Techniques de test

Compte rendu TP3

Profiling

Ricardo Rodríguez

Techniques de test
M1 ILSEN
Université d'Avignon et des Pays de Vaucluse

Introduction

Dans ce document on va décrire les tests de profiling réalisés pour le TP3 du cours Techniques de test. Les méthodes de profiling vont analyser le comportement de trois méthodes pour lire des fichiers et vont extraire des données qui nous permettront de choisir la meilleure option.

On va ensuite générer des diagrammes en étoile (Kiviat) pour accompagner l'analyse.

Données récupérées

Pour chaque méthode, on va l'exécuter en boucle 10 fois et on va récupérer les données suivantes :

- Durée totale de la boucle (millisecondes)
- Durée moyenne d'une itération (millisecondes)
- Durée dans le pire des cas (millisecondes)
- L'écart-type des durées
- Mémoire utilisée
    - Dans le sujet du cours il était demandé de récupérer la moyenne et l'écart type pour la mémoire. Néanmoins il existe une contrainte technique, pour obtenir la mémoire on a accès aux fonctions memory_get_usage() et memory_get_peak_usage(). La différence entre ces deux valeurs va nous dire quelle a été la quantité de bytes utilisés pendant l'exécution. Si on fait appel à memory_get_peak_usage() à l'intérieur de la boucle, après chaque appel à la méthode en test, on peut récupérer des valeurs fausses, parce que le peak n'est réinitialisé par PHP qu'à la fin de tout le script. C'est-à-dire que si une itération de la boucle utilise une quantité énorme de mémoire, pour les prochaines itération le peak récupéré sera le même.
    - J'ai donc décidé de ne récupérer que la quantité de mémoire utilisée pour toute la boucle. Cette valeur est calculée comme la différence entre la mémoire avant la boucle et la valeur de peak résultante après le for.
- Portabilité.
    - Une valeur qui représente la possibilité d'utiliser la méthode dans d'autres environnements, dans une échelle de 0 à 10.
- Complexité
    - Nombre de lignes de code nécessaires pour écrire la méthode.

Tous les tests ont été faits avec un fichier qui contient un million de lignes, chaque ligne contient un numéro aléatoire.

Résultats

Valeurs bruts après l'exécution des méthodes :

	Valeurs bruts		
Caractéristique	**Lire fichier**	**Charger fichier**	**Shell**
Durée totale	2,9988	4,462	0,2288
Durée moyenne	0,2999	0,4462	0,0229
Pire durée	0,3311	0,4522	0,0254
Écart-type	0,010467	0,004269	0,001271
Portabilité (sur 100)	100	70	30
Complexité	8	6	2
Mémoire utilisée	18056	183072912	17920

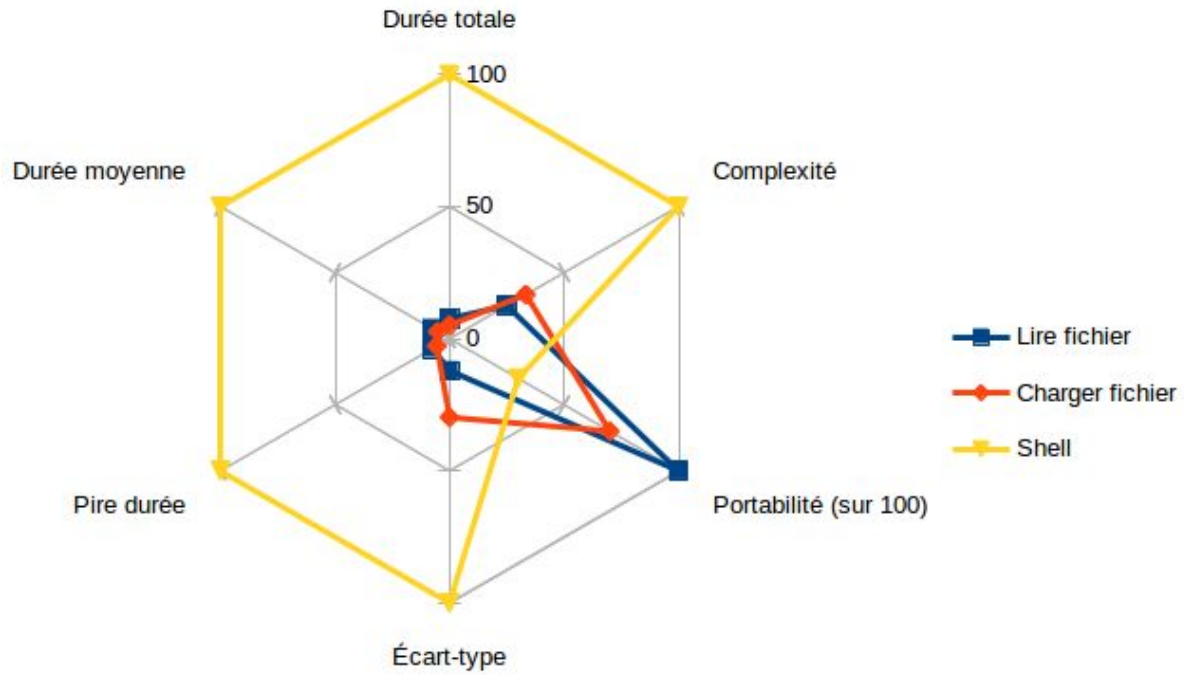
Valeurs normalisées dans une échelle de 0 à 10.
Pour certaines variables j'ai réalisé une transformation de la valeur mesurée : j'ai normalisé l'inverse de la valeur récupérée (1/valeur). Ceci permet de noter de manière positive une méthode qui a un écart-type bas (c'est-à-dire que le temps d'exécution ne change pas trop d'une exécution à l'autre) et de manière négative un écart-type élevé. C'est similaire pour la complexité, une complexité plus élevée est une qualité négative. Idem pour les durées.

	Valeurs normalisées [0, 10]		
Caractéristique	**Lire fichier**	**Charger fichier**	**Shell**
Durée totale	7,6297185541	5,1277454056	100
Durée moyenne	7,6358786262	5,1322277006	100
Pire durée	7,6713983691	5,6169836356	100
Écart-type	12,1429253845	29,7727805107	100
Portabilité (sur 100)	100	70	30
Complexité	25	33,3333333333	100
Mémoire utilisée	99,2467877714	0,0097884498	100

Analyse

Diagramme sans mémoire

On va d'abord analyser le diagramme Kiviat sans prendre en compte la consommation de mémoire.



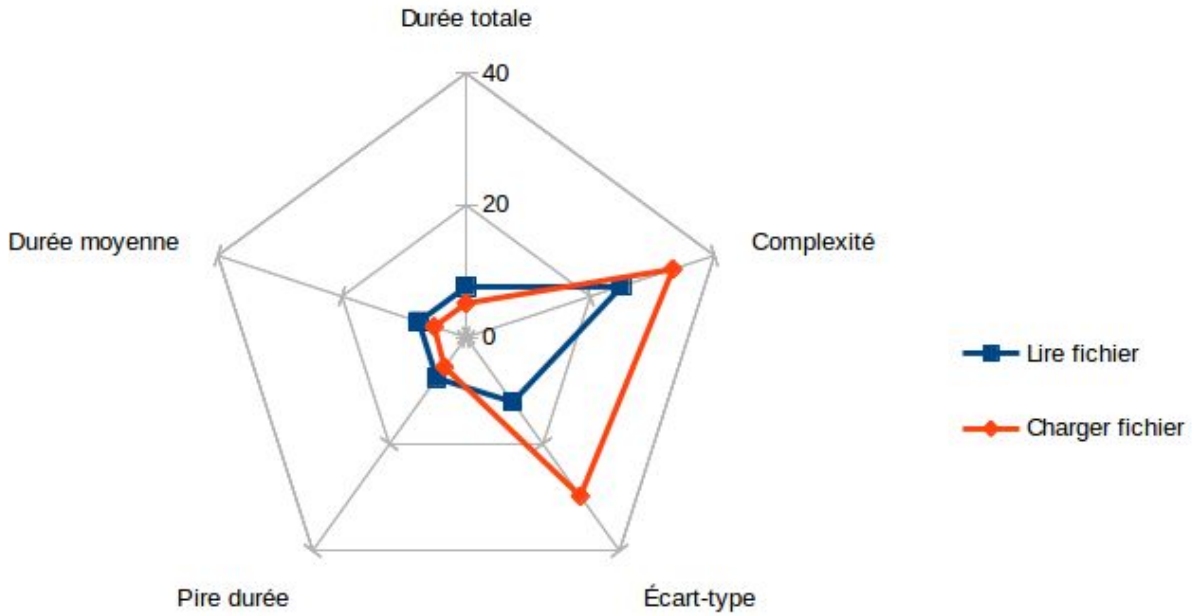
On peut observer que la méthode utilisant le shell est la plus performante en termes de temps d'exécution. En plus, d'une itération à l'autre il n'y a pas une grande différence dans le temps nécessaire pour lire le fichier, ce qui permet d'avoir une grande confiance dans l'utilisation de cette méthode. La complexité est très basse, il faut juste quelques lignes de code pour faire appel aux commandes shell.

On va utiliser un diagramme où l'on ne prend pas en compte le shell afin de mieux apprécier les différences entre les autres méthodes.

En ce qui concerne la portabilité, on a donné une valeur de 100 à la méthode classique qui ouvre un fichier et le lit ligne par ligne. Cette méthode n'utilise que des fonctions PHP et le temps d'exécution est très acceptable. La méthode qui charge tout le fichier avant de le lire à une valeur de portabilité inférieure puisqu'il faut avoir une machine puissante pour lire des gros fichiers. Finalement, l'utilisation du shell est le choix le moins portable du fait que l'on est obligé d'être dans un environnement Linux.

Diagramme sans shell

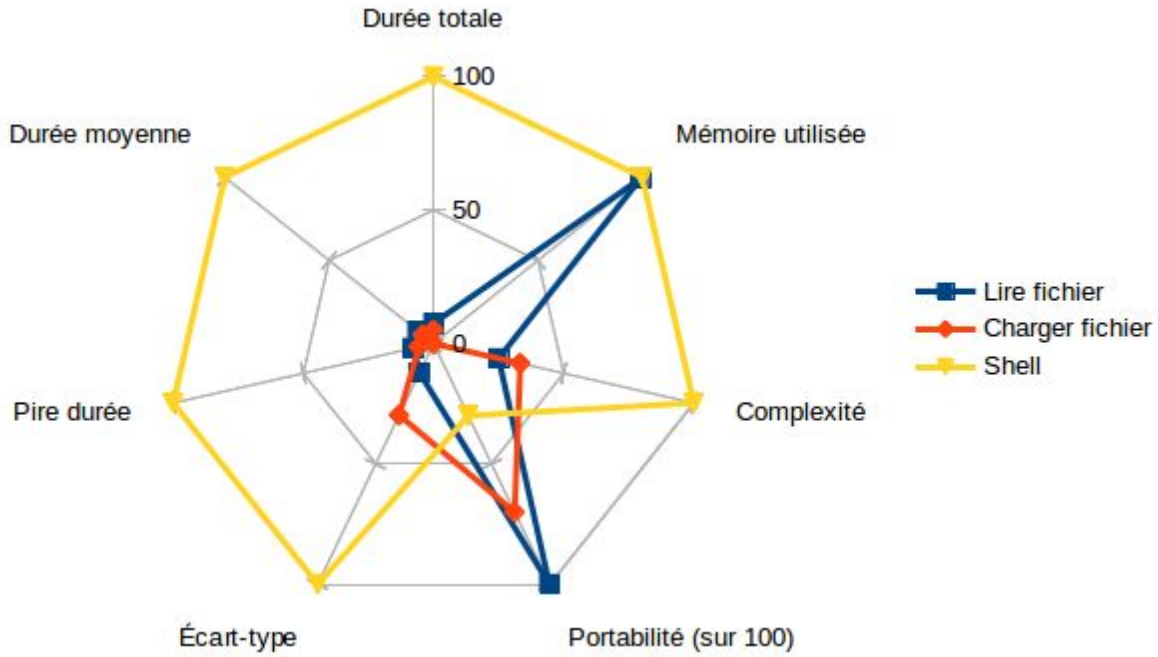
Dans le diagramme qui suit je n'ai pas inclu la méthode shell afin de mieux apprécier les différences entre les autres deux méthodes.



On observe que la lecture du fichier ligne par ligne prend moins de temps que la technique de charger et lire (autour d'un 50% moins d'après le tableau de données). La charge et lecture du fichier est juste un peu moins complexe à écrire et l'écart relatif entre les différentes itérations est aussi inférieur.

Diagramme avec mémoire

Finalement, on va analyser la consommation de mémoire à partir du diagramme avec toutes les variables.



On peut observer que la charge du fichier consomme extrêmement plus de mémoire (la valeur représentée sur le diagramme est très proche de zéro). En fait la consommation va augmenter par rapport à la taille du fichier, ce qui fait de cette méthode la pire pour traiter des grands fichiers.

En outre, les deux méthodes restantes consomment à peu près la même quantité de mémoire. Dans le diagramme c'est difficile à voir, mais on peut le vérifier dans le tableau de données.

Conclusions

Si on est sûrs de ne pas avoir besoin de porter l'application à des environnements non-Linux, on va choisir la méthode qui utilise le shell sans hésiter. Cette méthode apporte un gain énorme en termes de performance et si jamais les commandes shell sont actualisées afin d'apporter une performance encore supérieure, notre application bénéficiera de ses améliorations automatiquement.

Par contre, pour le cas général, on va préférer utiliser la méthode qui lit le fichier ligne par ligne. Les temps d'exécution sont très acceptables et le programme sera complètement portable d'un environnement à un autre.

Bien évidemment, l'option de charge complète ne sera prise en compte que si les fichiers à traiter sont petits et utilisés très souvent. On peut imaginer un cas d'utilisation dans lequel il faut accéder de manière très rapide à quelques petits fichiers qui peuvent être chargés qu'une seule fois. Cette méthode pourrait apporter un gain dans le temps d'accès puisque leur contenu se trouverait en mémoire. Dans le cas plus général, on ne va jamais l'utiliser.