UNIVERSIDADE DO MINHO
Departamento de Informática
Mestrado integrado em Engenharia Informática

Scripting no Processamento de Linguagem Natural

Elasticsearch's full text queries

Grupo 13

Daniel Fernandes (a78377)
Maria Helena Poleri (a78633)
Mariana Miranda (a77782)

Novembro 2018
Braga

Resumo

O presente documento tem como objetivo a realização de uma análise da ferramenta *Elasticsearch* focada essencialmente no sub-tópico *full text queries*. Tal análise passará por um breve descrição da ferramenta e elucidação dos casos de uso através de um cenário de teste ao qual dedicaremos o nosso estudo.

Capítulo 1

Introdução

O *Elasticsearch* é uma ferramenta de *full text search*, isto é, permite realizar procuras de acordo com um determinado critério sobre um conjunto de documentos. Possui ainda as vantagens de ser *open-source*, altamente escalável e funciona ainda como motor analítico.

Os casos de usos onde este tipo de ferramenta pode ser aplicado é muito vasto e, portanto, é nosso objetivo num capítulo inicial explorar todas estas aplicações, bem como dar a conhecer a ferramenta em si e identificar o seu potencial.

Sendo *open-source*, é possível fazer uma instalação local da ferramenta, contudo, não possuindo os recursos necessários e de forma a garantir escalabilidade, está adaptada para utilização distribuída em *cloud* com milhares de servidores, sendo utilizadas técnicas de *sharding* e replicação.

Numa segunda parte do relatório, tendo-se descrito a ferramenta e apresentado os seus *use cases*, dar-se-á início a uma vertente mais prática, onde exemplificaremos uma possível utilização da mesma, pela exposição de um caso de teste.

Capítulo 2

Descrição da Ferramenta

2.1 Full Text Search

O *Elasticsearch* é um motor distribuído de pesquisa e análise baseado no *Apache Lucene*, que através de APIs RESTful e JSON, permite não só armazenar documentos em formato JSON de uma forma não relacional, assim como, graças à sua alta escalabilidade, efetuar pesquisas em grandes volumes de dados praticamente em tempo real. Pode ainda ser integrado em várias linguagens tais como Java, Python, .NET, SQL, e PHP, o que em conjunto com o que foi referido anteriormente, contribuiu para as várias razões pela qual é utilizado hoje em dia por vários sistemas conhecidos tais como o Facebook, Accenture, eBay, Microsoft, GitHub, entre outros.

Apesar das várias funcionalidades que esta ferramenta oferece, vamos focar-nos numa das mais importantes: *full text search*, mais especificamente nas *full text queries*.

No entanto, antes de abordar em detalhe as *queries* é importante refletir como é que é feita a análise e o cálculo da relevância de um documento.

2.1.1 Análise

A análise de um documento consiste em converter o texto em *tokens* distintos e normalizados de forma a criar um *inverted index*, ao qual seja possível fazer as *queries*. Esta consiste em dois passos essenciais: separação do texto em *tokens* e a sua normalização de forma a melhorar a pesquisa.

Tal é feito recorrendo a analisadores que desempenham 3 funções essenciais: filtragem de caracteres, separação em *tokens* e filtragem dos *tokens*.

Durante a filtragem de caracteres estes "limpam" a *string* antes da *tokenization*, sendo exemplo disso converter '&' em 'e' ou mesmo remover possíveis extratos de HTML. De seguida, esta *string* é separada em termos individuais, seguindo certos critérios, como quando encontra um *whitespace* ou pontuação. Posteriormente, o resultado é processado por filtros dos *tokens* que, por exem-

plo, convertem em letras minúsculas, removem alguns termos, como *stopwords*, ou até adicionam palavras sinónimas.

Pode-se desenvolver um analisador personalizado ou utilizar um dos já dispostos pela ferramenta, como é o nosso caso, em que aplicamos o *standard analyzer*, que separa o texto nos limites das palavras definido pelo *Unicode Consortium*, remove a maior parte da pontuação e coloca os termos em letra minúscula.

É de notar que quando inserimos um documento este é analisado, o que implica que quando pesquisamos por uma *query* é necessário passar esta pelo mesmo processo para garantir que procuramos da mesma forma que existe armazenada.

2.1.2 Relevância

A relevância centra-se na habilidade de classificar os documentos conforme o quão relevantes são para uma dada *query*. A relevância de cada é representada pela sua `_score`, sendo que quanto mais alta esta for, mais relevante é considerado o documento.

O algoritmo pelo qual este valor é calculado varia conforme a *query* utilizada, mas sendo que neste caso estamos a lidar com *full text queries*, a norma é usar o algoritmo *term frequency/inverse document frequency*, isto é, *TF/IDF*. Este foca-se em 3 aspetos principais: a frequência de um termo (*term frequency*), a frequência inversa nos documentos (*inverse document frequency*) e o comprimento do campo (*field-length norm*).

O primeiro tem em atenção quantas vezes um termo surge num determinado campo, significando que quantas mais vezes este aparecer, maior a relevância. O segundo estuda a frequência do termo no *index*, pois um termo que aparece em muitos documentos, tem um peso menor que um menos comum. O último, considera que quanto mais longo for o campo, menos provável é que uma dada palavra neste irá ser relevante.

Este vários fatores são combinados de forma a atribuir uma `_score` ao documento.

2.1.3 Full text queries

O Elasticsearch oferece uma *Query DSL* (*Domain Specific Language*) baseada em JSON para definir *queries*, sendo que, como referido anteriormente, irão ser abordadas apenas as *full text queries*. Estas são *queries* de alto nível que aplicam um analisador à *string* da *query* e, posteriormente, para cada termo obtido, executam as respetivas *queries* de baixo nível, combinando as *_score* calculadas a fim de dar uma classificação a cada um dos documentos e, por conseguinte, sabermos os mais relevantes.

Existem 7 diferentes tipos *full text queries*, sendo estas: `match`, `match_phrase`, `match_phrase_prefix`, `multi_match`, `common`, `query_string` e `simple_query_string`.

1. Query `match`

A *query* `match` permite que a pesquisa seja efetuada em apenas um campo e é feita para cada um dos termos da *query*, sendo que por *default* são aceites documentos em que apenas surja um dos termos da *string* pretendida. Um exemplo da sua utilização encontra-se abaixo:

Extrato 2.1 Exemplo de utilização de `match`

```
GET /_search
{
  "query": {
    "match": {
      "Titulo":"Negócios Ampliam"
    }
  }
}
```

Esta *query* aceita documentos que tenham no campo 'Titulo' os termos 'Negócios' e 'Ampliam' e a sua `_score` é determinada seguindo o algoritmo *TF/IDF* referido no ponto 2.1.2. Existem ainda vários parâmetros que podemos especificar de forma a alterar a *query* para que esta vá ao encontro do que o utilizador pretende, sendo estes alguns:

- `operator` - pode ser colocado a `and` ou `or` (*default*), e faz com que seja necessário que todos os termos surjam no campo requerido, no caso do `and` ou apenas que seja um, no caso do `or`.
- `minimum_should_match` - estabelece um número mínimo de termos da *query* que devem estar no campo em questão.
- `analyzer` - especifica o tipo analisador que irá fazer a análise do texto.
- `lenient` - permite ignorar exceções, como diferenças no tipo dos dados.
- `fuzziness` - permite *fuzzy matching*, isto é, pesquisa inexata. Por exemplo, faz *match* entre *strings* que só variem num determinado número de caracteres, sendo esse número definido pelo nível de `fuzziness`. Para este caso existem ainda outros parâmetros que ainda são possíveis especificar.
- `zero_terms_query` - define o comportamente caso o analisador remova todos os *tokens* da *query*, sendo que pode ser colocado a `none` (*default*) para não fazer `match` com nenhum documento ou a `all` que basicamente transforma a *query* no tipo `match_all`.
- `cutoff_frequency` - permite definir a frequência limite para que uma palavra seja considerada de alta ou baixa frequência. Isto permite que só seja feita a pesquisa de termos de alta frequência uma vez garantido o *match* de um ou de todos de baixa frequência, o que torna a pesquisa mais eficiente.

É ainda possível definir sinónimos para uma dada palavra, oferecendo assim uma maior flexibilidade à *query*.

Apesar de todas as funcionalidades que esta *query* oferece, ainda não permite pesquisa em vários campos, nem prefixos, carateres *wildcard* ou outras mais avançadas.

2. Query `match_phrase`

Muito similar à `match`, a *query* `match_phrase` em vez de efetuar procura para cada um dos termos, tem que garantir que surge exatamente a frase especificada.

Um exemplo de utilização é o seguinte:

Extrato 2.2 Exemplo de utilização de `match_phrase`

```
GET /_search
{
  "query": {
    "match_phrase": {
      "Titulo":"Negócios Ampliam"
    }
  }
}
```

Esta procura pela frase 'Negócios Ampliam' no campo 'Titulo', podendo dar resultados que contém outros termos, mas que necessariamente deverão conter esta *string* exata.

3. Query `match_phrase_prefix`

Esta *query* funciona do mesmo que modo que a `match_phrase`, no entanto, permite a pesquisa por prefixo do último termo da *query*. Podemos ainda especificar o parâmetro `max_expansions` que impõe um limite no número de sufixos que o último termo aceita. Por exemplo, se este for colocado a 50 (*default*) faz *match* com os primeiros 50 termos que surgem que têm como prefixo o último termo. Esta é uma das razões pelas quais esta *query* não é das mais apropriadas, pois o que procuramos pode não surgir nos primeiros 50 termos.

Extrato 2.3 Exemplo de utilização de `match_phrase_prefix`

```
GET /_search
{
  "query": {
    "match_phrase_prefix": {
      "Titulo": {
        "query" : "Negócios Amp",
        "max_expansions" : 50
      }
    }
```

```
    }
}
```

4. Query `multi_match`

Ao contrário das *queries* anteriores que apenas permitem que a pesquisa seja feita num certo campo, esta possibilita que seja efetuada em vários.

Para além disso, admite o uso de *wildcards*, isto é, o carater '?' e '*' podem ser colocados ao invés de um ou vários caracteres, respetivamente. Possibilita ainda que seja atribuída a certos campos uma maior importância que as outras, através no carater '^' .

Podemos ainda especificar qual o tipo do `multi_match`, isto é, como queremos que a `_score` seja calculada, podendo este ser do tipo `best_fields` (*default*) que usa como `_score` do documento a do campo com a mais alta; `most_fields` que combina a `_score` de cada um dos campos; e `cross_fields` que ignora a separação por campos e trata tudo como um único só. É ainda possível ser do tipo `phrase_prefix` ou `match_phrase_prefix` que segue a ideia do `best_fields`, mas em vez de ser uma pesquisa do tipo `match` é do tipo do especificado.

Um exemplo é o seguinte:

Extrato 2.4 Exemplo de utilização de `multi_match`

```
GET /_search
{
    "query": {
        "multi_match": {
            "query" : "Negócios Ampliam",
            "fields" : ["Titulo", "Antetitulo"]
        }
    }
}
```

Este faz uma pesquisa do tipo `match` nos campos 'Titulo' e 'Antetitulo' pelos termos 'Negócios' e 'Ampliam', e dado que não foi especificado o tipo é usado o `best_fields`.

5. Query `common`

Este é um tipo de *query* que dá preferência a palavras menos comuns. A `common` analisa a frequência de cada um dos termos e etiqueta-os conforme se é de alta ou baixa frequência. Esta *query* baseia-se na ideia que ao pesquisar numa primeira instância por termos menos comuns, reduzimos consideravelmente o número de documentos com que estamos a lidar, e por consequência o número de *queries* que são precisas de ser efetuadas, ao invés de pesquisar por mais comuns, o que traduzir-se-ia num grande volume de documentos e, consequentemente, mais *queries*.

Esta começa por fazer a pesquisa por termos que são menos comuns e portanto mais importantes, pois aqueles que aparecem em menos documentos têm um maior impacto na relevância. De seguida, efetua o mesmo para os mais comuns, mas em vez de ser para todos os documentos é apenas para aqueles que fizeram `match` para os menos comuns.

Tal como no `match`, podemos decidir a partir de que frequência é que uma palavra é considerada de baixa ou de alta frequência, sendo tal conseguido através do `cutoff_frequency`. Para além deste, existem outros parâmetros que é possível explicitar, mas este é o mais relevante.

Extrato 2.5 Exemplo de utilização de `common`

```
GET /_search
{
    "query": {
        "common" : {
            "Titulo" : {
                "query": "Negócios Ampliam",
                "cutoff_frequency": 0.001
            }
        }
    }
}
```

Nesta exemplo, palavras que surgem mais que 0.1% no documento irão ser tratadas como comuns.

6. Query `query_string`

A `query_string` permite especificar condições do tipo `AND`, `OR` e `NOT`.

Tal como as outras *queries* funciona para vários campos, permite sinónimos, *wildcards*, expressões regulares, *fuzziness*, pesquisa por proximidade e intervalos, *boosting* de campos e operadores booleanos.

Extrato 2.6 Exemplo de utilização de `query_string`

```
GET /_search
{
    "query": {
        "query_string" : {
            "query": "Neg?cios AND Amp*",
            "fields": ["Titulo", "Antetitulo"]
        }
    }
}
```

Neste exemplo, é efetuada uma pesquisa nos campos 'Titulo' e 'Antetitulo' pelos termos que respeitem o termo 'Neg?cios' e 'Amp*', sendo que os símbolos

'?' e '*' representam *wildcards*. Se em vez de 'AND' fosse 'OR' seria pesquisado quer por um termo, quer por outro, e se fosse 'NOT' teriamos que garantir que o termo 'Amp*' não existe.

7. Query `simple_query_string`

Esta é uma versão simplificado da *query_string* que faz uso do *SimpleQueryParser*. No entanto, ao contrário da anterior não manda exceções e ignora partes inválidas de uma *query*. A tabela abaixo representa os símbolos aceites e os seus significados.

Símbolo	**Significado**
+	operação AND
—	operação OR
-	negação de um único *token*
”	pesquisa exata
*	pesquisa por prefixo
()	precedência
∼N	*fuzziness* ou *slop*

Extrato 2.7 Exemplo de utilização de `simple_query_string`

```
GET /_search
{
    "query": {
      "simple_query_string" : {
            "query": "Negócios -Ampliam",
            "fields": ["Titulo", "Antetitulo"]
        }
    }
}
```

Esta *query* pesquisa por documentos cujos campos 'Titulo' e 'Antetitulo' contenham o termo 'Negócios', mas não o termo 'Ampliam'.

Capítulo 3

Exemplo de Utilização

Com o intuito de demonstrar uma possível utilização da ferramenta *Elasticsearch*, decidiu-se implementar um filtro UNIX com as capacidades de procura oferecidas pelo *Full Text Queries*, que funcionasse tanto num índice invertido local como num instalado remotamente num *cluster*.

Tal filtro, desenvolvido em Python, será abordado na Secção 3.2, sendo importante discutir previamente como foi usada a API do *Full Text Queries* em Python (Secção 3.1).

3.1 API do Full Text Queries em Python

De forma a usar as *Full Text Queries* para a implementação do filtro UNIX desejado, decidimos criar um módulo onde implementamos cada uma das *queries* da API, com recurso ao cliente *low-level* oficial do Elasticsearch para Python.

O objetivo da sua criação é simplificar a escrita das *queries*, mantendo em cada função uma terminologia semelhante à da DSL em JSON do Elasticsearch, e escondendo alguns detalhes desnecessários para o nosso caso em particular.

As funções implementadas são:

- **load_documents(idx, dtype, files)**

  Esta é a função que permite indexar documentos (guardar e fazer com que sejam procuráveis). Nos argumentos recebe os ficheiros a serem indexados (*files*), o índice onde devem ser guardados (*idx*) e o tipo de documento de que se trata (*dtype*). Os ficheiros devem estar numa lista e a função garante ainda que o índice é criado caso não exista.

- **match(content, field, exact, idx, dtype)**

  Corresponde às duas *queries* mais básicas de pesquisa oferecidas pela ferramenta. Se o valor do booleano *exact* for False, a função usa a *query* match da DSL, que inclui *fuzzy matching* e *queries* de proximidade. No caso de o valor de *exact* for True, a pesquisa procura por frases exatamente iguais à pretendida.

- **match_as_you_type(prefix, curr_word, field, idx, dtype)**

  Função que será usada para a funcionalidade do interpretador de *search-as-you-type*. Recebe um prefixo exacto e identifica a lista de possíveis frases que o completam.

- **multi_match(content, fields, idx, dtype)**

  Faz o mesmo que a função match, mas possibilita dar mais do que um campo do documento para procura.

- **common_terms(content, field, idx, dtype, cutoff_frequency)**

  Faz o mesmo que a função match, mas dá mais relevância a termos menos comuns. Os termos que são considerados menos comuns são assim determinados pelo parâmetro cutoff_frequency. Caso um termo apareça menos vezes do que esta frequência, é considerado incomum.

- **query_string(content, fields, idx, dtype)**

  Query mais elaborada, cujo termo de pesquisa usa a sintaxe do Apache Lucene. Permite pesquisa em vários campos e é usada para *queries* mais elaboradas que permitem condições.

- **simple_query_string(content, fields, idx, dtype)**

  Uma versão mais simples e mais robusta da query_string, que usa uma síntaxe diferente no termo de pesquisa.

3.2 Filtro UNIX

O filtro Unix desenvolvido tem como principal objetivo, utilizando as facilidades disponibilizadas pela API referida do *ElasticSearch*, organizar conjuntos de documentos potencialmente muito numerosos, quer localmente ou remotamente. De forma a garantir rápidas pesquisas, os documentos são indexados, e distinguidos pelo tipo de documento.

3.2.1 Flags

As *flags* utilizadas permitem distinguir a ação a ser executada pelo programa, seja inserção de documentos ou pesquisa, mas mais que isso, dar a conhecer qual o tipo de *query* da *API* que o programa deve utilizar nas pesquisas. As mesmas podem ser divididos em vários grupos, de opções genéricas do programa, de indexação de documentos, de opções de pesquisa, de seleção do tipo de *query* a utilizar e de controlo de saída.

Opções genéricas do Programa	
–help	Utilizada para aceder ao manual de utilizacão do programa.
-i INDEX	Permite definir o índice (INDEX) a ser utilizada nas pesquisas, ou na indexação de documentos. Sendo que no primeiro caso é de carácter obrigatório.
-d DOC_TYPE	Permite definir o tipo dos documentos que serão passados ao programa para indexar ou o tipo de documento alvo das pesquisas.
Indexação de Documentos	
-b FILES	Permite declarar que se pretende realizar uma operação de indexação de documentos.
Opções de Pesquisa	
-f FIELD(S)	Permite identificar o campo ou campos a serem utilizados. (Obrigatório nas pesquisas)
-p PATTERN	Permite identificar o padrão a ser utilizado nas pesquisas. Caso esta opção seja utilizada apenas uma pesquisa será realizada. Caso contrário é iniciado o modo interpretador, onde sucessivos padrões podem ser fornecidos.
Seleção do Tipo de Query	
-c CUTOFF	Permite utilizar a query "common_terms"da API do Elasticsearch, onde é dada maior preferência a palavras menos comuns. Para tal, é necessário passar um valor CUTOFF que é a frequência que uma determinada palavra tem de possuir nos documentos para não ser considerada comum.
-e	Permite utilizar a query "match_phrase"da API do Elasticsearch, utilizada para pesquisar frases exatas ou palavras semelhantes.
-S	Permite utilizar a query "query_string"da API do Elasticsearch, que torna possível especificar, por exemplo, operadores condicionais tais como AND—OR—NOT. Pesquisas em múltipos campos também são suportadas.

Table 3.1 continued from previous page

-s	Permite utilizar a query "simple_query_string"da API do Elasticsearch que premite uma syntax mais robusta. Operadores tais como ['—', '+', '-'] podem ser utilizados. Partes inválidas da query são descartadas.
<Sem tipo>	Quando se pretende fazer uma pesquisa, a ausência da flag que define o tipo da query , significa que deverá ser utilizada a query "match"default da API, caso apenas um campo for definido para a procura, ou a query "multi_match", para mais do que 1 campo.
Controlo de Saída	
-n N_ITEMS	Permite estipular o limite máximo de documentos (N_ITEMS) a enviar para a saída.

3.2.2 Interpretador e *Search-as-you-type*

Como mencionado na secção 3.2.1, a não utilização da*flag* `'-p'`, que permite definir um padrão para a procura, desencadeia a iniciação do modo interpretador. Uma vez no interpretador e tendo em conta o tipo de *query* passada ao filtro, é possível introduzir sucessivamente vários padrões e obter várias respostas diferentes para o mesmo tipo de *query*.

Para tal, o interpretador baseia a sua atuação na execução de um único método, `input_loop`, que permanece num ciclo constante até que a string "*stop*"seja introduzida. Para cada *input* do utilizador é invocado o método `execute_query` que trata de passar os argumentos necessários à *query* da API devida.

```
def input_loop(query_type):
    line = ''
    while line != '\\stop':
        line = input('Prompt ("\\stop" to quit): ')
        line, f_d = spln_elastic.redirect_output(line)
        if line != '\\stop':
            execute_query(line, query_type, f_d)
```

A este interpretador foram ainda incluídas algumas funcionalidades extra que são bastante úteis para os utilizadores. São estas, a manutenção de um histórico dos pedidos anteriores, permitindo desta forma modificar e aperfeiçoar os padrões, bem como foi introduzida uma funcionalidade que permite completar padrões inacabados com sugestões específicas para o prefixo inserido e para o campo do documento a que a procura se destina. Naturalmente, estes padrões foram idealizados para campos que não possuam muitos caracteres, por exemplo

títulos de notícias, etc. É de referir que não é possível padrões quando é inserido mais do que um campo para a pesquisa.

Estas funcionalidades foram implementadas utilizando as facilidades e métodos que a interface *GNU readline* disponibiliza. O histórico é implementado com *logging* e as sugestões para completar as palavras com recurso a uma classe `Completer` que faz nada mais nada menos que utilizar, mais uma vez, a API do Elasticsearch, invocando a *query* `match_phrase_prefix` concebida para o efeito.

Capítulo 4

Conclusão

Neste trabalho prático, propusemos-nos a analisar com detalhe um dos maiores *use cases* da ferramenta Elasticsearch, as *Full Text Queries*. Foi analisada a API disponível e as capacidades relevantes à PLN oferecidas por esta. Para além disso, desenvolveu-se um filtro UNIX em Python com capacidades de procura num índice invertido que usa esta mesma ferramenta, tendo-se neste documento abordado as principais decisões de implementação tomadas.

Com o trabalho desenvolvido, consideramos ter atingido os objetivos propostos. Como trabalho futuro, o filtro poderá ser melhorado de forma a permitir que o *output* seja dado num tipo específico de ficheiro (XML, JSON, etc), para além do já implementado *output* para o *stdout*.