Construire un moteur d'indexation

Jacques Guyot & Gilles Falquet

Université de Genève
Département de systèmes d'information
Centre universitaire d'informatique
24, Général Dufour
CH-1211 Geneva 4
Switzerland

Karim Benzineb

Metaread SA
18 Chemin des Aulx
CH - 1228 Plan-les-Ouates - Geneva
Switzerland
www.metaread.com

RÉSUMÉ. *Nous présentons ici un moteur d'indexation ayant fait l'objet d'un transfert technologique entre l'université et l'industrie. Ce moteur est actuellement intégré dans des applications utilisées par les organisations internationales. Les corpus indexés sont volumineux et multilingues. En partant des spécificités du cahier des charges, nous examinons les choix d'architecture et de technologies effectués pour répondre aux contraintes de performance et de volumétrie. L'utilisation optimale des ressources de mémoire, de calcul et de stockage est discutée. Le séquençage et la parallélisation des processus sont examinés.*

ABSTRACT. *We present here an indexing engine which is covered by a technology transfer agreement between the University and the private sector. This engine is currently included in various applications used by international organizations. The document collections which are indexed are large and multilingual. The particular elements of the technical specifications are the starting point of our analysis; then we look at the design and technology choices made to meet the performance and volume constraints. The optimal use of memory, calculation and storage resources is discussed. The serialization and parallelization of processes are analyzed.*

MOTS-CLÉS : *indexation, document, performance, architecture.*

KEYWORDS: *indexing, document, performance, design.*

1. Introduction

Au cours des cinq dernières années, notre groupe de recherche a effectué plusieurs transferts de technologie dans le cadre des traitements automatiques linguistiques. Le domaine de ces recherches est celui de l'utilisation et de l'exploitation efficaces des documents dans les organisations. Dans le cadre des organisations internationales ou des Etats nations multilingues (comme la Suisse), nous avons développé des prototypes d'outils d'aide à la traduction. Le passage à l'échelle s'est effectué lors du transfert technologique. Les prototypes ont du être adaptés à des contraintes imposées par les problématique du client. Nous avons aussi développé des outils d'aide à la classification des documents et de création de classifications (clustering). Tous ces travaux utilisent un moteur d'indexation appelé VLI (Very Large Indexer) que nous présentons ici. L'intérêt de ce moteur est son architecture qui lui permet de s'adapter à une large variété de besoins de la part des clients en termes de volume des corpus, de performances demandées et de ressources disponibles.

La structure de l'article se décompose ainsi : la section 2 décrit les différents contextes d'utilisation du VLI ; la section 3 reprend sous la forme d'un cahier des charges les contraintes de développement et de mise en œuvre du système ; la section 4 définit les notations, le processus d'indexation et celui d'interrogation ; la section 5 esquisse l'architecture globale du système et décompose le moteur d'indexation en modules ; la section 6 décrit le module de gestion du dictionnaire de termes ; la section 7 est consacrée au module de gestion des index ; la dernière section présente des tests et une comparaison avec le moteur d'indexation Lucene [Hatcher 2004].

2. Le contexte d'utilisation du moteur d'indexation

Les organisations internationales (OMC, UIT, BIT, OMM, OMPI, CPI…) et les Etats nations (Suisse, Communauté européenne…) ont choisi d'aider leurs traducteurs dans leur travail en leur donnant des outils informatisés. Ces outils exploitent les corpus parallèles multilingues de tous les documents qui ont déjà été traduits ainsi que des lexiques multilingues. Ces organisations travaillent avec au minimum trois langues officielles dont pour certaines le chinois, le russe et l'arabe. Les corpus sont parfois composés de plusieurs centaines de milliers de documents. La taille de ces derniers varie de quelques pages à plusieurs centaines. Les outils proposés aux traducteurs sont :

– le *référenceur* qui cherche dans le document à traduire les plus longs passages existant dans un texte déjà traduit;

– le *gestionnaire de terminologie* qui permet d'accéder aux expressions et aux abréviations les plus usitées par l'organisation;

– la *recherche bi-texte* qui permet de rechercher toutes les occurrences d'une expression dans le corpus et les présenter côte à côte avec leurs traductions.

L'objectif des organisations est d'obtenir les traductions les plus cohérentes possibles (on cherche à minimiser les variations de la traduction des expressions) . Cet objectif est rendu difficile par le grand nombre de traducteurs impliqués et par la variété des domaines traités. Le corpus joue un rôle de mémoire essentiel pour l'organisation. L'indexation du corpus doit être au moins quotidienne; de plus, en période de conférence, les besoins de traduction sont accrus. En mode interactif, le traducteur choisit la paire de langues source et cible de sa traduction. Sa requête exécute dans le corpus source une recherche de documents satisfaisant les critères indiqués. Ensuite, le traducteur sélectionne le document le plus adéquat dans son contexte de traduction. Le système charge alors les documents source et cible, se positionne sur la première occurrence de l'expression recherchée et aligne le document cible [Guyot 2005b]. Le référençage des documents est une tâche intense en termes de requêtes car pour chaque mot de texte à référencer, il faut effectuer une requête au moteur de recherche. Ainsi, une page comportant 350 mots nécessite autant de requêtes. Les documents de moins de dix pages sont référencés interactivement; les autres sont soumis en traitement par lot.

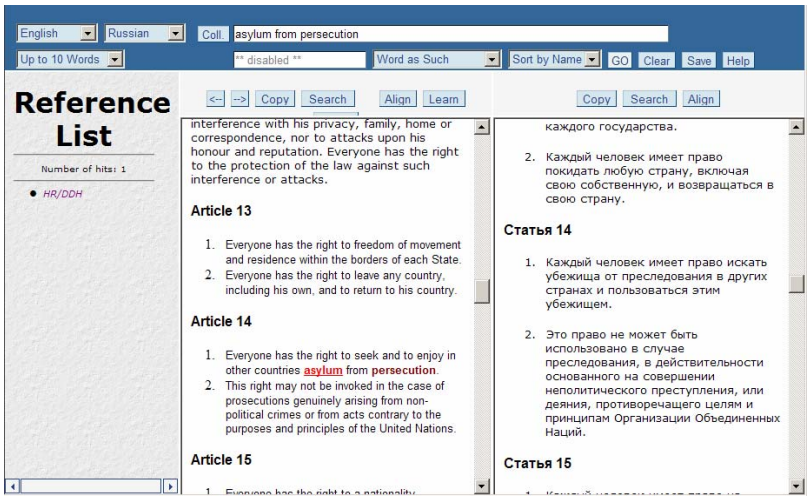

Figure 1. *Exemple de recherche bi-texte (sur le site de MetaRead)*

Un autre contexte d'utilisation du moteur VLI est celui de l'Organisation mondiale de la propriété intellectuelle (OMPI) [Fall *et al.,* 03] où le moteur est utilisé lors de l'entraînement d'un classifieur de brevets. Le but est d'indexer une collection d'entraînement de 8 Go qui comporte 2'000'000 de brevets rédigés en cinq langues: l'anglais, le français, l'allemand, l'espagnol et le russe. Le nombre total de termes est proche de trois millions. La rapidité d'indexation a été décisive dans l'implémentation et le réglage des paramètres du réseau de neurones utilisé par le classifieur. Outre la classification, l'application permet de rechercher en mode

interactif des brevets « proches » de celui que l'on tente de classer. Une fonction de recherche par les plus proches voisins est implémentée sur l'index (une mesure de similarité par le cosinus utilisant les pondérations TFxIDF [Witten 1999][1]).

Un troisième exemple d'utilisation du moteur est son intégration dans un système d'archivage. Les progiciels de gestion intégrés (ERP) de certaines organisations (les banques par exemple) produisent des rapports sur les transactions de leurs clients qui sont archivés quotidiennement. Un gigaoctet par jour est une taille courante pour ce type d'archives. Ces organisations souhaitent pouvoir rechercher des documents dans leurs archives comme s'il s'agissait d'une requête sur Google. Dans ce cas, le moteur doit accepter une volumétrie importante.

Finalement, nous utilisons le moteur pour des recherches académiques portant sur l'utilisation des ontologies multilingues dans le domaine de la recherche d'informations [Guyot 2005a] et dans celui des alignements bi-textes [Guyot 2005b].

La version du moteur d'indexation telle que nous la présentons est le résultat d'une collaboration étroite avec notre partenaire industriel et des contraintes de ses clients. Le moteur a fait l'objet de quatre versions successives. Dans chaque version, nous nous sommes affranchis des limites de taille ou de performance intrinsèques à l'architecture de la version précédente. Le résultat final est une architecture entièrement modulaire où chaque fonction peut recevoir plusieurs implémentations différentes. La configuration et le choix des implémentations sont déterminés par la taille et la nature du corpus du client. Nous présentons ici la dernière version du moteur, que nous allons disséquer pièce par pièce. Nous laisserons de côté le langage d'interrogation et les formules d'évaluation de l'ordonnancement des résultats ou des pondérations IDF, TF. Nous concentrerons notre attention sur les questions posées par la construction du moteur:

– comment gérer des dictionnaires comportant des millions de mots, un registre de documents comportant des dizaines de millions de noms de documents et des vecteurs d'indexation dont le volume peut atteindre des dizaines de gigaoctets ?

– comment gérer le multilinguisme des collections ?

– comment structurer l'index pour favoriser son utilisation et donc répondre au plus grand nombre de requêtes ?

– comment gérer les collections dynamiques et assurer un temps d'indexation constant pour les incréments ?

Dans la section suivante, nous détaillons les contraintes de mise en œuvre. Cette étape est un préalable nécessaire car elle définit un cadre pour les réponses et les solutions aux questions posées ci-dessus.

---

[1] Pour ne pas surcharger la bibliographie, nous considérerons que "Managing gigabytes" (MG) [Witten 1999] est l'ouvrage de référence pour la suite de cet article. Sauf autre indication de notre part, les concepts que nous utilisons sont donc identiques à ceux définis dans MG.

3. Cahier des charges

Les contraintes sont réparties en trois groupes : contraintes du domaine, de génie logiciel et de matériel.

3.1 *Contraintes du domaine*

Les contraintes du domaine d'application du moteur d'indexation sont déterminées par les types d'utilisation décrits dans l'introduction.

Deux types d'indexation sont possibles:

1) Le mode ***statique*** : le corpus est connu au moment de l'indexation et il ne sera plus modifié. Ce mode correspond à celui des collections comme CLEF ou TREC. Dans la pratique, les collections d'entraînement pour la classification sont aussi statiques.

2) Le mode ***dynamique*** : le corpus évolue au cours du temps. Pour ce type, nous distinguons deux modes de mise à jour :

– Le mode ***incrémental*** : après avoir indexé un corpus initial, on ajoute régulièrement des documents au corpus. Ce mode correspond aux situations d'archivage.

– Le mode ***différentiel*** : après avoir indexé un corpus initial, on modifie le corpus. Ce mode correspond aux situations d'indexation où des documents sont supprimés, modifiés ou ajoutés.

L'indexeur doit accepter des corpus multilingues: dans le même corpus doivent pouvoir cohabiter des documents en anglais, français, allemand, espagnol, russe, arabe, chinois, etc.

Le moteur doit être configurable en fonction de la taille des corpus. Un petit corpus (< 100'000 documents) doit pouvoir être traité entièrement en mémoire. Un grand corpus (> 2'000'000 documents) doit être traité sur disque.

La gestion des termes et du registre des documents doit accepter un nombre « illimité » d'entrées (par exemple, 256 millions de termes ou de noms de document)

La vitesse d'indexation doit atteindre 1 gigaoctet par heure dans le cas le plus défavorable (gros corpus et gros dictionnaire) et avec tous les traitements linguistiques (mots vides et lemmatisation), en utilisant une seule unité centrale (de type Intel P4- 3.2Ghz). Elle doit rester constante pour le mode dynamique.

3.2 *Contraintes de génie logiciel*

Le langage Java a été choisi par le partenaire industriel pour ses qualités de modularité, de vérification stricte du typage, de complétude et de robustesse des bibliothèques et d'absence de manipulation directe de la mémoire du programme. Seuls quelques doutes sur la vitesse d'exécution des programmes mettaient un voile

d'incertitude sur le projet (démarré en 2000). Cependant, Sun a rapidement sorti une nouvelle version de la machine virtuelle *HotSpot* [Sun 2001] qui par des améliorations appréciables de la gestion de la mémoire et par une compilation à la volée rendait Java aussi rapide que des langages compilés.

L'un des facteurs essentiels de ce choix tient au fait que Java s'inscrit dans un développement orienté vers l'Internet. Le moteur d'indexation n'est que le centre d'applications dont la mise en œuvre s'intègre dans une architecture Web ou client-serveur. Le moteur d'indexation peut être directement utilisé dans une *servlet* ou en mode serveur avec une interface *RMI* (Remote Method Invocation). L'aspect multilingue intégré à Java avec la norme Unicode a aussi été un élément essentiel de son choix pour gérer le multilinguisme des collections dans des jeux de caractères non latins. L'autre facteur déterminant en faveur de Java est son indépendance par rapport aux systèmes d'exploitation. Java permet de livrer indifféremment aux clients les applications dans un environnement Unix ou Windows. Il a par ailleurs un fort potentiel de pérennité et il est très stable dans la mesure où peu de versions ont été déployées au cours des dix dernières années. Enfin, une large communauté de développeurs s'est créée autour de ce langage. Le choix de notre partenaire était en accord avec notre propre point de vue. Nous sommes convaincus des avantages des langages orientés objet et en 1996, nous avons écrit un livre sur Java [Bonjour *et al.,* 1996].

Au début du projet (en 2000), nous avons recherché des moteurs existant en licence logiciel libre (*open source)*. Deux candidats étaient disponibles: MG[2] [Witten 1999] et SMART[3] [Salton 1971], mais tous deux étaient écrits en C. Ils ne géraient pas le multilinguisme et n'avaient pas d'opérateurs de proximité (NEAR, NEXT, …). A l'époque, nous n'avions rien trouvé en Java. Tout au long du projet, nous avons suivi l'évolution des indexeurs. A notre connaissance, MG et SMART n'ont pas connu de nouvelle version depuis 1999. Ils restent toutefois d'excellents moteurs à but pédagogique ou expérimental. Zettair[4] de l'université de RMIT (Melbourne) semble être la version « moderne » de ces moteurs, il est aussi écrit en C. Actuellement, le projet le plus abouti en Java est celui de Lucene[5] [Hatcher 2004]. Il peut être considéré à la fois comme une boîte à outils que l'on exploite selon les applications et comme un format de fichier pour l'échange d'index. Il gère les opérateurs de proximité et le multilinguisme. Lors des tests, nous le prendrons comme référence.

3.3 *Contraintes de matériel*

L'idée clé concernant le matériel est que le client achète une simple fonctionnalité à travers le moteur d'indexation; il ne veut donc pas trop complexifier son architecture informatique. Dès lors, la consigne était d'utiliser au maximum les

[2] Site MG: http://www.cs.mu.oz.au/mg/
[3] Site Smart: ftp://ftp.cs.cornell.edu/pub/smart/
[4] Site Zettair: http://www.seg.rmit.edu.au/zettair/
[5] Site Lucene: http://lucene.apache.org/java/docs/

ressources matérielles existantes avant de passer à une autre architecture. On entre ici dans une logique économique. Les solutions techniques doivent être économiquement saines.

L'approche de base est donc un serveur unique dédié à l'indexation et à la recherche. Une telle architecture doit pouvoir servir une organisation ayant un corpus de plusieurs gigaoctets et une centaine de traducteurs. Au niveau suivant, on spécialise les serveurs avec des tâches d'indexation et de recherche. On peut même envisager de répliquer les serveurs de recherche. Une telle architecture peut faire face à des corpus de plusieurs dizaines de gigaoctets et servir un millier de traducteurs (ce qui correspond aux besoins du Parlement européen, par exemple). Enfin, il semblait raisonnable de prévoir la possibilité d'une indexation parallèle pour faire face aux besoins à venir.

Pour mieux cerner les difficultés inhérentes à l'indexation, nous allons décrire en détail le processus d'indexation et d'interrogation.

4. Le processus d'indexation et d'interrogation

Avant tout, définissons quelques concepts avec lesquels nous allons travailler.

4.1 Notations

Un corpus $\mathcal{C}$ est un ensemble de documents $\{d_1, d_2, d_N\}$.

$\boldsymbol{N}$ est le nombre de documents du corpus.

Un document $\boldsymbol{d_i}$ est une séquence de termes $\boldsymbol{t}$, notée $<t_1,t_2, \ldots t_{li}>$.

$\boldsymbol{l_i}$ est la longueur du document d_i

Le dictionnaire $\mathcal{T}$ est l'ensemble des termes distincts du corpus $\mathcal{C}$.

$\boldsymbol{idx_j} = (o_1,o_2, \ldots o_N)$ est l'index du terme t_j pour le corpus $\mathcal{C}$ où o_k définit le nombre d'occurrences du terme t_j dans le document d_k.

$\boldsymbol{pos_{jk}} = (p_1,p_2, \ldots p_{ok})$ est le vecteur des positions du terme t_j dans le document d_k où p_m définit la position de la m^{ième} occurrence du terme t_j dans le document d_k

La spécification du moteur d'indexation peut simplement s'énoncer comme le calcul de idx_j et de pos_{jk} pour tous les termes t_j du dictionnaire $\mathcal{T}$ du corpus $\mathcal{C}$.

La réponse à une requête cherchant les documents qui contiennent le terme t_Q est directement obtenue avec idx_Q, l'index du terme t_Q . Si $o_k \neq 0$, cela indique la présence du terme recherché dans le document d_k. On parle de requêtes booléennes (AND, OR et NOT). Le vecteur des positions[6] pos_{jk} est nécessaire dans le cas où la

[6] Dans MG [Witten 1999], $\boldsymbol{idx_j}$ et $\boldsymbol{pos_{jk}}$ correspondent respectivement à des index de niveau document et mot. MG ne détaille pratiquement pas les méthodes concernant le niveau mot. Altavista (http://www.altavista.com/) implémente l'opérateur NEAR. Le moteur de recherche

requête stipule un rapport de distance ou de précédence entre deux termes recherchés (opérateurs NEAR et NEXT).

Nous distinguons deux modes d'utilisation du moteur. Dans l'un on construit l'index à partir des documents, dans l'autre on utilise l'index pour rechercher des documents à partir d'une requête soumise par l'utilisateur.

4.2. *Mode d'indexation*

Le processus suivant décrit l'utilisation du moteur pour ajouter des documents à un index existant :

Pour chaque document à indexer :

– enregistrer les propriétés du document dans le registre des documents

– ouvrir le document et chercher les termes qui le composent.

– Pour chaque terme à indexer :

– chercher l'identifiant du terme dans le dictionnaire (ajouter s'il est nouveau)

– ajouter la référence au document pour ce terme

– éventuellement ajouter sa position dans le document.

En examinant le processus, on voit que les difficultés principales sont les suivantes:

1) Chercher les termes du document. Nous ne traiterons pas de ce point mais il faut souligner que l'indexeur passe beaucoup de temps sur cette tâche, car il doit lire le document caractère par caractère et appliquer des règles de formation des termes. Ces règles dépendent des clients et des corpus.

2) Après avoir trouvé un terme, trouver l'identifiant qu'il porte pour pouvoir mettre à jour son index. Les dictionnaires ont plusieurs millions d'entrées et il faut pouvoir rapidement associer à un terme son identifiant. Finalement, on a un triplet (t_j, d_k, p).

3) Mettre à jour l'index. $\boldsymbol{idx_j} = (o_1, o_2, \ldots o_N)$ où o_k est le nombre d'occurrences du terme t_j dans le document d_k. On voit très vite que la plupart des o_k seront égaux à zéro. Il est donc plus simple de stocker l'ensemble de couples $((ki_1, o_{i1}), (k_2, o_{i2}), \ldots, (k_n, o_{in}))$dont les o_k sont différents de zéro. Pour $\boldsymbol{pos_{jk}}$, on n'échappe pas à la contrainte de mémoriser toutes les occurrences.

Pour un gigaoctet de texte (sans balisage), on peut estimer que l'on a 125 millions de termes. On peut retirer 20% pour les mots vides. Il reste donc 100 millions de termes que l'on doit traiter selon la procédure des points 1 à 3. Pour les traiter en une heure, il faut donc une vitesse minimum d'indexation de 28 Ktermes/sec.

---

de Lycos (http://www.lycos.fr/help/help_boolean.html) implémente les opérateurs de proximité NEAR, FAR, ADJ, …

L'autre problème posé par l'indexation est le stockage des index de termes. Ils ont des longueurs variables et ils se rallongent dynamiquement pendant le processus d'indexation. De plus, la loi de Zipf [Zipf 1949] met en évidence un rapport entre la fréquence d'un terme et son classement dans le dictionnaire en vertu duquel nous aurons une multitude d'index très courts (une ou deux occurrences) et un petit nombre de termes ayant une multitude d'occurrences, sans être vraiment des mots vides (*"Stop Words"*).

4.3. *Mode d'interrogation*

Le processus suivant décrit l'utilisation du moteur pour effectuer des recherches sur un index existant :

Pour une Requête :

– Rechercher les identifiants des termes de la requête dans le dictionnaire

– Rechercher les index des termes

– Evaluer des opérateurs de la requête (AND, OR, NEAR, NEXT, ...) sur les index

– Filtrer, ordonnancer le résultat

– Rechercher les noms des documents du résultat.

En examinant le processus, on voit que les difficultés principales consistent à récupérer les index associés aux termes de la requête et à évaluer les opérateurs de la requête. Nous avons choisi la stratégie en vertu de laquelle les index stockés doivent être pratiquement prêts pour être utilisés comme opérande. Les opérateurs binaires comme AND et OR peuvent être directement calculés sur les index. Nous avons choisi d'implémenter ces opérations selon la méthode du tri/fusion[7]. La phase de tri peut être éliminée si les identifiants des documents sont déjà triés dans l'index. Lors de la phase d'indexation, il faut donc créer des vecteurs dont les identifiants de documents sont déjà ordonnés. La recherche des noms des documents peut s'avérer assez longue si chaque nom nécessite un accès au disque.

5. Architecture logicielle

Nous allons examiner ci-après comment nous avons structuré notre moteur d'indexation.

5.1. *Découpage des services*

Nous avons décidé de découper le moteur d'indexation en différents services :

[7] Dans MG [Witten 1999], l'implémentation naïve du tri/fusion est discutée et une alternative par recherche dichotomique est décrite lorsque les listes à fusionner sont dissemblables par leur longueur. En poursuivant cette idée, nous avons implémenté une fusion dichotomique récursive dont on trouve une description théorique dans [Baeza 2004].

– Gestionnaire du dictionnaire de termes qui permet d'associer un terme à un identifiant et d'en assurer la persistance, de rechercher le terme associé à un identifiant et de rechercher l'identifiant associé à un terme.

– Gestionnaire du répertoire de documents qui permet d'associer un nom de document à un identifiant et d'en assurer la persistance, de rechercher le nom du document associé à un identifiant, et de rechercher l'identifiant associé à un nom de document. Ce gestionnaire est aussi chargé de mémoriser des propriétés pour chaque document telles que la date, la longueur, la langue, l'encodage, etc.

– Gestionnaire des index qui permet d'assurer toutes les fonctions associées à la maintenance des index des documents et des positions.

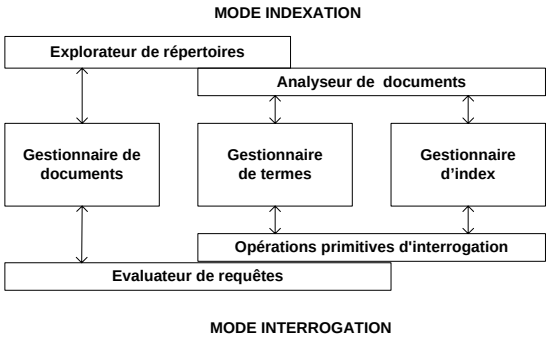

Figure 2. *Les trois gestionnaires de base dans les deux modes principaux d'utilisation*.

Chacun de ces gestionnaires est indépendant et ignore l'existence et l'implémentation des deux autres. Ces trois gestionnaires sont les composants de base dans chaque mode d'indexation et d'interrogation (ils le sont aussi dans les autres applications comme le classifieur, par exemple). Dans le mode d'indexation, un explorateur de répertoires est associé à un analyseur de document pour implémenter le processus d'indexation. Dans le mode d'interrogation, un évaluateur de requête est associé aux opérations sur les index pour implémenter le processus d'interrogation. La spécification des gestionnaires intervient par une déclaration des interfaces qui spécifie tous les services rendus par les gestionnaires. A chaque interface, il faut associer au moins une implémentation. Nous avons systématiquement implémenté plusieurs stratégies pour s'adapter à des tailles différentes: FAST (rapide), BIG (grand) et XXL (très grand). De plus, à chaque service est associée une implémentation en mode serveur (utilisant des appels RMI [Grosso 2001]), ce qui permet de déployer le moteur sur plusieurs serveurs.

5.2. *Configuration du moteur*

La définition du moteur d'indexation se fait dans un fichier où le configureur va indiquer les stratégies choisies ainsi que leurs paramètres. Dans ce fichier, on trouve aussi la spécification des répertoires choisis pour mémoriser les différentes

structures ainsi que les paramétrages des caches. Le fichier qui suit est un exemple de configuration:

```
// les stratégies
DOC_MAXBIT               22  // 4 Millions
TERM_MAXBIT              21  // 2 Millions
TERM_IMPLEMENTATION      implementationMode.BIG
DOC_IMPLEMENTATION       implementationMode.FAST
IDX_IMPLEMENTATION       implementationMode.BIG
...
// les répertoires
DOC_ROOT                 "c:/JG/idx/data/docdico"
TERM_ROOT                "d:/JG/idx/data/worddico"
...
// paramètres de fonctionnement
CACHE_IMPLEMENTATION     implementationMode.FAST
KEEP_IN_CACHE            80
INDEXING_CACHE_SIZE      256*MEGA
TERM_CACHE_COUNT 256*KILO
...
```

Dans les deux sections suivantes, nous allons examiner ces différentes implémentations pour le dictionnaire de termes et le gestionnaire d'index.

6. *Le gestionnaire de termes*

Nous décrirons d'abord l'algorithme du gestionnaire de termes, puis nous traiterons des différentes implémentations.

6.1. *L'algorithme de "hashage"*

Nous avons choisi la méthode du hashage qui permet un accès en O(1) opération (soit un seul accès si les collisions sont rares). Les B-arbres nécessitent O(ln(n)) opérations (si il est balancé). Nous gérons une table de hashage pour associer à un terme son identifiant. Les identifiants attribués aux termes sont des nombres consécutifs. En associant directement la valeur de hashage à l'identifiant, nous avons enregistré une très nette dégradation des performances due au fait que les identifiants étaient dispersés. En attribuant consécutivement les identifiants, nous regroupons les termes, ce qui favorise la réutilisation des informations dans les caches.

Pour réduire au maximum la taille de la table de hashage, nous ne stockons pas les termes à l'intérieur de celle-ci mais nous stockons une signature du terme que nous obtenons par une autre fonction de hashage. Une double collision des deux fonctions de hashage pour deux termes différents reste possible mais elle est si peu probable que le risque peut être envisagé. De plus, il n'y pas de perte d'information mais une confusion entre ces deux termes. Dans la version 32 bits, chaque entrée de la table occupe 8 octets.

Les chaînes de caractères associées aux termes sont stockées séquentiellement à intervalle fixe. L'identifiant du terme permet de retrouver par calcul l'emplacement de sa chaîne. Dans les cas extrêmes, on peut se permettre de ne pas stocker la chaîne de caractères du terme! En effet, la recherche du terme à partir de son identifiant n'est pas nécessaire au processus d'interrogation ou d'indexation. Elle est utile lors de travaux statistiques sur le dictionnaire.

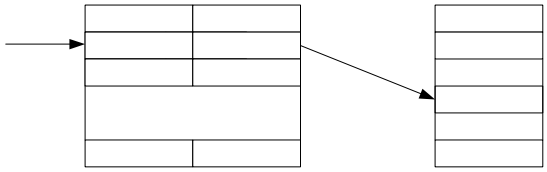

Figure 3. *Structures associées à la gestion du dictionnaire de termes.*

Nous utilisons une recherche linéaire pour gérer les collisions. Le déplacement linéaire permet une meilleure réutilisation des caches[8] par rapport à un déplacement quadratique. Le facteur de charge de la table est toujours inférieur à 0.5.

En résumé, la structure de hashage implémentée favorise une table compacte avec un accès aléatoire déterminé par la fonction de hashage. La gestion des collisions par une méthode linéaire assure une proximité au premier accès.

6.2. *L'implémentation de l'algorithme avec différentes stratégies*

Nous avons implémenté pour chaque stratégie le même algorithme de gestion de la table de hashage décrit ci-dessus; seule la façon d'accéder aux éléments de la table est différente. Nous avons les stratégies suivantes :

– **FAST** (rapide) est une stratégie implémentée en mémoire centrale ; les éléments de la table résident avec le programme en **mémoire**. Cette stratégie est la plus rapide, mais c'est celle qui déborde le plus vite: à partir d'une certaine taille[9], le programme commence à paginer avec la mémoire virtuelle, ce qui implique une grave chute des performances.

– **BIG** (grand) est une stratégie implémentée avec les MAP[10] i/o [Hitchen 2002]; les éléments de la table résident sur les **disques**, mais les sections des fichiers utilisées sont associées à des plages mémoires. Le programme travaille alors directement sur les plages mémoires sans se préoccuper des opérations sur les fichiers. Par rapport à un fichier en accès aléatoire, cette méthode permet d'avoir simultanément plusieurs plages d'accès. La stratégie BIG est moins rapide que la précédente mais elle accepte des tailles de table bien plus importantes. Cependant, elle bute aussi sur la limite de la mémoire réelle. Les performances se dégradent alors.

---

8 Les processeurs actuels ont plusieurs niveaux de cache L1, L2. La performance des algorithmes doit tenir compte de ces derniers. Ce qui est contre-intuitif, c'est que dans certain cas, il vaut mieux calculer que charger des informations qui ne sont pas disponibles dans les caches. Les travaux de D. Baskins sur les Judy Arrays sont un exemple de ce type de raisonnement. (http://judy.sourceforge.net/). Ces arguments sont aussi applicables aux structures stockées sur disque qui doivent être amenées en mémoire.

9 Les valeurs limites sont dépendantes de la configuration matérielle qui doit être calibrée, voir plus loin tableau 1 et figure 5.

10 Guide sur l'API des MAP i/o: http://java.sun.com/j2se/1.5.0/docs/guide/nio/index.html

– **XXL** (très grand) est une stratégie implémentée avec les entrées-sorties à accès direct (direct i/o) ; les éléments de la table résident sur les disques et les accès se font avec des fichiers en **accès aléatoire**. Le programme se positionne, lit ou écrit (et recommence). Les systèmes d'exploitation (Windows et Unix) utilisent la mémoire non allouée aux processus comme cache pour de telles activités. Les performances ne sont donc pas si mauvaises car jusqu'à une certaine taille, il existe une probabilité non négligeable que l'accès ne doive pas recourir au disque mais trouve son information dans le cache géré par le système d'exploitation.

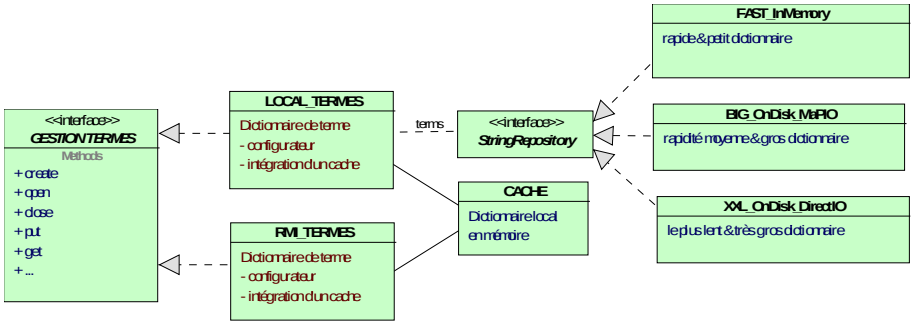

Figure 4. *L'interface GESTION_TERMES a deux implémentations, une en local et l'autre en RMI. Chacune utilise un cache d'accélération en mémoire. La stratégie LOCAL_TERMES peut recevoir trois implémentations FAST, BIG ou XXL. Cette dernière est déterminée par la configuration.*

Dans le tableau 1, nous reportons les mesures effectuées avec les différentes implémentation pour différentes tailles de dictionnaires. Un groupe reporte les mesures pour les écritures, l'autre pour les lectures. Les performances ne sont pas très bonnes pour la table de 2 millions d'entrées, ce qui s'explique par le nombre important de collisions. En effet, d'une part nous avons stocké dans la table les représentations des nombres de 0 à 999'999 sous forme de chaînes de caractères et les valeurs de hashage de ces termes ne sont pas aussi bien distribuées que des termes linguistiques. D'autre part, le facteur de charge est de 0,5. Dans le cas des écritures, pour chaque implémentation, les temps s'améliorent avec la taille croissante du dictionnaire car les collisions diminuent. Cependant, chacune atteint une limite où la mémoire réelle disponible n'est plus suffisante; c'est alors que les performances se dégradent.

En reportant les mesures dans un graphique, on voit émerger une limite vitesse/taille. Augmenter la mémoire réelle de l'ordinateur va faire pivoter la limite par rapport à son maximum. L'implémentation FAST en profitera le plus. L'autre moyen de modifier cette limite est de modifier le temps moyen d'accès au disque. On baissera ainsi le point maximum et on fera glisser toutes les valeurs vers les bas. Il existe deux moyens pour faire baisser cette valeur. La première consiste à utiliser des disques plus rapides, la vitesse de rotation étant directement liée au temps

moyen d'accès. On a typiquement[11] les couples suivants: 7'200 t/min - 8.5ms, 10'000t/min -4.7ms, 15'000t/min – 3.6ms. Une autre solution consiste à multiplier les disques. En utilisant quatre disques, il est possible de répartir sur chacun l'ensemble des informations du dictionnaire. La surface utilisée sera alors d'un quart; les mouvements moyen de la tête du disque seront donc diminués en amplitude, ce qui limitera le temps moyen d'accès au disque. Utiliser plus de disques de manière partielle est plus rentable économiquement que d'utiliser des disques plus rapides. Le mieux est bien sûr l'utilisation conjointe des deux.

	Nombres d'entrées du dictionnaire en million								
	2	4	8	16	32	64	128	256	8192
ECRITURE (temps en secondes pour écrire un million de termes dans le dictionnaire)									
FAST	1,7	0,84	0,54	0,56	4,9				
BIG	17	12,2	12,1	12	12,2	14	35		
XXL	524	87	71	63	59	61	61	68	8500[12]
LECTURE (temps en secondes pour lire un million de termes dans le dictionnaire)									
FAST (Read)	1,1	0,42	0,34	0,28	3,7				
BIG (Read)	2,1	0,9	0,8	0,8	0,8	0,9	0,9		
XXL (Read)	248	31	22,6	19	17	17	11	11	

Tableau 1. *Test des différentes implémentations du gestionnaire de termes en lecture et en écriture (P4M 1.8 Ghz +1Gb RAM +1 disque)*

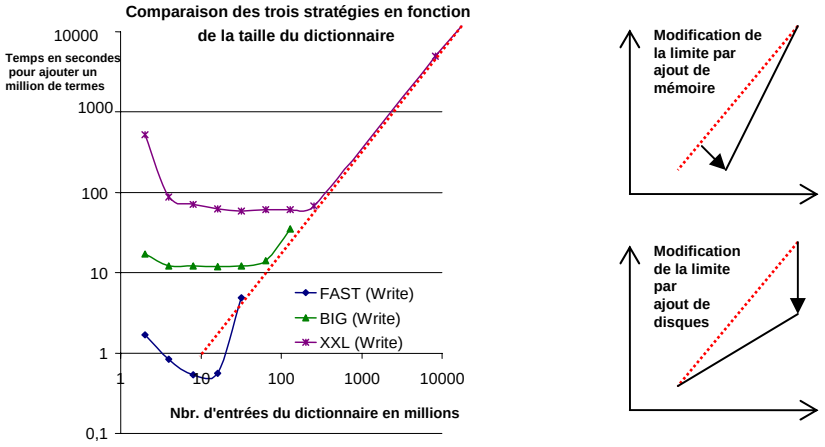

Figure 5. *A gauche les mesures de performance des stratégies en fonction de la taille des dictionnaires; à droite l'incidence de la modification du matériel sur la limite vitesse/taille. En ajoutant de la mémoire ou des disques, on déplace la limite.*

[11] Voir site des constructeurs Hitachi, IBM, Maxtor, etc.

[12] Ce point est calculé théoriquement : pour écrire un terme dans le dictionnaire, il faut un accès disque. Le disque utilisé pouvait contenir 8 GTermes et chaque accès prenait en moyenne 8.5ms.

6.3. *Combien de termes ?*

Dans les tests effectués aux fins du présent article, nous avons employé les collections suivantes:

– **DOC JDK + ORA** ; les fichiers HTML des documentations jdk1.5 et Oracle 9.2. Cette collection se présente sous forme de fichiers indépendants situés une hiérarchie de répertoires.

– **CLEF 2005 Multi8**; la collection multilingue de CLEF 2005[13]. Cette collection comporte huit langues et elle est composée d'articles de journaux des années 1994 et 1995.

– CLEF 2005.FR; la collection françaises de CLEF 2005.

– CLEF 2005.EN; la collection anglaise de CLEF 2005.

– ERP; Une collection provenant des rapports d'archives d'un ERP. Cette collection est utilisée pour les tests d'indexation acceptant des nombres comme des termes.

– TECH17 et TECH108; Une collection de descriptions courtes d'objets techniques.

Collection	Volume [Mb]	Nbr. documents	Nbr. termes	Nbr. occurrences en millions	Densité[14] [caract/terme]
DOC JDK + ORA	500	21'000	70'000	17.0	29.4
CLEF 2005.FR	409	180'000	330'000	33.5	12.2
CLEF 2005.EN	523	169'000	279'000	47.0	11.1
ERP avec nombres	7'600	1'115'000	5'100'000	343.0	22.2
CLEF 2005 Multi 8	3'500	1'590'000	4'000'000	306.0	11.4
TECH17	17'200	1'569'000	5'342'000	1'437.0	12.0
TECH108	108'000	10'297'000	24'510'000	10'297.0	10.5

Tableau 2. *Caractéristiques des collections utilisées pour les tests*

En examinant le tableau 2, on constate que le nombre de termes différents ne dépasse pas 25 millions. Il est cependant essentiel de se demander si le gestionnaire de termes est adapté à des collections plus grandes. Pour répondre à cette question, nous avons utilisé deux estimations du nombre de termes $T(N)$ en fonction du nombre d'occurrence N [Kornai 2002] :

$T(N)=K N^{0,5}$ pour une borne inférieure où K est une constante.

et $T(N)=N^{\delta}$ pour une borne supérieure où δ est une constante.

Avec les caractéristiques des collections du tableau 2, nous avons évalué les constantes K et δ. Nous avons aussi ajouté les chiffres concernant Google en

[13] Voir site « clef.isti.cnr.it/2005.html »

[14] La densité exprime le nombre moyen de caractères pour former un terme. La densité des fichiers HTML est assez faible, beaucoup de balises étant utilisées pour la mise en forme.

1999[Brin 1998], [Huang 2000]. Ensuite, nous avons appliqué un facteur de croissance et nous avons calculé les bornes minimum et maximum du nombre de termes estimés.

Dans le tableau 3 ci-dessous, les résultats montrent que 256 millions de termes pour un dictionnaire de termes est amplement suffisant pour couvrir les besoins des clients. Pour la ligne concernant l'ERP, un facteur de 50 correspond à 400 jours d'archivage, avec un ERP produisant 1Go par jour. Le nombre de termes se situe alors dans une fourchette entre 36 et 110 millions de termes. Pour la collection CLEF, un facteur de 50 correspond à 100 ans d'archive pour ces journaux européens. L'extrapolation faite sur Google permet de vérifier qu'avec ses 8 milliards de pages indexées actuellement, la totalité des termes reste gérable en mémoire. Pour la collection TECH17, le facteur (6.35) correspond au rapport avec la collection TECH108, le maximum prédit de 21 millions de termes est dépassé d'environ 20%, en effet la collection TECH108 est composée de nombreux documents obtenu par numérisation, la reconnaissance automatique des caractères a introduit de nombreux « nouveaux termes » dus aux erreurs de reconnaissance.

Collection	T(N) en millions	N en millions	K	δ	Facteur de croissance	$KN^{0,5}$ Min en millions	N^{δ} Max en millions
TECH17	5.3	1'437	140.92	0.73	6.35	13.5	20.8
ERP	5.1	343	275.37	0.79	50	36.1	110.3
CLEF 2005 Multi	4.0	306	228.66	0.78	50	28.3	83.9
Google 99[15]	14.0	10'000	140.00	0.71	80	125.2	320.7

Tableau 3. *Calcul des bornes minimum et maximum du nombre de termes pour différentes collections*.

Il est encore possible d'améliorer l'accès au dictionnaire de termes en lui ajoutant un cache local mémorisant les termes les plus fréquemment recherchés.

6.4. *Implémentation d'un cache local*

Lors de l'indexation, l'apparition des termes suit la loi de Zipf. Cette propriété rend intéressante l'implémentation d'un cache local en mémoire qui effectue le tampon entre l'indexeur et le gestionnaire de termes. Ce cache a une taille limitée: à chaque fois qu'un terme est requis, soit il est déjà dans le cache, soit on recherche dans le gestionnaire de termes. Dès que le cache est plein, nous le vidons complètement. Dans le tableau 4, nous reportons une série d'expériences sur la collection CLEF2005.FR où nous faisons varier la taille du cache. Pour 128k entrées, on a effectué 33.4 millions de requêtes au gestionnaire de termes. 32.6 millions se trouvaient déjà dans le cache, donc la probabilité d'être dans le cache était de 97.3 %. Le cache a été vidé 3 fois.

[15] En 1999, Google avait 100 millions de page HTML avec 14 millions de termes. Pour estimer le nombre d'occurrences, nous avons pris une moyenne de 2'000 caractères par page (égale à la moyenne de TeraTrec) et une densité de 20.

L'efficacité du cache local dépend de sa taille. A partir d'une certaine taille, seul un petit pourcentage des demandes nécessite un accès au gestionnaire de termes. Le corpus indexé était trop petit pour qu'une amélioration soit significative au regard du temps d'indexation.

Le gestionnaire de documents utilise un registre des noms de documents qui est traité avec une implémentation identique à celle du gestionnaire de termes. Des structures supplémentaires sont ajoutées pour gérer les propriétés du document telles que sa date, sa langue, son type, etc.

Taille du cache en K termes	Nbr. de requêtes	Nbr de requêtes trouvées dans le cache	Probabilité d'être dans le cache %	Nbr. de vidages du cache	Temps indexation [sec]
256	33'432'617	32'902'630	98,4%	0	134
128	33'432'617	32'616'374	97,6%	3	131
64	33'432'617	31'633'013	94,6%	22	135
32	33'432'617	29'569'326	88,4%	107	136
16	33'432'617	26'368'406	78,9%	411	136
8	33'432'617	22'404'799	67,0%	1305	134
4	33'432'617	18'333'828	54,8%	3604	130
2	33'432'617	14'685'341	43,9%	8988	141
1	33'432'617	11'693'218	35,0%	20887	142

Tableau 4. *Expérience avec la collection CLEF2005.FR sur la taille du cache*

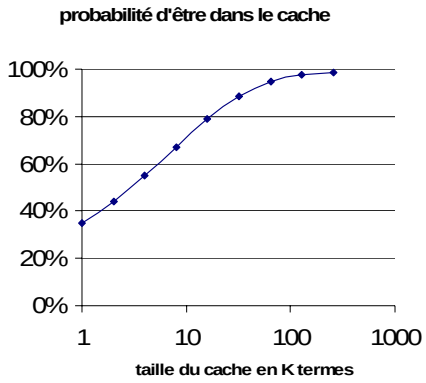

Figure 6. *Efficacité de l'utilisation d'un cache local en fonction de sa taille. Pour un corpus français, avec un cache de 256Ktermes, 98.5% des demandes pourront être satisfaites par le cache local.*

Nous venons de décrire comment le moteur VLI peut s'adapter à la taille des dictionnaires et des corpus de documents. Le problème de l'indexation peut à présent être reformulé de la manière suivante : à chaque nouveau document , nous

associons un identifiant d_k et à chaque terme lu à la position p dans ce document, nous associons un identifiant t_j et une position p. Le gestionnaire d'index est chargé de sauvegarder l'ensemble de ces triplets (t_j, d_k, p).

7. Le gestionnaire d'index

Le gestionnaire reçoit des triplets (t_j, d_k, p). Il travaille donc uniquement avec des nombres, des identifiants de termes, des documents et des positions.

Le gestionnaire d'index est scindé en deux parties. La première est *le cache d'indexation*: elle réside en mémoire et sert de tampon pour toutes les opérations sur l'index. La seconde est *la gestion de la persistance de l'index*: elle sert d'intermédiaire entre le cache et les fichiers sur disque.

Dans MG [Witten 1999], il est montré que l'indexation en mémoire est la plus rapide. Seules des considérations d'espace mémoire empêchent la pleine application de cette stratégie. Nous appliquons une stratégie d'indexation en mémoire combinée avec une mise à jour dynamique de l'index.

7.1. *Le cache d'indexation*

Pour chaque terme indexé, nous recevons un triplet (t_j, d_k, p). Le cache d'indexation doit refléter cette information. Nous avons regroupé les triplets appartenant au même terme en deux vecteurs, l'un pour les documents et les occurrences, l'autre pour les positions. Pour un terme t_j, nous aurons le modèle suivant:

t_j : $(d_1,o_1\ d_2,o_2,\ \ldots d_N,o_N)$ et $(pos_{tj1}, pos_{tj2}\ \ldots, pos_{tjN})$

Les avantages de ce regroupement sont les suivants :

– l'ajout des informations résultant de l'indexation d'un nouveau document se fait toujours par concaténation à la fin du vecteur déjà existant;

– les identifiants de documents sont alloués par ordre croissant. Cela implique que les vecteurs $(d_1,o_1\ d_2,o_2,\ \ldots d_N,o_N)$ sont triés par ordre croissant. Pour les opérateurs d'interrogation implémentés par tri/fusion, le tri ne sera pas nécessaire;

– la gestion séparée des positions permet de considérer celles-ci comme une option lors de l'indexation. En effet, certaines applications ne requièrent pas les positions des termes ou demanderaient trop de place sur disque.

Le cache d'indexation est une table (figure 5) dont les lignes sont les identifiants des termes et les colonnes correspondent à nos deux vecteurs d'indexation. Le stockage d'un nouveau triplet (t_j, d_k, p) est simple et peut se traiter selon deux cas :

1) si t_j est la première occurrence dans d_k, on ajoute au vecteur t_j le nouvel identifiant d_k et la valeur *1* pour o_k, et on ajuste p aux positions de t_j ;

2) si p est une nouvelle occurrence de t_j pour d_k, on incrémente o_k et on ajoute p au vecteur de positions.

Term	doc/occ	position
t_0	$d_1,o_1\ d_5,o_5$	pos_{01}, pos_{05}
t_1	$d_1,o_1\ d_2,o_2,\ \ldots d_N,o_N$	$pos_{11}, pos_{12}\ \ldots, pos_{1N}$
t_2		
...		
t_i	d_1,o_1	pos_{j1}

Figure 5. *Schéma de principe du cache d'indexation.*

Pour gérer l'aspect dynamique des vecteurs, nous allouons des vecteurs avec une réserve. Quand un vecteur devient trop petit pour insérer de nouvelles informations, nous le réallouons dans un vecteur ayant une taille double et nous recopions les informations de l'ancien vecteur.

La simplicité de la mise à jour des index permet de soutenir des débits d'indexation de plusieurs dizaines de milliers de termes à la seconde. Les caches sont rapidement saturés. Par exemple, un cache de 64Mo est rempli en 37 secondes (débit de 126 Ktermes/sec). Deux questions se posent alors: comment vider les caches et que faire de leur contenu ? Ces deux questions ne sont pas indépendantes, comme nous allons le voir.

7.2. *Gestionnaire des segments d'index.*

Nous appelons *segments d'index* les vecteurs (documents/occurrences et positions) d'un terme qui sont dans le cache et doivent être mémorisés sur disque. Pour un terme, soit ce segment est le premier à être mémorisé, soit il doit être associé aux autres segments déjà mémorisés pour ce terme. A la fin de l'indexation, l'index d'un terme est la concaténation de tous les segments. L'index d'un terme est fragmenté si les différents segments qui le composent ne sont pas contigus sur le disque. Ce phénomène de fragmentation est bien connu dans la gestion des fichiers par les systèmes d'exploitation. La performance des interrogations est affectée par la fragmentation, car accéder à la totalité de l'index nécessite d'accéder à chaque fragment et requiert souvent un accès au disque par fragment. Dans le cas du *référenceur* cité dans la section 2, une fragmentation même faible diminue considérablement la performance car chaque mot du texte à référencer donne lieu à une requête, donc à un chargement de l'index de ce mot.

La loi de Zipf prédit que la répartition de la taille des index et de leur nombre par taille suit une loi log/log. Nous aurons donc une multitude d'index ayant une longueur de quelques octets, et quelques index ayant des millions d'octets avec un continuum entre ces deux extrémités.

Nous avons implémenté trois stratégies de gestion des segments d'index :

– **COPIE_CONCATENE** : lorsqu'on ajoute un segment *S'* à l'index existant *S* d'un terme t_j, on recopie S et on lui concatène *S'*. Cette stratégie conserve la contiguïté de l'index, mais elle crée des trous et génère beaucoup de recopies. La gestion des trous est compliquée car les trous peuvent être petits et de taille non uniforme. Maintenir des listes de trous ordonnés par taille en vue de leur

réutilisation est une opération très coûteuse. Nous avons préféré implémenter un processus de compactage qui recopie les index sans laisser de trous. La copie des index suit une progression quadratique ; pour un index de taille *L*, obtenu par *n* segments de taille *l*, nous effectuons environ $n^2l/2$ lectures et écritures. Cette méthode ne tient pas compte des copies lors du processus de compactage.

– **CHAINAGE_ARRIERE** : lorsqu'on ajoute un segment S à un index existant S', on ajoute S à la fin du fichier et on lie S au dernier segment de S'. On reconstituera donc l'index en partant du dernier segment et en suivant les pointeurs. L'avantage de cette stratégie est de ne pas créer de trous. En revanche la fragmentation est maximum. Pour défragmenter, il suffit de compacter les index et de les recopier dans une autre structure.

Aucune de ces deux stratégies n'est suffisante pour aborder une indexation quotidienne de grand volume (1 Go, par exemple). En effet, la stratégie COPIE_CONCATENE sera de plus en plus lente car les copies seront de plus en plus grandes (ainsi que le processus de compactage). La stratégie CHAINAGE_ARRIERE sera relativement constante dans le processus d'indexation; par contre, la performance lors des interrogations se dégradera à cause de la fragmentation. La défragmentation sera elle aussi de plus en plus longue.

Nous avons cherché une stratégie qui conserve la contiguïté des index et qui ne nécessite pas de compactage. Google utilise une telle stratégie, fondée sur les travaux de [Brow 1994] qui utilise les principes d'un gestionnaire de persistance pour des langages objets Mneme [Eliot 1990].

– **ZONE_FIXE** : lorsque l'on ajoute un segment *S* à un index existant *S'*où *S'*est dans une zone de taille *n*, deux cas sont possibles. Si $|S|+|S'| \leq n$ alors on concatène *S* à *S'*; sinon on alloue à l'index une zone libre plus grande, on recopie *S'* et *S* dans cette zone et on libère l'ancienne zone. La gestion des zones se fait par niveau. A chaque niveau, les zones ont toutes la même taille; la taille du niveau supérieur s'obtient en multipliant celle du niveau inférieur par un facteur *P/Q*. Dans notre cas, on travaillera avec *P/Q=2*. En d'autres termes, à chaque niveau la taille des zones double. Avec cette stratégie, la copie des index suit une progression linéaire ; pour un index de taille *L*, obtenu par n segments de taille *l*, nous allons effectuer environ 2*nl* lectures et écritures. Le remplissage des zones est au pire de 50%. En moyenne, nous avons un taux de remplissage des zones de 75%. Les zones libérées sont réallouées en cas de nécessité. Un des inconvénients de cette stratégie est qu'elle est plus lente que les deux précédentes car les ajouts de segments sont dispersés dans la structure de données alors que pour les autres stratégies, on ajoute les segments à la fin du fichier de manière séquentielle.

En examinant le tableau 5 de comparaison des stratégies, on peut constater que la stratégie ZONE_FIXE est un peu moins performante que le chaînage arrière au moment de l'indexation. Cependant, elle élimine totalement la nécessité du compactage ou de la défragmentation – des opérations qui deviennent prohibitives quand les corpus sont très grands. On peut dire que cette stratégie investit un peu

plus que les autres au moment de l'indexation pour garder une situation saine à long terme.

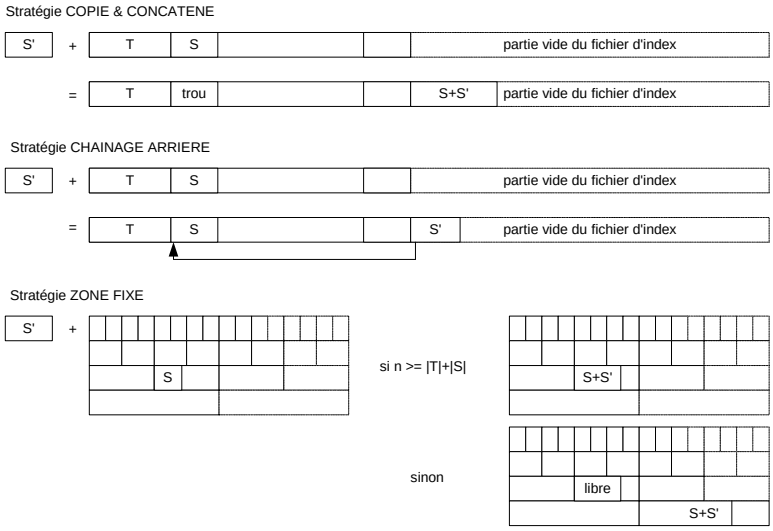

Figure 7. *Résumé des trois stratégies de gestion de la segmentation de l'index. Pour chacune, on montre comment la structure est modifiée quand un segment S' est ajouté à un segment S existant.*

	fragmentation	trou	recopie	Index contigu	remplissage séquentiel
Copie & concatène	non	oui- nécessite un compactage	O(n*n/2)	oui	oui
Chaînage arrière	oui – nécessite une défragmentation	non	non	non	oui
Zone fixe	non	max 25%	O(2*n)	oui	non

Tableau 5. *Résumé des caractéristiques des stratégies de gestion des index.*

Nous appelons *ObjSto*[16] l'implémentation gérant les zones fixes. Dans [Brow 1994], les zones fixes vont de 16, 32, 64 à 8'192. En dessous de 16, un traitement spécial est appliqué, de même que pour les zones au-dessus de 8'192. Nous avons aussi distingué les petits index qui sont très nombreux (plus de 80% des segments

[16] Nous avons étendu la spécification d'*ObjectStore* afin de stocker des vecteurs d'octets. La première fois, on enregistre son objet et un ID est retourné. Cet ID sert à récupérer l'objet, à lui ajouter des octets, à le détruire, le remplacer, etc…

ont une taille inférieur à 32 octets). Par contre, le mécanisme des zones fixes est étendu après 8'192. Pour améliorer les performances d'*ObjSto*, nous l'avons intégré à d'autres mécanismes.

7.3. *Rendre asynchrone le gestionnaire*

Quand on confie une information à ObjSto, celui-ci retourne un identifiant qui permettra de la retrouver ou de la manipuler par la suite. Dans la version d'*ObjSto* pour le processus d'indexation, nous avons imposé une valeur à cet identifiant du vecteur d'index du terme t_j, qui doit être égal à t_j. Cela permet de ne pas attendre que l'*ObjSto* détermine cet identifiant et le retourne. Pendant le processus d'indexation, aucune information n'est donc attendue en retour des gestionnaires d'index. On peut rendre asynchrone le rangement des segments d'index. Après avoir vidé le cache d'indexation, on va continuer à indexer pendant que le gestionnaire d'index range les segments. Nous pouvons ainsi paralléliser deux taches: construire l'index et le ranger.

Nous avons créé une nouvelle implémentation du gestionnaire d'index en lui ajoutant une queue de commandes où sont placés les segments en attente de rangement. Malgré le fonctionnement asynchrone du rangement, la queue est assez rapidement pleine et l'indexeur doit attendre que la queue se vide. On ne range pas assez vite ! Le goulot d'étranglement de notre moteur est maintenant le rangement des segments sur le disque. Pour s'affranchir de cette limite, nous allons paralléliser le gestionnaire afin qu'il utilise plusieurs disques.

7.4. *Paralléliser le gestionnaire*

Si pour ranger 10'000 segments un gestionnaire prend 50 secondes, *n* gestionnaires répartis sur des disques différents prendront *n* fois moins de temps. Nous avons donc implémenté une version parallèle du gestionnaire dans laquelle le nombre de gestionnaires est une puissance de deux. Nous avons expérimenté notre implémentation sur des architectures matérielles comportant quatre et huit disques; effectivement, les temps diminuent proportionnellement au nombre de disques.

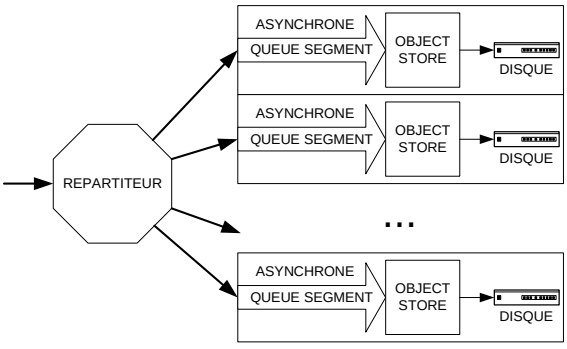

Figure 8. *Parallélisation du gestionnaire de segments d'index*

N'oublions pas que nous avions des contraintes économiques (budgétaires) pour le matériel. La réalisation d'une telle architecture est simplifiée avec des disques SATA. Des disques de 80 Go reviennent à 50 € et un contrôleur de 4 disques coûte 80 €. Pour moins de 300 €, on peut donc avoir une grappe de 4 disques. Pour 1'200 €, il est possible de créer une machine d'indexation de 16 disques (sans compter le boîtier pour les ranger). Dès lors, cette solution technique est viable du point de vue économique; le nombre de disques dépend de la taille du corpus et de la vitesse attendue par le client.

Il nous reste à traiter de la sélection des segments à mémoriser.

7.5. *Retour à la gestion des caches*

Au vu de ce qui précède, il est évident que la gestion du cache d'indexation doit minimiser la segmentation des index. Il faut essayer de maintenir aussi longtemps que possible les index dans le cache dans l'espoir qu'ils seront réutilisés. Nous avons donc implémenté une stratégie qui vide partiellement le cache. Les segments vidés du cache sont les plus grands. Dans le tableau 6 (à gauche), nous voyons l'impact de la taille du cache d'indexation et du taux de remplissage sur le nombre de segments. Pour l'indexation de la collection CLEF.FR 2005, nous avons généré 417'930 segments pour un cache de 64Mo, soit un taux de remplissage de 80% (après vidage). Pour un cache de 16Mo, nous doublons le nombre de segments, ce qui a une incidence sur le temps d'indexation.

Pour conclure sur le traitement des index, nous allons examiner la question de leur compression.

7.6. *La question de la compression*

La compression peut être utilisée pour deux raisons. La première est de gagner de l'espace sur les disques. La deuxième concerne un gain de temps lors du transfert de l'information (sur le réseau, ou dans notre cas du disque à la mémoire). Ces gains sont payés en temps CPU pour compresser et décompresser.

Le gain d'espace sur disque n'est pas un objectif dans notre cas, car nous utilisons plusieurs disques qui sont généralement peu remplis.

Le gain de transfert est intéressant pour des volumes importants. On peut actuellement compter sur des vitesses de transfert de 32Mo/sec; un vecteur de 1Mo demande 31ms de transfert. On peut escompter un gain de 2 avec une compression (voir MG [Witten 1999]); on gagne donc environ 16ms. Par contre, il faut encore compresser/décompresser, ce qui va réduire notre gain.

Nous avons simulé un algorithme de compression, en réduisant par deux la taille des vecteurs d'indexation (le coût de compression est de zéro dans cette simulation). Le résultat est décevant: en effet, le gain enregistré n'est que de 2%. En examinant le tableau 6 (à droite) sur la distribution des segments, on constate que seuls 1,3% des segments ont plus de 8Ko; dès lors, peu d'accès sont réellement concernés par la compression. Cette constatation reflète bien la réalité: peu d'accès avec de grands

volumes et un grand nombre d'accès avec quelques octets. L'accent doit donc être mis sur les accès, ce que nous faisons en distribuant l'index sur plusieurs disques.

Nous avons temporairement renoncé à utiliser la compression en production. Cependant, dans le cas d'une distribution de l'indexeur sur un réseau, la diminution des taux de transfert peut rendre la compression intéressante. Celle-ci peut être aussi utilisée pour conserver plus d'informations dans les caches d'indexation, limitant encore davantage la segmentation. Par ailleurs, nous utilisons la compression dans la gestion des propriétés des documents. Ces propriétés sont représentées sous forme de vecteur de bits qui sont compressibles (le facteur de compression peut être de l'ordre de 1'000).

Taille du cache en Mo	Keep %	#seg	Temps total indexation [sec]
256	80	332361	157
64	90	411333	166
64	80	417930	167
64	40	468816	171
64	20	552524	173
64	0	928983	177
16	80	995602	187

taille des zones en octets	nombre d'occurrences
<=32	2216264
64	143292
128	104464
256	74916
512	53232
1024	37820
2048	25588
4096	19080
8192	12900
16384	9612
32768	7524
65536	5884
131072	4600
262144	3596
524288	2548
1048576	1204
2097152	292
4194304	8

Tableau 6. A gauche, expériences sur la segmentation (CLEF2005.FR) ; à droite, répartition des zones en fonction de leur taille (CLEF2005 Multi8)

8. *Quelques tests et remarques*

Le tableau 7 rassemble les tests effectués avec les collections décrites dans le tableau 2. Les indexations sont réparties en deux groupes : avec ou sans les positions. Le groupe avec les positions est comparé avec le système Lucene [Hatcher 2004]. Le groupe sans les positions est comparé avec MG[Witten 1999]. Les comparaisons ont portés uniquement sur des collections en anglais car les systèmes MG et Lucene sont déjà paramétrés avec des listes de mots vides pour cette langue. VLI fait l'objet de deux mesures, l'une appliquant la stratégie « chaînage arrière » et l'autre « ObjSto ». Nous indiquons un débit en Goctets/heure pour la stratégie « ObjSto ». À la fin du tableau, nous avons ajouté des tests

couvrant d'autres domaines de l'indexation. Sauf mention explicite, les tests ont été effectué sur un AMD 64 4000+.

TEST sur *AMD 64 4000+ RAM 2Gb* *(VLI :4 dsk / Win 2003)* *(MG :1 dsk pour MG / Suse 10)* *(LUCENE:2 dsk 7.2K+10K / Win 2003)*	MG [sec]	LUCENE [sec]	VLI – Chaînage arrière [sec]	VLI-ObjSto [sec]	Débit VLI ObjSto [Go/h]
Indexation avec les positions					
DOC JDK + ORA *(P4M 1.8GHz + 1 dsk)*		1'280	640	745	2.4
CLEF2005.EN		592	108	179	10.5
TECH17		40'174	4'101	6'264	9.9
Indexation sans les positions					
CLEF2005.EN	390		78	90	20.9
TECH17	abort[17]		3'008	3'368	18.4
TECH108			29'008	77'992	5.0
Autres tests sans les positions					
CLEF2005-Multi 8			865	1'003	12.6
CLEF2005-Multi 8 + lemmatisation			947	1'074	11.7
ERP *(Xeon 3G/ 4 dsk)* + nombres			2'083	3'138	8.7

Tableau 7. *Ensemble des tests effectués.*

En examinant l'indexation de la collection *DOC JDK + ORA* Nous obtenons les résultats suivant : pour Lucene, 1'280 secondes ; pour VLI, 745 secondes. Notre indexeur est presque deux fois plus rapide. Ce temps est cohérent avec les performances annoncées sur le site de Lucene, qui sont de 20Mo/sec pour un P4 M 1.5GHz. Regardons comment les 745 secondes sont distribuées : 10% du temps est passé à explorer des répertoires et à ouvrir/fermer des fichiers. Presque 35% du temps est passé à trouver les termes à indexer. 44% est passé à indexer (les segments sont temporairement sauvegardés en mode chaînage arrière). 11 % sert à sauvegarder les segments dans l'ObjectStore, donc à optimiser l'index.

Temps passé à gérer des fichiers	Temps passé à lire et à analyser le html	Temps passé à indexer et à sauvegarder (mode chaînage arrière)	Temps passé à sauvegarder les segments
77 sec	258 sec	325 sec	85 sec
10,3%	34,6%	43,6%	11,4%

Tableau 8. *Répartition des activités pendant l'indexation de la collection DOC JDK* + ORA pour VLI.

[17] Après 4800 secondes, dans la phase de `mg_compression_dict` une erreur de segmentation survenait !

Pour éviter de perdre du temps dans les répertoires de fichiers, les collections sont généralement fournies sous forme d'un seul fichier avec des balises XML. Les collections CLEF2005 et TECH sont de ce type.

La philosophie d'indexation de Lucene est radicalement inverse de la nôtre, qui consiste à séparer les termes, documents et index. Dans Lucene, termes, documents et index sont toujours présents ensemble dans un segment d'indexation (ce qui est parfait pour uniformiser l'échange d'index). Lucene indexe un lot de documents et lui associe un segment. Il procède de même avec le lot suivant. Dès qu'un certain nombre de segments sont en attente, Lucene procède à la fusion de ces segments; nous avons alors un plus gros segment, et ainsi de suite. Le temps d'indexation d'un lot est constant. Par contre, celui de la fusion des segments augmente avec la taille des segments. Avec la collection TECH17, les temps pour les derniers niveaux de fusion se comptaient en dizaines de minutes. Cette façon de procéder est semblable à la stratégie du chaînage arrière, qui requiert une défragmentation des segments. Les arguments sur l'inefficacité de cette stratégie à long terme pour de très grands corpus sont transposables à la façon de procéder de Lucene. Pour être précis, il faudrait parler de l'implémentation Java de l'indexation. En effet, nous pourrions exporter le résultat de notre indexation au format Lucene et fournir ainsi une nouvelle solution d'indexation. Ces remarques sont confirmées par les tests sur les collections CLEF2005.EN et TECH17 avec Lucene (nous avons augmenté le nombre de fichiers traités à 1'000 et le nombre de segments en attente à 20). Dans le tableau 7, nous reportons les résultats. Le rapport de performance entre VLI et Lucene est de 3.3 pour CLEF2005.EN. Le rapport de performance entre VLI et Lucene est de 6.41 pour TECH17. Il semble bien que plus la collection croît, plus l'écart se creuse.

Pour le groupe sans position, nous avons uniquement pu comparer MG sur la collection CLEF2005.EN car pour la collection TECH17 une erreur survenait pendant l'indexation. Le rapport de performance entre VLI et MG est de 4.3 pour CLEF2005.EN. Il est intéressant de noter que le débit varie peu entre l'indexation de la collection CLEF2005.EN et celle de TECH17, toutes deux sont indexées avec une stratégie FAST. Par contre la collection TECH108 utilise une stratégie BIG son débit diminue d'un facteur quatre.

Le dernier groupe rassemble les tests portant sur des variations portant sur la constitution des termes à indexer. Pour la collection CLEF2005-Multi 8, on faisait varier les paramètres d'indexation : avec/sans lemmatisation pour huit langues (FR, EN, SP, IT, DU, DE, NW, FI). Dans le domaine de l'archivage, nous avons l'archive d'un ERP où les nombres étaient aussi indexés. La richesse terminologique et la complexité de l'analyseur ont fait baissé les débits.

L'ensemble de ces expériences nous conduit à formuler les remarques suivantes :

– la performance est sensible au conditionnement des fichiers à indexer ; les collections de fichiers dans des répertoires sont pénalisées par la nécessité d'ouvrir et fermer des fichiers;

– la performance varie avec les opérations sur les termes, la lemmatisation, la normalisation (mise en minuscules, suppression des accents, …);

– la performance est sensible à la nature de l'indexation: garder les positions pour exécuter des opérateurs de proximité est coûteux en temps et en espace;

– l'indexation utilisant le chaînage arrière est très rapide, mais rappelons qu'elle laisse les index fragmentés. L'absence de fragmentation est le prix payé avec ObjSto. La vitesse de pointe n'est pas toujours le critère principal, la régularité et la qualité de l'indexation sont à long terme plus importantes. Rappelons que Google [Brin 1998], [Huang 2000] annonçait une vitesse moyenne de 45 documents par seconde (ce qui correspond à environ 350Mo/heure), faible vitesse qui n'a pas empêché son succès;

– la notion de débit en Go/heure n'est pas fiable car elle dépend finalement de la densité du fichier qui indique le nombre moyen d'octets par terme. La variation entre un corpus en TXT et en HTML peut aller jusqu'à un facteur cinq;

– les spécificités du corpus du Teratrack de TREC 2004 [Clarke 2004] sont (TERA ~ 426Go – 25 millions de documents HTML). En extrapolant nos résultats, l'indexeur mettrait 40 heures pour traiter ce corpus. Ce résultat est proche de celui obtenu par Amberfish [Nassar 2004] sur un Xéon 3GHtz;

Pour dépasser cette limite, la distribution de l'indexation devient nécessaire. Par sa modularité, le moteur d'indexation est un bon candidat pour une telle distribution (par exemple avec les services RMI). Cependant, nous pensons que ces architectures distribuées ne posent pas uniquement un problème d'indexation; elles s'accompagnent de problèmes concernant la sécurité, la sauvegarde, la robustesse, etc. L'article concernant le système de fichiers de Google illustre ces problématiques [Ghemawat 2003].

Bien que les autres modules de notre moteur d'indexation n'aient pas été traités ici, notamment la gestion de la concurrence, les opérateurs d'interrogation, l'usage des propriétés des documents, le langage d'interrogation, les signatures des documents, etc. , pour vérifier la qualité des index, nous avons effectué un test portant sur des requêtes. Les résultats sont reportés dans le figure 9. Nous avons préparé un fichier comportant 100'000 requêtes dont les termes sont choisis dans la collection testée. Les types de requêtes sont : simple sur un terme ; un ET portant sur deux termes et un ET portant sur trois termes. La distribution des termes respecte la loi de Zipf et la distribution des types de requête est uniformément réparties. Ce lot de requêtes constitue une approximation simple du comportement des utilisateurs tel qu'il est analysé dans [Jansen 2000]. Ce lot de requête est ensuite présenté aux indexeurs LUCENE et VLI sur un ordinateur qui vient de démarrer pour la collection TECH17. Sur la figure, on peut constater, pour les deux systèmes, que les premières requêtes s'exécutent lentement et que progressivement les débits augmentent pour finalement s'établir à environ 2000 requêtes/seconde pour VLI et 700 pour LUCENE. En fait, les deux systèmes profitent du cache de Windows, plus il y a d'interrogations, plus grande est la probabilité de retrouver l'index d'un terme dans une page mémoire déjà lue. L'écart entre VLI et LUCENE

s'explique par le fait qu'à l'interrogation VLI gère un cache d'indexation pour les termes et donc en profite pleinement dans le cas où l'on réutilise les mêmes termes. Une comparaison approfondie des mécanismes d'interrogation est hors du cadre de cet article. Cependant on peut constater que pour les deux systèmes, les index ne sont pas fragmentés car cela aurait donné des temps de réponse de plusieurs secondes, le temps nécessaire à la reconstitution des index de chaque terme. Les débits obtenus sont possibles si le serveur d'index est placé dans une architecture trois rangs et donc est dédié à l'indexation et à l'interrogation.

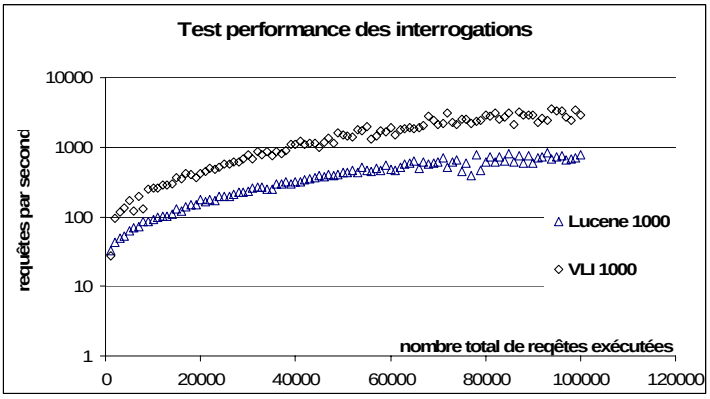

Figure 9. *Tests d'interrogation sur la collection TECH17 avec un jeu de 100'000 requêtes. LUCENE a achevé sa tâche en 451 sec et VLI en 188 sec. Le graphique montre le débit en requête par seconde pendant la progression du test.*

8. Conclusion

La méthodologie utilisée pour la construction de notre indexeur et son passage à l'échelle possède la trame suivante :

1. effectuer une revue de détail des techniques, algorithmes et systèmes existants
2.
3. démonter, analyser les processus et les flux ; chercher les limites, les avantages et désavantages de chaque implémentation.
4. écrire un système « simple » qui servira de référence (pour la correction des tâches) et de modèle pour la décomposition modulaire. implémenter et tester les stratégies alternatives.
5. chercher les nouvelles limites, préparer des réponses possibles

Lors de l'analyse des systèmes et des stratégies, il faut se replacer dans le contexte technologique et des objectifs visés du système étudié. De notre point de vue, MG et LUCENE sont issus d'une période où les mémoires étaient relativement petites (32-128Moctets) et donc la stratégie est entièrement tournée vers une

organisation sur disque avec des fusions successives. VLI exploite une relative abondance de mémoire (2 Goctets) et donc sa stratégie est de construire les index en mémoire et de minimiser les coûts de la segmentation des index.

Le passage à l'échelle est un mouvement dynamique. Une nouvelle ère se profile où les mémoires sont plus vastes (8-24Goctets) et les systèmes d'exploitation et machines virtuelles Java ont un adressage sur 64 bits. Nous avons déjà entrepris des tests avec ces nouvelles configurations. Dans cette nouvelle ère, les processeurs sont multi cœurs et permettent donc de paralléliser la construction des segments d'index. La modularisation des fonctions principales d'indexation se prête bien à ce genre d'exercice. L'avènement des « solid state disk » à prix raisonnable pourrait aussi transformer le paysage des stratégies d'indexation.

Néanmoins, au-delà des arguments techniques discutés, il semble que la construction du moteur d'indexation et son intégration dans un environnement de production aient les facteurs de succès suivants : un cahier des charges clair, avec des objectifs précis à atteindre et des données réelles pour vérifier si les contraintes sont satisfaites; un langage de programmation robuste, avec un large ensemble d'API stables, bien documentée, avec son polymorphisme et ses classes abstraites, avec ses vérifications de type à la compilation, et avec ses vérifications de bon usage des variables et des débordements à l'exécution; une décomposition complète des fonctionnalités, avec une modularisation de celles-ci, et avec un ensemble de stratégies implémentant la même fonction pour couvrir des besoins différents en vitesse et en volume; une vision à long terme, tant en termes de performance pour l'indexeur qu'en termes de durée et de maintenance du moteur; une sélection d'algorithmes performants et des implémentations adaptées aux problèmes à résoudre et à l'architecture matérielle; un cycle de développement rapide (Xtreme Programming) permettant de tester les fonctionnalités et de corriger rapidement les erreurs; la remise en question permanente des opinions et des hypothèses de travail, qui conduit à réaliser de nombreux tests pour les infirmer ou les confirmer; et enfin, la communication et le respect entre les différents partenaires.

Nous remercions Joël Bourquard de sa participation à la mise au point de ObjSto. Nous tenons à remercier aussi Saïd Radhouani pour la relecture attentive de cet article. Enfin, nous remercions toutes les personnes qui ont bien voulu nous écouter et nous conseiller au cours de ces cinq dernières années.

Bibliographie

Baeza-Yates R., *A Fast Set Intersection Algorithm for Sorted Sequences,* in 15th Combinatorial Pattern Matching 2004, LNCS, Springer, Istanbul, Turkey, July 2004.

Bonjour M., Falquet G., Guyot J., Le Grand A., *Java: de l'esprit à la méthode. Distribution d'applications sur internet*, International Thomson Publishing France, Paris, 1996.

Brin S., Page L., *The anatomy of a large-scale hypertextual Web search engine*, Proceedings of the seventh international conference on World Wide Web, Brisbane, Australia, 1998.

Brow E. W., Callan J.P., Croft W.B., *Fast incremental indexing for full-text information retrieval*, Proc. VLDB conference Santiago, Chile, 1994

Clarke C.; Craswell N.; Soboroff I. , *Overview of the TREC 2004 Terabyte Track*, The Thirteenth Text Retrieval Conference (TREC 2004) , NIST 2004

Eliot J., Moss. B., *Design of the Mneme persistent object store*, ACM, Transactions on Information Systems, 8(2):103–139, April 1990.

Fall C. J., Benzineb K., Guyot J., Törcsvári A., Fiévet P., *Computer-Assisted Categorization of Patent Documents in the International Patent Classification*, in Proceedings of the International Chemical Information Conference (ICIC'03), Nîmes, France, October 2003

Ghemawat S., Gobioff H., Leung S.-T., *The Google File System*, SOSP'03, October 19–22, 2003, Bolton Landing, New York, USA.

Grosso W., *Java RMI*, 572 p., O'Reilly, 2001

Guyot J., Radhouani S., Falquet F., *Ontology-Based Multilingual Information Retrieval*, Working Notes for the CLEF 2005 Workshop, Vienna, Austria, Sept 2005

Guyot J., *YAAA: yet another alignment algorithm - Alignement ontologique bi-texte pour un corpus multilingue*. Rapport technique, CUI, Université de Genève, 2005.

Jansen B. G., Spink A., Saracevic T. *Real life, real users, and real needs: a study and analysis of user queries on the web*, ACM, Information Processing and Management: an International Journal, Volume 36 , Issue 2 , Jan. 2000

Hatcher E., Gospodnetić O., *Lucene in Action*, , Manning, 2004

Hitchen R., *Java NIO*, 302 p., O'Reilly, 2002

Huang L., *A Survey on Web Information Retrieval Technologies* , (www.ecsl.cs.sunysb.edu/tech_reports.html) site : Feb 2000.

Kornai A., *How many words are there?,* p 61-86, Glottometrics 4, Ram-verlag, 2002

Nassar N., *Amberfish at the TREC 2004 Terabyte Track*, The Thirteenth Text Retrieval Conference (TREC 2004) , NIST 2004

Salton G., *The SMART retrieval system : experiments in automatic document processing*, Automatic Computation, Prentice-Hall, Englewood Cliffs, New Jersey, 1971.

Sun Microsystems, *The Java HotSpot virtual machine white paper*, http://java.sun.com/products/hotspot/, 2001

Witten I. H., Moffat A., Bell T. C., *Managing gigabytes*, Morgan Kaufmann, second edition, 1999.

Zipf G. K. , « *Human behavior and the principle of least effort* », Addison-Wesley, Reading, Ma. 1949.