Vers une plate-forme MapReduce tolérant les fautes byzantines

*** Luciana Arantes, Jonathan Lejeune, Madeleine Piffaretti, Olivier Marin, Pierre Sens et Julien Sopena**
**** Alysson N. Bessani, Vinicius V. Cogo, Miguel Correia, Pedro Costa et Marcelo Pasin**
***** Fabricio Silva**

* *LIP6 - Université de Pierre et Marie Curie - CNRS - INRIA*

4, Place Jussieu 75252 Paris Cedex 05, France.

** *LASIGE - Universidade de Lisboa, Faculdade de Ciencias*

Campo Grande 1749-016 - Lisboa, Portugal.

*** *Centro Tecnológico do Exército*

Av. das Américas, 28705 - Rio de Janeiro, Brasil.

RÉSUMÉ. *Les pannes arbitraires sont inhérentes aux calculs massivement parallèles tels que ceux visés par le modèle MapReduce ; or les implémentations courantes du MapReduce ne fournissent pas d'outils permettant de tolérer les fautes byzantines. Il est donc impossible de certifier l'exactitude de résultats obtenus au terme de traitements longs et coûteux. Nous présentons dans cet article une architecture permettant de répliquer les tâches dans le modèle MapReduce afin de garantir l'intégrité des traitements et d'isoler les tâches défaillantes. Une étude préliminaire de performances a permis d'évaluer certains mécanismes liés à la réplication tandis qu'une deuxième, effectuée avec un prototype implémentant l'ensemble de l'architecture proposée, a permis d'en valider certains choix en montrant qu'il est possible de minimiser le surcoût de la tolérance aux fautes byzantines.*

ABSTRACT. *Byzantine faults are inherent in massive parallel computation, including those based on the MapReduce model. Yet, the current MapReduce framework implementations do not tolerate Byzantine failures. Therefore, it is not possible to verify if the final results of a MapReduce application are correct. We present in this article a MapReduce architecture where tasks are replicated aiming at ensuring the correctness of task execution results and isolation of faulty tasks. A preliminary performance study has evaluated some of our proposed replication mechanisms while a second one, conducted on top of a prototype that implements our architecture, has validated some of our choices, showing that it is possible to minimize the cost of Byzantine fault tolerance.*

MOTS-CLÉS : *MapReduce, fautes byzantines, Hadoop, HDFS*

KEYWORDS: *MapReduce, Byzantine fault tolerance, Hadoop, HDFS*

1. Introduction

Le modèle MapReduce est aujourd'hui l'un des modèles de programmation parallèle les plus utilisés. Définissant une architecture *Maître-Esclave*, dans laquelle s'exécute une succession d'ensemble de tâches indépendantes, elle permet le traitement parallèle de grandes masses de données. Nous proposons dans cet article d'intégrer aux plateformes existantes une série de mécansimes permettant d'assurer une tolérance aux fautes byzantines.

En effet, la gestion des pannes arbitraires est un élément essentiel dans les applications réparties, et à plus forte raison dans les calculs massivement parallèles (Rodrigues *et al.*, 2007) puisque la probabilité de comportements byzantins augmente proportionnellement avec le nombre de processus impliqués. Or, les implémentations du MapReduce, comme celle proposée avec la plate-forme Hadoop (Apache, n.d.b), se contentent de reprendre un calcul lorsque le résultat tarde à émerger. En effet, le modèle MapReduce est généralement associé aux centres de traitements de données (*datacenters*) mis en place par de grandes entreprises informatiques telles que Google ou Amazon. Ces datacenters, en intégrant de la redondance, offrent des garanties vis-à-vis des fautes potentielles en assurant essentiellement la disponibilité des ressources et des données (Barroso *et al.*, 2009). Cependant, ils restent sensibles aux erreurs de calcul produites soit par des processus malveillants soit par des fautes matérielles temporaires. Par ailleurs, il est tout à fait concevable de déployer des calculs massivement parallèles sur des architectures plus ouvertes et moins coûteuses en termes de maintenance : le "Cloud Computing" (Armbrust *et al.*, 2009) et les architectures orientées service (SOA) (Valipour *et al.*, 2009) visent explicitement à répartir les charges de calcul sur un grand nombre de machines de manière décentralisée. Cette approche ôte toute maîtrise de la plate-forme physique, et donc toute garantie quant aux résultats retournés une fois le code déployé.

Cet article propose une solution pour tolérer les fautes byzantines dans le modèle MapReduce. Notre approche s'intègre à l'implémentation Hadoop, qui a le double avantage d'être open-source et d'être la plus utilisée actuellement. Nos contributions principales sont les suivantes :

– Nous proposons une architecture permettant de répliquer des calculs dans le modèle MapReduce. Les résultats des calculs peuvent ainsi être comparés pour garantir l'intégrité des traitements et isoler les processus défaillants.

– Une étude préliminaire de performances a permis d'évaluer certains mécanismes, et plus particulièrement d'identifier plusieurs bonnes pratiques liées à la réplication dans le modèle MapReduce.

– Une étude effectuée avec un prototype implémentant l'ensemble de l'architecture proposée a permis d'en valider les choix en montrant qu'il était ainsi possible de minimiser le surcoût de la tolérance aux fautes byzantines.

La section 2 présente le modèle de programmation MapReduce ainsi que l'architecture Hadoop. Ensuite, la section 3 détaille notre solution de réplication de calculs

pour la gestion des fautes byzantines et justifie, performances à l'appui, les choix exprimés lors de la conception de notre architecture. Puis, nous présentons quelques détails clés de l'implémentation de notre plate-forme à la section 4, avant d'en étudier les performances générales à la section 5. Enfin, nous situons notre travail par rapport à l'état de l'art dans le domaine en section 6 avant de conclure.

2. MapReduce

Le MapReduce est un modèle de programmation associé à une implémentation proposée initialement par Google (Dean *et al.*, 2004). Il permet le traitement et la génération de données volumineuses.

Le format des données en entrée est spécifié par l'utilisateur et dépend des applications. La sortie est un ensemble de couples $< cl, valeur >$. L'utilisateur décrit son algorithme en utilisant les deux fonctions *Map* et *Reduce*. La fonction *Map* prend un ensemble de données et produit une liste intermédiaire de couples $< cl, valeur >$. La fonction *Reduce* est appliquée à toutes les données intermédiaires et fusionne les résultats ayant la même clé. Un pseudo-code classique de ces fonctions est donné ci-dessous. Il s'agit d'un exemple qui compte le nombre d'occurrences de chaque mot dans un ensemble de documents.

```
void Map(key, string value) {
    // key: document name
    // value: document contents
    for each word w in value {
        EmitIntermediate(w, "1");
    }
}
```

```
void Reduce(string key, list<string> values) {
    // key: a word
    // values: a list of contents
    int count = 0;
    for each v in values {
        count += StringToInt(v);
    }
    Emit(key, IntToString(count));
}
```

Un des principaux avantages de ce modèle est sa simplicité. Chaque *Job* MapReduce soumis par un utilisateur est une unité de travail composée de données en entrée, des deux fonctions *Map* et *Reduce* et d'informations de configuration. Le *Job* est alors automatiquement parallélisé et exécuté sur une grappe ou sur un ensemble de datacenters. La figure 1 présente l'architecture générale (White, 2009). La donnée en entrée est découpée en M unités notées *splits*. Chaque *split* est traité en parallèle par une tâche *Map* invoquant la fonction *Map*. Les clés intermédiaires produites par les tâches *Map* sont divisées en R fragments, appelés *partitions*, qui sont distribués aux tâches *Reduce*. Les données des fragments reçus par une tâche *Reduce* sont alors triées par clé et traitées par la fonction *Reduce*. Comme les tâches *Map*, les tâches *Reduce* sont exécutées en parallèle.

Les données initiales et celles produites à la fin d'un MapReduce sont stockées dans un système de stockage distribué tolérant aux fautes tel que HDFS(White, 2009)

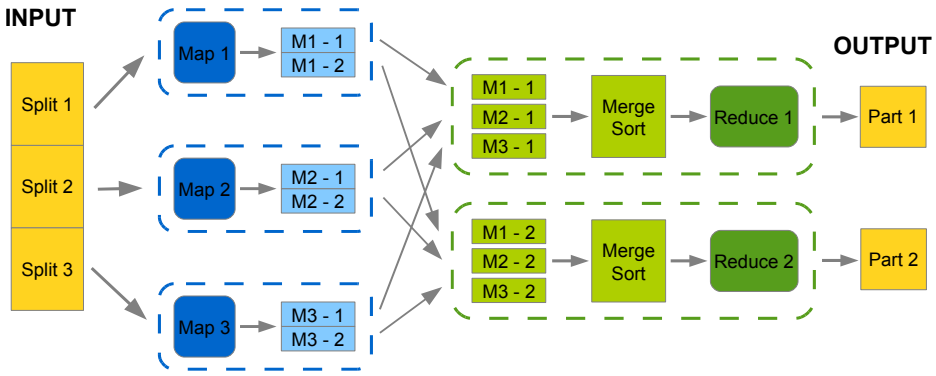


Figure 1. *Flot de données du MapReduce*

ou Google File System (Dean *et al.*, 2004). Les résultats intermédiaires produits par les tâches *Map* sont en revanche uniquement gardés sur les disques des nœuds exécutant les tâches sans mécanisme particulier pour tolérer les fautes.

Les implémentations courantes reposent sur une architecture maître-esclave. Un MapReduce est soumis par l'utilisateur à un nœud maître qui choisit des nœuds esclaves libres et leur attribue des tâches *Map* ou *Reduce* à exécuter. Les fautes franches de nœud sont traitées automatiquement. Si un nœud est suspecté d'être défaillant, le maître relance ses tâches sur une nouvelle machine. En revanche, les défaillances du maître ne sont pas prises en compte. D'autre part, si un nœud exécutant des tâches est peu performant, le maître peut lancer de façon spéculative une copie de ses tâches sur un autre machine pour obtenir un résultat plus rapidement. Le mécanisme de spéculation consiste donc à ordonnancer une réplique supplémentaire si le calcul de la tâche n'a pas été effectué dans un délai pré-défini.

Hadoop (White, 2009) est une implémentation open-source de MapReduce largement utilisée. Elle a été proposée par Apache et utilise le système de stockage HDFS (Hadoop Distributed File System). Dans HDFS, chaque fichier est découpé dans un ensemble de blocs de taille fixe (e.g., 64Mo) stockés dans des unités indépendantes sur les disques de la plate-forme. HDFS fournit une haute disponibilité et la fiabilité en répliquant chaque donnée avec un degré de réplication fixé par défaut à 3.

Pour contrôler l'exécution des tâches, Hadoop définit un maître unique, le *JobTracker*, qui gère l'état des *Job* soumis et ordonnance l'exécution des tâches (figure 2). Sur les machines esclaves, un processus, le *TaskTracker*, contrôle la disponibilité des créneaux d'exécution de cette machine. Pour améliorer les performances, les phases de calcul et les entrées/sorties peuvent être recouverts en exécutant plusieurs tâches *Map* ou *Reduce* en parallèle sur la même machine esclave. Chaque fois qu'un *TaskTracker* détecte un créneau disponible sur sa machine, il contacte le JobTracker pour réclamer une nouvelle tâche. S'il s'agit d'une tâche *Map*, le *JobTracker* prend en compte la localisation de l'esclave et choisit une tâche dont la donnée en entrée (son

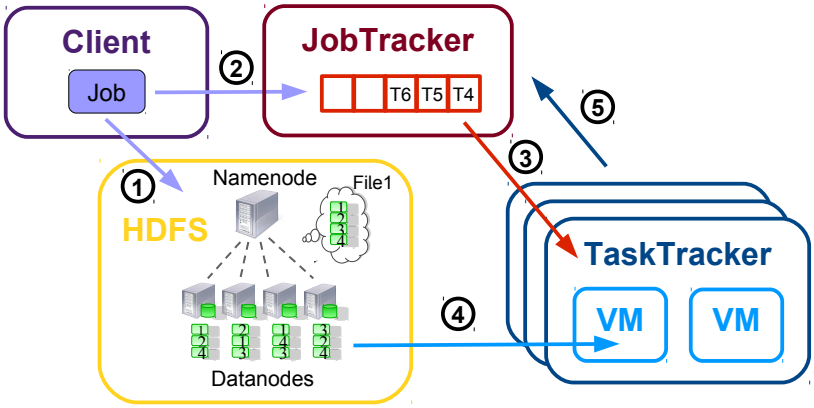


Figure 2. *Architecture de Hadoop*

"split") et la plus proche possible de la machine du TaskTracker. En revanche pour les tâches *Reduce*, il n'est pas possible de prendre en compte la localité des données.

3. MapReduce tolérant les fautes byzantines : architecture proposée

Nous proposons de traiter des pannes byzantines sur les *TaskTrackers*. Nous supposons que le *JobTracker* est fiable (comme proposé par *Hadoop* (White, 2009)), ainsi que le *HDFS*. En effet, pour ce dernier il existe dans la littérature des solutions qui permettent d'y assurer l'intégrité des données (Clement *et al.*, 2009). L'utilisation de *HDFS* dans notre architecture sera discutée dans la section 3.2. Enfin, on considère que les clients sont toujours corrects.

Notre approche repose sur l'utilisation de votes majoritaires sur le résultat de chaque tâche *Map* et *Reduce*. Si l'on considère que f est le nombre maximal de fautes pour une tâche, au moins $f+1$ résultats sur $2f+1$ répliques d'une tâche seront identiques. Autrement dit, il suffit d'avoir $f+1$ fois le même résultat pour le considérer comme correct. En pratique, on peut envisager deux approches : (1) ordonnancer directement $2f+1$ répliques en même temps pour chaque tâche et arrêter (supprimer des listes d'ordonnancement) les répliques toujours en cours lorsque l'on a obtenu $f+1$ résultats identiques ou (2) ne commencer que par $f+1$ répliques puis, le cas échéant, lancer des répliques supplémentaires. Le choix entre ces deux méthodes est discuté en section 3.1.

La méthode du vote majoritaire implique aussi un autre choix pour l'architecture, celui du processus qui fera les comparaisons des résultats. Nous proposons ici de valider les résultats directement sur le *JobTracker*. Ce choix présente deux avantages : d'une part il simplifie l'arrêt ou l'ajout de nouvelles répliques si nécessaire, d'autre part il évite de multiplier les validations puisque le *JobTracker* est considéré

comme fiable. Ce choix pose néanmoins un problème car dans le framework original de Hadoop MapReduce les résultats intermédiaires des différentes tâches de *Map* sont stockés en local sur les *TaskTrackers*. Une stricte comparaison sur le *JobTracker* impliquerait donc une forte charge réseau, la taille des résultats des *Maps* pouvant, suivant les applications, être relativement volumineux. Nous proposons donc de mettre en place un mécanisme reposant sur l'utilisation de *sommes de contrôles* (*digests*) de façon à minimiser le surcoût en transferts ainsi que le temps de comparaison.

Le principe de notre approche est le suivant : à la fin de l'exécution d'une tâche *Map* m, le *TaskTracker* calcule le *digest* du résultat de la tâche m (*digest global*) ainsi qu'un *digest* pour chacune des R partitions de ce résultat. L'ensemble de ces *digests* est ensuite envoyé par le *TaskTracker* au *JobTracker*. À chaque réception, ce dernier compare le *digest global* à ceux déjà reçu pour la tâche m et, dès lors qu'il obtient $f+1$ *digest* résultats identiques, il considère le résultat comme valide et enregistre les *digests* de chaque partition de la tâche m.

La phase de *Reduce* commence lorsque le *JobTracker* a validé l'ensemble M des résultats produits par les tâches *Map*. Comme dans le framework initial, il va alors assigner les tâches *Reduce* (et les répliques) aux *TaskTrackers*. Nous proposons d'ajouter au message d'assignation l'ensemble des *digests* des partitions. Pour chacun de ces *digests* la tâche *Reduce* r va récupérer une copie valide de la partition correspondante : elle charge une copie depuis l'un des $f+1$ *TaskTrackers* qui ont calculé la tâche *Map* validée par le *JobTracker*, calcule son *digest* puis le compare avec l'un des *digests* reçus du *JobTracker*. On peut ainsi détecter une corruption de partition après validation (les deux *digests* ne correspondent pas). Dans ce cas, le *TaskTracker* exécutant le *Reduce* charge d'autres copies jusqu'à obtenir une copie valide. Notons que par hypothèse il ne peut y avoir plus de f corruptions ; au moins une des $f+1$ copies correspondra donc au *digest* reçu.

Le calcul du *Reduce* r commence lorsque le *TaskTracker* en charge de cette tâche a récupéré et validé toutes les partitions nécessaires. Les résultats des *Reduces* sont ensuite validés sur le *JobTracker* par le mécanisme de *digest*, le job ne se terminant que lorsque le *JobTracker* a validé tous les résultats des *Reduces* ($f+1$ fois le même *digest* pour chaque tâche *Reduce*).

3.1. *Gestion des répliques*

Comme nous l'avons vu précédemment, deux méthodes sont envisageables pour obtenir $f+1$ résultats identiques par tâche : commencer par lancer $f+1$ répliques puis en ajouter dynamiquement lorsque les réponses diffèrent ou lancer directement $2f+1$ répliques puis stopper les répliques devenues inutiles lorsque l'on a obtenu les $f+1$ réponses nécessaires. Dans cette section nous allons étudier ces deux approches.

Pour commencer détaillons le mécanisme de lancement d'une tâche dans la plateforme *Hadoop*. Lorsqu'il décide de lancer une tâche, le *JobTracker* l'ajoute dans une liste de tâches à ordonnancer. Plusieurs implémentations d'ordonnanceur sont dispo-

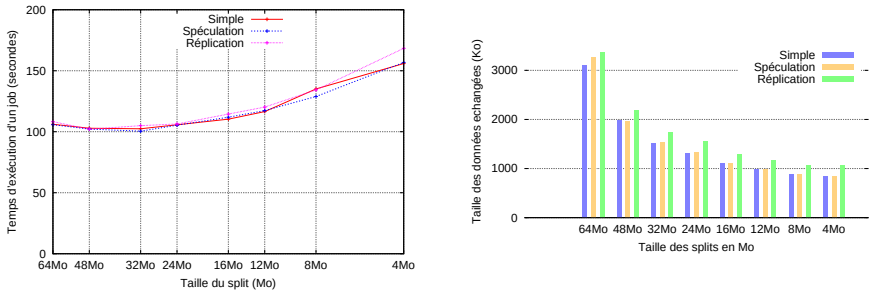
nibles mais leur principe reste le même : il parcourt la liste et affecte si possible les tâches à des *TaskTrackers* dont la charge (nombre de tâches à exécuter) est faible. Chaque ordonnanceur possède alors ses propres critères de choix des *TaskTrackers* : niveau de charge, utilisation de la localité des données, etc.

Chaque tâche est donc ajoutée une fois dans l'un des créneaux disponibles des *TaskTrackers*, mais *Hadoop* propose aussi d'ajouter un mécanisme dit de *spéculation* pour tolérer les pannes franches, comme décrit dans la section 2. Ce mécanisme de *spéculation* est en fait relativement proche de la première méthode proposée, puisqu'il s'agit d'ajouter dynamiquement des répliques supplémentaires. Mais cette approche, développée dans *Hadoop* pour tolérer un nombre non borné de pannes franches, sera-t-elle efficace pour tolérer au plus f fautes byzantines ? Ne vaut-il pas mieux pour gagner en réactivité ordonnancer directement $2f + 1$ répliques ? Pour répondre à ces questions, nous avons comparer le surcoût de l'ordonnancement de toutes répliques avec la suppression des répliques inutiles par rapport à l'approche spéculative à base de réplication dynamique.

Le mécanisme de validation du résultat étant orthogonal au mécanisme de réplication, nous étudierons ce dernier en restreignant le problème aux pannes franches (une étude du coût de la réplication dans le cas des fautes byzantines sera présentée à la section 5). En effet, si ce type de fautes nécessite de répliquer les calculs pour pallier à la défaillance d'un nœud, il ne nécessite pas de valider le résultat obtenu : l'obtention d'au moins un résultat suffit. Nous avons donc implémenté une version d'*Hadoop* (notée *Hadoop* avec réplication) permettant de tolérer f pannes franches en ordonnançant initialement $f + 1$ répliques de chaque tâche et supprimant les répliques inutiles dès qu'un résultat a été calculé. Nous avons ensuite comparé ses performances avec une version d'*Hadoop* utilisant la spéculation et la version simple (sans spéculation, ni réplication). Pour cette étude nous avons lancé un benchmark classique composés de 15 *Job WorldCount* sur un fichier d'1Go dans une plate-forme composée d'un *JobTracker* et de 10 *TaskTrackers* (1 par machine d'un cluster dédié). f a été fixé à 2, le nombre de *Reduces* à 10 et les résultats présentés sont une moyenne excluant le premier et le dernier *Job* de façon à observer les phénomènes en régime stationnaire.

Lors de ces expériences, nous avons fait varier la taille des *splits* de $4Mo$ à $64Mo$. Ainsi, les figures 3(a) et 3(b) présentent pour chacun de ces *splits* respectivement le temps moyen pour calculer un *Job* et la quantité de messages de contrôle (assignation, localisation des données, etc) nécessaires.

L'étude de ces figures montre deux choses en absence de panne : premièrement, la spéculation n'entraîne pas une augmentation significative du temps d'exécution et de la charge réseau et ce quelle que soit la taille du *split* ; deuxièmement la réplication influence peu le temps de calcul pour des tailles de *split* supérieures ou égales à $24Mo$ et l'augmente progressivement lorsque la taille du split tend à se rétrécir. Le surcoût réseau dû à la réplication est lui constant en volume mais son importance relative tend à diminuer lorsque le nombre de *splits* augmente. Il est intéressant de remarquer que ce surcoût devient négligeable (par rapport à la spéculation) si l'on considère le réglage



(a) Temps d'exécution du MapReduce

(b) Volume des données échangées

Figure 3. *Comparaison des stratégies de réplication*

d'*Hadoop* par défaut de $64Mo$ et qu'il reste peu important pour un split de $32Mo$ qui correspond au réglage optimum, en termes de temps d'exécution.

Ces résultats montrent donc dans ce cadre l'utilisation de la réplication de tâche impacte peu les performances du système. Ceci s'explique pour deux raisons :

– l'assignation des tâches aux *TaskTrackers* par le *JobTracker* ne constitue pas un mécanisme à part entière. Il ne génère pas de message supplémentaire, mais se superpose (*piggybacking*) à certains messages de contrôle que s'échangent périodiquement le *JobTracker* et le *TaskTracker*. L'augmentation du nombre de répliques ne génère donc pas de message d'assignation supplémentaire.

– les tâches ne sont pas assignées sur des *TaskTrackers* oisifs mais sont assignées à l'un des créneaux disponibles (*slot*) d'un *TaskTracker*. Par conséquent, il est possible qu'une réplique soit assignée puis annulée avant même d'avoir commencé. Nous avons donc modifié l'ordonnanceur de façon à allouer en priorité les répliques des tâches qui ont le moins de répliques déjà assignées. Dès que les tâches sont suffisamment longues, seule une réplique est réellement exécutée. On comprend alors pourquoi le surcoût augmente quand la taille du *split* devient trop petite.

Cette étude de performances nous a montré qu'il était possible d'utiliser la réplication initiale totale sans craindre pour les performances du système en l'absence de panne. Cette approche permettra entre autre d'optimiser le temps de réaction en cas de conflit dans les résultats, puisque les répliques supplémentaires seront déjà ordonnancées.

3.2. *Utilisation d'un HDFS tolérant aux fautes byzantines*

L'architecture d'*Hadoop* repose sur l'utilisation du système de fichiers *HDFS* (voir section 2). La conception d'un mécanisme de tolérance aux fautes byzantines peut donc passer par l'utilisation d'un *HDFS* garantissant la pérennité des données en présence d'éléments byzantins (*BFT-HDFS*) (Clement *et al.*, 2009).

Dans la plate-forme originale, *HDFS* n'est utilisé que pour stocker les données d'entrée et le résultat final du *Job* ; toutes les autres données sont stockées en local sur les disques des différents *TaskTrackers*. Une solution serait donc d'étendre l'utilisation d'un *BFT-HDFS* pour fiabiliser tout ou une partie de ces résultats intermédiaires.

Pour mesurer le surcoût d'une telle approche, nous avons instrumenté le code original de la plate-forme et mené une étude quantitative sur l'importance des transferts des données locales entre les *Maps* et les *Reduces*. Ainsi, nous avons ajouté des sondes dans les tâches *TaskTracker* pour mesurer la durée des différentes phases lors du calcul d'une tâche de type *Reduce*. La figure 4 indique la durée du temps passé à télécharger les différentes partitions nécessaires au calcul de la fonction *Reduce* (noté *download*), le temps passé à trier les données (noté *sort*) et le temps de calcul à proprement parler (noté *reduce*).

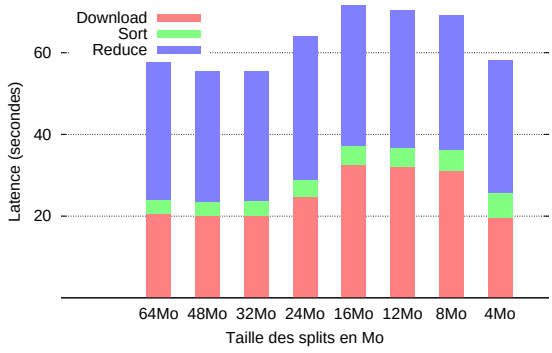


Figure 4. *Durées des phases d'un Reduce*

Les résultats montrent que la phase de *download* constitue une part importante de la tâche. Par conséquent, une simple utilisation d'un *BFT-HDFS* pour fiabiliser les données nécessaires au calcul entraînerait un surcoût relativement important. Une solution permettant de l'amortir serait d'exploiter la réplication pour augmenter la localité des données. Cependant, cette solution demanderait à réimplémenter tous les ordonnanceurs, car ceux-ci n'exploitent pas la localité pour l'assignation des *Reduces*.

Il faudrait aussi modifier *BFT-HDFS* de façon à synchroniser tous les placements des *Maps* en regroupant les partitions utilisées par un même *Reduce*. En effet, les reduces utilisent une combinaison de partitions produites plusieurs *Maps*.

L'utilisation d'un *BFT-HDFS* reste cependant utile pour fiabiliser les entrées et les résultats du *Job*. La validation du résultat final, enregistré sur *BFT-HDFS*, sera faite par le *JobTracker* à partir du *digest* validé. Cette solution a toutefois quelques limites, notamment si l'on considère des applications utilisant un chaînage de plusieurs MapReduces (Olston *et al.*, 2008). En effet, dans ce cas les résultats des Reduces, servant d'input aux *Map* de la phase suivante, sont laissées en local sur les disques des *TaskTrackers*. Une solution serait de réutiliser notre approche basée sur les *digests*, en conservant $f + 1$ copies des résultats *Reduces* ainsi que les *digests* associés.

Nous avons donc, dans un premier temps, choisi de ne pas étendre l'utilisation du *BFT-HDFS* aux données intermédiaires. En contre-partie, nous proposons de conserver localement $f+1$ copies (validées par le *JobTracker*) de tous les *Maps*, et ce jusqu'à validation de la terminaison du *Job*. Ainsi, on se prémunit contre toute corruption, après validation, des résultats intermédiaires.

4. MapReduce tolérant les fautes byzantines : implémentation

La présente section s'attache à détailler des points d'implémentation permetant la mise en œuvre des mécanismes proposés dans la section précédente. Ainsi nous détaillerons ci-dessous les modifications apportées à la plate-forme *Hadoop* pour d'une part intégrer la réplication des tâches, et d'autre part permettre la validation des résultats.

Réplication des tâches

Notre implémentation requiert un mécanisme de réplication des tâches pour pouvoir tolérer les fautes. Pour cela, nous avons préféré adopter une solution peu intrusive consistant à instancier chaque réplicat comme une tâche indépendante. Cette approche évite de modifier en profondeur le code du *JobTracker*. Elle simplifie également le contrôle des exécutions redondantes : la terminaison d'un des réplicats n'a pas d'impact sur l'exécution des autres réplicats associés à la même tâche.

Cette approche nécessite cependant de conserver une trace des exécutions redondantes pour pouvoir comparer et valider leurs résultats. A cet effet, notre implémentation repose sur une simple convention de nommage relative aux identifiants de tâches : une valeur entière est ajoutée en suffixe pour distinguer chacun des réplicats associés à une même tâche. Cette convention de nommage a le mérite de simplifier le placement des réplicats sur les *TaskTrackers* : en effet il est impératif de ne pas exécuter deux réplicats associés à une même tâche sur un unique nœud potentiellement malicieux. Pour cela, il faut toutefois intégrer une modification de la politique de placement du *JobTracker*.

Validation des traitements

Lors de leur exécution, tous les réplicats d'une même tâche *Map* récupèrent les données du même split. Cependant, comme expliqué en Section 3, chacun d'eux va produire localement des données résultats ainsi que des *digests* associés à ces résultats. Tous les *digests* sont remontés par les *TaskTrackers* auprès du *JobTracker* pour être validés. Lorsqu'un *digest* associé à une tâche est validé par le *JobTracker*, il se peut que des réplicats de cette tâche soient encore en train de s'exécuter. Le *JobTracker* notifie les *TaskTrackers* des réplicats qui n'ont pas encore renvoyé leur *digest* pour que ces réplicats soient étiquetés comme spéculatifs. Ce mécanisme laisse la main à la plate-forme *Hadoop* pour l'annulation des exécutions devenues inutiles.

La validation d'un *digest* entraîne également sa transmission aux réplicats initialement assignés aux tâches *Reduce*. Par comparaison des *digests* avec les partitions récupérées auprès des réplicats de tâches *Map*, notre implémentation guarantit l'intégrité des données traîtées en entrée de chaque *Reduce*. Pour permettre la validation des données résultant de la réplication des tâches *Reduce*, les fichiers produits en sortie sont renommés : le suffixe qui distingue chaque réplicat est accolé au fichier correspondant.

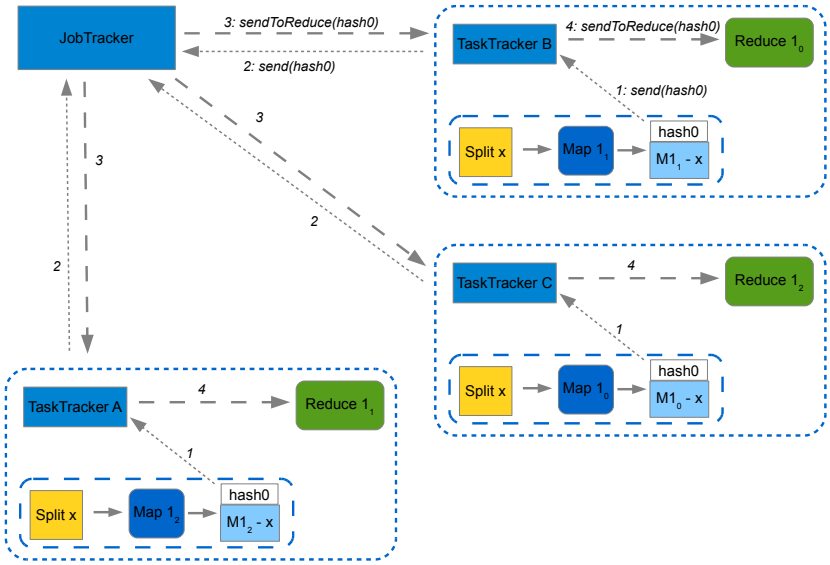


Figure 5. *Utilisation des digests pour valider un résultat de map répliqué*

La figure 5 illustre ce fonctionnement. Dans un premier temps, chacun des *TaskTrackers* sur lesquels s'exécute une réplique du *Map* 1 récupère le hash du résultat,

et le transmet au *JobTracker*. Lorsque ce dernier dispose de suffisamment de *digests* pour valider le résultat, c'est-à-dire $f + 1$ *digests* identiques, il renvoie le *digest* valide aux *TaskTrackers* pour que les répliques de *Reduce* puissent vérifier l'intégrité de la partition associée une fois qu'ils l'auront chargée.

5. Évaluation de performance de la plateforme tolérante aux fautes byzantines

5.1. *Environnement, paramètres et application*

Dans cette section, nous présentons une étude de performance de notre plate-forme afin de mesurer le coût du mécanisme de tolérance aux fautes byzantines par rapport à la plate-forme initiale. Ces expérimentations ont été réalisées sur une grappe de 6 nœuds appartenant à la plate-forme nationale de tests Grid'5000 [1]. Les machines ayant servi aux tests sont toutes équipées de processeurs Bi-Opteron et de 2Go de RAM. Une des machines est dédiée à l'exécution du JobTracker ; les cinq autres servent de Task Trackers.

Pour cette série d'expériences, nous avons utilisé l'application MonsterQuery qui appartient à l'ensemble de tests *Gridmix* (Apache, n.d.a). *Gridmix* est un ensemble de tests de performance qui a grandement participé à l'amélioration des performances de la plate-forme Hadoop. Il est composé de six applications dont la plupart génère un grand nombre d'entrées/sorties. *Gridmix*, qui partage cette caractéristique, est un filtre de texte itératif. Le rôle des *Maps* est de calculer une paire clé/valeur sur une partie du texte, la clé étant calculée sur le début du texte et le reste composant la valeur. Les *Reduces* convertissent alors les paires clé/valeur en blocs de données qui seront ensuite repris par une nouvelle itération. Le test *Gridmix* est constitué d'une série de trois de ces itérations. Cette succession d'itérations MapReduce qui permet de charger le système et l'importance accordée aux entrées/sorties (goulot d'étranglement pour nombre d'applications MapReduce) font de *Gridmix* un test particulièrement intéressant.

L'étude présentée à la section 3.1 et des expériences préliminaires ont montré qu'un réglage de 64MB pour la taille des splits permettait d'obtenir de très bonnes performances. Dans cette étude nous avons donc choisi de conserver cette valeur pour toutes les expériences. En contrepartie, nous allons agrandir la taille des données d'entrée lorsque le nombre de *Maps* augmente. En effet, comme chaque tâche *Map* traite un split, la taille des données d'entrées est égale au nombre de tâches *Map* multiplié par 64MB. En faisant varier le nombre de *Maps*- selon les valeurs suivantes : 10, 20, 40, 60, 80, 100, 140, 180, 220 - nous allons pouvoir étudier l'influence de la charge du système sur les performances de notre plate-forme.

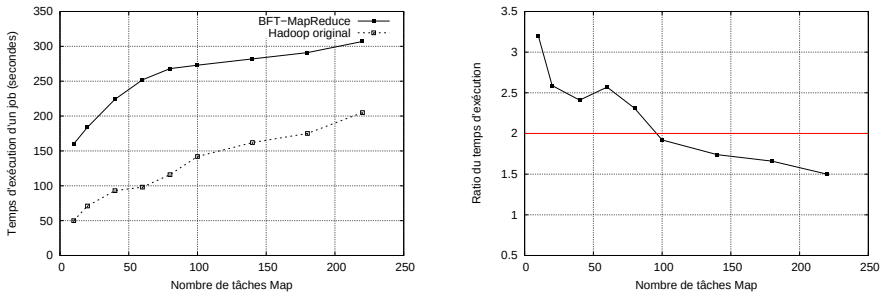
Les autres paramètres des expériences sont définis comme suit. Le nombre de tâches *Reduce* a été fixé à 5. Nous avons considéré un nombre de fautes byzantines

1. Grid'5000 est une plate-forme de test financée par le ministère de la recherche français, l'INRIA, le CNRS et les régions. Voir https ://www.grid5000.fr

maximum pour chaque tâche $f = 1$. Par conséquent, le nombre maximum de répliques d'une tâche ($n = 2f + 1$) équivaut à 3. Chaque expérience a été répétée 5 fois, et nous présenterons ici une moyenne des résultats.

5.2. *Étude de performance*

Le résultats des expériences sont présentés sur les figures 6(a) et 6(b). Sur chacune d'elles, on retrouve en abscisse le nombre de *Maps* composant le Job, respectivement : 10, 20, 40, 60, 80, 100, 140, 180, 220.



(a) Temps d'exécution d'un job

(b) Taux de ralentissement

Figure 6. *Comparaison de notre plate-forme avec la plate-forme Hadoop originale.*

La première figure (fig 6(a)) présente une comparaison du temps total d'exécution pour l'ensemble du Job sur la plate-forme Hadoop originale et sur notre plate-forme intégrant le mécanisme de tolérance à $f = 1$ faute byzantine. Une première analyse des résultats montre un comportement commun des deux plates-formes à savoir une augmentation du temps d'exécution lorsque le nombre de *Maps* augmente. En effet, dans cette expérience, la taille de l'input augmente avec le nombre de *Maps* (voir section 5.1). Sur les deux courbes, le temps d'exécution s'accroit toutefois moins vite que le nombre de *Maps* car une grande partie du benchmark MonsterQuery est constitué d'un grand nombre d'entrées/sorties qui peuvent être en partie effectuées en parallèle des calculs.

Si les formes des courbes de la figure 6(a) restent similaires, le surcoût dû au mécanisme de tolérance aux fautes byzantines varie lui avec le nombre de *Maps*. Ainsi la figure 6(b), qui présente le ralentissement du temps d'exécution d'un job sur notre plate-forme par rapport à la plate-forme Hadoop originale, montre que le surcoût diminue avec l'augmentation du nombre de *Maps*. Il est alors intéressant de comparer ce résultat à l'étude préliminaire de la réplication présenté à la section 3.1. Dans cette étude, nous avions conclu qu'en l'absence de faute, en répliquant 3 fois une tâche

($2f + 1$ répliques), seules les 2 répliques nécessaires ($f + 1$) sont réellement exécutées, engendrant ainsi un surcoût d'un rapport 2 (ligne horizontale de la figure 6(b)). En regardant la courbe, on s'aperçoit que ce rapport est en fait proche de 2,5 pour 20, 40, 60 et 80 *Maps*. En effet pour ces points la charge du système induite par les *Maps* n'est pas suffisante pour : d'une part empêcher certaines répliques inutiles de s'exécuter, d'autre part masquer les problèmes de synchronisation des tâches. On observe même qu'avec 10 *Maps* le rapport est supérieur à 3 (nombre de répliques totales). Cependant, le ratio de 2 (pour calculer au moins deux fois le résultat) est atteint à partir de 100 *Maps* et finit par n'être que 1,5 pour 220 *Maps*. Ce dernier résultat, très positif, s'explique par un effet de cache sur les données et par un meilleur recouvrement des entrées/sorties. En effet, notre approche constitue à répliquer chaque tâche et non l'ensemble du job. Les téléchargements effectués par chacune des répliques des *Reduces* peuvent donc se faire en parallèle. L'importance du phénomène est d'autant plus grande que les données sont volumineuses. Ceci explique que le surcoût ne soit que de 0,5 lorsque le nombre de *Maps* est important.

6. Travaux connexes

Il y a très peu de travaux sur la sécurité et le traitement des fautes Byzantines dans les architectures MapReduce (Wei *et al.*, 2009). Dans (Bessani *et al.*, 2010), nous avons proposé une version préliminaire de notre solution. À notre connaissance, seuls (Wei *et al.*, 2009) et (Moca *et al.*, 2011) traitent explicitement de la sécurité dans les architectures MapReduce. Les auteurs proposent un protocole distribué pour vérifier la validité des résultats produits par les tâches *Map* et *Reduce*. Le principe est proche de notre solution : de manière similaire aux digests, un "commitment" est envoyé au maître qui le valide et le retransmet aux exclaves. Ces derniers peuvent alors réclamer les données et vérifier leur contenu en regénérant les commitments. En revanche, les auteurs abordent peu la réplication et n'adressent pas les problèmes liés au nombre de copies. Dans (Moca *et al.*, 2011), les auteurs proposent un mécanisme distribué pour valider les résultats produits pour un MapReduce dans le cadre de DesktopGrid. Chaque tâche (map et reduce) est répliquée en r exemplaires. Pour éviter de surcharger le maître, les résultats intermédiaires des *Maps* sont directement envoyés aux *Reduces*. Chaque tâche *Reduce* reçoit donc r copies. La partition est supposée valide si $r/2 + 1$ copies sont identiques. En revanche, les répliques de résultats produits par les tâches *Reduce* sont envoyés directement au master qui effectue la validation finale. Un mécanisme de digest permet également de vérifier que les données produites par un *Map* correspondent bien à un *split* en entrée. Même, si ce protocole évite de surcharger les masters, la quantité de données échangées entre les *Maps* et *Reduces* est directement proportionnelle au degré de réplication, ce qui peut entraîner une charge de communication importante si les données produites sont volumineuses.

D'autres travaux se rapprochent également de notre problématique. (Singh *et al.*, 2009; van Renesse *et al.*, 2008) montrent qu'il est nécessaire de fournir des services distribués s'exécutant de manière continue malgré toutes sortes de fautes, y compris

celles byzantines. D'autre part, ils affirment que les protocoles classiques BFT (Castro *et al.*, 2002; Kotla *et al.*, 2009; Cowling *et al.*, 2006) basés sur une cohérence forte et où le système de réplication apparaît aux clients comme un seul serveur séquentiel correct sont très coûteux dans les grands systèmes en termes de disponibilité et de performance. Pour surmonter ces contraintes, certains auteurs ont proposé d'assouplir la cohérence des protocoles BFT. Par exemple, dans (Singh *et al.*, 2009), les auteurs proposent le protocole Zeno qui assure uniquement à terme une tolérance aux fautes byzantines cohérente. BFT2F (Li *et al.*, 2007) quant à lui prévoit, lorsqu'il y a entre f et $2f$ pannes, une cohérence faible appelée fork* qui limite le type de violations pouvant être fait par les serveurs malveillants. Une discussion intéressante à propos de l'importance de la tolérance aux pannes byzantines dans les *datacenters* peut être trouvée dans (van Renesse *et al.*, 2008).

Bhatotia et al. (Bhatotia *et al.*, 2010) affirment que le modèle des fautes franches et byzantines n'est pas adapté aux datacenters. Ils présentent ainsi une nouvelle approche pour traiter des fautes non-franches et non-byzantines. Plus précisément, ils proposent un mécanisme de détection de panne pour Pig, un système de traitement de données volumineuses (Olston *et al.*, 2008). Pig offre un langage de haut niveau, Pig Latin, qui permet de faire des requêtes semblables à SQL. Le programme réalise ensuite une série d'étapes de compilation qui produisent un MapReduce pouvant être exécuté dans un cluster Hadoop. L'idée de base est que Pig Latin possède un nombre limité de constructions ayant une sémantique basée sur l'algèbre relationnelle. Des vérifications sur la sémantique des constructions permettent alors de détecter les fautes à l'exécution des programmes Pig.

Dans (Fernandez *et al.*, 2005; Fernandez *et al.*, 2006), les auteurs considèrent un système distribué composé d'un processeur maître fiable et d'un ensemble de n processeurs esclaves exécutant les tâches. Les esclaves peuvent être byzantins. Chaque tâche renvoie une valeur binaire. L'objectif du maître est d'accepter avec une forte probabilité les valeurs correctes tout en minimisant la quantité de travail (le nombre de processeurs exécutant la tâche). Le nombre d'esclaves fautifs est borné soit par une valeur fixe $f < n/2$, soit en considérant une probabilité $p < 1/2$ de défaillance des processeurs. Les auteurs démontrent alors qu'il est possible d'obtenir avec une grande probabilité un niveau correct de succès avec une faible surcharge.

Enfin, (Clement *et al.*, 2009) propose la bibliothèque UpRight dont l'objectif est, par réplication, d'aider à rendre le système tolérant aux fautes byzantines. Les auteurs ont utilisé UpRight pour rendre HDFS tolérant aux byzantins.

7. Conclusion

Réalisés dans le cadre du projet *FTH-Grid* (international EGIDE/PESSOA, n.d.), les travaux présentés dans cet article ont permis de proposer une nouvelle architecture pour le *framework MapReduce* qui tolère jusqu'à f fautes byzantines par tâche. Basée sur un mécanisme de réplication des tâches et sur l'envoi de sommes de contrôle

(*digests*), notre approche tend à minimiser le surcoût en messages et en temps d'exécution. Certains choix architecturaux s'appuient sur des résultats expérimentaux en environnement réel. Nous avons ainsi montré qu'il était possible, en modifiant l'ordonnanceur, de lancer provisoirement $2f + 1$ répliques avec un faible surcoût additionnel. De plus, nous avons étudié l'utilisation d'un HDFS tolérant aux fautes byzantines, notamment pour la fiabilisation des données intermédiaires. L'implémentation d'un prototype résultant de ces travaux a permis de montrer qu'il était possible de minimiser le coût de la tolérance aux fautes byzantines. Elle a, entre autres, montré que des effets de recouvrement de tâches permettaient d'obtenir les $f+1$ résultats nécessaires, en un temps inférieur à $f + 1$ fois le temps calcul du job original.

8. Bibliographie

Apache, « Gridmix », , , n.d.a. `http://hadoop.apache.org/mapreduce/docs/current/gridmix.html`.

Apache, « Hadoop », , , n.d.b. `http://hadoop.apache.org/mapreduce/`.

Armbrust M., Fox A., Griffith R., Joseph A. D., Katz R. H., Konwinski A., Lee G., Patterson D. A., Rabkin A., Stoica I., Zaharia M., Above the Clouds : A Berkeley View of Cloud Computing, Technical Report n° UCB/EECS-2009-28, EECS Department, University of California, Berkeley, Feb, 2009.

Barroso L. A., Hölzle U., « The Datacenter as a Computer : An Introduction to the Design of Warehouse-Scale Machines », *Synthesis Lectures on Computer Architecture*, vol. 4, n° 1, p. 1-108, 2009.

Bessani A. N., Cogo V. V., Correia M., Costa P., Pasin M., Silva F., Arantes L., Marin O., Sens P., Sopena J., « Making Hadoop MapReduce Byzantine Fault-Tolerant. Fast abstract », *40th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2010)*, June, 2010.

Bhatotia P., Wieder A., Rodrigues R., Junqueira F., Reed B., « Reliable Data-Center Scale Computations », *LADIS*, 2010.

Castro M., Liskov B., « Practical byzantine fault tolerance and proactive recovery », *ACM Trans. Comput. Syst.*, vol. 20, n° 4, p. 398-461, 2002.

Clement A., Kapritsos M., Lee S., Wang Y., Alvisi L., Dahlin M., Riche T., « Upright cluster services », *SOSP*, p. 277-290, 2009.

Cowling J. A., Myers D. S., Liskov B., Rodrigues R., Shrira L., « HQ Replication : A Hybrid Quorum Protocol for Byzantine Fault Tolerance », *OSDI*, p. 177-190, 2006.

Dean J., Ghemawat S., « MapReduce : Simplified Data Processing on Large Clusters », *OSDI*, p. 137-150, 2004.

Fernandez A., Georgiou C., López L., Santos A., « Reliably Executing Tasks in the Presence of Malicious Processors », *DISC*, p. 490-492, 2005.

Fernandez A., Lopez L., Santos A., Georgiou C., « Reliably Executing Tasks in the Presence of Untrusted Entities », *Reliable Distributed Systems, IEEE Symposium on*, vol. 0, p. 39-50, 2006.

international EGIDE/PESSOA P., « FTH-Grid », , , n.d. `http://fth-grid.di.fc.ul.pt/index.php?title=Public:Main_Page`.

Kotla R., Alvisi L., Dahlin M., Clement A., Wong E. L., « Zyzzyva : Speculative Byzantine fault tolerance », *ACM Trans. Comput. Syst.*, 2009.

Li J., Mazières D., « Beyond One-Third Faulty Replicas in Byzantine Fault Tolerant Systems », *NSDI*, 2007.

Moca M., Silaghi G. C., Fedak G., « Distributed Results Checking for MapReduce in Volunteer Computing », *Fifth Workshop on Desktop Grids and Volunteer Computing Systems (PCGrid 2011) held in conjunction with the IEEE International Parallel and Distributed Processing Symposium*, 2011.

Olston C., Reed B., Srivastava U., Kumar R., Tomkins A., « Pig latin : a not-so-foreign language for data processing », *SIGMOD Conference*, p. 1099-1110, 2008.

Rodrigues R., Kouznetsov P., Bhattacharjee B., « Large-scale byzantine fault tolerance : safe but not always live », *Proceedings of the 3rd workshop on on Hot Topics in System Dependability*, USENIX Association, Berkeley, CA, USA, 2007.

Singh A., Fonseca P., Kuznetsov P., Rodrigues R., Maniatis P., « Zeno : Eventually Consistent Byzantine-Fault Tolerance », *NSDI*, p. 169-184, 2009.

Valipour M., Amirzafari B., Maleki K., Daneshpour N., « A brief survey of software architecture concepts and service oriented architecture », *Computer Science and Information Technology, 2009. ICCSIT 2009. 2nd IEEE International Conference on*, p. 34 -38, Aug, 2009.

van Renesse R., Rodrigues R., Spreitzer M., Stewart C., Terry D., Travostino F., « Challenges facing tomorrow's datacenter : summary of the LADiS workshop », *Proceedings of the 2nd Workshop on Large-Scale Distributed Systems and Middleware*, p. 1-7, 2008.

Wei W., Du J., Yu T., Gu X., « SecureMR : A Service Integrity Assurance Framework for MapReduce », *Twenty-Fifth Annual Computer Security Applications Conference, (ACSAC 2009), Honolulu, Hawaii*, p. 73-82, 2009.

White T., *Hadoop : The Definitive Guide*, first edition edn, Oreilly, 2009.