Données sur le Web

TP Mondial

Colin FRAPPER
Paul-Alexandre TESSIER

1. Introduction

Dans le cadre du module de Données sur le web, il nous a été demandé de réaliser un programme permettant d'obtenir un document XML d'exemple pour le contexte sur les espaces maritimes vu en cours.
Nous avons donc dû réaliser plusieurs programmes dans différents langages, trois principalement ; xslt, SAX, et DOM. Nous avons aussi réalisé avec XML Writer et XML Reader, ainsi que les deux versions de JDOM avec et sans filtre.

2. Algorithme

Pour commencer, nous avons réalisé ensemble un algorithme qui nous semble le plus optimisé. Il permettra de résoudre le problème avec un temps d'exécution faible. Il suffira ensuite de l'adapter aux différents langages ou de le changer complètement si la mécanique du langage ne le permet pas.

L'algorithme suit cette logique :

- Dans un premier temps, on va créer la balise <liste-espace-maritime> et ajouter dans celle ci les balises <espace-maritime>. Pour cela on ne va prendre que les balises <sea> et créer une balise fille <cotoie> pour chaque pays présent dans l'attribut country de la balise <sea>.
- Ensuite, on crée la balise <liste-pays> et les balises <pays> provenant des pays qui sont côtoyés par un espace maritime. Il ne doit pas y avoir de doublons, il faut donc faire attention lors de la création que cela soit réalisé de manière unique. Lors que la création d'un pays, il faut bien faire attention à prendre la dernière population indiqué car il peut y avoir plusieurs fois la balise <population>
- Puis, on parcours les fleuves dans le but de créer les balises <fleuve> fille d'une balise <pays>. Il faut bien faire attention à ne prendre que les fleuves qui se jettent dans une mer ou un océan et non un lac. Une fois la balise créé avec les informations demandés, on va créer ses balises filles <parcourt>. Pour cela on va récupérer les pays présent dans l'attribut country et créer une balise <parcourt> pour chaque pays. Enfin, il faut ajouter la balise <fleuve> à un pays. Pour déterminer cela on ajoute le fleuve au pays où le fleuve y prend sa source, si il a 2 sources on ne prend pas le fleuve. Si le pays n'existe pas à ce moment, la il faut le créer.
- Pour finir, on crée les pays qui sont uniquement parcouru et donc pas créé précédemment.

Pour vérifier nos réalisations nous avons principalement comparé le nombre de balises. Ainsi il faut 218 pays, 154 fleuve, 251 balises parcourt, 44 espaces maritimes et 340 balises cotoie.

3. XSLT

Dans un premier temps, ils nous étaient demandés de réaliser une version en xslt pour obtenir le résultat attendu.
Rappelons que xslt est un langage qui sait analyser un arbre XML en utilisant le langage XPath et nous avons utilisé ce langage dont nous détaillerons l'utilisation un peu plus loin, xslt permet également de définir des règles de transformation à appliquer aux éléments de l'arbre XML, règles dont chacune produit un fragment du document résultat.

Le document produit est donc produit en "deux" partie, dans une première, une liste de pays avec les fleuves et les pays qui parcourt ce fleuve. La deuxième partie concerne la liste d'espace maritime, on recherche toutes les mers ou océans ainsi que tous les pays qui viendront à côtoyer cette mer (balise <cotoie>).

Dans un premier temps afin de créer la liste de pays, et uniquement les pays qui nous intéresse en l'occurrence, on aura donc 3 cas qui nous intéresse :

- Le pays cotoie une mer ou un océan
- Le pays est la source d'un fleuve qui se jette dans une mer ou un océan
- Le pays est traversé par un fleuve qui se jette dans une mer ou un océan

A l'aide de ces trois cas, il nous faut maintenant déterminer leur expression XPATH.

Premier cas ; Le pays cotoie une mer ou un océan :

```
/mondial/country[@car_code = ../sea/id(@country)/@car_code]
```

Explication : On va donc utiliser la fonction id, du fait que l'attribut country de la balise sea est une clé étrangère relative à l'attribut car_code de la balise pays. À l'aide de cette expression on aura donc tous les pays qui sont côtoyés par une mer ou un océan.

Deuxième cas ; Le pays est la source d'un fleuve qui se jette dans une mer ou un océan

```
/mondial/country[@car_code = ../river[id(to/@water)/@id =
../sea/@id]/id(source/@country)/@car_code]
```

Explication : Celle-ci est un tout petit peu plus complexe, on va d'abord "filtrer" tous les fleuves qui se jettent dans une mer ou un océan et dans un second temps vérifier si la source de ces fleuves sont égale à l'identifiant du pays, si c'est le cas on va aller "visiter" le pays qui nous intéresse

Troisième cas ; Le pays est traversé par un fleuve qui se jette dans une mer ou un océan.

```
/mondial/country[../river[id(to/@water)/@id =
../sea/@id]/id(@country)/@car_code = @car_code]
```

Explication : Dans le même style que la précédente, on va filtrer les fleuves recherchés et on ira “visiter” tous les pays traversés par ce fleuve.

Une fois avoir écrit toutes les balises pays, ils nous faut écrire les balises fleuves, à l’aide de cette expression : `../river[(id(source/@country)/@car_code = current()/@car_code and id(to/@water)/@id = ../sea/@id)]`

qui nous permet d’avoir tous les fleuves qui se jettent dans la mer ou océan et dont le pays est la source de ce fleuve.
À chaque fleuve, il nous faut donc associé une balise parcourt représentant les pays traversés par ce fleuve, on va à partir de cette partie faire appelle à la modalité d'application des règles pour pouvoir différencier le fait que le fleuve traverse un ou plusieurs pays afin de connaître la valeur de l’attribut distance. Dans le cas où le fleuve ne traverse qu’un seul pays, on va passer en paramètre du template la longueur de ce fleuve.

La première “partie” du document étant fini, il nous faut maintenant écrire la seconde. Il nous suffit de parcourir toutes les mers ou océans (balise <sea>), et d’appliquer une règles sur l’attribut country de la balise sea qui correspond à tous les pays côtoyés par cette mer, on utilisera une nouvelle fois la modalité d’application des règles pour pouvoir différencier cette règle des autres.

Note : Nous avons également réalisé une version plus procédurale en utilisant les foreach (que vous trouverez en commentaires dans le code).

Parlons maintenant de la complexité du programme, comme vous nous l’avez recommandé, nous avons utilisé saxon afin de calculer le temps nécessaire à l'exécution du programme

En terme de performance, xslt n’est pas le plus rapide par rapport aux autres (cf tableau), en terme de clarté du code xslt est sans aucun doute le meilleur de tous les langages de traitements de fichier XML vu dans ce TP.
Et pour finir en terme de maintenabilité le code, le code est facilement adaptable.

4. SAX

Sax est une des plus anciennes API utilisé pour traiter du XML. Pour réaliser ce programme, nous avons eu quelques difficultés à conceptualiser comment nous allions réaliser cet algorithme. Voici comment fonctionne SAX (de ce que nous avons compris) : Un "parseur" parcours le document dans l'ordre de lecture et, à chaque fois qu'il rencontre un objet XML (commentaire, élément, texte, etc.), il produit un événement.

Le fait que l’on ne peut pas parcourir l’arbre comme on le souhaite, il nous est venu à l’idée de stocker les éléments qui nous préoccupent et d’ensuite les parcourir afin de les écrire. Pour cela nous avons créé 3 classes ; la classe Country, Fleuve, et Mer, avec pour chacune

de ces classes différents attributs. l'idée est la suivante : À chaque fin de balise rencontrée on va donc créer un pays, fleuve, ou mer dépendant de la balise et l'insérer dans un tableau ayant été créé dans la classe "Mère". A l'issue de la fin du document, dans la fonction endDocument() on va donc parcourir ces différents tableaux, selon ce qu'on cherche, effectuer des tests classiques afin de trouver ce que l'on recherche. Pour être honnête, nous n'avions pas trouvé d'autres solutions à cette question et la notre nous semble un peu "tordu" et peu performante (peut-être est ce le langage qui veut ça).
La complexité de ce programme : SAX n'autorise que des traitements quasi-linéaires (les références avant/arrière sont difficiles).
Le traitement de SAX en lui même ne prend pas vraiment énormément de temps, par contre le fait de devoir parcourir nos tableaux afin d'écrire les éléments souhaités nous prend pas mal de temps.

Le principal atout de SAX est de ne pas nécessiter une mise en mémoire de l'intégralité du document. Seules les informations des unités syntaxiques courantes sont en mémoire à un moment donné.

En terme de performance, le fichier s'exécute en environ 900 ms, le langage SAX est le plus maintenable de tous. Néanmoins avec notre solution, en terme de clarté ce n'est pas le plus lisible (spécialement lors de la réécriture d'élément).

Note : Nous avions utilisé votre librairie Sax4PHP pour réaliser cet exercice.

5. DOM sans Xpath

En utilisant DOM, il était possible de respecter l'algorithme défini de base. Sans Xpath il fallait donc parcourir le document avec un foreach et en prenant les éléments fils de la balise principal avec childNodes.

Comme c'était en PHP, certaines fonctions étaient pratique comme les tableaux pour pouvoir stocker les pays à ajouter par la suite. Il suffisait ensuite de faire un array_unique pour supprimer les doublons. On pouvait aussi utiliser in_array pour vérifier si un élément était dans le tableau.

La seule contrainte de cette méthode est qu'on ne peut pas accéder directement à un élément fils déterminé par exemple <name> dans la balise <country>. Il fallait parcourir tous les éléments fils avec un foreach et vérifier si la balise courante avait comme nom celle voulu.

La complexité du programme est la suivante :

- Pour les espaces maritimes on aura n si n est le nombre d'éléments du document de base. Cela est en raison du fait qu'on est obligé de parcourir tout le document pour accéder aux éléments souhaités.
- Ensuite, pour ajouter les pays, il s'agit encore de n car pour ajouter les pays on doit encore parcourir tous les éléments et voir si il s'agit d'une balise <country> et que son identifiant est dans le tableau des pays à ajouter.

- Puis, pour les fleuves on est encore obligé de parcourir tout le document donc n. Ensuite pour chaque fleuve valide ou pour chaque pays parcouru , on doit ajouter le pays si il n'est pas déjà créé. Cela est encore en complexité de n.

On a donc une complexité de n.

Lors de l'exécution on obtient un temps de 350 millisecondes environ.

6. DOM avec Xpath

En utilisant DOM avec Xpath, cela permet d'écrire moins de code et de résoudre le problème précédent car avec Xpath, on peut directement accéder à un élément. Normalement, le temps d'exécution devrait être inférieur car dans la précédente méthode, on était obligé de parcourir tous les éléments dans une boucle.

Lors de plusieurs exécutions, on obtient un temps moyen de 300 millisecondes environ, soit moins que la version sans Xpath.

7. XML Reader et XML Writer

Créer le fichier avec XML Reader et XML Writer est une partie facultative dans l'optique de comparer d'autres possibilités de parser le fichier. Pour ce faire, nous avons utilisé la même base que le fichier SAX. On lit le document exactement comme SAX et on peut détecter s'il s'agit d'une balise ouvrante ou fermante. Il suffit juste d'utiliser les fonctions propres à XML Reader pour créer les tableaux de données que l'on souhaite utiliser par la suite.

Pour écrire le fichier final, on utilise XML Writer. Il correspond à la première version du fichier SAX sauf qu'on remplace les echo par les fonctions de XML Writer.

Le temps d'exécution est d'environ 410 ms. Cela permet de garder la maintenabilité du fichier SAX mais en aillant un temps d'exécution 2 fois plus faible.

8. JDOM

JDOM est une API open source Java, ce n'est pas un parseur en lui même, il a d'ailleurs besoin d'un parseur externe de type SAX ou DOM pour analyser un document et créer la hiérarchie d'objets relative à un document XML. L'utilisation d'un parseur de type SAX est recommandée car elle consomme moins de ressources que DOM pour cette opération

Il nous était demandé de réaliser deux versions de JDOM, une avec filtre et une sans. Pour réaliser ces deux programmes, nous nous sommes appuyés sur l'algorithme que nous avons réalisé pour le programme SAX. Les deux versions ont été réalisées cependant la version avec filtre nous a semblé plus difficile à comprendre.

9. Conclusion

En ce qui concerne SAX, il est plus facile de traiter de gros documents qu'avec DOM qui à son contraire demande de charger tout le document en mémoire avant même toute

manipulation. On peut ainsi générer un résultat dès lors qu'on parcourt un élément dès le début de la lecture du document. Le principal avantage de SAX est la maintenabilité du code par rapport à par exemple DOM.
Néanmoins, SAX ne permet d'effectuer que des traitements quasi-linéaires, il est impossible de revenir en arrière après avoir passé l'élément ou du moins c'est très difficile. Il est également difficile de détecter les erreurs de validations.

Comme nous l'avons vu en cours, DOM est un standard du W3C permettant de parser des fichiers XML. DOM contient certaines spécificités qui peuvent poser problème, le principal problème est l'occupation en mémoire, avant même de pouvoir traiter un document, il est nécessaire que ce dernier soit complètement chargé en mémoire contrairement à SAX. On remarque également que pour des traitements dits linéaires, DOM va parcourir deux fois le document. Ces inconvénients sont résolus dans l'API dont nous avons parlé plus haut : SAX. DOM est surtout pratique, car c'est un langage plus trivial que SAX ou XSLT.

Pour XMLReader/XMLWriter, l'association des 2 méthodes n'est pas obligatoire ce qui fait qu'il est possible de l'améliorer si on trouve une meilleure méthode qui substitue l'un des 2. Son fonctionnement ressemble fortement à SAX mais on est pas obligé de le suivre, on a plusieurs fonctions qui permettent une liberté d'utilisation.

Le fichier bash que nous avons créé exécute les différents langages. Il permet de comparer les temps d'exécution facilement. On peut conclure que l'utilisation de XSLT ou JDOM donne un temps d'exécution long par rapport à DOM ou XML Writer/Reader.

Pour l'écriture des résultats, il est plus aisé de le réaliser en DOM puis sauvegarder en XML plutôt que d'écrire directement le fichier avec des commandes echo ou les fonctions de XML Writer.

Figure 1 :
Tableau représentant les temps d'exécutions
des différents programmes.

Nous pensons également qu'il est important d'avoir un esprit critique sur notre travail et avec du recul, il nous a manqué un poil d'organisation au niveau du temps pour finir la dernière question facultative, mais dans l'ensemble nous sommes plutôt content de notre travail.

En fin de conclusion, il a été très intéressant de travailler avec tous ces parseurs de fichiers XML et de comparer lequel est le "meilleur" de tous, notre conclusion est qu'il n'y a pas vraiment de meilleur parseur, chaque parseur va s'adapter à différents cas, il convient juste de choisir le bon parseur pour un problème posé.