HadoopTrace - Rapport de Projet de fin d'étude

Rédigé par : Dan Nguyen, Rémy Zirnheld, Encarant : François Trahay

Introduction

Depuis les premiers pas d'Hadoop au début des années 2000, beaucoup de frameworks permettant de faire du calcul distribué pour traiter des gros volumes de données ont vu le jour : ElasticSearch, Storm ou plus récemment Spark. Aujourd'hui, peu d'outils existent pour profiler les applications utilisant ces frameworks, la plupart de ces outils servant pour le calcul haute performance.

EzTrace est l'un de ces outils : écrit en C, il permet de profiler des applications elle aussi écrites en C utilisant les frameworks classiques tels que OMP ou MPI, ou plus récemment CUDA. Le projet HadoopTrace a justement pour objectif d'étendre les champs d'application d'ezTrace pour permettre de profiler les applications utilisant le framework Hadoop, lui écrit en Java.

Objectifs du projet

L'objectif général du projet était de contribuer au projet EzTrace, en ouvrant la voie sur le profiling d'applications Java. Nous nous sommes donc fixé trois objectifs :

- Créer un module ezTrace pour Hadoop MapReduce
- Faciliter la création de modules ezTrace pour des applications Java
- Tester le module ezTrace pour limiter l'overhead induit par son utilisation

Pour cela, nous avons du nous familiariser avec plusieurs technologies :

- Le fonctionnement du framework Hadoop MapReduce et d'HDFS (Hadoop Data File System)
- Java Native Interface (JNI), afin d'appeler des bibliothèques écrites en C depuis du Java
- L'instrumentation en Java, avec l'utilisation de Javassist notamment
- La création d'un module ezTrace

Réalisation

Nous avons tout d'abord réaliser des mini-projets pour maîtriser JNI, l'instrumentation Java et le framework Hadoop MapReduce. Nous avons ensuite fusionné ces mini-projets afin d'aboutir à un prototype stable et fonctionnel du module ezTrace final.

Prise en main des outils

Hadoop MapReduce

Le MapReduce est un paradigme de programmation permettant d'effectuer des traitements relativement simples sur de gros volumes de données. Développé chez Google par Jeffrey Dean et Sanjay Ghemawat, il simplifie grandement le passage à l'échelle ainsi que la gestion des fautes.

Le traitement de données utilisant le paradigme de programmation MapReduce est consistuée de différentes étapes, dont deux sont implémentées par l'utilisateur : Map et Reduce. Voici les différentes fonctions sur l'exemple WordCount que l'on a implémenté :

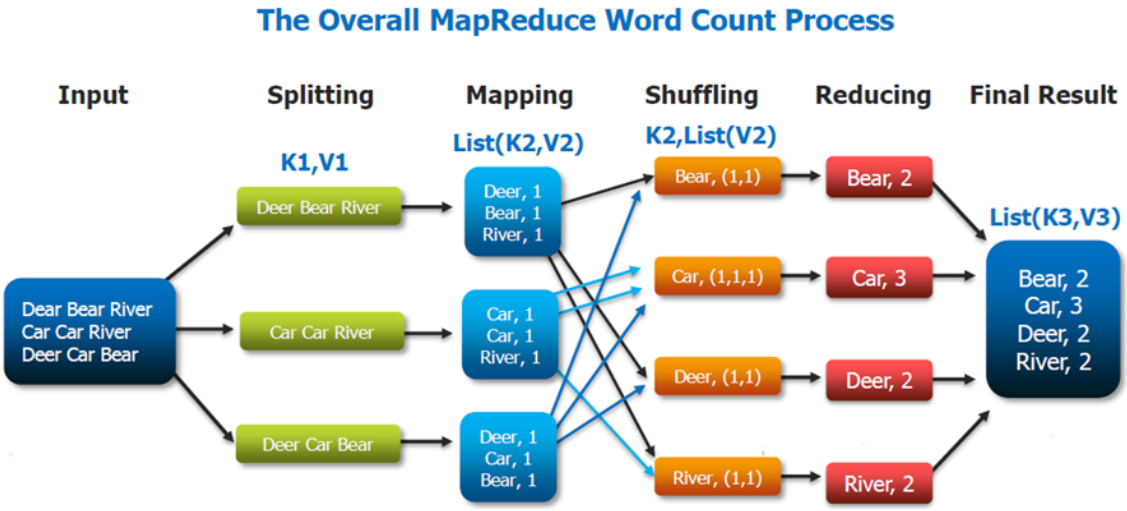


Figure 1: Fonctionnement du MapReduce sur le WordCount

- Splitting : Les données en entrée sont tout d'abord partitionnées pour être envoyés sur les noeuds du cluster : chaque noeud est ainsi responsable d'une partie des données
- **Map** : Cette fontion réalise le premier traitement effectué sur les données : il doit avoir pour sortie un ensemble de couple clé-valeur. Dans le cas d'un WordCount, les clés sont les mots et les valeurs associées sont le nombre d'occurences du mot dans la partition de données traitées par le noeud.
- Group by Key/Shuffling : Les données sont regroupé par clés. Tout les couples ayant une même clés sont envoyé sur le même noeud.
- **Reduce** : Cette fonction réalise le second traitement sur les données : il prend en entré des ensemble de couples ayant la même clé, et dépend complètement l'application. Dans le cas du WordCount, on additionne, pour chaque clé, toute les valeurs des couple ayant cette clé. On obtient ainsi l'occurence du mot dans tout le fichier d'entrée.

Ce paradigme de programmation peut être utilisé pour d'autres problèmes, tel que des tris ou d'autres problèmes d'analyse de données.

Dans le cadre du projet, on souhaite analyser la répartition de charge entre les noeuds du cluster, et donc repérer l'execution des fonctions maps et reduces sur les différents noeuds,

puisque ce sont ces fonctions qui réalisent des calculs. Ce sont donc ces deux fonctions que l'on a cherché à instrumenter.

JNI : Java Native Interface

JNI regroupe toute les commandes et fichiers sources permettant d'éxecuter du code écrit en C, dit natif, à partir d'une application Java.

Afin de définir des fonctions natives, il suffit d'ajouter le mot clés `native` dans la déclaration de la fonction :

```
public class MyClass {
    public native void function();
    static {
        System.loadLibrary("mylib");
    }
}
```

En compilant le fichier `.java` à l'aide de l'option h, on obtient alors un fichier `header` à implémenter :

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class mypackage.MyClass */

#ifndef _Included_MyClass
#define _Included_MyClass
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:     mypackage.MyClass
 * Method:    function
 * Signature: ()V
 */
JNIEXPORT void JNICALL Java_MyClass_function
        (JNIEnv *, jobject);

#ifdef __cplusplus
}
#endif
#endif
```

Il faut alors compiler le fichier source en tant que bibliothèque partagée. Cette dernière sera alors chargée au chargement de la classe grâce au bloc static décrit dans MyClass.java.

Pour notre projet, nous nous somme servie de JNI pour implémenter les fonctions à executer avant et après les appels des fonctions map et reduce, en utilisant les bibliothèques ezTrace.

L'instrumentation Java avec Javassist

L'instrumentation en Java est native, c'est-à-dire qu'elle est déjà prévu par le langage : en effet il suffit pour cela de créer ce qu'on appel un agent. Un agent est tout simplement une classe qui contient la methode `static void premain(String, Instrumentation)`, méthode executée avant le main lorsque l'on précise l'option `javaagent` à l'execution d'une application Java.

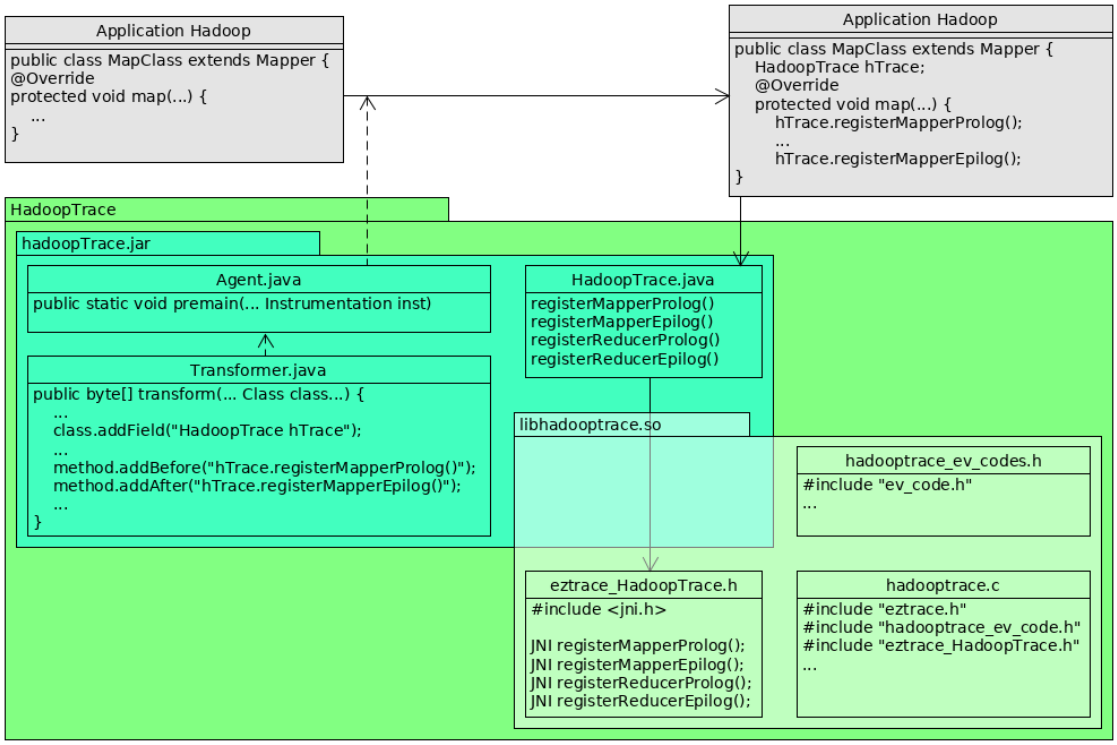
Pour modifier le code à l'execution, cet agent doit être accompagné d'un ou plusieurs objet de type `java.lang.instrument.ClassFileTransformer`, qui specifie le code à ajouter à l'execution. Pour implémenter son propre transformer, il suffit de surcharger la méthode `transform`, qui prend est executé pour chaque `.class`, et qui prend en entré le nom de la classe et le byteCode de la classe notamment, et en sortie le nouveau byteCode de cette classe. Afin de pouvoir modifier ce dernier facilement, nous avons utilisé Javassist. Cette bibliothèque fournit un ensemble de classe et méthode facilitant l'injection de byteCode.

Dans le cadre de notre projet, il suffit de créer un transformer qui ajoute des appels appropriés en C avant et après chaque appels des fonctions maps et reduce.

Création d'un module ezTrace

Réalisation du module HadoopTrace

A ce stade, nous avions une idée bien plus précise du logiciel à programmer :



Afin de se familiariser avec Hadoop MapReduce et de pouvoir tester notre application, nous avons créer un mini-projet WordCount, application considérée comme le `Hello World`

d'Hadoop MapReduce. Ce mini-projet a été découpé en deux modules Maven :

- hadoop-application : le WordCount à proprement parlé, qui contient les trois classes nécessaires à l'implémentation d'un MapReduce.
- hadoop-agent : l'ensemble des fichiers sources permettant d'instrumenter notre application

Notre WordCount contient trois classes : la classe WorCount qui permet de lancer l'application, et les classes Map et Reduce qui sont celles à instrumenter. Ces classes surchargent les méthodes map et reduce, méthodes qui définissent le comportement de l'application.

Analyse des performances

- Analyses sur la machine 32 coeurs

Améliorations possibles

- Système permettant de merge les traces générées sur HDFS
- Plugin ezTrace pour faciliter la création de modules ezTrace pour des API Java