Projet systèmes concurrents/intergiciels
Implantation et utilisation du modèle Map-Reduce

2AI

14 novembre 2017

Le projet a pour but d'expérimenter le développement d'applications concurrentes pour le calcul intensif et le traitement de masses de données. Pour cela, il s'agira

- de réaliser une plateforme comportant un système de fichiers adapté au traitement concurrent de masses de données, ainsi qu'un noyau d'exécution axé sur l'ordonnancement et la gestion de tâches selon le schéma « diviser pour régner » (map-reduce). L'architecture et les fonctionnalités de cette plateforme reprendront (de manière simplifiée) celles de la plateforme Hadoop.
- de développer des applications concurrentes basées sur l'utilisation de cette plateforme, afin
  - d'une part d'évaluer les choix de conception possibles pour ces applications,
  - et d'autre part de tester et évaluer la plateforme réalisée.
- de développer des outils d'instrumentation, facilitant la supervision de l'exécution des applications.

Le schéma MapReduce permet d'effectuer en parallèle (sur une grappe (*cluster*) de machines) des traitements sur un grand volume de données. Les données sont découpées en fragments qui sont répartis (stockés) sur les machines de traitement. Les fragments sont alors traités en parallèle sur les différentes machines, et les résultats partiels issus du traitement des fragments sont alors agrégés pour donner le résultat final. Ainsi, une application Map-Reduce spécifie essentiellement

- une procédure map(), qui est exécutée en parallèle sur les machines de traitement sur tous les fragments (auxquels on accède donc localement),
- et une procédure reduce() qui exploite le résultat des map pour obtenir le résultat final.

Plan

- La section suivante présente l'architecture de principe de la plateforme Hadoop, ainsi que ses fonctionnalités essentielles.
- **Les sections 2 et 3 détaillent ensuite le contenu attendu du projet : plateforme, applications, évaluation.**
- Enfin, les modalités pratiques (organisation, échéances, livrables...) sont précisées.

1 La plateforme Hadoop : architecture et principe de fonctionnement

Note : *La présentation qui suit se limite aux points essentiels de l'architecture et du fonctionnement de Hadoop, en mettant l'accent sur les éléments directement utiles à la réalisation de votre projet. De nombreux aspects relatifs à l'optimisation, à l'initialisation, au placement, à l'administration, ou à l'interface programmatique ou utilisateur sont omis. Si vous êtes curieux, ou désireux d'approfondir certains aspects (par exemple en vue de réaliser la seconde étape du projet), les références données sur la page Moodle de l'UE (rubrique « Projet ») constituent un bon point de départ.*

La philosophie générale de la plateforme Hadoop est de permettre de résoudre des problèmes par la force brute : on se propose de permettre au programmeur de considérer qu'il dispose de ressources (stockage, calcul) en quantité illimitée, et d'un coût négligeable. Sous ces hypothèses, on peut chercher à minimiser les temps de traitement, en négligeant l'efficacité des traitements.

La mise en œuvre de ce paradigme de programmation implique de résoudre les difficultés résultant du recours à une grande quantité de ressources (processeur, stockage, réseau...) :

- dès lors que des milliers de machines et périphériques concourent à un traitement, la défaillance d'un ou plusieurs composants en cours d'exécution devient la règle. Il est donc indispensable de mettre en place des mécanismes de *tolérance aux pannes*, permettant de masquer (gérer automatiquement) les défaillances. Ces mécanismes s'appuient notamment sur la redondance et sur la capacité à ajuster dynamiquement les ressources allouées à un traitement.
- Une gestion efficace des ressources physiques est cruciale au regard des volumes de calcul et de données en jeu. Les besoins d'un traitement excèdent en effet de plusieurs ordres de grandeur les capacités d'un composant pris isolément. Pour tirer profit du grand nombre de ressources, il faut donc éviter que certains composants se trouvent surchargés alors que d'autres restent inemployés. C'est l'objet des stratégies d'*équilibrage de charge*. Par ailleurs, les calculs intermédiaires peuvent s'avérer particulièrement gourmands en termes d'espace mémoire/de stockage ; il faut donc veiller à ce que chaque machine dispose de suffisamment de ressources libres pour mener à bien les tâches qui lui sont échues.
- Enfin, la gestion de la synchronisation entre tâches d'un même traitement impacte fortement les temps d'exécution : il s'agit de limiter le plus possible les points de synchronisation globaux.

1.1 Système de fichiers

Le système de fichiers HDFS (Hadoop Distributed File System) est conçu pour faciliter un accès massivement concurrent à de larges volumes de données [1]. Pour cela, les fichiers utilisateur sont fragmentés en blocs (*chunks*) de taille identique (typiquement 64 MB), qui sont dupliqués à des fins de tolérance aux pannes et d'efficacité. Un pool (*cluster*) de serveurs HDFS est coordonné par un serveur maître, le *NameNode*, qui gère le catalogue du système de fichiers (métadonnées et localisation des différentes copies des chunks de chaque fichier). Les serveurs du pool (*DataNodes*) gèrent les accès aux chunks conservés sur la machine à laquelle ils sont affectés. Habituellement chaque machine de la grappe HDFS accueille un *DataNode*.

1. Un traitement Hadoop typique met en jeu des milliers de processeurs travaillant sur des dizaines de téraoctets de données.

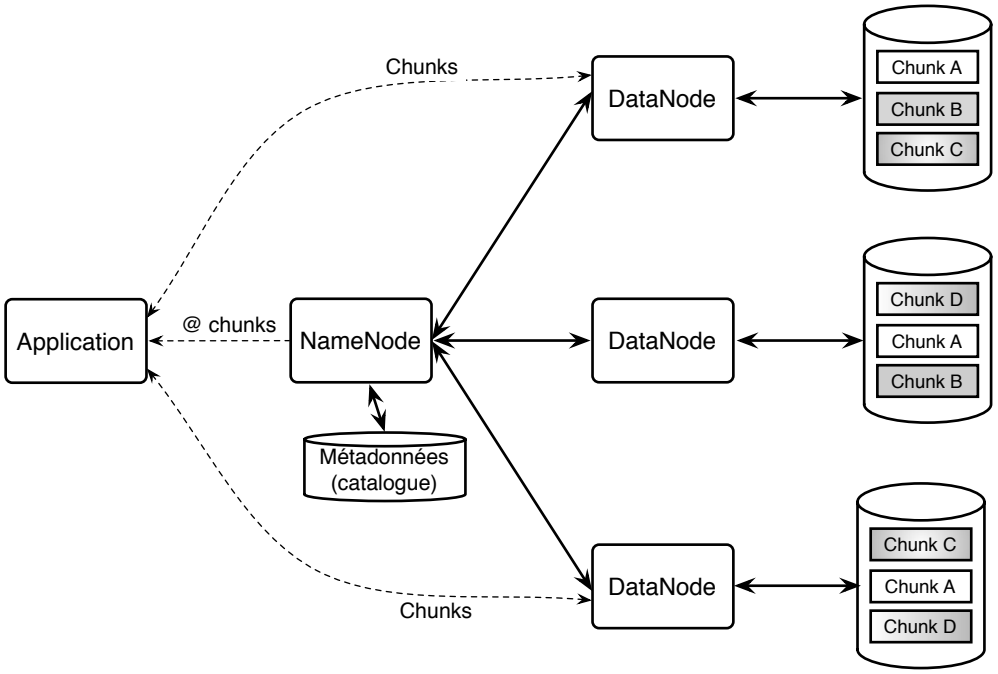

FIGURE 1 – architecture de HDFS

1.1.1 Applications cibles

HDFS est conçu pour faciliter l'accès aux données d'applications suivant le modèle MapReduce. Ces applications présentent des caractéristiques très particulières :

- le mode d'accès principal est le parcours séquentiel en lecture de fichiers de données. Ce type d'accès est propice à la parallélisation, puisqu'il permet de lire le fichier en parallèle, par fragments.
- les fichiers de données sont écrits une seule fois, puis lus (utilisés) plusieurs fois. Dans ces conditions, les difficultés liées aux conflits entre écritures ou entre lectures et écritures sont réduits au minimum.

Sur cette base, HDFS est nettement moins efficace en ce qui concerne

- les accès aléatoires à un fichier
- les écritures sur un fichier existant ; il existe cependant un mode d'écriture qui reste raisonnablement efficace compte tenu du profil des applications MapReduce, à savoir l'ajout de données en fin de fichier.

1.1.2 Architecture et structures de données

Le *NameNode* gère les métadonnées du système de fichiers : pour chaque fichier, une structure analogue à un i-nœud regroupe les informations utiles à la gestion du fichier. Cette structure comporte en particulier :

- la taille du fichier
- la taille des blocs (*chunks*)
- la liste des identifiants de blocs (*chunk handles*) constituant le fichier
- le facteur de duplication des blocs du fichier (habituellement 3) : chacun des blocs est en effet dupliqué et ses copies sont distribuées sur différents *DataNodes*.

La taille importante des chunks permet à cet ensemble de métadonnées (*image*) de rester suffisamment compact pour être conservé en mémoire vive, ce qui assure l'efficacité des accès aux données du *NameNode*.

Outre l'*image*, le *NameNode* gère l'arborescence des noms de fichiers, ainsi qu'une table d'implantation des copies de chaque *chunk* sur les différents *DataNodes*.

La fiabilité du *NameNode* et de ses métadonnées est essentielle. Aussi,

— les métadonnées sont elles régulièrement sauvegardées en mémoire stable ;

— un (ou des) serveurs de secours réalisent des copies de ces métadonnées (périodiquement ou en continu), afin de prendre le relai en cas de défaillance du *NameNode*.

Les *DataNodes* conservent les copies des blocs (*chunks*). Ces blocs sont généralement conservés dans un répertoire spécifique du nœud hôte, chaque bloc apparaissant comme un fichier dont le nom est l'identifiant de bloc (*chunk handle*). Afin de permettre un contrôle d'intégrité des données du bloc, un fichier est associé à chaque bloc, contenant un checksum du contenu du bloc.

Le *NameNode* supervise la disponibilité des *DataNodes* au moyen d'un protocole systolique : les *DataNodes* contactent le *NameNode* à intervalles réguliers. Un *DataNode* dont la requête ne parvient pas dans les délais est considéré comme défaillant, et les copies des blocs qu'il gère sont considérées comme indisponibles. Il faut noter que ce protocole permet un fonctionnement asynchrone, donc plus efficace du *NameNode*.

1.1.3 Mise en œuvre de la duplication : opérations sur les fichiers

Lecture d'un fichier

1. Contacter le *NameNode*.
2. Obtenir les métadonnées : identifiants de blocs (*chunk handles*) formant le fichier.
3. Pour chaque *chunk handle*, obtenir la liste des *DataNodes*.
4. Contacter l'un quelconque des *DataNodes* pour obtenir les données (sans passer par le *NameNode*).

Ecriture d'un fichier

1. Une requête d'écriture auprès du *NameNode* fournit un *DataNode* responsable (primaire) et un nouveau numéro de version.
2. Envoi des données
   — le client envoie ses données à écrire au *DataNode* primaire ;
   — le primaire propage à l'une des copies (secondaire), qui propage à la suivante, etc... Les données sont conservées en mémoire, non écrites.
3. Écriture des données
   — le client attend l'acquittement de toutes les copies (ou de la dernière) ;
   — il envoie un message de validation au primaire ;
   — le primaire ordonne les écritures (numéro de version), effectue son écriture et informe les copies secondaires ;
   — les secondaires effectuent l'écriture et l'acquittent au primaire ;
   — le numéro de version garantit que l'ordre des opérations est identique sur tous les nœuds.
4. Le primaire informe le client.

1.1.4 Interface utilisateur

Le système de fichiers HDFS est implanté au dessus du système de fichiers hôte. Il offre une interface programmatique, ainsi qu'un shell proposant des opérations basiques sur les répertoires (mkdir, rmdir, ls) et les fichiers (cp, mv, rm, cat...).

1.2 Plateforme d'exécution

La plateforme d'exécution ($Yarn$ [2]) est calquée sur l'architecture de HDFS. Elle est constituée
- d'un *RessourceManager*, qui arbitre et supervise globalement l'allocation de ressources aux différentes applications ; ce composant critique est exécuté par un nœud dédié.
- de *NodeManagers* exécutés sur les différentes machines de la grappe Hadoop. Chaque *NodeManagers* fournit l'ensemble de ressources locales [3] (mémoire, processeur...) nécessaires à l'exécution des différentes tâches qui lui sont soumises. Il assure la réservation et la supervision de ces *containers/slots*. De manière analogue à HDFS, les *NodeManagers* permettent au *RessourceManager* de suivre la disponibilité des ressources au moyen d'un protocole systolique.
- d'*ApplicationMasters* (un pour chaque application lancée), contrôlant l'ordonnancement et supervisant l'exécution des tâches de chacune des applications. L'*ApplicationMaster* obtient du *RessourceManager* la liste des *NodeManagers* qui accueilleront les tâches à exécuter, et leur soumet celles-ci. La progression des *ApplicationMasters* est suivie par le *RessourceManager* au moyen d'un protocole systolique.

Note : $Yarn$ est conçu comme une plateforme d'exécution générique, pouvant supporter différents modèles et moteurs d'exécution, mais pour la suite nous nous limiterons au modèle MapReduce, qui est le modèle initial et central de la plateforme $Yarn$.

1.2.1 Le schéma MapReduce

Conçu pour le traitement concurrent de masses de données, le schéma MapReduce est le modèle de programmation privilégié de la plateforme Hadoop. Un programme MapReduce reçoit un ensemble de données au format ⟨Clé,Valeur⟩ et produit un ensemble de résultats au format ⟨Clé,Valeur⟩. Le programmeur spécifie des fonctions *Map*, qui vont produire des données intermédiaires à partir des données d'entrée, et des fonctions *Reduce*, qui vont agréger les résultats des *Map* pour produire les résultats finaux.

1.2.2 Exécution d'un programme MapReduce

L'exécution d'un programme MapReduce comporte ainsi plusieurs phases :
- le *RessourceManager* crée un *ApplicationMaster*, chargé de contrôler et superviser l'exécution de l'application ;
- l'*ApplicationMaster* détermine les *chunks* utiles au calcul, les localise auprès du $NameNode$, définit un ensemble de tâches Map (une tâche par *chunk* de données d'entrée [4]) et un ensemble de tâches Reduce, et planifie chaque tâche pour un *slot* d'exécution auprès d'un *NodeManager* ;
- les tâches Map sont exécutées en parallèle. Chaque tâche Map produit (en mémoire vive) des paires ⟨Clé,Valeur⟩.
- Ces données intermédiaires peuvent (éventuellement) être prétraitées avant d'être transmises aux tâches Reduce. Ce prétraitrement ($Shuffle$) consiste à regrouper et trier par clés les paires ⟨Clé,Valeur⟩, ce qui aboutit à associer à chaque clé une liste de valeurs. Cet ensemble ⟨Clé, Liste de valeurs⟩ est alors partitionné par clés, selon les clés associées aux tâches Reduce.
- Le résultat du prétraitement (ou sinon l'ensemble des paires ⟨Clé,Valeur⟩) est alors enregistré sur disque (un fichier par partie), puis transmis aux nœuds correspondant aux

2. Yet Another Resource Negociator
3. appelé *container* (ou *slot* dans la première version d'Hadoop)
4. Avec un ajustement nécessaire, dans le cas où un enregistrement se trouve à cheval sur 2 blocs.

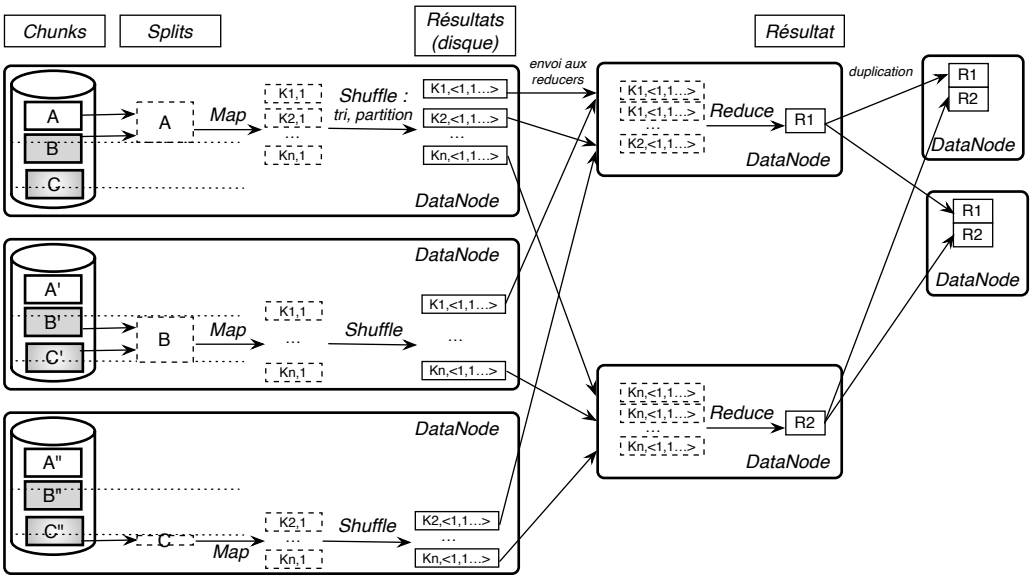

FIGURE 2 – Flot d'exécution MR

différentes tâches Reduce. [5]

— Une fois qu'elles disposent de l'ensemble des données des tâches Map, les tâches Reduce traitent les données intermédiaires et produisent leur résultat sur HDFS.

Exemple Fréquence d'occurrence des mots d'un texte Les tâches Map comptent le nombre d'occurrences des mots de leur fragment, et produisent des listes triées mot (clé),nb occurrence(valeur)), qui sont ensuite agrégées. L'annexe B est un exemple de mise en œuvre de cette application.

1.2.3 Ordonnancement des tâches

Le *ResourceManager* choisit la liste de machines (*NodeManager*) qui devront exécuter les Map et les Reduce. D'une manière générale, les stratégies mises en œuvre par les ordonnanceurs diffèrent selon les critères visés, mais reposent sur quelques principes récurrents :

— une partie est constituée d'heuristiques classiques en ordonnancement, mises en œuvre dans le cadre de la gestion des ressources : mixer les profils, afin d'obtenir un usage globalement équilibré des ressources, recourir à un ordonnancement hiérarchique/multiniveaux afin de composer des politiques...

— une partie est spécifique au schéma d'exécution MapReduce, structuré par le traitement d'une masse de données en phases successives :

  — la stratégie essentielle est de colocaliser autant que possible les tâches et les données qu'elles ont à traiter (chunks, données intermédiaires), ceci afin de réduire les temps d'accès aux données à traiter, ainsi que le trafic réseau ;

  — dans cette optique, certaines politiques reportent l'exécution de tâches en attendant que leurs données deviennent localement disponibles ;

  — d'autres politiques synchronisent l'exécution des tâches, à partir des dépendances (établies ou estimées) dans le flux de données traité ;

  — une stratégie adaptative originale consiste à superviser les tâches, et à lancer des tâches en renfort si une tâche est estimée en retard ou défaillante ;

5. Une étape (optionelle) de prétraitement supplémentaire (*Combiner*) assimilable à un Reduce peut être effectuée avant l'envoi aux nœuds des tâches Reduce, afin d'alléger le traitement des Reduce.

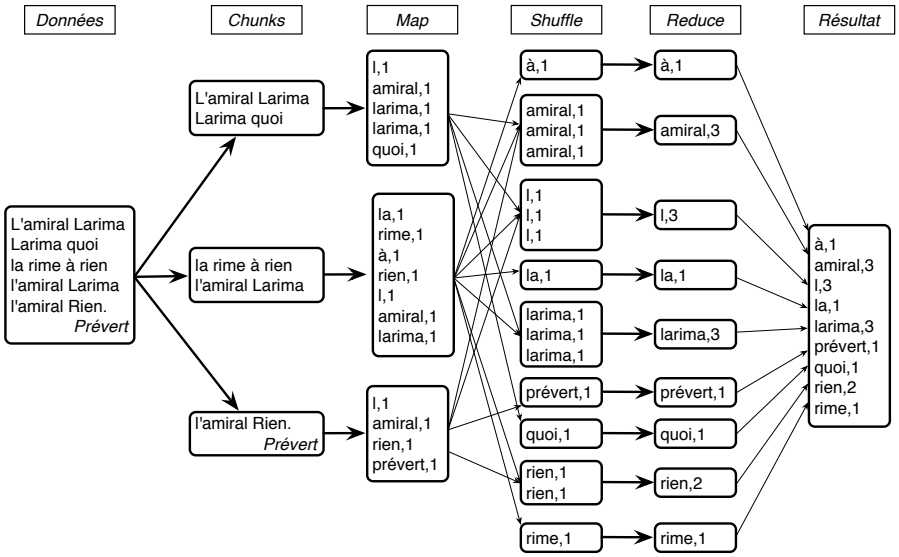

FIGURE 3 – WordCount

— enfin, certaines politiques allouent les nœuds les plus fiables aux tâches coûteuses, afin de réduire la déperdition de ressources induite par la nécessité de relancer une tâche en cas de panne.

Quelques politiques classiques

— La stratégie *Full-Map* consiste à lancer un processus Map sur chaque machine hébergeant un bloc de données à traiter, via le *NodeManager* correspondant. Les processus Reduce sont affectés en fonction du nombre de slots restant disponibles. L'avantage de cette stratégie est que chaque processus Map réalisera un traitement local. Son principal inconvénient est la contention possible au niveau des slots disponibles, lors de l'exécution d'un grand nombre d'applications MapReduce.

— Dans la *stratégie sans duplication*, le *ResourceManager* utilise uniquement la copie originale des blocs, et non leurs copies dupliquées. Dans cette stratégie, le nombre des Map et de Reduce est déterminé par l'utilisateur lors de la soumission de l'application. Le *ResourceManager* utilise la même politique que précédemment pour placer les tâches Reduce.

— Dans la *stratégie avec duplication*, le *ResourceManager* peut utiliser les copies dupliquées d'un bloc. L'avantage de cette politique est que le *ResourceManager* va éviter de chercher d'autres blocs sur d'autres machines, si cette machine en possède des copies dupliquées. Pour cette stratégie également, le nombre des Map et de Reduce est fixé par l'utilisateur lors de la soumission de l'application, et la même politique est suivie pour les tâches Reduce.

— L'objectif de *la stratégie spéculative* est de réduire le temps d'exécution en lançant des tâches de renfort en appui des tâches Map dont la progression paraît ralentie.

2 Mise en œuvre de la plateforme

La plateforme sera développée en deux étapes, la première consistant en une version minimale, et la seconde proposant un développement plus complet intégrant notamment des aspects plus directement liés à la gestion de l'exécution concurrente et/ou répartie comme la tolérance aux pannes, le placement, l'équibrage de charge... Chaque étape donnera lieu à la production d'une version distincte.

2.1 Hidoop v0

Cette implantation sera composée de deux services :

— un service HiDFS (Hidoop Distributed File System). Il s'agit d'un système de gestion de fichiers répartis dans lequel un fichier est découpé en fragments, chaque fragment étant stocké sur un des nœuds du cluster.

— un service Hidoop contrôlant l'exécution répartie et parallèle des traitements map, la récupération des résultats et l'exécution du reduce.

Chacun de ces services devra proposer, outre une implémentation de l'API, une interface en ligne de commande pour la gestion des fichiers et des tâches.

Pour cette version,

— on pourra considérer que la plateforme est réservée à l'exécution d'une application unique. Le $RessourceManager$ sera donc réduit au suivi des ressources allouées, et une instance unique d'$ApplicationManager$ sera lancée.

— La tolérance aux pannes ne sera pas prise en compte : les blocs gérés par HDFS ne seront pas dupliqués, le $RessourceManager$ et le $NameNode$ n'auront pas de secours, et les pannes de nœuds ne seront pas gérées.

— L'espace de nommage des fichiers HDFS sera plat : en d'autres termes, la notion de répertoire ne sera pas implantée/proposée.

— La taille des blocs pourra être gérée de telle sorte qu'un enregistrement ne se trouve jamais à cheval sur 2 blocs.

— Les applications ne comporteront pas de shuffle, ni de combiner, et n'utiliseront qu'une tâche Reduce.

2.1.1 La gestion de format

Chaque bloc (*chunk*) contient une suite d'enregistrements, dont le format est spécifique à l'application. Pour chaque format, une classe doit être définie, qui permet de lire et écrire dans un fichier dans ce format. Ces classes de lecture/écriture de formats sont très importantes, car étant donné que les fichiers peuvent être coupés en fragments par HDFS, il faut qu'HDFS sache faire une coupure cohérente. En particulier, un fichier de structures ne peut pas être coupé au milieu d'une structure. Dans ce qui suit, on ne considère que 2 formats : le format texte ($LINE$) et le format Clé-Valeur (KV), mais d'autres formats peuvent être envisagés et introduits par la suite.

Une classe implantant un format implante l'interface Format.

```
public interface Format extends FormatReader, FormatWriter, Serializable {
        public enum Type { LINE, KV };
        public enum OpenMode { R, W };
        public void open(OpenMode mode);
        public void close();
        public long getIndex();
        /* position courante (octet) sur le fichier traite */
        public String getFname();
        public void setFname(String fname);
        /* fname identifie (dans le systeme de fichiers local)
         * le fichier/fragment sur lequel opere le Format */
}
```

```
public interface FormatReader {
        public KV read();
}

public interface FormatWriter {
        public void write(KV record);
}
```

Le format KV correspond au format des données intermédiaires produites durant l'exécution de l'application MapReduce.

```
public class KV {
        public static final String SEPARATOR = "<->";
        public String k;
        public String v;
        public KV() {}
        public KV(String k, String v) {
                super();
                this.k = k;
                this.v = v;
        }
        public String toString() {
                return "KV [k=" + k + ", v=" + v + "]";
        }
}
```

2.1.2 Le service HDFS

Le système de fichier est composé d'un démon (HdfsServer) qui doit être lancé sur chaque machine. Une classe HdfsClient permet de manipuler les fichiers dans HDFS. Voici un template de cette classe :

```
public class HdfsClient {
  public static void HdfsDelete(String hdfsFname) {...}
  public static void HdfsWrite(Format.Type fmt, String localFSSourceFname,
     int repFactor) {...}
  public static void HdfsRead(String hdfsFname, String localFSDestFname) {...}
  public static void main(String[] args) {
    // appel des methodes precedentes depuis la ligne de commande
  }
}
```

La classe HdfsClient peut être utilisée depuis la ligne de commande ou directement depuis du code pour lire, écrire ou détruite des fichiers :

- `HdfsWrite(Format.Type fmt, String localFSSourceFname, int repFactor)` permet d'écrire un fichier dans HDFS. Le fichier $localFSSourceFname$ est lu sur le système de fichiers local, découpé en fragments (soit de taille fixe, soit pour équilibrer la charge entre machines) et les fragments sont envoyés pour stockage sur les différentes machines. fmt est le format du fichier (Format.Type.LINE ou Format.Type.KV). $repFactor$ est le facteur de duplication des fragments ; pour cette version il sera considéré comme valant 1 (pas de duplication).
- `HdfsRead(String hdfsFname, String localFSDestFname)` permet de lire un fichier de nom $hdfsFname$ à partir de HDFS. Les fragments du fichier sont lus à partir des différentes machines, concaténés et stockés localement dans un fichier de nom $localFSDestFname$.
- `HdfsDelete(String hdfsFname)` permet de supprimer les fragments d'un fichier stocké dans HDFS.

Nous recommandons d'utiliser les sockets en mode TCP pour implanter la communication entre HdfsClient et HdfsServer. Chaque interaction entre HdfsClient et HdfsServer sera alors

initiée par l'envoi d'un message spécifiant la commande à exécuter par HdfsServer. Une syntaxe minimale pour ces messages pourrait être : Le code commande, suivi de la taille du premier nom de fichier, suivi du premier nom de fichier (suivi éventuellement (pour une version ultérieure) soit du facteur de réplication, soit de la taille du second nom de fichier, et du second nom de fichier) Le code commande pourra être défini par un type énuméré tel :

```
public enum Commande {CMD_READ, CMD_WRITE, CMD_DELETE};
```

2.1.3 Le service Hidoop

Le service Hidoop fournit le support pour l'exécution répartie. Un démon (Daemon) doit être lancé sur chaque machine. Nous proposons d'utiliser RMI pour la communication entre ce démon et ses clients. L'interface du démon est interne à Hidoop, donc libre à vous de la définir. Voici toutefois une proposition d'interface du démon, **pour la gestion des tâches** *Map* :

```
public interface Daemon extends Remote {
        public void runMap (Mapper m, Format reader, Format writer, CallBack
            cb)
                throws RemoteException;
}
```

Les paramètres sont les suivants :

- Mapper m : il s'agit du programme map à appliquer sur un fragment hébergé sur la machine où s'exécute le démon.
- Format reader : il s'agit du fichier (dans un format donné par la classe de reader) sur la machine où s'exécute le démon, contenant le fragment sur lequel doit être appliqué le map.
- Format writer : il s'agit du fichier (dans un format donné par la classe de writer) sur la machine où s'exécute le démon, dans lequel les résultats du map doivent être écrits.
- CallBack cb : il s'agit d'un objet de rappel, appelé lorsque l'exécution du map est terminée.

Pour lancer un calcul parallèle, Hidoop fournit une classe *Job*, implantant une interface $JobInterface$ et fournissant en particulier une méthode *startJob* :

```
public class Job {
        public void setInputFormat(Format.Type format) {...}
        public void setInputFname(String fname) {...}
        public void startJob (MapReduce mr) {...}
}
```

- MapReduce mr : correspond au programme MapReduce à exécuter en parallèle (voir modèle de programmation),
- Format.Type format : indique le format du fichier en entrée (Format.Type.LINE ou Format.Type.KV),
- String fname : le nom du fichier HDFS contenant les données à traiter.

Le comportement de startJob est de lancer des map (avec runMap) sur tous les démons des machines, puis attendre que tous les map soient terminés. Les map ont généré des fichiers locaux sur les machines qui sont des fragments. On peut alors récupérer le fichier global avec HDFS, puis appliquer le reduce localement. Le résultat final est un fichier dont le nom est dérivé de $fname$ (par exemple par l'ajout du suffixe "-res".)

2.1.4 Le modèle de programmation

Le modèle de programmation MapReduce permet de définir une classe implémentant l'interface MapReduce et fournissant les méthodes map() et reduce(). Une procédure map() peut lire les données (des KV) provenant de son fragment local avec le paramètre reader, et écrire ses résultats (des KV) dans un fragment local avec le paramètre writer. Une procédure reduce() lit également des KV avec reader et écrit des KV sur writer, mais Hidoop la fait opérer sur un fichier unique

et local à la machine de lancement du programme (ce fichier est le résultat des map extrait de HDFS).

```
public interface Mapper extends Serializable {
        public void map(FormatReader reader, FormatWriter writer);
}

public interface Reducer extends Serializable {
        public void reduce(FormatReader reader, FormatWriter writer);
}

public interface MapReduce extends Mapper, Reducer {
}
```

2.1.5 Configuration

Il sera sans doute utile de développer des outils permettant d'initialiser Hidoop à partir d'un fichier de configuration fournissant les nœuds et ports utilisés par les différents serveurs.

2.1.6 Exemples d'application

En annexe A, vous trouverez l'exemple d'application canonique de comptage des occurrences de mots dans un très grand fichier, programmé en MapReduce. En annexe B, vous trouvez le même programme en itératif.

2.2 Hidoop v1

La seconde version de Hidoop étendra la plateforme en s'attachant à développer un aspect relevant de la gestion de la concurrence et/ou de la répartition.

Pour cette version, néanmoins, on pourra continuer à considérer que la plateforme est réservée à l'exécution d'une application unique. Le *RessourceManager* sera donc réduit au suivi des ressources allouées, et une instance unique d'*ApplicationManager* sera lancée. Un bénéfice attendu de cette restriction est qu'il sera possible d'observer les caractéristiques de comportement d'une application, en termes d'usage des ressources et de temps d'exécution, données qu'il est plus difficile d'évaluer lorsque l'allocation de ressources doit être arbitrée entre plusieurs applications.

- pour HDFS, par exemple : gestion de la réplication des blocs, placement des blocs, fiabilisation du NameNode...
- et pour le moteur d'exécution, par exemple : placement, équilibrage de charge, gestion (à chaud) des pannes de DataNodes, ordonnancement des tâches...

2.3 Outils de supervision

Les applications tournant sur la plateforme Hidoop, et la plateforme Hidoop elle-même, devront être testées et évaluées de manière suffisamment complète. A cette fin, il sera utile de développer un certain nombre d'outils *simples*, comme :

- un interpréteur de commandes, permettant
  - de jouer des scénarios interactifs ou prédéfinis, pouvant permettre en ce qui concerne la seconde version de simuler des pannes ou des variations de charge ;
  - d'évaluer (sommairement) les performances des applications exécutées ;
- de consulter et de fixer des paramètres relatifs aux ressources utilisées comme la taille (maximale ou courante) de l'espace utilisé pour les calculs intermédiaires, la taille des files d'attente, le nombre de tâches map ou reduce actives ou échues, le facteur de duplication ;
- ...

Notez que cette liste n'est qu'indicative : elle n'est ni exhaustive, ni prescriptive. Cependant, il est attendu qu'un certain nombre de ces outils soient développés et utilisés.

3 Applications

La plateforme servira de support à l'exécution de différentes applications concurrentes. L'objectif sera d'une part d'évaluer différents choix de conception et d'architecture pour les applications concurrentes et d'autre part de permettre de comparer et valider les différentes versions de Hidoop.

3.1 Tests élémentaires

La validation de la plateforme s'appuiera sur des tests et scénarios de base afin d'en évaluer les aspects fonctionnels et les performances. Ces tests mettront en jeu des threads réalisant différentes configurations d'accès, sans que ces accès aient forcément une signification particulière.

Les tests fonctionnels reposeront sur la méthodologie et les outils de test – comme Junit – vus en première année, dans le cadre des enseignements de programmation, ainsi que sur la définition d'un ensemble de scénarios suffisamment représentatif de situations d'utilisation mettant en jeu des activités concurrentes. L'objectif de ces scénarios est de valider la cohérence et la vivacité des opérations sur l'espace de tuples.

Les tests de performances devront permettre de comparer les différentes versions développées et d'évaluer leurs limites en termes (quantitatifs) de volumes de données traitées ou conservées, de débit d'opérations concurrentes, de nombre de processus concurrents possibles, de temps d'exécution... Ces tests reposeront sur un jeu d'applications de référence *simples*, qui pourront être inspirées de ce qui a été fait dans le cadre des TPs sur les threads ou la concurrence.

Pour cette phase, le développement d'outils *simples* de supervision ou d'instrumentation paraît particulièrement approprié.

3.2 Schémas et problèmes de base

Une bonne manière de se familiariser avec le modèle de programmation Map-Reduce consiste à réaliser dans ce modèle des exercices simples et classiques traités précédemment avec d'autres outils :

- comptage de mots dans un texte
- analyses statistiques sur des journaux, des mesures...
- tri externe
- recherche d'expressions régulières (grep)

3.3 Expérimentation de stratégies/paramètres d'ordonnancement

Les applications que vous devrez développer se focaliseront sur la catégorie d'applications à laquelle le modèle Map-Reduce est destiné : le traitement intensif, mettant en jeu des masses de données.

Note : *Il est demandé de choisir* ***une***[6] *application parmi les exemples ci-dessous.*
Chaque application peut demander une petite recherche bibliographique, qui doit rester minimale. La page Moodle de l'UE (rubrique « Projet ») fournit, pour certaines applications, un point de départ, avec quelques éléments d'explication ainsi que quelques références.

La distribution d'Hadoop, disponible sur le site *Apache* propose un certain nombre d'exemples, qui pourront être adaptés à votre plateforme, et/ou étendus en vue de mesures de performances :

- calcul des premières décimales de π
  - par une méthode de quasi-Monte-Carlo (voir page Moodle de l'UE) ;
  - par la méthode de Bailey-Borwein-Plouffe ;
- applications de l'algorithme X de Knuth pour la résolution du problème de la couverture exacte (voir page Moodle de l'UE) :

6. à la rigueur deux, si vous avez du mal à choisir...

- résolveur de pentominos de bonne taille,
- résolveur de sodokus, du problème des 8 reines ; pour ces deux derniers, seule la solution récursive est donnée : il faudra donc passer à une version distribuée, sur des problèmes de plus grande taille.

Par ailleurs, vous pouvez envisager des applications moins amusantes mais plus classiques :

- évaluation de la popularité des pages Web à travers le référencement (page ranking), dont le principe est présenté sur la page Moodle de l'UE.
- regroupement de données en K groupes (voir page Moodle de l'UE)
- apprentissage automatique, ou toute autre application pertinente pour Map-Reduce, sous réserve de validation préalable lors des séances de suivi...

Le développement de l'application devrait permettre

- d'expérimenter des schémas de parallélisation par MapReduce,
- de compléter la plateforme par la prise en compte d'opérateurs de tri, shuffle, combiner, partitionnement,
- d'évaluer et réfléchir sur l'efficacité de différentes combinaisons de paramètres (nombre de tâches map, reduce, combiner, politiques d'ordonnancement),
- de valider les stratégies de tolérance aux pannes réalisées.

4 Modalités pratiques

4.1 Structure du projet

Le projet comporte 3 étapes. Les étapes sont décomposées de la manière suivante :

Etape 1

partie 1.1 HDFS v0,

partie 1.2 Hidoop v0

Outre les fonctionnalités de base, chacune de ces parties devrait voir ses fonctionnalités accessibles via une interface en ligne de commande, et offrir/prévoir des possibilités d'instrumentation.

Etape 2

partie 2.1 HDFS v1

partie 2.2 Hidoop v1

Etape 3 Développement et évaluation d'applications (et évaluation de la plateforme). Ces évaluations pourront utiliser l'une ou l'autre des versions développées lors de l'étape 2.

Bonus : *La note pour cette partie sera doublée si l'application peut être utilisée sur une version de Hidoop intégrant complètement les fonctionnalités développées pour les* ***deux*** *parties de l'étape 2*

4.2 Constitution des groupes

Projet réalisé en binôme (ou trinôme si l'organisation le nécessite, voir plus bas). Les binômes sont constitués au sein d'un même groupe de TD. Dans le cas où le groupe de TD comporte un nombre impair d'étudiants, un trinôme est constitué. Chaque groupe de TD doit avoir un nombre pair de binômes/trinôme. Dans le cas où ce nombre est impair, un binôme est réparti pour constituer deux trinômes. Les binômes/trinômes d'un groupe de TD sont répartis en un nombre égal de binômes/trinômes "A" et de binômes/trinômes "B".

Les "A" et les "B" sont enfin appariés en groupes. Un groupe est constitué par un "A" et un "B", avec la contrainte que l'effectif du groupe ne peut être 6.

La constitution des binômes/trinômes, des groupes et le choix des lettres sont libres, sous réserve du respect des contraintes précédentes. Les cas de blocage ou l'absence de choix seront réglés par tirage au sort.

Les "A" réaliseront les parties 1.1 et 2.2, et évalueront les parties 1.2 et 2.1.

Les "B" évalueront les parties 1.1 et 2.2, et réaliseront les parties 1.2 et 2.1.

L'étape 3 sera réalisée par le groupe complet.

4.3 Déroulement des étapes 1 et 2

Chacune des étapes 1 et 2 se déroule en trois temps :

- chaque groupe dispose d'une dizaine de jours pour réaliser la partie qui lui est affectée. A l'issue de ce travail, il remet une archive contenant :
  - un rapport provisoire succinct présentant l'architecture, les algorithmes des opérations essentielles, une explication claire des points délicats et de leur résolution.
  - le code complet de la partie réalisée.
- Ce travail n'est pas noté mais il est transmis, pour évaluation et validation, à l'autre binôme/trinôme du groupe. Cette étape d'évaluation consiste à :
  - valider le code fourni et en évaluer les performances par des tests élémentaires ;
  - réaliser une revue critique du code et des choix de conception effectués ;
  - proposer des améliorations, ou non. Les propositions d'amélioration, ou la validation du code et des choix existants doivent être étayés dans tous les cas.

  Cette évaluation aboutira à un rapport qui sera remis et transmis à l'autre binôme/-trinôme du groupe. Ce rapport est noté. **L'annexe C fournit quelques indications supplémentaires sur le travail d'évaluation.**

— Enfin, chaque binôme dispose d'une semaine pour intégrer (ou non) les remarques et propositions issues de l'évaluation. A l'issue de ce travail, il remet une archive contenant
  — un rapport succinct présentant l'architecture, les algorithmes des opérations essentielles, une explication claire des points délicats et de leur résolution.
  — le code complet de la partie réalisée.

  Ce travail est noté.

4.4 Echéancier

Note : *lorsqu'une semaine est indiquée, les livrables doivent être remis au plus tard* ***la veille du jour de la séance de suivi*** *de la semaine indiquée.*

16/10 constitution des groupes, binômes...
semaine du 6/11 remise des rapports et livrables provisoires de l'étape 1
14/11 remise des évaluations de l'étape 1
semaine du 20/11 remise des rapports et livrables finaux de l'étape 1 et présentation par le groupe du travail réalisé
semaine du 4/12 remise des rapports et livrables provisoires de l'étape 2
11/12 remise des évaluations de l'étape 2
18/12 remise des rapports et livrables finaux de l'étape 2
semaine du 8/1 remise des livrables de l'étape 3 ; présentation par le groupe de l'ensemble du travail réalisé

Les dates ci-dessus sont des dates fermes, car le travail des binômes est interdépendant.

Note : il est possible (et même recommandé) de démarrer l'étape $i + 1$ dès lors que les évaluations de l'étape i ont été réalisées.

4.5 Notation

Le poids respectif des étapes 1, 2, et 3 sera de 40%, 40%, 20%.

Les notes des étapes 1 et 2 seront attribuées par binôme/trinôme, et la note de l'étape 3 sera attribuée au groupe, sauf dysfonctionnement avéré [7].

Pour ce qui est des parties de réalisation, la notation portera sur le contenu du rapport définitif (dont les éléments ont été présentés plus haut), et sur nos propres évaluations, lectures et tests du code fourni. A ce propos, il est impératif que l'API qui vous est fournie soit strictement respectée, afin que nos tests puissent être opérationnels. Ainsi, le code des exemples qui vous sont fournis doit compiler et s'exécuter correctement *sans que vous y touchiez le moindre caractère.* Ces tests valident votre respect de l'API.

Pour ce qui est des parties d'évaluation, les deux éléments pris en compte seront
— le caractère complet de la démarche de validation du code, suivant les lignes présentées en 3.1
— la pertinence de la revue critique et des propositions d'amélioration présentées, par rapport à notre propre évaluation du rapport provisoire, dont nous disposerons : si les rapports provisoires ne sont pas notés en eux-mêmes, ils servent de base pour noter les évaluations.

L'étape 3, réalisée en groupe sera évaluée sur le code et les éléments de validation présentés.

4.6 Fournitures

Les différents documents et fournitures utiles sont (ou seront) mis en ligne sur la page `Moodle` de l'enseignement, au fur et à mesure de la progression du projet, et notamment :
— une liste des groupes, binômes, trinômes
— une archive contenant
  — l'arborescence de fichiers dans laquelle devront se trouver les livrables ;
  — le code source des interfaces, classes et squelettes de base, exemples élémentaires ;

7. En cas (exceptionnel) d'anomalies importantes constatées dans le fonctionnement/travail du groupe ou du binôme/trinôme, les notes seront individualisées.

- un fichier d'entrée pour l'application de comptage de mots, ainsi que le fichier résultat correspondant.
  Votre implémentation devra permettre d'exécuter cette application, qui devra obligatoirement donner exactement le même résultat à partir de ce fichier d'entrée.
- pour les applications,
  - le code source des exemples fournis avec les distributions d'Hadoop, destiné à servir de points de départ ou de source d'inspiration pour les applications Hidoop. **Attention : il s'agit d'exemples pour Hadoop** ; les interfaces appelées sont similaires mais différentes (elles sont sensiblement plus complètes/complexes).
  - la description du principe et des algorithmes de base de quelques applications envisageables, éventuellement accompagnée de classes utiles.

Annexes

Annexe A (comptage de mots en MapReduce)

```
public class MyMapReduce implements MapReduce {
        private static final long serialVersionUID = 1L;

        // MapReduce program that computes word counts
        public void map(FormatReader reader, FormatWriter writer) {

                Map<String,Integer> hm = new HashMap<>();
                KV kv;
                while ((kv = reader.read()) != null) {
                        StringTokenizer st = new StringTokenizer(kv.v);
                        while (st.hasMoreTokens()) {
                                String tok = st.nextToken();
                                if (hm.containsKey(tok)) hm.put(tok,
                                    hm.get(tok).intValue()+1);
                                else hm.put(tok, 1);
                        }
                }
                for (String k : hm.keySet()) writer.write(new
                    KV(k,hm.get(k).toString()));
        }

        public void reduce(FormatReader reader, FormatWriter writer) {
                Map<String,Integer> hm = new HashMap<>();
                KV kv;
                while ((kv = reader.read()) != null) {
                        if (hm.containsKey(kv.k)) hm.put(kv.k,
                            hm.get(kv.k)+Integer.parseInt(kv.v));
                        else hm.put(kv.k, Integer.parseInt(kv.v));
                }
                for (String k : hm.keySet()) writer.write(new
                    KV(k,hm.get(k).toString()));
        }

        public static void main(String args[]) {
                Job j = new Job();
                j.setInputFormat(Format.Type.LINE);
                j.setInputFname(args[0]);
                long t1 = System.currentTimeMillis();
                j.startJob(new MyMapReduce());
                long t2 = System.currentTimeMillis();
                System.out.println("time in ms ="+(t2-t1));
                System.exit(0);
        }
}
```

Annexe B (comptage de mots en itératif)

```
public class Count {

        public static void main(String[] args) {
                try {
                        long t1 = System.currentTimeMillis();

                        HashMap<String,Integer> hm = new
                            HashMap<String,Integer>();
                        LineNumberReader lnr = new LineNumberReader(
                                        new InputStreamReader(
                                        new
                                            FileInputStream(Project.PATH+"data/"+args[0])));
                        while (true) {
                                String l = lnr.readLine();
                                if (l == null) break;
                                StringTokenizer st = new StringTokenizer(l);
                                while (st.hasMoreTokens()) {
                                        String tok = st.nextToken();
                                        if (hm.containsKey(tok))
                                                hm.put(tok,
                                                    hm.get(tok).intValue()+1);
                                        else hm.put(tok, 1);
                                }
                        }
                        BufferedWriter writer = new BufferedWriter(
                                        new OutputStreamWriter(new
                                            FileOutputStream("count-res")));
                        for (String k : hm.keySet()) {
                                writer.write(k+"<->"+hm.get(k).toString());
                                writer.newLine();
                        }
                        writer.close();
                        lnr.close();
                        long t2 = System.currentTimeMillis();
                        System.out.println("time in ms ="+(t2-t1));
                } catch (Exception e) {
                        e.printStackTrace();
                }
        }

}
```

Annexe C : Revue (évaluation) du travail réalisé par le binôme complémentaire

Cette sous-étape, notée, devrait être organisée en deux parties. Le rapport total ne devrait pas dépasser 4 à 5 pages (plus les annexes éventuelles)

Partie technique

Présentation et résultats des tests (le détail étant fourni en annexe) Correction : validation (ou non) des fonctionnalités Performances, lorsque cela est pertinent (c'est le cas pour une application de calcul concurrent, ou pour le noyau ; c'est moins évident pour un analyseur syntaxique ou un utilitaire) Bugs détectés, et propositions de correction éventuelles Commentaires et propositions sur la qualité du code (rester synthétique, se limiter aux points particulièrement flagrants/gênants. Inutile de développer s'il n'y a pas de remarque particulière)

Synthèse (1 à 2 pages)

Evaluer le travail en termes de

- Correction : est ce que le produit fonctionne correctement ? Les résultats sont ils justes ?
- Complétude : est ce que tous les points de la spécification ont été abordés et traités ?
- Pertinence : le travail présenté répond-il à ce qui est demandé ? les réponses apportées sont-elles appropriées ?
- Cohérence : le résultat obtenu a-t-il une structure, une architecture, une logique claires, les fonctions proposées et réalisées sont-elles bien complémentaires ou au contraire se recoupent-elles, font doublon ? C'est le point le plus difficile et subjectif, car au fond il s'agit de juger de la qualité d'une conception.

Notez que ces critères (correction complétude, pertinence, cohérence) nous serviront de base pour évaluer vos rendus, y compris votre revue d'évaluation

Enfin proposer des pistes d'amélioration, s'il y a lieu, en indiquant des choix de conception alternatifs qui vous paraîtraient préférables, ou intéressants à envisager. La présentation de ces pistes peut prendre la forme d'une table à deux colonnes :

1. Point d'amélioration. Par exemple : Remplacer un (gros) mutex associé à chaque accès à l'espace de tuple par une synchronisation adaptée au type des opérations d'accès.
2. Motivation de l'amélioration proposée. Par exemple : augmenter le degré de parallélisme

Les pistes d'amélioration n'ont pas à être nombreuses (entre 1 et 3), mais elles doivent être pertinentes. Si vous ne voyez pas de pistes d'amélioration, indiquez les choix de conception que vous avez envisagés, et dites pourquoi la solution proposée vous semble convenable.