LO17 rapport2

1. Grandes lignes du rapport

1.1 Bref rappel du projet

Au cours de l'UV LO17, nous allons former une interface qui permettra à l'utilisateur de faire une recherche dans un corpus d'articles parus dans une revue scientifique avec une phrase écrite en français. Une première partie du projet a été de découper ce corpus en unités quantifiables et analysables, en en extrayant le maximum de sens pour pouvoir effectuer des recherches intéressantes. Nous avons joué sur une analyse des mots présents pour en déceler les mots inutiles car vides de sens de ceux possédant un sens utile. Nous avons aussi procédés à une lemmatisation pour permettre à la recherche de récupérer aussi des mots dont le sens est proche de celui rentré (ex : rechercher le mot chien inclurait aussi le mot chienne car le sens est le même malgré le fait que les mots soient différents).

Avec cette structure d'information, nous devons maintenant créer une interface, qui devra tout d'abord identifier une requête de l'utilisateur en français, en tirer les éléments clés et les transformer en une instruction valide par la structure de la base de données.

1.2 Introduction

Pour réaliser cette partie, nous allons séparer notre travail en 3 parties :

- la morphologie : corriger les mots rentrés par l'utilisateur.
- la grammaire : interpréter les mots rentrés "corrigés" en une requête proche d'être interprétable par le langage.
- génération de l'arbre SQL : transformation du retour de la grammaire en un phrase interprétable par le langage d'interrogation de la base de donnée, ici SQL.

Pour réaliser ces 3 parties nous allons utiliser plusieurs langages de programmation.

Tout d'abord la base de notre interface utilisera le langage JAVA pour être implémenté. Nous allons aussi utiliser une partie de nos résultats obtenus dans les précédents TDs. Ensuite la partie sur la grammaire utilisera le logiciel Antlr, qui permet de définir une grammaire, c'est à dire définir des types de mots et des enchaînements qui peuvent se représenter de façon visuelle par un graphe, et d'interpréter et représenter une phrase que nous lui donnons sous forme d'un arbre. Antlr permet aussi d'exporter la grammaire de façon exploitable pour être incorporée dans notre architecture JAVA. Enfin notre base de donnée est hébergée sous PostgreSQL, dont le langage d'interrogation est SQL. JAVA est pratique car la relation avec SQL est très souvent utilisée.

2. La partie morphologie

2.1) Programme de correction orthographique

Le but de cette partie est d'obtenir une correction orthographique pour chaque mot de la phrase entrée au clavier par l'utilisateur. Le résultat retourné est le lemme associé au mot. Nous utilisons la classe "StringTokenizer" pour obtenir tous les mots de la phrase un à un. Pour chaque mot, on fait les analyses ci-dessous :

(a) le convertir en minuscules (motActuel.toLowerCase ())

(b) si le mot existe dans la liste des lemme, on retourne directement le lemme. (Ici, nous n'avons pas pris en compte la *stoplist*, mais après lors de la combinaison avec la partie grammaire, nous ajouterons la *stoplist*. Si le mot existe dans la *stoplist*, on retournera alors une *string* vide.)

(c) Sinon, on compte le nombre de lettres communes entre le début du mot et le début d'un mot du lexique (algorithme de recherche par préfixe) et si ce nombre est suffisamment grand : stocker le lemme dans une table de hachage (mot->proximité) de lemmes candidats, et à la fin on retourne la sous-liste des meilleurs lemmes candidats (d'un au trois meilleurs lemmes).

(d) Si la recherche par préfixe n'a rien retourné, on teste l'algorithme de Levenshtein. Si la réponse est positive : stocker le lemme dans une table de hachage (mot->distance-Levenshtein) de lemmes candidats, et à la fin on retourne la sous-liste des meilleurs lemmes candidats (entre un et trois meilleurs lemmes).

(e) Sinon, on indique qu'aucun mot été trouvé (pour la combinaison avec la partie grammaire, si un mot est dans ce cas, on retourne le mot lui-même).

2.2) Calculs de proximité orthographique

Pour réaliser cette partie, nous commençons par la création de la classe "Lexique" et la méthode "getLemmes". Cette méthode sera la méthode publique à appeler pour obtenir les meilleur lemmes pour chaque mot, et elle sera appelée pour chacun des mots de la phrase entrée. Cette méthode va appeler successivement, si besoin, les différents types de recherche évoqués ci-dessus. Tout d'abord elle recherche si le mot est directement présent dans la liste de lemmes.

Dans un second temps, nous essayons la recherche par préfixe en appelant la méthode "prox". Dans cette méthode, la proximité est mesurée par le rapport entre le nombre de lettres communes entre le début des deux chaînes, et la longueur maximale des deux chaînes. La réalisation de cette fonction en JAVA est présentée ci-dessous :

```
public float prox(String mot1, String mot2){
    int longueur1 = mot1.length();
    int longueur2 = mot2.length();
    float  proxM1M2 = 0;
    if(longueur1 < seuilMin || longueur2 < seuilMin)
        proxM1M2 = 0;
    else if(Math.abs(longueur1 - longueur2) > seuilMax)
        proxM1M2 = 0;
    else{
        int i =0;
        while(i < Math.min(longueur1, longueur2) && (mot1.charAt(i) == mot2.charAt(i))){
            i++;
        }
        proxM1M2 = i * 100 / Math.max(longueur1, longueur2);
    }
    return proxM1M2;
}
```

Dans la méthode “getLemmes”, cette méthode “prox” est appelée pour comparer le mot avec chacun des mots présents dans la liste des lemmes.

```
Set<String> keys = Lemmes.keySet();
String[] motLexique = new String[keys.size()];
keys.toArray(motLexique);
//recherche par préfixe
float prox;
for(String m : motLexique){
    prox = prox(mot, m);
    if(prox >= 50){
        lemmesCandidats.put(Lemmes.get(m), prox);
        }
    }
ArrayList<Map.Entry<String, Float>> list = new ArrayList(lemmesCandidats.entrySet());
Collections.sort(list,(o1,o2)->(int)(o2.getValue()-o1.getValue()));

String Key =null;
for(int i=0;i<list.size()&&i<meilleureLemmes.length;i++){
  Key = list.get(i).getKey();
  if(Key!=null)
    meilleureLemmes[i] = Key;
}
if(meilleureLemmes[0] != "")
  return meilleureLemmes;
```

Ici, si la proximité entre le mot entré et le mot dans lexique est supérieur ou égale à 50%, on met le lemme correspondant dans le lexique comme lemme candidat dans une table de hachage avec le lemme comme clé et la proximité comme valeur (lemme->proximité). Après, on trie la table de hachage en ordre décroissant en fonction de la proximité, comme ça on peut bien extraire la sous-liste des meilleurs lemmes candidats. Si de tels lemmes vérifient cette tolérance, le nombre de meilleurs lemmes retourné est compris entre 1 et 3 (nous avons fixé la longueur du tableau “meilleurLemmes” à 3).

Si on n'a pas trouvé lemme par la recherche par préfixe, on utilise l’algorithme de Levenshtein. Cette méthode détermine la proximité des deux chaînes en calculant la distance de Levenshtein entre les deux chaînes. Dans notre programme, on prend des coûts de remplacement, d’insertion et de suppression tous égaux à 1. La réalisation de cette fonction en JAVA est présentée ci-dessous :

```
private int levenshteinDist(String mot1, String mot2){
    int d1=0,d2=0,d3=0;  //d1:remplace; d2:suppression; d3:insertion
    int longueur1 = mot1.length();
    int longueur2 = mot2.length();
    int dist[][] = new int[longueur1+1][longueur2+1];
    int minD1D2 = 0;
    dist[0][0] = 0;
    for(int i = 1; i <= longueur1; i++)
        dist[i][0] = dist[i-1][0] + 1;
    for(int j = 1; j <= longueur2; j++)
        dist[0][j] = dist[0][j-1] + 1;
    for(int i = 1; i <= longueur1; i++){
        for(int j = 1; j <= longueur2; j++){
            if(mot1.charAt(i-1) == mot2.charAt(j-1))
                d1 = dist[i-1][j-1];
            else{
                d1 = dist[i-1][j-1] + 1;
            }
            d2 = dist[i-1][j] + 1;
            d3 = dist[i][j-1] + 1;
            minD1D2 = (d1 < d2)?d1:d2;
            dist[i][j] = (minD1D2 < d3)?minD1D2:d3;
        }
    }
    return dist[longueur1][longueur2];
}
```

L'idée d'utilisation de la fonction "levenshteinDist" est pareille que la précédente. Si la distance entre le mot entré et le mot de lexique est inférieure à 3, on met le lemme correspondant comme un lemme candidat dans une table de hachage. Ensuite, nous trions la table de hachage en ordre croissant en fonction de la distance Levenshtein. Nous retournons ici tous les lemmes candidats car une distance de 3 est très petite.

```
//Levenshtein
int levenshteinDist;
for(String m : motLexique){
   levenshteinDist = levenshteinDist(mot, m);
  if(levenshteinDist < 3){
    lemmesCandidats.put(Lemmes.get(m), (float) levenshteinDist);
    }
}
ArrayList<Map.Entry<String, Float>> list2 = new ArrayList(lemmesCandidats.entrySet());
Collections.sort(list2,(o1,o2)->(int)(o1.getValue()-o2.getValue()));
String Key2 ="";
for(int i=0;i<list2.size()&&i<meilleureLemmes.length;i++){
  Key2 = list2.get(i).getKey();
  if(Key2!=null)
    meilleureLemmes[i] = Key2;
}

if(meilleureLemmes[0]!="")
  return meilleureLemmes;
```

2.3) Réglages des coefficients

Pour choisir les coefficients, nous avons fait certaines analyses sur le lexique du corpus. On a obtenu les chiffres ci-dessous :

Le nombre total de mot dans lexique : 13974, noté “n”,

La moyenne de la longueur des mots dans lexique : 8.1, noté “mLM”,

La moyenne de la longueur des lemmes dans le lexique : 6.89, noté “mLL”,

La moyenne de la longueur du lemme associé sur la longueur du mot initial : 87.9%, noté “mLLsurLM” = (Σ(longueur du lemme/longueur du mot))/ longueur du lexique,

La proportion de mot dont la longueur est inférieure ou égale à 13 : 96.1%,

La proportion de mot dont la longueur est inférieure ou égale à 4 : 9.4%,

La proportion de mot dont la longueur est inférieure ou égale à 3 : 3.8%,

La proportion de mot dont la longueur est inférieure ou égale à 2 : 0.6%,

Pour la méthode de la recherche par préfixe, on a décidé d’une borne de 50%, c’est à dire que si la proximité entre deux mots est supérieure ou égale à 50%, on pense que l’on a trouvé le mot (candidat). Tout d’abord, nous avions pensé à utiliser “mLLsurLM”, 87.9%, comme borne, mais cette valeur est trop haute car pour un mot de 10 lettres, on doit taper correctement 9 lettres au début d’un mot (ce qui est presque équivalent à tout rentrer correctement). Donc, si une personne peut entrer correctement la première moitié ou plus d‘un mot, alors nous pensons qu'elle connaît le mot. Cela permet de reconnaître des mots dont la déclinaison syntaxique est longue (exemple : premières et premier).

Sur la base des calculs ci-dessus, nous avons déterminé que le “seuilMin” dans l'algorithme est égal à 3 et le “seuilMax” est égal à 6. C’est à dire que si la longueur du mot d'entrée est inférieure à 3, nous n'utilisons pas cette méthode. Nous estimons que les mots de longueur 1 ou 2 sont plus susceptibles d'exister dans la *stoplist*. Sans “seuilMin” si une personne ne connaît que les deux premières lettres d'un mot de longueur 3 ou 4, un tel algorithme pourrait également trouver le mot correct.

Il existe deux raisons principales de choisir “seuilMax” égal à 6. Premièrement, pour 96% des mots, leur longueur est inférieure ou égale à 13. Si la différence entre la longueur du mot saisi et le mot en lexique est supérieure à 6, leur similarité est certainement inférieure à 50%. Par conséquent, nous n'avons pas besoin de compter le nombre de lettres communes une par une, nous pouvons gagner plus de temps. Dans un second temps, nous avons considéré le vocabulaire dérivé. Par exemple, s'il n'y a que « fonctionnement » dans lexique, mais que l'on souhaite interroger « fonction », leur similarité est de 57%. Si “seuilMax” était défini sur une valeur trop petite, nous ne pourrions pas l'interroger. Ici, leur écart est exactement de 6. Et « nement » est presque le suffixe dérivé le plus long présent en français, donc se baser sur cette longueur semble être une approximation correcte.

Pour la méthode Levenshtein, on a choisi un coefficient de 3, c’est à dire que si la distance entre deux mot est inférieur à 3, on pense que l’on a trouvé un bon mot candidat. La méthode de Levenshtein est appelée après la méthode de recherche par préfixe, donc,

elle est plus précise. Deux cas se présentent souvent : il y a un article et une apostrophe juste avant un mot qui commence par une voyelle, et deux lettres du mot sont inversées lors de la saisie. Pour les deux cas, la distance de Levenshtein est 2, nous avons donc choisi un seuil égal à 3.

2.4) Résultat obtenus

Nous avons testé plusieurs situations avec le programme. Nous prenons le mot “article” comme un exemple. D’abord, nous saisissons “article”.

Il renvoi le meilleur lemme : “article”. Car le mot “article” existe exactement dans le lexique et le lemme de “article” est “article”.

```
saisie : article
j'ai saisi article

les meilleurs lemmes pour article :
--article
```

Si nous saisissons “articlzs” (“articles” et “articlzs” sont très similaires. Comme sur le clavier, la lettre ‘z’ est à côté de la lettre ‘e’, ce genre d'erreur est possible). On peut voir que le programme retourne trois meilleurs lemmes : “article”, “articul” et “artichaut”. En fait, le lemme “article” est exactement ce que on veut. Dans le lexique, on a mot “article” et “articles”. Si le mot ’articlzs’ est comparé avec ‘article’, la longueur de ‘articlzs’ est supérieur à 3 et la différence de longueur entre ‘article’ et ‘articlzs’ est 1, ce qui est inférieur à 5, donc la proximité est 6/8 =75%, on retourne le lemme de ‘article’ dans le lexique.

```
saisie : articlzs
j'ai saisi articlzs

les meilleurs lemmes pour articlzs :
--article
--articul
--artichaut
```

Dans un cas comme ‘‘l’article’’, la classe “StringTokenizer” ne peut pas le séparer au niveau de «l’ » et «article». Donc, on doit considérer «l’article» comme un ensemble. Mais ça ne marche pas avec la méthode de recherche par préfixe parce que le nombre de lettres communes au début est trop faible. C’est donc traité par la méthode de Levenshtein. Pour transformer «l’article» en «article», on a besoin de juste deux suppressions ayant un coût total 2. La distance levenshtein est donc égale à 2 qui est inférieur à 3. Par conséquence, on prend le lemme de ‘article’ dans le lexique comme meilleur lemme.

```
saisie : l'article
j'ai saisi l'article

les meilleurs lemmes pour l'article :
--article
```

Après nous avons testé un deuxième cas pour la méthode de Levenshtein. Le mot ‘superior’ avec deux lettres inversés est ‘sueprior’. La méthode de Levenshtein a bien trouvé le mot le plus proche et a retourné son lemme correspondant ‘super’.

```
saisie : sueprior
j'ai saisi sueprior

les meilleurs lemmes pour sueprior :
--super
```

Après, on a testé le programme sur un exemple de phase.

```
saisie : articlzss gonctionne sueprior
j'ai saisi articlzss gonctionne sueprior

les meilleurs lemmes pour articlzss :
--article--artichaut--articul

les meilleurs lemmes pour gonctionne :
--fonction

les meilleurs lemmes pour sueprior :
--super
```

2.5) Commentaires critiques sur les résultats

En général, nous arrivons à compléter la correction des mots entrés. Dans le test ci-dessus, nous avons retourné 1 à 3 des meilleurs Lemmes. Selon les règles de tri, nous pensons que le premier élément du tableau retourné est le meilleur et lors de la génération ultérieure au format SQL, nous utiliserons également le premier élément du tableau retourné. En plus, ici, si nous ne parvenons pas à corriger les mots entrés avec les trois méthodes, nous vous indiquons qu’aucun mot est trouvé. Évidemment, cela ne convient pas à la génération SQL. En fait, nous retournerons le mot d'entrée lui-même pour la génération SQL.

Il existe également certaines limitations à cet algorithme, notamment Levenshtein. Puisque nous autorisons uniquement les erreurs de deux lettres, pour l'exemple «l'article» du test précédent, si nous commettons une erreur dans le mot « article », tel que « l'articlzs », notre programme ne trouvera pas le mot « article ». Afin d'éviter cette situation, nous allons pré-supprimer les « l’», « d’ », « qu’ » et « s’ » dans un mot pendant la phase de normalisation.

3. Partie grammaire

3.1) Introduction

La grammaire va nous permettre d’interpréter et convertir en requête compréhensible par la base de donnée la demande de l’utilisateur. Pour ce faire il faut définir une structure

pour comprendre la formulation des demandes de l'utilisateur. L'exemple le plus basique sur lequel nous nous sommes basés pour commencer à former notre grammaire est du type :

"Je veux les articles qui parlent de mitochondries".

On peut donc commencer à formuler une grammaire qui va se découper en plusieurs parties :

- Tout d'abord un verbe (ou expression), qui va définir le type de retour que l'on attend. Une sélection, un nombre etc.
- Un objet sur lequel porte la requête : les fichiers, les bulletins etc.
- Des critères qui permettent de limiter la recherche.

La grammaire doit répondre à des exemples de la forme ci-dessus, mais ils n'arriveront pas tels quels pour être interprétés, ils sont d'abord normalisés et corrigés.

3.2) La grammaire

Nous avons implémenté deux types d'expression sur le type de retour : "le nombre de …" et "retourner les ...".

Pour identifier une requête sur le nombre, nous nous sommes basés sur le fait qu'une requête doit commencer par "je veux savoir combien de … sont ..." ou bien "je veux savoir le nombre de … "

Le premier élément de la structure de notre grammaire correspond à un mot correspondant à une sélection. Les mots (sous forme lemmatisée) que nous avons retenus pour correspondre à cette demande sont "selection", "veux", "affich", "savoir", "voulais", "recherch", "donn", "souhait" et "list". Ainsi les phrases commencent par "je veux ... | affiche …| sélectionne moi …".

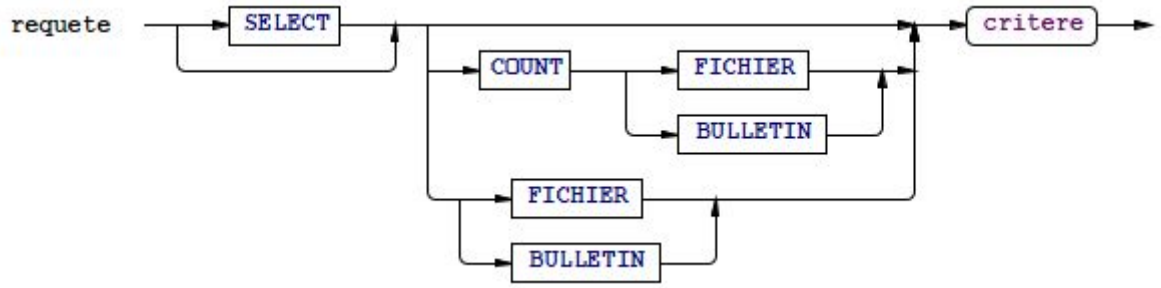
Cet élément peut être ommis, afin de perdre d'accepter des formes nominales (directement ce que l'on souhaite sans mettre "je voudrai obtenir les"), et aussi d'accepter les phrases commençant par "Combien" par exemple.

Ensuite, nous identifions ce que l'utilisateur veut avoir comme retour, ce qui correspond aussi au sujet sur lequel se porte la requête bien souvent. Nous avons décidés d'implémenter la recherche sur les articles ou sur les bulletins. Pour les identifier, nous repérons les lemmes "articles" et "bulletin".

Pour illustrer la grammaire jusqu'ici, nous allons par exemple commencer une requête par "je veux les articles …", sachant que les articles et pronoms sont éliminés avant de rentrer dans l'interprétation de la grammaire.

En se basant sur les deux sujets de requête ci-dessus, nous implémentons une notion de quantité, pour répondre aux questions sur le nombre d'articles ou de bulletins qui vérifient X.

Pour résumer, le début de la structure suit l'arbre ci-dessous :



avec critère, une gamme de mots qui permettent de restreindre la portée de la recherche.

Nous pouvons dors et déjà trouver des limites à cette grammaire. Tout d'abord sa portée ne prends pas en compte les recherches sur les auteurs par exemple ou pour combiner les retours, comme par exemple retourner un article et sa rubrique. Ensuite, les opérations arithmétiques effectuées ne prennent pas en compte les moyennes ou l'ordinalité, comme par exemple retourner le plus gros X ou le plus ancien Y etc.

La seconde grosse partie de notre grammaire consiste à identifier les paramètres qui vont permettre de contraindre la requête. Cela correspond à la case critère présentée ci-haut. Nous avons décidés d'implémenter 4 types de critères : les critères sur les mots contenus par le fichier, sur le numéro de bulletin, sur la rubrique de l'article et sur la date.

Pour le "mot", nous considérons qu'une recherche doit commencer par l'un des lemmes suivants : "mot", "contenir", "parle", "sujet", "contenant", "évoqu" et "traite". De plus nous faisons l'hypothèse que l'utilisateur ne cherche que des mots "seuls", par exemple "événement Paris" ne sera pas reconnu, mais "événement et Paris" le sera, comme une recherche indépendante sur les deux mots événement et Paris. Pour des limitations liées à la forme des tables de la base de données, nous avons représenté le et joignant deux mots comme étant un ou non exclusif, ce qui augmente le taux de rappel pour les requêtes avec un "et" pur et celles où l'on attend un "ou" exclusif (plus difficile à déceler en Français de toute façon).

Pour la requête sur les numéros, nous estimons que le mot clé "numéro" permet d'identifier pour sûr que la suite va bien parler du numéro de bulletin. Nous récupérons ensuite un mot numérique uniquement.

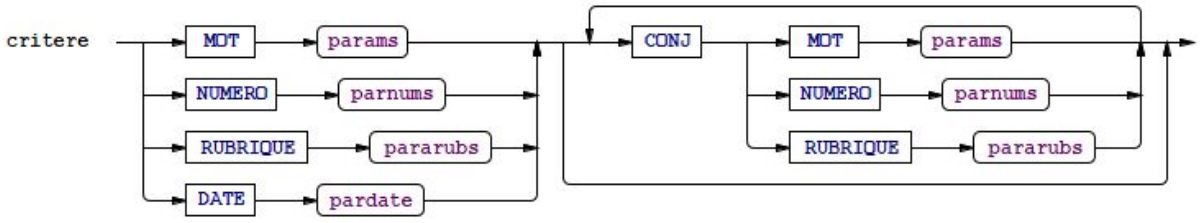
Ensuite, pour la rubrique, nous commençons une requête par le mot rubrique. Après l'étude des différentes rubriques présentes dans le corpus, nous avons remarqué que plusieurs rubriques sont composées de plusieurs mots, comme par exemple la rubrique "actualités innovations". Pour palier ce problème, nous avons organisé la structure telle qu'elle interprète les mots à la suite de "rubrique" comme étant un morceau du nom de la rubrique, et le fait jusqu'à trouver un mot de coordination ("et" ou "ou") ou bien la fin de la phrase. Comme la phrase parvient sous forme lemmatisée, certains mots de liaisons qui font parti du nom de la rubrique peuvent avoir disparu et les mots modifiés, nous avons donc retranscrit cette approximation pour l'interrogation de la base de données par l'utilisation de

“%” en amont et aval de chacun des mots et du *like* de SQL pour pouvoir récupérer les rubriques contenant chacun des mots en son sein. Le taux de rappel peut parfois être un peu plus grand que ce qu’il aurait du être, mais globalement il est très satisfaisant.

Enfin le dernier critère est la date. Beaucoup de dates sont et de formats de dates sont possible, nous avons donc fait des choix. Tout d’abord nous avons implémenté la date sous les formats “2 décembre 2011” et “2 12 2011”. Ensuite, au niveau de la flexibilité, nous avons implémenté les formats “jour mois année”, “mois année”, “année”. Enfin nous avon implémenté la recherche par date exacte. Les limites de cette recherches sont liées au manque de flexibilité du format, qui ne fait pas toutes les combinaisons possibles (juste jour et mois, ou juste jour par exemple) ainsi que la non prise en compte du format souvent utilisé 02/12/2011.

Une dernière limite, cette fois plus limitante pour permettre d’avoir une syntaxe SQL valide, est la place de la date dans la phrase. En effet, la date étant stockée dans une table différente, il faut spécifier la table au début de la requête et non en plein milieu. Pour permettre de pouvoir utiliser la date, celle-ci ne peut être utilisée que si elle est présente en début de requête, c’est à dire comme premier critère, ce qui est contre-intuitif comparé à nos habitudes de recherche, car bien souvent nous commençons par rechercher un sujet puis on l’affine en rajoutant des critère à la fin.

La structure permet ensuite de combiner tous ces critères entre eux grâce à des coordinations les reliant (“et” et “ou”). Les critères sont ajoutés de façon additive, c’est à dire que nous ne pouvons pas faire une recherche qui exclut un critère.



3.3) Lier la morphologie et la grammaire

Afin de lier la morphologie à la grammaire, nous avons tout d’abord commencé par identifier les mots principaux de notre structure et vérifier comment ils se comportent dans la partie morphologie. Nous nous sommes penchés sur les connecteurs que nous utilisons : “et” et “ou”. Ces mots étaient dans la stoplist, c’est à dire qu’ils auraient été supprimés car ils n’avait pas de sens pour le corpus, alors que pour la requête ils en apportent beaucoup. Nous les avons donc retirés de cette liste pour qu’ils restent. Aussi nous avons par exemple rajouté des mots tels que “liste”, car demander “la liste des articles” ou “les articles” est équivalent pour nous et le mot liste n’est pas utile donc peut être supprimé par la stoplist.

Nous avons ensuite regardé comment se comportaient les mots que nous avons gardés pour exprimer un critère. Nous avons vérifié leur lemmatisation et les avons modifiés dans plusieurs cas : le lemme associé est lié à un mot dont le sens ne colle pas avec celui que nous voulons pour notre structure, ou bien l’ajouter si nous souhaitons remplacer le mot

par un autre par choix. Par exemple, un mot comme Décembre sera remplacé par 12, car dans les tables SQL, le mois est stocké comme un nombre à deux caractères, il est donc plus facile de remplacer décembre par son nombre associé à l'étape de lemmatisation que de modifier la grammaire à l'étape d'après.

Une fois les lemmes définis pour chacun des mots intéressants de notre grammaire, nous les écrivons dans la grammaire. La limite que nous pouvons trouver à cette méthode est que certains mots que l'on représente peuvent à la fois servir comme mot de structure (le "sens habituel") mais aussi il peut être une variable que nous souhaitons rechercher dans le document. Dans ce cas, Antlr n'arrivera pas à identifier, car il ne cherche à unifier qu'avec le premier type qu'il rencontre, ce qui provoquera une erreur car la grammaire ne permet pas d'avoir ce type à un certain endroit de l'arbre par exemple.

Bien que notre grammaire soit assez large sur la morphologie de la phrase, elle ne reste pas suffisante pour interpréter toutes les formulations possibles du français. En effet certaines formulations ont un sens strictement équivalent mais la structure est complètement différente, et introduire les deux à la fois augmenterait fortement la forme de la grammaire avec le risque d'accepter des phrases qui elles seraient sans sens en français. Par exemple la phrase "Combien d'articles parlant de Robotique ont pour Rubrique « Focus » ?" a strictement le même sens que la phrase "Combien d'articles parlent de Robotique et sont parus dans la Rubrique « Focus » ?", et pourtant l'une des deux utilise l'auxiliaire "avoir" pour établir une conjonction entre les critères et l'autre utilise un "et".

4. La génération de l'arbre SQL

Après avoir entré une requête, nous normalisons la requête avec le programme précédent. Dans un premier temps, nous vérifions si le mot est présent dans la *stoplist* et si oui le remplaçons par une chaîne vide, sinon nous utilisons la méthode de correction orthographique pour rechercher le lemme correspondant. Si un mot n'existe pas dans la *stoplist* et que nous ne pouvons pas trouver son lemme, nous le retournons lui-même. De cette façon, nous obtenons une requête normalisée.

Cette requête normalisée sera utilisée comme entrée dans la partie grammaire. Grâce aux fichiers JAVA générés par Antlr, nous avons pu générer l'arbre SQL.

Enfin, nous effectuons le traitement sur l'arborescence SQL, par exemple en supprimant les parenthèses générées par la structure arborescente d'Antlr. Nous avons reçu l'instruction SQL qui peut être appliquée avec l'API SQL de JAVA.

Nous avons essayés différentes requêtes avec notre programme et affiché les résultats intermédiaires :

```
Requête : Je veux les articles qui parlent philosophie.
requêt normalisé :  veux article parle philosoph.
Arbre SQL :    (  select distinct  tt.fichier  from public.titretexte tt    (  where    (  tt.mot    ( = 'philosoph'  ) ) ) )
SQL :  select distinct tt.fichier from public.titretexte tt where tt.mot = 'philosoph' ;
fichier
70161.htm
74169.htm
67938.htm
72932.htm
72113.htm
```

```
Requête :Donnez-moi des articles parus au 21 juin 2011.

requêt normalisé :  donn article paru 21 06 2011.

Arbre SQL :    (  select distinct  tt.fichier  from public.titretexte tt    (  ,public.date d
where    (  d.jour    ( = '21'  ) AND d.mois    ( = '06'  ) AND d.annee    ( = '2011'  ) AND
d.fichier=tt.fichier  ) ) )

SQL :  select distinct tt.fichier from public.titretexte tt ,public.date d where d.jour = '21' AND
d.mois = '06' AND d.annee = '2011' AND d.fichier=tt.fichier ;

fichier
67071.htm
67068.htm


Requête : Je veux les articles de la rubrique Focus et qui parlent de philosophie.

requêt normalisé :  veux article rubrique focus et parlent philosoph.

Arbre SQL :    (  select distinct  tt.fichier  from public.titretexte tt    (  where
(  tt.rubrique    ( like '%focus%'  ) ) AND    (  tt.mot    ( = 'philosoph'  ) ) ) )

SQL :  select distinct tt.fichier from public.titretexte tt where tt.rubrique like '%focus%' AND
tt.mot = 'philosoph' ;

fichier
70161.htm
74169.htm
67938.htm
72932.htm
72113.htm


Requête : Affiche l'article dont le numéro est 290

requêt normalisé :  affich article numero 290.

Arbre SQL :    (  select distinct  tt.fichier  from public.titretexte tt    (  where
(  tt.numero    ( = '290'  ) ) )

SQL :  select distinct tt.fichier from public.titretexte tt where tt.numero = '290' ;

fichier
75794.htm
75789.htm
75790.htm
75793.htm
75788.htm
75797.htm
75796.htm
75792.htm
75791.htm
75795.htm
```