LO17: Rapport TP2

VUATRIN Arus - ZHONG Jianzhe

June 6, 2016

1 Introduction

Dans ce TP nous allons continuer notre application pour requeter sur le corpus de texte précédemment étudié. Nous essayerons à la fin de ces séances de TP d'arriver à une application nous permettant de poser des questions dans un français courant par rapport aux différents bulletins qui composent le corpus. L'application devrait être capable de comprendre ces questions avant de retourner le résultat de notre requête. Pour cela nous commenceront par créer un correcteur orthographique afin de permettre à l'application de comprendre les mots employés par l'utilisateur, puis nous essayerons de créer une grammaire, permettant de transformer une requête en français courant vers une requête en langage SQL. Enfin, la dernière étape consistera à créer l''application finale à partir des deux travaux effectués précédemment. L'utilisateur pourra alors taper une requête en français avant de recevoir une réponse par l'application qui aura été chercher dans la base de données les informations demandées.

2 Correcteur Ortographique

La première application à réaliser est le correcteur orthographique. Celui-ci doit être capable de récupérer des mots entrés par l'utilisateur et lemmatiser les mots entrés. On essaye donc de remplacer la phrase entrée par l'utilisateur par une autre phrase qui sera connue du lexique de l'application, ainsi, elle sera d'avantage capable de reconnaître les mots écrits par l'utilisateur. Pour cela, on commence par faire des tests sur un lexique créé pour l'occasion, contenant des mots proches les uns des autres et susceptibles d'être mal lemmatisé par l'application. Ce lexique sera ensuite remplacé par le lexique complet du corpus.

2.1 Architecture

Nous avons choisi d'avoir une fonction principale de l'application s'occupant de demander continuellement à l'utilisateur d'entrer une phrase. Chaque phrase sera ensuite lemmatisée et affichée après transformation dans la sortie standard. Tant que l'utilisateur n'a pas entré une phrase vide, l'application continuera de tourner.

Lorsque l'utilisateur entre une phrase, chaque mot doit être récupéré individuellement pour être envoyé au lexique. On utilise pour cela la classe $StringTokenizer$ sur la phrase qui va boucler sur chaque mot pour les lemmatiser.

C'est un objet de type Lexique qui s'occupe de la lemmatisation d'un mot.

Cet objet a pour attributs :

- $fichierLemme$: le nom du fichier contentant les lemmes associés aux mots.
- *Dictionnary* : Un objet de type Map associant les mots à leurs lemmes. Cet objet contient les lignes du fichier de l'attribut $fichierLemme$, dans un objet permmettant des accès plus facile. Cet objet peut etre considéré comme un tableau associatif.

– *thresholdDis* : Seuil de comparaison de préfixe entre deux mots à comparer (en nombre de lettre).

– *thresholdPrefix* : Seuil de comparaison de préfixe entre deux mots à comparer (en pourcentage).

– *thresholdWord* : Seuil minimal en nombre de lettres à avoir pour avoir un lemme.

– *thresholdDisLev* : Seuil minimal obtenu par distance de levenshtein pour etre considéré commme lemme d'un mot.

Nous avons choisi de mettre les seuils dans des constantes attributs de classe afin d'avoir un accès rapide et centralisé à toutes ces constantes pour les nombreuses modifications à faire durant les tests.

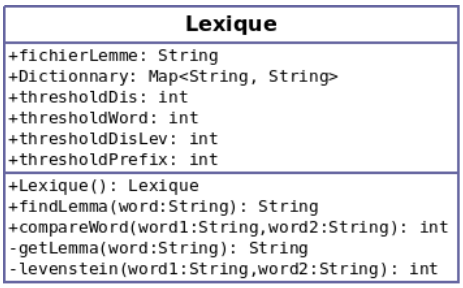

Figure 1: Diagramme UML de la classe Lexique du correcteur orthographique

2.2 Fonctions

L'objet Lexique a pour fonctions :

– *Lexique* : construit l'objet de type Lexique.

– *getLemma* : fonction privée prenant en paramètre un mot, cherche dans l'objet attribut *Dictionnary* si ce mot existe dans la liste et renvoie simplement son lemme s'il existe. Sinon, renvoie *null*.

– *compareWord* : cette fonction compte le nombre de lettres en commun entre les deux mots entrés en paramètre. Renvoie un entier étant le nombre de letttre en commun.

– *levenshtein* : fonction effectuant l'algorithme de levenshtein pour trouver la distance de levenshtein entre deux mots entrés en paramètre et renvoie un $float$.

– $findLemma$: Cette fonction prend en paramètre le mot dont on veut trouver le lemme. Elle va réaliser une série de test afin de trouver son meilleur lemme. Elle renvoie le lemme considéré comme le meilleur.

La fonction $findLemma$ commence par vérifier si le mot existe dans le dictionnnaire. Si oui, elle se contentera de renvoyer le resultat obtenu via la fonction *getLemma*. Le test suivant sera de vérifier si le mot demandé n'est pas en dessous de la longeur minimale. Si c'est le cas, on renvoie une valeur nulle. En effet, on considère ici que les mots en dessous d'un certain seuil n'apportent aucune valeur informative, il ne sera donc pas nécessaire de trouver son lemme pour essayer de l'analyser. Si le mot est assez long, on va parcourir chaque mot du dictionnaire et les comparer avec le mot demandé, dans le cas où le mot du dictionnaire est assez long. Le test préliminaire consiste à utiliser la fonction *compareWord* afin de trouver si les préfixes des deux mots ont assez de lettres en commun pour pouvoir être comparés entre eux. On utilise pour cela l'attribut $thresholdPrefix$. Si le préfixe commun a un nombre de lettre qui dépasse ce seuil,

on peut effectuer les tests suivants. On effectue le calcul simple pour connaitre le pourcentage de proximité entre les préfixes du mot demandé et du mot comparé :

$$proximity = \frac{commonPrefix \times 100}{min(wordAsked.length(), wordCompared.length())}$$

On compare ensuite ce nombre avec le seuil de proximité des préfixes pour savoir si ce mot du dictionnaire est le plus proche de celui demandé à cet instant du parcours. Si c'est le cas, on met à jour la constante de proximité des préfixes avec la proximité que l'on vient de calculer. Le prochain mot du dictionnaire comparé devra donc avoir une proximité supérieure à celle trouvée en dernier. On s'assure avec cette technique de trouver l'unique mot le plus proche.

Dans le cas où l'on a aucun mot trouvé via l'algorithme précédent, on peut passer au test de levenshtein, sinon, on renvoie le lemme associé au mot avec la meilleure proximité des préfixes trouvé.

Encore une fois, on parcourt tous les mots du dictionnaire, on rejette encore une fois les mots qui n'ont pas assez de lettres. On calcule la distance de levenshtein sur chaque mot et on vérifie que le seuil de levenshtein est bien au dessus de la constante de seuil *thresholdDisLev* avec ce test :

$$\frac{levenstein(wordCompared, wordAsked)}{min(wordAsked.length(), wordCompared.length())} \times 100 > thresholdDisLev$$

Si c'est le cas, on met à jour ce seuil et on selectionne ce mot en tant que plus proche. On obtiendra ainsi au final un seul lemme associé au mot le plus proche.

Si au terme de cette fonction aucun mot n'a été trouvé, on renvoie *null*.

2.3 Ajustements

Afin d'obtenir le mot le plus proche avec l'algorithme de recherche par préfixe et l'algorithme de levenshtein, nous devions ajuster les constantes et ajouter des fonctionalités supplémentaires. Tout d'abord, un mot long est suceptible d'avoir plus d'erreur qu'un mot court. C'est pourquoi les tests de comparaison entre le mot demandé et un mot du dictionnaire est toujours faite avec des pourcentages et non via des constantes en nombre de lettres. Si l'on comprare *moisi* avec *moisa*, on trouve une seule erreur, tout commme avec *moisir*, c'est pourquoi le mot le plus long doit être choisi, en l'occurence *moisir* étant donné que l'erreur est plus faible due au nombre de lettres.

De plus inverser deux lettres dans un mot ne devrait pas être considéré comme deux erreurs au même titre que si l'utilisateur avait tapé deux lettres totalement différentes dans le mot. Si l'on demande le lemme associé à *greande*, on devrait trouver le lemme associé au mot *grenade* plus qu'à celui du mot *grande* puisque les mots *grenade* et *greande* ont pour seule différence une permutation, l'algorithme de levenshtein considerea qu'il s'agit de deux erreurs et non d'une seule, c'est pourquoi il choisira le lemme de *grande* puisqu'il n'y a qu'une erreur entre les deux mots, qui est une omission. On a donc implémenté un test supplémentaire afin de vérifier si lorsque l'on a plusieurs erreurs dans l'algorithme, il n'y a pas eu de permutation. S'il en existe est effectivement une, on ne comptera qu'une seule erreur au lieu de deux. Donc, puisqu'on considère aussi le nombre de lettres des mots, *grenade* sera choisi commme mot le plus proche puisqu'il contient plus de lettres que *grande*.

Enfin, on devrait être dans la capacité de trouver les fautes de frappe de l'utilisateur. S'il a tapé un *r* au lieu d'un *e*, l'erreur est moins grande que s'il avait tapé un *r* au lieu d'un *m* qui est à l'opposé sur le clavier AZERTY. C'est pourquoi lors du calcul de la distance de levenshtein, on vérifie lors d'une erreur si les lettres erronées sont voisines, si elles le sont, on ne compte qu'une erreur moindre, cette valeur est stockée dans une constante. Nous avons considéré qu'une faute de frappe était une erreur commune, mais une erreur peut parfois être considéré comme une faute de frappe bien qu'elle n'en soit pas. Le clavier d'ordinateur ayant été étudié pour éviter ce genre de considération erronée, nous décidons de placer le coût d'une faute de frappe à 0,2 plutot que 1, comme l'algorithme de levenshtein l'aurait fait. Ainsi, taper *poibre* au lieu de *poivre* nous renverra bien le mot *poivre* que l'on désirais taper et non le mot *poicre* qui pourtant, contient le même nombre d'erreur.

On doit ensuite ajuster la valeur de chaque constante afin de trouver les meilleurs lemme pour chaque mot demandé.

Tout d'abord, en observant les mots du fichier contenant tous les lemmes du corpus, on se rend rapidement compte que les mots ayant moins de 3 lettres n'ont pour la grande majorité, aucune valeur informative. C'est pourquoi nous décidons de mettre la limite : *thresholdDis* à 2, les mots de 2 lettres ou moins ne seront pas traité par l'application. Il faut comme expliqué, avoir un nombre de lettre en commun proportionnel la longueur des mots comparés pour être selectionné par l'algorithme des préfixes. Après divers tests, on considère que 80% des préfixes doivent etre les mêmes, par rapport à la longeur du mot le plus long. C'est à dire que lorsque *mois* sera comparé à *moisi*, le préfixe commun est de 4 lettres (supérieur au seuil fixé précédemment) , soit 80% du mot le plus long qui contient 5 lettres, pourtant, *moisi* ne devrait pas être extrait par le correcteur orthographique. En revanche *moisir* pourrait être choisi comme mot le plus proche puisqu'ils ont 5 lettres de préfixe commun sur 8, ce qui nous donne une proximité de 83%, ce qui est supérieur strictement au seuil fixé.

Enfin pour le seuil de levenshtein on ne considère cette fois encore qu'une limite proportionnelle à la longeur du mot le plus long des deux mots comparés. Avec cet algorithme, la ou les différences entre deux mots peuvent être à n'importe quelle position dans le mot, c'est pourquoi il est nécessaire de faire attention au seuil qui mal défini, peut renvoyer des mots au sens très différents. Ainsi *comte* et *compte* ne doivent pas être rapprochés, ils n'ont pourtant qu'une seule erreur, tandis que *compter* peut être rapproché de *compte*. C'est pourquoi nous placerons la limite tel que de $\frac{1}{7} < X < \frac{1}{6}$, soit moins de 15% d'erreur par rapport au mot le plus long peuvent être accepté pour trouver le lemme d'un mot demandé.

3 Grammaire

En utilisant ANTLRWORK 1.5.2, on réalise la transformation du language naturel en requête SQL et on génère les fichiers Java correspondants. Le fichier sqlLexer s'occupe des tokens et le fichier sqlParser définit les règles de grammaire. Le lexer lit une phrase mot par mot et les regroupe en ensembles distincts qui pourront être reconnus par le parser. On essayera dans cette partie du TP de trouver des règles dans le langage naturel afin de comprendre comment se structure une requête en français. Cette structure peut varier selon le type de résultat que l'on attend. Il faut déjà distinguer les requêtes qui attendent des informations sur les articles comme *"A quelle date a été écrit tel article ?"*, avec les requetes demandant un ensemble d'information en résultat *"Quels sont les articles écrits par cet auteur ?"* et avec les requêtes dont la réponse est un nombre *"Combien d'articles traitent de ce sujet ?"*. Les requêtes peuvent aussi poser des question ouvertes telles que *"Comment soigner cette maladie ?"*, mais trouver une réponse à cette question impliquerait de comprendre la question, mais aussi tout le contenu des articles. C'est pourquoi nous étudierons seulement les trois premiers type de requête présenté, le dernier étant trop complexe. Pour finir, les requêtes doivent se terminer par une marque de ponctuation.

3.1 Organisation générale de la grammaire

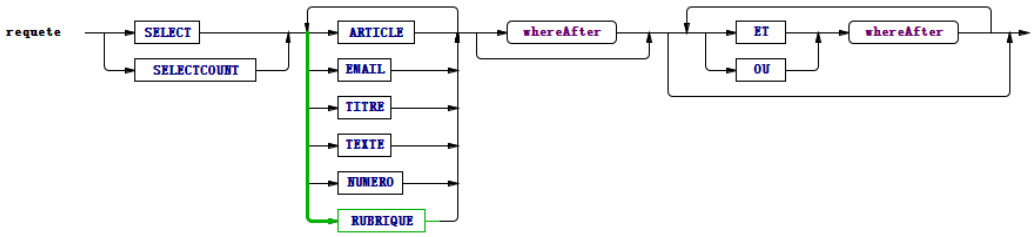

Figure 2: Structure ANTLR de la requete

Les questions peuvent donc prendre deux formes : celles qui attendent une réponse tirée d'un aggrégat de type COUNT, commencent par un mot de la classe SELECTCOUNT (ex : combien), et celles qui attendent un ensemble de résultats qui commencent par un mot de type SELECT (ex : quels, vouloir, ...). On commence par selectionner l'entité sur laquelle porte la requête, elle peut être sur les différents types de données que l'on trouve dans le corpus, ce qui représente en réalité les différentes tables de la base de données. Ce type de donnée est important à identifier pour savoir sur quelle table portera la requête. Dans tous les cas de question, on devra ensuite créer des filtres selon la demande de l'utilisateur. Ces filtres seront reconnus par la classe *WhereAfter*. Il peut y en avoir un nombre indéterminé, entre 0 et N. Chacun de ces filtres suit la même structure. Elle commence par un mot introuduisant le filtre (ex : parler, ...), suivi d'un mot permettant de rendre ce filtre optionnellement négatif (ex : pas). Nous avons choisi de créer des filtres sur seulement trois types de données :

- La date : pouvant représenter un ou plusieurs de ces paramètres : jour, mois, année. Ces données seront reconnues par la classe *paramDate* selon différents formats.
- Un mot : si l'entité que l'on selectionne doit être filtrée selon un mot qu'elle contient, on doit trouver une chaîne de la classe *paramMot* dans la requête.
- Filtres numériques : la grammaire nous permet aussi de filtrer les résultats selon des valeurs numériques diverses grâce à la classe *paramNumber* avec des comparateurs et les entités à compter.

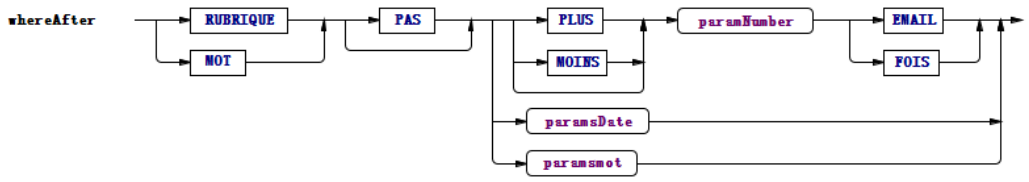

Figure 3: parser whereAfter

3.2 Utilisation de la grammaire

Pour être transformée, les question de l'utilisateur doivent d'abord être formatées. On va pour cela utiliser le correcteur orthographique créé dans la première partie de ce TP. Les phrases seront tapées par

l'utilisateur, puis les mots que l'utilisateur aura tapé seront identifiés par la classe Lexique et seront remplacés par les mots les plus proches du dictionnaire que l'on aura associé au lexique. Ce premier dictionnaire associera chaque mot rencontré dans le corpus à son lemme (ce fichier a été généré lors du TP pécédent). Dans un second temps on créera un dictionnaire associé à notre grammaire. Ce second dictionnaire associe chaque lemme à un mot de chaque classe de notre grammaire. En utilisant cette méthode de double dictionnaire, nous avons remarqué que les mots recherchés dans les requêtes des utilisateurs étaient mieux compris par notre application. La classe lexique a donc vu ses méthodes *getLemma* et $findLemma$ adaptées à ces modificaions. Dans le second dictionnaire, on doit dans un premier temps retirer tous les mots que l'on considère comme inutiles à la compréhension de la requête pour notre grammaire. On y trouve donc des mots tels que : 'des', 'de', 'qui, 'dont'... Ces mots n'auront aucun autre mot en face d'eux-même dans le dictionnaire, l'application comprendra alors que ces mots doivent tout simplement être supprimés de la phrase entrée. Puis on doit identifier les mots de chaque classe de la grammaire. Par exemple, l'utilisateur est suceptible d'utiliser le terme 'citer' ou 'traiter' au lieu de 'parler', mots que l'on a classé dans notre grammaire dans la classe MOT. Pour nous aider dans notre construction du dictionnaire, on s'appuie sur un ensemble de questions que les utilisateurs poseront potentiellement à notre application, on a ainsi un bon échantillon des mots de chaque classe constituant la grammaire.

3.3 Types de requêtes reconnues

Les lignes suivantes sont consacrés à la présentations de diverses requêtes pouvant être comprises par notre grammaire.

Cette liste s'organise comme explqué ci dessous :

- Requête en langue naturelle, telle que l'a demandé l'utilisateur.
- Requête corrigée par le dictionnaire.
- Requête lemmatisée après le passage par le correcteur orthographique associé à la grammaire.
- Requête transformée en SQL.

1. - Je veux les articles dont le titre contient issue ou vision.
   - je veux les article dont le titre contien issue ou vision.
   - vouloir articles titre parler issue ou vision.
   - select distinct fichier from titre where mot= 'issue' OR mot= 'vision'
2. - Je veux les articles dont le titre contient bel et physique.
   - je veux les article dont le titre contien bel et physique.
   - vouloir articles titre parler bel et physique.
   - select distinct fichier from titre where mot= 'bel' INTERSECT select distinct fichier from titre where mot= 'physique'
3. - Combien d'articles datent du 24 janvier 2012.
   - combien d article datent du 24 janvier 2012.
   - combien articles dater 24 janvier 2012.
   - SELECT COUNT(*) AS count FROM (select distinct fichier from date where (date.jour= '24' AND date.mois= '01' AND date.annee= '2012'))I
4. - Combien d'articles datent du 24 janvier 2012 ou du 29 septembre 2011.
   - combien d article datent du 24 janvier 2012 ou du 29 sept 2011.

- combien articles dater 24 janvier 2012 ou 29 sept 2011.
- SELECT COUNT(*) AS count FROM (select distinct fichier from date where (date.jour= ’24’ AND date.mois= ’01’ AND date.annee= ’2012’) OR (date.jour= ’29’ AND date.mois= ’09’ AND date.annee= ’2011’))I ;

5. – Combien d’articles datent du 21 juin 2011 et contiennent bel et physique.
   - combien d article datent du 21 juin 2011 et contien bel et physique.
   - combien articles dater 21 juin 2011 et parler bel et physique.
   - SELECT COUNT(*) AS count FROM (select distinct fichier from date where (date.jour= ’21’ AND date.mois= ’06’ AND date.annee= ’2011’) INTERSECT select distinct fichier from titretext where mot= ’bel’ INTERSECT select distinct fichier from titretext where mot= ’physique’)I

6. – Je veux les articles qui ont été publiés en avril ou le 24-01-2012.
   - je veux les article qui ont été publiés en avril ou le 24-01-2012.
   - vouloir articles publier en avril ou 24-01-2012.
   - select distinct fichier from date where (date.mois= ’04’) OR (date.jour= ’24’ AND date.mois= ’01’ AND date.annee= ’2012’)

7. – Je veux les articles qui ont été publiés en avril 2012.
   - je veux les article qui ont été publiés en avril 2012.
   - vouloir articles publier en avril 2012.
   - select distinct fichier from date where (date.mois= ’04’ AND date.annee= ’2012’)

8. – Quels articles ont été publiés le 20 juin 2014.
   - quel article ont été publiés le 20 juin 2014.
   - vouloir articles publier 20 juin 2014.
   - select distinct fichier from date where (date.jour= ’20’ AND date.mois= ’06’ AND date.annee= ’2014’)

9. – Je veux les articles qui ont été publiés avant 2012 et contiennent physique.
   - je veux les article qui ont été publiés avant 2012 et contien physique.
   - vouloir articles publier avant 2012 et parler physique.
   - select distinct fichier from date where (date.annee¡ ’2012’) INTERSECT select distinct fichier from titretext where mot= ’physique’

10. – Quels sont les articles dont le texte contiennent compris?
    - quel sont les article dont le text contien compris.
    - vouloir articles parler compris.
    - select distinct fichier from texte where mot= ’compris’;

11. – Quels sont les articles dont le texte ne contiennent pas compris?
    - quel sont les article dont le text ne contien pas compris.
    - vouloir articles parler pas compris.
    - select distinct fichier from texte where mot= ’compris’;

12. – Je veux les articles qui ont été publiés entre 2012 et 2013 mais pas juin.

- je veux les article qui ont été publiés entre 2012 et 2013 mais pas juin.
- vouloir articles publier entre 2012 et 2013 mais pas juin.
- select distinct fichier from date where (date.annee¿= ’2012’) AND (date.annee¡= ’2013’) AND (date.mois!= ’06’)

13. – Quels sont les articles dont le texte contient plus de 700 mots ?
    - quel sont les article dont le test contien plus de 700 mots.
    - vouloir articles test parler plus 700 mot.
    - select distinct fichier from texte GROUP BY fichier HAVING COUNT(mot)¿=700;
14. – Je veux les articles qui datent du 1 decembre 2012 et dont la rubrique est Actualites Innovations.
    - je veux les article qui datent du 1 decembre 2012 et dont la rubrique est actualites innovation.
    - vouloir articles dater 1 decembre 2012 et rubrique actualites innovation.
    - select distinct fichier from date where (date.jour= ’1’ AND date.mois= ’12’ AND date.annee= ’2012’) INTERSECT select distinct fichier from RUBRIQUE where rubrique= ’actualites innovation’
15. – Quelles sont les rubriques dont les articles parlent de sport?
    - quel sont les rubriques dont les article parlent de sport.
    - vouloir rubrique articles parler sport.
    - select distinct rubrique , fichier from titretext where mot= ’sport’
16. – Je veux tous les articles.
    - je veux tous les article.
    - vouloir tous articles.
    - select distinct fichier from titretext

Comme on peut le remarquer les deux premières lignes de chaque point ne sont pas très différentes. Mais elles s’assurent de ne pas avoir de fautes de frappes . Par exemple, si l’utilisateur tape : *quels articles parlent d’innobation et de rechreche*, le texte corrigé sera : *quel article parlent d innovation et de recherche.*, puisque les mots seront reconnus par le dictionnaire.

3.4 Présentation des arbres associés aux requêtes

Dans cette section, on présentera les arbres qui seront générés avec des exemples pour chaque type de requête.

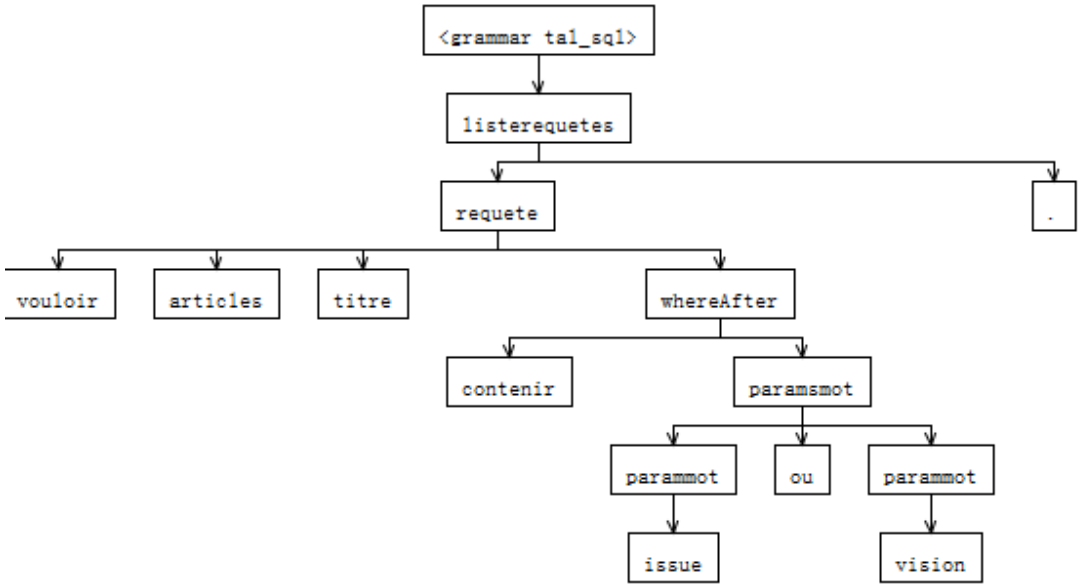

Figure 4: Je veux les articles dont le titre contient issue ou vision.

Dans un premier temps, on retire les mots comme *je*, *dont*, *le* qui ne sont pas importants. Lorsqu'on detecte les mots clés *article* et *titre* dans cet ordre, on comprend que l'on veut les articles selon un filtre sur leur titre. Nous avons choisi que si un utilisateur demande des résultats en tant qu'article, on lui renvoie le numéro de fichier. A la détection du mot *contenir*, on sait que l'on va filtrer selon le contenu, cela permet donc de placer un noeud de classe $whereAfter$ dans l'arbre. La classe *paramsMot* constitue les filtres mot= 'issue' OR mot= 'vision' étant donné que l'on détecte la conjonction *ou*.

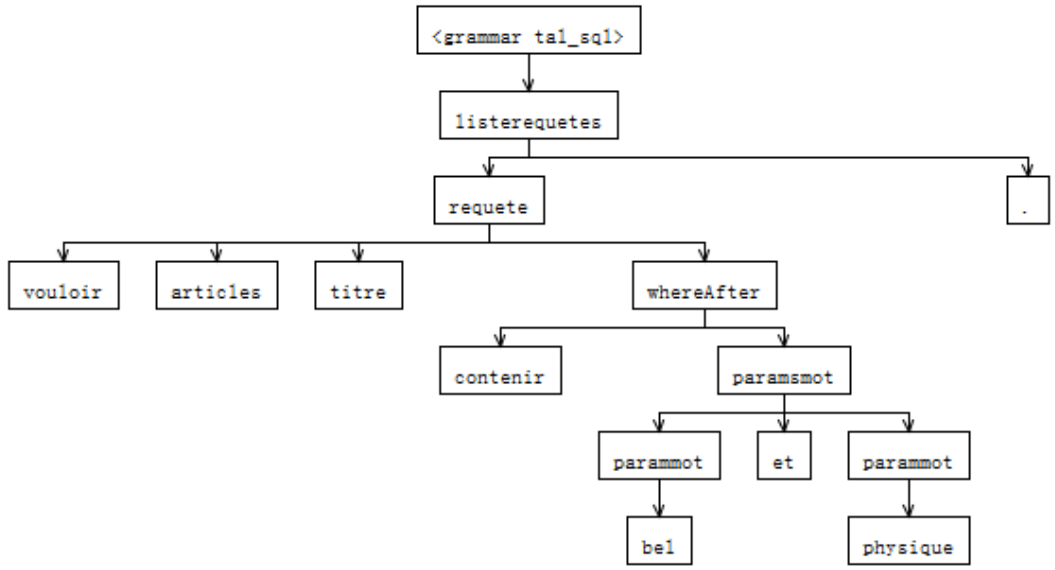

Figure 5: Je veux les articles dont le titre contient issue et vision.

Tout comme pour le prmier arbre, mais le mot *et* est détecté au lieu du mot *ou*.

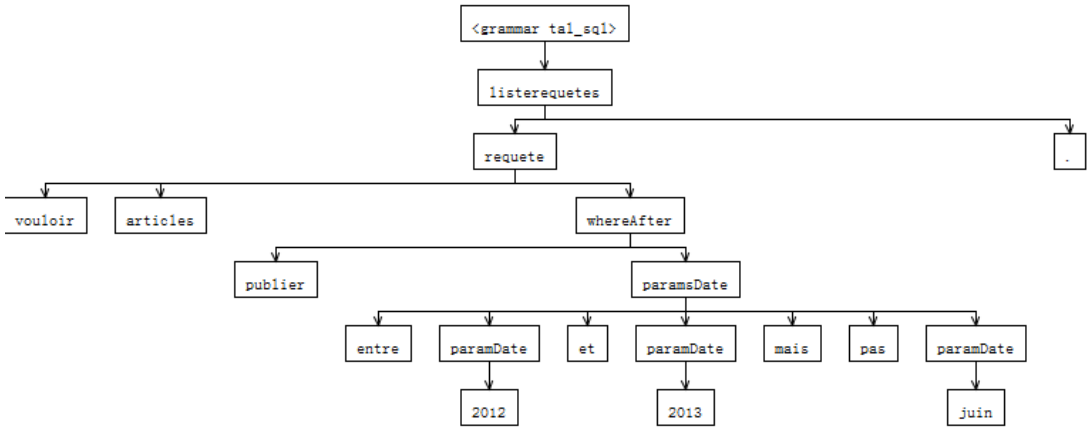

Figure 6: Je veux les articles publiés entre 2012 et 2013 mais pas en juin.

On détermine d'abord ce que veut recevoir l'utilisateur en posant cette requête. Ici, il désire les articles. Le mot *publié* indique que l'on se trouve dans une condition. Puis une analyse des mots suivants nous permet de dire que une date a été écrite. Le mot *entre* ne doit pas être classé comme mot indésirable. En effet, il nous indique que deux dates ont été renseignées dans la suite de la requête. Puis on essaye de trouver le format dans laquelle la date a été renseignée. Ici, on ne détecte qu'un nombre à 4 chiffres, ce qui est automatiquement assimilé à une année. On détecte donc les deux années qui font partie des classes *paramDate*. Puis, différentes conjonctions sont détectés comme *mais*, *pas* et enfin, un mot classé

comme un mois. Comme on peut le voir, il s'agit simplement ici de détecter les paramètres de date et les conjonctions indiquant des conditions. En classant chaque mot selon sa classe, ce type de phrase est très simple à comprendre.

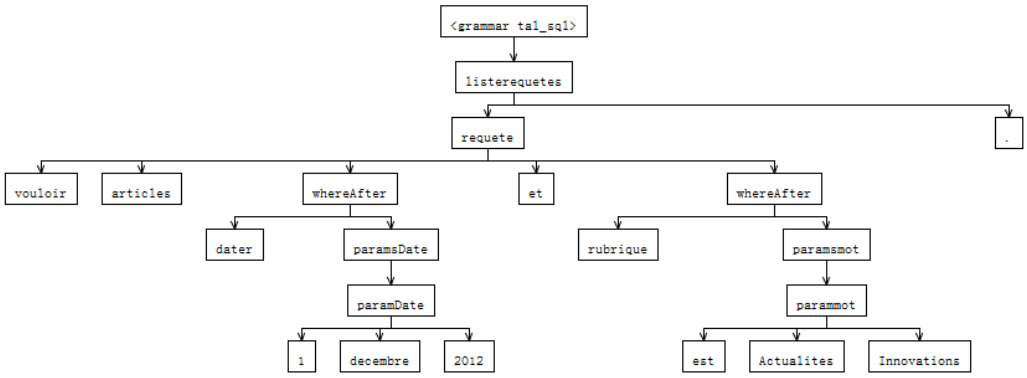

Figure 7: Je veux les articles qui datent du 1 decembre 2012 et dont la rubrique est Actualites Innovations.

Dans ce cas-là, on rencontre la une condition sur la date introduit par le mot *dater* ainsi qu'une autre portant sur la rubrique de l'article introuduit par le mot *rubrique*. Nous avons choisi qu'un mot de la classe *Rubrique* pouvait être considéré comme un mot introduisant une condition, à partir du moment où elle n'est pas juste après un mot de type *select*. Pour la date, on doit détecter le format, dans le cas présent on détecte le format : un nombre de moins de 31 , un nom de mois suivi d'un nombre de 4 chiffres. Ainsi, on peut classer facilement chaque paramètre de la date. Pour la détection de la rubrique, on cherche tous les mots après le terme de la classe *rubrique* qui n'est pas un mot d'une autre classe. On trouve ici deux mots dans la classe *Actualites* et *Innovation*, le mot *est* étant supprimé.

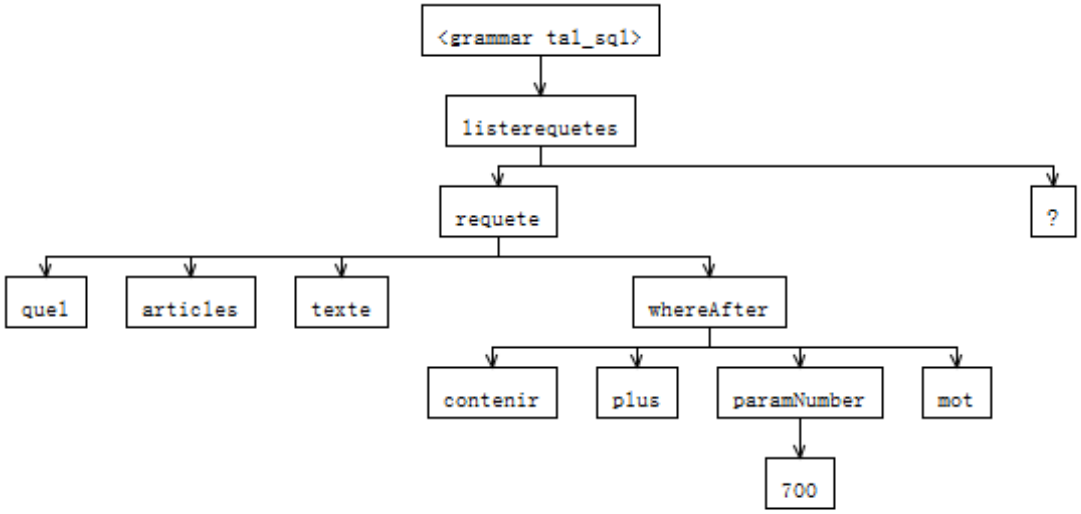

Figure 8: Quels sont les articles dont le texte contiennent plus de 700 mots ?

Dans ce cas, on a fait une requête portant sur une caracteristique du texte. Le format du début de la requête est classique, il ne peut pas être d'une autre forme dans notre implémentation lorsque l'on pose des questions sur une caractéristique du texte. Cet exemple ne peut porter que sur le nombre de mots du texte ou du titre d'un article. On detecte ici le format de filtre *whereAfter* : mot indiquant le contenu, condition sur le *paramNumber* comme *plus*, *moins* ou *égal à*, puis le paramètre est détécté comme un nombre, et enfin le mot sur lequel porte cette condition est détécté. Dans notre cas, ce dernier sera toujours mot, car écrire email ou illustration impliquerait de faire des jointures supplémentaires.

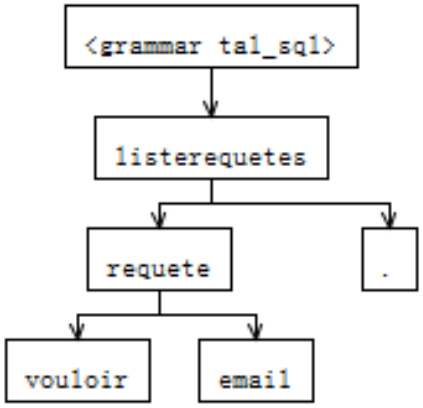

Figure 9: Je veux tous les emails.

4 Application Finale

Enfin, nous devons finaliser l'appication en mettant en place l'interface dont l'utilisateur se servira pour poser ses questions. A partir du correcteur orthographique et de la grammaire, nous allons essayer dans un premier temps de faire en sorte que notre application comprenne les requêtes qui lui sont transmises, qu'il les lemmatisent à partir des lemmes trouvés dans l'étape précédente, transforme cette requête en francais courant vers une requête SQL à partir de la grammaire effectuée précédemment, puis communique avec la base de données afin d'aller chercher les réponses avant de les afficher à l'utilisateur.

4.1 Format de la Base de données

La base de données est construite comme expliqué sur le diagramme simplifié ci dessous :

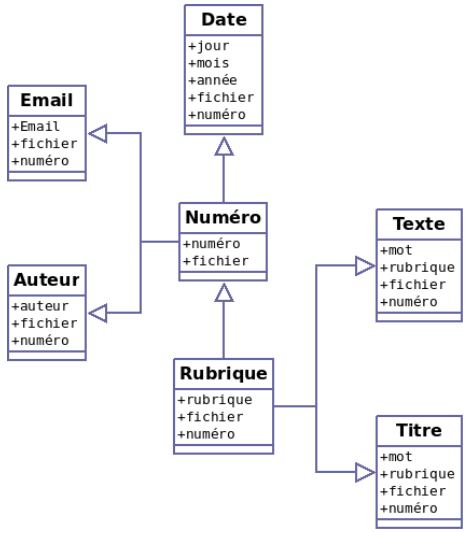

Figure 10: Diagramme UML simplifié de la base de données

Comme on peut le remarquer, chaque article est décrit par son numéro et le nom du fichier associé. La table numéro référence donc tous les articles. La table rubrique référence l'appartenance aux différentes rubriques de chaque fichier. La table email reprend les attributs descriptifs de chaque articles pour les associer aux différents emails apparaissant dans la section contact de l'article. La table date décrit le jour le mois et l'année d'écriture de l'article en tant que des entiers. La table texte reprend tous les attributs de la table rubrique et ajoute une entrée dans la table pour chaque mot apparaissant dans l'article. La table Titre fait de même avec chaque mot du titre d'un article. Dupliquer l'attribut rubrique nous permet d'avoir des requêtes moins complexes lors de demandes telles que *"Dans quelle rubrique apparaît le mot innovation ?"*

5 Communication avec la BDD via une Interface Java/Postgresql

Pour finir, nous allons mettre en place l'interface Java/Postgresql pour pouvoir faire communiquer la base de données contenant les tables sur lesquelles nous feront les requêtes, avec notre application java qui récupérera les questions, de l'utilisateur, les formatera avant de les transformer en SQL. Les fonctions nécessaires à cette interface sont dans le fichier $postgresql - 9.0 - 801.jdbc4.jar$. Pour se connecter, nous devons renseigner les informations de la base, ce qui nous renvoie un objet de type *Connexion*. C'est à partir de cet objet que l'on pourra créer des requêtes via la fonction $createStatement()$. On execute la requête et

on récupére un objet de type *ResultSet*. Cet objet contient toutes les données en réponse à notre question initiale. Cependant, on ne peu pas savoir quelles colonnes contient cet objet étant donné que les colonnes selectionnées par la requête executée est différente à chaque fois que l'utilisateur pose une nouvelle question. C'est pourquoi on utilise un objet de type *ResultSetMetaData* qui nous permet d'obtenir chqaue entrée pour chaque colonne demandée. On récupère ainsi toutes les réponses renvoyées par la base colonne par colonne.

5.1 Ajustements supplémentaires

Afin d'avoir moins de problèmes au cours de l'execution de l'application, nous avons organisé notre code comme expliqué ci-après. Au lancement, on vérifie si la connexion avec la base de données est possible, si elle l'est, on demande à l'utilisateur de taper des phrases tant qu'il n'a pas tapé un caractère marquant la fin de l'execution. A chaque requête tapé par l'utilisateur, l'application formatte la phrase, c'est à dire qu'elle remplace les caractères indésirables tel que les guillements et les signes de ponctuation. Nous avons choisi de retirer la ponctuation des phrases de l'utilisateur car ceux-ci n'apportent pas d'information, elle sera rajoutée après transformation, avant d'etre envoyé à la grammaire. Pour la transformation de la requete, on utilise les fichiers expliqués dans la partie 3.2. Chaque token tapé par l'utilisateur sera remplacé par le lemme d'un mot existant dans le corpus, puis par un mot reconnu par la grammaire. On obtient ainsi une requête transformée, compréhensible par la grammaire. On envoie cette phrase à la grammaire qui, si elle ne la comprend pas, lève une exception, sinon, execute la requête SQL et affiche les résultats comme expliqué précédemment.

6 Conclusion

Au final, nous obtenons une application capable de reconnaitre quelques une de nos demandes. La variété des phrases pouvant être reconnues ne dépend que de la grammaire créée. La grammaire est le travail qui nous a demandé le plus de temps, le français étant une langue particulièrement riche et avec de nombreuses variations, il est difficile de penser à toutes les requêtes possibles. Cependant, on ne peut que remarquer que le langage naturel reste assez proche du langage SQL. Les requêtes simples ont le même format, ce qui rend la grammaire plus simple pour les questions simples du type de celle de la figure 4, tandis que lorsque les requêtes demandent des jointures entre différentes tables de la base de données, la grammaire s'en trouve grandement complexifiée, comme pour les requêtes ressemblant à celle de la figure 7.