UNIVERSITÉ D'AVIGNON ET DES PAYS DE VAUCLUSE

M2 ILSEN – 2019/20
UE Ingénierie du document et de l'information
UCE Indexation & recherche

Vincent Labatut

TP 3 | Résolution de requêtes

Ce TP est basé sur les classes développées lors des deux premiers TP. Nous avons maintenant un outil d'indexation basique mais fonctionnel. Le but de ce TP est d'implémenter des classes permettant de l'interroger au moyen de requêtes.

L'écriture d'un analyseur syntaxique permettant de décomposer les requêtes en fonction des opérateurs utilisés est un travail qui pourrait en soi faire l'objet d'une série de TP. Cependant, ce n'est pas l'objectif de cette UE, aussi allons-nous simplifier cette partie au maximum, afin de nous concentrer sur les aspects directement liés à l'indexation.

1 Opérateur ET

Du point de vue de la syntaxe des requêtes, nous allons considérer que tout séparateur (i.e. tout caractère non-alphanumérique) est interprété comme un opérateur ET implicite. Ceci vaut en particulier pour le caractère espace `' '`.

Exemple : la requête `bateau course monde` sera interprétée comme la recherche des documents contenant à la fois les mots `bateau ET course ET monde`.

Exercice 1

Dans le package `query`, ouvrez la classe `AndQueryEngine`, qui est destinée à traiter des requêtes ne contenant que des opérateurs ET. Repérez le champ `index`, qui représente l'index utilisé par cette classe pour résoudre les requêtes.

Complétez la méthode `void splitQuery(String query, List<List<Posting>> result)` qui reçoit une requête `query` sous forme de chaîne de caractères et une liste vide `result`. Cette requête est supposée ne contenir que des opérateurs ET (i.e. pour nous : n'importe quel séparateur). Comme son nom l'indique, la liste est destinée à être complétée par la méthode.

La méthode doit tokéniser et normaliser la chaîne de caractères reçue, *en utilisant les objets utilisés lors de la construction de l'index* (et qui ont été enregistrés avec lui) de manière à obtenir une liste de termes. Elle doit ensuite récupérer la liste de postings associée à chaque terme. Le résultat final qui est renvoyé correspond à la liste de listes de postings.

Exemple : pour la requête `recherche ET INFORMATION ET Web`, on obtient les 3 listes suivantes (pour `corpus`) :

```
[ [5, 6, 9, 11, 12, 14, 21, 22, 27, 29, 30, 31, 32, 34, 37, 38, 41, 43, 48, 49...] ,
  [1, 9, 11, 13, 14, 17, 25, 34, 38, 41, 42, 44, 46, 48, 51, 53, 54, 55, 57, 58...] ,
  [7, 8, 12, 15, 16, 19, 22, 26, 30, 34, 35, 37, 38, 41, 46, 48, 50, 58, 70...] ]
```

> **Remarque :** n'oubliez pas que `Normalizer.normalizeType` est susceptible de renvoyer `null` si le type normalisé n'est pas considéré comme un terme. Vous devez donc traiter ce cas dans `splitQuery`. Notez qu'il est aussi possible que le terme recherché n'apparaisse pas du tout dans l'index. Dans ces deux cas, la liste de postings associée au terme doit être une liste vide construite pour l'occasion.

Exercice 2

Dans `AndQueryEngine`, complétez la méthode `List<Posting> processConjunction(List<Posting> list1, List<Posting> list2)` qui reçoit deux listes de postings en paramètres, et calcule leur intersection, puis la renvoie comme résultat.

Attention : les listes sont supposées triées, aussi vous devez *obligatoirement* utiliser l'algorithme décrit en cours. Et surtout pas des méthodes Java toutes faites comme `retainAll`, qui n'exploitent pas cette propriété.

Exemple : pour les listes (1,2,3,4,5,6,7,17,18) et (3,4,10,16,17,19), on obtient le résultat (3, 4, 17).

Exercice 3

Dans `AndQueryEngine`, repérez le champ constant statique appelé `COMPARATOR` et de classe `Comparator<List<Posting>>`. Cet objet est un comparateur, i.e. il est conçu uniquement pour comparer des objets et décider lequel est plus grand que l'autre, selon un ordre pré-établi.

Ici, on veut utiliser un comparateur pour comparer deux listes de postings, en utilisant exclusivement leurs longueurs respectives. Complétez la méthode `int compare (List<Posting> l1, List<Posting> l2)` de `COMPARATOR` de manière à implémenter ce comportement.

Remarque : l'intérêt du champ constant statique `COMPARATOR` est qu'on n'instancie le comparateur qu'une seule fois, et qu'on peut ensuite le réutiliser à chaque fois qu'on en a besoin.

Exercice 4

Dans `AndQueryEngine`, complétez la méthode `List<Posting> processConjunctions (List<List<Posting>> postings)` qui reçoit une liste de listes de postings en paramètre, et calcule leur intersection, puis la renvoie comme résultat. Là encore, les listes sont supposées triées. Il est nécessaire d'utiliser la méthode `processConjunction`. Vous devez appliquer l'algorithme décrit en cours pour effectuer le calcul de façon optimale, en déterminant l'ordre dans lequel les listes doivent être combinées en fonction de leur longueur.

Exemple : pour les listes (1,2,3,4,5,6,7,17,18), (3,4,10,16,17,19), et (1,4,19,26), on prend d'abord l'intersection des deux dernières listes, puis du résultat obtenu et de la première liste, et on obtient au final (4).

Remarque : pour trier votre liste de listes, vous devez utiliser `Collections.sort(List<T> collection, Comparator<T> comparator)`. Notez que cette fois, la méthode a besoin d'un comparateur : vous devez bien sûr utiliser le champ `COMPARATOR` défini précédemment.

Exercice 5

Dans `AndQueryEngine`, complétez la méthode `List<Posting> processQuery (String query)`, qui prend en paramètre une requête et renvoie la liste des documents (ou plutôt de leur docID) qui la satisfont. Elle doit être capable de traiter une requête contenant uniquement des opérateurs ET.

La méthode doit effectuer les tâches suivantes :

1. Décomposer la requête, grâce à `splitQuery` ;
2. Réaliser le traitement lié à l'opérateur ET, grâce à `processConjunctions` ;
3. Renvoyer la liste de postings finale.

Pensez également à traiter le cas où la requête ne contient qu'un seul terme.

La méthode doit aussi mesurer et afficher le temps total nécessaire au traitement effectué, ainsi que le nombre de postings renvoyés. On veut un affichage de la forme suivante :

```
Processing query "xxxx xxxx xxxx"
Query processed, returned xxxx postings, duration=xxxx ms
```

Exercice 6

Dans `Test1`, complétez `testQuery` de manière à enchaîner le traitement de plusieurs requêtes. La méthode doit charger l'index à partir d'un fichier, puis traiter chaque requête en affichant les résultats correspondants sous la forme de listes de postings.

Exemple : affichages obtenus pour 3 variantes d'une même requête

```
Loading the index
Index loaded, duration=29413 ms
```

Requête `project` :

```
Processing query "recherche"
Query processed, returned 1197 postings, duration=1 ms
[5, 6, 9, 11, 12, 14, 21, 22, 27, 29, 30, 31, 32, 34, 37, 38, 41, 43, 48, 49...]
Files:
[007ae9bc-3464-4447-8967-5a78ddf854ae.txt...]
```

Requête `project SOFTWARE` :

```
Processing query "recherche INFORMATION"
Query processed, returned 661 postings, duration=2 ms
[9, 11, 14, 34, 38, 41, 48, 51, 55, 58, 61, 62, 66, 68, 69, 72, 76, 84, 87...]
Files:
[00fb55b5-7bfa-4346-ae92-8bd356a7799b.txt...]
```

Requête `project SOFTWARE Web` :

```
Processing query "recherche INFORMATION Web"
Query processed, returned 289 postings, duration=1 ms
[34, 38, 41, 48, 58, 72, 84, 87, 99, 103, 106, 118, 121, 150, 156, 165, 170...]
Files:
[03775cf1-869f-4242-abbe-c7290f7a5a08.txt...]
```

Remarque : pour pouvoir effectuer des vérifications, affichez aussi la liste des noms de fichiers correspondant aux docIDs listés, en utilisant la méthode `getFileNamesFromPostings`.

Exercice 7

On veut comparer les 3 structures de données utilisées pour stocker le lexique (tableau, table de hachage, arbre). Pour chacun des 3 index correspondant, chargez-le et utilisez-le pour résoudre 1 000 fois la liste de requêtes suivantes, en mesurant le temps total nécessaire.

- `recherche d'information`
- `microsoft windows`
- `berners-lee`
- `internet mondial`
- `moteur de recherche`
- `modèle de travail`
- `contrôle des connaissances`
- `système de notation`
- `points forts`
- `réseaux sociaux`

Comparez les 3 performances, et discutez-les.

2 Opérateur OU

Remarque : le traitement de l'opérateur OU (donc toute cette section) est **optionnel**, et le code source correspondant ne sera pas requis lors de l'examen de TP.

Nous allons maintenant ajouter la possibilité d'utiliser l'opérateur OU, que l'on représentera dans nos requêtes par une virgule. On peut donc avoir une requête de la forme : `bateau monde,solitaire,course`, qui sera interprétée comme `bateau ET monde OU solitaire OU course`.

L'opérateur ET étant prioritaire sur l'opérateur OU, on réalisera une évaluation équivalente à `(bateau ET monde) OU solitaire OU course`. Cependant, pour ne pas complexifier inutilement le traitement, on se limitera à des requêtes *sans parenthèses*.

Les trois différentes formes que peut prendre une de nos requêtes sont donc les suivantes :

1. `terme` : forme la plus simple, ne contenant qu'un seul terme ;
2. `terme_1 ET ... ET terme_k` : conjonction, que l'on sait déjà évaluer ;
3. `expression_1 OU ... OU expression_k` : disjonction d'`expressions`, chacune pouvant être ici :
   - Soit un simple terme : pas d'évaluation particulière, ex. : `solitaire`.
   - Soit une conjonction : on sait déjà évaluer, ex. : `bateau ET monde`.

Exercice 8

Nous allons définir une classe différente pour traiter simultanément ET et OU. Faites une copie de `AndQueryEngine`, que vous nommerez `AndOrQueryEngine`. Renommez la méthode `splitQuery` en `splitAndQuery`, puisqu'elle se concentre sur l'opérateur ET.

Exercice 9

Dans `AndOrQueryEngine`, écrivez une méthode `void splitOrQuery(String query, List<List<List<Posting>>> result)`, qui traite une requête de la troisième forme présentée ci-dessus. La méthode renvoie une liste de listes de listes de postings. Décomposons cette structure un peu compliquée :

- `List<Posting>` : ce premier niveau correspond à une liste de postings associés à un terme ;
- `List<List<Posting>>` : ce deuxième niveau correspond aux opérandes d'opérateurs ET, i.e. chaque liste de postings représente un terme et devra être combinée aux autres (plus tard) par intersection.
- `List<List<List<Posting>>>` : ce troisième niveau correspond aux opérandes d'opérateurs OU, qui sont des conjonctions de termes représentées par des listes à combiner (plus tard) par union.

D'abord, la méthode reçoit la requête et la partage en plusieurs sous-requêtes en y cherchant l'opérateur OU (représenté par une virgule). Vous devez utiliser la méthode `String.split` pour cela. Puis, la méthode applique `splitAndQuery` à chaque sous-requête afin d'obtenir autant de listes de listes de postings (i.e. `List<List<Posting>>`). Enfin, elle place ces listes dans une autre liste de type `List<List<List<Integer>>>`, qui est le résultat final.

Exemple : pour la requête `recherche ET INFORMATION ET Web OR document ET ordinateur`, on obtient deux sous requêtes : d'une part `recherche ET INFORMATION ET Web` et d'autre part `document ET ordinateur`. Après traitement de ces deux sous-requêtes par `splitAndQuery`, le résultat final est :

```
[ [ [5, 6, 9, 11, 12, 14, 21, 22, 27, 29, 30, 31, 32, 34, 37, 38, 41, 43, 48, 49...] ,
    [1, 9, 11, 13, 14, 17, 25, 34, 38, 41, 42, 44, 46, 48, 51, 53, 54, 55, 57...] ,
    [7, 8, 12, 15, 16, 19, 22, 26, 30, 34, 35, 37, 38, 41, 46, 48, 50, 58, 70...]
  ] ,
  [ [7, 8, 12, 19, 22, 30, 37, 38, 41, 46, 48, 51, 91, 92, 96, 99, 104, 105, 108...] ,
    [13, 16, 22, 25, 37, 38, 48, 49, 50, 59, 61, 70, 71, 78, 87, 98, 101, 105...]
  ]
]
```

Exercice 10

Dans `AndOrQueryEngine`, écrivez une méthode privée `List<Posting> processDisjunction(List <Posting> list1, List<Posting> list2)` qui reçoit deux listes de postings en paramètres, et calcule leur union, puis la renvoie comme résultat. Comme pour l'intersection, les listes sont supposées triées, et vous devez utiliser l'algorithme décrit en cours.

Exemple : pour les listes (1,2,3,4,5,6,7,17,18) et (3,4,10,16,17,19), on obtient le résultat (1, 2, 3, 4, 5, 6, 7, 10, 16, 17, 18, 19).

Exercice 11

Dans `AndOrQueryEngine`, écrivez une méthode privée `List<Posting> processDisjunctions(List<List<Posting>> postings)` qui reçoit une liste de listes de postings en paramètre, et calcule leur union, puis la renvoie comme résultat. Là encore, les listes sont supposées triées. Il est nécessaire d'utiliser la méthode `processDisjunction`. Vous devez appliquer l'algorithme décrit en cours pour effectuer le calcul de façon optimale en déterminant les priorités d'évaluation en fonction des longueurs des listes.

Exemple : pour les listes (1,2,3,4,5,6,7,17,18), (3,4,10,16,17,19), et (1,4,19,26), on prend d'abord l'union des deux dernières listes, puis du résultat et de la première liste, et on obtient au final (1, 2, 3, 4, 5, 6, 7, 10, 16, 17, 18, 19, 26).

Exercice 12

Dans `AndOrQueryEngine`, modifiez la méthode `processQuery` de manière à pouvoir traiter des requêtes contenant les opérateurs ET (représenté par un espace) et OU (représenté par une virgule). La méthode doit effectuer les tâches suivantes :

1. Décomposer la requête, grâce à `splitOrQuery` ;
2. Réaliser le traitement lié à l'opérateur ET, grâce à `processConjunctions` ;
3. Réaliser le traitement lié à l'opérateur OU, grâce à `processDisjunctions` ;
4. Renvoyer la liste de postings finale.

Exercice 13

Dans la classe `Test`, complétez la méthode `testQuery()` et testez notre nouveau moteur.

Exemple : affichages obtenus pour 3 variantes d'une même requête

```
Loading the index
Index loaded, duration=30286 ms
```

Requête `project,SOFTWARE,Web,pattern,computer` :

```
Processing query "recherche,INFORMATION,Web,document,ordinateur"
Query processed, duration=2 ms
Result: 2002 document(s)
[1, 5, 6, 7, 8, 9, 11, 12, 13, 14, 15, 16, 17, 19, 21, 22, 25, 26, 27, 29, 30...]
Files:
[002637bd-64b5-424d-b3b1-6dd0331847b2.txt...]
```

Requête `project SOFTWARE Web pattern computer` :

```
Processing query "recherche INFORMATION Web document ordinateur"
Query processed, duration=1 ms
Result: 65 document(s)
[38, 48, 121, 195, 276, 339, 393, 402, 404, 486, 557, 565, 615, 741, 795, 807...]
Files:
[03c4340f-eaf2-40bc-989d-9c9bec4b6faf.txt]
```

Requête `project SOFTWARE Web,pattern computer` :

```
Processing query "recherche INFORMATION Web,document ordinateur"
Query processed, duration=1 ms
Result: 381 document(s)
[22, 34, 37, 38, 41, 48, 58, 72, 84, 87, 99, 103, 105, 106, 118, 121, 150, 156...]
Files:
[01df2722-3acb-4d85-9ed1-927bbbfea8e9.txt...]
```