Moteurs de recherche full-text NoSQL (1)

François Role

Université Paris Descartes

Objectifs du cours

- Faire découvrir les concepts et les techniques de base de la recherche full-text
- Montrer comment les "moteurs de recherche orientés documents" s'inscrivent dans le mouvement NoSQL

Plan

Le mouvement NoSQL

Le terme NoSQL apparaît en 2009. Au delà de l'effet de mode, il correspond essentiellement au changements d'échelle induits par le Web. Il correspond aussi à l'évolution des techniques de génie logiciel

- **Evolution des données à traiter**
  - Plus volumineuses ("web scale", "big data") donc sur lesquelles on est tenté de faire des traitements analytiques massifs
  - Moins structurées
  - Plus évolutives (sites Web, réseaux sociaux, tweets, etc.)
- **Solutions**
  - Répartir les données sur des clusters de machines
  - Utiliser des schémas moins rigides ou mêm pas de schéma ("schemaless")
  - Versionner systématiquement les données

Bases relationnelles et bases NoSQL - distribution, réplication

- **Bases relationnelles**
  - Non nativement distribuées
  - Techniques transactionnelles élaborées
  - Non nativement répliquées
- **Bases NoSQL**
  - Nativement distribuées
    - maître-esclave (MongoDB), sans maître (Dynamo) avec gossip et consistent hash
    - sharding = partitionnement avec distibution automatique
    - améliore les performance
  - Nativement répliquées
    - limite les risques de pertes de données et améliore les performances

Bases relationnelles et bases NoSQL - modélisation

- **Bases relationnelles**
  - La structure des entités du domaine et leurs relations sont d'abord modélisés à l'avance dans un schéma. Les données sont insérées ensuite
  - Les relations entre entités se représentent par des "clés primaires" et des "clés étrangères"
  - Des règles de "normalisation" doivent être respectées, par exemple : chaque attribut des entités contient une valeur atomique
- **Bases NoSQL**
  - La structure des données n'est pas forcément définie à l'avance. Un schéma peut être inféré dynamiquement
  - des structures de données peuvent être imbriquées pour représenter des relations
  - D'une manière générale, il n'y pas de règles de "normalisation" précises. Par exemple, les attributs peuvent avoir pour valeur des listes ou même des objets structurés plus complexes

Bases relationnelles et bases NoSQL - versioning

- **Bases relationnelles**
  - Non nativement versionnées
- **Bases NoSQL**
  - Nativement versionnées
    - Facilite l'évolution des schémas de données

Bases relationnelles et bases NoSQL - interactions client

- **Bases relationnelles**
  - Pas conçues nativement pour dialoguer de façon fluide et intensive avec des applications clientes
  - Utilisation de langages dédiés (donc à apprendre), d'ORM (parfois assez lourds)
  - Conformes à l'approche Waterfall
- **Bases NoSQL**
  - Conçues nativement pour dialoguer de façon naturelle et simple avec des applications Web
  - API REST
  - Clients écrits dans les langages déjà connus des développeurs Web (Javascript) et très simples à utiliser

Bases relationnelles et bases NoSQL - conclusion

- **Argument des partisans du NoSQL**
  - Flexibilité, souplesse, pragmatisme contre rigidité
  - Bien adapté aux applications Web
- **Arguments des partisans du relationnel**
  - Importance de la conception, distinction physique logique
  - Parc installé, toutes les applications ne sont pas "web scale"

Plan

Valeurs exactes et valeurs similaires

L'apparition des Bases NoSQL orientées "full-text" traduit bien l'évolution de la nature des données à traiter
Les bases de données traditionnelles traitent majoritairement des champs avec valeur exacte, comme par exemple :

- *dateCreation* : '2014-08-20'
- Une requête portant sur ce champ, par exemple "dateCreation= '2014-08-20'" va retrouver les enregistrements où ce champ a **exactement** pour valeur '2014-08-20'

Mais il existe d'autres types de champs, comme par exemple :

- *tweet* : 'Barack Obama devrait se rendre jeudi en Ukraine'
- Une requête portant sur ce champ, par exemple "tweet='Obama Ukraine'", va retrouver les documents (ici des tweets) qui sont **similaires** à la requête

La recherche sur des champs full-text s'appuie sur des **index inversés**

Index inversé

Index inversé = liste des mots distincts apparaissant dans la collection de documents avec pour chaque mot la liste des documents où ils apparaissent

Soient les deux documents (où chaque lettre majuscule représente un "token") : Doc_1 "$A\ B\ E\ D$" Doc_2 "$B\ F\ E\ C\ A$"

	Doc_1	Doc_2
A	x	x
B	x	x
C		x
D	x	
E	x	x
F		x

Si la requête est "$B\ C$", on voudrait trouver des documents **similaires** à "$B\ C$"

Recherche booléenne

- Constituer une base full-text à partir des deux documents précédents consiste à découper le contenu de ces documents en tokens pour créer un indice inversé
- La base peut alors être interrogée en utilisant cet index
  - Requête "B C" (documents contenant "B" et/ou "C") :

$$Docs_B = \{Doc_1, Doc_2\} \quad Docs_C = \{Doc_2\}$$

$$Res = Docs_B \cup Docs_C = \{Doc_1, Doc_2\}$$

  - Requête "B and C" (documents contenant "B" et "C") :

$$Res = Docs_B \cap Docs_C = \{Doc_2\}$$

Similarité entre documents

- La technique booléenne ne permet pas d'ordonner (*ranking*) les résultats. Par exemple, pour la requête "B C" (documents contenant "B" et/ou "C") elle a renvoyé l'ensemble (par définition non ordonné) $\{Doc_1, Doc_2\}$
- Doc_2 semble plus similaire à la requête et on aurait préféré une liste ordonnée, comme dans les moteurs de recherche :
  1. Doc_2
  2. Doc_1

Représentation vectorielle des documents

- Pour trouver des documents similaires à une requête donnée, on représente les documents et la requête comme des vecteurs et on mesure la similarité entre ces vecteurs
- Trois documents (chaque document se réduit à une ligne)

```
aaa bbb ccc aaa eee aaa ggg.
ggg fff eee.
ddd aaa ggg aaa eee eee.
```

- Trois vecteurs formant une "matrice document-terme" :

$$\begin{bmatrix} 3 & 1 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 2 & 0 & 0 & 1 & 2 & 0 & 1 \end{bmatrix}$$

- chaque vecteur (ligne) a sept composantes puisqu'il y a sept termes $w_1, ..., w_7$ distincts dans la collection

Détermination naïve du poids des termes

- Dans la matrice :

$$\begin{bmatrix} 3 & 1 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 2 & 0 & 0 & 1 & 2 & 0 & 1 \end{bmatrix}$$

  chaque vecteur (ligne) a sept composantes puisqu'il y a sept termes $w_1, ..., w_7$ distincts dans la collection
- chaque composante correspond à un terme (vu comme un vecteur d'une base). Pour un document $\mathbf{d}$, la composante c_i a pour valeur le nombre de fois ou le terme w_i apparaît dans $\mathbf{d}$.
- C'est un choix simple mais qui a de gros défauts. Dans un texte économique des mots courants comme "grand", "petit", etc. risquent d'avoir plus de poids que des mots comme "industrie", "agricole", etc.

Pondération TF-IDF

- Une meilleure façon de pondérer : un terme a de l'importance dans un document s'il apparaît fréquemment dans ce document mais peu ailleurs
- On mesure si le terme i est fréquent dans la collection (***inverse document frequency***) :

$$idf_i = \ln \frac{n}{df_i}$$

  idf_i prend une valeur (positive) d'autant plus importante qu'un terme i est rare dans la collection.
- On mesure si un terme i est fréquent dans le document j :

$$tf_{i,j}$$

- On multiplie pour estimer le poids de i dans j : $w_{i,j} = tf_{i,j} * idf_i$

EXERCICE 1

En utilisant la pondération TF-IDF calculer les trois vecteurs correspondant à chacun des documents précédents

Constitution de la liste des termes

- Les étapes standard pour y parvenir :

```
                    "Laa bb& cc aaa ff& aaa hhh"

--->  (filtrage - niveau caractère)
                  "Laa bbe cc aaa ffe aaa hhh"
--->  (tokenization)
                  "Laa" "bbe" "cc" "aaa" "ffe" "aaa" "hhh"
--->  (normalisation)
                  "laa" "bbe" "cc" "aaa" "ffe" "aaa" "hhh"
 --->  (filtrage - niveau token )

                  "laa" "bbe" "aaa" "ffe" "aaa" "hhh"
```

Unicode : code point et encodage

```
U+0030 0 30 DIGIT ZERO = 48
...
U+0041 A 41 LATIN CAPITAL LETTER A = 65
...
U+0061 a 61 LATIN SMALL LETTER A = 97
U+0062 b 62 LATIN SMALL LETTER B
U+0063 c 63 LATIN SMALL LETTER C
...
U+00E9 é c3 a9 LATIN SMALL LETTER E WITH ACUTE = 233
```

Encodage/décodage : exemple 1

```
> u'é'
  u'\xe9'   # code point = 233

u'é'.encode('utf-8')
   '\xc3\xa9'   # utf-8 encoding

> print 'universit\xc3\xa9'  # décodage implicite
  université
> print u'universit\xe9'
  université
```

Encodage/décodage : exemple 2

```
u'\u0642'.encode('utf-8')
```

```
'\xd9\x82'
```

```
print u'\u0642'
```

ق

```
print '\xd9\x82'
```

ق

EXERCICE 2

Vous avez trouvé un fichier contenant la suite d'octets suivante :
D984D985D986D987
Retrouvez la suite de lettres correspondante. Vous pouvez utiiser Google.

Normalisation

- Mise en minuscules
- Stemming
  - fonder –> fond
  - fondation –> fond
- Lemmatisation
  - universités –> université
  - chevaux –> cheval
  - ira –> aller

Filtrage

- Liste de stopwords
- Calcul de fréquences

EXERCICE 3

Calculez la similarité cosinus entre la requête "aaa ggg" et

EXERCICE 1 : CORRECTION

```
idf_i values
[ 0.40546511  1.09861229  1.09861229  1.09861229  0.
  0.        ]
w_ij values
[[ 1.21639532  1.09861229  1.09861229  0.          0.
  [ 0.          0.          0.          0.          0.
    0.        ]
  [ 0.81093022  0.          0.          1.09861229  0.
normalized w_ij
[[ 0.61645842  0.55676702  0.55676702  0.          0.
  [ 0.          0.          0.          0.          0.
  [ 0.59387587  0.          0.          0.80455668  0.
```

EXERCICE 2 : CORRECTION

U+0644	ل	d9 84
U+0645	م	d9 85
U+0646	ن	d9 86
U+0647	ه	d9 87

EXERCICE 3 : CORRECTION

```
query= [ 1. * 0.405, 0., 0., 0., 0., 0., 1. * 0.]

[ 0.24966566    0.          0.24051973]
```