCaseFilterFactory`).

Successivamente, con l'ausilio del file di configurazione `enums.xml` mostrato nel listato 3.4, vengono inoltre definiti gli *enum* che descrivono i possibili valori assunti dei campi `document_type`, `project_type` e `sex`.

```
<enumsConfig>
  <enum name="DocumentType">
    <value>person</value>
    <value>company</value>
    <value>project</value>
    <value>attachment</value>
    <value>education</value>
    <value>employment</value>
  </enum>
  <enum name="ProjectType">
    <value>bip</value>
    <value>mailing</value>
    <value>recruiting</value>
  </enum>
  <enum name="Sex">
    <value>female</value>
    <value>male</value>
  </enum>
</enumsConfig>
```

Listato 3.4: `enums.xml` - dichiarazione dei tipi *enum*

```
<fieldType name="document_type" class="solr.EnumFieldType" docValues="true"
            enumsConfig="enums.xml" enumName="DocumentType"/>

<fieldType name="project_type" class="solr.EnumFieldType" docValues="true"
            enumsConfig="enums.xml" enumName="ProjectType"/>

<fieldType name="sex" class="solr.EnumFieldType" docValues="true"
            enumsConfig="enums.xml" enumName="Sex"/>
```

Listato 3.5: `schema.xml` - utilizzo di campi *enum*

Fields

Definiti i `fieldType`, si procede ad elencare i campi che costituiranno la struttura dei documenti da indicizzare.

```
<uniqueKey>id</uniqueKey>
<field name="id" type="string" required="true" />
<field name="doc_type" type="document_type" required="true" />
<field name="arkivium_id" type="pint" />
<field name="heading" type="text_general" />
<field name="updated_on" type="pdate" />
```

Listato 3.6: `schema.xml` - campi comuni a tutti i documenti

I campi comuni a tutti i *tipi* di documento vengono dichiarati per primi. Dall'XML del listato 3.6 si evince che il campo `id` serve a identificare univocamente un documento (`uniqueKey`) e, come `doc_type`, è obbligatorio. `Heading` è un campo di *testo*, mentre `arkivium_id` e `updated_on` sono rispettivamente puntatori a un *intero* e a una *data*.

```
<field name="name" type="text_general" />
<field name="description" type="text_general" />
<field name="notes" type="text_general" />
<field name="location" type="text_general" />
<field name="default_email" type="string" />
<field name="relations" type="strings" />
```

Listato 3.7: `schema.xml` - alcuni campi specifici di alcuni *tipi* di documento

Successivamente si definiscono tutti gli altri campi, che per brevità non sono qui interamente elencati. Da questo piccolo campione emerge `default_email`, che è del tipo *stringa* e cioè, nonostante di natura testuale, trattato come se il suo contenuto costituisse un unico termine. Anche `relations` è dello stesso tipo, ma *multivalore*.

```
<field name="_text_" type="text_general" multiValued="true" stored="false"/>
<copyField source="name" dest="_text_" />
<copyField source="description" dest="_text_" />
<copyField source="notes" dest="_text_" />
<copyField source="location" dest="_text_" />
<copyField source="default_email" dest="_text_" />
```

Listato 3.8: `schema.xml` - altri campi di esempio

Il campo `_text_` è un elemento fondamentale ed assume il cosiddetto ruolo di *catchAll field*, adibito alla raccolta delle informazioni più importanti provenienti dagli altri campi. È utile impostarlo come `defaultField` di ricerca (nel file `solrconfig.xml`), affinché una query possa interrogare questo campo per cercare nella totalità delle

informazioni a disposizione. Si noti che il suddetto è `multiValued` ma non memorizzato (`stored="false"`), poiché ha solo finalità di ricerca e non è necessario che venga restituito fra i risultati di una query. Per creare un campo *catchAll* si fa uso dei `copyField`[8], che consentono di copiare un campo (`source`) all'interno di un altro (`dest`), **prima** che il suo contenuto venga analizzato.

```
<field name="classifications" type="text_general" multiValued="true" />
<copyField source="classifications" dest="classifications_str" />
```

Listato 3.9: `schema.xml` - campo `classifications`

L'ultimo elemento qui presentato è il campo `classifications`. Raccoglie le informazioni provenienti dalle tabelle `PersonCls`, `CompanyCls`, `ProjectCls` e `ClsCat` del database, che servono a suddividere i dati in "categorie". Data la natura e lo scopo di questa categorizzazione, tale campo si presta molto bene ad una ***faceted search***, motivo per cui si rivela fondamentale preservare invariato il testo all'interno dell'indice affinché possa essere presentato all'utente nella sua interezza durante il *faceting*, che altrimenti verrebbe suddiviso in *token* e sottoposto alla catena di filtri risultando irriconoscibile per l'utente. È comune, quindi, effettuare una copia dei campi sui quali si hanno questo tipo di esigenze.

Più nello specifico, per effettuare la copia si sfrutta il concetto di `dynamicField`[9], che permette di definire dei tipi di natura dinamica, sfruttando il *suffisso* (o il *prefisso*) del nome di un campo.

```
<dynamicField name="*_str" type="strings"
              docValues="true" indexed="false" stored="false"/>
```

Listato 3.10: `schema.xml` - esempio di `dynamicField`

Il risultato è che dal campo `classification` viene creato un nuovo campo di nome `classification_str`, non indicizzato, né memorizzato o analizzato (`strings`), ma ottimizzato per il *faceting* (`docValues="true"`).

3.3 Processo di indicizzazione

La costruzione (*build*) degli indici è il primo problema che si è affrontato durante lo sviluppo del sistema. Il processo ha lo scopo di fornire al server le informazioni da indicizzare ed è fondamentale, poiché durante questa fase che Solr analizza i documenti ricevuti per indicizzarli secondo le modalità descritte nello schema. L'importazione in Solr dell'intera mole di informazioni a disposizione avviene una tantum, durante l'inizializzazione del sistema, e successivamente ogni volta che si ha la necessità di ricostruire

[8] `https://lucene.apache.org/solr/guide/7_4/copying-fields.html`
[9] `https://lucene.apache.org/solr/guide/7_4/dynamic-fields.html`

gli indici in seguito ad una modifica dello schema che prevede la reindicizzazione dei documenti (*rebuild*).

Qualsiasi successiva modifica, da parte dell'utente, alle informazioni contenute in Arkivium deve essere propagata anche al server Solr affinché le ricerche risultino consistenti. L'operazione di aggiornamento degli indici di Lucene non è banale, ma Solr mette a disposizione specifiche funzionalità per rendere il processo quanto più trasparente ed immediato possibile, affinché l'utente non si accorga di nulla.

Vengono ora descritte le fasi di creazione a aggiornamento degli indici, ponendo particolare attenzione alle metodologie con le quali sono state implementate dall'applicazione. Si presentano alcune strategie risolutive strettamente legate all'ambiente di sviluppo di Arkivium, ma replicabili da qualsiasi ecosistema con simili funzionalità.

3.3.1 Esportazione del database

Build e *rebuild* di un indice sono operazioni che possono richiedere molto tempo e capacità computazionali, sia per reperire le informazioni da inviare a Solr sia per quanto riguarda il processo di analisi a cui ciascun documento deve essere sottoposto da parte del server. Si è scelto di gestire questo tipo di operazioni tramite appositi programmi batch, da eseguire all'occorrenza.

Arkivium è sviluppato in ambiente .NET e sfrutta un database relazionale come principale unità di memorizzazione delle informazioni. L'importazione dei dati in Solr ha avuto inizio con l'esportazione del database. A tale scopo si è sfruttato *Entity Framework*[10], il più famoso *Object Relationship Mapping* (ORM) di casa Microsoft, per effettuare le dovute query sul database e reperire tutte le informazioni da inviare a Solr.

Più nello specifico, partendo dalle tabelle `Person`, `Company`, `Project` e `Attachment`, viste in figura 3.1, si sono effettuate le opportune operazioni di *join* sul database per ricavare dei *Data Transfer Object* (DTO) atti a rappresentare i documenti prescelti durante la fase di modellazione dei dati; sono state create le classi C# `SolrPerson`, `SolrCompany`, `SolrProject` e `SolrAttachment`.

Gli oggetti di tipo `SolrPerson`, `SolrCompany` e `SolrProject` sono stati convertiti in JSON, seguendo le opportune convenzioni Solr, e inviati al server affinché il *request handler*[11] che risponde all'indirizzo `#host#/update` potesse interpretare i documenti ricevuti e indicizzarli secondo le disposizioni contenute nello schema.
Al contrario, gli `Attachment` sono stati sottoposti ad una procedura differente poiché differente è il *request handler*[12] che gestisce l'estrazione del contenuto testuale di un file attraverso Apache Tika (sezione 2.3.2) e risponde all'indirizzo `#host#/update/ex`

[10] `https://docs.microsoft.com/it-it/ef/`

[11] `https://lucene.apache.org/solr/guide/7_4/uploading-data-with-index-handlers`

[12] `https://lucene.apache.org/solr/guide/7_4/uploading-data-with-solr-cell-using-apache-tika`

`tract`. In questo caso il parametro di input principale è il file stesso, le cui proprietà devono essere elencate non come JSON bensì attraverso la *querystring* della richiesta, serializzando opportunamente la classe `SolrAttachment`.

A causa dell'elevato numero di *join* sul database e conversioni necessarie per modellare opportunamente ciascun documento, come spiegato nella sezione 3.2.2, ogni esecuzione del processo batch di esportazione richiede un quantitativo di risorse non indifferente e può impiegare anche diverse ore per essere portato a termine. Allo stesso modo, anche l'importazione dei dati in Solr è soggetta ad una richiesta di risorse non indifferente, tanto più alta quanto più complicata è l'analisi testuale cui si sottopone ciascun documento. Per questi motivi è bene che le operazioni di *rebuild* siano effettuate solo quando strettamente necessario e preferibilmente durante periodi in cui l'utilizzo del sistema sia minimo (ad esempio di notte o nel weekend), affinché l'utente non si accorga di eventuali rallentamenti.

3.3.2 Dimensioni degli indici

La dimensione degli indici dipende dal quantitativo di documenti indicizzati, dal numero dei termini che entrano a far parte del dizionario dell'indice e dalla quantità di testo che si decide di memorizzare per ciascun documento (attributo `stored`). Inoltre, le dimensioni sono soggette a periodici cambiamenti causati dalla gestione a basso livello degli indici. Nello specifico, Solr deve occuparsi di rendere disponibili per la ricerca le informazioni indicizzate come se facessero parte di un unico indice, quando nella realtà è consuetudine avere a che fare con molteplici indici che attendono di essere sottoposti ad un *merge*. Tale operazione viene eseguita durante l'ottimizzazione di un *core* (o *collection*) e richiede un elevato numero di risorse; è da eseguirsi periodicamente, durante momenti di minimo utilizzo del sistema, e può causare un aumento temporaneo delle dimensioni degli indici.

In funzione di queste considerazioni non è possibile fornire una stima precisa della crescita di un indice Solr. Per quanto riguarda i test effettuati durante questa prima fase dello sviluppo si è riscontrato che, sfruttando un'analisi testuale molto basica come quella prescelta per il primo schema (descritto nella sezione 3.2.3), le dimensioni degli indici sono davvero ridotte. Nello specifico, indicizzare circa 65mila documenti ha richiesto solamente uno spazio di circa **100 MB**[13], considerando che l'unica informazione non memorizzata (`not stored`, non reperibile fra i risultati di una query) si trattava del contenuto estratto dai file, esclusivamente indicizzato, nel campo `content`.

Si mostrerà in seguito che la memorizzazione di `content`, nonché un'analisi testuale decisamente più complessa atta a migliorare il *matching* fra query e documenti, farà aumentare considerevolmente la dimensione degli indici. Per lo stesso numero di documenti saranno necessari ben **10 GB** di indici **non ottimizzati**.

[13]Un test con circa 1 milione di documenti ha prodotto un indice dalla dimensione di 1,5 GB

3.3.3 Near real time search

Quando un utente inserisce o aggiorna un dato appartenente al database di Arkivium è necessario che tale modifica venga propagata al server Solr affinché le informazioni ricercabili rimangano coerenti. I due sistemi infatti, nonostante siano progettati per cooperare, esistono indipendentemente l'uno dall'altro ed è premura del programmatore fare in modo che rimangano allineati.

A questo scopo si è deciso di implementare tramite *Entity Framework* (EF) un meccanismo che invii al server Solr un documento ogni qualvolta viene effettuata la modifica di una tabella del database sulla quale si hanno esigenze di ricerca. In breve, con EF ciascuna tabella del database è modellata per mezzo di una classe C#, di cui un'istanza rappresenta un *record* della tabella stessa. Per aggiornare o inserire un nuovo *record* nel database è sufficiente istanziare un oggetto e indicare al framework che lo si vuole salvare nel database; un apposito metodo (`SaveChanges`) si occuperà di gestire il salvataggio. È possibile ridefinire il metodo di salvataggio affinché esegua operazioni personalizzate, ed è questa la strategia che si è rivelata fondamentale per aggiornare il server Solr ogni qualvolta se ne rilevi il bisogno. Più dettagliatamente, si è creata un'interfaccia C# con il solo scopo di contrassegnare le classi indicizzabili; tale interfaccia è quindi stata implementata dalle classi `Person`, `Company`, `Project` e `Attachment`, affinché durante l'esecuzione del metodo `saveChanges` ci si potesse accorgere della necessità di indicizzare questo tipo di oggetti e, di conseguenza, serializzarli per l'invio al server Solr. Tale procedura è opportunamente configurata per le operazioni `INSERT`, `UPDATE` e `DELETE` sul database.

Questa soluzione è risultata molto innovativa rispetto a quella adottata dal precedente sistema di ricerca implementato in Lucene. Infatti, come precedentemente accennato, ogni modifica ad un indice implica la creazione di strutture temporanee le cui informazioni possono essere rese visibili per la ricerca solo in un secondo momento. Per questo motivo, utilizzando Lucene diventa necessario programmare l'aggiornamento e il *merge* degli indici con cadenza periodica; nel caso di Arkivium, questi venivano aggiornati tutte le notti e ciò stava a significare che un utente era in grado di cercare nuove informazioni solo il giorno seguente il loro inserimento.

Con Solr questo problema è quasi totalmente superato, poiché introduce funzionalità di ricerca e indicizzazione *near real time*[14]. Ciò significa che, nonostante la gestione degli indici rimanga affidata a Lucene, la piattaforma mette a disposizione alcune operazioni che coinvolgono log, memoria volatile e memoria stabile in maniera assolutamente trasparente per l'utente e per il programmatore, rendendo possibile la visualizzazione delle modifiche in tempi relativamente brevi.

[14] `https://lucene.apache.org/solr/guide/7_4/near-real-time-searching.html`

Si introducono i concetti di:

- *soft commit*, che rende visibili i cambiamenti pur lasciandoli su memoria volatile;
- *(hard) commit*, che salva le ultime modifiche su memoria stabile;
- *optimize*, che effettua il *merge* dei vari segmenti costituenti l'indice.

Le tre operazioni hanno costi importanti, sia computazionali che di spazio, ed è opportuno che vengano utilizzate con cognizione di causa, specie in un ambiente soggetto a frequenti aggiornamenti. È possibile configurare la frequenza con la quale devono essere eseguiti *soft commit* e *hard commit*.

Non c'è una regola precisa per pianificare l'esecuzione dei comandi, ma personalmente consiglio un *soft commit* ogni 10-240 secondi, in base alle esigenze informative dell'utente e il carico di lavoro al quale il server Solr è sottoposto. Gli *hard commit* trasferiscono a tutti gli effetti le nuove informazioni su memoria stabile e prevengono perdite di dati dovute ad eventuali crash del software, ma sono più costosi ed è per questo motivo che potrebbero bastare un paio di esecuzioni all'ora. È comunque consigliato disattivare entrambe le funzioni durante *build* e *rebuild* dell'indice.

Infine, *optimize* è un processo decisamente più oneroso che può richiedere anche molte ore per essere portato a termine; pertanto se ne consiglia un utilizzo periodico, con cadenza giornaliera o settimanale, per ottimizzare considerevolmente la gestione dello spazio su disco ed eventualmente agevolare le prestazioni di ricerca.

3.4 Interfaccia di ricerca

L'interfaccia utente è parte fondamentale della realizzazione di un motore di ricerca, specie in un contesto di *enterprise search* che, a differenza della classica *web search*, prevede l'utilizzo di numerose funzionalità e parametri di ricerca altamente specializzati. Come per il resto del sistema, anche lo sviluppo della GUI ha seguito un approccio di sviluppo incrementale ed è forse questa la fase in cui si evince maggiormente l'evoluzione graduale del motore di ricerca.

3.4.1 Progettazione dell'esperienza utente

Esigenze informative e intuitività di utilizzo sono i punti cardine che hanno condotto la progettazione del sistema dal punto di vista utente, che usufruisce dell'applicazione attraverso il proprio *web browser*.

Il precedente sistema integrato in Arkivium consentiva di effettuare delle ricerche focalizzate solamente su `Persone` e `Documenti`, prevedendo un approccio di ricerca *bottom-up*: per sottoporre le richieste, l'utente era costretto a recarsi nelle apposite pagine e comporre la propria query indicando il testo da cercare attraverso multipli input testuali, ognuno dei quali corrispondente a un campo dell'indice Lucene sottostante.

Nonostante tale soluzione si riveli in molti casi adatta alle esigenze informative di un *recruiter*, si è pensato che questo tipo di *workflow* potesse rivelarsi tutto sommato abbastanza limitante. È per questo motivo che riprogettando l'esperienza utente si è pensato di affrontare le ricerche con una metodologia *top-down*: predisporre una barra di ricerca principale **sempre accessibile** all'utente, qualunque sia l'operazione che si sta svolgendo, che sia in grado di **cercare su tutte le informazioni a disposizione**, e cioè sul fatidico campo `_text_` che è stato introdotto nello schema (listato 3.7). L'interfaccia che presenta il risultato di questa generica interrogazione dovrebbe quindi ben suddividere le informazioni reperite affinché l'utente riesca immediatamente a discriminare il *tipo* dei documenti trovati. A questo punto sarà possibile raffinare ulteriormente la ricerca con apposite funzioni.

Ciascuna delle funzionalità avanzate è stata introdotta gradualmente all'interno dell'interfaccia, nell'ottica di un miglioramento incrementale non invasivo e costantemente intuitivo per l'utente, che dovrebbe essere in grado di utilizzare il motore di ricerca senza che vi sia la necessità di una particolare esperienza o formazione pregressa.

3.4.2 Invio di richieste e interpretazione dei risultati

Nonostante questo capitolo sia principalmente dedicato ad elementi di usabilità del software, è opportuno specificare che ogni funzione messa a disposizione dell'utente, semplice o complessa che si riveli, è il risultato di uno sviluppo a 360 gradi: studio teorico della funzionalità, codifica della richiesta, *parsing* dei risultati e realizzazione dell'interfaccia grafica che consenta all'utente di sottoporre la query e visualizzarne i risultati in maniera opportuna.

Il paradigma MVC ha consentito un'adeguata separazione delle responsabilità. La pagina dei risultati (la *view*) è sempre renderizzata dal web server (quindi dal *controller*) e l'utilizzo di Javascript è stato ridotto al minimo. Il front-end è stato progettato affinché ciascuna modifica dei parametri di ricerca provochi un *refresh* della pagina, che passa il controllo al back-end e nello specifico a un `client` .NET creato *ad hoc* per comunicare con Solr e soddisfare la richiesta espressa nella *querystring* (che costituisce il *model*). Tale `client` interpreta il *model* e invia l'opportuna richiesta HTTP al server Solr, attende il risultato e ne effettua il *parsing*. Infine lo restituisce al *controller*, il quale renderizza la *view* da inviare al browser dell'utente, che finalmente può consultare i risultati della propria query.

Dal momento in cui viene sottoposta richiesta, l'applicazione è in grado di presentare un risultato strutturato all'utente in meno di un secondo. L'effettiva esecuzione della query sul server Solr impiega un tempo molto ridotto rispetto a quanto richiesto invece dalla renderizzazione della *view* da parte del *controller*, che costituisce la fase computazionalmente più dispendiosa a causa della numerosità degli elementi da visualizzare adeguatamente in pagina a seguito dell'interpretazione della risposta ricevuta dal server.

In futuro si potrebbe pensare di implementare il processo esecutivo tramite AJAX per ridurre ulteriormente i tempi di attesa e migliorare l'esperienza utente.

3.4.3 Interrogazioni di base

La possibilità di poter effettuare una ricerca in qualsiasi istante è stato il primo obiettivo che ci si è posti per lo sviluppo e ha dato inizio alla realizzazione dell'interfaccia grafica. A tale scopo si è optato per una soluzione molto classica che ha visto l'introduzione di una casella di testo nella parte superiore di tutte le pagine dell'applicazione. È stata concepita affinché la ricerca di una o più parole attraverso la stessa provochi sempre l'apertura di una *nuova scheda del browser*, affinché l'utente, quando si accorge dell'esigenza di cercare qualcosa, non debba interrompere bruscamente l'operazione in corso di svolgimento.

Si è tentato di approcciare questo tipo di ricerca tenendo in considerazione le aspettative dell'utente nei confronti di una barra di ricerca globale, che per tradizione sul Web ha lo scopo di cercare quanto richiesto nella totalità della mole di informazioni disponibili, discriminando opportunamente la rilevanza dei contenuti sui quali viene effettuata la ricerca. Se ad esempio l'utente sta cercando un nome di persona, è naturale che fra i risultati della ricerca appaiano ben in evidenza le persone con tale nome, mentre minore importanza venga riservata, per esempio, ai *curriculum* nei quali il nome è presente ma compare sotto una voce meno significativa, come può essere l'elenco dei correlatori di una tesi di laurea.

Tale esigenza può essere soddisfatta con Lucene assegnando un peso a ciascuna delle parole o dei campi sui quali si sta effettuando la ricerca. Tale pratica prende il nome di **boosting** e prevede che la query venga formulata con una specifica sintassi[15], non particolarmente complicata ma sicuramente poco intuitiva da ricordare da parte di un utente. Progettando una soluzione che si rivelasse trasparente per l'utente, si è scoperto che Solr mette a disposizione nuovi parametri e sintassi per la composizione delle query, tramite l'utilizzo di opportuni *query parser*.

Fra questi, particolarmente utile è stato il *query parser* `edismax`[16], che si dimostra più tollerante agli errori sintattici e aggiunge nuove funzionalità al *parser* standard di Lucene. Fra queste, vi è l'introduzione del parametro *query field* (`qf`), attraverso il quale è possibile elencare i campi dei documenti sui quali deve essere effettuata la ricerca, permettendo inoltre di specificare dei *boost* su ciascuno di essi.
Si è quindi scelto di utilizzare `edismax` come *parser* di tutte le ricerche e insieme ad esso si è stabilito un insieme di parametri comuni a ciascuna richiesta, affinché l'utente possa sempre beneficiare di alcune comodità che verranno di volta in volta introdotte durante tutto lo sviluppo del motore di ricerca. La barra di ricerca finora descritta costituirà sempre la query principale, e cioè il parametro `q` di una richiesta.

[15] `https://lucene.apache.org/core/2_9_4/queryparsersyntax.html#Boosting%20a%20Term`
[16] `https://lucene.apache.org/solr/guide/7_4/the-extended-dismax-query-parser.html`

La prima pagina realizzata per la visualizzazione dei risultati di una ricerca è davvero molto basilare e la si può osservare in figura 3.2. Consiste in una semplice tabella che elenca i documenti reperiti dall'esecuzione della query, suddivisi per pagine ma poco interpretabili da parte dell'utente. In questa fase dello sviluppo, le uniche personalizzazioni consentite sono la scelta del numero di documenti da visualizzare per ciascuna pagina e la possibilità di decidere se l'insieme delle parole inserite nell'apposita search bar siano da cercare in `AND` (tutte devono essere presenti nei documenti affinché vengano considerati rilevanti) oppure in `OR` (è sufficiente che i documenti contengano almeno una delle parole cercate).

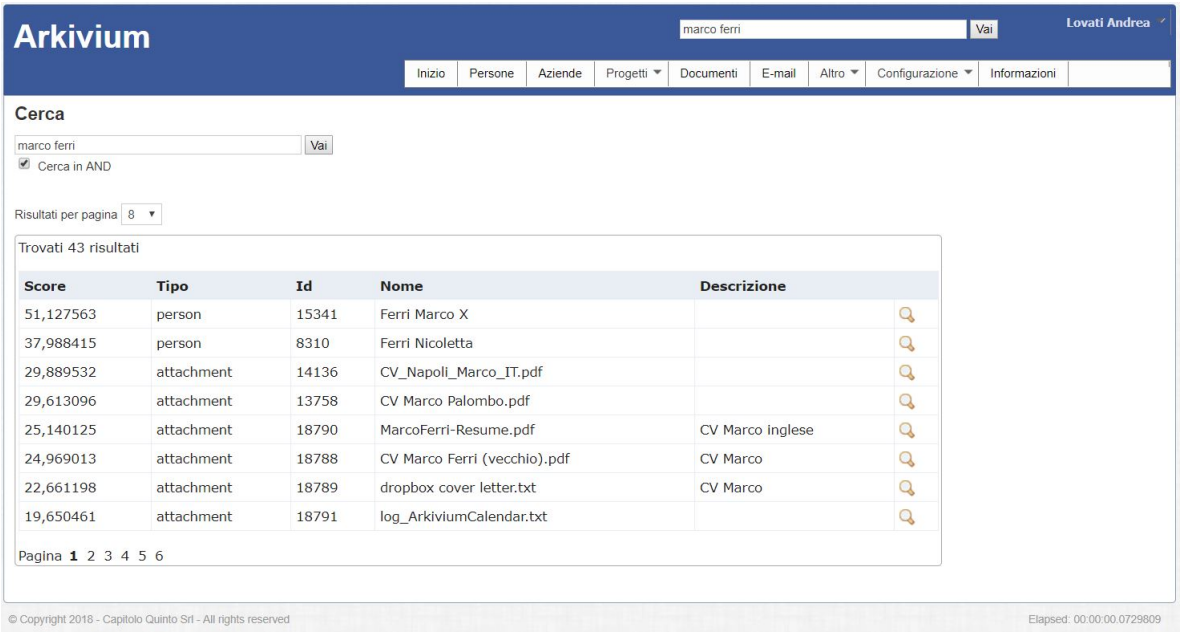

Figura 3.2: Prima versione della nuova pagina di ricerca pensata per Arkivium

Con così poche informazioni non si è tuttavia in grado di discriminare immediatamente né il tipo di documento cui ciascuna riga appartiene, né il motivo per cui esso sia stato stimato come rilevante da parte del sistema. Tale pagina è stata sottoposta a miglioramenti graduali atti ad ampliarne usabilità e funzionalità.

3.4.4 Faceted search

Per consentire una maggiore leggibilità dei risultati di ricerca ed in particolare comprendere immediatamente la natura delle informazioni trovate, si è deciso di realizzare un'interfaccia che presenti i documenti reperiti suddivisi per *tipo*: `Persone`, `Aziende`, `Progetti` e `Documenti`. Questo approccio consente inoltre di personalizzare il modo in cui viene visualizzato ciascun *tipo* di documento, progettando un meccanismo che permetta all'utente di passare da una visualizzazione all'altra, ciascuna delle quali è stata implementata tramite un apposito *tab*.

È importante che l'utente sia consapevole del numero di documenti reperiti per ciascun *tipo*, affinché possa opportunamente decidere su cosa concentrare la propria attenzione. Ciò si concretizza nell'implementazione di una ricerca "sfaccettata"[17], che consente di ottenere il conteggio dei documenti (fra quelli reperiti) che assumono specifici valori per un determinato campo.

Se il campo su cui si effettua il *faceting* è il `doc_type`, si ottengono le informazioni necessarie ad associare il conteggio a ciascun *tab*. La scelta di un *tab* da parte dell'utente determina il *tipo* dei documenti attraverso cui filtrare la query, per mezzo del parametro *filter query* (`fq`). Il filtro è stato implementato in modo che la scelta di un *tab* non influisca sul conteggio dei documenti appartenenti ad altri *tab* (cioè ad un *tipo* diverso da quello prescelto dall'utente); tale conteggio risulterebbe altrimenti sempre `0`, dato che l'appartenenza di un documento ad un certo *tipo* esclude l'appartenenza agli altri.

È stato anche predisposto un *tab Generale* che consenta la visualizzazione dei documenti reperiti indipendentemente dal *tipo*. É doveroso inoltre far notare che `Formazione` ed `Esperienze professionali`, nonostante siano informazioni innestate all'interno di una `Persona`, vengono trattati da Solr come se fossero documenti veri e propri e di conseguenza sono stati dedicati loro appositi *tab*.
Nella figura 3.3 viene mostrato il risultato.

Applicando il *faceting* ad altri campi è possibile ottenere un'ulteriore suddivisione dei documenti in "categorie", che possono essere sfruttate per raffinare il risultato della ricerca. La funzionalità è stata implementata per default in tutte le richieste, solamente sui campi che assumono valori prestabiliti o circoscritti in un dominio ristretto. Il risultato consiste in gruppi di *checkbox*, uno per campo, che hanno preso posto sulla zona destra dell'interfaccia e consentono di filtrare la query principale.

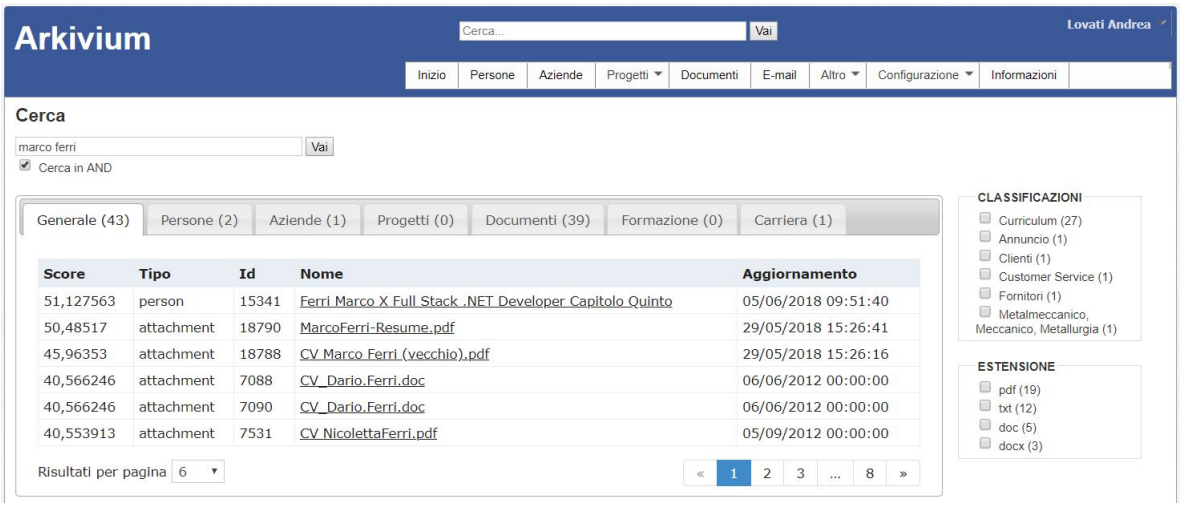

Figura 3.3: Implementazione grafica del *faceting*

[17] `https://lucene.apache.org/solr/guide/7_4/faceting.html`

3.4.5 Ricerca e ordinamento per campo

Nonostante - se correttamente implementata - una ricerca globale sia in grado di produrre ottimi risultati, in un contesto come quello delle risorse umane può rivelarsi necessario sottoporre delle query più strutturate per filtrare i documenti con una maggiore precisione. È per questo motivo che vengono introdotte le interrogazioni e l'ordinamento *per field*, che consentono di avere un maggiore controllo dei risultati.

In base ai campi che si vogliono interrogare, la ricerca può essere testuale oppure focalizzata su tipi di dati numerici, temporali, geografici o enumerativi; è compito dell'applicazione mettere a disposizione i componenti grafici adatti a sottoporre richieste che siano coerenti rispetto ai campi da interrogare.

Nel corso dello studio ci si è focalizzati principalmente sui campi di testo, ma come si può osservare in figura 3.4 sono stati effettuati dei test anche su range numerici (`anno`) ed *enum* (`sesso`). Ciascun *tab* è stato dotato degli appositi filtri ed è stata introdotta la possibilità di ordinare i risultati per campo, oltre che per rilevanza.

Le ricerche per campo sono state implementate attraverso il parametro *filter query* (`fq`), che applica una selezione preventiva sull'indice prima di effettuare la query primaria (parametro `q`): ciò significa che i *filter query* non hanno alcuna influenza sul punteggio (`score`) dei documenti e quindi sul concetto di rilevanza.

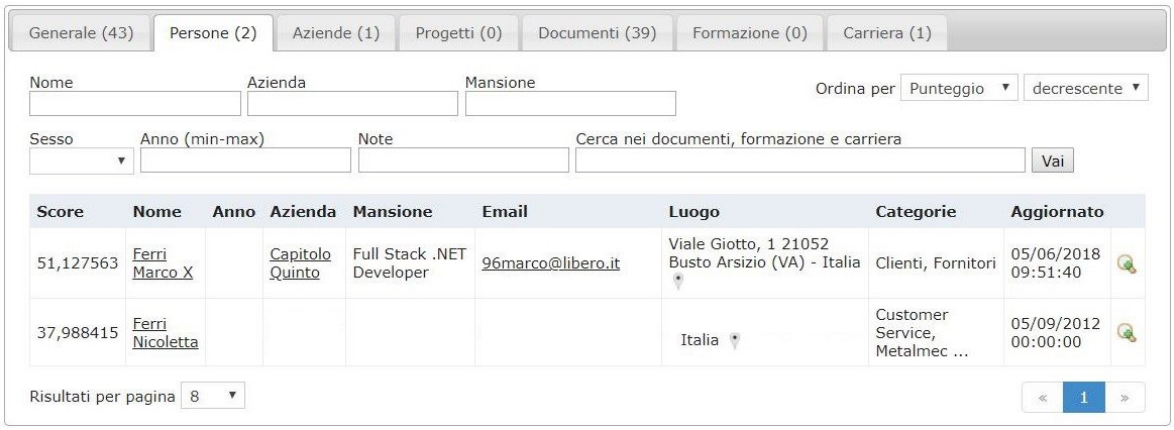

Figura 3.4: Esempio di ricerche e ordinamento *per field*

Fra i filtri caratterizzanti una `Persona`, è stato inoltre introdotta un'ulteriore casella di testo con la dicitura *"Cerca nei documenti, formazione e carriera"*. Questa costituisce una funzionalità molto importante per il caso di studio in esame, perché attraverso opportune *join query*[18] consente di effettuare interrogazioni congiunte all'interno di documenti Solr fra loro in relazione, come quelli che rappresentano `Persone` e `Documenti`. In questo modo, l'utente può combinare la ricerca sulle `Persone` alla ricerca sui `Documenti` delle persone filtrate, e quindi ad esempio analizzare i *curriculum* in

[18] `https://wiki.apache.org/solr/Join`

maniera più precisa. È inoltre possibile sfruttare la medesima tecnica anche per cercare nei documenti innestati, come nel caso di `Formazione` e `Carriera`.

3.4.6 Highlighting

Gli strumenti che sono finora stati presentati e fanno parte dell'interfaccia utente consentono di effettuare query abbastanza complesse ma efficienti.

Un possibile esempio di query potrebbe essere: "Cerca tutte le `Persone`, ordinate per `nome`, in cui compaia la parola *developer*, che lavorino in *Google*, abbiano la parola *Java* nel `curriculum` e abbiano frequentato un `corso` di *informatica* in *Bicocca*."

Se la query è così articolata risulta semplice comprendere i motivi che hanno portato un determinato documento a essere considerato rilevante: soddisfa tutte le richieste.

Viceversa, se la query è espressa più vagamente - come avviene nella maggior parte dei casi - diventa difficile comprendere i motivi che hanno prodotto un determinato risultato. Per questo motivo nasce l'***highlighting***[19], una funzione che consente di mettere in evidenza i termini della query che sono stati trovati nei documenti. Ciò si rivela particolarmente utile nel caso in cui si stia effettuando la ricerca su testi molto lunghi, per i quali diventa molto difficile individuare le occorrenze delle parole cercate.

A tal proposito si è deciso di introdurre l'*highlighting* come parametro comune a tutte le richieste, facendolo operare su tutti i campi dei documenti. Affinché funzioni, i campi su cui agisce devono essere indicizzati (`indexed`) e memorizzati (`stored`), pertanto si è rivelato necessario modificare lo schema per memorizzare anche il `content` estratto dai `Documenti`, precedentemente solo indicizzato. Dopo un processo di *rebuild* si è notato che la dimensione degli indici **non ottimizzati** è quadruplicata; ciò si rivelerà un elemento critico nella fase di miglioramento dell'analisi testuale.

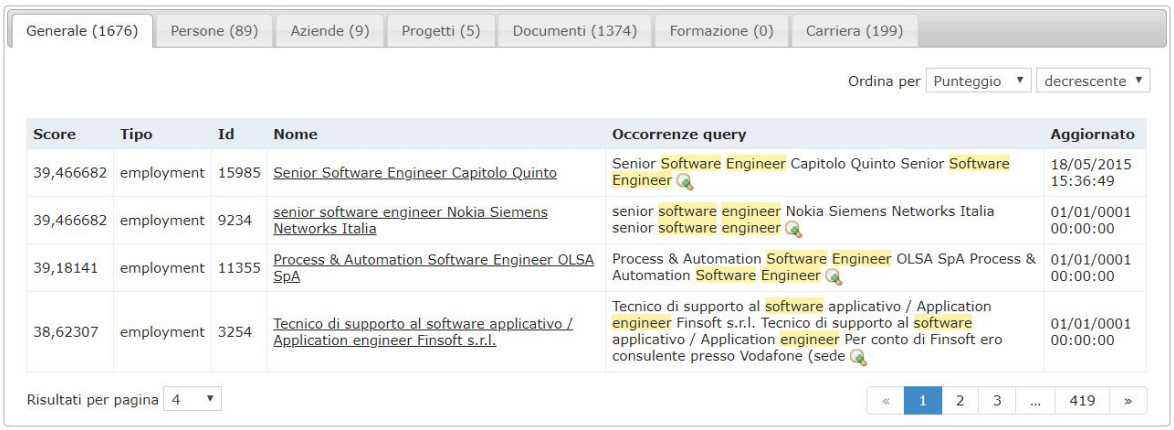

Figura 3.5: Implementazione della funzione di *highlighting*

[19] `https://lucene.apache.org/solr/guide/7_4/highlighting.html`

3.4.7 Una nuova *search bar*

Con Lucene (e Solr) è consentito effettuare ricerche di parole multiple in `AND` oppure in `OR`, ma è anche possibile indicare, con un'opportuna sintassi, che si desidera cercare frasi esatte (***phrase query***) o termini simili a quelli indicati nella query (***fuzzy query***)[20].

Questo tipo di ricerche possono rivelarsi utili in determinate situazioni, ma è poco probabile che l'utente medio ricordi le opportune sintassi, che non dovrebbe essere tenuto a conoscere. Si è quindi deciso di rendere trasparenti le funzionalità all'interno dell'interfaccia, riprogettando la barra di ricerca secondo quanto mostrato nella figura 3.6. All'elenco di funzioni mancherebbe la possibilità di specificare le parole che l'utente vuole **non** compaiano all'interno dei risultati.

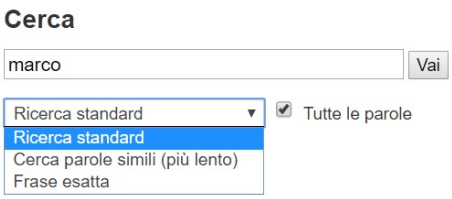

Figura 3.6: Nuova *search bar*

Infine, sfruttando il *suggester*[21] di Solr è stato implementato anche l'**autocompletamento** della query attraverso opportuni dizionari, ricavati dai termini memorizzati nell'indice inverso; ciò consente all'utente di essere guidato durante la composizione dell'interrogazione.

In futuro si potrà pensare di sfruttare lo *spellcheck*[22] per implementare il cosiddetto *"forse cercavi..."* o eventualmente correggere (espandere) automaticamente la query dell'utente, in caso di errori di battitura.

3.5 Miglioramento dei risultati di ricerca

L'ultimo problema che si è affrontato riguarda il processo di analisi da operare sul contenuto testuale dei documenti, prima che essi vengano indicizzati, affinché i risultati di un'interrogazione si rivelino davvero utili per le esigenze informative dell'utente. Lo scopo è quello di massimizzare la frazione di documenti rilevanti che vengono reperiti (*richiamo*), minimizzando il reperimento di documenti non rilevanti (*precisione*). Si è deciso di lasciare alla fine la trattazione dell'argomento poiché decisamente complesso e indipendente rispetto alla progettazione dell'interfaccia, che ha richiesto tempistiche di sviluppo importanti.

In prima battuta si è previsto uno schema che proponesse una catena di analisi molto minimale (*tokenization* > *stop word removing* > *lowercasing*), completamente indipendente dalla lingua di utilizzo e anche piuttosto limitante. Il meccanismo di *matching* di Lucene si basa sul **confronto esatto** dei termini ricavati durante le

[20] `https://lucene.apache.org/solr/guide/7_4/the-standard-query-parser.html`
[21] `https://lucene.apache.org/solr/guide/7_4/suggester.html`
[22] `https://lucene.apache.org/solr/guide/7_4/spell-checking.html`

fasi di indicizzazione e interrogazione, quindi è importante che le corrispettive analisi producano termini adeguatamente confrontabili. Se le parole contenute nel testo venissero indicizzate così come sono, l'utente potrebbe avere serie difficoltà nel sottoporre query che reperiscano una buona percentuale di risultati rilevanti rispetto alla totalità dei documenti rilevanti indicizzati (il concetto di *richiamo* enunciato nella sezione 1.5).

L'aspetto principale da considerare durante l'analisi testuale è la **lingua**: ciascuna possiede regole morfologiche ben precise, pertanto sarebbe opportuno che ogni documento (o campo) venga analizzato per mezzo di operazioni linguistiche altrettanto specifiche.

Ciò non è sufficiente, perché per ottenere un *matching* realmente versatile è necessario trattare adeguatamente tutti i caratteri separatori ed eventualmente consentire anche il **confronto parziale** fra termini. Se un utente cerca la parola *develop*, tendenzialmente si aspetta che vengano reperiti anche i documenti contenenti la parola ***develop**er*. Con il meccanismo di *matching* di Lucene questo non avviene e non risulta per niente banale configurare il sistema per ottenere risultati che siano accettabili sia dal punto di vista dell'efficacia (*precisione/richiamo*) che dell'efficienza (*spazio/tempo*).
Questo è un problema ben noto[23], che richiede una considerevole esperienza sul campo ed è stato affrontato con un approccio di tipo sperimentale, atto al raggiungimento di una soluzione qualitativamente accettabile.

3.5.1 Ristrutturazione dello schema - seconda versione

Analisi linguistica

L'analisi linguistica è un argomento estremamente complesso, che per essere affrontato nella maniera corretta dovrebbe innanzitutto prevedere un sistema di rilevamento automatico della lingua, affinché si possano successivamente applicare al testo un insieme di operazioni altamente specifiche che consentano di interpretarne correttamente i contenuti e fornire un adeguato supporto alla ricerca.

Le principali operazioni linguistiche consistono nello *stemming*, ossia il "troncamento" di ciascuna parola ad una forma più basilare, e l'espansione dei termini della query attraverso i *quasi-sinonimi*, parole con una forte correlazione semantica rispetto ai termini originali. Entrambi i passaggi sono molto utili affinché ciascuna interrogazione sia sufficientemente versatile: ad esempio, è logico che la ricerca della parola *developer* in una base documentale in lingua italiana dovrebbe reperire anche tutti i documenti nei quali compaiono le parole *programmatore*, *programmatrice*, *sviluppatori*, *software engineer*, etc.

L'implementazione dello ***stemming***, in Solr, consiste esclusivamente nella scelta del corretto algoritmo da utilizzare[8] fra i molti a disposizione, motivo per cui appare di fondamentale importanza il processo di rilevamento linguistico cui si accennava

[23] Algolia si propone su questo come alternativa a Lucene[7]

poc'anzi. Tuttavia, anche supponendo di essere in grado di discriminare la lingua di un documento (o campo), il processo di rilevamento linguistico non si rivela particolarmente immediato per quanto riguarda la query, che è solitamente composta da un numero di parole decisamente esiguo. É importante che la catena operazionale applicata ai *token* in fase di indicizzazione e interrogazione sia compatibile, pertanto se documenti e query venissero sottoposti a processi linguistici differenti il *matching* ne risulterebbe fortemente alterato.

In funzione di questa considerazione si è deciso, per Arkivium, di omettere ciascun tipo di rilevamento linguistico in favore di una soluzione che potesse rivelarsi adeguata per la maggior parte delle situazioni. In particolare, si è considerato che la stragrande maggioranza dei documenti con cui il software si trova a che fare sono in lingua italiana e inglese, perciò si è pensato di tentare un approccio insolito, di doppia analisi: ciascun termine è stato sottoposto prima allo *stemming* italiano e successivamente a quello inglese, entrambi implementati da algoritmi poco invasivi. Ciò si è tradotto nell'utilizzo di ben 6 `Filter` consecutivi, mostrati nel blocco di codice 3.12 e successivamente approfonditi. Anche la rimozione delle *stop word* è stata condotta con approccio bi-lingue, raggruppando le parole tipiche dei due linguaggi nel medesimo file di testo.

L'**espansione dei termini** richiede invece l'utilizzo di opportuni dizionari e tesauri linguistici; nonostante ne esistano alcune versioni pubblicamente disponibili, non sempre si rivelano adatte alle esigenze di ricerca poiché molte relazioni di *quasi-sinonimia* fra termini sono fortemente dipendenti dal contesto applicativo e sarebbe pertanto opportuno che tali risorse venissero costruite *ad hoc* per ciascuna situazione.

Per il caso di studio in esame si è pensato di semplificare il problema, limitandosi ad utilizzare un unico file di testo che sarà adibito a contenere tutti i sinonimi più importanti delle lingue italiano e inglese. In futuro verrà data la possibilità all'utente di inserire nel file i propri sinonimi, affinché l'elenco risulti personalizzato rispetto alle esigenze del contesto nel quale il motore di ricerca deve essere di volta in volta utilizzato.

Matching parziale dei termini

Come precedentemente accennato, il *matching* di Lucene si basa sul **confronto esatto** dei termini. Ciò significa che se per esempio un utente cerca la parola *consulting* nel campo *ragione sociale* di un'`Azienda`, normalmente non verranno reperite anche le `Aziende` che si chiamano ad esempio *ConsultingPro*, *srlconsulting* o *12consulting34*. Per risolvere questo problema si è proceduto in due modi.

Innanzitutto si è introdotto un filtro che si occupi di separare i termini che contengono caratteri speciali non trattati dallo `StandardTokenizer` (ad esempio il punto), lettere maiuscole (notazione *PascalCase*) e numeri. Ciò è implementato in Solr dal `WordDelimiterGraphFilter`, che si rivela particolarmente utile per la ricerca di indirizzi email, siti web o nomi dei file.

Successivamente si è pensato di abilitare il vero e proprio ***matching* parziale** attraverso un'ulteriore suddivisione di ciascun *token* in sequenze di caratteri che Solr definisce **N-Grams** e costituiscono tutte le possibili combinazioni di *prefissi*, *suffissi* e *infissi* del *token* stesso. Tale operazione consente al sistema di *matchare* tutti i termini che **contengono** una determinata stringa, come avviene per l'operatore `LIKE` del linguaggio SQL. L'`NGramFilter`[24] è da applicare solo in fase di indicizzazione, perché se utilizzato anche durante l'analisi della query è causa di innumerevoli falsi positivi.

Presentazione del nuovo schema

La catena di analisi testuale predisposta nella prima versione dello schema Solr, per comodità riportata nel listato 3.11, è stata completamente rivista per adattarsi alle nuove considerazioni.

```
<fieldType name="text_general" class="solr.TextField">
  <analyzer type="index">
    <tokenizer class="solr.StandardTokenizerFactory"/>
    <filter class="solr.StopFilterFactory" words="stopwords.txt" ignoreCase="true"/>
    <filter class="solr.LowerCaseFilterFactory"/>
  </analyzer>
  <analyzer type="query">
    <tokenizer class="solr.StandardTokenizerFactory"/>
    <filter class="solr.StopFilterFactory" words="stopwords.txt" ignoreCase="true"/>
    <filter class="solr.SynonymGraphFilter" expand="true" synonyms="synonyms.txt"/>
    <filter class="solr.LowerCaseFilterFactory"/>
  </analyzer>
</fieldType>
```

Listato 3.11: `schema.xml` - prima procedura prevista per l'analisi del testo

Il blocco 3.12 riporta invece il nuovo schema.
Per quanto riguarda l'analisi predisposta per la **fase di indicizzazione**, il `Tokenizer` e il processo di rimozione delle *stop word* sono rimasti inalterati, ma prima del *lowercasing* è stato inserito il filtro per la separazione avanzata dei *token* (`WordDelimiter`). Seguono due filtri per proteggere (`KeywordMarker`) alcune parole scelte arbitrariamente (contenute nel file `protwords.txt`) dallo *stemming* e preservare i *token* originali (`KeywordRepeat`) prima dello *stemming* in italiano (`Elision` + `ItalianLightStem`) e poi in inglese (`EnglishMinimalStem`). A questo punto vengono rimossi i *token* duplicati (`RemoveDuplicatesToken`), cioè quelli non affetti dallo *stemming* che risultano doppi in seguito all'applicazione del `KeywordRepeatFilter`. Per finire si suddividono ulteriormente i *token* per abilitare il *matching* di parole parziali (`NGram`).
All'analisi effettuata durante l'**interrogazione** viene aggiunto il `SynonymGraphFilter` per l'espansione della query attraverso i sinonimi ed è stato omesso l'`NGramFilter`, che altrimenti sarebbe causa di numerosi risultati errati.

[24] `https://lucene.apache.org/solr/guide/7_4/filter-descriptions.html#n-gram-filter`

```
<fieldType name="text_general" class="solr.TextField" positionIncrementGap="100">
<analyzer type="index">
  <tokenizer class="solr.StandardTokenizerFactory"/>
  <filter class="solr.StopFilterFactory" words="stopwords.txt" ignoreCase="true"/>
  <filter class="solr.WordDelimiterGraph" catenateAll="1" preserveOriginal="1"/>
  <filter class="solr.FlattenGraphFilterFactory" /> <!-- required after graph -->
  <filter class="solr.LowerCaseFilterFactory"/>
  <filter class="solr.KeywordMarkerFilterFactory" protected="protwords.txt" />
  <filter class="solr.KeywordRepeatFilter" /> <!-- duplicate before stemming -->
  <filter class="solr.ElisionFilterFactory" articles="lang/contractions_it.txt"/>
  <filter class="solr.ItalianLightStemFilterFactory"/>
  <filter class="solr.EnglishMinimalStemFilterFactory"/>
  <filter class="solr.RemoveDuplicatesTokenFilterFactory"/>
  <filter class="solr.NGramFilterFactory" minGramSize="2" maxGramSize="255"/>
</analyzer>
<analyzer type="query">
  <tokenizer class="solr.StandardTokenizerFactory"/>
  <filter class="solr.SynonymGraphFilter" synonyms="synonyms.txt" expand="true"/>
  <filter class="solr.StopFilterFactory" words="stopwords.txt" ignoreCase="true"/>
  <filter class="solr.WordDelimiterGraph" catenateAll="1" preserveOriginal="1"/>
  <filter class="solr.LowerCaseFilterFactory"/>
  <filter class="solr.KeywordMarkerFilterFactory" protected="protwords.txt" />
  <filter class="solr.KeywordRepeatFilterFactory" />
  <filter class="solr.ElisionFilterFactory" articles="lang/contractions_it.txt"/>
  <filter class="solr.ItalianLightStemFilterFactory"/>
  <filter class="solr.EnglishMinimalStemFilterFactory"/>
  <filter class="solr.RemoveDuplicatesTokenFilterFactory"/>
</analyzer>
</fieldType>
```

Listato 3.12: `schema.xml` - nuovo processo di analisi del testo

3.5.2 Un calcolo troppo complesso

Questo tipo di aggiornamenti allo schema richiedono una reindicizzazione. Durante l'esecuzione del processo batch sviluppato per tale scopo ci si è accorti che il server Solr non aveva a disposizione abbastanza risorse per gestire un'analisi testuale così complessa; dai file di log ci si è accorti che il problema era dovuto alla mancanza di spazio nell'*heap*, causa di repentini *overflow*. Documentandosi online[9] si è tentato di risolvere il problema dedicando più RAM al server, ma questa soluzione non si è rivelata adatta a ridurre l'uso intensivo di risorse per intervalli di tempo davvero importanti.

Si è quindi pensato di alleggerire il processo di analisi e per farlo ci si è affidati allo strumento di analisi messo a disposizione dall'interfaccia web di Solr[25]. Facendo alcuni test si è scoperto che il numero di *token* generati per una query basilare era davvero

[25] `https://lucene.apache.org/solr/guide/7_4/analysis-screen.html`

imponente - nell'ordine delle centinaia - e si è compreso che moltissime delle operazioni effettuate dai *filters* erano superflue.

Infatti, ripercorrendo i propri passi si è notato che l'inserimento di un `NGramFilter` al termine dell'intera analisi linguistica rendeva di fatto inutile l'intero processo di separazione dei *token* e *stemming* precedentemente affrontato: se di ciascun *token* vengono computati tutti i prefissi, è ovvio che *stemmare* i termini si rivela superfluo in fase di indicizzazione, oltre che un enorme spreco di risorse.

3.5.3 Ottimizzazione dello schema - versione finale

Per i motivi poc'anzi elencati si è deciso di ottimizzare e semplificare lo schema, la cui nuova versione è mostrata nel listato 3.13.

Nello specifico si è pensato di rimuovere completamente dalla **fase di indicizzazione** i filtri relativi all'analisi linguistica, lasciando svolgere la maggior parte delle operazioni all'`NGramFilter` che consente di abilitare il *matching* parziale delle parole. Anche il `WordDelimiterGraphFilter` si è rivelato superfluo ed è stato rimosso.

Per quanto riguarda invece la **fase di interrogazione**, suddivisione avanzata dei *token* e *stemming* si sono rivelati necessari per sopperire alla mancanza dell'`NGramFilter`, che se fosse aggiunto alla catena causerebbe gravi errori nella ricerca. Come ultima operazione, prima del *lowercasing* si è aggiunta la gestione dei sinonimi; è opportuno che tali sinonimi vengano sottoposti ai medesimi algoritmi di *stemming* prima di essere inseriti all'interno del file `synonyms.txt`, affinché l'espansione della query possa svolgersi nella maniera corretta. In futuro verrà eventualmente valutata una composizione più adeguata, sia dal punto di vista funzionale che prestazionale.

```
<fieldType name="text_general" class="solr.TextField" positionIncrementGap="100">
  <analyzer type="index">
    <tokenizer class="solr.StandardTokenizerFactory"/>
    <filter class="solr.StopFilterFactory" words="stopwords.txt" ignoreCase="true"/>
    <filter class="solr.LowerCaseFilterFactory"/>
    <filter class="solr.NGramFilterFactory" minGramSize="2" maxGramSize="60"/>
  </analyzer>
  <analyzer type="query">
    <tokenizer class="solr.StandardTokenizerFactory"/>
    <filter class="solr.StopFilterFactory" words="stopwords.txt" ignoreCase="true"/>
    <filter class="solr.WordDelimiterGraphFilterFactory" preserveOriginal="1"/>
    <filter class="solr.ItalianLightStemFilterFactory"/>
    <filter class="solr.EnglishMinimalStemFilterFactory"/>
    <filter class="solr.SynonymGraphFilter" synonyms="synonyms.txt" expand="true"/>
    <filter class="solr.LowerCaseFilterFactory"/>
  </analyzer>
</fieldType>
```

Listato 3.13: `schema.xml` finale che descrive il processo di analisi testuale ottimizzato

Nuova dimensione degli indici

L'analisi testuale della fase di indicizzazione determina il numero di termini prodotti e indicizzati. L'introduzione di un `NGramFilter`, seppur essenziale per ottenere risultati di ricerca appropriati, provoca una massiccia espansione del numero di termini da indicizzare e influisce considerevolmente sulla dimensione degli indici inversi.

La conseguenza di questa scelta sono degli indici decisamente più esigenti di spazio rispetto a quelli generati dalla prima versione dello schema, che risultava di soli 400 MB in seguito alla memorizzazione (attributo `stored`) del contenuto testuale estratto dai file all'interno dell'apposito campo `content`. Nello specifico, si parla di ben **10 GB** di indici **non ottimizzati** su un totale di 65mila documenti, di cui si memorizza tutto. Ottimizzare gli indici è un processo che richiede un'enorme quantità di risorse, in particolare lettura molto rapida su disco e RAM sufficientemente capiente; per limitazioni tecniche dovute alla macchina su cui si sono effettuati i test[26] non è stato possibile ottimizzare gli indici e valutarne l'occupazione effettiva, che si stima potrebbe aggirarsi intorno alla metà dello spazio occupato dai medesimi indici non ottimizzati.

[26]Notebook con Intel Core i7-6700HQ, 8GB RAM, HDD 5400rpm

3.6 Sviluppi futuri

Il motore di ricerca sviluppato si è rivelato un ottimo prototipo, che sarà soggetto ad ulteriori miglioramenti nel corso dei mesi successivi:

- Verrà approfonditamente valutato ed eventualmente migliorato il processo di analisi linguistico predisposto nello schema. Inoltre si darà all'utente la possibilità di inserire nel sistema i propri sinonimi.
- L'interfaccia subirà un importante miglioramento grafico e verranno introdotte ulteriori funzionalità per integrare le principali operazioni usualmente portate a termine tramite Arkivium.
- Verrà introdotta la possibilità di effettuare ricerche geo-spaziali, si migliorerà il supporto alle *range query* e verranno introdotti *range faceting*.
- Verranno aggiunti ulteriori campi su cui poter cercare, ricavandoli dal database; si pensa di aumentare anche la quantità dei *tipi* di documento ricercabili, importando in Solr altre tabelle della base di dati.
- Verrà implementato un metodo per il salvataggio delle ricerche effettuate, per poter usufruire ripetutamente della medesima query ed eventualmente condividere una ricerca con i propri colleghi.
- Si proverà ad inserire la funzionalità *MoreLikeThis*, probabilmente per le `Persone`.
- Si potrebbe migliorare la ricerca sulle `Aziende`, integrando informazioni reperite dal Web tramite appositi *crawler*, ad esempio utilizzando Apache Nutch[27].
- Si proverà ad inserire degli *agent* per categorizzare automaticamente le `Persone` attraverso le informazioni contenute nei *curriculum*.
- Il client .NET sviluppato per l'interazione con Solr verrà adeguatamente rimodellato e astratto affinché possa esserne ricavata una libreria riutilizzabile dall'azienda anche all'interno di altri progetti.

[27] `http://nutch.apache.org/`

Conclusioni

Allo stato attuale di sviluppo del motore di ricerca si può affermare che i risultati siano soddisfacenti e si stima che la nuova funzionalità sarà sicuramente apprezzata da parte degli utenti che utilizzano quotidianamente Arkivium. I pareri di quest'ultimi verranno tenuti in considerazione per valutare l'efficacia del sistema e, nei mesi successivi al rilascio, seguiranno eventuali raffinamenti per migliorare l'attendibilità dei risultati ottenuti da un'interrogazione. Come accennato nella sezione degli sviluppi futuri, verranno implementate nuove funzionalità per migliorare sia la ricerca che l'integrazione con le funzionalità di Arkivium già esistenti.

A valle del lavoro svolto è sicuramente interessante notare che, nonostante Solr si prefigga l'obiettivo di rendere trasparente l'infrastruttura e il funzionamento sottostante, ciò sia vero solamente per quanto riguarda l'utilizzo degli indici inversi e dei modelli matematici che consentono di effettuare il *matching* fra query e documenti; tuttavia, non si dimostra altrettanto immediata la progettazione dell'**analisi** da applicare al testo né le modalità di ***boosting*** dei vari campi che costituiscono i documenti. Queste sono due caratteristiche di cui il progettista del sistema deve occuparsi personalmente e che richiedono un certo livello di esperienza affinché i risultati presentati siano coerenti con le reali esigenze dell'utente; per questo motivo rappresentano anche i primi parametri sui quali si agirà in futuro qualora si riscontrassero risultati di ricerca non pienamente soddisfacenti. Un ulteriore margine di miglioramento lo si potrà ottenere tramite la gestione dei termini attraverso l'espansione dei *quasi-sinonimi*, sfruttando opportuni dizionari linguistici o soluzioni personalizzate per ciascun cliente.

In futuro si pensa che alcune funzionalità di *enterprise search* verranno estese anche ad altri software dell'azienda, sfruttando le competenze apprese nell'utilizzo di Apache Solr o sperimentando piattaforme analoghe, come potrebbero essere per esempio ElasticSearch[28] o Algolia[29]. Infatti, appresi i principali concetti di Information Retrieval e viste le esigenze e le problematiche che scaturiscono dalla progettazione di un motore di ricerca, potrebbe emergere - a seguito di un accurato confronto - che altri strumenti potrebbero essere più adatti di Apache Solr a soddisfare alcune specifiche necessità funzionali o architetturali.

[28] `https://www.elastic.co/`
[29] `https://www.algolia.com/`

Bibliografia

[1] Prabhakar Raghavan e Hinrich Schütze Christopher D. Manning. *Introduction to Information Retrieval*. Cambridge University Press, 2008.

[2] Erik Hatcher Michael McCandless and Otis Gospodnetić. *Lucene in Action, Second Edition*. Manning Publications, 2010.

[3] Trey Grainger and Timothy Potter. *Solr in Action*. Manning Publications, 2014.

[4] *Apache Solr Reference Guide (version 7.4)*. URL `https://lucene.apache.org/solr/guide/7_4/index.html`.

[5] Dati delle ricerche Google a livello mondiale. 2017. URL `https://www.matteogiovanelli.it/dati-google-a-livello-mondiale.html`.

[6] BM25 the next generation of Lucene relevance. URL `https://opensourceconnections.com/blog/2015/10/16/bm25-the-next-generation-of-lucene-relevation/`.

[7] Algolia for consumer-grade search part 2: Relevance isn't luck. URL `https://blog.algolia.com/algolia-v-elasticsearch-relevance/`.

[8] Elasticsearch. *Choosing a Stemmer*. URL `https://www.elastic.co/guide/en/elasticsearch/guide/current/choosing-a-stemmer.html`.

[9] Apache Foundation. *Solr Performance Problems*. URL `https://wiki.apache.org/solr/SolrPerformanceProblems`.