Alma Mater Studiorum Università di Bologna
Campus di Cesena

Scuola di Ingegneria e Architettura
Corso di Laurea Magistrale in Ingegneria e Scienze Informatiche

Apache Solr: applicazioni di NLP per l'estrazione della conoscenza dal testo e dal codice sorgente

Relatore:
Prof. Andrea Omicini

Presentata da:
Nicola Atti

Terza Sessione di Laurea
Anno Accademico 2022-2023

A tutti miei cari
che mi hanno sostenuto
fino a questo traguardo

Introduzione

I motori di ricerca svolgono un ruolo importante nel recupero delle informazioni, in quanto sono ormai lo strumento preferito dagli utenti per ricercare e gestire le informazioni desiderate. La ricerca tramite parole chiave è il paradigma di ricerca più popolare, che richiede all'utente di cercare all'interno di un intero repository sulla base di poche parole che riassumono le informazioni desiderate.

La disponibilità di repository di software open source continua a crescere, e dunque anche la necessità di strumenti in grado di analizzarli automaticamente su scala sempre più grande. L'analisi automatizzata di tali repository risulta importante, ad esempio per comprendere la struttura del software, le sue funzioni, complessità ed evoluzione, nonchè per identificare le relazioni tra gli esseri umani e il software che producono, facilitando il riutilizzo interno del codice ed il processo di refactoring.

Queste relazioni possono essere incapsulate da parole chiave e tendono a corrispondere a concetti o caratteristiche implementati dal software e sono strettamente correlati alle classi dei sistemi software. Usando questi topic come guida, uno sviluppatore può essere in grado di capire meglio e più rapidamente la struttura dei sistemi, che potrebbe non essere riflessa dal gerarchia dei pacchetti o dalla documentazione.

Lo scopo di questa tesi è di esplorare l'utilizzo di tecnologie, quali Apache Solr e Apache Spark, per effettuare il lavoro di analisi ed estrazione dei topic. In primo luogo verrà esaminato il caso d'uso del testo in linguaggio naturale, in particolare andando ad esaminare le note delle consuntivazioni

all'interno dei progetti sviluppati in azienda, e successivamente il caso d'uso del codice sorgente, andando ad esaminare direttamente il codice di progetti open-source assieme al contenuto dei messaggi di commit.

Indice

Elenco delle figure

Elenco delle tabelle

Listings

Capitolo 1

Tecnologie utilizzate

1.1 Apache Solr

In questa prima sezione sarà preso in esame Apache Solr, una piattaforma di ricerca open-source ampiamente usata per enterprise-search e analytics, andando a descrivere il funzionamento della tecnologia e delle sue funzionalità principali, soffermandoci maggiormente sugli elementi utilizzati nei nostri casi d'uso, ed esaminando nel dettaglio il concetto della *relatedness* per la scoperta delle relazioni tra i concetti indicizzati.

1.1.1 Struttura di solr e metodi di deploy

Solr permette di effettuare il deploy in due modi:

- Standalone: una versione semplice, già compresa della propria istanza di Zookeeper. Semplice e veloce da configurare, ma con limitate capacità di scalabilità e robustezza per casi d'uso di produzione. L'indice sarà formato da un solo nodo che conterrà tutti i singoli core.

- SolrCloud: una versione predisposta alla realizzazione di un cluster di server Solr per fornire fault-tolerance e availability. Richiede la definizione di istanze di Apache Zookeeper a cui il cluster si appoggerà per

gestire i nodi e distribuire il carico automaticamente. L'indice avrà una struttura distribuita.

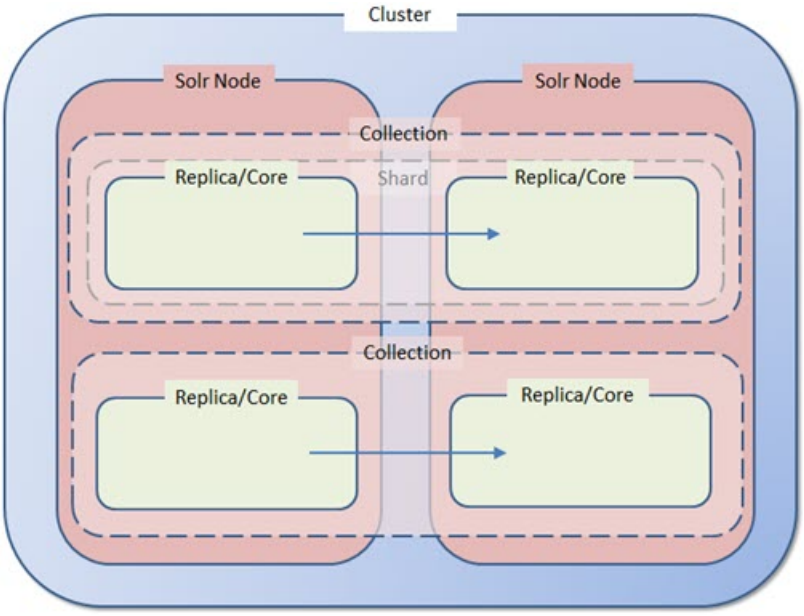

Figura 1.1: La struttura del cluster di SolrCloud. Ogni nodo contiene un core rappresentante un singolo indice, che assieme vanno a formare una collezione.

Un cluster SolrCloud è coposto da concetti logici definiti sulla base di concetti fisici, come descritto in figura 1.1. Un cluster può contenere multiple collezioni di documenti, queste collezioni possono essere ripartite in multipli *Shard* che a loro volta contengono sottoinsiemi di documenti della collezione. Il numero di shard di una collezione determina il limite teorico di documenti che questa può ragionevolmente contenere e quanto sia possibile parallelizzare una singola richiesta.

A livello fisico un cluster è composto da *Nodes* che effettivamente eseguono le istanze dei processi server di Solr. Ognuno di essi è composto da *Core*, cioè una replica fisica di uno shard logico, il cui numero determina il livello di ridondanza della collezione e quanto il cluster sia resistente ai guasti, nel caso della perdita di uno dei nodi, ed il limite delle richieste gestibili in concorrenza dal cluster sotto alto carico.

SolrCloud si appoggia ad Apache Zookeeper per le operazioni di coordinazione tra i membri del cluster. Per il corretto funzionamento viene richiesta la definizione di un ensemble esterno di server Zookeeper, e per la nostra implementazione del cluster si è deciso di utilizzare un ensemble composto da tre server, in quanto è il numero minimo di server per garantire il corretto funzionamento del sistema di elezione e raggiungimento del consenso.

Per effettuare il processo di elezione ogni server partecipante genera un un nodo effimero a cui viene assegnato un id sequenziale. Il nodo con la sequenza più bassa viene selezionato come leader, mentre gli altri si metteranno in ascolto sul prossimo nodo con la sequenza più bassa, creando effettivamente una linked list di nodi. Quando il nodo leader cade, gli altri nodi cercheranno un altro nodo con un numero di sequenza minore, nel caso non esistesse diventano loro stessi il nuovo leader.

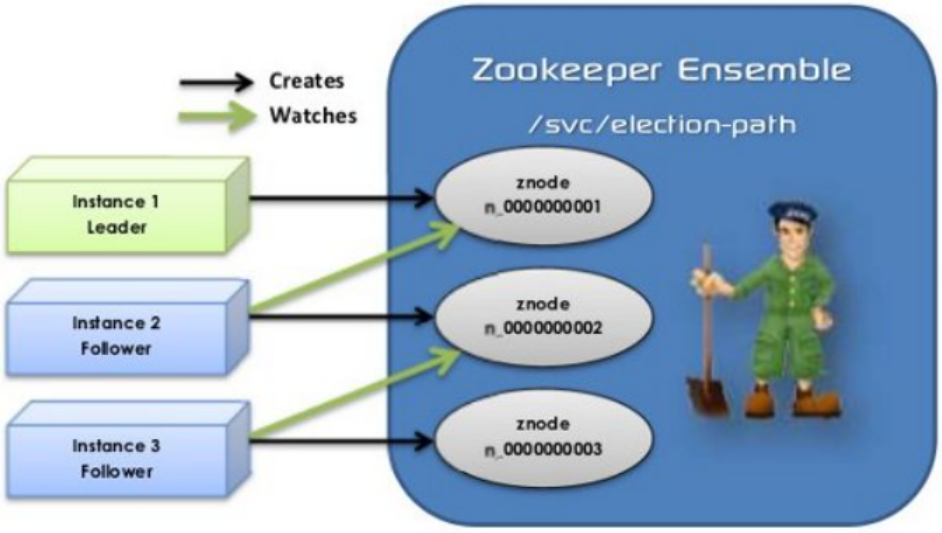

Figura 1.2: La figura mostra schematicamente il processo di generazione delle sequenze dei singoli nodi per il meccanismo di elezione del leader.

SolrCloud non è composto da nodi master o slave, invece le repliche fisiche degli shard possono essere elette come leader utilizzando un sistema basato sul consenso come per Zookeeper. Alla perdita di un leader un'altra delle

repliche sarà automaticamente eletta come sostituto. Quando un documento viene inviato al cluster per essere indicizzato, il sistema per prima cosa determina a quale shard esso appartiene, e poi quale nodo stia attualmente ospitando il leader di tale shard. Il documento verrà inoltrato al leader corrente che lo indicizzerà, per poi propagare l'update a tutte le altre repliche.

Ogni documento all'interno di una collezione viene immagazzinato all'interno di un singolo shard, definito dalla strategia di sharding implementata dalla collezione. Quando un nodo riceve una richiesta, questa viene dirottata ad una replica di uno degli shard che fa parte della collezione che viene interrogata. La replica funge da aggregatore, creando richieste ad altre repliche di shard della collezione scelte casualmente, facendosi carico di coordinare le risposte e di effettuare eventuali richieste locali per processare i risultati, per poi comporre la risposta finale per il client.

Per il nostro caso d'uso si è deciso di realizzare tre nodi SolrCloud su tre macchine separate (le stesse che ospitano l'ensemble Zookeeper) e per ogni collezione si è optato per una suddivisione in due shard, che a loro volta saranno suddivisi in due core.

Autenticazione e autorizzazione

Solr rifiuterà di utilizzare configurazioni caricate senza aver prima aver configurato le funzionalità di sicurezza. Per poter usufruire delle API nelle istanze realizzate tramite SolrCloud è necessario preparare il sistema all'utilizzo di un metodo di autenticazione ed autorizzazione. Solr mette a disposizione diversi metodi, configurabili tramite l'upload di specifici file di configurazione all'interno delle istanze Zookeeper su cui fa appoggio. Si è deciso di abilitare il protocollo di SimpleAuthentication, permettendo di autenticarsi tramite nome utente e password.

1.1.2 L'indice Solr

La funzione base di Solr è quella di immagazzinare grandi quantità di informazioni, per poi permettenre facilmente il recupero tramite operazio-

ni appositamente strutturate. L'operazione con la quale vengono inserite le informazioni è detta *Indexing* o indicizzazione, mentre l'operazione definita per la realizzazione delle richieste per il recupero delle informazioni viene chiamata *Query*. L'unità di informazione alla base dell'indice Solr è il *Document*, che in sostanza è un insieme di dati che rappresenta un concetto. Ogni document è poi composto da *Field* che vanno a definirne informazioni specifiche di varia natura sulla base del *field type* definito.

Operazione di Indexing

Quando del contenuto viene aggiunto all'indice tramite un'operazione di indexing, lo rendiamo ricercabile tramite Solr. Per caricare dati all'interno dell'indice, Solr mette a disposizione diversi metodi:

- Tramite linea di comando.
- Tramite Api REST.
- Usando librerie per la realizzazione di client, come ad esempio SolrJ.

Normalmente Solr opera immagazzinando i documenti in formato JSON, ma permette l'upload di file anche in altri formati come XML e CSV o file di testo come PDF, Word, HTML, etc.

Solr supporta anche l'indicizzazione di documenti innestati e offre metodi per effettuarne la ricerca in modo efficiente. Questo tipo di documenti è composto da documenti *parent* e *children* che assieme vanno a comporre un blocco. Come esempio di documenti innestati si possono prendere i post presenti sui social network, in cui il contenuto del post stesso può essere indicizzato come docuemnto parent ed i commenti possono essere considerati come documenti children. L'utilizzo di questa struttura e dei metodi messi a disposizione da Solr per la loro interrogazione li rende una struttura più efficiente per gestire relazioni tra elementi, rispetto ai *query time join*, ma risulta meno flessibile.

Operazione di Search

Quando un utente effettua una operazione di ricerca in Solr la query viene processata da un plug-in, chiamato *request handler*, che definisce la logica da usare per l'elaborazione della richiesta. Solr supporta vari request handler, alcuni progettati per l'elaborazione di query di ricerca, mentre altri si occupano di gestire incarichi come ad esempio la replicazione dell'indice. Questi handler si avvalgono di *query parser* per interpretare i termini e i parametri delle query.

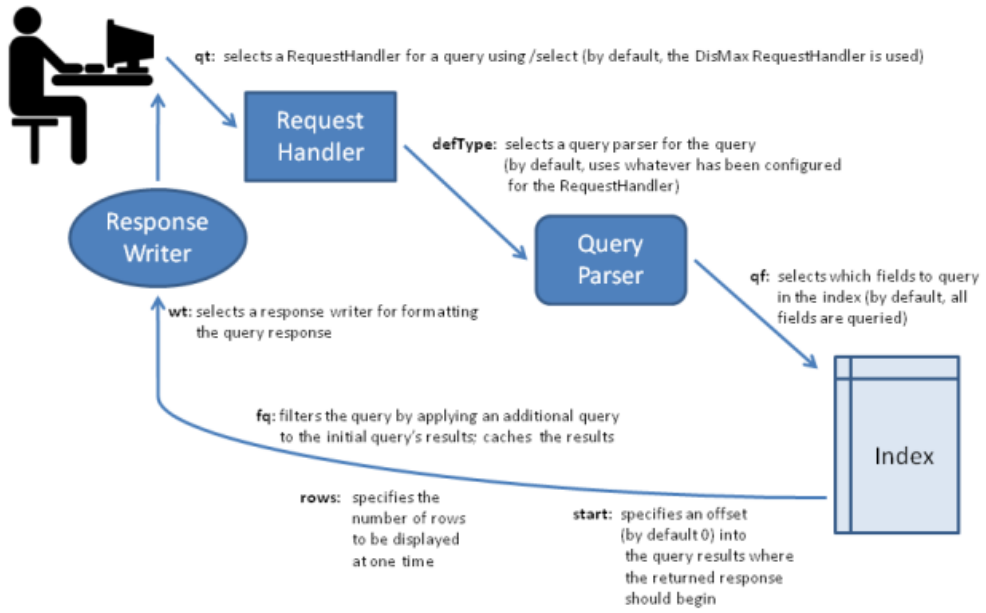

Figura 1.3: Flusso di controllo per l'esecuzione di una query di ricerca in Solr.

Il meccanismo di ricerca di Solr è molto flessibile e mette a disposizione diversi parametri per poter effettuare il fine-tuning delle query. Queste *filter query* eseguono richieste sull'intero indice ed inseriscono i risultati all'interno della cache. Oltre ad essere utili per poter filtrare i risultati delle richieste, l'utilizzo di questi filtri permette anche di migliorarne le performance delle ri-

cerche grazie all'utilizzo del meccanismo di caching. Il flusso per l'esecuzione di una query di ricerca viene illustrato nel dettaglio in figura 1.3.

Oltre al meccanismo di ricerca base è possibile anche effettuare ricerche di tipo *faceting*, in cui viene effettuato un accorpamento dei risultati per categorie. All'interno di ogni categoria Solr riporta il numero di occorrenze per ogni termine. Esistono diversi tipi di faceting principali:

- Facet che partizionano e categorizzano i dati all'interno di multipli *bucket*, come i facet di tipo *term* e *range*. La prima tipologia raggruppa il i risultato sulla base dei valori univoci presenti in un campo, mentre il secondo produce più bucket su date o campi numerici.

- Facet che calcolano dati per un singolo bucket, normalmente una metrica o una funzione statistica o analitica. Uno di questi è il tipo *query* che produce un singolo bucket di documenti che corrispondono sia al dominio che alla query specificata, e l'altro è il tipo *heatmap* che genera una griglia 2D di conteggi di facet per i documenti che contengono dati spaziali in ogni cella della griglia.

Per ottenere ulteriori raggruppamenti Solr permette anche la definizione di *nested facets* che permettono di innestare comandi all'interno di altri. Questi sotto-facet sono poi valutati sui domini definiti dall'insieme di tutti i documenti all'interno di ogni bucket definito dalla query genitore.

Cursori e paginazione

Molte applicazioni necessitano di ottenere questi risultati in pagine di una grandezza ben definita, tramite la tecnica della paginazione. In Solr una forma di paging di base è supportata utilizzando i parametri *start* e *rows* del meccanismo di realizzazione delle query. Il parametro *start* specificato in una richiesta a Solr indica un "offset" assoluto, all'interno dell'elenco ordinato di corrispondenze, che il client desidera che Solr utilizzi come inizio della pagina corrente. Il parametro *rows* invece indica la dimensione effettiva della pagina.

Questo semplice sistema ha però delle limitazioni. Nel caso si verifichi una modifica dell'indice (come l'aggiunta o la rimozione di documenti) che influisce sulla sequenza di documenti ordinati che corrispondono a una query tra due richieste per pagine successive di risultati, allora è possibile che queste modifiche possano comportare che lo stesso documento venga restituito su più pagine o vengano ignorati dei documenti. Inoltre quando si deve recuperare un numero molto elevato di risultati ordinati da Solr, l'utilizzo di valori molto grandi per i parametri start o rows può essere molto inefficiente. L'impaginazione che utilizza start e rows non solo richiede a Solr di calcolare (e ordinare) in memoria tutti i documenti corrispondenti che dovrebbero essere recuperati per la pagina corrente, ma anche tutti i documenti che sarebbero apparsi nelle pagine precedenti. Questo risulta particolarmente problematico nel caso si abbia a che fare con un indice distribuito (nel caso si utilizzi SolrCloud), dove per ogni shard vengono recuperati tutti i documenti che corrispondono ai parametri usati per la query.

Come alternativa Solr supporta l'utilizzo di *Cursori* per eseguire la scansione dei risultati. I Cursori in Solr sono un concetto logico che non necessita la memorizzazione nella cache di alcuna informazione di stato sul server. Invece i valori di ordinamento dell'ultimo documento restituito al client (chiamati *cursorMark*) vengono utilizzati per calcolare un "segn" che rappresenta un punto logico nello spazio ordinato dei valori di ordinamento. In questo modo viene essenzialmente creata una linked list di documenti, dove il parametro *nextCursorMark* punta al documento successivo (o allo stesso documento in caso di conclusione della lista dei risultati).

Poichè le richieste effettuate tramite Cursori sono stateless e i valori cursorMark incapsulano i valori di ordinamento assoluti dell'ultimo documento restituito da una ricerca, è possibile continuare a recuperare risultati aggiuntivi da un cursore che ha già raggiunto la fine, nel caso vengonano aggiunti nuovi documenti (o aggiornati documenti esistenti).

1.1.3 Semantic Knowledge Graph

Solr permette di fare leva sui documenti all'interno dell'indice per poter individuare e valutare la forza delle relazioni semantiche tra entità. Per far ciò offre funzionalità per la realizzazione di un knowledge graph, chiamato *Semantic Knowledge Graph* o SKG, che facendo leva su un indice invertito, assieme ad un indice non invertito complementare, rappresentanti i nodi (termini) e gli edge (i documenti con liste di messaggi intersecanti per multipli termini/nodi), è in grado di scoprire e valutare relazioni tra qualsiasi combinazione di entità (parole, frasi o concetti estratti) attraverso la materializzazione dinamica di nodi ed edge per formare una rappresentazione grafica automaticamente a partire da un corpo di dati rappresentativi del dominio della conoscenza. In questa sezione verrà descritto l'SKG e come Solr lo utilizza per fornire la metrica della *relatedness*.

Il linguaggio naturale, nella forma di documenti di testo, contiene un'enorme quantità di significato tra le strutture linguistiche, rappresentato da molteplici livelli di astrazione:

- Corpo (Corpus): una lista di documenti rappresentativa del dominio della conoscenza.
- Documento (Document): una lista di campi collegati fra loro attraverso un'entità sottostante.
- Campo (Field): un gruppo di zero o più sequenze di termini rappresentativa di una relazione all'interno di un documento.
- Sequenza di termini (Term Sequence): una rappresentazione ordinata di uno o più termini.
- Termine (Term): una sequenza di caratteri che rappresenta un significato conosciuto (ad esempio una parola riconoscibile).
- Sequenza di caratteri (Character Sequence): una combinazione ordinata di uno o più caratteri.

- Carattere (Character): una lettera o simbolo usato nel linguaggio naturale (che non presenta significato di per sé).

Serve un modo per preservare queste interconnesioni semantiche che sono incorporate all'interno di un corpo di documenti di testo.

Consideriamo un grafo non orientato G = (V,E) dove V ed E $\subseteq V \times V$ denotano insiemi di nodi ed edge rispettivamente. Viene definito:

- $D = \{d_1, d_2, ..., d_m\}$ un insieme di documenti che rappresenta il corpo che verrà utilizzato dal SKG per estrarre e valutare le relazioni semantiche.

- $X = \{x_1, x_2, ..., x_k\}$ un insieme di termini contenuti in D. Questi oggetti possono essere parole chiave, frasi o un qualsiasi altra rappresentazione linguistica presente in D.

- $di = \{x | x \in X\}$ dove possiamo pensare ad ogni documento d appartenente a D come ad un insieme di oggetti.

- $T = \{t_1, t_2, ..., t_n\}$ dove ti è un tag che assegna ad un tipo di entità ad un oggetto (come una parola chiave, titolo, luogo, compagnia, scuola, persona etc...)

L'insieme di nodi V nel grafo sarà definito come $V = \{v_1, v_2, ..., v_n\}$, dove vi immagazzina un oggetto $x_i \in X$ taggato con il tag $t_j \in T$. Mentre $Dv_i = \{d | x_i \in d; d \in D\}$ è un insieme di documenti che contiene l'oggetto x_i assieme al suo tag appropriato t_j.

Infine definiamo e_{ij} come un edge tra (v_i, v_j) tramite la funzione $f(e_{ij}) = \{d \in Dv_i \cap Dv_j\}$ che immagazzina su ogni edge l'insieme di documenti che contengono entrambi gli oggetti x_i e x_j assieme ai loro tag. Inoltre possiamo definire $g(e_{ij}, v_k) = \{d : d \in f(e_{ij}) \cap Dv_k\}$ che immagazzina sull'edge e_{jk} l'insieme di documenti in comune tra $f(e_{ij})$ e D_k.

Invece che essere direttamente connessi tra loro tra edge espliciti, i nodi sono connessi bidirezionalmente ai documenti, in questo modo si può dire che

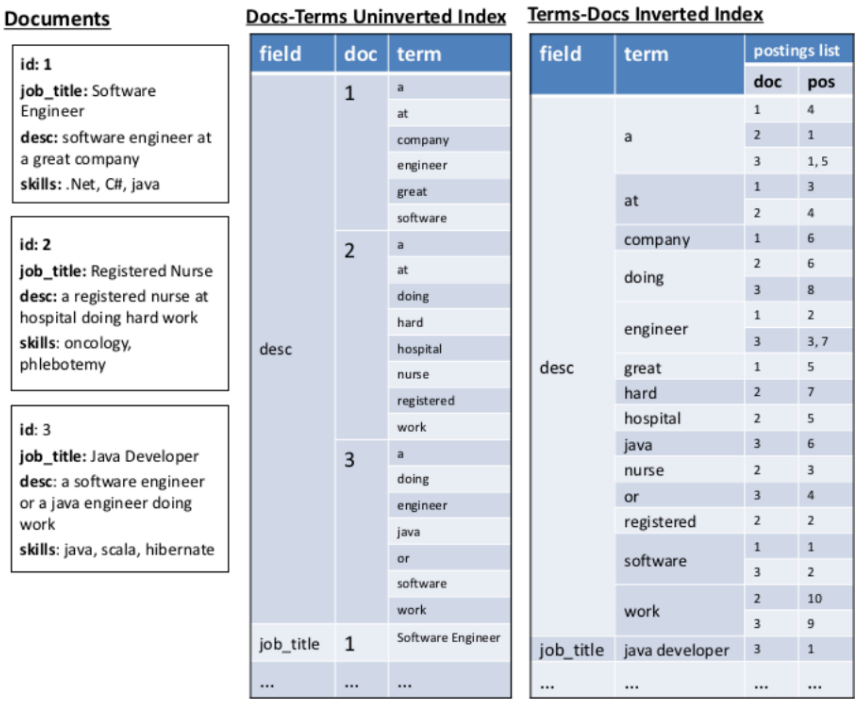

Figura 1.4: I due indici necessari per la realizzazione del Semantic Knowledge Graph.

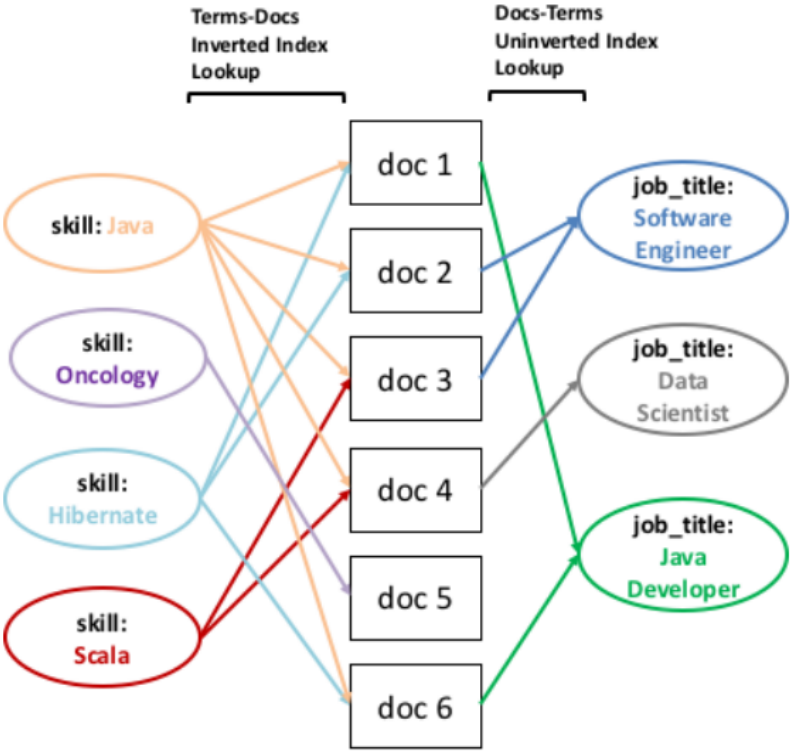

Figura 1.5: Materializzazione degli edge per i termini che hanno almeno un documento in comune.

l'edge e_{ij} tra i nodi v_i e v_j è *materializzabile* quando $|f(e_{ij})| > 0$. Quindi per attraversare il grafico a partire dal nodo sorgente v_i al nodo di destinazione v_j, il nostro sistema richiede un indice di lookup, che colleghi il nodo v_i ad un insieme di documenti, ed anche un indice di lookup separato che mappi questi documenti al nodo v_j o a ogni altro nodo che possa essere necessario per l'attraversamento. Il primo sarà l'indice invertito termini-documenti, mentre il secondo sarà l'indice non invertito documenti-termini. Useremo lo z score per valutare tale ipotesi:

$$z(v_i, v_j) = \frac{y - n * p}{\sqrt{n * p(1-p)}} \tag{1.1}$$

Dove $n = |DFG|$ è il numero di documenti nell'insieme di foreground, $p = \frac{|Dv_g|}{|DBG|}$ è la probabilità di trovare il termine x_j con il tag t_k all'interno dell'insieme di background, e $y = |f(e_{ij})|$ è il numero di documenti contenenti sia x_i e x_j. In molti casi, potremmo voler attraversare multipli livelli di profondità ($n > 2$) per trovare e valutare relazioni tra più di due nodi. Ad esempio, se volessimo attraversare dall'entità "java" a "big data" ad "hadoop", tale che il peso assegnato all'edge tra "big data" e "hadoop" sarebbe più significativo se fosse anche condizionato dal percorso impiegato per arrivare a "big data" tramite "java". Il sistema realizza tutto ciò su n nodi e un percorso $P = \{v_1, v_2, ..., v_n\}$, dove ogni nodo immagazzina un oggetto x_i e il suo tag t_j. Per applicare lo stesso $z(v_i, v_j)$ tra i nodi, ma condizionando il valore in base all'intero percorso P.

$$\begin{cases} f(eij) & if n = 3 \\ \{\bigcap_{i=1,j=i+1,k=j+1}^{n-3} g(e_{ij}, Dv_k)\} & if n > 3 \end{cases} \tag{1.2}$$

Si normalizza infine lo z score usando una funzione sigmoide per portare il valore in un range [1,-1]. Questo valore normalizzato è detto "relatednes" ed il suo valore indica la tipologia di relazione che esiste fra due termini, in particolare:

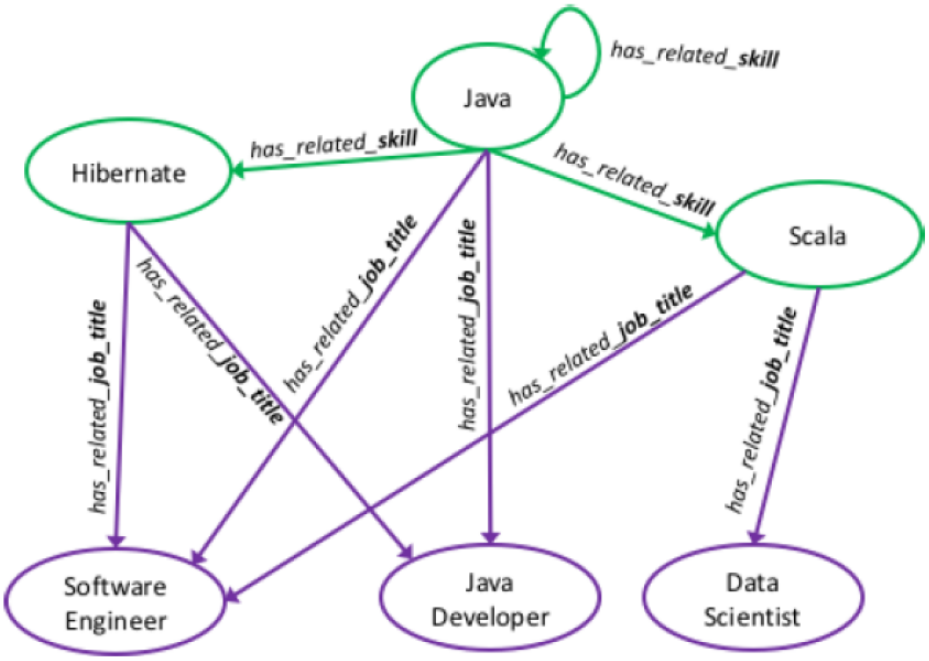

Figura 1.6: Esempio di attraversamento di un Semantic Knowledge Graph.

- Un valore di 1 significa "relazione completamente positiva" che rappressenta una buona probabilità che compaiano sempre insieme.
- Un valore di 0 significa "nessuna relazione" cioè altrettanto probabile che i due termini possano apparire assieme esattamente come ogni altro termine.
- Mentre un valore di -1 significa "completa relazione negativa" e quindi improbabile che compaiano insieme.

Mentre il valore di relatedness fornisce un peso ad ogni edge corrispondende alla forza della relazione semantica tra due nodi, poiché questo punteggio viene calcolato al momento dell'attraversamento, è anche possibile sostituire in altre funzioni di punteggio a seconda del caso d'uso in questione. Per scoprire gli oggetti correlati con un tag specifico t_k ad un oggetto x_i con un tag t_j, si inizia con l'interrogare l'indice invertito per l'oggetto x_i, a cui viene assegnato come nodo v_i, corrispondende con l'insieme di documenti

Dv_i. Viene fatta l'interrogazione sull'indice non invertito documenti-termini per il tag t_k e si immagazzina i documenti ottenuti in $Dt_k = \{d|x \in d, x : t_k\}$.

Si definisce poi $Vv_i, t_k = \{v_j|x_j \in d, d \in Dt_k \cap Dv_i\}$ dove v_j è un nodo che contiene l'oggetto x_j, e definiamo Vv_i, t_k come l'insieme di nodi che contendo oggetti con una relazione potenziale con oggetti x_i di tipo t_k. Infine si applica $\forall v_j \in Vv_i, t_k$ la funzione $relatedness(v_i, v_j)$ per pesare la relazione tra v_i e v_j, che ci permette di classificare tali relazioni e selezionare le prime m o definire una soglia t ed accettare solo le relazioni che la superano. Questa operazione di scoperta delle relazioni può avvenire ricorsivamente.

Chiameremo quindi Dv_i il nostro insieme di documenti di foreground DFG, mentre D sarà il nostro insieme di documenti di background. L'ipotesi alla base della tecnica di valutazione è che se x_i tende ad essere semanticamente legato a x_j, allora la presenza di x_j nell'insieme di foreground dovrebbe essere maggiore della presenza media di x_j nell'insieme di background.

1.1.4 Deploy Cluster SolrCloud

Per il deploy della nostra architettura è stato deciso di definire un cluster SolrCloud composto da tre host, in modo da migliorare le performance di indicizzazione e risoluzione delle query senza però introdurre eccessiva complessità. Per realizzare il cluster per prima cosa è stato necessario definire un ensemble Zookeeper, come definito dalla documentazione, distribuito sugli stessi tre host utilizzati per il cluster SolrCloud. Questo ensemble è composto da tre host, in quanto è il numero minimo di nodi necessari a zookeeper per ottenere un consenso sull'elezione del leader, ciascuno configurato tramite la seguente configurazione:

Infine su ogni host, all'interno della cartella */data* il cui percorso è definito all'interno del file di configurazione, è stato creato il file di testo contente l'id che verrà assegnato all'interno del cluster Zookeeper, necessario per la distinzione dei singoli nodi. Una volta completata la configurazione è bastato eseguire lo script di inizializzazione del cluster *./zkServer.sh* start su tutti i nodi per inizializzare il processo di setup ed elezione del leader. Dopo aver

```
dataDir=/var/lib/zookeeper/data

clientPort:2181

server.1= IPSERVER1:2888:3888
server.2= IPSERVER2:2888:3888
server.3= IPSERVER3:2888:3888
```

Figura 1.7: Configurazione cluster Zookeeper.

constatato il corretto funzionamento del cluster zookeeper è stato possibile eseguire lo script di inizializzazione degli host SolrCloud.

```
bin/solr start -c -z "zookeeper1:2181,zookeeper2:2181,zookeeper3:2181" -noprompt
```

Figura 1.8: Comando per il lancio di un nodo di un cluster SolrCloud, specificando gli indirizzi del cluster Zookeeper. Il parametro -noprompt effettua il comando di lancio con i parametri di default.

Lo script provvederà al setup dell'instanza SolrCloud sull'host interfacciandosi all'ensemble Zookeeper specificato, anzichè appoggiarsi a quello fornito di default da Solr. L'opzione -noprompt indica allo script di non utilizzare l'approccio guidato alla realizzazione degli host, utilizzando direttamente i valori di default. Lo script è stato eseugito su tutte e tre le macchine, ed una volta completata l'esecuzione vi si potrà accedere tramite interfaccia web all'indirizzo ip della macchina desiderata alla porta 8983, dove è possibile verificare il corretto funzionamento e l'effettivo collegamento a Zookeeper.

Una volta effettuati questi passaggi si è essenzialmente pronti ad utilizzare il cluster, ma prima sono necessari ulteriori configurazioni in modo da poterlo utilizzare a pieno. In particolare, come accennato in precedenza, SolrCloud di default impedisce l'utilizzo di configurazioni e schemi caricati da fonti non attendibili, quindi risulta fondamentale la definizione di un meccanismo di autenticazione. Solr permette la definizione di diversi sistemi per questo, ma per i nostri scopi è stato sufficiente configurare il modulo per implementare la Basic Authentication. Per prima cosa è stato necessario definire il file security.json, andando a specificare quale classe andrà

```
{
"authentication":{
   "blockUnknown": true,
   "class":"solr.BasicAuthPlugin",
   "credentials":{"solr":"IV0EHq1OnNrj6gvRCwvFwTrZ1+z1oBbnQdiVC3otuq0=
Ndd7LKvVBAaZIF0QAVi1ekCfAJXr1GGfLtRUXhgrF8c="}
},
"authorization":{
   "class":"solr.RuleBasedAuthorizationPlugin",
   "permissions":[{"name":"security-edit",
      "role":"admin"}],
   "user-role":{"solr":"admin"}
}}
```

Figura 1.9: Configurazione BasicAuthentication Solr.

```
bin/solr zk cp file:path_to_local_security.json zk:/security.json -z zookeeperaddr:9983
```

Figura 1.10: Comando per l'upload del file per configurare la Basic Authorization su Zookeeper.

ad effettuare l'autenticazione ed username e password (criptata con l'algoritmo Sha256+sale). Tramite questo file è possibile definire anche i livelli di autorizzazione per ogni account registrato. Un esempio di questo file di configurazione è presente in figura 1.9. Questo file dovrà poi essere inserito all'interno del cluster Zookeeper, SolrCloud vi farà riferimento per definire le politiche di autenticazione ed autorizzazione). Questo può essere fatto in modo semplice utilizzando gli script forniti da solr, in particolare utilizzando il comando presente in figura 1.10.

```
curl --user solr:SolrRocks -X PUT --header "Content-Type:application/octet-
    stream" --data-binary @commits-schema.zip "http://natti-solrcloud.imolab
    .it:8983/api/cluster/configs/commits"
```

Listato 1.1: Upload configurazione di una collezione Solr.

L'ultimo passaggio effettuato è stata la creazione delle collezioni per i nostri casi di studio. Dopo aver definito gli schemi e le configurazioni necessarie, come definito nei capitoli seguenti relativi ai singoli casi d'uso, è stato fatto

uso delle API messe a disposizione da Solr per interagire con le collezioni, come mostrato nel listato 1.1. L'API effettuerà l'upload della configurazione sul nodo del cluster, permettendone poi l'utilizzo per la creazione delle collezioni. Per realizzare si può sempre fare uso delle API di gestione delle collezioni, eseguendo una chiamata come mostrato nel listato 1.2, oppure utilizzando direttamente la dashboard di gestione del cluster, come nell'esempio mostrato in figura 1.11.

```
curl -X POST --user solr:SolrRocks -H 'Content-type: application/json' --
    data-binary '{
  "create": {
    "name": "nome_collezione",
    "numShards": 2,
    "replicationFactor": 2,
    "config": "nome_configurazione"
  }
}' http://natti-solrcloud.imolab.it:8983/admin/collections
```

Listato 1.2: Comando per la creazione di una nuova collezione Solr.

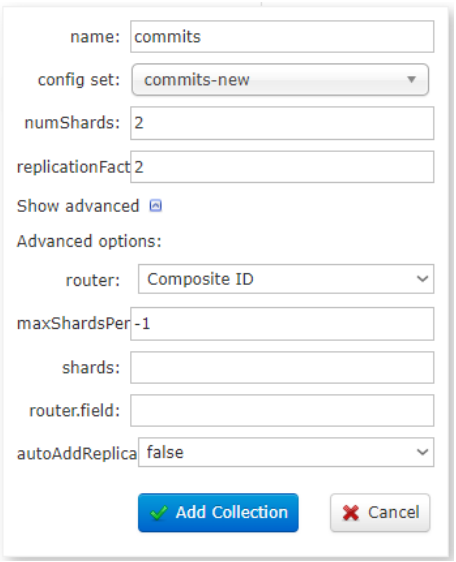

Figura 1.11: Menu per la creazione tramite interfaccia web delle collezioni.

Al momento della creazione della collezione è necessario specificare il numero di shard che la dovranno comporre tramite il parametro *numShards* e il numero di repliche che ciascun shard dovrà avere con il parametro *re-*

plicationFactor. Inoltre quando vengono create le collezioni, gli shard e i nodi vengono suddivisi su tutti i nodi disponibili e attivi, con il vincolo che due repliche dello stesso shard non si trovino mai sullo stesso nodo. Se un nodo non dovesse essere attivo quando viene chiamata l'azione CREATE, non otterrà alcuna parte della nuova collezione, il che potrebbe portare alla creazione di troppe repliche su un singolo nodo attivo. La definizione di *maxShardsPerNode* imposta un limite al numero di repliche che l'azione CREATE distribuirà a ciascun nodo, impostarlo al valore -1 rimuove questo limite.

1.2 Apache Spark

In questa sezione, esploreremo Apache Spark, un framework di elaborazione distribuita progettato per affrontare le sfide dell'elaborazione dei big data in modo efficiente e scalabile. Viene utilizzato in diverse applicazioni in cui è necessario elaborare grandi volumi di dati, strutturati e non strutturati, in modo rapido e affidabile. Inoltre vedremo come questa tecnologia si puó integrare con Apache Solr per migliorare le performance delle operazioni di indicizzazione e recupero delle informazioni.

1.2.1 Overview

Apache Spark è un framework multi linguaggio open-source per effettuare data engineering, data science e machine learnign su singoli nodi o cluster. Nato come progetto di ricerca al AMPLab della California nel 2009, reso poi open-source nel 2010. Molte delle idee alla base del sistema sono state presentate in numerosi articoli di ricerca nell'arco degli anni.

Dopo il suo rilascio, Spark ha ottenuto una vasta comunità di sviluppatori, ed è stato donato alla Apache Software Foundation nel 2013. Ad oggi, il progetto è sviluppato collaborativamente da una comunità di centinaia di sviluppatori appartenenti a centinaia di organizzazioni. Spark e la struttura degli RDD sono stati svilluppati nel 2012 in risposta alle limita-

zioni del paradigma di computazione MapReduce, che forza un particolare struttura di flusso dei dati lineare ai programmi distribuiti: in particolare i programmi MapReduce leggono dati di input dal disco, li mappano seguendo una funzione, riducono i risultati della mappa e immagazzinano i risultati della riduzione di nuovo sul disco. Gli RDD di spark funzionano come working set per i programmi distribuiti e offrono una forma ristretta di memoria distribuita condivisa.

Apache Spark necessita di un cluster manager e supporta modailità di deploy standalone (cluster Spark nativo, in cui è possibile lanciare un cluster manualmente o utilizzare gli script forniti dal pacchetto di installazione. è inoltre possibile eseguire i daemon su una singola macchina per effettuare test), Hadoop YARN, Apache Mesos o Kubernetes. Necessita inoltre di un sistema di archiviazione distribuito e Spark è in grado di interfacciarsi con un'ampia varietà che include Alluxio, HDFS (Hadoop Distributed File System), MapR File System, Cassandra, OpenStack Swift, Amazon S3, Kudu, Lustre file system, o permette anche la realizzazione di una soluzione custom. Spark inoltre supporta una modalità locale pseudo-distribuita solitamente utilizzata solo per scopi di sviluppo e di test, dove l'archiviazione distribuita non è richiesta ed il filesystem locale è utilizzabile al suo posto; in tal caso Spark viene eseguito su una singola macchina con un esecutore per core CPU.

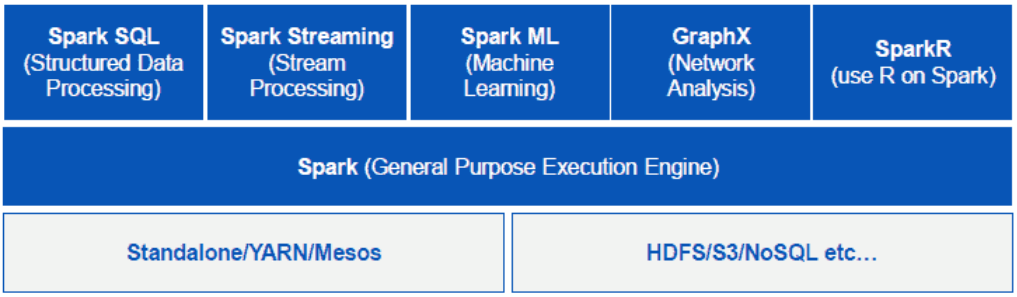

Figura 1.12: Struttura componenti di Apache Spark.

Apache Spark viene suddiviso nei seguenti componenti:

- **Spark Core** è alla base dell'intero progetto, e fornisce funzionalità di invio dei task distribuito, di schedulazione e funzionalità di I/O base, esposte tramite un interfaccia di programmazione di applicazioni (che supporta linguaggi come Java, Python, Scala, .NET e R) incentrata sull'astrazione degli *RDD*. Questa interfaccia rispecchia la modalità di programmazione funzionale/higher-order: un programma "driver" invoca operazione in parallelo come *map*, *filter* o *reduce* su un RDD passando la funzione a Spark, che poi ne schedula l'esecuzione in parallelo sul cluster. Queste operazioni prendono in input gli RDD e ne producono di nuovi e siccome gli RDD sono strutture immutabili e le loro operazioni sono "lazy" si riesce ad ottenere fault-tolerance tenendo traccia del "lineage" di ogni RDD, cioè la sequenza di operazioni che lo ha prodotto, così che possa essere ricostruito in caso di perdita di dati.

- **Spark SQL** è un componenete, costruito al di sopra di Spark Core, che introduce una nuova struttura dati chiamata *DataFrame* per il supporto all'utilizzo di dati strutturati e semi-strutturati. Il componente fornisce un domain-specific language per manipolare questi Dataframe in numerosi linguaggi (come Java, Scala, Python o .NET) oltre a fornire supporto per il linguaggio SQL grazie a un'interfaccia a linea di comando e server ODBC/JDBC.

  Sebbene l'astrazione dei Dataframe manchi del type-checkinga tempo di compilazione che hanno gli RDD, questa funzionalità è presente nella struttura dei DataSet fortemente tipati, che sono pienamente supportati da Spark SQL a partire dalla versione 2.0 di Spark. Un grande vantaggio di questa libreria è la sua modularità che gli permette di essere accoppiato con gli altri moduli, ottenendo così, funzionalità aggiunte per il processing di stream di dati. Ne è un esempio l'accoppiamento con MLlib che permette l'esecuzione di tecniche di machine learning su stream o con Graphx che abilita l'elaborazione di grafi.

- **Spark Streaming** utilizza le capacità di scheduling rapido fornite da

Spark Core per effettuare analisi di streaming. Effettua l'ingestion dei dati in mini-batch e effettua delle trasformazioni RDD su di esse. Questo design consente di utilizzare lo stesso set di codice applicazione scritto per l'analisi in batch anche nell'analisi in streaming, facilitando così la facile implementazione dell'architettura a lambda. Questa convenienza porta però con se una penalità di latenza uguale alla durata del mini-batch. Spark Streaming ha il supporto integrato per consumare dai socket Kafka, Flume, Twitter, ZeroMQ, Kinesis e TCP/IP, ed inoltre a partire da Spark 2.0 supporta anche una tecnologia basata sui DataSet, chiamata Structured Streaming.

- **Spark MLlib** è un framework di machine learning distribuito su Spark Core. Implementa molti degli algoritmi di machine learning più comuni e semplifica le pipeline di apprendimento automatico su larga scala tra cui:
  - Linear regression, logistic regression
  - Support Vector Machines
  - Naive Bayes classifier
  - Means clustering
  - Decision trees
  - Recommendations using Alternating Least Squares
  - Basic statistics
  - Chi-squared test, Pearsons or Spearman correlation, min, max, mean, variance
  - Feature extraction
  - Term Frequency/ Inverse Document Frequency useful for search
- **GraphX** è un framework di elaborazione dei grafi distribuito. Siccome è basato sugli RDD, che sono immutabili, anche i grafi lo sono.

GraphX fornisce due API separate per l'implementazione di algoritmi come PageRAnk. GraphX è utile per fornire informazioni generali sulla rete di grafi ed è in grado di misurare elementi come quanto un grafo è connesso, il grado di distribuzione, lungezza media dei percorsi e altre misure di alto livello dei grafi. Può anche unire i grafi e trasformarli rapidamente. Spark GraphX fornisce Resilient Distributed Graph (RDG, un'astrazione di RDD) le cui API sono utilizzate dai data scientist per effettuare numerose operazioni sui grafi sfruttando varie primitive computazionali. Analogamente alle operazioni di base degli RDD come map, filter, anche i grafi delle proprietà sono costituiti da operatori di base , questi operatori prendono UDF (funzioni definite dall'utente) e producono nuovi grafici. Inoltre, questi sono prodotti con proprietà e struttura trasformate.

- Il linguaggio di programmazione R è ampiamente utilizzato dai data scientist grazie alla sua semplicità e capacità di eseguire algoritmi complessi, ma la sua capacità di elaborazione dei dati è limitata a un singolo nodo. Ciò rende R non utilizzabile durante l'elaborazione di un'enorme quantità di dati. Il problema è risolto da **SparkR** che fornisce l'implementazione di dataframe che supporta operazioni come selezione, filtraggio, aggregazione e così via su set di dati di grandi dimensioni distribuiti. SparkR supporta anche l'apprendimento automatico distribuito tramite Spark MLlib.

1.2.2 Architettura spark

L'architettura di Apache Spark si basa su un insieme di livelli e componenti debolmente accoppiati. In generale, Spark sfrutta un approccio master/slave o più precisamente, come definito nella documentazione, master/worker. All'interno dell'architettura vi sono 3 nodi principali:

- Driver Node.
- Worker Node.

- Cluster Manager Node.

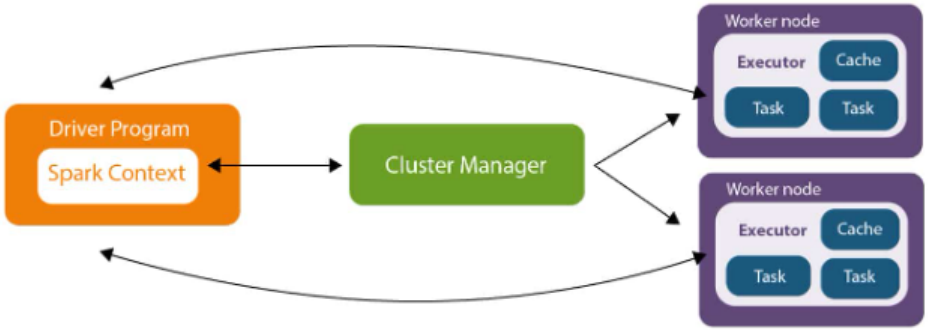

Figura 1.13: Architettura dei cluster Spark.

Il Driver Node, o Master Application Node, è il nodo che contiene i servizi che si occupano di gestire e coordinare l'esecuzione di un'applicazione. Il Driver Node contiene lo Spark Context ossia la prima entità che viene creata all'interno di una applicazione che permette di interfacciarsi con il cluster. Ogni operazione effettuata dall'utente passa attraverso lo Spark Context che si occuperà di gestirla ed eventualmente restituire una risposta. Driver Node e Worker Node possono essere installati in varie configurazioni e supportano l'esecuzione sia su di uno stesso nodo fisico che su nodi diversi ma anche in configurazioni ibride. Le applicazioni Spark vengono eseguite come set indipendenti di processi in un cluster, coordinati dallo SparkContext del programma principale (chiamato programma driver). Lo Spark Context può connettersi a diversi tipi di cluster manager (il cluster manager autonomo di Spark, Mesos, YARN o Kubernetes), che allocano le risorse tra le applicazioni. Una volta connesso, Spark acquisisce executor sui nodi del cluster, che sono processi che eseguono calcoli e archiviano dati per l'applicazione. Successivamente, invia il codice della applicazione (definito dai file JAR o Python passati a Spark Context) agli esecutori. Infine, SparkContext invia le attività agli esecutori per l'esecuzione.

Ogni applicazione ottiene i propri processi esecutori, che rimangono attivi per la durata dell'intera applicazione ed eseguono attività in più thread.

Ciò ha il vantaggio di isolare le applicazioni l'una dall'altra, sia dal lato della pianificazione (ogni driver pianifica le proprie attività) sia dal lato dell'esecutore (attività di diverse applicazioni eseguite in diverse JVM). Ciò significa anche che i dati non possono essere condivisi tra diverse applicazioni Spark (istanze di SparkContext) senza scriverli su un sistema di archiviazione esterno. Il driver program deve ascoltare e accettare le connessioni in entrata dai suoi esecutori per tutta la sua durata, per cui il driver program deve essere raggiungibile da tutti i nodi worker.

Per effettuare la schedulazione dei task all'interno del cluster distribuito Spark fa uso di uno schedulatore basato sul concetto dei grafi direzionati aciclici (Directed Acyclic Graphs o DAG). Il DAG è considerato direzionato in quanto le operazioni vengono eseguite in un ordine specifico, mentre è considerato aciclico appunto per la mancanza di loop o cicli all'interno del piano di esecuzione. Ciò significa che ogni stage di esecuzione dipende dal completamento di quello precedente ed ogni task all'interno di ciascuno stage può essere eseguito indipendentemente dagli altri. Con una prospettiva di alto livello si può dire che il DAG rappresenta un piano d'esecuzione logico di un job Spark. Quando un'applicazione viene sottomessa, Spark ne traduce le operazioni in stage e task all'interno di un DAG.

La costruzione del grafo si basa su tre concetti fondamentali:

- Stage: sono un insieme di task che può essere eseguito in parallelo. Ne esistono di due tipi: *shuffle* che comprendono lo scambio di dati tra i nodi e *non-shuffle* che invece non lo comprendono.

- Task: rappresentano una singola unità di lavoro che può essere eseguita su una singola partizione di un RDD, e quindi la più piccola unità di parallelismo in Spark.

- Dependencies: indicano l'ordine in cui i task possono essere eseguiti, definiti dalle dipendenze tra gli RDD. Si possono distinguere due tipologie: *narrow* che indicano che ogni partizione dell'RDD padre è utilizzata al massimo da una partizione dell'RDD figlio, e *wide* che in-

vece indicano che ogni partizione dell'RDD padre può essere utilizzata da più partizioni dell'RDD figlio.

In questo modo si permette a Spark di effettuare varie ottimizzazioni, come ad esempio pipelining, riordinamento dei task ed eliminare operazioni non necessarie in modo da rendere più efficiente il processo di esecuzione dei job. Inoltre la suddivisione dei job in elementi logici più piccoli (task e stage) permette di eseguirli in parallelo e distribuirli su tutto il cluster per velocizzarne la risoluzione.

L'utilizzo del DAG permette di avere fault tolerance, grazie a due meccanismi principali:

- Persistenza degli RDD: quando un RDD viene contrassegnato come "persistente", Spark manterrà i dati della partizione in memoria o su disco, a seconda del livello di archiviazione utilizzato. Ciò garantisce che, in caso di errore di un nodo, Spark possa ricostruire le partizioni perse dai dati persistenti, anzichè ricalcolare l'intero RDD.

- Checkpointing: è un meccanismo per salvare periodicamente gli RDD in una memoria stabile come HDFS. Questo meccanismo riduce la quantità di ricalcolo richiesta in caso di errori. In caso di guasto di un nodo, gli RDD possono essere ricostruiti dall'ultimo checkpoint e dalla loro storia di computazione.

Siccome il Driver Node mantiene la memoria di ogni operazione applicata sugli RDD tramite il DAG, esso conosce ogni dettaglio di ciascun task eseguito e su che nodo è andato in esecuzione. Per questo motivo, in caso di un fallimento, il Driver Node riesce ad attuare una procedura di recovery in cui richiama gli RDD dei dati originali, la cui computazione non è andata a buon fine e risottomette le operazioni per completare l'elaborazione richiesta dall'applicazione.

Infine il DAG permette il riuso dei risultati intermedi generati da un lavoro. Ciò significa che se una parte dei dati viene elaborata una volta, può essere riutilizzata in lavori successivi, riducendo così i tempi di elaborazione e

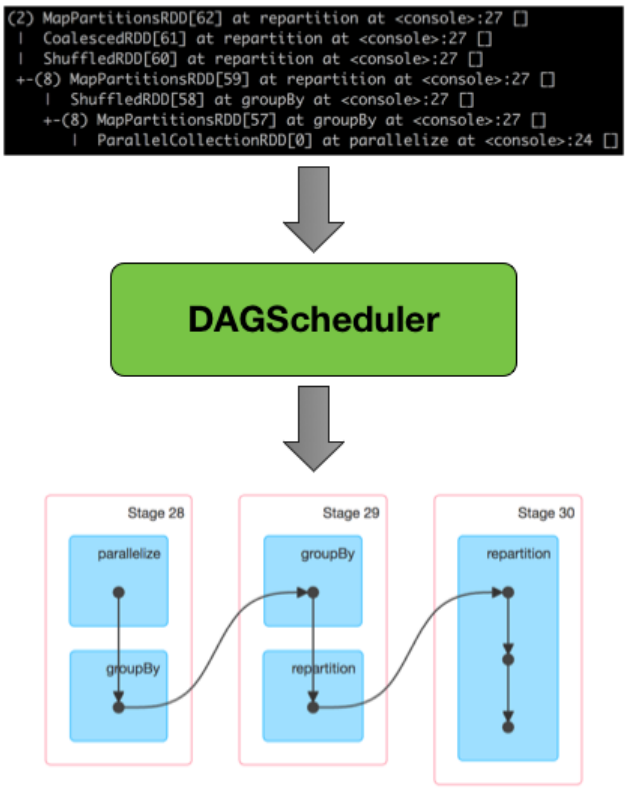

Figura 1.14: Trasformazione di un piano di esecuzione logico, in un piano di esecuzione fisico suddiviso in stage.

migliorando le prestazioni. Inoltre permette di ottenere una rappresentazione visiva del piano di esecuzione logico del lavoro, che può aiutare gli utenti a comprendere meglio il lavoro e identificare eventuali colli di bottiglia o problemi di prestazioni.

Il Driver Node si occupa anche di monitorare lo stato di salute dei nodi Worker e dei task in esecuzione. Per fare ciò, oltre a ricevere un feedback al completamento di ogni Stage, viene sfruttato un meccanismo ad heartbeat per assicurarsi di ottenere lo stato aggiornato di ogni singolo executor e task.

Infine il Driver Node ospita anche l'interfaccia Web dell'applicazione. Attraverso questa pagina interattiva è possibile:

- Conoscere lo stato dell'applicazione.

- Visionare il modo in cui è stata suddivisa per essere sottomessa.
- Ottenere la lista dei task completati con successo e di quelli falliti con relativo output e log di errore.
- Visualizzare informazioni relative allo Storage utilizzato dall'applicazione.
- Ottenere informazioni sulle caratteristiche degli executor e le risorse utilizzate.
- Visionare l'elenco di tutte le variabili d'ambiente relative all'applicazione.

I Worker Node hanno il cui compito è di mettere in esecuzione i task. Questi task vengono eseguiti sulle partizioni di RDD presenti sul nodo e il risultato restituito allo Spark Context presente nel Driver Node. Avere un numero maggiore di Worker corrisponde ad una maggior capacità di parallelismo sfruttabile per la suddivisione ed esecuzione parallela dei task, oltre ad un aumento dello spazio in memoria, permettendo così di gestire moli di dati più grandi.

Tipicamente gli esecutori hanno un ciclo di vita corrispondente a quello dell'applicazione. Questo tipo di allocazione è definita come allocazione statica degli executor ed è il comportamento di default in Spark. È possibile utilizzare un' allocazione dinamica che permette uno sfruttamento migliore delle risorse del cluster, in quanto vengono allocate solo nel momento di effettivo bisogno al costo però di un maggior tempo per l'inizio dell'elaborazione.

All'interno di un nodo Worker sono, tipicamente, presenti più executor. Come comportamento di default, Spark istanzia un numero di esecutori pari al numero di core presente su ogni macchina meno uno. I nodi Worker sono in grado di comunicare fra di loro, questa capacità è sfruttata in particolare dal servizio di Block Manager che permette ai nodi di scambiarsi blocchi di dati senza dover passare ogni volta dal Driver Node.

1.2.3 RDD, Dataset e Dataframe

L'astrazione principale fornita da Spark è un *dataset distribuito resiliente*, o meglio conosciuto tramite l'acronimo **RDD**, cioè una raccolta di elementi partizionati tra i nodi del cluster su cui è possibile operare in parallelo. Gli RDD vengono creati a partire da un file all'interno del file system Hadoop (o qualsiasi altro file system supportato da Hadoop) o da una collezione Scala esistente nel driver program e trasformandolo. Gli utenti possono anche chiedere a Spark di mantenere un RDD in memoria, consentendone il riutilizzo efficiente in operazioni parallele.

Gli RDD supportano due tipi di operazioni:

- Trasformazioni, che creano un nuovo dataset a partire da uno esistente.
- Azioni, che restituiscono un valore al driver program dopo aver eseguito dei calcoli sul dataset.

Ad esempio, *map* è una trasformazione che passa ogni elemento del set di dati e restituisce un nuovo RDD che rappresenta i risultati. D'altra parte, *reduce* è un'azione che aggrega tutti gli elementi dell'RDD utilizzando una funzione e restituisce il risultato finale al driver program.

Tutte le trasformazioni in Spark sono lazy, in quanto non calcolano immediatamente i risultati. Invece, si tiene traccia di solo le trasformazioni applicate a un dataset di base (ad esempio un file). Le trasformazioni vengono calcolate solo quando un'azione richiede la restituzione di un risultato al driver program. Questo design consente a Spark di lavorare in modo più efficiente.

Come accennato dai paragrafi precedenti, per eseguire i job Spark suddivide l'elaborazione delle operazioni RDD in task, ognuno dei quali viene eseguito da un esecutore. Prima dell'esecuzione, Spark calcola la *closure* del task. La closure rappresenta le variabili e metodi che devono essere visibili affinché gli esecutori possano essere in grado di eseguire i calcoli sull'RDD. Una delle funzionalità più importanti di Spark è la persistenza (o la memorizzazione all'interno della cache) di un dataset in memoria tra le operazioni.

Quando si persiste un RDD, ogni nodo memorizza tutte le sue partizioni che vengono calcolate e le riutilizza in altre azioni su quel dataset o derivati da esso. Ciò consente alle azioni future di essere molto più veloci e risulta uno strumento chiave per algoritmi iterativi e un uso interattivo rapido.

La prima volta che un RDD viene calcolato in un'azione, verrà mantenuto in memoria sui nodi e dato che la cache di Spark è fault-tolerant, se una qualsiasi partizione di un RDD viene persa verrà automaticamente ricalcolata utilizzando le trasformazioni che l'hanno creata originariamente.

Come discusso precedentemente, a differenza dell'astrazione di base degli RDD, le interfacce fornite da Spark SQL forniscono a Spark ulteriori informazioni sulla struttura dei dati e del calcolo eseguito. Internamente, Spark SQL utilizza queste informazioni aggiuntive per eseguire ulteriori ottimizzazioni. Un Dataset è una raccolta distribuita di dati. Il Dataset è una nuova interfaccia aggiunta in Spark 1.6 che offre i vantaggi degli RDD (tipizzazione forte, possibilità di usare funzioni lambda) con i vantaggi del motore di esecuzione ottimizzato di Spark SQL. Un DataFrame è un Dataset organizzato in colonne denominate. è concettualmente equivalente a una tabella in un database relazionale o a un dataframe in R/Python, ma con maggiori ottimizzazioni.

I DataFrame in Spark hanno la loro esecuzione ottimizzata automaticamente da un ottimizzatore di query. Prima dell'avvio di un qualsiasi calcolo su un DataFrame, l'ottimizzatore Catalyst compila le operazioni utilizzate per creare il DataFrame in un piano per l'esecuzione. Poichè l'ottimizzatore comprende la semantica delle operazioni e la struttura dei dati, può prendere decisioni intelligenti per accelerare il calcolo. Ad alto livello, ci sono due tipi di ottimizzazioni:

- Inizialmente vengono applicate ottimizzazioni logiche come il predicate pushdown. L'ottimizzatore può inserire i predicati del filtro nell'origine dati, consentendo all'esecuzione fisica di ignorare i dati irrilevanti.

- In secondo luogo, vengono compilate le operazioni in piani per l'esecuzione e viene generato il bytecode JVM.

Possono anche essere eseguite ottimizzazioni di livello più basso come l'eliminazione di costose allocazioni di oggetti e la riduzione delle chiamate di funzioni virtuali. Poichè l'ottimizzatore genera il bytecode JVM per l'esecuzione, gli utenti Python sperimenteranno le stesse prestazioni elevate degli utenti Scala e Java.

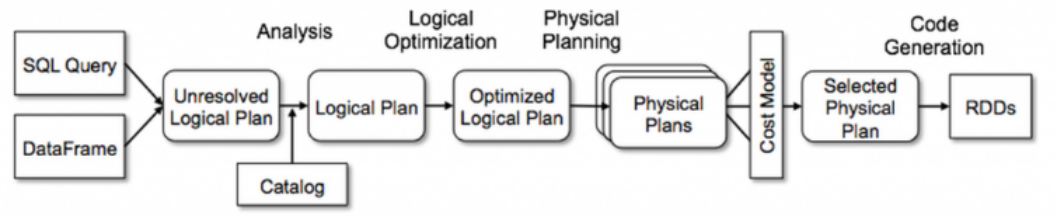

Figura 1.15: Flusso per la realizzazione degli RDD a partire da un input, come DataFrame,DataSet o query Sql.

Una volta creati, i DataFrame forniscono un domain-specific language per la manipolazione dei dati distribuiti, ed inoltre supportano la lettura dei dati dai formati più diffusi, come file JSON, file Parquet e tabelle Hive. Può leggere da file system locali, file system distribuiti (HDFS), cloud storage (S3) e sistemi di database relazionali esterni tramite JDBC. I DataFrame possono essere estesi per supportare qualsiasi formato o origine dati di terze parti come Avro, CSV, ElasticSearch e Cassandra, inoltre possono essere utilizzati direttamente all'interno delle pipeline di machine learning fornite da MLlib.

1.2.4 Deploy cluster Spark

Grazie ai numerosi script messi a disposizione direttamente da Spark la realizzazione di un cluster in modalità standalone risulta un processo lineare e semplice.

Per sfruttare le capacità di parallelizzazione fornite da Spark si è deciso di utilizzare un cluster in modalità standalone composto da tre host, suddivisi tra un nodo master e due nodi worker.

Il primo passo per il deploy, oltre ad installare le versioni corrette di Java e Scala su tutte le macchine, è stato quello di definire l'indirizzo di risoluzione di ciascuna macchina all'interno del file hosts di sistema e di definire una coppia di chiavi ssh sul nodo master, fornendo ai nodi worker la chiave pubblica. Questo passaggio è necessario per permettere ai nodi di comunicare fra loro e permettere la distribuzione dei task. Dopodichè è stato necessario configurare l'environment sul nodo master e su tutti i nodi worker, andando a modificare il file *spark-env.sh* aggiungendo l'indirizzo ip del nodo master e andando a definire all'interno di un file *workers* gli indirizzi dei nodi worker che saranno parte del cluster.

```
SPARK_MASTER_HOST='192.168.151.130'
```

(a)

```
# A Spark Worker will be started on each of the machines listed below.
192.168.151.145
192.168.151.244
```

(b)

Figura 1.16: Esempi del contenuto dei file spark-env.sh (a) e workers (b).

Per concludere il setup del cluster infine è stato eseguito lo script *./start-all.sh* sul nodo master, che banalmente si occupa di eseguire lo script *start-master.sh* per l'istanza Spark sul nodo master e lo script *start-worker.sh* in sequenza su tutti i nodi worker specificati. Infine lo stato del cluster risulta visibile tramite interfaccia web alla porta 8080 di ciascuno degli host.

1.2.5 Spark-Solr

Nel caso ci si voglia appoggiare a librerie o progetti esterni per la realizzazione di applicazioni, Spark permette di lanciare direttamente applicazioni su un cluster tramite gli script *spark-submit.sh*. Questo script permette di importare librerie esterne, a supporto del funzionamento delle applicazio-

ni, tramite l'import di un assembly jar (o uber jar) o specificando i singoli jar delle librerie utilizzate come parametro dello script. Alternativamente Spark mette a disposizione lo script *spark-shell.sh* per effettuare l'esecuzione di semplice codice senza dover realizzare file .jar.

Per il nostro caso d'uso si è deciso di utilizzare la libreria Spark-Solr, progettata dagli stessi sviluppatori di Solr, per facilitare l'integrazione tra le due tecnologie.

Spark-Solr si interpone tra Spark (ed il codice custom per il processing dei dati) e il cluster Solr. Ha il compito di interfacciarsi con il cluster per l'invio o la lettura di dati, e la conversione di questi in un formato riconoscibile da Solr e viceversa. Esistono diversi modi per effettuare questa conversione come ad esempio il meccanismo della Reflection Java o file l'utilizzo dei file Json.

Per fare ciò la libreria fa uso a sua volta di SolrJ. SolrJ è un'API che facilita la comunicazione tra le applicazioni Java e Solr, fornendo astrazioni per ridurre lo sforzo di impostare la connessione con Solr, e consentire alle applicazioni di interagirci utilizzando metodi di alto livello. Tramite queste API è possibile connettersi con il cluster creando un oggetto *SolrClient*, in particolare tramite le implementazioni *HttpSolrClient* (specifico per cluster con un singolo endpoint) o *CloudSolrClient* (usato per istanze di SolrCloud configurandosi tramite Zookeeper).

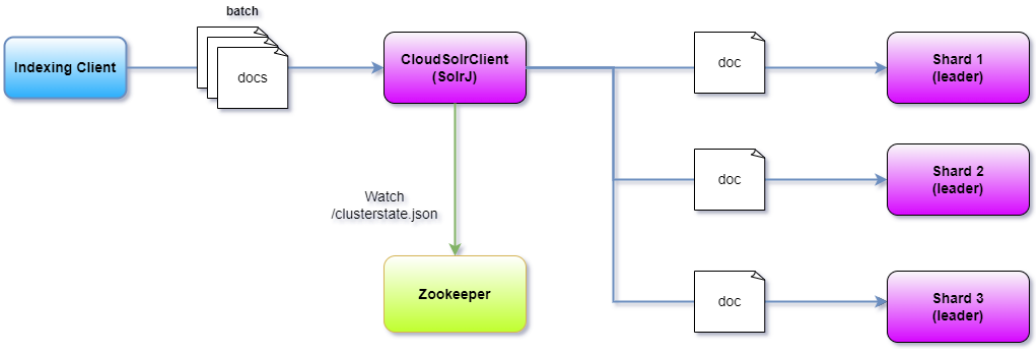

Figura 1.17: Flusso per l'indicizzazione dei documenti con la libreria Solr-Spark.

La figura 1.17 mostra il funzionamento dietro le quinte della libreria nel caso dell'indicizzazione di batch di documenti verso un cluster Solr. Inizialmente il programma Spark si occuperà di suddividere i dati in esame in mini batch di documenti per poi inviarli al SolrClient. Questo interrogherà Zookeeper per ottenere lo stato del cluster e per scoprire gli indirizzi degli shard leader, ed eseguire così in parallelo le operazioni di indicizzazione dei singoli documenti che compongono i batch. In caso di fault di uno dei leader, il client andrà automaticamente a reperire gli altri shard attivi. Questo permette di parallelizzare ulteriormente il processo di indicizzazione, riducendo notevolmente l'overhead di rete.

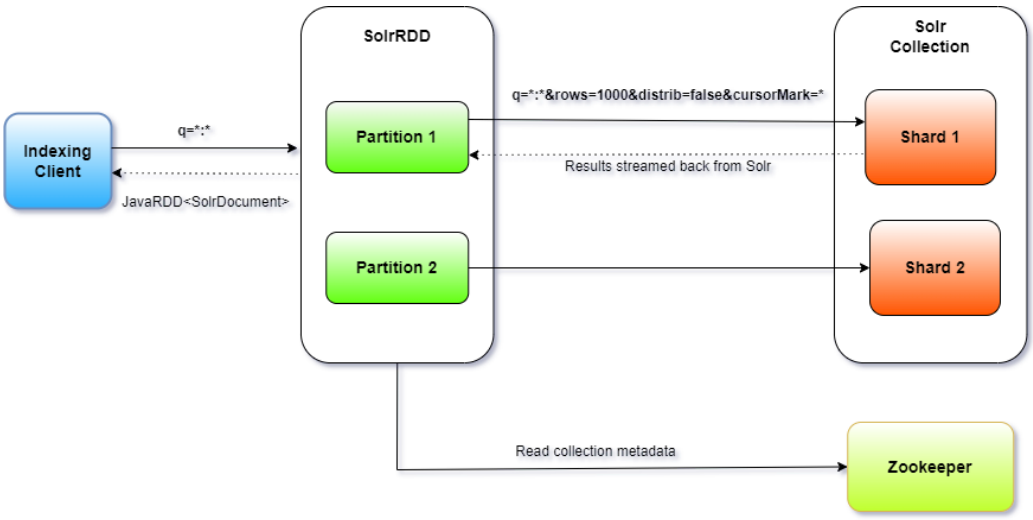

Figura 1.18: Processo di lettura da Solr a Spark e realizzazione di SolrRDD.

La figura 1.18 mostra invece come avviene il processo di lettura di dati provenienti da un cluster Solr verso un'istanza di Spark. Le query verso il cluster Solr possono essere effettuate tramite l'uso dei *SolrRDD*. Questi SolrRDD producono dei JavaRDD contententi oggetti chiamati SolrDocument (definiti da SolrJ), che definiscono una rappresentazione concreta di un documento all'interno di un indice Solr, in cui è possibile definire i campi dei dati tramite coppie chiave-valore.

Analogalmente al processo di indicizzazione il SolrRDD interrogherà Zoo-

keeper per ottenere i metadati del cluster e gli indirizzi degli shard della collezione su cui effettuare le query. Una volta individuati verranno effettuate delle micro-query in parallelo su ogni shard (definito dal parametro *distrib=false*). Inoltre se il cluster Spark possiede più executor disponibili rispetto al numero di shard, si può aumentare ulteriormente il parallelismo durante la lettura, suddividendo ogni shard in sottointervalli (sub ranges) utilizzando come divisore un campo *split*. Un buon valore da associare al campo split è ad esempio il campo *version* che viene allegato a ogni documento dal leader dello shard durante l'indicizzazione.

La libreria è anche fortemente integrata con i moduli SparkSql e MLlib. In particolare grazie al supporto a SparkSql è possibile esporre ogni query da e verso Solr tramite le astrazioni dei Dataframe o Dataset, permettendo di dedurre automaticamente lo schema dei documenti e quindi rendere l'accesso ai dati ancora più semplice ed efficiente. Il collegamento con MLlib facilita invece la creazione di un indice TF-IDF, dato che viene già utilizzato da Solr per realizzare il proprio indice per la ricerca, fornendo automaticamente i metadati.

Capitolo 2

Estrazione di conoscenza: consuntivazioni Emt

Nel seguente capitolo andremo ad esporre il primo caso di studio che abbiamo esplorato per mettere in pratica i concetti e le tecnologie esposte nel capitolo precedente. Nello specifico è stata preso in considerazione l'applicazione utilizzata per la consuntivazione delle attività del personale aziendale, chiamata EMT, andando ad esaminare i messaggi, espressi in linguaggio naturale, lasciati da dipendenti e collaboratori riguardo le attività lavorative, per estrarre da essi i concetti e le tecnologie utilizzate.

L'obiettivo finale è quello di poter applicare un valore di correlazione tra i concetti estratti ed i progetti o i singoli dipendenti, aiutando così il processo decisionale delle risorse umane per l'assegnazione del personale più adatto e favorire il recupero di informazioni in merito alle competenze.

2.1 Integrazione con Oracle Db

Come accennato in precedenza è stato necessario individuare un metodo per integrare Apache Solr con un database Oracle già esistente. Solr mette a disposizione, a partire dalla versione 1.3, uno strumento chiamato *Data Import Handler* (DIH) che permette, tramite la realizzazione di speci-

fiche query, l'estrazione di informazioni da database relazionali, sfruttando le relazioni presenti sullo schema E-R.

Come caso di studio è stato preso in considerazione un database per la rendicontazione dei progetti all'interno dell'azienda, raccogliendo dati associati alle consuntivazioni sui singoli progetti, gli utenti che le hanno effettuate e le note relative alle ore di lavoro registrate, con l'obiettivo di rilevare ed assegnare topic ai progetti ed estrarre le skill maturate dagli utenti sui vari progetti.

```
<dataConfig>
  <dataSource driver="oracle.jdbc.OracleDriver" url="jdbc:oracle:thin:@<
    oracle-url>/xe" user="*********" password="*********"/>
  <document>
    <entity name="PRESENZE" query="select wp.*,sp.NOTE as NOTE_SOTTOP, sp.
    DESCRIZIONE as DESCRIZIONE_SOTTOP,sp.DESCRIZIONE||' '||sp.NOTE||' '||wp.
    NOTE AS COMPLETE_NOTE
     FROM view_presenze wp LEFT OUTER JOIN SOTTOPROGETTI sp ON wp.
    idsottoprogetto=sp.idsottoprogetto WHERE wp.note is not null"
    deltaQuery="select id from VIEW_PRESENZE where last_modified > '${
    dataimporter.last_index_time}'">
    <field column="NOTE" name="nota" />
    <field column="NOTE_SOTTOP" name="nota_sottoprogetto"/>
    <field column="DESCRIZIONE_SOTTOP" name="descrizione_sottoprogetto"/>
    <field column="PROGETTO" name="progetto"/>
    <field column="CODCOMMESSA" name="codcommessa"/>
    <field column="DATA" name="data"/>
    <field column="CODTEMATICA" name="codtematica"/>
    <field column="CODAZIENDAPROGETTO" name="azienda"/>
    <field column="COLLABORATORE" name="utente"/>
    <field column="CODAZIENDAPROGETTO" name="azienda"/>
    <field column="COMPLETE_NOTE" name="nota_completa"/>
    </entity>
  </document>
</dataConfig>
```

Listato 2.1: Configurazione query Data Import Handler.

Per poter utilizzare il *Data Import Handler* per prima cosa è stato necessario configurarne i parametri per la connessione al database, quali:

- Url di accesso al database.
- Credenziali d'accesso.

- Driver da utilizzare, relativo al tipo di database in questione.

Solr possiede già un'ampia selezione di driver, adatta all'uso dei database più utilizzati, ma è sprovvisto del driver per accedere ai database Oracle come quello preso in considerazione come caso d'uso. Per rimediare alla mancanza è stato scaricato il driver in questione ed è stato poi inserito all'interno della home di Solr e referenziata all'interno del file di configurazione della collezzione per l'utilizzo.

La struttura dello schema per l'estrazione dei dati è mostrata nel listato 2.1. Questa mostra la corrispondenza tra le tabelle ed entità del database con i campi dello schema Solr. L'entità principale su cui verrà effettuata l'interrogazione è stata denominata PRESENZE, su di essa è stata definita una query SQL per ottenere tutti i campi relativi ai progetti, più le note dei sottoprogetti effettuando una JOIN sull'id del progetto e concatenando le stringhe relative ai campi. Le colonne utili al nostro caso d'uso sono poi state assegnate ad un elemento dello schema di Solr che andranno ad immagazzinarne i valori all'interno dell'indice. Durante la realizzazione di questa configurazione si è sperimentato con diversi metodi di estrazione, in paricolare il *Data Import Handler* mette a disposizione un meccanismo definito dal comando *delta query*, che permette di ottenere un comportamento uguale a quello di un INNER JOIN. Tale configurazione è stata però accantonata a causa delle performance inferiori nel tempo di esecuzione della query.

Il DIH si è rivelato ad ogni modo uno strumento molto flessibile, in quanto permette di dichiarare multiple query all'interno dello stesso file di confgurazione, in modo da poter indicizzare diversi tipi di documenti.

2.2 Schema

Per immagazzinare e indicizzare i contenuti del database una volta estratti, è stato prima necessario realizzare uno schema ad hoc per la collezione. Il campo principale che si vuole esaminare nel caso di studio è il campo *nota_completa*, che contiene la concatenazione delle note riguardanti le con-

suntivazione delle ore sui progetti. Questo campo contiene principalmente testo in lingua italiana, ma spesso contiene anche termini inglesi o inglesismi, dovuti spesso al dominio tecnico relativo ai progetti software. Sebbene Solr definisca già numerosi tipi di dato all'interno dello schema di default, utilizzato come base, si è reso necessario definirne uno specifico, in modo da poter definire una serie di filtri per poterne analizzare il contenuto. Questo nuovo tipo è stato denominato *emt-extraction*, ed definito nel seguente frammento di configurazione.

```
<fieldType name="emt_extraction" class="solr.TextField" positionIncrementGap
    ="100">
  <analyzer>
    <charFilter class="solr.HTMLStripCharFilterFactory"/>
    <tokenizer class="solr.ClassicTokenizerFactory"/>
    <filter class="solr.ElisionFilterFactory" ignoreCase="true" articles="
    lang/contractions_it.txt"/>
    <filter class="solr.SynonymGraphFilterFactory" synonyms="
    synonyms_case_sensitive.txt" />
    <filter class="solr.LowerCaseFilterFactory"/>
    <filter class="solr.SynonymGraphFilterFactory" synonyms="synonyms_emt.
    txt" />
    <filter class="solr.ShingleFilterFactory" maxShingleSize="3"
    tokenSeparator="_"/>
    <filter class="solr.SynonymGraphFilterFactory" synonyms="synonyms_emt.
    txt" />
    <filter class="solr.KeepWordFilterFactory" words="keepwords_extracted.
    txt" ignoreCase="true"/>
  </analyzer>
</fieldType>
```

Listato 2.2: Definizione tipo di dato per estrazione dei termini dalle note di progetto.

In Solr per ogni tipo di dato può essere definito un *Analyzer*, un elemento di configurazione dei tipi di dato contenuti negli schemi che permette di analizzare il contenuto dei campi sulla base dei componenti al suo interno quali:

- *CharFilter*, permettono di effettuare operazioni di pre-processing sul testo prima della suddivisione in token.

- *Tokenizer*, elementi che si occupano della scomposizione del testo in token sulla base di pattern predefiniti.
- *Filter*, elementi che definiscono politiche di esclusione o modifica dei token seguendo regole preconfigurate definiti.

Il contenuto di un analyzer viene eseguito in cascata, creando così una catena di operazioni per l'analisi dei campi presenti nello schema che lo utilizzano.

Come primo elemento è stato definito un *HTMLStripCharFilter* per eliminare ogni eventuale carattere HTML dal testo, dopodiché è stato utilizzato un *ClassicTokenizer* separando il testo in base alla punteggiatura e gli spazi vuoti con le seguenti regole:

- I punti che non sono seguiti da uno spazio sono mantenuti come parte del token.
- Le parole vengono divise dal simbolo "-" a meno che non sia presente un numero. In tal caso la parola non viene divisa e il simbolo non viene scartato.
- Il tokenizer è in grado di riconoscere domini internet e indirizzi email, e li preserva come unico token.

In seguito alla suddivisione in token viene applicata una serie di filtri che mirano a eliminare elementi esterni al dominio e a raggruppare eventuali sinonimi. Come primo filtro viene utilizzato un *ElisionFilter* per eliminare da ogni token eventuali apostrofi. Data la tipologia di testo esaminato, in cui possono essere presenti molti acronimi espressi in maiuscolo, è stato deciso di utilizzare due *SynonymGraphFilter* separati da un *LowerCaseFilter*. Il primo filtro per sinonimi si occuperà di trasformare i token degli acronimi nella loro forma estesa (es. CI → continuous integration) o in token che rappresentano un raggruppamento più ampio (es. POST,PUT,DELETE,GET → REST), il secondo filtro per sinonimi invece effettua una conversione più ampia, andando a riunire ogni token all'interno di categorie più ampie, dopo

essere stati prima portati in un formato lowercase dal filtro precedente. Un filtro molto importante per il funzionamento dell'estrazione è lo *ShingleFilter*. Questo filtro si occupa di creare dei costrutti chiamati *shingle*, composti dalla fusione di sequenze di token all'interno di uno solo. È stato scelto di limitare questa sequenza di token ad un massimo di tre elementi, separandoli dal simbolo "_". Questi token passeranno attraverso un ultimo filtro sinonimi per effettuare un ultimo raggruppamento. Infine verrà applicato un *KeepWordFilter* per eliminare tutti i termini che non fanno parte di un dizionario di parole predefinito.

I file di configurazione dei *SynonymGraphFilter* e il dizionario del *KeepWordFilter* sono stati realizzati assieme ad esperti del dominio e nella versione riportata forniscono già validi risultati, ma con l'aumentare dei dati a nostra disposizione e con l'inevitabile evoluzione del dominio risulterà necessario effettuare aggiornamenti periodici per mantenerne o eventualmente aumentarne le performance.

Questo tipo di dato è stato definito come base, ma in seguito a diverse discussioni e feedback sono stati aggiunti altri due tipi di dato:

- Termini tecnici, denominati *emt-technical*, legati a tematiche di natura tecnica come tecnologie, tecniche o legate ai processi di sviluppo. Questi termini rappresentano le conoscenze tecniche affrontate all'interno dei progetti, o acquisite dai collaboratori.

- Termini trasversali, denominati *emt-cross-technical*, legati alle soft-skill o a processi di gestione e coordinamento utilizzati dai collaboratori all'interno dei progetti.

Questi due nuovi tipi utilizzano come base i filtri descritti in precedenza ai quali vengono però aggiunti altri due *KeepWordFilter* configurati con altri dizionari specifici per effettuare la distinzione. Il primo filtro utilizza un dizionario che si basa sui termini più popolari estratti dal campo *nota_completa* definito inizialmente, mentre il secondo ne utilizza uno contentente solamen-

te termini considerati tecnici o trasversali, a seconda del tipo di dato che si vuole esaminare.

Per ridurre il più possibile le operazioni ridondanti, dovuti alla suddivisione descritta precedentemente, è stata utilizzata una struttura del *copy field* messa a disposizione da Solr. Questa struttura si occupa di copiare i dati da una struttura ad un'altra senza dover utilizzare metodi come il Data Import Handler, equindi senza andare ad impattare sulle performance della query di indicizzazione.

La struttura completa della catena delle operazioni di analisi di questi campi è visibile nei seguenti estratti dello schema mostrato nei listati 2.3 e 2.4.

```
<fieldType name="emt-technical" class="solr.TextField" positionIncrementGap="100">
    <analyzer>
        <charFilter class="solr.HTMLStripCharFilterFactory"/>
        <tokenizer class="solr.ClassicTokenizerFactory"/>
        <filter class="solr.ElisionFilterFactory" ignoreCase="true" articles="lang/contractions_it.txt"/>
        <filter class="solr.SynonymGraphFilterFactory" synonyms="synonyms_case_sensitive.txt" />
        <filter class="solr.LowerCaseFilterFactory"/>
        <filter class="solr.SynonymGraphFilterFactory" synonyms="synonyms_emt.txt" />
        <filter class="solr.ShingleFilterFactory" maxShingleSize="3" tokenSeparator="_"/>
        <filter class="solr.SynonymGraphFilterFactory" synonyms="synonyms_emt.txt" />
        <filter class="solr.KeepWordFilterFactory" words="keepwords_extracted.txt" ignoreCase="true"/>
        <filter class="solr.KeepWordFilterFactory" words="filters/technical_terms.txt" ignoreCase="true"/>
    </analyzer>
</fieldType>
```

Listato 2.3: Definizione tipo di dato per termini tecnici.

```
<fieldType name="emt-cross-technical" class="solr.TextField"
    positionIncrementGap="100">
    <analyzer>
        <charFilter class="solr.HTMLStripCharFilterFactory"/>
        <tokenizer class="solr.ClassicTokenizerFactory"/>
```

```
5         <filter class="solr.ElisionFilterFactory" ignoreCase="true" articles
      ="lang/contractions_it.txt"/>
6         <filter class="solr.SynonymGraphFilterFactory" synonyms="
      synonyms_case_sensitive.txt" />
7         <filter class="solr.LowerCaseFilterFactory"/>
8         <filter class="solr.SynonymGraphFilterFactory" synonyms="
      synonyms_emt.txt" />
9         <filter class="solr.ShingleFilterFactory" maxShingleSize="3"
      tokenSeparator="_"/>
10        <filter class="solr.SynonymGraphFilterFactory" synonyms="
      synonyms_emt.txt" />
11        <filter class="solr.KeepWordFilterFactory" words="
      keepwords_extracted.txt" ignoreCase="true"/>
12        <filter class="solr.KeepWordFilterFactory" words="filters/
      cross_technical_terms.txt" ignoreCase="true"/>
13    </analyzer>
14</fieldType>
```

Listato 2.4: Definizione tipo di dato per termini trasversali.

2.3 Relatedness query

Per scoprire le relazioni presenti tra i singoli token estratti nel passo precedente si è fatto uso della funzionalità di *relatedness* discussa nel capitolo precedente, realizzando tre query di tipo facet:

1. La prima query mette a confronto l'insieme di tutti i documenti dell'indice (insieme di background) con quello dei documenti relativi ad un progetto specificato (insieme di foreground), andando ad esaminare il campo *nota* dello schema. Il testo presente nel campo sarà diviso in token e filtrato in base ai meccanismi descritti in precedenza, per poi procedere con il conteggio delle occorrenze dei termini presenti nei due insiemi. Tali conteggi verranno poi utilizzati per calcolare la relatività dei singoli termini rispetto al progetto preso in esame. Si è deciso di mantenere i primi cinque termini risultanti in ordine decrescente di valore di relatedness. Questo valore è stato scelto dato che oltre ad esso generalmente i risultati iniziavano ad avere un grado di precisione poco soddisfacente.

2. La seconda query differisce dalla prima solo dal fatto che agisce su un diverso insieme di foreground, andando ad esaminare i documenti relativi ad un utente specifico. Questo ha lo scopo di scoprire le relazioni tra gli utenti e gli argomenti affrontati durante la realizzazione di tutti i progetti su cui hanno lavorato.

3. La terza invece è una query innestata basata sui concetti delle due precedenti, con lo scopo di assegnare ai termini più relativi di un progetto, gli utenti ad essi stetti più collegati. L'insieme di foreground è ancora quello dei documenti relativi ad un progetto specificato, e tramite la prima query si trovano i cinque termini più relativi. Con la seconda query innestata si scoprono gli utenti maggiormente associati ad ogni termine. In questo modo è possibile vedere su quali tecnologie o argomenti gli utenti hanno lavorato, non più con una visione generale, ma più nell'ambito specifico dei singoli progetti.

```
curl -sS http://192.168.150.88:8983/solr/emt-staging/query -d 'rows=0&q=*:*&back=*:*&fore=utente:"Mario Rossi"&json.facet={nota_completa:{type:terms, field: nota_completa, limit: 5, sort: {r1: desc}, facet: {r1: "relatedness($fore,$back)"}}}'
{
  "responseHeader":{
    "status": 0,
  "QTime":15,
  "params":{
    "q":"*:*",
    "json.facet":"{nota_completa:{type: terms, field: nota_completa, limit: 5, sort: {r1: desc}, facet: {r1: \"relatedness($fore,$back)\"}}}",
    "back":"*:*",
    "rows":"0",
    "fore":"utente:\"Mario Rossi\""}},
  "response":{"numFound":58863,"start":0,"numFoundExact":true,"docs":[]
  },
  "facets":{
    "count":58663,
 "nota_completa":{
   "buckets":[{
     "val":"solr",
  "count":39,
  "r1":{
    "relatedness":0.8173,
```

```
22      "foreground_popularity":6.6E-4,
23      "background_popularity":6.6E-4}},
24      {
25       "val":"semantics",
26    "count":64,
27    "r1":{
28      "relatedness":0.50343,
29      "foreground_popularity":3.1E-4,
30      "background_popularity":0,00109}},
31      {
32       "val":"search",
33    "count":63,
34    "r1":{
35      "relatedness":0.30064,
36      "foreground_popularity":1.5E-4,
37      "background_popularity":0.00107}},
38      {
39       "val":"servizi",
40    "count":3001,
41    "r1":{
42      "relatedness":0.18936,
43      "foreground_popularity":7.2E-4,
44      "background_popularity":0.05116}},
45      {
46       "val":"backend",
47    "count":1865,
48    "r1":{
49      "relatedness":0.15012,
50      "foreground_popularity":4.4E-4,
51      "background_popularity":0.03179}}]}}}
```

Listato 2.5: Esempio di query per ottenere il valore di relatedness su un utente.

```
curl -sS http://192.168.150.88:8983/solr/emt-staging/query -d 'rows=0&q=*:*&
    back=*:*&fore=codcommessa:"INV-IMOLAB"&json.facet={nota_completa:{type:
    terms, field: nota_completa, limit: 5, sort: {r1: desc}, facet: {r1: "
    relatedness($fore,$back)"}}}'
{
  "responseHeader":{
    "status": 0,
  "QTime":15,
  "params":{
    "q":"*:*",
    "json.facet":"{nota_completa:{type: terms, field: nota_completa, limit:
    5, sort: {r1: desc}, facet: {r1: \"relatedness($fore,$back)\"}}}",
    "back":"*:*",
```

```
    "rows":"0",
    "fore":"codcommessa:\"INV-IMOLAB\""}},
  "response":{"numFound":58863,"start":0,"numFoundExact":true,"docs":[]
  },
  "facets":{
    "count":58663,
 "nota_completa":{
   "buckets":[{
     "val":"solr",
  "count":39,
  "r1":{
      "relatedness":0.77224,
      "foreground_popularity":6.6E-4,
      "background_popularity":6.6E-4}},
    {
     "val":"infrastruttura",
  "count":236,
  "r1":{
      "relatedness":0.52099,
      "foreground_popularity":7.8E-4,
      "background_popularity":0,00402}},
    {
     "val":"semantics",
  "count":64,
  "r1":{
      "relatedness":0.43182,
      "foreground_popularity":6.6E-4,
      "background_popularity":6.6E-4}},
    {
     "val":"react",
  "count":140,
  "r1":{
      "relatedness":0.24194,
      "foreground_popularity":2.4E-4,
      "background_popularity":0.00239}},
    {
     "val":"search",
  "count":63,
  "r1":{
      "relatedness":0.23254,
      "foreground_popularity":1.5E-4,
      "background_popularity":0.00107}}]}}}
```


Listato 2.6: Esempio di query per ottenere il valore di relatedness su un progetto.

```
curl -sS http://192.168.150.88:8983/solr/emt-staging/query -d 'rows=0&q=*:*&
    back=*:*&fore=codcommessa:"INV-IMOLAB"&json.facet={nota_completa:{type:
    terms, field: nota_completa, limit: 3, sort: {r1: desc}, facet: {r1: "
    relatedness($fore,$back)"}, utenti: {type:terms, field: utente, limit:
    2, sort: {r2: desc}, facet: {r2: "relatedness($fore,$back)")}}}}}}'
{
  "responseHeader":{
    "status": 0,
  "QTime":15,
  "params":{
    "q":"*:*",
    "json.facet":"{nota_completa:{type: terms, field: nota_completa, limit:
    3, sort: {r1: desc}, facet: {r1: \"relatedness($fore,$back)\", utenti: {
    type:terms, field: utente, limit: 2, sort: {r2: desc}, facet: {r2: "
    relatedness($fore,$back)")}}}}}}'
    "back":"*:*",
    "rows":"0",
    "fore":"codcommessa:\"INV-IMOLAB\""}},
  "response":{"numFound":58863,"start":0,"numFoundExact":true,"docs":[]
  },
  "facets":{
    "count":58663,
 "nota_completa":{
   "buckets":[{
     "val":"solr",
  "count":39,
  "r1":{
    "relatedness":0.77224,
    "foreground_popularity":6.6E-4,
    "background_popularity":6.6E-4},
    "utenti":{
      "buckets":[{
     "val": "Mario Rossi",
     "count":39,
     "r2":{
       "relatedness":0.81734,
    "foreground_popularity":6.6E-4,
    "background_popularity":0.00116}}]}},
    {
     "val":"infrastruttura",
  "count":236,
  "r1":{
    "relatedness":0.52099,
    "foreground_popularity":7.8E-4,
    "background_popularity":0.00402},
    "utenti":{
      "buckets":[{
```

```
    "val": "Maria Bianchi",
    "count":36,
    "r2":{
      "relatedness":0.76545,
   "foreground_popularity":1.2E-4,
   "background_popularity":0.00141}},
   {
    "val": "Paolo Verdi",
    "count":7,
    "r2":{
      "relatedness":0.30209,
   "foreground_popularity":1.2E-4,
   "background_popularity":9.5E-4}}]}}},
   {
    "val":"semantics",
 "count":64,
 "r1":{
   "relatedness":0.43182,
   "foreground_popularity":3.1E-4,
   "background_popularity":0.00109},
   "utenti":{
     "buckets":[{
    "val": "Mario Rossi",
    "count":18,
    "r2":{
      "relatedness":0.73542,
   "foreground_popularity":3.1E-4,
   "background_popularity":0.00116}},
   {
    "val": "Carlo Neri",
    "count":2,
    "r2":{
      "relatedness":-0.00292,
   "foreground_popularity":0.0,
   "background_popularity":0.00382}}]}}]}}}
```


Listato 2.7: Esempio di query con funzione relatedness innestata.

I risultati ottenuti utilizzando query di tipo 1 e 2 sono stati integrati all'interno dell'interfaccia dell'applicazione, come mostrato in figura 2.1, in modo da avere un accesso rapido alle informazioni relative alla conoscenza sviluppata nei singoli progetti e dai singoli collaboratori.

Figura 2.1: Esempio risultati di una query di recupero conoscenza su un progetto, integrati su interfaccia web.

Capitolo 3

Estrazione di conoscenza: progetti software e codice

La quantità di repository è in costante crescita, e sempre più spesso emerge la necessità di utilizzare strumenti per la loro analisi. L'analisi di questi repository risulta importante per diversi fattori, come ad esempio per comprendere la struttura del codice, le sue funzionalità e complessità, la sua evoluzione o aiutare nel riuso e refactory del codice, così come identificare le relazioni tra gli esseri umani e il codice che producono. Inoltre il compito di mantenere o aggiungere funzionalità a un sistema software di grandi dimensioni richiede che gli sviluppatori abbiano una comprensione generale dell'architettura del sistema. Ma per uno sviluppatore che si avvicina a un sistema sconosciuto, acquisire anche una comprensione di alto livello di quali componenti costituiscono una parte del software e di come questi componenti lavorano insieme può essere un'attività che richiede molto tempo.

L'estrazione della conoscenza semantica sotto forma di topic viene incontro a queste necessità, permettendo la creazione di una corrispondenza tra tali topic e i concetti o funzionalità implementate dai software. Sfruttando questi topic si facilita il processo di comprensione delle caratteristiche del software, permettendo agli utenti di apprendere più rapidamente il funzionamento e la struttura dei sistemi anche in mancanza di documentazione

esaustiva.

Pensiamo però che esaminando solo il contenuto dei file di codice si possa osservare solo una visione parziale della conoscenza di un progetto, quindi risulta utile e necessario osservare diverse fonti. In particolare si è scelto di utilizzare anche la conoscenza che è presente nei messaggi di commit, in quanto spesso contiene informazioni aggiuntive su come gli sviluppatori hanno lavorato, le decisioni che hanno preso, tenendo traccia dell'intera evoluzione del progetto. Inoltre è possibile estrarre relazioni di associazione tra il lavoro svolto e chi lo ha effettivamente svolto in modo più preciso.

Nel seguente capitolo andremo ad illustrare il processo con cui viene estratta la conoscenza direttamente dal codice e dal testo presente nei messaggi di commit dei progetti software.

3.1 Indicizzazione commit e codice

La risoluzione di questo caso di studio è iniziata con l'analisi e la creazione dei dataset contenenti la conoscenza relativa ai repository di codice. In particolare tramite analisi iniziali sono stati determinate due aree principali da cui poter estrarre la conoscenza: lo storico dei commit di progetto e i file di codice stessi. Il primo gruppo è stato scelto in quanto solitmente contiene le informazioni più dettagliate sugli argomenti e le metodologie utilizzate all'interno di un progetto, mentre il secondo contiene conoscenza più specifica riguardo a linguaggi e tecnologie.

Si è deciso quindi di analizzare ogni progetto (scelti in modo casuale da github) tramite un batch Spring-Boot in Java. Il processo può operare in due modi, definiti da un unico parametro passato tramite linea di comando:

- Commit: il processo recupera i dati relativi allo storico dei commit del progetto preso in esame, tramite le API messe a disposizione da Github, per poi mapparle all'interno di un file .csv (di cui un frammento è visibile in figura 3.1 utilizzando la libreria *org.apache.commons.commons-csv*.

- Codice: viene scaricato da github il codice sorgente del progetto (utilizzando il branch master) e successivamente si percorre il file system esaminando i file, prendendo in considerazione solo determinate estensioni, e compilando così un altro file .csv che andrà a contenere tutti i file da indicizzare associati al nome del rispettivo progetto.

Una volta estratti i dati si è sperimentato con due tecniche per l'indicizzazione verso Solr, la prima basata sull'uso di Apache Spark, e la seconda sfruttando la libreria java SolrJ.

	A	B	C
1	project	author	message
2	Aerial	Guillaume Louel	Fix playback speed slider not showing on videos (again)
3	Aerial	Guillaume Louel	Fix local repository update to latest beta
4	Aerial	Guillaume Louel	Merge pull request #1292 from dnicolson/fix-window-sizing Fix window constraints
5	Aerial	Guillaume Louel	Merge pull request #1290 from dnicolson/patch-1 Add spaces after colons
6	Aerial	Dave Nicolson	Add spaces after colons
7	Aerial	Dave Nicolson	Prevent filter info text truncation
8	Aerial	Dave Nicolson	Fix credits height constraint
9	Aerial	Dave Nicolson	Add fixed or greater than window size constraints
10	Aerial	Guillaume Louel	Merge pull request #1287 from danchr/secrets Unbreak build straight from Git

Figura 3.1: Esempio di file .csv contentente i documenti relativi ai commit su un progetto.

3.1.1 Indicizzazione commit

Per indicizzare il contenuto dei commit è stato fatto uso di Spark, in particolare della libreria Spark-Solr descritta nei precedenti capitoli. Nello specifico l'operazione inizia con l'upload della cartella contenente tutti i file .csv su uno dei nodi worker del cluster spark, per poi eseguire su di esso lo script *spark-shell*, importando il jar *Spark-Solr* assieme ad altri jar di supporto alla libreria, eseguendo un semplice script Scala contenente le istruzioni per l'indicizzazione (codice in figura 3.2).

```
1 ./spark-shell --master spark://192.168.151.192:7077 --jars /home/spark/spark
    -solr-4.0.0-shaded.jar,/home/spark/jetty-util-9.4.39.v20210325.jar,/home
    /spark/jetty-io-9.4.39.v20210325.jar,/home/spark/jetty-client-9.4.39.
    v20210325.jar,/home/spark/jetty-http-9.4.39.v20210325.jar --conf 'spark.
```

```
    driver.extraJavaOptions=-Dbasicauth=solr:SolrRocks' --conf 'spark.
    executor.extraJavaOptions=-Dbasicauth=solr:SolrRocks' -i /home/spark/
    index-commits.scala
```

Listato 3.1: Esecuzione script spark-shell.

```
import org.apache.spark.sql.types._

val csvFileLocation = "/home/spark/Solr-Repos"
val customSchema = StructType(Array(
  StructField("project", StringType, true),
  StructField("author", StringType, true),
  StructField("message", StringType, true),
  StructField("date", StringType, true)))

var csvDF = spark.read.format("csv").option("header", "true").option("
    multiLine",true).schema(customSchema).load(csvFileLocation)

val options = Map(
  "zkhost" -> "192.168.151.243:2181,
              192.168.151.245:2181,
              192.168.151.246:2181",
  "collection" -> "commits",
  "commit_within" -> "5000"
)

// Write to Solr
csvDF.write.format("solr").options(options).mode(org.apache.spark.sql.
    SaveMode.Overwrite).save
```

Listato 3.2: Contenuti del file index-commits.scala.

Gli executor Spark ricevono l'istruzione di leggere riga per riga ogni file .csv contenuti all'interno della cartella specificata, seguendo lo schema definito (che è una copia speculare dello schema definito all'interno della collezione Solr), e inserine i dati all'interno di un DataFrame. Un'accorgimento molto importante è stato quello dell'utilizzo dell'opzione *multiline* per indicare che ciascun campo del .csv può essere composto da stringhe su più di una linea, e quindi non considerare l'andata a capo come un nuovo record. Questo è stato necessario in quanto il testo dei commit è spesso sviluppato su multiple righe, e senza di esso il processo di lettura e trasformazione in DataFrame non riportava i risultati corretti.

Una volta ottenuto il DataFrame gli executor recuperano i metadata della collezione tramite Zookeeper e, tramite le API Spark-Solr, effettuano l'operazione di scrittura verso di essa, sovrascrivendo ogni i dati precedenti nel caso fossero già presenti così che ogni indicizzazione possa procedere in modo incrementale rispetto alla precedente.

Per la configurazione Solr come prima cosa sono stati definiti all'interno del *managed-schema* i campi relativi ai commit, come mostrato nel listati 3.3 e, analogalmente al lavoro svolto per EMT, è stato definto il tipo di dato specifico per il campo che andrà a contenere il testo dei commit.

```
<field name="project" type="string" indexed="true" stored="true"/>
<field name="author" type="string" indexed="true" stored="true"/>
<field name="message" type="commit" indexed="true" stored="true"/>
<field name="date" type="rdate" uninvertible="false"/>
```

Listato 3.3: Schema documenti dei commit dei progetti.

Al termine del processo dell'intero dataset dei commit, l'ammontare di documenti indicizzati all'interno di Solr è di 164502.

3.1.2 Indicizzazione file di codice

Per effettuare l'indicizzazione del contenuto dei file di codice si è deciso di evitare l'utilizzo di Spark, ma di usufruire direttamente delle API della libreria SolrJ.

Il primo passo è stato quello di generare la lista completa dei file per l'indicizzazione. Durante questa prima fase si percorre iterativamente il file-system a partire da una cartella contenete il codice sorgente di tutti i progetti scaricati da Github.

```
@Service
@Slf4j
public class CodeController {

    private String currProjectName;

    private ArrayList<String> availableExtensions = new ArrayList();

    public void cycleCodeFolders() throws IOException {
```

```
10
11          availableExtensions.add("java");
12          availableExtensions.add("cs");
13          availableExtensions.add("c");
14          availableExtensions.add("cc");
15          availableExtensions.add("cpp");
16          availableExtensions.add("h");
17          availableExtensions.add("js");
18          availableExtensions.add("ts");
19          availableExtensions.add("kt");
20          availableExtensions.add("scala");
21          availableExtensions.add("py");
22          availableExtensions.add("lua");
23          availableExtensions.add("rs");
24          availableExtensions.add("go");
25
26          String csvFile = "E:\\Tesi Magistrale\\CodeFileList.csv";
27          BufferedWriter writer = Files.newBufferedWriter(Paths.get(csvFile));
28
29          CSVPrinter csvPrinter = new CSVPrinter(writer, CSVFormat.DEFAULT
30                  .withHeader("fileUrl","project"));
31
32
33          File dir = new File("E:\\Tesi Magistrale\\Solr-Code\\");
34          for (File file : dir.listFiles()
35          ) {
36              if (file.isDirectory()) {
37                  currProjectName = file.getName();
38
39                  try {
40                      cycleProjectFiles(file.listFiles(), csvPrinter);
41                  } catch (SolrServerException e) {
42                      csvPrinter.close();
43                      throw new RuntimeException(e);
44                  }
45              }
46          }
47          csvPrinter.close();
48      }
49
50
51      public void cycleProjectFiles(File[] leaves, CSVPrinter csvPrinter)
        throws IOException, SolrServerException {
52          for (File file : leaves) {
53              if (file.isDirectory()) {
54                  cycleProjectFiles(file.listFiles(), csvPrinter);
55              } else {
```

```
56                  String fileExtension = FilenameUtils.getExtension(file.
    getName());
57                  if (availableExtensions.contains(fileExtension)) {
58
59                      csvPrinter.printRecord(file.getAbsolutePath(),
    currProjectName);
60                      csvPrinter.flush();
61
62                  }
63              }
64          }
65      }
66  }
```

Listato 3.4: Indicizzazione file di codice.

Il processo esaminerà ogni nodo del filesystem, prendendo come radice la cartella del progetto in esame e considerando i file di codice come nodi foglia. I file vengono poi effettivamente presi in considerazione solo se l'estensione combacia con una di quelle inserite nella lista di estensioni supportate. Questa lista mira ad escludere file che non rientrano nel caso di studio, perché in formati che normalmente non contengono la conoscenza che interessa al nostro caso di studio o perché scritti in linguaggi troppo diversi da quelli scelti.

Se il file preso in considerazione rispetta i criteri verrà inserito l'uri all'interno di un file .csv, per poter poi essere indicizzato durante la seconda fase del processo. Durante questa seconda fase, per ogni riga del file .csv, verrà creato un oggetto *SolrInputDocument* che andrà a contenere i campi del documento da indicizzare. Il contenuto di questi documenti segue lo schema presentato nel listato 3.5.

```
1 <field name="project" type="string" indexed="true" stored="true"/>
2 <field name="name" type="string" indexed="true" stored="true"/>
3 <field name="extension" type="string" indexed="true" stored="true"/>
4 <field name="code" type="content_code" indexed="true" stored="true"/>
5 <field name="comments" type="content_comment" indexed="true" stored="true"/>
```

Listato 3.5: Struttura documento Solr per file di codice.

Per lo schema dei documenti relativi ai file sorgenti si è deciso di fare distinzione tra due tipologie di contenuti principali:

- L'effettivo codice sorgente, che contiene conoscenza legata ai package utilizzati, evidenziando così eventuali tecnologie o librerie utilizzate. Inoltre può contenere conoscenza legata al dominio applicativo, solitamente contenuta nei nomi dati a classi, funzioni e variabili. Normalmente scritti usando linguaggi di programmazione.

- I commenti del codice, che possono contenere documentazione e elementi vari a discrezione dei programmatori, e possono contenere sia linguaggio naturale che linguaggi di programmazione.

Per effettuare la suddivisione del contenuto dei file nelle due tipologie è stato applicato un passo di pre-processing in cui il contenuto del file viene sottoposto a due espressioni regolari, come mostrato nel listato 3.6.

```
1  private FileContents separateFileContents(String originalInput) {
2      int lastIndex = 0;
3      StringBuilder comments = new StringBuilder();
4      StringBuilder codeMinusSingleLineComm = new StringBuilder();
5      StringBuilder code = new StringBuilder();
6      FileContents contents = new FileContents();
7      contents.setCodeContent("");
8      contents.setCommentContent("");
9
10     Pattern singleLineCommentRegex = Pattern.compile("\\s*(?<!:|"|')
       (\\/\\/|#)[^\\n]+(|\\*\\))$",Pattern.MULTILINE);
11     Matcher singleLineMatcher = singleLineCommentRegex.matcher(originalInput
       );
12
13     while (singleLineMatcher.find()) {
14         codeMinusSingleLineComm.append(originalInput, lastIndex,
       singleLineMatcher.start());
15         comments.append(originalInput, singleLineMatcher.start(),
       singleLineMatcher.end());
16         lastIndex = singleLineMatcher.end();
17
18     }
19     if (lastIndex < originalInput.length()) {
20         codeMinusSingleLineComm.append(originalInput, lastIndex,
       originalInput.length());
21     }
22
23     Pattern multiLineCommentRegex = Pattern.compile("\\/\\*((?:(?!\\*\\/)
       .|\\s)*+)\\*\\/");
```

```
24      Matcher multiLineMatcher = multiLineCommentRegex.matcher(
        codeMinusSingleLineComm);
25
26      lastIndex = 0;
27      while (multiLineMatcher.find()) {
28          code.append(codeMinusSingleLineComm, lastIndex, multiLineMatcher.
        start());
29          comments.append(codeMinusSingleLineComm, multiLineMatcher.start(),
        multiLineMatcher.end());
30          lastIndex = multiLineMatcher.end();
31
32      }
33
34      if (lastIndex < codeMinusSingleLineComm.length()) {
35          code.append(codeMinusSingleLineComm, lastIndex,
        codeMinusSingleLineComm.length());
36      }
37      contents.setCodeContent(code.toString());
38      contents.setCommentContent(comments.toString());
39
40      return contents;
41
42  }
```

Listato 3.6: Funzione di pre-processing del contenuto dei file sorgente.

Il processo è a sua volta suddiviso in due fasi definite dall'applicazione delle due espressioni regolari. La prima:

```
\s*(?<!:|"|')(\/\/|#)[^\n]+(|\*\))$
```

si occupa di individuare i commenti su linea singola, seguendo i pattern utilizzati in linguaggi come Java, C, C# Javascript e Python. L'espressione presenta alcuni accorgimenti in modo da evitare di effettuare match con dei falsi positivi (come ad esempio le stringhe riguardanti gli web url, o stringhe che terminano in "*/"). Il pattern viene compilato specificando il modificatore *multiline*, in questo modo l'espressione cercherà i match su ogni singola riga, individuando ogni possibile match.

La seconda invece:

```
\/\*((?:(?!\*\/).|\s)*+)\*\/
```

ha l'obiettivo di individuare i blocchi di commento su multipla linea, delimitati dalle keyword "/*" e "*/".

Ogni match individuato dalle due espressioni verrà inserito all'interno della stringa dei commenti, mentre tutto il contenuto rimanente sarà inserito all'interno della stringa contenente il codice sorgente.

Le due espressioni regolari suddividono in modo accettabile buona parte del contenuto dei file del dataset, ma possono produrre suddivisioni errate dovute principalmente dal fatto che ogni linguaggio segue una propria convenzione per la scrittura dei commenti e, sebbene spesso siano molto simili, ogni file presenta delle differenze di stile dovute anche ai diversi autori che li hanno scritti. Ad ogni modo pensiamo che possano servire come base per la suddivisione dei due contenuti e che con ulteriori iterazioni, dopo aver esaminato i diversi casi anomali, si possano ottenere risultati migliori e meno proni a falsi positivi.

Una volta effettuato il passo di pre-processing l'algoritmo provvederà a creare il *SolrInputDocument* seguendo lo schema definito nella configurazione della collezione Solr e definendo un id univoco incrementato ad ogni iterazione. Ognuno di questi oggetti verrà poi aggiunta al *CloudSolrClient* per l'indicizzazione tramite il metodo add.

Al termine dell'iterazione della lista verrà eseguito un comando esplicito per effettuare il commit dei file indicizzati, in modo da migliorare le performance.

```
1  public void uploadFileList(CloudSolrClient solr, Map<String,String> fileMap)
       {
2      int documentId = 0;
3      for (String fileUrl : fileMap.keySet()
4      ) {
5          documentId++;
6          File codeFile = new File(fileUrl);
7          SolrInputDocument projectDocument = new SolrInputDocument();
8
9          projectDocument.addField("id", documentId);
10         projectDocument.addField("project", fileMap.get(fileUrl));
11
12         String title = FilenameUtils.getName(codeFile.getName());
```

```
13          String fileExtension = FilenameUtils.getExtension(codeFile.getName()
    );
14          try {
15              byte[] contentBytes = Files.readAllBytes(codeFile.toPath());
16              log.info("File esaminato {}", codeFile.getAbsolutePath());
17
18              String contentString = StringUtils.newStringUtf8(contentBytes);
19
20
21              FileContents contents = separateFileContents(contentString);
22              projectDocument.addField("name", title);
23              projectDocument.addField("extension", fileExtension);
24              projectDocument.addField("code", contents.getCodeContent());
25              projectDocument.addField("comment", contents.getCommentContent()
    );
26
27              solr.add(projectDocument);
28
29          } catch (Exception e) {
30              e.printStackTrace();
31          }
32      }
33  }
```

Listato 3.7: Funzione di upload verso Solr dei documenti legati ai file sorgenti.

Al termine del processo sono stati indicizzati 29006 documenti.

3.2 Querying

Dopo aver effettuato l'indicizzazione di entrambi i tipi di informazione si è proseguito con l'esaminazione dei termini e dei collegamenti che compongono la conoscenza dei progetti.

Come prima cosa è stato necessario definire, anche in questo caso, un tipo di dato personalizzato all'interno dello schema delle collezioni Solr. Nello specifico sono stati definiti 3 tipi in totale : uno per modellare il contenuto dei commit, e due utilizzati per modellare il contenuto dei file sorgente, presenti nei listato 3.8 e 3.9 rispettivamente.

Messaggi di commit

```
<fieldType name="commit" class="solr.TextField" positionIncrementGap="100">
  <analyzer>
      <charFilter class="solr.HTMLStripCharFilterFactory"/>
              <tokenizer class="solr.ClassicTokenizerFactory"/>
              <filter class="solr.WordDelimiterGraphFilterFactory"
    splitOnCaseChange="0" splitOnNumerics="0"/>
              <filter class="solr.FlattenGraphFilterFactory"/>
              <filter class="solr.LowerCaseFilterFactory"/>
              <filter class="solr.PatternReplaceFilterFactory" pattern="(^\d
    +$)" replacement="" replace="all" />
              <filter class="solr.StopFilterFactory" words="stopwords_it_en.
    txt"/>
              <filter class="solr.ShingleFilterFactory" maxShingleSize="3"
    tokenSeparator="_"/>
  </analyzer>
</fieldType>
```

Listato 3.8: Catena di filtri per l'analisi del contenuto dei messaggi di commit.

La catena di filtri utilizzata per l'analisi del testo dei messaggi di commit è molto simile a quella utilizzata nel caso di studio delle note di consuntivazione, in quanto espresse sempre in linguaggio naturale e relative ad un simile dominio.

Il processo di analisi del contenuto dei messaggi di commit segue i seguenti punti:

1. Eliminazione di ogni carattere html (come ad esempio \n, \r, \b etc..) in modo che non vadano ad interferire con la catena di filtri.

2. Applicazione di un *ClassicTokenizer*, mantenendo così all'interno di singoli token url ed indirizzi email.

3. Applicazione di un *WordDelimiterGraphFilter*, per suddividere i token all'occorrenza di delimitatori specifici, nel nostro caso all'occorrenza di una lettera maiuscola all'interno dei token che seguono il pattern *CamelCase*, ma evitando di separare numeri. Questo filtro necessita di essere seguito da un *FlattenGraphFilter* in modo da permettere a Solr di effettuare correttamente l'indicizzazione dei token.

4. Passaggio di tutti i token in minuscolo tramite un *LowerCaseFilter*.

5. Rimozione di tutti i token numerici, applicando un *PatternReplaceFilter*, e di congiunzioni e preposizioni della lignua italiana ed inglese.

6. Creazione di token composti tramite l'applicazione di uno *ShingleFilter*, così da poter catturare termini che presi singolarmente non portano informazioni, ma che accostati gli uni agli altri possono rappresentare concetti o tecnologie.

File Sorgente

Come discusso nella sezione precedente si è deciso di suddividere il contenuto dei file sorgente in due categorie, l'effettivo codice sorgente ed il testo dei commenti. Questa decisione si è riflessa lato Solr con la definizione di due tipi di dato, ognuno definito da una propria catena di filtri specifici per poterli analizzare al meglio.

Le due tipologie seguono uno schema molto simile, ma che differiscono nella scelta del tokenizer utilizzato. Questa scelta è stata fatta per mantenere intatti token composti, come indirizzi email o separati da simboli come ad esempio "-". L'impatto di questa scelta risulta comunque minimo.

```
<fieldType name="content_code" class="solr.TextField" positionIncrementGap="100">
  <analyzer>
      <charFilter class="solr.HTMLStripCharFilterFactory"/>
      <tokenizer class="solr.StandardTokenizerFactory"/>
      <filter class="solr.WordDelimiterGraphFilterFactory" splitOnNumerics="0"/>
      <filter class="solr.FlattenGraphFilterFactory"/>
      <filter class="solr.LowerCaseFilterFactory"/>
      <filter class="solr.PatternReplaceFilterFactory" pattern="(^\d+$)" replacement="" replace="all" />
      <filter class="solr.StopFilterFactory" words="stopwords_it_en.txt" />
      <filter class="solr.StopFilterFactory" words="programming_lang_keyword_remover.txt"/>
  </analyzer>
</fieldType>

<fieldType name="content_comment" class="solr.TextField" positionIncrementGap="100">
  <analyzer>
```

```
16          <charFilter class="solr.HTMLStripCharFilterFactory"/>
17          <tokenizer class="solr.ClassicTokenizerFactory"/>
18          <filter class="solr.WordDelimiterGraphFilterFactory"
    splitOnNumerics="0"/>
19          <filter class="solr.FlattenGraphFilterFactory"/>
20          <filter class="solr.LowerCaseFilterFactory"/>
21          <filter class="solr.PatternReplaceFilterFactory" pattern="(^\d+$)"
     replacement="" replace="all" />
22          <filter class="solr.StopFilterFactory" words="stopwords_it_en.txt"
    />
23        </analyzer>
24</fieldType>
```

Listato 3.9: Catena di filtri per l'analisi del contenuto dei file di codice.

A differenza del caso di studio delle consuntivazioni EMT, queste catene di filtri implementano un'analisi di tipo general purpose. Per ottenere i risultati ottimali sarebbe necessario sviluppare dizionari domain specific per ottenere solamente i termini specifici dei domini applicativi e raggruppare eventuali sinonimi. La realizzazione di questi dizionari richiede tempo, per essere migliorati in numerose iterazioni, ed un'ampia conoscenza dei domini di applicazione, percui si è deciso di rimandare la loro realizzazione. Le query di estrazioni dei termini hanno prodotto risultati soddisfacenti, nonostante la mancanza dei dizionari, producendo termini coerenti con il dominio preso in esame, come mostrato in tabella 3.1.

Semantic Knowledge Graph

Dopo aver verificato il funzionamento soddisfacente dei filtri si è passati ad esaminare alcune possibili relazioni presenti all'interno del dataset, sempre utilizzando la funzionalità di *relatedness* per la costruzione di un Semantic Knowledge graph. Come nel caso di studio delle consuntivazioni su EMT, anche qui è stato possibile effettuare query con diversi set di *foreground*, come ad esempio i singoli progetti o gli autori dei messaggi di commit, come mostrato nel listato 3.10.

Le query seguono tutte le strutture definite nel capitolo 2, utilizzando query di tipo facet su cui poi viene appliccato il calcolo del valore di *related-*

Commit		Codice		Commenti	
Termini	**Occorrenze**	**Termini**	**Occorrenze**	**Termini**	**Occorrenze**
fix	21628	file	12330	file	8499
https	15164	name	11123	copyright	8271
github	11835	java	10623	license	8034
build	6147	copy	10453	use	7102
tests	6062	util	10322	reserved	6110
release	5713	value	10015	rights	6067
branch	4666	http	9325	copy	5989
amazon	4070	license	9284	http	5977
cache	2418	copyright	8910	apache	5804
android	2132	software	8676	specific	5804
windows	1977	implied	8584	software	5624
remote	1896	distributed	8551	language	5607
maven	1822	language	8307	except	5606
kernel	1771	specific	8208	writing	5592
refactor	1700	kind	8198	kind	5582
values	1696	except	8191	implied	5579
gl	1670	list	8078	authors	5571
log	1648	licenses	8019	licenses	5561
server	1558	licensed	7873	distributed	5558
net	1374	writing	7849	permissions	5548
toolchain	1261	apache	7848	express	5540
git	1205	conditions	7802	obtain	5533
ui	1173	permissions	7752	applicable	5522
client	1161	applicable	7688	warranties	5510
bit	1128	create	7684	basis	5498
header	1124	express	7667	limitations	5497
redisson	1087	obtain	7661	licensed	5494
event	1063	warranties	7596	law	5492
integration	1052	basis	7592	compliance	5488

Tabella 3.1: Primi trenta termini con più occorrenze per i tre tipi di documento indicizzati.

ness, ottenendo i primi cinque elementi più legati all'insieme di foreground definito.

```
curl -sS -X POST --user solr:SolrRocks http://natti-solrcloud.imolab.it:8983/solr/commits/query -d 'rows=0&q=*:*&back=*:*&fore=author:"nicolaAtti"&json.facet={message:{type: terms, field: message, limit: 5, sort: {r1: desc}, facet: {r1: "relatedness($fore,$back)"}}}'
{
"responseHeader":{
"zkConnected":true,
"status":0,
"QTime":268,
"params":{
  "q":"*:*",
  "json.facet":"{message:{type: terms, field: message, limit: 5, sort: {r1: desc}, facet: {r1: \"relatedness($fore,$back)\"}}}",
  "back":"*:*",
  "rows":"0",
  "fore":"author:\"nicolaAtti\""}},
"response":{"numFound":164502,"start":0,"maxScore":1.0,"numFoundExact":true,"docs":[]
},
"facets":{
"count":164502,
"message":{
  "buckets":[{
      "val":"leaflet",
      "count":3,
      "r1":{
        "relatedness":0.17468,
        "foreground_popularity":1.0E-5,
        "background_popularity":4.0E-5}},
    {
      "val":"webapp",
      "count":41,
      "r1":{
        "relatedness":0.14286,
        "foreground_popularity":2.0E-5,
        "background_popularity":2.5E-4}},
    {
      "val":"branch",
      "count":4666,
      "r1":{
        "relatedness":0.13347,
        "foreground_popularity":2.0E-4,
        "background_popularity":0.02836}},
    {
      "val":"github",
```

```
41        "count":11835,
42        "r1":{
43          "relatedness":0.06918,
44          "foreground_popularity":1.9E-4,
45          "background_popularity":0.07194}},
46      {
47        "val":"draft",
48        "count":32,
49        "r1":{
50          "relatedness":0.05885,
51          "foreground_popularity":1.0E-5,
52          "background_popularity":3.9E-4}}]}}}
```

Listato 3.10: Query relatedness per autore.

Sono state effettuate numerose query come quelle dell'esempio, ed in tutte sono emersi numerosi termini pertinenti (definiti da un valore di relatedness abbastanza alto), tutto ciò nonostante le limitazioni dovute alla mancanza dei dizionari domain specific definiti nel paragrafo precedente.

Alcuni dei Semantic Knowledge Graph prodotti da alcune query di test sono visualizzati nelle immagini seguenti.

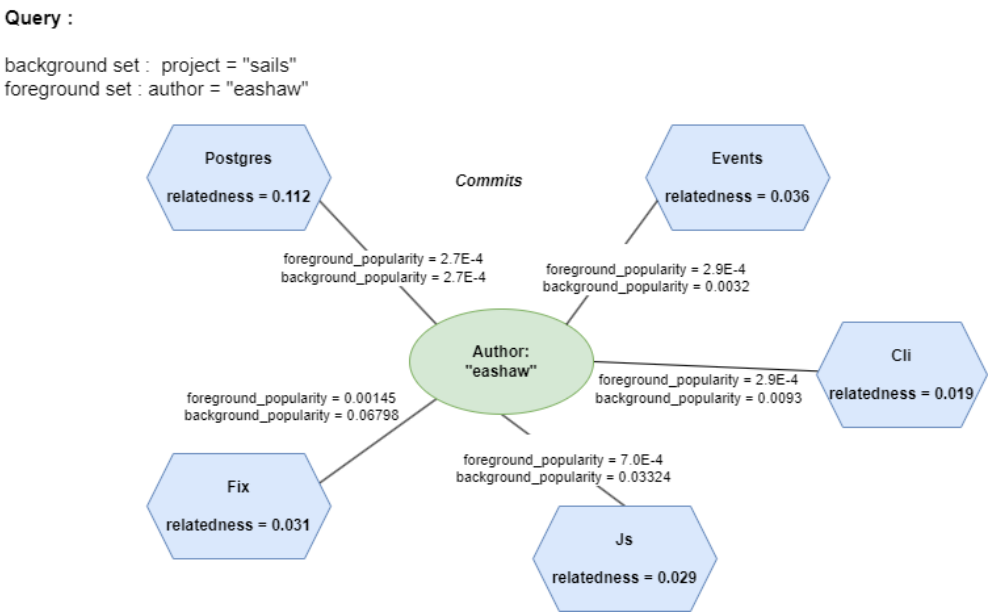

Figura 3.2: Query per i cinque termini più relativi ad un determinato utente su un singolo progetto, sulla base dei commit.

In figura 3.2 in particolare viene mostrato il risultato di una query che mostra i primi cinque termini, presenti all'interno dei messaggi dei commit, maggiormente in relazione con un determinato utente su un singolo progetto. Si può notare come il termine *Postgres* possiede la stessa popolarità sia nella query di background (cioè tutti i commit sul progetto) che in quella di foreground (tutti i commit dell'utente), producendo così un valore di relatedness significativo, attribuendo così ogni utilizzo della tecnologia completamente all'utente.

Gli altri termini al contrario, sebbene abbiano un'elevata popolarità, raggiungo livelli di relatedness molto più bassi a causa di un elevato numero di occorrenze nell'insieme di background, ciò significa che l'utente è collegato ai termini evidenziati, ma non in modo esclusivo come per il primo termine.

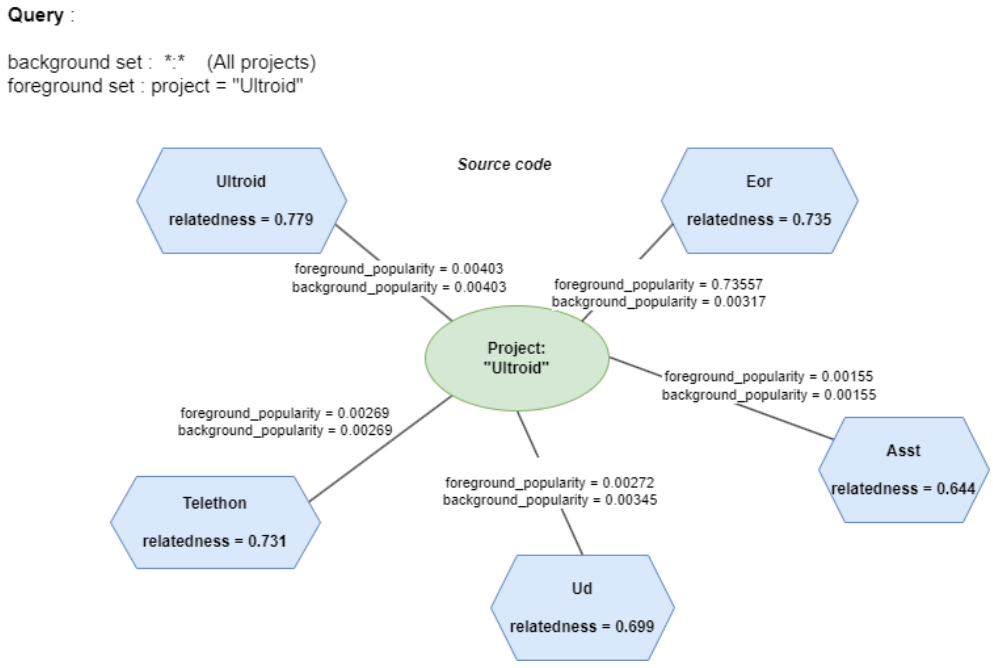

Figura 3.3: Query per i cinque termini più relativi ad un determinato progetto, sulla base del codice sorgente.

Nelle figure 3.3 e 3.4 vengono mostrate le query effettuate per la collezione contenente il codice sorgente e per i commenti rispettivamente. Dalle query

effettuate si nota che le informazioni ottenute esaminando il codice sorgente sono molto impattate dall'utilizzo o meno di un dizionario ben costruito, in quanto le query producono all'interno dei risposta molti più termini difficilmente riconducibili al dominio (o alcune volte non correlati) rispetto alle query sui messaggi di commit. In particolare questo accade maggiormente quando si effettuano query sul campo del commenti.

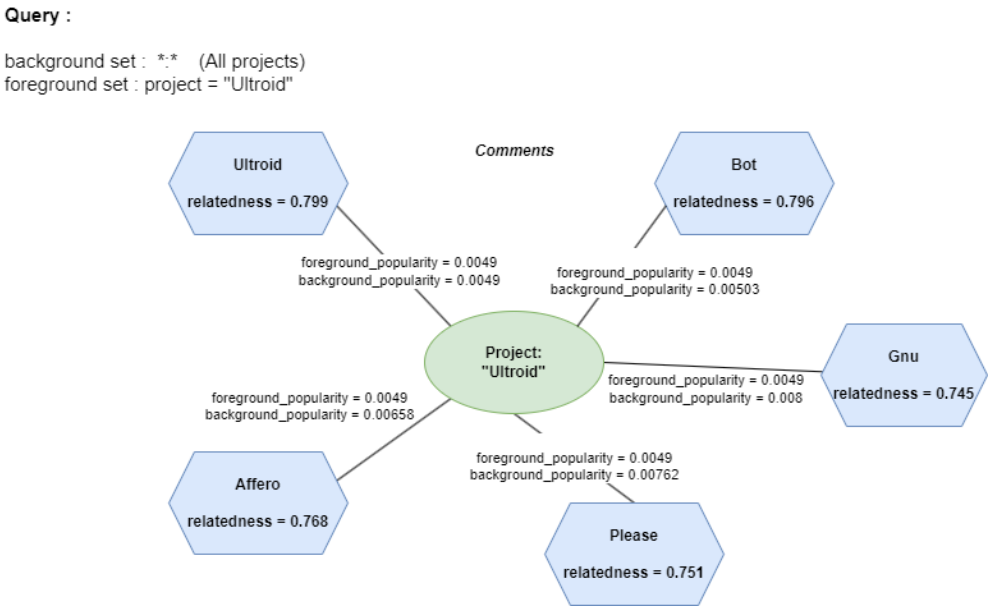

Figura 3.4: Query per i cinque termini più relativi ad un determinato progetto, sulla base dei commenti.

Ad ogni modo questo non impedisce di far affiorare dove possibile termini corretti, qualora il valore di relatedness fosse abbastanza alto, come mostrato nelle due figure, ipotiziamo quindi che con la realizzazione dei dizionari entrambi i tipi di dato possono essere sfruttati per il recupero di informazioni.

Conclusioni

In conclusione l'utilizzo di Solr come strumento di estrazione della conoscenza si è dimostrato abbastanza efficace nello scoprire informazioni rilevanti esaminando vasti dataset. La sua capacità di recuperare informazioni rilevanti si è rivelata efficace per i casi d'uso presi in considerazione.

Tuttavia, risulta evidente che i suoi risultati sono intrinsecamente vincolati dall'utilizzo di dizionari specializzati. Questa limitazione mostra l'importanza di dotare Solr di terminologia e contesto specifici del dominio, migliorando senza dubbio la sua capacità di estrarre conoscenza con precisione e profondità. Tuttavia, con le giuste risorse e configurazioni, Solr rimane uno strumento promettente per la scoperta della conoscenza, aprendo nuove strade per il recupero e l'analisi delle informazioni in numerosi contesti.

Bibliografia

[1] https://solr.apache.org/guide/8_9/ Reference guide Apache Solr.

[2] https://www.databricks.com/session/integrating-spark-and-solr Integrating Spark and Solr.

[3] https://spark.apache.org/docs/latest/index.html Documentazione Apache Spark.

[4] https://github.com/lucidworks/spark-solr Documentazione libreria Spark-Solr

[5] https://www.slideshare.net/treygrainger/searching-on-intent-knowledge-graphs-personalization-and-contextual-disambiguation Searching on Intent: Knowledge Graphs, Personalization, and Contextual Disambiguation

[6] https://lucidworks.com/post/poor-mans-entity-extraction-with-solr/ How to Perform Entity Extraction in Solr

[7] Grainger, T., AlJadda, K., Korayem, M. & Smith, A. The Semantic Knowledge Graph: A compact, auto-generated model for real-time traversal and ranking of any relationship within a domain. (2016)

[8] Maskeri, G., Sarkar, S. & Heafield, K. Mining Business Topics in Source Code using Latent Dirichlet Allocation. *Proceedings Of The 2008 1st India Software Engineering Conference, ISEC'08*. pp. 113-120 (2008,2)

[9] Ma, H., Du, W., Xu, S. & Li, W. Searching Tourism Information by Using Vertical Search Engine Based on Nutch and Solr. *2019 IEEE 17th International Conference On Software Engineering Research, Management And Applications (SERA)*. pp. 128-132 (2019)

[10] Chantamunee, S., Fung, C., Wong, K. & Dumkeaw, C. Knowledge Discovery from Thai Research Articles by Solr-Based Faceted Search. *Recent Advances In Information And Communication Technology 2018*. pp. 337-346 (2019)

[11] `https://www.slideshare.net/treygrainger/relevance-of-the-apache-solr-semantic-knowledge-graph` Relevance of the Apache Solr Semantic Knowledge Graph

[12] `https://www.slideshare.net/arafalov/searching-for-ai-leveraging-solr-for-classic-artificial-intelligence-tasks` Searching for AI - Leveraging Solr for classic artificial intelligenge tasks

[13] `https://www.slideshare.net/treygrainger/searching-on-intent-knowledge-graphs-personalization-and-contextual-disambiguation?next_slideshow=1` Searching on intent: Knowledge Graphs, personalization and contextual disambiguation

[14] `https://towardsdatascience.com/knowledge-graphs-in-natural-language-processing-acl-2020-ebb1f0a6e0b1` Knowledge Graphs in Natural Language Processing