Projet de programmation comparée

2010/2011

WIKILINKS

I) Travail réalisé

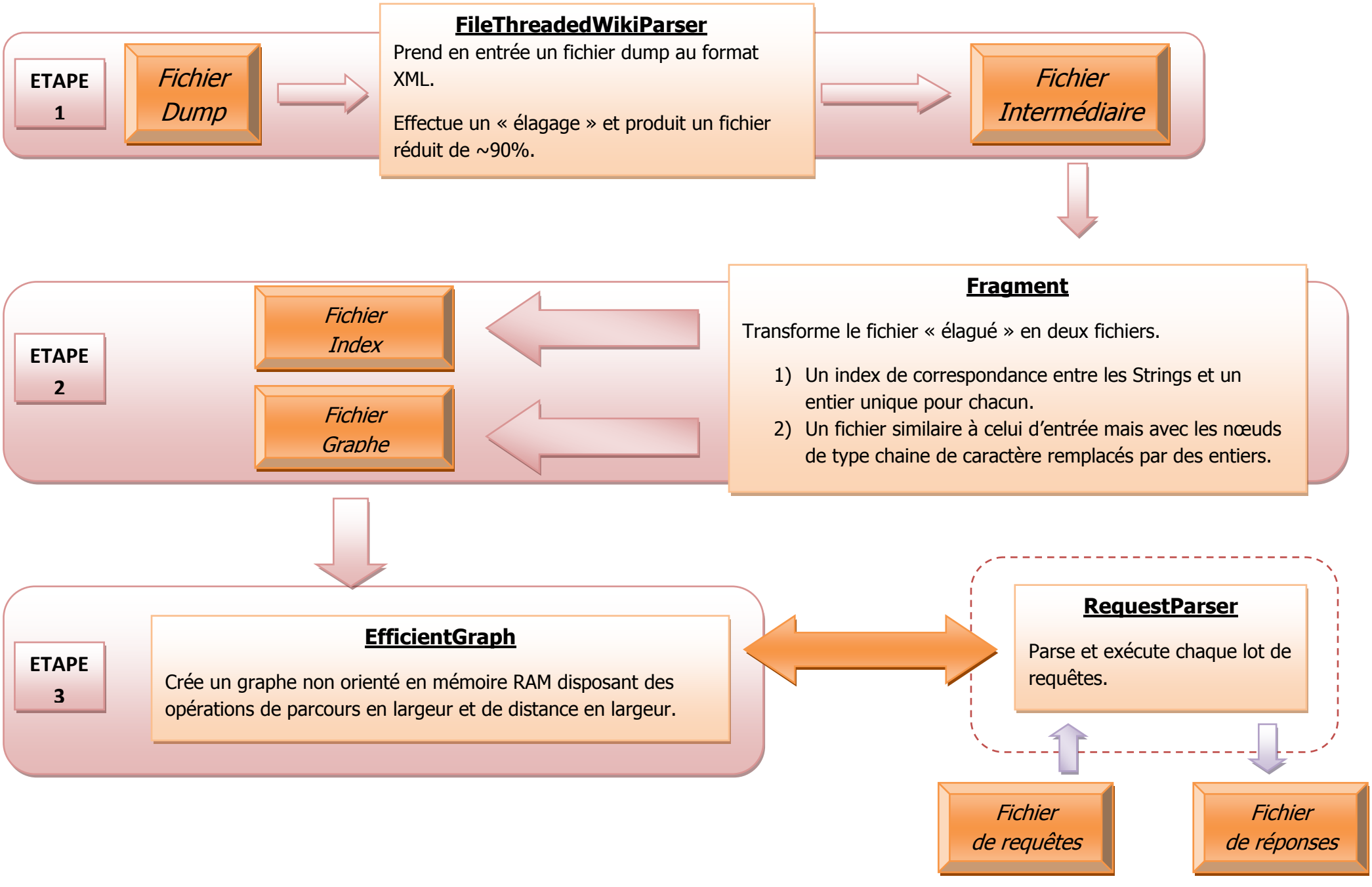
Le programme réalisé dans le cadre de ce projet permet d'analyser un fichier xml contenant un « dump » du site wikipédia (France) selon deux paramètres : la distance en largeur et en profondeur définis sur le sujet du projet.

Les fichiers d'entrée et de sortie sont conformes aux règles définies.

II) Algorithme

L'architecture utilisée permet de traiter le fichier dump en plusieurs étapes distinctes.

Le diagramme suivant présente de manière générale l'algorithme de transformation de fichier dump en graphe utilisable.


ETAPE 1
Fichier Dump
FileThreadedWikiParser
Prend en entrée un fichier dump au format XML.
Effectue un « élagage » et produit un fichier réduit de ~90%.
Fichier Intermédiaire
ETAPE 2
Fichier Index
Fichier Graphe
Fragment
Transforme le fichier « élagué » en deux fichiers.
1) Un index de correspondance entre les Strings et un entier unique pour chacun.
2) Un fichier similaire à celui d'entrée mais avec les nœuds de type chaine de caractère remplacés par des entiers.
ETAPE 3
EfficientGraph
Crée un graphe non orienté en mémoire RAM disposant des opérations de parcours en largeur et de distance en largeur.
RequestParser
Parse et exécute chaque lot de requêtes.
Fichier de requêtes
Fichier de réponses

a) Etape 1

L'étape 1 utilise un parseur XML pour transformer le fichier de 7 Go en fichier réduit de 700Mo.
On garde le contenu des tags <title> et <text>, plus précisément les liens de type «[[» «]] ».
Le parseur développé est multi-threadé afin d'accélérer le traitement.

b) Etape 2

Pour des raisons de performances et de conservation de mémoire, on souhaite remplacer toutes les occurrences des noms des pages wikipédia contenus dans le fichier réduit par des entiers.
On crée donc deux fichiers, un index d'une part pour retrouver un entier affecté à un nom de page par la suite, et le fichier réduit de l'étape 1 contenant des entiers.

c) Etape 3

On crée un graphe non orienté à l'aide du fichier précédemment crée. Ce graphe ne contenant que des entiers, est léger et peut être placé en mémoire RAM ce qui accélère considérablement toute requête ultérieure.

Ce graphe est représenté dans un int [ ] [ ] [ ].

Premier tableau : Position représentant le numéro du nœud.
Longueur : Nombre de nœuds.
Second tableau : Position 0 : Relations sortantes / Position 1 : relations entrantes.
Longueur : 2
Troisième tableau : Liste de nœuds en relation.

d) Requêtes

L'algorithme utilisé pour le calcul de la distance en profondeur est un simple BFS.
Pour la distance en largeur, l'algorithme est trivial vu qu'on dispose dans le graphe d'une liste des liens pointant vers le nœud avec un accès de complexité O(1) …

III) Choix effectués et contraintes d'implémentation

Cette implémentation est particulièrement intensive en utilisation mémoire, c'est pourquoi il est conseillé de la faire tourner sur une machine disposant de 6 Gigaoctets minimum.

Cela est dû à deux facteurs majeurs :

- L'utilisation d'une HashMap à l'étape 2 pour accélérer le remplacement des chaînes de caractères du fichier en entiers. Cette HashMap contient tous les nœuds sous forme de String ce qui prend pour 2M de nœuds environ 1,8Go en mémoire.
- Le stockage de tous les nœuds et relations du graphe dans un triple tableau d'entiers, ce qui est beaucoup moins couteux que des objets et prend à peu près 1Go de mémoire RAM pour 25M de liens.
- Les opérations d'intersection, de jointure et d'élimination de doublons utilisées pour traiter les requêtes couteuses en mémoire.

1) Avantages de l'implémentation et performances

Stocker un graphe en mémoire permet d'effectuer des requêtes dans un temps record. De plus les premières étapes du traitement sont effectuées dans des fichiers, ce qui permet de les ignorer pour tout traitement ultérieur.

En tout et pour tout, sur une machine milieu de gamme (notebook dual-core & 4Go RAM), les trois premières étapes prennent moins de 10 minutes et le remplissage du graphe (3ème étape) prend quand à lui 3 minutes.

2) Désavantages

Cette implémentation a l'énorme désavantage de nécessiter un matériel disposant d'énormément de mémoire afin d'être performante.

Néanmoins on peut nuancer cet argument car ce type de programme est destiné à être exécuté sur des serveurs. Or ces machines disposent d'une quantité de mémoire largement supérieure à la moyenne.

3) Justification des choix sous forme de FAQ

Pourquoi avoir choisi un tableau d'entiers pour représenter le graphe ?

Le graphe est représenté par un simple tableau d'entiers, peut être moins extensible, flexible et élégant qu'une librairie orientée objet mais la consommation mémoire est divisée par dix.

Etait-il vraiment utile de stocker le graphe en RAM ?

Je pense que le gain de performances compense la limitation mémoire, sachant que ce type de calculs est destiné à des serveurs largement pouvus en RAM.

Pourquoi n'avoir multithreadé que la première étape du traitement ?

Lors de la première étape, une grande partie des calculs venaient d'opérations sur les résultats lus. Le calcul était facilement fragmentable.
Les étapes suivantes en revanche sont toutes limitées par des facteurs comme des appels systèmes ce qui empêche toute parallèlisation.

Tous les liens sont-ils pris en charge ?

Non ! Les liens contenant des « : » ne sont pas examinés, on regarde uniquement les liens sur des pages « brutes » de wikipédia et on ignore les pages de discussion et autres.
De plus il était difficile de trouver un listing de toutes les possibilités de préfixe (comme [[category :…]], [[discussion:…]] …) et pour des raisons de temps cela a été abandonné.

Que peut on configurer ?

Le fichier Parameters.java permet de configurer les PATH vers les différents fichiers, et des options de performances pour l'étape 1 du traitement.

4) Solutions essayées et abandonnées

- La librairie **JGraphT** libre permettant de stocker les graphes sous forme d'objets en RAM. Cette librairie bien que très pratique est extrêmement couteuse en mémoire.
- La base de données **neo4j** abandonnée pour des raisons de temps de remplissage incroyablement lent, et une récupération des données peu pratique.
- **Postgresql** pour des raisons similaires.