IUT Nancy Charlemagne

Université de Lorraine

2 ter Boulevard Charlemagne

54052 Nancy Cedex

Département Informatique

Recherche & Développement autour de la problématique du Big Data.

Rapport de stage pour DUT informatique

Observatoire Astronomique de Strasbourg

Joris VIGNERON

2014/2015

IUT Nancy Charlemagne

Université de Lorraine
2 ter Boulevard Charlemagne
54052 Nancy Cedex
Département Informatique

Rapport de stage pour le DUT Informatique

Observatoire Astronomique de Strasbourg
11, rue de l'Université
67000 Strasbourg

Joris VIGNERON
Tuteur : Monsieur André Schaaff
Parrain de stage : Monsieur Denis Roegel

Table des matières

Introduction 5
Présentation de l’entreprise..... 6
L’Observatoire Astronomique de Strasbourg 6
Les équipes de recherches 7
Galaxies 7
Hautes énergies..... 7
CDS 7
Le CDS 7
International Virtual Observatory Alliance 9
Sujet du stage..... 10
Déroulement du stage 10
Etat de l’art 11
MapReduce :..... 11
Hadoop 13
Spark :..... 16
Maven :..... 18
Pig : 18
Hive : 19
Recherches et Expérimentations..... 20
Généralités sur le Big Data..... 20
Débuts avec Hadoop..... 22
Choix et utilisation d’une distribution Hadoop..... 22
Utilisation de l’interface HortonWorks, les projets Pig et Hive 23
Essais avec Pig : 24
Essais avec Hive : 26
Fonctionnement d’un job MapReduce avec Hadoop..... 28
Premier programme avec Hadoop 28
Déroulement d’un job MapReduce avec Hadoop :..... 29
Schéma représentatif du job WordCount avec Hadoop..... 30
Découverte de Apache Spark 31
Premiers essai avec Apache Spark : 31
Premier job : Compare 32
Second job : Filtre 32

Réalisation d’une application : CrossMatch 33
Objectif d’une application de CrossMatch 33
Descriptions des sources utilisées 34
Réalisation de l’application avec Hadoop 34
TychoJoinMapper (Cf. Annexe 3.1) : 35
HipMainJoinMapper(Cf. Annexe 3.2) : 36
Phase de Shuffle et Sort : 37
CrossMatchReducer (Cf. Annexe 3.3) : 37
CrossMatchDriver (Cf. Annexe 3.4) : 38
Réalisation de l’application avec Spark 39
Configuration et chargement des fichiers 39
Création des couples clé/valeur 39
Réalisation de la jointure 40
Résultat du job 40
Conclusion 42
Remerciement 43
Index 44
Bibliographie 45
Annexes 46
Annexes 1: Wordcount avec Hadoop 46
1.1 Le WordCountMapper 46
1.2 Le WordCountReducer 46
1.3 Le WordCountDriver 47
Annexes 2: Essais avec Spark 48
2.1 Comparaison 48
2.2 CrossSearch 49
Annexes 3: CrossMatch avec Hadoop 50
3.1 TychoJoinMapper 50
3.2 HipMainJoinMapper 51
3.3 CrossMatchReducer 52
3.4 CrossMatchDriver 53
Annexes 4: CrossMatch avec Spark 54

Introduction

Je viens d'effectuer un stage de dix semaines au sein de l'Observatoire astronomique de Strasbourg encadré par Monsieur André Schaaff ayant la fonction d'ingénieur de recherche en informatique. Ce stage m'a permis de découvrir le monde de l'entreprise mais surtout il m'a donné la possibilité d'appliquer les connaissances théoriques vu tout au long de mes 2 années de formation à l'IUT.

Le thème de mon stage s'inscrit dans la problématique du Big Data. L'objectif étant d'étudier l'environnement technologique gravitant autour de ce domaine et notamment le framework Hadoop.

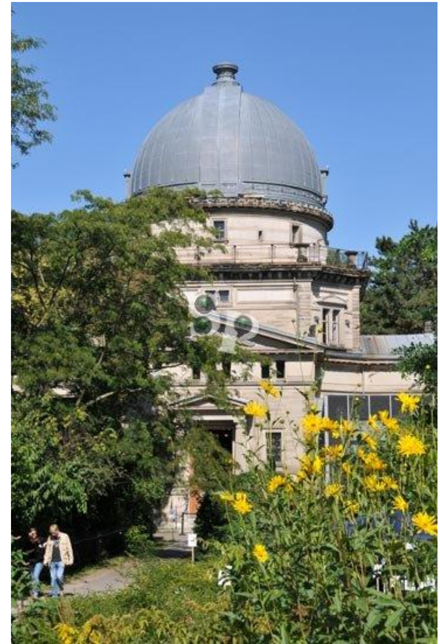
Mon choix pour ce stage a été motivé par 2 raisons :

- La première est parce qu'il se déroulait au sein d'un centre de recherche en informatique et en astronomie, un domaine que je souhaitais découvrir.
- La seconde est que le sujet proposé était un sujet de Recherche et Développement, une méthode de travail que je voulais expérimenter.

Tout d'abord, je présenterai l'entreprise et son activité. Puis je parlerais des différentes technologies que j'ai utilisées durant mon stage. Ensuite j'aborderai ma mission et la façon dont je l'ai mené à bien. Pour finir, je ferai un bilan de mon expérience à l'Observatoire astronomique de Strasbourg.

Présentation de l’entreprise

L’Observatoire Astronomique de Strasbourg



L'Observatoire Astronomique de Strasbourg est un Observatoire des Sciences de l'Univers, une école interne et UFR de l'Université de Strasbourg, ainsi qu'une Unité Mixte de Recherche entre l'Université et le CNRS. Il est actuellement dirigé par Hervé WOZNIAK.

Il est structuré en trois équipes de recherche et deux Services d'Observation de l'Institut National des Sciences de l'Univers, le Survey Science Centre d'XMM-Newton et le Centre de Données astronomique de Strasbourg. Le CDS est labellisé depuis 2008 « Très Grande Infrastructure de Recherche » par le Ministère de l'Enseignement Supérieur et de la Recherche.

Son statut d'OSU place l'Observatoire Astronomique au cœur du dispositif national mis en œuvre par l'INSU.
Il est composé des équipes de recherche — Galaxies, Hautes énergies et le CDS — et de deux services d’observation de l’Institut National des Sciences de l’Univers — SSC-XMM et le CDS.

L'Observatoire a été construit en 1881 sur le campus historique de l'Université de Strasbourg et héberge la 3ème plus grande lunette de France par sa taille ainsi que le Planétarium de Strasbourg dont il a eu la responsabilité de 1986 à 2008 et qui est désormais intégré au Jardin des Sciences de l'Université de Strasbourg.

L'Observatoire Astronomique de Strasbourg se doit de contribuer aux progrès de la connaissance par :

- l'acquisition de données d'observation
- le développement et l'exploitation de moyens appropriés
- l'élaboration des outils théoriques nécessaires

Il est également chargé :

- de fournir des services liés à son activité de recherche
- d'assurer la formation des étudiants et des personnels de recherche
- d'assurer la diffusion des connaissances
- des activités de coopération internationale

Les équipes de recherches

Galaxies

Le champ d'action de l'équipe Galaxies regroupe tout ce qui concerne la formation des galaxies ainsi que l'étude et le recensement des populations stellaires qui composent ces galaxies. Plus particulièrement, cette équipe va s'intéresser à notre propre galaxie, la Voie Lactée, ainsi qu'à ses voisines du Groupe Local. Outre les recherches sur les populations stellaires, l'équipe Galaxies s'intéresse également à la dynamique gravitationnelle qui régit les étoiles et les matériaux à l'intérieur des galaxies. Le but de cela est de réunir suffisamment d'informations afin de pouvoir établir l'histoire précise de l'évolution d'un système stellaire et de reconstituer les événements clés dans la vie d'une galaxie.
En plus de cela, l'équipe Galaxies mène des recherches sur les amas stellaires — composants fondamentaux d'une galaxie — afin de mieux comprendre la formation de celles-ci.
Enfin, l'équipe s'implique également dans des missions satellitaires telles que Gaia, mission lancée en décembre 2013 par l'Agence Spatiale Européenne ayant pour but de fournir des relevés sur la Voie Lactée.

Hautes énergies

L'équipe Hautes énergies s'intéresse aux sources émettrices de rayon X, aux objets compacts — étoiles à neutrons par exemple — et aux noyaux actifs des galaxies.
Elle est impliquée dans le Survey Science Center d'XMM (SSC-XMM), un consortium international de laboratoires sélectionnés par l'ESA qui est en charge de fournir des catalogues complets d'objets observés par le satellite XMM-Newton.
L'équipe Hautes énergies participe également à des projets communautaires tels que le projet européen Arches, en collaboration avec le CDS.

CDS

Le Centre de Données astronomiques de Strasbourg est à la fois un service d'observation et une équipe de recherche. Le CDS a entre autre développé des services d'accès aux données astronomiques — Simbad, VizieR — et de visualisation d'images — Aladin — qui sont utilisés par l'ensemble de la communauté internationale.
C'est aussi l'un des acteurs majeurs du développement de l'International Virtual Observatory Alliance (IVOA). Depuis 2008, le CDS a été labellisé « Très Grande Infrastructure de Recherche », ce qui le place au même niveau que des infrastructures européennes comme l'European Southern Observatory.

Le CDS

Ayant effectué mon stage au sein de ce service, je vais maintenant le présenter plus en détail.

Créé en 1972, le CDS est d'abord connu sous le nom de Centre de Données Stellaires avant de devenir le Centre de Données astronomiques de Strasbourg en 1983.
Hébergeant entre autres la base de données de référence pour l'identification d'objets astronomiques, ces services — Simbad, VizieR et Aladin — sont largement utilisés par la communauté astronomique internationale.

Ses missions sont nombreuses et comprennent entre autres :

- le rassemblement des informations utiles concernant les objets astronomiques
- la mise à jour de ces données,
- la distribution de ces données à la communauté internationale,
- la mise en place de recherche à propos de ces données.
-

Le CDS emploie actuellement 33 personnes réparties de la façon suivante : 8 chercheurs/astronomes-adjoints, 1 chercheur postdoctoral, 11 informaticiens, 10 documentalistes et 3 administratifs.
Il y a environ 1 000 000 de requêtes par jour sur l'ensemble des services.

<u>Les différents services du CDS :</u>

Simbad

Simbad est la base de données de référence mondiale en ce qui concerne la nomenclature et la bibliographie d'objets astronomiques. Possédant l'équivalent de plus de 40 ans de données, Simbad permet aux astronomes de trouver facilement des informations sur plus de 8 millions d'objets grâce à plus de 290 000 références bibliographiques.
Simbad dispose également d'un résolveur de noms permettant de retrouver n'importe quel objet désigné par l'un des 18 millions d'identifiants que la base contient.
Simbad a reçu en moyenne 284 000 requêtes par jour pour l'année 2011, soit plus de 3 requêtes par secondes.

VizieR

VizieR est une base de données regroupant des catalogues d'objets astronomiques.
Ces catalogues sont constitués de données relevées durant des missions d'observation et sont ajoutés à VizieR par les documentalistes du CDS. A l'heure actuelle, la base est constituée de plus de 13 000 catalogues.
Interrogeable sur de nombreux critères (longueur d'ondes, nom de la mission, etc.), VizieR permet de rassembler et d'homogénéiser les données astronomiques afin de pouvoir les comparer et les exporter.
Le nombre de requêtes faites sur la base de données est en moyenne de 300 000 par jour, avec des pics à plusieurs millions.

Aladin

Aladin est un atlas interactif du ciel permettant de visualiser et de comparer des images du ciel. Il a été entièrement développé en Java par Pierre Fernique, Thomas Boch et François Bonnarel.
Aladin utilise des images provenant d'observatoires au sol ou spatiaux et génère des cartes du ciel en trois dimensions (technologie HEALPix). Ces images peuvent être issues de la base de données inclus à Aladin, d'autres bases telles que la NASA Extragalactical Database, ou encore provenir de l'utilisateur lui-même.
A l'heure actuelle, Aladin est disponible sous quatre formes [4] : une application téléchargeable, une applet Java, une version JavaScript et un simple previewer en ligne. Aladin, c'est actuellement 50 To d'images (près de 200 relevés HEALPix) qui vont fortement augmenter dans les années à venir.

International Virtual Observatory Alliance

L'International Virtual Observatory Alliance est une organisation internationale regroupant les projets d'Observatoire Virtuel de plus de 20 pays (France, Allemagne, Chine, États-Unis, Brésil, Australie, . . .). Le but de l'IVOA est d'établir, au niveau mondial, des standards ainsi que de définir les stratégies à adopter. Cela permet de faciliter les échanges internationaux au même titre que la collaboration entre les différents pays.

Le CDS participe activement à ce projet par ces trois services cités auparavant, mais aussi par son implication au sein des groupes de travail de l'IVOA.
En effet, plusieurs de ces groupes de travail qui composent l'IVOA sont présidés ou
vice-présidés par des membres du CDS.

Sujet du stage

Le CDS a une forte activité de R&D afin de maintenir ses services et outils au meilleur niveau. Le thème générale du stage s'inscrit dans la problématique du Big Data et notamment des technologies qui gravitent autour comme Hadoop ou Spark.

Actuellement, l'Observatoire a déjà mis en place une technologie visant à traiter les larges volumes de données que représentent les catalogues astronomiques. Ce stage a pour objectif principal de trouver une solution avec Hadoop qui pourrait s'avérer plus efficace que le système existant.

Il me fallait donc dans un premier temps appréhender la notion de Big Data et les différentes technologies que je devrais utiliser. Je devais réaliser beaucoup d'essais, par la réalisation d'applications, l'exécution de divers programmes et l'analyse des outils à ma disposition. C'était une part importante de ce stage car c'est cette partie qui allait me permettre de comprendre les différents éléments nécessaire pour coder une application avec Hadoop.

En effet, il me fallait réfléchir et identifier avec mon tuteur les différents cas concrets réalisables que je pouvais mettre en œuvre pour le CDS. A partir d'un cas concret d'utilisation défini, je devais réaliser une application et réaliser des tests avec les catalogues de l'Observatoire.

Déroulement du stage

Etat de l'art

Réalisation de l'application

Essais avec Hadoop

Essais avec Spark

Rédaction du rapport

Etat de l’art

MapReduce :

MapReduce est un patron d’architecture de développement informatique, inventé par Google, dans lequel sont effectués des calculs parallèles, et souvent distribués, de données potentiellement très volumineuses, typiquement supérieures en taille à 1 téraoctet.

MapReduce permet de manipuler de grandes quantités de données en les distribuant dans un cluster de machines pour être traitées. Ce modèle connaît un vif succès auprès des sociétés possédant d’importants centres de traitement de données telles Amazon ou Facebook. De nombreux framework ont vu le jour afin d’implémenter le patron MapReduce. Le plus connu reste Hadoop développé par Apache Software Foundation.

Les programmes adoptant ce modèle sont automatiquement parallélisés et exécutés sur des clusters d’ordinateurs. Le principe du MapReduce repose donc sur l’emprunt du langage fonctionnel, utilisé pour implémenter des opérations sur les données : tri, filtrage, projection, agrégation ou regroupement.

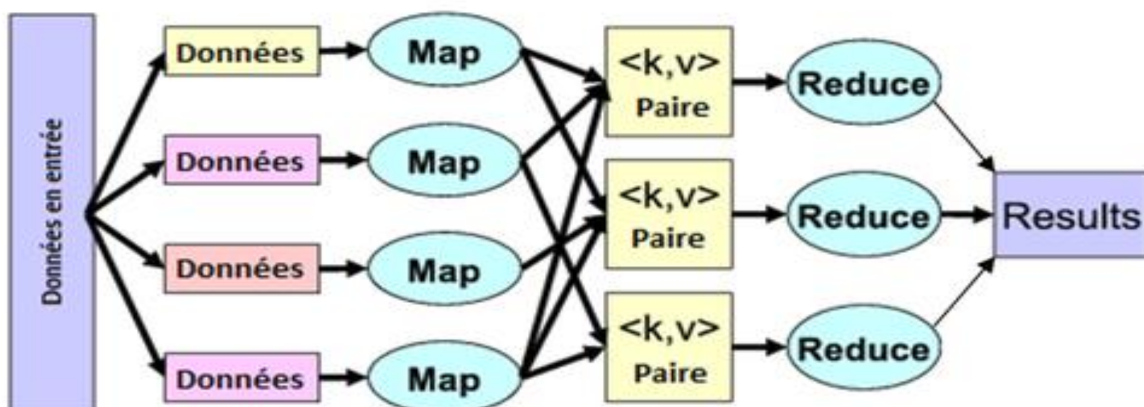
Son fonctionnement :

Le patron MapReduce consiste en grande majorité en deux fonctions : *map()* et *reduce()*

- Dans l’étape « Map », le nœud analyse un problème, le découpe ensuite en sous problèmes, et les délègue à d’autres nœuds, qui peuvent en faire de même de façon récursive. Les sous-problèmes sont ensuite traités par les différents nœuds à l’aide de la fonction *reduce()* qui associe à un couple (clé,valeur) un ensemble de nouveaux couples (clé,valeur) :

  Map(clé1, valeur1) -> list(clé2,valeur2)
- Ensuite vient l’étape « Reduce », durant laquelle les nœuds les plus bas font remonter leurs résultats aux nœuds parent qui les avait sollicités. Celui-ci calcule un résultat partiel à l’aide de la fonction *reduce()* qui associe toutes les valeurs correspondantes à la même clé à une unique paire. Puis il remonte l’information à son tour. A la fin du processus, le nœud d’origine peut recomposer une réponse au problème qui lui avait été soumis :

  Reduce(clé1, list(valeur1)) -> list(valeur2)



Une fois qu'un nœud a terminé une tâche, on lui affecte un nouveau bloc de données. Grâce à cela, un nœud rapide fera beaucoup plus de calculs qu'un nœud plus lent. Le nombre de tâches « Map » ne dépend pas du nombre de nœuds, mais du nombre de blocs de données en entrée. Chaque bloc se fait assigner une seule tâche « Map ». De plus, toutes les tâches « Map » n'ont pas besoin d'être exécutées en même temps en parallèle, les tâches « Reduce » suivent la même logique.

Un cluster MapReduce utilise donc une architecture de type Maître-Esclave, c'est-à-dire qu'un nœud maître dirige tous les nœuds esclaves.

Le patron MapReduce possède quelques caractéristiques :

- Le modèle de programmation du MapReduce est simple mais très expressif. Bien qu'il ne possède que 2 fonctions, *map()* et *reduce()*, elles peuvent être utilisées pour de nombreux types de stockage et peut manipuler de nombreux types de variable.
- Le système découpe automatiquement les données en entrée en bloc de données de même taille. Puis, il planifie l'exécution des tâches sur les nœuds disponibles.
- Il fournit une tolérance aux fautes à grain fin grâce à laquelle il peut redémarrer les nœuds ayant rencontré une erreur ou affecter la tâche à un autre nœud.
- La parallélisassions est invisible à l'utilisateur afin de lui permettre de se concentrer sur le traitement des données.

Le MapReduce est apparu en 2004. La technologie est encore jeune. Elle souffre de quelques points faibles :

- Elle ne supporte pas les langages de haut niveau comme le SQL
- Elle ne gère pas les index. Une tâche MapReduce peut travailler après que les données en entrée soient stockées dans sa mémoire. Cependant, MapReduce a besoin d'analyser chaque donnée en entrée afin de la transformer en objet pour la traiter, ce qui provoque des baisses de performance.
- Elle utilise un seul flot de données. MapReduce est facile à utiliser avec une seule abstraction mais seulement avec un flot de données fixe. Par conséquent, certains algorithmes complexes sont difficiles à implémenter avec seulement les méthodes *map()* et *reduce()*. De plus, les algorithmes qui requièrent de multiples éléments en entrée ne sont pas bien supportés car le flot de données du MapReduce est prévu pour lire un seul élément en entrée et génère une seule donnée en sortie.
- Quelques points peuvent réduire les performances de MapReduce. Avec sa tolérance aux pannes et ses bonnes performances en passage à l'échelle, les opérations de MapReduce ne sont pas toujours optimisées pour les entrées/sorties. De plus, les méthodes *map()* et *reduce()* sont bloquantes. Cela signifie que pour passer à l'étape suivante, il faut attendre que toutes les tâches de l'étape courante soient terminées. MapReduce n'a pas de plan spécifique d'exécution et n'optimise pas le transfert de données entre ces nœuds.

Hadoop

Hadoop est un framework java Open Source destiné à faciliter la création d'application distribuées et échelonnables, permettant aux applications de travailler avec des milliers de nœuds et des pétaoctets de données. Hadoop a été créé par Doug Cutting et fait partie des projets de la fondation logicielle Apache depuis 2009. Ce framework a été inspiré par les publications MapReduce, GoogleFS et BigTable de Google.
Hadoop est le framework le plus utilisé actuellement pour manipuler et faire du Big Data. C'est lui qui va gérer la distribution des données au cœur des machines du cluster, leurs éventuelles défaillances mais aussi l'agrégation du traitement final. Son architecture est du type « Share nothing », c'est-à-dire qu'aucune donnée n'est traitée par deux nœuds différents même si les données sont réparties sur plusieurs nœuds, suivant le principe d'un nœud primaire et de nœuds secondaires.

Hadoop Distributed File System (HDFS) :

Le HDFS est un système de fichiers distribué, extensible et portable développé par Hadoop à partir de GoogleFS. Il est écrit en Java, et conçu pour stocker de très gros volumes de données sur un grand nombre de machines. Il permet l'abstraction de l'architecture physique de stockage, pour manipuler un système de fichiers distribué comme s'il s'agissait d'un disque dur unique.

Son architecture machine repose sur 2 composants majeurs :

- NameNode (noeud de noms) : gère l'espace de noms, l'arborescence du système de fichiers et les métadonnées des fichiers et des répertoires. Il est unique mais dispose d'une instance secondaire qui gère l'historique des modifications dans le système de fichiers, un rôle de backup.
- DataNode (nœud de données) : stocke et restitue les blocs de données. Lors du processus de lecture d'un fichier, le NameNode est interrogé pour localiser l'ensemble des blocs de données. Pour chacun d'entre eux, le NameNode renvoie l'adresse du DataNode le plus accessible, c'est-à-dire le DataNode qui dispose de la plus grande bande passante.

Hadoop dispose d'une implémentation complète de l'algorithme de MapReduce.

Fonctionnement :

Hadoop va tout d'abord découper les données, c'est-à-dire les segmenter sous forme de blocs. Ensuite il va transférer sa fonction *map()* sur chaque nœud/bloc de données et l'appliquer ; la fonction *map()* étant développée en fonction des besoins.

Une fois que toutes les tâches « map » auront été réalisées sur tous les nœuds, on va passer à l'application des méthodes « reduce ». L'entrée d'une fonction *reduce()* correspond généralement à la sortie d'une fonction *map()*.

Il est possible d'appliquer une fonction Combiner en sortie de la fonction *map()* pour ne pas enregistrer les résultats et surtout réaliser une succession de 2 fonctions : une fonction de combinaison et une fonction de tri. La combinaison se charge de regrouper tous les couples possédant la même clé et la fonction de tri va trier les clés. Ces deux fonctions sont déjà implémenter par le framework Hadoop.

L'un des avantages de Hadoop est qu'en se servant du patron d'architecture MapReduce on peut développer des applications réalisant des tâches en parallèles. De plus, ce framework simplifie en grande partie le travail, notamment parce que l'on n'a pas besoin de gérer les enregistrements. En effet, le développeur doit juste écrire le code du « mapper » et du « reducer » pour traiter un seul enregistrement et Hadoop se chargera de passer d'un enregistrement à un autre.

Les utilisations possibles de ce framework :

Grâce à Hadoop, il est possible de mettre en place des programmes comptant le nombre d'occurrences d'une thématique ou d'une valeur, ou encore chercher le maximum ou le minimum. Cependant, il est plus difficile de réaliser une application de calcul de la moyenne des données.

Il est notamment utilisé par des entreprises comme Facebook, Yahoo ou encore Microsoft. Cependant, bien que Hadoop soit le framework le plus connu il possède certains inconvénients qui réduisent ses performances notamment en milieu hétérogène.

Pour pallier à ces soucis, l'Apache Software Foundation a réalisé un certain nombre de projet visant à améliorer Hadoop. Notamment Hbase qui est une base de données distribuées disposant d'un stockage structuré pour les grandes tables. Ce système repose sur des couples clé/valeur appelé aussi stockage binaire.

Il y a également Hive qui met en place une base de données relationnelle à l'intérieur du HDFS pour permettre de réaliser des requêtes en utilisant un langage proche du SQL nommé HiveQL traduites ensuite comme des programmes MapReduce. Ce projet permet donc de mettre en œuvre un langage de requêtes connu par les développeurs.

Tout comme Hive, Pig propose un langage de haut niveau pour accéder aux données. Il s'adresse plus aux développeurs habitués à faire des scripts Bash ou Python.

Hadoop est distribuée par quatre acteurs qui proposent des services de formation et un support commercial, en ajoutant des fonctions supplémentaires :

- Cloudera, première distribution historique d'Hadoop intégrant des packages classiques et certains développement propriétaires.
- HortonWorks
- MapR Technologies a développé un système de fichier pour Hadoop palliant les limites du HDFS. A également réalisé des technologies permettant la suppression du NameNode qui est un point de contentions dans l'architecture d'Hadoop .
- IBM BigInsights for Hadoop, qui est une distribution 100 % Open Source Apache Hadoop, proposant des extensions analytiques et d'intégration dans le SI d'entreprise.

Avantages

Le principal atout des clusters Hadoop est leur adéquation à l'analyse de gros volumes de données. Le Big Data est le plus souvent non structuré et largement distribué.

Si Hadoop convient bien à l'analyse de ce type de données, c'est parce qu'il possède un mode de fonctionnement qui vise à fractionner les données en éléments et attribue chacun d'entre eux à un nœud donné du cluster. Les données n'ont pas à être uniformes, car chaque élément de donnée est traité par un processus distinct d'un nœud distinct du cluster.

Le second avantage des clusters Hadoop réside dans leur évolutivité. Un des problèmes de l'analyse du BigData est sa croissance continue. De plus, le BigData est surtout utile s'il est analysé en temps réel ou dans un délai aussi proche que possible du temps réel. Les capacités de traitement en parallèle de Hadoop participent à la rapidité de l'analyse mais à mesure que le volume de données à analyser augmente, la puissance de traitement du cluster peut devenir insuffisante. Heureusement l'ajout de nœuds peut faire évoluer le framework.

Le troisième avantage de ce type d'architecture réside dans son coût. En effet, l'analyse du BigData fait partie de l'informatique d'entreprise qui est traditionnellement coûteuse, pourtant, les clusters Hadoop peuvent s'avérer être une solution économique. Ils sont généralement peu onéreux pour 2 raisons, d'abord le logiciel est Open Source. La distribution Apache Hadoop est même proposée en téléchargement gratuit. De plus, l'utilisation de matériel standard contribue à limiter les coûts d'un projet Hadoop. On peut donc construire un cluster puissant sans dépenser beaucoup en serveurs.

Un autre avantage de ces clusters est leur résistance aux pannes. Quand un élément de donnée est envoyé à un nœud pour analyse, les données sont également copiées vers d'autres nœuds : en cas de défaillance sur un nœud, d'autres copies existent ailleurs dans le cluster et il est toujours possible de les analyser.

Inconvénients

Cependant les clusters Hadoop ne conviennent pas aux besoins d'analyse de données de toutes les organisations. Notamment, pour des petits volumes de données, les bénéfices seront presque inexistants.

L'un des inconvénients des clusters Hadoop réside dans le principe même de la solution de clustering, qui suppose que les données puissent être fractionnées et analysées par des processus parallèles exécutés sur des nœuds cluster distincts. Si l'analyse ne peut se dérouler dans un environnement de traitement parallèle, un cluster Hadoop n'est tout simplement pas le bon outil.

L'un des plus grand désavantages reste probablement la difficulté d'apprentissage associée à la construction, au fonctionnement et à la prise en charge de la solution.

Spark :

Le framework Spark est un outil permettant de faire du traitement de larges volumes de données, et ce, de manière distribuée, du cluster computing. Le framework offre un modèle de programmation plus simple que celui d'Hadoop et permet des temps d'exécution jusqu'à 100 fois plus courts en mémoire et 10 fois plus cours sur le disque.

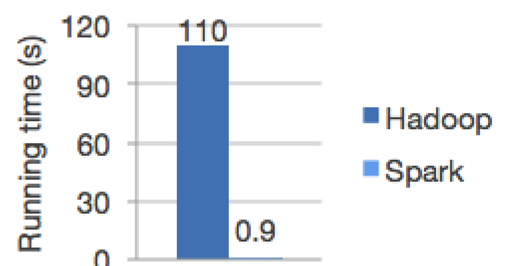
Spark est né dans le laboratoire AMPLab de l'université de Berkeley et partant du principe que : d'une part la RAM coûte de moins en moins cher et les serveurs en ont donc de plus en plus à disposition ; d'autre part, beaucoup d'ensemble de données dits BigData ont une taille de l'ordre de 10Go et tiennent donc en RAM.

Le projet a intégré l'incubateur Apache en juin 2013 et est devenu un « Top-Level Project » en février 2014.

L'écosystème Spark comporte donc plusieurs outils :

- Spark pour les traitements en batch
- Spark Streaming pour le traitement en continu de flux de données
- Mllib pour le machine learning
- GraphX pour les calculs de graphes
- Spark SQL, une implémentation SQL-like d'interrogation de données

Spark est capable de s'intégrer parfaitement avec l'écosystème Hadoop, notamment avec HDFS et des intégrations avec Cassandra et ElasticSearch sont prévues.



Ce framework est écrit en Scala et propose un binding Java qui permet de l'utiliser sans problème en Java.

Notions de Base

L'élément de base à manipuler est le RDD : Resilient Distributed Dataset. Il s'agit d'une abstraction de collection sur laquelle les opérations sont réalisées de manière distribuée tout en étant tolérante aux pannes matérielles. Le traitement que l'on écrite semble ainsi s'exécuter au sein de notre JVM mais il sera découpé pour s'exécuter sur plusieurs nœuds. En cas de perte d'un nœud, le sous traitement sera automatiquement relancé sur un autre nœud par le framework sans impacté le résultat.

Les éléments manipulés par le RDD peuvent être des objets simples comme des String ou des Integer, nos propres classes ou plus couramment des tuples. Dans ce dernier cas les opérations offertes par l'API permettront de manipuler la collection comme une map clé-valeur.

L'API exposé par le RDD permet d'effectuer des transformations sur les données :

- *map()* permet de transformer un élément en un autre
- *mapToPair()* permet de transformer un élément en un tuple clé-valeur
- *filter()* permet de filtrer les éléments en ne conservant que ceux qui correspondent à une expression
- *flatMap()* permet de découper un élément en plusieurs autres éléments
- *reduce()* et *reduceByKey()* permet d'agréger des éléments entre eux.

Ces transformations ne s'exécuteront que si une opération finale est réalisée en bout de chaine. Les opérations finales sont :

- *count()* pour compter les éléments
- *collect()* pour récupérer les éléments dans une collection Java dans la JVM de l'exécuteur(dangereux en cluster)
- *saveAsTextFile()* pour sauver le résultat dans des fichiers texte.

MapReduce avec Spark

Avec Spark comme avec Hadoop, une opération de « Reduce » est une opération qui va permettre d'agréger les valeurs 2 à 2, en procédant par autant d'étapes que nécessaire pour traiter l'ensemble des éléments de la collection. C'est ce qui permet au framework de réaliser des agrégations en parallèle, éventuellement sur plusieurs nœuds.

Le framework va choisir 2 éléments et les passer à une fonction qu'il faut définir au préalable. La fonction doit retourner le nouvel élément qui remplacera les 2 premiers.

Il en découle que le type reçu en entrée doit être le même que celui en sortie de fonction : les valeurs doivent être homogènes. C'est nécessaire pour que les opérations soit répétée jusqu'à ce que l'ensemble des éléments ait été traité.

Le framework Spark présente donc l'intérêt d'être plus rapide qu'Hadoop et d'avoir un code plus compact et lisible.

Maven :

Apache Maven est un outil produit par la fondation Apache pour la gestion et l'automatisation de production de projets Java en générale et Java EE en particulier. L'objectif de ce projet est de produire un système comparable au Make sous Unix, soit produire un logiciel à partir de ses sources en optimisant les tâches réalisées à cette fin et en garantissant le bon ordre de fabrication.

Il s'agit d'un outil semblable à Ant mais qui offre des moyens de configuration plus simples, eux aussi basés sur le format XML.

Maven utilise un paradigme connu sous le nom de Project Object Model ou POM afin de décrire un projet logiciel, ses dépendances avec des modules externes et l'ordre à suivre pour la production. Il est livré avec un grand nombre de tâches prédéfinies, comme la compilation de code Java.

Ce POM se matérialise par un fichier pom.xml placé à la racine du projet. Ainsi on permet l'héritage des propriétés du projet. Ainsi si une propriété est redéfinie dans le POM du projet, elle recouvre celle qui est définie dans le projet parent.

Je me suis servi de cet outil pour pouvoir structurer mes différentes applications Hadoop ou Spark. Maven m'a donc permis de compiler mes programmes, de générer les .jar correspondant et de pouvoir placer les fichiers de tests correspondant à l'application au sein de sa structure pour faciliter l'exécution du programme.

Pig :

Le projet Pig est une plateforme de programmation de haut-niveau pour créer des programmes MapReduce utilisables avec Hadoop. Le langage utilisé par ce projet est appelé PigLatin. On peut ainsi écrire un

programme MapReduce sous forme de script. Cela peut faciliter grandement la réalisation des programmes MapReduce.

Le projet Pig a été développé à l'origine par la section de recherche de Yahoo vers 2006 pour permettre aux chercheurs de la compagnie de réaliser et d'exécuter des programmes MapReduce sur de larges ensembles de données de la manière dont le fait ad-hoc. En 2007, ce projet est repris par la fondation Apache.

Pig est considéré comme un outil excellent pour développer rapidement des jobs de type MapReduce à l'opposé du code classique d'un job MapReduce en Java.

Le projet Pig fonctionne de la façon suivante : chaque élément d'un ensemble de donnée est appelé un atome, une collection d'atome est un tuple. Ces tuples sont collectés dans des « sacs ». D'une manière générale, un script en Pig va charger 1 ou plusieurs ensembles de données dans un sac et créer des nouveaux sacs en les modifiants. On prend donc un sac en entrées, puis chaque ligne modifie les données dans ce sac puis on place ces nouvelles données dans un nouveau sac. Et ainsi de suite jusqu'à obtenir le résultat souhaité.

Hive :

Le projet Apache Hive est une infrastructure construite par-dessus Hadoop pour mettre en place un centre de données permettant de rassembler les données, réaliser des requêtes et faire des analyses sur les données stockées. Il s'agit d'un projet initialement développé par Facebook, Hive est maintenant utilisé et développé par d'autres compagnies comme Netflix.

Hive propose des mécanismes de structuration de projet dans les données et surtout permet de réaliser des requêtes en utilisant un langage proche du langage SQL appelé le HiveQL qui lit et convertit de manière transparente les requêtes en programme MapReduce. En même temps, ce langage propose aussi aux programmeurs map/reduce qui le souhaite de réaliser une programmation plus classique en apportant leurs propres mappeurs et leurs propres « reducers » lorsque le langage HiveQL ne convient pas ou devient inefficace.

Donc Hive permet de réaliser des requêtes comme dans des bases de données classiques mais pour des volumes de données bien plus importants. Il permet également de réaliser des jobs MapReduce de façon plus simple est plus rapide que de réaliser un programme complet. Cependant, ce projet peut se révéler inefficace si le job à réaliser devient trop compliquer. On le considère de plus comme étant un outil d'exécution longue et, non en temps réel. Il est excellent pour analyser et gérer des données ressemblant à celle gérer habituellement par les requêtes SQL classique.

Recherches et Expérimentations

Au début du stage, Le Big Data était pour moi un domaine inconnu que nous n'avions pas abordé durant ma formation à l'IUT. J'ai donc dû commencer mon stage par réaliser un certain nombre de recherches sur ce qu'est le Big Data, les technologies qui tournent autour et notamment acquérir des connaissances sur le framework Hadoop.

Généralités sur le Big Data

Le Big Data, appelé littéralement « grosses données » ou encore données massives, désigne des ensembles de données tellement volumineux qu'ils en deviennent difficile à travailler avec des outils classiques de gestion de base de données ou de l'information. Pour pouvoir manipuler avec ce nouvel ordre de grandeur, la capture, le stockage, la recherche, le partage, l'analyse et la visualisation de toutes ces données doivent être revues. Les perspectives du Big Data sont énormes et encore insoupçonnées.

En effet, chaque jour, l'humanité génère près de 2,5 trillions d'octets de données. A tel point que 90 % des données dans le monde ont été créées au cours des deux dernières années. Ces données proviennent de partout : de capteurs utilisés dans les différents domaines de la recherche : astronomique, climatique, médical,… mais également des messages sur des médias sociaux notamment sur Facebook ou Twitter, et l'ensemble des fichiers numériques, images, vidéos, publiées en ligne. Il y a bien évidemment beaucoup d'autres sources de données, et c'est ce qui explique la masse de données produite chaque jour.

La notion du Big Data est donc une combinaison de progrès technologiques, d'innovations d'usage et d'évolutions sociales qui amène les entreprises à repenser leurs priorités. La première dimension fondamentale du Big Data c'est la composante technologique. En effet, le Big Data s'appuie sur un ensemble d'innovations qui transforment la façon dont les entreprises et les individus génèrent, transmettent, stockent et utilisent des données. Cela passe par la massification des échanges de données, la révolution des systèmes de stockage (cloud) et la structuration des données,…

Cependant la montée en puissance du Big Data n'est pas essentiellement due à la technologie. Il y a aussi les évolutions culturelles vis-à-vis de la génération et du partage d'information et les nouveaux usages et nouvelles possibilités de monétisation qui sont des éléments clés de l'augmentation du volume de données.

Le Big Data se définit par 3 dimensions également appelés les 3V :

- Le Volume des données stockées aujourd'hui est en pleine expansion les données numériques créées dans le monde seraient passées de 1,8 zettaoctets en 2011 à 2,8

zettaoctets en 2012 et d'après des estimations elles s'élèveront à près de 40 zettaoctets en 2020.

- La Variété des données est un réel défi. Il ne s'agit généralement pas de données relationnelles traditionnelles, ces données sont généralement brutes et non structurées. Les systèmes doivent donc être capable d'analyser toutes ces données diverses et variées.
- La Vélocité représente à la fois la fréquence à laquelle les données sont générées, capturées et partagées. Les systèmes doivent être capables de répondre aux attentes des utilisateurs, par exemple les systèmes mis en place pour la bourse et les entreprises doivent être capables de traiter ces données avant qu'un nouveau cycle de génération ne commence, avec un risque de perte d'une partie des données. Les systèmes doivent être plus rapides pour permettre d'avoir toutes l'information nécessaire à temps et surtout ne pas avoir de pertes de données.

Ces domaines d'application :

Le Big Data trouve une application dans de nombreux domaines, dans de grands programmes scientifiques, dans des entreprises, dans des laboratoires,…

Les méthodes actuelles de modélisation de données ainsi que les systèmes de gestion de base de données ont été conçus pour des volumes de données très inférieur. Il ne s'agit pas ici de faire des requêtes mais plutôt une fouille de données. Dans le futur il faudrait des modélisations et des langages de requêtes permettant :

- une représentation des données en accord avec les besoins de plusieurs disciplines scientifiques ;
- de décrire des aspects spécifiques à une discipline (modèles de métadonnées) ;
- de représenter la provenance des données ;
- de représenter des informations contextuelles sur la donnée ;
- de représenter et supporter l'incertitude ;
- de représenter la qualité de la donnée

Les bases de données relationnelles classiques ne permettant pas de gérer les volumes de données du Big Data, de nouveaux modèles de représentation permettent de garantir les performances souhaitées. Ces technologies dites de Business Analytics & Optimization (BAO) permettent de gérer des bases massivement parallèles. Des patrons d'architecture sont proposés par les acteurs du marché du Big Data comme MapReduce développé par Google et utilisé dans le framework Hadoop. Ce système permet de séparer les requêtes et de les distribuer à des nœuds parallélisés, puis de les exécuter en parallèle. Les résultats sont ensuite rassemblés et récupérés.

Débuts avec Hadoop

L'objectif du stage est d'apprendre à utiliser le framework Hadoop pour pouvoir réaliser des applications de gestion de très larges volumes de données. Pour cela, il est nécessaire de comprendre les fondements de ce framework, notamment l'utilisation du patron méthode MapReduce qui est à la base de son fonctionnement.

Hadoop est donc un framework de calcul distribué inspiré du paradigme fonctionnel. A l'aide de son système de fichier, le HDFS, Hadoop va répartir les données sources entre les différents nœuds du cluster, sous forme de blocs, et va ensuite appliquer des méthodes *map()* et *reduce()* sur chacun de ces nœuds de manière parallélisée. Toutes ces actions sont faites de manière transparente, c'est-à-dire que le programmeur n'a pas la main sur la répartition des données entre les nœuds au moment de l'exécution.

Le programmeur Hadoop aura donc pour but principal de configurer son cluster, écrire le code des méthodes *map()* et *reduce()* et paramétrer le job. Cependant, la mise en œuvre de certains jobs peut s'avérer difficile à réaliser et présente l'inconvénient d'obliger le programmeur Hadoop à écrire des codes compliqués même lorsqu'il s'agit d'exécuter une simple requête.

Choix et utilisation d'une distribution Hadoop

Actuellement plusieurs distributions existent, permettant une installation plus facile du framework Hadoop mais aussi une programmation plus facile et plus rapide de jobs MapReduce. Actuellement, il en existe 3 importantes :

- HortonWorks
- MapR
- Cloudera

Pour pouvoir programmer plus simplement avec Hadoop il est donc nécessaire d'en choisir au moins une.

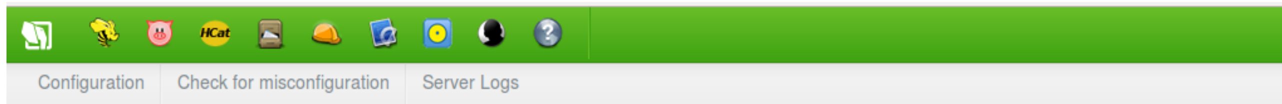
Mon choix s'est porté sur la distribution HortonWorks qui est l'une des plus réputées actuellement. Pour mettre cette distribution en œuvre j'ai téléchargé la version Sandbox qui offre une configuration de Hadoop à installer sur une machine virtuelle. Il est possible d'accéder aux différents projets proposés par la fondation, comme Pig, Hive ou encore HCatalog, par l'intermédiaire d'un portail web.

Ce portail web est relativement facile d'utilisation et permet donc à un programmeur non-initié à la programmation MapReduce de réaliser des jobs pour la gestion de larges volumes de données.

De plus j'ai également tenté d'installer une distribution en local sur ma machine. Cependant, cette tâche s'est avérée plutôt ardue pour diverses raisons, notamment les tutoriels que j'ai utilisés n'étaient pas forcément adaptés à ma version de Hadoop ou encore à mon système d'exploitation. Finalement j'ai opté pour une version stable sans distribution mais associé avec l'IDE eclipse.

Utilisation de l'interface HortonWorks, les projets Pig et Hive

L'interface web proposé par la distribution HortonWorks se présente de la façon suivante :



Hortonworks Sandbox with HDP 2.2

Leave Feedback

Component	Version	
Hue	2.6.1-2	
HDP	2.2.4	
Hadoop	2.6.0	
Pig	0.14.0	
Hive-Hcatalog	0.14.0	
Oozie	4.1.0	
Ambari	2.0-151	Disable
HBase	0.98.4	
Knox	0.5.0	
Storm	0.9.3	
Falcon	0.6.0	
Sandbox Build	f1dc3df 09:23 03-04-15	

On peut voir sur la partie supérieure de l'écran différents onglets permettant de naviguer vers les différents projets proposés par la distribution notamment Hive, Pig, HCatlog et bien d'autres. Ici il va être question de présenter le fonctionnement de cette plateforme afin de voir ce qu'il est possible de réaliser avec les outils de programmation de jobs MapReduce. L'objectif de ces divers projets est de proposer au programmeur non spécialisé dans la programmation avec Hadoop de pouvoir mettre en œuvre des jobs MapReduce.

Pour travailler avec cette interface et notamment avec Pig et Hive les 2 projets qui vont être présentés, il faut tout d'abord charger les fichiers sources dans le système de fichiers. Pour cela il faut se rendre dans l'onglet nommé « File Browser ».

Cet onglet permet de voir l'ensemble des fichiers chargés dans le système de fichiers et notamment de les consulter. Il est possible de changer leurs noms, de les supprimer,… en d'autres termes de réaliser un certain nombre d'opérations de gestion pour nos fichiers sources. Il est également possible de déposer les fichiers sources que l'on souhaite.

Une fois notre fichier source chargé dans le système de fichiers, il faut le transformer en base de données relationnelle pour pouvoir travailler avec Hive et Pig. Pour cela le projet HCatalog offre la possibilité de choisir un fichier présent dans le système de fichiers puis de produire une base de données relationnelle à partir du fichier source.

On doit renseigner le nom de notre table, le chemin dans le système de fichiers menant à la source. HCatalog propose alors un aperçu de la base en bas, notamment la séparation des données ainsi que le nom des champs. Il est possible de choisir le séparateur qu'utilise notre fichier, par exemple les virgules, les tabulations ou encore les espaces. Il faut faire attention à ce que le fichier que l'on souhaite utiliser ait ses différentes données délimitées par des séparateurs connus par HCatalog sinon, il est possible de rencontrer des problèmes durant la réalisation.

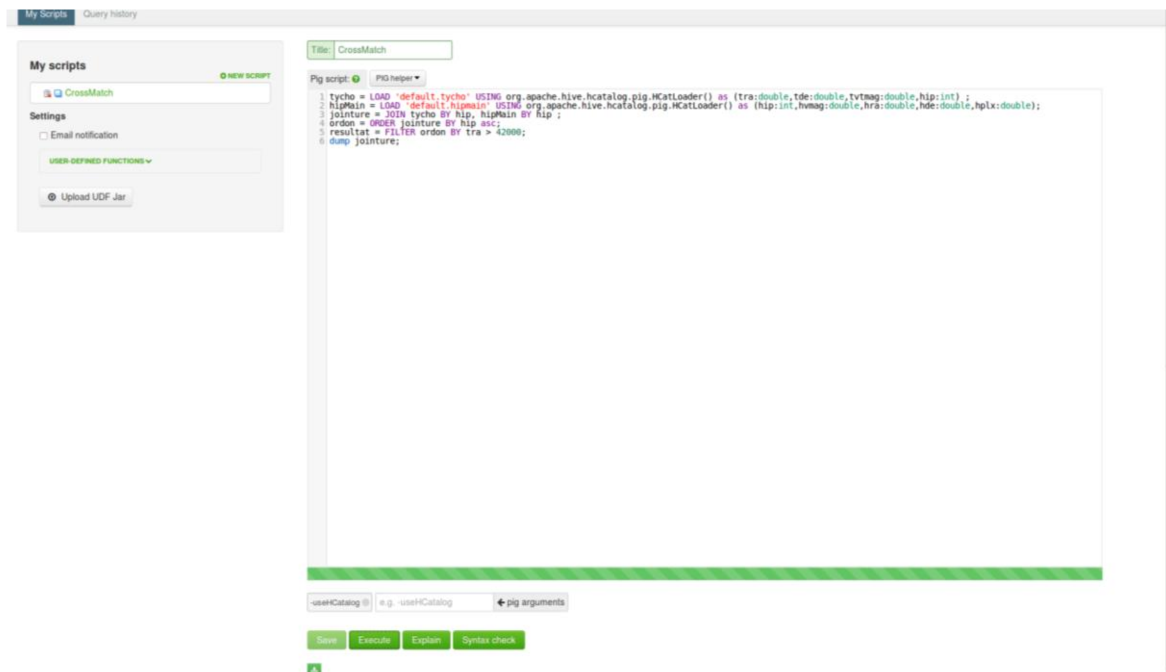
Une fois la base ainsi créée, il est possible de travailler sur cette source en utilisant les projets Hive et Pig.

Essais avec Pig :

L'interface de travail du projet Pig se présente de la façon suivante :

Une large zone de travail dans laquelle on peut écrire son script. Un champ de titre au-dessus ainsi qu'un champ d'arguments. Une série de boutons : Save, Execute, Explain ou encore Syntax Check situés en dessous de la zone de travail et qui permettent de sauvegarder un job, l'exécuter ou de vérifier que la syntaxe est correcte. En-dessous du champ de titre se trouve

un menu déroulant dans lequel est proposé une auto-implémentation des différentes fonctions réalisables avec Pig, cela simplifie en grande partie le travail du programmeur car il a ainsi accès à une syntaxe correcte des fonctions qu'il souhaite utiliser. Sur le côté gauche on observe une zone dans laquelle sont stockés les différents scripts réalisés. On peut également associer des jar au besoin. Il s'agit donc d'une interface de travail simple à comprendre et donc à utiliser.



La programmation de job MapReduce avec Pig utilise le langage de script nommé PigLatin. Ce langage offre la possibilité au programmeur non habitué à la programmation MapReduce avec Java de réaliser des jobs à partir d'un langage de script plus facile à prendre en main.

Ce langage présente un certain nombre de fonctions qu'il est possible d'utiliser directement. Notamment des fonctions d'opération sur les tables comme Filter, Join, Union, foreach, Group .. by, Order , … Ou encore des fonctions de calcul comme Avg, Concat, Diff, Max, Size, Sum,… En d'autres termes le programmeur Pig a accès à un très large choix de fonctions pour réaliser ses jobs MapReduce. On remarque également que PigLatin propose un langage réalisant des opérations proches du langage SQL. C'est pourquoi ce langage est recommandé pour des programmeurs souhaitant faire des opérations classiques de jointure, filtre ou calcul de moyenne.

En effet, l'utilisation de Pig pose quelques soucis lorsqu'il s'agit de réaliser des jobs MapReduce plus complexes. De plus bien, qu'il est possible d'exécuter rapidement des programmes sur de larges volumes de données, ceci reste toutefois assez long car Pig se doit de convertir le programme du PigLatin en job MapReduce puis de l'exécuter. Ce qui pour certains jobs donne un temps d'exécution supérieur à un temps d'exécution d'un programme fait avec Hadoop en Java.

Exemple d'un script réalisé avec Pig :

```
Tycho = LOAD 'default.tycho' USING org.apache.hive.hcatalog.pig.HCatLoader()
AS (ra  : double , de : double , vmag  : double , hip : int ) ;
HipMain = LOAD 'default.hipMain' USING
org.apache.hive.hcatalog.pig.HCatLoader()
AS  (hip : int , vmag  : double, ra : double , de : double, plx : double) ;
Jointure = JOIN Tycho BY hip, HipMain BY hip ;
Ordon = ORDER Jointure BY hip asc;
Resultat = FILTER Ordon BY ra > 42000;
DUMP resultat;
```

La portion de code ci-dessus réalise les actions suivantes :

Le fichier tycho présent dans le HCatalog est d'abord chargé puis stocké dans la variable Tycho par la commande LOAD, tout en configurant le schéma relationnel à l'aide du AS écris après, c'est-à-dire donner le noms des colonnes ainsi que leur type ; cela facilite l'utilisation des tables pour réaliser des opérations. Il est fait ensuite de même pour le fichier hipMain. Une jointure des 2 fichiers par la valeur de la colonne hip est réalisé ligne 3. Ligne 4 les fichiers sont mis dans l'ordre croissant par la commande ORDER. La commande FILTER va filtrer les résultats et enlever tout ceux dont la valeur de la colonne 'ra' est inférieure à 42000. Enfin la commande DUMP affiche les données présentes dans la variable nommée ici résultat. Il existe bien évidemment plein d'autres possibilités de réalisation.

Il faut savoir que pour pouvoir charger les fichiers à partir du HCatalog , il faut ajouter l'argument : « -useHCatalog » dans le champ des arguments sinon le programme ne parviendra pas à trouver les fichiers sources.

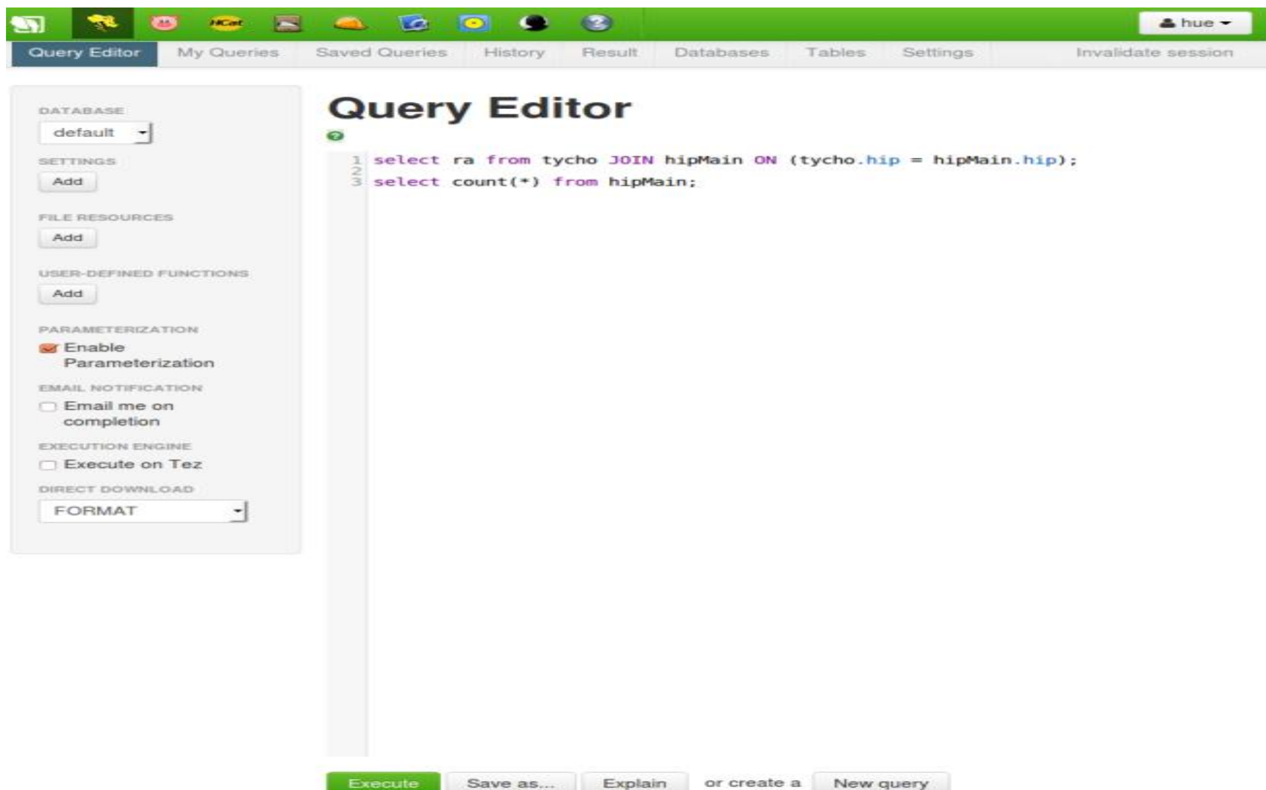
Dans l'exemple ci-dessus, j'affiche les valeurs contenu dans « résultat » il est possible également de décrire le schéma relationnel par la commande DESCRIBE ou encore d'écrire le résultat dans un nouveau fichier à l'intérieur du système de fichier par la commande STORE :

```
DESCRIBE résultat;
STORE résultat INTO '<nomdufichier>';
```

Essais avec Hive :

Comme on peut le voir sur l'image ci-dessous, l'interface de travail du projet Hive proposé par HortonWorks présente des similitudes avec l'interface de Pig. Tout d'abord, une large zone de travail où écrire son code, des boutons en bas de l'écran pour exécuter, sauvegarder, …. A gauche un menu de configuration du job que l'on veut lancer. Ce projet offre la possibilité aux programmeurs non-initiés à la programmation MapReduce de se servir d'un langage proche du langage SQL, nommé HiveQL. Ce langage offre des possibilités d'opération équivalentes à ce que pourrait permettre un langage de base de données classique, soit des jointures, des unions, des différences mais également des calculs de moyennes, de la

taille, des sommes,…il est possible de réaliser des opérations visant à modifier, créer ou supprimer des bases dans le HCatalog tout comme en SQL.



Ce projet à l'instar du projet Pig, offre la possibilité de programmer un job MapReduce plus simplement que de le réaliser avec du code Java. Cependant, tout comme Pig, Hive possède les inconvénients d'être presque inefficace lorsqu'il faut réaliser un programme trop complexe et ce d'un point de vue écriture de code mais également de temps d'exécution.

Il faut toutefois savoir que Hive a des limites :

- Tous les standards du SQL ne sont pas supportés
- Le langage ne supporte pas les Update et les Delete
- Le langage ne supporte pas les insertions d'une seule ligne
- Le langage est relativement limité en nombre de fonctions à construire
- Il n'y a pas de type pour les dates ou pour le temps

Quelques exemples de code Hive :

```
SELECT hip FROM tycho WHERE ra > 42000 LIMIT 100;
```

Sélectionne 100 valeurs de la colonne hip de la table tycho quand les valeurs de la colonne ra sont supérieures à 42000.

```
CREATE TABLE etoile (ra DOUBLE, de DOUBLE, hip, INT)
ROW FORMAT DELIMITED
```

FIELDS TERMINATED BY \t
STORES AS TEXTFILE

Création d'une table nommée « etoile » avec 3 attributs, « ra » un double, « de » un double et « hip » un int, les valeurs seront sous forme de ligne délimitées, chaque valeurs sera séparée par des tabulations et le fichier sera sauvegardé en tant que fichier texte.

Fonctionnement d'un job MapReduce avec Hadoop

Une fois une distribution Hadoop mise en place, il est possible de commencer à réaliser de premiers jobs MapReduce. Pour lancer l'exécution d'un programme Hadoop, il faut tout d'abord créer un jar qui contiendra notre application. Pour cela, il m'a fallu utiliser l'outil Maven qui permet de créer et configurer des projets Java et ainsi pouvoir gérer les imports et générer un .jar.

Pour cela il faut mettre en place la structure des répertoires puis placer à la base de l'arborescence un fichier que l'on nomme pom.xml est qui est donc un fichier xml permettant de configurer le projet Maven. Ce fichier contient une description détaillée du projet, avec plus particulièrement des informations concernant le versionnage et la gestion des configurations, les dépendances, les ressources de l'application, les tests, les membres de l'équipe, la structure,…

Une fois le projet Maven configuré, il suffit de mettre les sources java et les fichiers de tests au bon endroit dans l'arborescence et ensuite d'utiliser la commande :

mvn package

Cela va compiler les sources et générer un jar dans un dossier target qui sera placé à la base du projet. Lors de l'exécution du job MapReduce il suffira de donner à la commande le jar généré par le projet Maven.

Premier programme avec Hadoop

Un job MapReduce se réalise de la façon suivante. Lorsque l'on exécute son programme sur le cluster, les données sources passées au programme vont être découpé sous forme de bloc et répartis sur les différents nœuds du cluster. Une fois que les données sont toutes réparties, on commence à appliquer le programme sur chacun des nœuds de façon parallèle. Une fois l'exécution terminé sur un nœud, celui-ci prévient son nœud maître que son travail est terminé. Le résultat produit par le job est déposé dans un dossier de sortie.

Un programme MapReduce en Java se découpe en différentes phases :

- Une phase « map » qui sera chargé de transformer chaque ligne de valeur en un couple clé/valeur
- Une phase de tri et de combinaison pour rassembler les valeurs associés à une même clé
- Et une phase « reduce » qui prendra une clé et une collection de valeurs associées et qui produira un résultat en sortie

Il faut savoir cependant, que le framework Hadoop se charge du travail de répartition des données et de l'exécution des différentes phases. De plus, il a déjà des méthodes de Tri et de combinaison qu'il exécute de lui-même dès que la tâche « map » est terminée.

En d'autres mots le programmeur MapReduce avec Hadoop va devoir écrire : une fonction *map()*, une fonction *reduce()* et un main, aussi appelé driver, qui sera chargé de configurer et d'exécuter le job. Cependant, il est également possible de réaliser ses propres fonctions de Tri et de Combinaison, je ne l'ai pas fait dans un souci de simplicité et parce que les fonctions utilisés par Hadoop sont bien suffisantes.

Déroulement d'un job MapReduce avec Hadoop :

Afin de mieux comprendre le fonctionnement du job, je vais ici me baser sur un exemple simple de job, le WordCount.

Le WordCount est un programme dont l'objectif est de compter le nombre d'occurrence des différents mots dans un texte. Il s'agit d'un premier job facile à réaliser et qui utilisera toutes les facettes des jobs MapReduce

Tout d'abord Hadoop va récupérer les fichiers sources puis va les découper en blocs qui vont être ensuite répartis sur les différents nœuds du cluster. Il s'agit de l'opération de Split.

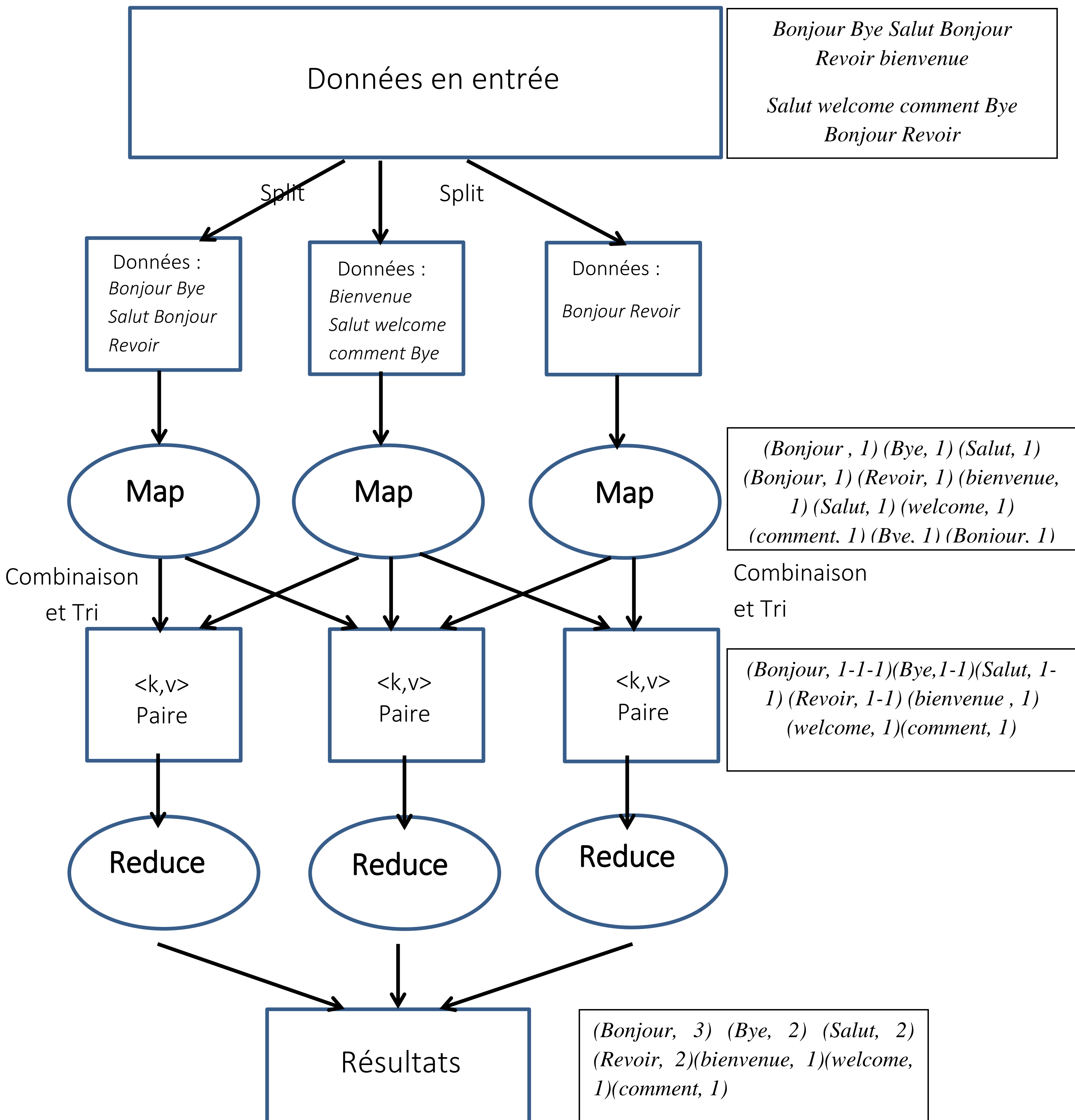
Une fois que toutes les données sont sur les nœuds, Hadoop lance l'exécution de la fonction *map()* sur chacun des nœuds. La fonction *map()* va prendre chaque ligne de données et à partir de ces lignes va générer des couples de clé/valeur, dans notre exemple il s'agit d'un couple <mot, 1>.

Dès que la tâche « map » est terminée sur tous les nœuds, Hadoop va lancer une opération de Combinaison et de Tri qui a pour objectif de combiner à l'intérieur d'un Itérateur toutes les valeurs dont les clés sont communes. Dans notre exemple, cette action va rassembler un certain nombre de « 1 », en fonction du nombre de fois qu'apparait le mot qui est la clé.

Ensuite Hadoop lance l'exécution de la méthode *reduce()* qui va prendre une clé et l'Itérateur associés et qui va parcourir cette liste de valeurs et générer un résultat, dans notre cas le « reducer » fait la somme de tous les « 1 » présents dans l'Itérateur pour donner en résultat le mot ainsi que le nombre d'occurrences de celui-ci. Lorsque toutes les opérations de « Reduce » sont terminées, Hadoop rassemble tous les résultats et les stocke dans le fichier de sortie.

Voici donc un exemple simple du fonctionnement d'un Job MapReduce en utilisant le framework Hadoop. (cf Annexe 1)

Schéma représentatif du job WordCount avec Hadoop



Découverte de Apache Spark

Au bout de quelques semaines de stage, mon tuteur ma présenté au cours d'une de nos discussions le framework Apache Spark. Il s'agit d'un framework du BigData produit par la fondation Apache en 2014. Il s'agit d'un outil proposant des performances d'exécution jugées 100 fois plus rapide que ce que permettait Hadoop. Suite à cette discussion j'ai décidé de me lancer dans la découverte de ce framework jugé plus rapide et plus simple que Hadoop. Ainsi je pourrais avoir une vue sur une autre technologie du Big Data susceptible de convenir à l'Observatoire.

Il faut notamment savoir que Spark n'a pas vocation à remplacer Hadoop mais vise plutôt son amélioration. En effet, il est possible d'utiliser entre autres le HDFS de Hadoop comme système de fichier. L'installation de Spark s'est avéré bien plus simple que celle d'Hadoop, cela s'est limité à désarchiver un .tar puis à installer le framework.

Le framework Spark propose l'utilisation de 3 langages de programmation : Java, Scala ou Python ; de plus il est possible d'utiliser un shell en scala ou en Python pour programmer en directe sur le framework. En ce qui concerne le code et la façon de l'écrire, Spark est très différent d'Hadoop. En effet, Spark ne s'inspire pas vraiment du patron MapReduce pour sa programmation. Il n'y a pas de méthode *map()* ou *reduce()*, il même plutôt de rigueur de tout faire dans le main.

J'ai donc réalisé comme pour Hadoop un certain nombre d'essais visant à maîtriser le framework et l'API qui y est associée. J'ai également utilisé l'outil Maven pour structurer mes projets et pouvoir générer des .jar facilement.

Premiers essai avec Apache Spark :

Le code Spark se réalise de la façon suivante : tout d'abord il faut configurer le job en donnant son nom, ensuite il faut charger les fichiers dans des *JavaRDD* en utilisant la méthode de la classe *Context* de l'API Spark, textFile() qui va charger le fichier dont le chemin est en paramètre.

Les *JavaRDD* sont des objets représentants des variables spéciales du système de fichier de Spark. Puis il faut utiliser des fonctions comme *flatMap* ou *mapToPair* qui prennent en paramètre une classe. La particularité de l'API Spark, c'est qu'il existe des classes *Function*, Function2, Function3 ou encore *FlatMapFunction* qui sont des classes dont on peut choisir le type des paramètres et de la sortie. Par exemple il est possible d'utiliser une *Function<String,String>* qui prendra donc en paramètre un String et qui retournera un String. A l'intérieur de cette classe se trouve une méthode call qui va être appelée pour chaque élément du RDD, et c'est cette fonction qui fera le traitement souhaité.

Une fois que les différents traitements réalisés sur nos fichiers il suffit d'utiliser une méthode de la classe *JavaRDD, saveAsTextFile(<chemin>)* qui va écrire le contenu de la variable dans le fichier dont le chemin est donné en paramètre.

Premier job : Compare

Il s'agit d'un job très simple qui va charger 2 fichiers possédant le même schéma relationnel et qui va donner en résultat l'intersection de ces 2 fichiers. Ce job permet de voir les bases du chargement, l'application d'un premier traitement et la sauvegarde du fichier en sorti.

La première étape consistera donc a chargé chacun de ces 2 fichiers dans 2 variables de type *JavaRDD* grâce à la méthode *textFile()*, ainsi j'obtiens 2 collection contenant mes données séparé par des lignes.

Ensuite je parcours ces collections pour pouvoir découper les données comme je le souhaite, notamment l'utilisation de la fonction *flatMap()* qui permet de transformer chaque élément de la collection en 1 ou plusieurs autres éléments :

```
JavaRDD<String> words = file.flatMap(new FlatMapFunction<String,String>(){
    public Iterable<String> call(String s){
            return Arrays.asList(s.split("\n"));
    }       }
);
```

Une fois ces traitements réalisés sur mes 2 collections, je peux réaliser mon intersection en utilisant la méthode de l'API Spark intersection, qui réalise donc une intersection de 2 *JavaRDD*. Enfin j'utilise la méthode *saveAsTextFile()* qui permet donc décrire le contenu d'une variable *JavaRDD* dans un dossier de sortie.

Second job : Filtre

Il s'agit d'un job un peu plus complexe dont l'objectif est de prendre 2 catalogues fournit par l'observatoire, de les filtrer suivant une valeur puis de réaliser une union. Avec ce job j'ai pu aller un peu plus loin dans l'utilisation des fonctions fournit par l'API Spark.
Tout comme dans l'exemple précédant, je dois d'abord charger mes différents fichiers en entrées dans des *JavaRDD* pour pouvoir ensuite les traiter.

Ici je récupère également 2 valeurs passées en argument pour pouvoir les utiliser pour filtrer mes données, ces variables sont du type final pour pouvoir être utilisées au sein des classes internes.
Comme dans l'exemple précédent je réalise un traitement sur mes fichiers pour pouvoir les découper et récupérer les valeurs que je souhaite.

Je réalise à présent mon opération de filtration, la fonction utilisée est la fonction *filter* qui prendra en paramètre une *Function<String,Boolean>*, c'est-à-dire qu'elle prend en entrée un String et retourne un booléen, si vraie la valeur courante est conservée si faux la valeur n'est pas conservée.

```
JavaRDD<String> tychofiltre = tycho.filter(new Function<String,Boolean>(){
        public Boolean call(String s){
          String[] t = s.split(",");
          Double ra = Double.parseDouble(t[1]);
          return (ra < centre+rayon) && (ra >centre-rayon);
        }
      } );
```

La fonction split les valeurs pour pouvoir en récupérer une en particulier, je la convertis en Double pour pouvoir ensuite la comparer avec les valeurs que j'ai récupéré plus tôt.
Enfin je réalise une union de mes fichiers puis je les écris en sortie grâce à la fonction *saveAsTextFile()*.

Voici donc 2 essais que j'ai réalisés avec le framework Apache Spark est qui montre les différentes facettes de la réalisation d'un job avec Spark. (cf. Annexe 2)

Réalisation d'une application : CrossMatch

Après une discussion avec mon tuteur et avec Mr François-Xavier Pineau, un ingénieur de l'observatoire, il a semblé intéressant de réaliser une application de CrossMatch. Ce genre d'application est déjà utilisé par l'Observatoire mais par le biais d'une application réalisée par Mr. Pineau qui est une application en Java multi-threadées. L'objectif de la réalisation d'une application de CrossMatch avec Hadoop ou Spark permet de voir si les technologies actuelles du Big Data que je suis en train d'étudier peuvent éventuellement remplacer les solutions existantes actuellement à l'Observatoire.

Pour pouvoir réaliser mon application, j'ai demandé de l'aide à François-Xavier qui m'a expliqué les objectifs d'une telle application, qui m'a fournis des catalogues pertinents sur lequel je pourrais faire mes tests mais aussi une méthode, accompagné de sa librairie, pour que je puisse calculer un identifiant.

Objectif d'une application de CrossMatch

L'objectif d'une telle application est de réaliser une jointure sur 2 catalogues dont les schémas relationnels sont différents. Pour cela, il faut calculer un identifiant en se servant des valeurs de la « right ascension » et de la « déclinaison » qui se trouve dans les 2 catalogues. Pour obtenir l'identifiant il faut utiliser une fonction que François-Xavier m'a fournie et qui la calcule à partir de ces 2 valeurs. Il faut calculer cette identifiant pour chaque ligne de chaque catalogue. Puis réaliser une jointure des 2 catalogues en se servant de cette identifiant comme clé de jointure.

De plus, healpix est une fonction qui calcul les id des sources en suivant des pixels sous forme de losange. On a donc en sortis uniquement des sources contenu dans le losange traité. Il est intéressant de récupérer les voisins de certaines sources même si ceux-ci appartiennent à un autre losange. Il faut donc également récupérer ces sources.

Une application de CrossMatch demande également de filtrer une partie des données pour ne garder que la portion du ciel que l'on souhaite utiliser. Pour cela, il faut utiliser une fonction calculant la distance entre deux sources sur lesquelles il est possible de réaliser une jointure, si la distance est trop grande, il ne faut pas garder ces sources.

Descriptions des sources utilisées

J'utilise dans cette application 2 catalogues différents que m'a fournis François-Xavier. Ces catalogues ont été choisis car nous sommes sûr qu'une jointure entre ces 2 catalogues générerait des résultats.

- Le premier catalogue est Tycho, de la forme :

RA,DE,VTmag,HIP
45.034050, +0.235443, 8.411,13989
45.164941, +0.200411, 10.807,-1

Il s'agit d'un catalogue de référence contenant 2 539 913 valeurs dont les positions et les mouvements propres de 2.5 millions des étoiles les plus brillantes du ciel. Le catalogue possède les valeurs de right ascension, la déclinaison, la magnitude et hip qui est une id calculée à partir des 2 premières valeurs.

- Le second catalogue est HipMain, de la forme :

HIP, Vmag, RA, DE, Plx
1,9.1, 9.1185E-4, 1.08901332, 3.54
2,9.27, 0.00379737,-19.49883745, 21.9

Il s'agit d'un catalogue contenant 119 218 valeurs. Ce catalogue est l'un des premiers produits par l'European Space Agency's astrometric mission, Hipparcos. Ce satellite, qui a opéré pendant 4 ans a produit des données scientifique de grande qualité. Le catalogue présente les valeurs de l'identifiant calculées à partir de la right ascension et de la déclinaison, la magnitude, la right ascension, la déclinaison et la corrélation.

Réalisation de l'application avec Hadoop

J'ai tout d'abord réalisé cette application avec le framework Hadoop. Cependant, pour pouvoir la faire il m'a fallu réaliser un certain nombre de recherches au préalable car les jointures ne sont pas les actions les plus faciles à réaliser en MapReduce. Après des recherches, j'ai choisi une méthode de jointure parmi plusieurs autres, celle-ci en étant une simple mais restant efficace pour notre cas. (Cf. Annexe 3)

Notre code se construira de la façon suivante :

- 2 mappers, un pour chaque catalogue. Ils auront pour objectif de calculer les identifiants et de créer un couple clé/valeur pour chaque ligne de chaque catalogue, dont la clé sera l'identifiant calculé et la valeur sera la ligne courante du catalogue à

laquelle j'aurai ajouté une lettre pour être capable de déterminer son catalogue d'origine dans la suite du programme.

- 1 reducer, qui va récupérer les clés et leurs listes de valeurs associée et qui réalisera une jointure sur ces valeurs.
- 1 driver qui configurera le job

TychoJoinMapper (Cf. Annexe 3.1) :

Mon TychoJoinMapper est composé de deux éléments : ma fonction *map()*, et une fonction *nestIdx()*. Cette dernière est la fonction qui m'a été fourni par l'Observatoire. Celle-ci prend 2 doubles en entrée, calcul l'identifiant puis le renvoi en sortie.

Ma fonction *map()* va tout d'abord, découper la ligne pour récupérer chaque valeur séparément. Ensuite je récupère la valeur de right ascension et de déclinaison contenu dans le tableau de String que je viens d'obtenir. Je convertis mes String en double pour récupérer leurs valeurs. Grâce à la fonction *nestIdx()* je calcule l'identifiant en passant en paramètre la right ascension et la déclinaison de la valeur courante. Je prends cet identifiant est je l'affecte en tant que clé de mon couple puis j'ajoute à la ligne courante, qui est un String, la lettre « A » afin de pouvoir reconnaître durant la phase « reduce » les valeurs appartenant au catalogue Tycho.

```
public abstract class TychoJoinMapper extends MapReduceBase implements Mapper {
    private Text outkey = new Text();
    private Text outvalue = new Text();

    private final int nside = 2048;
    private  HealpixBase hb;
    public long nestIdx(final double raDeg, final double decDeg) {
        try {
            this.hb = new HealpixBase(nside, Scheme.NESTED);

            return this.hb.ang2pix(new Pointing(
                Math.toRadians(90 - decDeg),
                Math.toRadians(raDeg)));
        } catch (Exception e) {
            throw new Error(e);
        }
    }


    public void map(Object key, Text value, Context context) throws IOException {
        //decoupage de la ligne pour pouvoir recuperer chaque valeur
                String[] parsed = value.toString().split(",");
                //recuperation de la valeur ra et de
                String ra = parsed[0];
                String dec = parsed[1];
                //conversion des String en Double
                double raDeg = Double.parseDouble(ra);
                double decDeg = Double.parseDouble(dec);
                //calcul de l'id
                long id = nestIdx(raDeg,decDeg);
                //on passe l'id comme clé du couple
                outkey.set(id+"");
                //on passe la valeur de la ligne + A pour désigner la src d'origine
                outvalue.set("A" + value.toString());
                //on passe le couple valeur en sortie
                try {
                    context.write(outkey,outvalue);
                } catch (InterruptedException e) {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
                }
```

HipMainJoinMapper(Cf. Annexe 3.2) :

Le HipMainJoinMapper réalise la même tâche que le TychoJoinMapper à la différence qu'il ne travaille pas sur le même catalogue. En effet, les 2 catalogues ne sont pas structurés à l'identique, il m'a donc fallu en réaliser un par catalogue pour pouvoir récupérer les valeurs de right ascension et de déclinaison. Par exemple, la valeur de la déclinaison se trouve en 2ème place pour le catalogue Tycho alors qu'elle est en 4ème position pour le catalogue HipMain. (Voir « Description des sources utilisées »)

```
public class HipMainJoinMapper extends MapReduceBase implements Mapper {
    private Text outkey = new Text();
    private Text outvalue = new Text();
    private static final int nside = 2048;
    private static HealpixBase hb;
    static{
        try {
            hb = new HealpixBase(nside, Scheme.NESTED);
        } catch (Exception e) {
            throw new Error(e);
        }
    }
    public static long nestIdx(final double raDeg, final double decDeg) {
        try {
            return hb.ang2pix(new Pointing(
                Math.toRadians(90 - decDeg),
                Math.toRadians(raDeg)));
        } catch (Exception e) {
            throw new Error(e);
        }
    }
  @Override
    public void map(Object key, Object value, OutputCollector context,
            Reporter arg3) throws IOException {
        String[] parsed = value.toString().split(",");
        if(parsed.length>4){
        String ra = parsed[2];
        String dec = parsed[3];
            double raDeg = Double.parseDouble(ra);
            double decDeg = Double.parseDouble(dec);
            //calcul de l'id
            long id = nestIdx(raDeg,decDeg);
            //on passe l'id comme clé du couple
            outkey.set(id+"");
            //on passe la valeur de la ligne + A pour désigner la src d'origine
            outvalue.set("B" + value.toString());
            //on passe le couple valeur en sortie
             context.collect(outkey,outvalue);
            try {
                for (long nei : hb.neighbours(id)) {
                    outkey.set(nei+"");
                    context.collect(outkey,outvalue);
                }
            } catch (Exception e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
```

Il y a cependant une différence entre mes 2 Mappers. En effet, on peut observer que après avoir écrit mon premier couple dans le HipMainJoinMapper, je réaliser un autre traitement. Ce traitement va utiliser une fonction de la librairie healpix fournit par l'Observatoire. Cette fonction va prendre mon id précédemment calculer et retourner un tableau d'id qui sont ces voisins. En effet, il faut savoir que healpix est une fonction qui travaille sur des pixels sous forme de losange, et si l'un de ses voisins est en dehors de ce losange il faut tout de même le récupérer. C'est le but de ce traitement. Je parcours donc ma liste d'identifiant voisin puis j'écris des couples pour chacun de ces voisins, la clé est l'identifiant, la valeur est toujours celle de la ligne courante.

Phase de Shuffle et Sort :

Dès que toutes les tâches de « Mapper » seront achevées, Hadoop va réaliser une opération de Combinaison pour rassembler les valeurs possédant le même identifiant. Cette opération est importante car c'est en fait à ce moment que la jointure va se faire. En effet, si toutes les valeurs ayant la même clé sont rassemblées dans une liste, il ne reste plus qu'à vérifier d'où viennent chacune d'entre elles puis d'écrire les valeurs en sorties Après l'opération de Combinaison Hadoop fait également une opération pour trier les clés.

CrossMatchReducer (Cf. Annexe 3.3) :

C'est dans mon « reducer » que va se réaliser ma jointure. Je rappelle que mon « reducer » à en paramètre une clé et toutes les valeurs associées à cette clé. Je construis donc 2 listes, listA et listB que je vide à chaque début de ma fonction *reduce()*. Dans cette fonction je parcours ma liste de valeurs dans le but d'identifier leurs catalogues d'origine. Le traitement réaliser dans les mapper me permet de les identifier plus facilement. Une fois que j'ai déterminé l'origine de ma valeur courante je la place dans l'une des 2 listes, listA si elle vient de Tycho, listB si elle vient de HipMain. Une fois toutes les valeurs réparties je peux lancer ma fonction *executeJoinLogic()* qui va réaliser la jointure.

Ma méthode *executeJoinLogic()* va parcourir ensuite les 2 listes afin de réaliser la jointure. Cependant elle va avoir également pour rôle de filtrer les valeurs. Pour cela, je récupère les

```
public class CrossMatchReducer extends MapReduceBase implements Reducer {
    private ArrayList<Text> listA = new ArrayList<Text>();
    private ArrayList<Text> listB = new ArrayList<Text>();
    private Text tmp = new Text();
    private String joinType = null;
    public static double havDist(final double ra1Deg, final double dec1Deg,final double ra2Deg, final double dec2Deg) {
        final double d = Math.sin(Math.toRadians(0.5 * (dec2Deg - dec1Deg)));
        final double a = Math.sin(Math.toRadians(0.5 * (ra2Deg - ra1Deg)));
        final double h3 = d * d + a * a * Math.cos(Math.toRadians(dec1Deg)) * Math.cos(Math.toRadians(dec2Deg));
        return Math.toDegrees(2.0 * Math.asin(Math.sqrt(h3)));
    }
    public static final double xmatchDistance = 5.0/3600.0  ;
    public void setup(Context context){
        joinType = context.getConfiguration().get("join.type");
    }
    private void executeJoinLogic(OutputCollector context) throws IOException, InterruptedException {
        if(!listA.isEmpty() && !listB.isEmpty()){
            for(Text A : listA){
                String[] a = A.toString().split(",");
                double ra1 = Double.parseDouble(a[0]) ;
                double dec1 = Double.parseDouble(a[1]) ;

                for(Text B : listB) {
                    String[] b = B.toString().split(",");
                    double ra2 = Double.parseDouble(b[2]);
                    double dec2 = Double.parseDouble(b[3]);
                    double distance = havDist(ra1,dec1,ra2,dec2);
                    if (distance < xmatchDistance) {
                        context.collect(A,B);
                    }
                }}
        }}
    @Override
    public void reduce(Object key, Iterator values, OutputCollector context,
            Reporter arg3) throws IOException {
                listA.clear();
                listB.clear();
                while(values.hasNext()){
                    tmp = (Text) values.next();
                    if(tmp.charAt(0) == 'A'){
                        listA.add(new Text(tmp.toString().substring(1)));
                    }else if(tmp.charAt(0) == 'B'){
                        listB.add(new Text(tmp.toString().substring(1)));
                    }
                }
                try {
                    executeJoinLogic(context);
```

valeurs de « right ascension » et de « declinaison » pour les valeurs courantes de chaque liste puis grâce à une autre fonction fournit par l'Observatoire, je calcul la distance entre les 2 sources, que je compare avec une distance de référence. Si la distance calculée est inférieur à la distance de référence, j'écris le résultat de ma jointure, sinon je ne l'écris pas

CrossMatchDriver (Cf. Annexe 3.4) :

Mon driver va donc configurer mon job MapReduce, en lui donnant notamment son nom mais en configurant les types des clés et des valeurs en sortie. Je lui spécifie le « Reducer » que je vais utiliser, ici j'utilise donc le CrossMatchReducer, dans ce programme la particularité revient à utiliser 2 mapper. En effet, nos catalogues n'ayant pas la même structure des données, il est difficile de réaliser un mapper qui pourra calculer l'id sans savoir dans quelle colonne se trouve les données dont il a besoin. C'est pourquoi j'utilise 2 « mapper », le HipMainJoinMapper qui réalisera la tâche « map » sur toutes les lignes du catalogues HipMain, et le TychoJoinMapper qui fera la tâche « map » sur toutes les valeurs du catalogue Tycho. Finalement je fournis à mon job les chemins vers mes catalogues et vers le dossier de sortie.

```
public class CrossMatchDriver {
    public static void main(String[] args) throws Exception {
        //lancement de l'execution du job
        int res = ToolRunner.run(new Configuration(),(Tool) new CrossMatchDriver(),args);
        System.exit(res);
    }

    public int run(String[] args) throws Exception{
        //configuration du job
        Configuration config = new Configuration();

        JobClient client = new JobClient();
        JobConf conf = new JobConf(config);

        conf.setJobName("CrossMatch");
        //configuration du type de Sortie pour la cle et la valeur
        conf.setOutputKeyClass(Text.class);
        conf.setOutputValueClass(Text.class);
        //choix du reducer pour ce job
        conf.setReducerClass(CrossMatchReducer.class);
        //choix des mapper pour ce job
        MultipleInputs.addInputPath(conf, new Path(args[0]),HipMainJoinMapper.class);
        MultipleInputs.addInputPath(conf , new Path(args[1]),TychoJoinMapper.class);
        //configuration de la sortie
        FileInputFormat.setInputPaths(conf, new Path(args[2]));
        FileInputFormat.setInputPaths(conf,new Path(args[3]));
        FileOutputFormat.setOutputPath(conf,new Path(args[4]));

        //lancement de l'execution
        client.runJob(conf);
        return 0;

    }
```

Réalisation de l'application avec Spark

Il m'a semblé intéressant de réaliser l'application de CrossMatch également avec le framework Spark car j'ai réalisé depuis le début de mon stage des essais avec ces 2 technologies. De plus, comme nous l'avons vu, Spark offre une approche différente du problème et de la façon d'écrire le code.

Le traitement réalisé avec le code Spark aura le même objectif. Tout d'abord généré des couples clé/valeur dont la clé sera l'identifiant calculé et la valeur la ligne courante, l'exception sera qu'il n'est pas nécessaire de mettre une lettre pour identifier le catalogue. Ensuite il suffira d'appliquer une fonction de jointure sur les 2 catalogues ainsi préparé puis d'enregistrer le résultat dans le dossier de sortie. (Cf Annexe 4)

Configuration et chargement des fichiers

Notre application de CrossMatch va donc devoir charger mes deux catalogues Tycho2 et HipMain. J'utilise donc la méthode *textFile()* de la classe *JavaSparkContext*. J'obtiens donc de variables de type *JavaRDD<String>* il s'agit donc d'une collection de String stocké sur le système de fichier de Spark. Maintenant que j'ai ces catalogues à ma disposition je peux réaliser les traitements que je souhaite.

Création des couples clé/valeur

Pour réaliser mon job de CrossMatch je dois mettre en place des Tuples <clé/valeur> pour pouvoir réaliser ma jointure. Pour cela ma fonction va s'écrire de la façon suivante :

```
//transformation de chaque valeur du fichier tycho en couple clé/valeur
//ou la clé est l'id calculer et la valeur à conserver
JavaPairRDD<String,String> tychoPairs = tychoInputFile.mapToPair(new Pa
    public Tuple2<String,String> call(String s){
        //je découpe la ligne pour récupérer chacune des valeurs
        String[] t = s.split(",");
        //je stocke la valeur ra et de et je les converti en double
        Double ra = Double.parseDouble(t[0]);
        Double de = Double.parseDouble(t[1]);
        //je calcule la valeur de l'id
        Long hmid = nestIdx(ra,de);
        return new Tuple2<String,String>(hmid.toString(),s);

    }
});
```

Je vais stocker le résultat de ma fonction dans un JavaPairRDD<String,String> ce qui veut dire que chaque élément de cette collection sera un Tuple<String,String>. Pour obtenir ce résultat j'applique sur ma variable contenant mon catalogue la fonction *mapToPair()* dont le rôle est de transformer 1 élément en 1 Tuple. J'utilise la classe PairFunction en paramètre qui prend en entrée 1 String et en sortie un couple de String.

La fonction *call(String s)* renvoie un *Tuple2<String,String>*. Elle réalise les actions suivantes :

- Tout d'abord elle découpe le String par la fonction *split()* pour récupérer chacune des valeurs contenues dans la ligne courante.

- Ensuite elle récupère les valeurs de « right ascension » et de « declinaison » contenu dans mon tableau de String obtenu précédemment et je les convertis de String en Double pour pouvoir les utiliser dans la fonction.
- Tout comme pour le programme Hadoop, je vais utiliser la fonction *nestIdx()* fournit par l'Observatoire pour calculer l'identifiant qui me servira à réaliser ma jointure.
- Finalement la fonction renvoie un Tuple contenant l'identifiant et la ligne courante.

A la fin de ce traitement j'obtiens donc une collection composée de *Tuple<String,String>* avec l'identifiant et la valeur de la ligne. Ce traitement est réalisé pour les catalogues de façon séparée.

Réalisation de la jointure

Maintenant que je dispose de 2 collections de couple clé/valeur je vais pouvoir réaliser ma jointure. Cette action va s'avérer assez simple puisque l'API Spark propose une fonction *join()* de la classe *JavaPairRDD* prenant en paramètre un *JavaPairRDD* est qui donc réalise une jointure de ces 2 collections puis stocke le résultat dans un autre *JavaPairRDD*. Il ne me reste alors plus qu'à écrire le résultat en sortie grâce à la fonction *saveAsTextFile()*.

Résultat du job

En entrée du job j'ai donc ces 2 catalogues

Tycho : 2.5 millions valeurs

```
45.086141,+0.248856,10.722,-1
45.066560,+0.248301,10.977,-1
45.136055,+0.335060,10.558,-1
45.141726,+0.360084,10.354,-1
45.152796,+0.386348,10.103,-1
45.112791,+0.380918,11.214,-1
45.010102,+0.351034,10.186,-1
45.094862,+0.476808,9.970,-1
44.974594,+0.473637,10.302,-1
```

HipMain : 120 000 valeurs

```
2,9.27,0.00379737,-19.49883745,21.9
3,6.61,0.00500795,38.85928608,2.81
4,8.06,0.0083817,-51.89354612,7.75
5,8.55,0.00996534,-40.5912244,2.87
6,12.31,0.01814144,3.94648893,18.8
7,9.64,0.02254891,20.03660216,17.74
8,9.05,0.0272916,25.88647445,5.17
9,8.59,0.03534189,36.58593777,4.81
10,8.59,0.03625309,-50.8670736,10.76
```

En sortie du job j'obtiens ce résultat en une dizaine de seconde : 119 167 valeurs

```
235.424659,+35.998467,9.259,76859   76859,9.16,235.42465724,35.99846313,5.36
234.053259,+35.705569,7.570,76398   76398,7.49,234.05328206,35.70557943,19.0
233.965276,+35.923588,8.296,76369   76369,8.27,233.9652786,35.92358797,6.45
234.559321,+36.247464,9.019,-1  76563,7.07,234.55398295,36.24674611,11.42
234.559321,+36.247464,9.019,-1  76566,7.4,234.55914245,36.24704829,11.42
234.559128,+36.247052,8.065,76566   76563,7.07,234.55398295,36.24674611,11.42
234.559128,+36.247052,8.065,76566   76566,7.4,234.55914245,36.24704829,11.42
234.553974,+36.246754,7.851,76563   76563,7.07,234.55398295,36.24674611,11.42
234.553974,+36.246754,7.851,76563   76566,7.4,234.55914245,36.24704829,11.42
234.844510,+36.635828,4.962,76669   76669,4.64,234.84451583,36.63582763,6.89
234.766794,+36.754309,9.267,76643   76643,9.3,234.76679796,36.75431232,3.87
232.462412,+34.736357,9.182,75872   75872,9.16,232.46241277,34.73635742,2.65
232.435800,+35.175317,8.015,75862   75862,7.96,232.43579942,35.17531938,16.09
232.675639,+35.168828,9.018,75947   75947,9.01,232.67563992,35.16882799,3.9
232.663515,+35.784574,11.142,75942  75942,11.1,232.66350834,35.78455846,0.47
```

Il est possible alors de remarquer que le code écrit avec Spark semble bien plus simple que celui écrit avec Hadoop. N’ayant pas tout à fait terminé l’application de CrossMatch avec Spark je n’ai pas pu faire une comparaison de vitesse entre les deux framework.

Conclusion

Le stage que je viens de réaliser a été pour moi une expérience enrichissante car j'ai pu découvrir le monde du travail au sein d'une structure à la pointe de la technologie comme l'Observatoire Astronomique de Strasbourg. J'ai pu travailler dans le domaine de la recherche, sur un sujet informatique des plus concrets. J'ai pu explorer une nouvelle part de l'informatique qui m'était encore inconnu avant ce stage, le Big Data et une partie des technologies gravitant autour. J'ai pu utiliser les connaissances acquises durant mes 2 années d'études à l'IUT en les approfondissant. J'ai appris à réaliser des applications visant à traiter d'importants volumes de données. J'ai découvert de nouvelles technologies comme Hadoop ou Spark et une partie des technologies qui y sont liées : Hive, Pig, HDFS,…

Cette expérience m'a offert un bel aperçu de ce qu'est la vie en entreprise. Cela me conforte dans mon choix de carrière professionnel et me pousse à poursuivre plus loin dans mes études pour acquérir de plus amples connaissances dans le domaine de l'informatique.

Pour conclure également sur mon sujet, je pense qu'Hadoop, ou même Spark, est une technologie offrant la possibilité de réaliser des applications de manière plus simple car le programmeur n'a pas à gérer toute la partie parallélisassions et gestion des données. Cependant, il est plutôt compliqué de réaliser des jobs complexes si l'on n'a pas eu de formation sur ce framework. Enfin je pense que Spark pourrait offrir de bien meilleurs résultats d'un point de vue performance, bien que je ne l'ai pas testé, mais également pour l'écriture du code qui reste bien plus simple à réaliser qu'avec Hadoop.

Remerciement

Je tiens à remercier mon maître de stage, Monsieur André Schaaff, pour m’avoir apporté son assistance tout au long de mon stage, pour m’avoir conseillé et aidé. Il m’a consacré du temps tout au long de cette période, a su répondre à mes différentes questions et s’est assuré que je ne reste pas bloqué sur un problème. Nos discussions m’ont permis de mettre au clair certaines notions encore obscur et m’ont permis de rester sur la bonne voie tout au long de mon travail.

Je remercie également Monsieur Denis Roegel, mon parrain de stage qui m’a encadré, qui a su répondre à mes interrogations et qui a pris de son temps pour m’aider.

Je souhaite également remercier l’IUT Nancy Charlemagne, tous ses enseignants et intervenant professionnels du département informatique. Ils m’ont apporté un bagage informatique important qui m’a beaucoup servi durant ce stage et qui va sûrement m’être très utile pour ma poursuite d’étude.

Je suis satisfait du travail que j’ai réalisé au sein de l’Observatoire Astronomique de Strasbourg. Les conditions de travails furent idéales aussi bien pour le matériel mis à notre disposition que pour le cadre et l’ambiance de travail. J’ai senti une bonne cohésion entre les différents professionnels qui n’hésitaient pas à prendre du temps pour apporter leur aide.

Cette expérience de travail s’avère donc pour moi positive d’autant plus que j’ai pu découvrir, pour mon plus grand plaisir, l’astronomie par le biais des différentes conférences auxquelles j’ai pu assister.

Index

Map : Il s’agit de parcourir une collection dans le but de modifier les données qui y sont stockés.

Reduce : Il s’agit de réaliser une opération sur les éléments contenus dans une collection

Jointure : Il s’agit d’une opération visant à associer plusieurs tables ou vues de la base par le biais d’un lien logique de données. Le résultat est une nouvelle table.

Cluster : Il s’agit, en calcul distribué, d’un système informatique composé d’unités de calcul (microprocesseur, cœurs, unités centrale) autonomes qui sont reliés entre elles à l’aide d’un réseau de communication.

Bibliographie

Developpez.com : http://www.developpez.com

StackOverFlow: www.stackoverflow.com

Wikipédia: https://fr.wikipédia.org

HortonWorks: https://fr.hortonworks.com

Apache: http://apache.org

Spark: http://spark.apache.org

MapReduce Design Patterns, O'Reilly

MapReduce: Simplified Data Processing on Large Clusters, *Jeffrey Dean and Sanjay Ghemawat*-Google,Inc.

An Efficient Cross-Match Implementation based on Directed Join Algorithm in MapReduce, *Cuncang Mi,Qian Chen and Taoying Liu*, Institute of Computing Technology.

Annexes

Annexes 1: Wordcount avec Hadoop

1.1 Le WordCountMapper

```
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.io.Writable;
import org.apache.hadoop.io.WritableComparable;
import org.apache.hadoop.mapred.MapReduceBase;
import org.apache.hadoop.mapred.Mapper;
import org.apache.hadoop.mapred.OutputCollector;
import org.apache.hadoop.mapred.Reporter;

public abstract class WordCountMapper extends MapReduceBase implements Mapper<LongWritable, Text, Text, IntWritable> {

    private final IntWritable one = new IntWritable(1);
    private Text word = new Text();

    //La classe de Map
    @Override
    public void map(LongWritable key, Text value,OutputCollector<Text, IntWritable> output, Reporter reporter)throws IOException {

        String line = value.toString();
        StringTokenizer itr = new StringTokenizer(line.toLowerCase());
        while (itr.hasMoreTokens()) {
            word.set(itr.nextToken());
            output.collect(word, one);
        }
    }

}
```

1.2 Le WordCountReducer

```
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.io.WritableComparable;
import org.apache.hadoop.mapred.MapReduceBase;
import org.apache.hadoop.mapred.OutputCollector;
import org.apache.hadoop.mapred.Reducer;
import org.apache.hadoop.mapred.Reporter;


public abstract class WordCountReducer extends MapReduceBase implements Reducer<Text, IntWritable, Text, IntWritable> {

    //La methode de Reduce
    @Override
    public void reduce(Text key, Iterator<IntWritable> values, OutputCollector<Text, IntWritable> output,Reporter reporter) throws IOException {

        int sum = 0;
        while (values.hasNext()){
            sum += values.next().get();
        }
        output.collect(key, new IntWritable(sum));
    }
}
```

1.3 Le WordCountDriver

```java
public class WordCountDriver {

    public static void main(String[] args) {

        Configuration config = new Configuration();
        // config.set("fs.default.name", "hdfs://latdevweb02:9000/");
        // config.set("mapred.job.tracker", "latdevweb02:9001");

        JobClient client = new JobClient();
        JobConf conf = new JobConf(config);

        // TODO: specify output types and set jar
        conf.setOutputKeyClass(Text.class);
        conf.setOutputValueClass(IntWritable.class);
        conf.setJar("WordCount.jar");

        // TODO: specify a mapper and a reducer and the combiner
        conf.setMapperClass(WordCountMapper.class);
        conf.setCombinerClass(WordCountReducer.class);
        conf.setReducerClass(WordCountReducer.class);

        // TODO: specify input and output DIRECTORIES (not files)
        conf.setInputFormat(TextInputFormat.class);
        conf.setOutputFormat(TextOutputFormat.class);

        // TODO: specify input and output DIRECTORIES (not files)
        FileInputFormat.setInputPaths(conf, new Path(
                "/home/hadoop/hadoop/input"));
        FileOutputFormat.setOutputPath(conf, new Path(
                "/home/hadoop/hadoop/output"));

        System.out.println("Entrée dans le programme MAIN !!!");

        client.setConf(conf);
        try {
            JobClient.runJob(conf);
            System.out.println("Sortie du programme MAIN!!!");
        } catch (Exception e) {
            e.printStackTrace();
        }

    }
```

Annexes 2: Essais avec Spark

2.1 Comparaison

```
package org.sparkexample;

import org.apache.spark.SparkConf;
import org.apache.spark.api.java.JavaPairRDD;
import org.apache.spark.api.java.JavaRDD;
import org.apache.spark.api.java.JavaSparkContext;
import org.apache.spark.api.java.function.FlatMapFunction;
import org.apache.spark.api.java.function.Function2;
import org.apache.spark.api.java.function.PairFunction;
import scala.Tuple2;
import java.util.Arrays;

public class Compare {

  public static void main(String[] args) {
    if(args.length < 2) {
         System.err.println("Ajouter le chemin vers fichier d'entrée dans l'argument");
         System.exit(0);
    }
    SparkConf conf = new SparkConf().setAppName("org.sparkexample.Compare");
    JavaSparkContext context = new JavaSparkContext(conf);

    JavaRDD<String> file = context.textFile(args[0]);
    JavaRDD<String> words = file.flatMap(new FlatMapFunction<String,String>(){
        public Iterable<String> call(String s){
                return Arrays.asList(s.split("\n"));
        }
        }
    );
    JavaRDD<String> file2 = context.textFile(args[1]);
    JavaRDD<String> words2 = file.flatMap(new FlatMapFunction<String,String>(){
    public Iterable<String> call(String s){
        return Arrays.asList(s.split("\n"));
    }
    }
    );

    JavaRDD<String> result = words.intersection(words2);
    result.saveAsTextFile(args[2]);

  }
```

2.2 CrossSearch

```
public static void main(String[] args){
        if(args.length< 1){
          System.err.println("Ajouter les arguments manquant");
          System.exit(0);
        }

        SparkConf conf = new SparkConf().setAppName("org.sparkexample.CrossSearch");
        JavaSparkContext context = new JavaSparkContext(conf);

        final Integer centre = Integer.parseInt(args[2]);
        final Integer rayon = Integer.parseInt(args[3]);

        JavaRDD<String> tychofile = context.textFile(args[0]);
        JavaRDD<String> tycho = tychofile.flatMap(new FlatMapFunction<String,String>(){
            public Iterable<String> call(String s){
                  return Arrays.asList(s.split("\n"));
            }
            }
        );
        JavaRDD<String> tychofiltre = tycho.filter(new Function<String,Boolean>(){
           public Boolean call(String s){
              String[] t = s.split(",");
              Double ra = Double.parseDouble(t[1]);
              return (ra < centre+rayon) && (ra >centre-rayon);
           }
        } );

        JavaRDD<String> hipMainfile = context.textFile(args[1]);
        JavaRDD<String> hipMain = hipMainfile.flatMap(new FlatMapFunction<String,String>(){
           public Iterable<String> call(String s){
                 return Arrays.asList(s.split("\n"));
            }
            }
        );

        JavaRDD<String> hipMainfiltre = hipMain.filter(new Function<String,Boolean>(){
            public Boolean call(String s){
                  String[] t = s.split("\t");
                  Double ra = Double.parseDouble(t[2]);
                  return (ra< centre + rayon) && (ra > centre - rayon);
            }
            }
       );
       JavaRDD<String> result = tychofiltre.union(hipMainfiltre);
        result.saveAsTextFile(args[4]);
```

Annexes 3: CrossMatch avec Hadoop

3.1 TychoJoinMapper

```java
import healpix.core.HealpixBase;
import healpix.core.Pointing;
import healpix.core.Scheme;
import java.io.IOException;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.io.Writable;
import org.apache.hadoop.io.WritableComparable;
import org.apache.hadoop.mapred.MapReduceBase;
import org.apache.hadoop.mapred.Mapper;
import org.apache.hadoop.mapred.OutputCollector;
import org.apache.hadoop.mapred.Reporter;
import org.apache.hadoop.mapreduce.Mapper.Context;
public class TychoJoinMapper extends MapReduceBase implements Mapper {
    private Text outkey = new Text();
    private Text outvalue = new Text();
    private final int nside = 2048;
    private  HealpixBase hb;
    public long nestIdx(final double raDeg, final double decDeg) {
        try {
            this.hb = new HealpixBase(nside, Scheme.NESTED);
            return this.hb.ang2pix(new Pointing(
                Math.toRadians(90 - decDeg),
                Math.toRadians(raDeg)));
        } catch (Exception e) {
            throw new Error(e);
        }
    }
    @Override
    public void map(Object key, Object value, OutputCollector context,
            Reporter arg3) throws IOException {
        String[] parsed = value.toString().split(",");
        String ra = parsed[0];
        String dec = parsed[1];
        double raDeg = Double.parseDouble(ra);
        double decDeg = Double.parseDouble(dec);
        long id = nestIdx(raDeg,decDeg);
        outkey.set(id+"");
        outvalue.set("A" + value.toString());
        context.collect(outkey,outvalue);
    }
}
```

3.2 HipMainJoinMapper

```
public class HipMainJoinMapper extends MapReduceBase implements Mapper {
    private Text outkey = new Text();
    private Text outvalue = new Text();
    private static final int nside = 2048;
    private static HealpixBase hb;
    static{
        try {
            hb = new HealpixBase(nside, Scheme.NESTED);
        } catch (Exception e) {
            throw new Error(e);
        }
    }
    public static long nestIdx(final double raDeg, final double decDeg) {
        try {
            return hb.ang2pix(new Pointing(
                Math.toRadians(90 - decDeg),
                Math.toRadians(raDeg)));
        } catch (Exception e) {
            throw new Error(e);
        }
    }
  @Override
    public void map(Object key, Object value, OutputCollector context,
            Reporter arg3) throws IOException {
        String[] parsed = value.toString().split(",");
        if(parsed.length>4){
        String ra = parsed[2];
        String dec = parsed[3];
            double raDeg = Double.parseDouble(ra);
            double decDeg = Double.parseDouble(dec);
            //calcul de l'id
            long id = nestIdx(raDeg,decDeg);
            //on passe l'id comme clé du couple
            outkey.set(id+"");
            //on passe la valeur de la ligne + A pour désigner la src d'origine
            outvalue.set("B" + value.toString());
            //on passe le couple valeur en sortie
             context.collect(outkey,outvalue);
            try {
                for (long nei : hb.neighbours(id)) {
                    outkey.set(nei+"");
                    context.collect(outkey,outvalue);
                }
            } catch (Exception e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
```

3.3 CrossMatchReducer

```
public class CrossMatchReducer extends MapReduceBase implements Reducer {
    private ArrayList<Text> listA = new ArrayList<Text>();
    private ArrayList<Text> listB = new ArrayList<Text>();
    private Text tmp = new Text();
    private String joinType = null;
    public static double havDist(final double ra1Deg, final double dec1Deg,final double ra2Deg, final double dec2Deg) {
        final double d = Math.sin(Math.toRadians(0.5 * (dec2Deg - dec1Deg)));
        final double a = Math.sin(Math.toRadians(0.5 * (ra2Deg - ra1Deg)));
        final double h3 = d * d + a * a * Math.cos(Math.toRadians(dec1Deg)) * Math.cos(Math.toRadians(dec2Deg));
        return Math.toDegrees(2.0 * Math.asin(Math.sqrt(h3)));
    }
    public static final double xmatchDistance = 5.0/3600.0  ;
    public void setup(Context context){
        joinType = context.getConfiguration().get("join.type");
    }
    private void executeJoinLogic(OutputCollector context) throws IOException, InterruptedException {
        if(!listA.isEmpty() && !listB.isEmpty()){
            for(Text A : listA){
                String[] a = A.toString().split(",");
                double ra1 = Double.parseDouble(a[0]) ;
                double dec1 = Double.parseDouble(a[1]) ;

                for(Text B : listB) {
                    String[] b = B.toString().split(",");
                    double ra2 = Double.parseDouble(b[2]);
                    double dec2 = Double.parseDouble(b[3]);
                    double distance = havDist(ra1,dec1,ra2,dec2);
                    if (distance < xmatchDistance) {
                        context.collect(A,B);
                    }
                }}
        }}
    @Override
    public void reduce(Object key, Iterator values, OutputCollector context,
            Reporter arg3) throws IOException {
                listA.clear();
                listB.clear();
                while(values.hasNext()){
                    tmp = (Text) values.next();
                    if(tmp.charAt(0) == 'A'){
                        listA.add(new Text(tmp.toString().substring(1)));
                    }else if(tmp.charAt(0) == 'B'){
                        listB.add(new Text(tmp.toString().substring(1)));
                    }
                }
                try {
                    executeJoinLogic(context);
```

3.4 CrossMatchDriver

```
import org.apache.hadoop.conf.Configured;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapred.FileInputFormat;
import org.apache.hadoop.mapred.FileOutputFormat;
import org.apache.hadoop.mapred.JobClient;
import org.apache.hadoop.mapred.JobConf;
import org.apache.hadoop.mapred.Mapper;
import org.apache.hadoop.mapred.Reducer;
import org.apache.hadoop.mapred.TextInputFormat;
import org.apache.hadoop.mapred.lib.MultipleInputs;
import org.apache.hadoop.mapreduce.InputFormat;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.util.Tool;
import org.apache.hadoop.util.ToolRunner;
public class CrossMatchDriver extends Configured implements Tool {
    public static void main(String[] args) throws Exception {
        int res = ToolRunner.run(new Configuration(), new CrossMatchDriver(),args);
        System.out.println("fin du job");
        System.exit(res);
    }
    public int run(String[] args) throws Exception{
        JobConf conf = new JobConf();
        JobClient client = new JobClient();

        conf.setJobName("CrossMatch");

        MultipleInputs.addInputPath(conf, new Path(args[0]),TextInputFormat.class,TychoJoinMapper.class);
        MultipleInputs.addInputPath(conf,new Path(args[1]),TextInputFormat.class,HipMainJoinMapper.class);

        FileOutputFormat.setOutputPath(conf, new Path(args[2]));

        conf.setReducerClass(CrossMatchReducer.class);

        conf.setOutputKeyClass(Text.class);
        conf.setOutputValueClass(Text.class);

        System.out.println("debut du job");
        client.init(conf);
        client.runJob(conf);

        return 0;
    }
```

Annexes 4: CrossMatch avec Spark

```java
import java.util.List;
import healpix.core.HealpixBase;
import healpix.core.Pointing;
import healpix.core.Scheme;

public class CrossMatch {
  private final static int nside = 2048;
  private static HealpixBase hb ;
  public static long nestIdx(final double raDeg, final double decDeg) {
    try {hb = new HealpixBase(nside,Scheme.NESTED);
        return  hb.ang2pix(new Pointing(
                Math.toRadians(90 - decDeg),
                Math.toRadians(raDeg)));
    } catch (Exception e) {
        throw new Error(e);
    } }

  public static void main(String[] args){
    SparkConf conf = new SparkConf().setAppName("org.spark.CrossMatch");
    JavaSparkContext context = new JavaSparkContext(conf);
    JavaRDD<String> tychoInputFile = context.textFile("/data/spark-1.3.0/travaux/Test3/src/test/resources/tycho2.csv");
    JavaRDD<String> hipMainInputFile = context.textFile("/data/spark-1.3.0/travaux/Test3/src/test/resources/I_239_hip_main.csv");

    JavaPairRDD<String,String> tychoPairs = tychoInputFile.mapToPair(new PairFunction<String, String, String>(){
        public Tuple2<String,String> call(String s){
            String[] t = s.split(",");
            Double ra = Double.parseDouble(t[0]);
            Double de = Double.parseDouble(t[1]);
            Long hmid = nestIdx(ra,de);
            return new Tuple2<String,String>(hmid.toString(),s);
        }
    });
    JavaPairRDD<String,String> hipMainPairs = hipMainInputFile.mapToPair(new PairFunction<String, String, String>(){
        public Tuple2<String,String> call(String s){
            String[] t = s.split(",");
            Double ra = Double.parseDouble(t[2]);
            Double de = Double.parseDouble(t[3]);
            Long hmid = nestIdx(ra,de);
            return new Tuple2<String,String>(hmid.toString(),s);
        }
    });
    JavaPairRDD<String, Tuple2<String, String>> joinsOutput = tychoPairs.join(hipMainPairs);
    joinsOutput.saveAsTextFile("/data/spark-1.3.0/travaux/Test3/result");
  }
}
```