Chapitre 5

Technologie de l'écosystème d'Hadoop

La programmation distribuée, sous la forme de processus répartis sur un ensemble (un cluster, un cloud, une grille) de machines, est la seule solution qui permet de traiter en temps raisonnable de gros problèmes et de gros volumes de données (c'est-à-dire de *passer à l'échelle*, voir section 3.6). Selon les besoins en échanges de données, et selon les capacités relatives de traitement et de transfert de données, une application distribuée peut être simple ou très complexe à développer.

Pour débarrasser le *data scientist* d'une partie de ces préoccupations, des intergiciels distribués (ou *middleware*) spécialisés dans le stockage et l'analyse de données ont émergé, comme *Hadoop*. Ce chapitre présente les principale caractéristiques de l'architecture logicielle d'*Hadoop*, destinée finalement à faciliter le stockage distribué des gros volumes de données et à supporter une chaîne de traitements de type *Map-Reduce*.

5.1 Approches de la localité des calculs et des données

5.1.1 Démarche de localité en analyse de données

Lorsque les données sont très volumineuses il peut être plus rapide de router le code d'analyse jusqu'aux données et de le faire tourner sur les processeurs des nœuds de stockage, plutôt que de ramener les données vers des baies de calcul. En fait, tout dépend de la taille des données, de la complexité des calculs et de la vitesse du réseau.

La première étape d'une chaîne d'analyse de données peut comporter la lecture et le filtrage de très gros volumes de données pour n'en retenir qu'une fraction des données. Habituellement ces opérations de filtrage des données initiales ne sont pas très gourmandes en puissance de calcul, mais réalisent beaucoup d'entrées-sorties. Si l'on utilise des disques et un interconnect standards, la bande passante des disques et du réseau seront probablement les maillons lents de la chaîne. Dès lors il est logique de router les codes de lecture et de filtrage vers les nœuds de données et de les transformer momentanément en nœuds de calcul pour filtrer les données initiales, plutôt que de déplacer les données brutes à travers le réseau.

Une fois les données filtrées, on cherche à les regrouper selon une ou plusieurs caractéristiques, puis à appliquer de nouveaux traitements sur ces données filtrées-regroupées pour en extraire des caractéristiques de groupe. On peut ainsi reproduire des requêtes de type *Select-From-Where-Groupby*, et conserver les résultats sur les nœuds de données du même système de stockage. Le paradigme *Map-Reduce* est très adapté à cette démarche (voir section 8.1.3).

5.1.2 Démarche de localité en calcul intensif

Le calcul à haute performance a pour objectif de tirer le maximum de performance du matériel employé. L'objectif est que les coœurs de calcul soient ainsi tout le temps en train de calculer, qu'il n'y ait plus de temps mort à attendre des données. On se rapproche de cet objectif en traquant toutes les sources de perte de performance, notamment en luttant contre la latence et la bande passante limitée des communications des réseaux d'interconnexion, mais aussi en luttant contre les *défauts de cache*, et contre la latence et la bande passante limitée des mémoires globales !

Cette démarche *HPC* n'est pas celle mise en avant dans le *Big Data*, pourtant elle est tout à fait pertinente dans le dernier étage d'une architecture mixte *d'analyse de données & de calcul intensif* (voir section 1.3). Elle est indispensable pour réaliser rapidement certains traitements itératifs de *Machine Learning*.

Un peu de détail : *La limite inférieure du temps de latence est imposée par la physique et la vitesse de la lumière. Dans le cas où l'on devrait traverser un réseau longue distance pour échanger des données, emprunter Internet par exemple, les conséquences de la latence pourraient être catastrophiques. Par exemple, deux machines installées à Marseille et à Lille sont distantes d'environ 1000km, ce qui impose une latence minimale de l'ordre de 3,3ms (= 1000km / 300000km/s). Temps pendant lequel une machine d'aujourd'hui peut réaliser beaucoup de calculs, et donc gâcher beaucoup de ressources si elle est simplement en attente d'une donnée en cours de transfert. On évite donc de calculer intensivement à travers un réseau longue distance.*

5.2 Vue d'ensemble d'*Hadoop*

5.2.1 Caractéristiques essentielles d'Hadoop

Hadoop est une suite/une architecture/une plate-forme logicielle de stockage et d'analyse de données, dont on peut lister les propriétés suivantes :

1. *La plate-forme possède un système de fichiers distribué très facilement extensible*. Hadoop gère seul la distribution et le stockage des données sur ses différents nœuds, et pour augmenter la capacité de stockage il suffit d'ajouter des nœuds de données dans la plate-forme.

2. *Les codes des traitements sont routés jusqu'aux données*. Cette stratégie est la plus efficace pour de grosses volumétries de données stockées sur des machines standard reliées par des réseaux standard. Les nœuds de données se transforment donc en nœuds de calculs le temps des traitements, et par conséquent, augmenter le nombre de nœuds de données pour accroître la capacité de stockage augmente aussi la capacité de traitement.

3. *Des mécanismes de tolérance aux pannes sont intégrés à la plate-forme*. Hadoop étant conçu pour fonctionner sur du matériel standard (bon marché), des pannes fréquentes sont supposées inéluctables, et les données sont répliquées sur plusieurs nœuds afin d'être toujours accessibles. Quand un réplicat disparaît (suite à une panne), ses copies sont à nouveau répliquées pour maintenir un bon taux de réplication. De même, les tâches de traitements exécutées sur les nœuds de données sont monitorées et relancées sur le nœud d'un autre réplicat si une panne survient. L'utilisateur n'a pas à se soucier de la tolérance aux pannes.

4. *Un paradigme de programmation Map-Reduce est intégré à la plate-forme*. Ce paradigme convient à la récupération et au filtrage de données stockées dans l'ensemble des nœuds

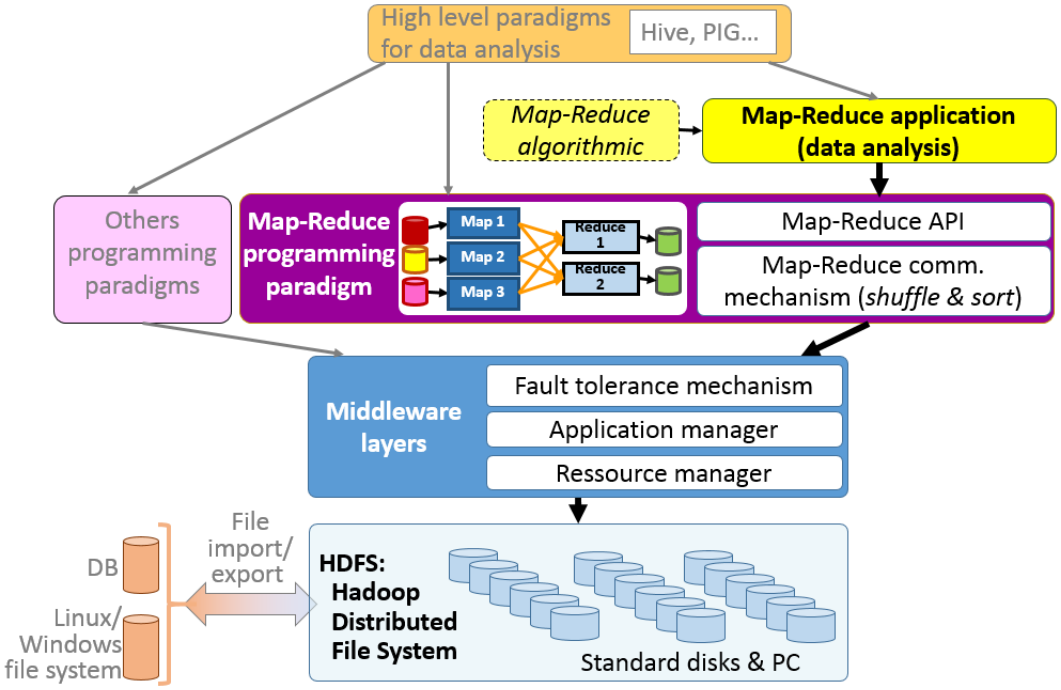

FIGURE 5.1 – Pile logicielle simplifiée de l'écosystème *Hadoop*

de données, ainsi qu'à la réalisation de quelques traitements sur les données retenues. Son intégration dans la pile logicielle d'Hadoop rend son utilisation très simple.

Il est possible d'enchaîner des appels *Map-Reduce* pour réaliser des opérations plus complexes, il ne faut toutefois pas chercher à implanter n'importe quel algorithm en *Map-Reduce*. Ce paradigme de programmation n'est pas adapté pour les calculs de *data analytics* complexes, et surtout itératifs. Mieux vaut alors poursuivre avec un autre paradigme et sur une autre plate-forme, comme introduit à la section 1.3.

5.2.2 Principes et pile logicielle d'Hadoop

La figure 5.1 montre la pile logicielle simplifiée d'Hadoop et d'une partie de son écosystème. A la base se trouve un système de fichiers distribué (voir section 5.3) appelé HDFS (*Hadoop Distributed File System*), sur lequel s'appuie un ensemble de composants permettant d'identifier puis de choisir les ressources sur lesquelles déployer les processus d'une application, de gérer des applications distribuées, et de réagir en cas défaillance pour être tolérant aux pannes. Cet ensemble de composants et de fonctionnalités forment un couche de *middleware* quasi-invisible à l'utilisateur et qui a beaucoup évolué entre les versions 1 et 2 d'*Hadoop* (la version 2 est aussi appelée *YARN : Yet Another Resource Negociator*). Le monitoring des ressources a été renforcé, et la supervision des applications *Map-Reduce* en cours d'exécution a cessé d'être un goulot d'étranglement qui limitait le passage à l'échelle (voir section 5.5).

Au dessus, une API permet une implantation aisée d'applications dans le paradigme *Map-Reduce*. L'utilisateur pouvant se contenter de développer la classe Java d'une tâche *Map* et celle d'une tâche *Reduce* (ou même une seule des deux classes, celle manquante devenant automatiquement une fonction identité). Le mécanisme de redistribution des sorties des tâches *Map* vers les entrées des tâches *Reduce*, qui est un composant clé du *Map-Reduce* d'*Hadoop*, est également fourni par la pile logicielle et reste une boite noire pour l'utilisateur. Toutefois ce dernier

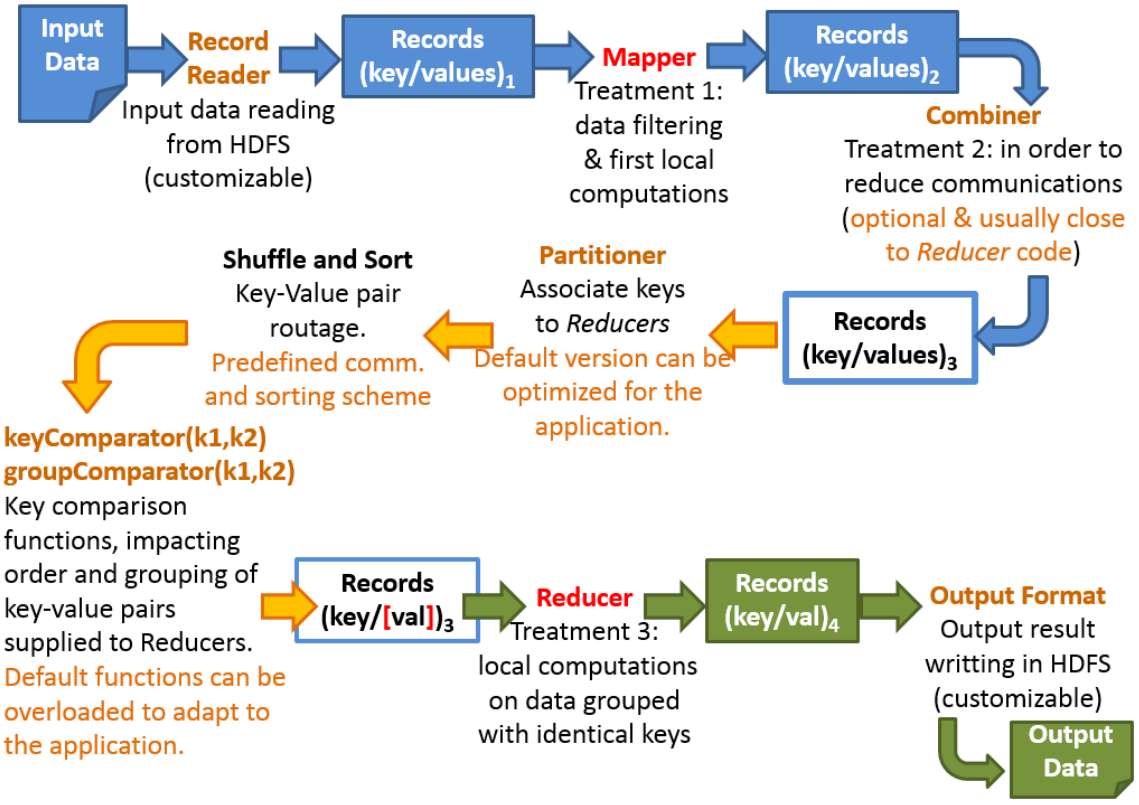

FIGURE 5.2 – Détails de toutes les étapes d'un *Map-Reduce* d'*Hadoop*

peut influencer la redistribution des données, par un réglage en amont et deux réglages en aval du mécanisme de communication, et en tirer partie dans ses applications *Map-Reduce*. Il est ainsi possible de développer facilement des applications d'analyse de données basées sur une algorithmique *Map-Reduce* (voir la partie haute de la figure 5.1), avec peu de connaissances d'informatique distribuée.

Toutefois, *Hadoop* peut supporter d'autres types de paradigmes de programmation distribuée, dont les tâches seront déployées en se basant toujours sur le gestionnaire de ressources et le gestionnaire d'application du milieu de pile. Enfin, *l'écosystème* d'Hadoop est très riche, et on compte notamment des applications de plus haut niveau permettant par exemple de traiter des données dans un formalisme proche de SQL (comme dans une base de données relationnelle), et des outils permettant d'importer des données extérieures dans le système de fichiers distribué d'*Hadoop* ou d'exporter des données d'*Hadoop* vers l'extérieur (en bas à gauche de la figure 5.1).

5.2.3 Etapes complètes d'une chaîne *Map-Reduce* d'*Hadoop*

La figure 5.2 illustre la chaîne complète de *Map-Reduce* d'*Hadoop* avec toutes ces fonctionnalités. *Hadoop* permet de définir, ou d'affiner, plus d'étapes que les simples fonctions *map* et *reduce*.

– On retrouve bien sur les classes *Mapper* et *Reducer* a définir par l'utilisateur (en rouge sur la figure 5.2). En l'absence de définition explicite elles seront remplacées par la fonction identité. Donc un *Map-Reduce* vide reproduira en sortie les données d'entrée, simplement regroupées par clés identiques.

– En entrée de la chaîne se trouve une classe de lecture des données, appelée *Record Reader*).

Cette classe lit le fichier HDFS d'entrée et le convertit en paires clé-valeur que traitera la fonction *map*. Par défaut elle lit ligne par ligne un flux d'entrée sensé être un fichier texte, et crée une paire dont la clé est l'offset (en octets) de cette ligne dans le fichier et dont la valeur est la ligne elle-même. L'ensemble du *Record Reader* est redéfinissable par l'utilisateur, ainsi que le type des données d'entrée (qui pourraient ne pas être du texte ASCII).

- La classe *Combiner* est un *Reducer* local et optionnel. Cette classe applique un traitement aux paires de sortie de la fonction *map* pour les combiner. En général elle fait la même chose ou presque que la classe *Reducer* sur un ensemble de paires de même clé, et permet de réduire le volume de données routées ensuite vers les *Reducer*. Sa définition n'est pas obligatoire, et si cette classe est définie, alors *Hadoop* décide librement à quels moments l'appeler pour combiner les sorties de chaque *Mapper*.
  L'utilisation d'un *Combiner* permet d'améliorer les performances en réduisant le traffic lors du *Shuffle & Sort*, mais entraîne quelques contraintes algorithmiques.

- Le *Partitioner* est une classe optionelle qui permet de spécialiser la répartition des clés sur les différents *Reducer*. Toutes les paires de même clé sont envoyées à un même *Reducer*, mais un *Reducer* peut gérer plusieurs clés. Par défaut, *Hadoop* effectue une répartition aléatoire grossière des clés sur les *Reducer*. Il est possible de définir une politique de répartition de charge plus équilibrée si on a connaissance des clés possibles et de la distribution des volumes de valeurs associés.

- L'étape de *shuffle & sort* n'est pas redéfinissable. Le schéma de communication est imposé et ressemble à un *all-to-all* orienté des *Mappers* vers les *Reducers* (pour emprunter le vocabulaire du HPC). Cependant, la stratégie de redistribution des clés vers les *Reducers* peut être redéfinie par le *Partitioner* présenté précédemment, ainsi que la stratégie de tri des clés et de regroupement des valeurs en entrée de chaque *Reducer* (voir ci-après).

- Deux fonctions de comparaison de clés à l'entrée des *Reducers* (*keyComparator* et *groupComparator*) permettent de contrôler l'ordre dans lequel les paires *clé - listes de valeurs* seront constituées et présentées aux *Reducers* (ce qui permet d'optimiser fortement certains algorithmes).

- Enfin, le format de sortie peut-être modifié dans l'interface *OutputFormat*, qui définit quels types de clés et valeurs de sortie sont attendues et comment les écrire sur disque. Par défaut, n'importe quel types de données *Writable* peut être écrit dans un fichier texte, et les paires de sortie des *Reducer* sont bien celles écrites sur disques.

La chaîne *Map-Reduce* d'*Hadoop* est donc optimisable ou spécialisable à plusieurs niveaux, mais fonctionne avec des comportements par défaut à toutes les étapes.

5.2.4 Déploiement d'un *Map-Reduce* en *Hadoop*

Une chaîne de traitement *Map-Reduce* se déploie sous la forme d'un ensemble de tâches sur un ensemble de nœuds de données transformés momentanément en nœuds de calculs. Dans le cas d'une implantation en Java, les tâches sont souvent des JVM. La figure 5.3 montre un exemple de déploiement d'une chaîne de tâches *Mapper* et *Reducer* sur 5 nœuds de données d'un système HDFS.

En HDFS chaque fichier est stocké sous forme de blocs répartis sur différents nœuds de don-

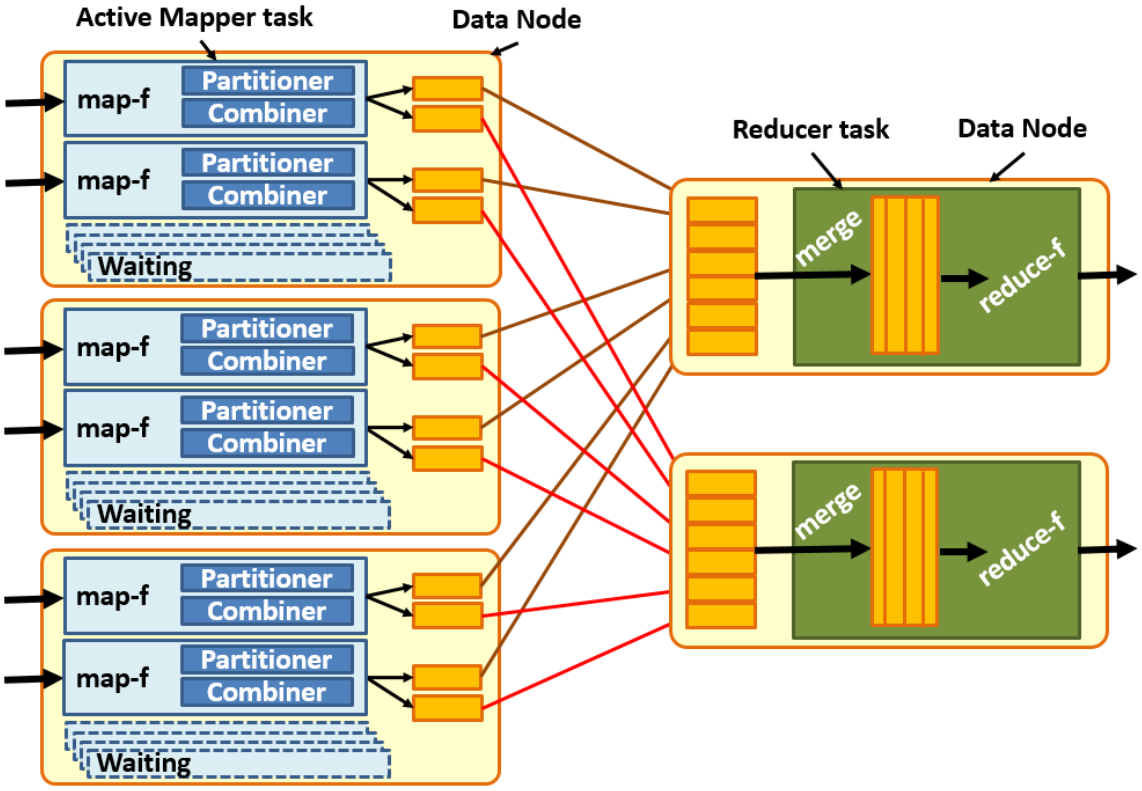

FIGURE 5.3 – Exemple de déploiement d'un *Map-Reduce* d'*Hadoop* sur 5 nœuds, avec 2 tâches *Mapper* actives au maximum en même temps sur chaque nœud, et 2 tâches *Reducer*

nés, et *Hadoop* lance une tâche *Mapper* par bloc lors d'un *Map-Reduce*. Le nombre de tâches *Mapper* exécutées au total dépend donc de la taille des fichiers lus et de la taille des blocs d'HDFS (taille configurable). En revanche, le nombre maximum de tâches *Mapper* actives simultanément sur un même nœud peut être spécifié, et doit dépendre de la puissance du nœuds en capacité de traitement, en mémoire et en bande passante disque. Par exemple : un fichier de 2Go stocké en HDFS avec des blocs de 64Mo sera décomposé en 32 blocs, et mènera donc à exécuter 32 tâches *Mapper*. Or plusieurs de ces blocs de données, et donc de ces *Mappers*, pourraient être localisés sur la même machine. Si on imagine avoir des machines avec seulement 2 cœurs on spécifiera de n'exécuter simultanément que 2 tâches *Mapper* par nœud, et Hadoop ordonnancera ses *Mapper* en conséquence. La figure 5.3 illustre un cas où 3 nœuds de données sont concernés, avec une limite maximale de 2 *Mapper* simultanés par nœud. Une fois activé, chaque *Mapper* exécute sa fonction *map*, fait appel aux classes *Combiner* et *Partitioner*, et stocke finalement ses paires clé-valeur de sortie dans des buffers organisés par clé.

Le nombre de tâches *Reducer* peut en revanche être clairement imposé à *Hadoop*. Attention toutefois à ne pas en imposer plus qu'il n'y aura de clés de sorties différentes, ni à en imposer trop peu. Déployer un seul *Reducer* signifie qu'une seule machine recevra toutes les paires clé-valeurs de sortie des *Mapper*. Cela risque de saturer cette machine, et de ralentir les traitements en séquentialisant l'étape de *Reduce*. Comme illustré sur la figure 5.3, chaque tâche *Reduce* va successivement recevoir les données provenant des *Mapper*, les agréger en regroupant celles ayant la même clé, puis appliquer la fonction *reduce* sur chaque liste de valeurs de même clé. Enfin, elle stockera sur disque ses paires clé-valeur de sortie.

Les tâches *Mapper* et *Reducer* peuvent très bien être déployées sur les mêmes nœuds, qui sont tous des nœuds de données HDFS. Ceci est en fait décidé par *Hadoop*, et par son gestion-

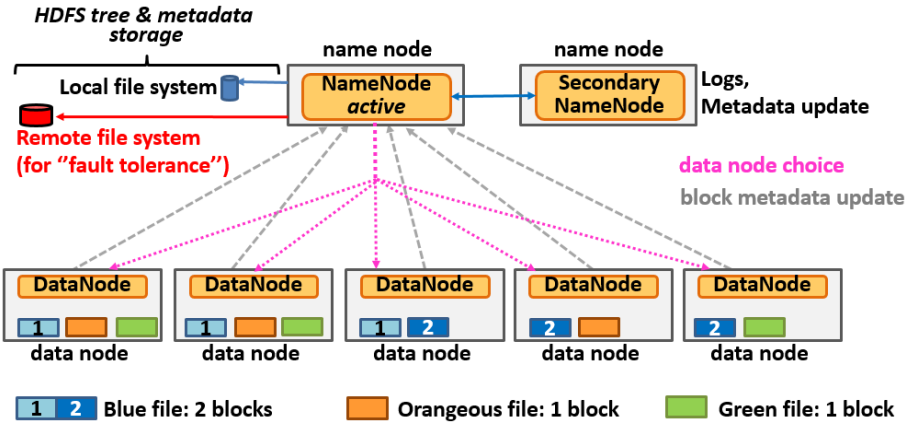

FIGURE 5.4 – Principes de stockage de données en HDFS

naire/allocateur de ressources. Dès lors se pose la question du recouvrement des activités des *Mapper* et des *Reducer*. En fait, les *Reducer* peuvent commencer tôt à recevoir puis à agréger les données par clés identiques, mais ils doivent attendre d'avoir reçu toutes les paires clé-valeurs des *Mapper* avant d'exécuter leur fonction *reduce*. Il n'y aura donc pas de recouvrement entre l'exécution des fonctions *map* et *reduce*, mais *Hadoop* permet de contrôler le pourcentage de complétion des *Mapper* à partir duquel les *Reducer* sont instanciés et commencent la réception puis l'agrégation de leurs données (mais pas leurs traitements). La valeur par défaut est faible (5%) et provoque le lancement des *Reducer* très tôt, cependant il peut parfois être efficace de retarder leur lancement. Par exemple, quand on utilise un *Combiner* et que peu de paires clé-valeur sont finalement envoyées aux *Reducer* : autant laisser les nœuds exécuter les *Mapper* sans être perturbés, puis router et réduire toutes les paires générées un peu plus tard.

5.3 Système de fichiers distribué HDFS d'*Hadoop*

5.3.1 Principes d'HDFS

La figure 5.4 illustre le découpage en blocs, la réplication et le stockage distribué des fichiers HDFS, afin d'atteindre à la fois de bonnes performances d'accès et une forte tolérance aux pannes. Un nœud appelé le *NameNode actif* établit et conserve une cartographie de la répartition de tous les fichiers stockés dans HDFS. De temps en temps cette cartographie est remise à jour. En attendant les insertions de nouveaux fichiers et suppressions d'anciens donnent lieu à des modifications de la cartographie qui sont stockées sous forme de *logs* à la fois dans le *NameNode actif* et dans le *NameNode secondaire*. Une cartographie à jour est donc obtenue en appliquant les évolutions décrites dans les *logs* aux meta-données du *NameNode actif*. Le *NameNode secondaire* est chargé de recalculer de temps en temps une cartographie à jour du système de fichiers distribué en appliquant tous les *logs*, et de mettre ensuite à jour celle du *NameNode actif* sans que ce-dernier n'ait été ralenti par les calculs effectués.

Chaque fichier est découpé en *blocs*, typiquement de $64Mo$ ou $128Mo$, et chaque bloc est répliqué n fois, habituellement 3 fois. Les réplicats d'un même bloc sont stockés sur des machines différentes, afin de toujours résister à la perte d'un nœud, ou même de deux ! En cas de disparition d'un nœud et de ses blocs, chaque bloc réplicat disparu est reconstitué sur un nouveau nœud à partir d'un de ses réplicats encore accessible. HDFS reconstitue ainsi rapidement un ensemble de n réplicats pour chaque bloc. Sur la figure 5.4 le fichier bleu est assez gros pour être stocké sous la forme de deux blocs, chacun répliqué en 3 exemplaires, alors que les fichiers orange et

vert sont chacun composés d'un seul bloc répliqué 3 fois. Un même nœud de données peut très bien stocker plusieurs blocs différents, provenant du même fichier ou de fichiers différents, mais il ne peut pas stocker plusieurs fois le même bloc. Dans un cluster *Hadoop* de grande taille, les réplicats d'un même bloc doivent être stockés sur des nœuds situés dans des racks différents (donc électriquement indépendants). Lors de la création d'un nouveau fichier le *NameNode actif* va répartir les réplicats de ses blocs sur les différents nœuds de données disponibles (flèches roses sur la figure 5.4), et chaque nœuds de données va maintenir le *NameNode actif* au courant de son état, et du succès ou de l'échec de ses créations de blocs (flèches grises sur la figure 5.4). Le *NameNode actif* maintiendra ainsi une connaissance à jour du système de fichier HDFS.

Remarque : La taille typique des blocs (64 ou $128Mo$) a été établie pour à la fois masquer le temps de positionnement en début de bloc sur le disque (temps de *seek* sur des disques rotatifs standards) et pouvoir paralléliser les accès à un même fichier. On estime que sur du matériel classique : $t_{seek} = 10ms$ et $Bw^{disk} = 100Mo/s$, et l'on souhaite :

$$t_{seek} < \frac{1}{100}.t_{transfert} \quad (5.1)$$

$$t_{seek} < \frac{Q}{100.Bw^{disk}} \quad (5.2)$$

$$100.t_{seek}.Bw^{disk} < Q \quad (5.3)$$

$$100.10ms.100Mo/s < Q \quad (5.4)$$

$$100Mo < Q \quad (5.5)$$

Il faut donc environ 100Mo de taille de bloc à lire pour que le temps de *seek* ne représente que 1% du temps de transfert des données, soit finalement des blocs de taille 64 ou $128Mo$. Prendre des blocs plus gros masquerait encore plus le temps de *seek*, mais fractionnerait moins les fichiers et donc empêcherait de bien paralléliser leur lecture depuis plusieurs nœuds à la fois.

5.3.2 Tolérance aux pannes et haute disponibilité d'HDFS

Les données stockées sur HDFS sont répliquées (comme vu ci-dessus), mais le *NameNode actif* constitue un *Single Point Of Failure* (SPOF) du système. Sans lui les données sont toujours stockées et répliquées sur les nœuds de données, mais inaccessibles faute de cartographie ! Une panne du *NameNode actif* pourrait donc rendre le système de fichiers HDFS totalement inexploitable.

Pour lutter contre cette faiblesse, deux mécanismes complémentaires de *tolérance aux pannes* puis de *haute disponibilité* ont été introduits :

- *Tolérance aux pannes* : Les meta-données du *NameNode actif* sont régulièrement sauvegardées sur un système de fichiers local (accès rapide) mais aussi sur un système distant (voir figure 5.4). En cas d'incident sur le *NameNode actif* on pourra ainsi reconstituer le système de fichiers d'*Hadoop* à partir d'une copie distante de ses meta-données et continuer à exploiter ses nœuds de données.

- *Haute disponibilité* : il est possible de dupliquer le *NameNode actif*, et de créer un *NameNode standby* qui reçoit et stocke en permanence les mêmes meta-données et *logs* que le *NameNode actif* (voir figure 5.5). Le *NameNode standby* est donc lui aussi éveillé mais n'agit pas sur les nœuds de données. En revanche, il est prêt à remplacer et à devenir le *NameNode actif* à tout moment et quasiment sans délai (voir figure 5.6), rendant la panne presque imperceptible. Evidemment, cette stratégie demande une machine supplémentaire.

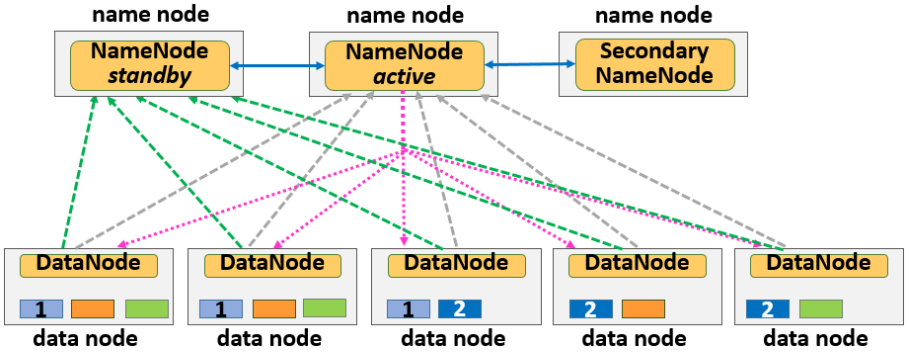

FIGURE 5.5 – Solution à haute disponibilité pour HDFS en cas de panne sur le *NameNode*

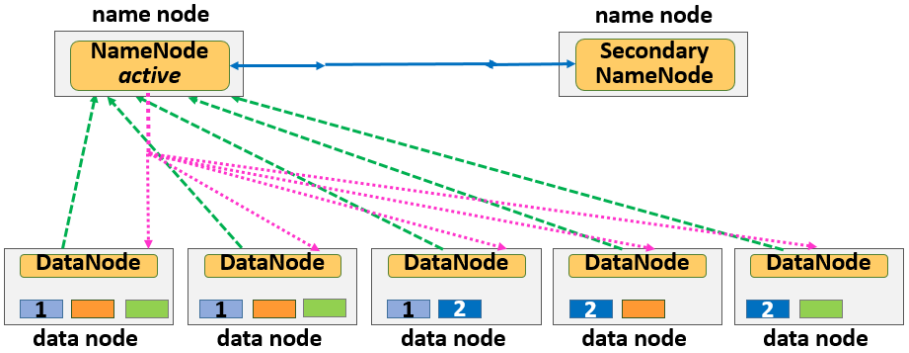

FIGURE 5.6 – Solution à haute disponibilité pour HDFS après apparition d'une panne sur le *NameNode*

5.3.3 Mécanismes de lecture d'HDFS

La lecture d'un fichier en HDFS est assez simple, et se résume sur la seule figure 5.7. Le code client commence par créer un objet local de la classe *DistributedFileSystem*, qui agira comme un *stub* ou *proxy* avec le système de fichier HDFS. Le code client s'adresse donc à cet objet local et demande à ouvrir un fichier (étape 1, commande *open*). Le *stub* s'adresse alors au *NameNode* d'HDFS pour connaître l'emplacement des réplicats de tous les blocs du fichier (étape $2a$). Ensuite le *stub* crée un autre objet local, de la classe *FSDataInputStream* (étape $2b$), qui agira comme un lecteur spécialisé dans la lecture du fichier visé, ayant connaissance des nœuds à contacter.

Le client fera alors des opérations *read* sur ce lecteur spécialisé (étape 3), qui ira interroger un des noeuds stockant le premier bloc du fichier (étape 4). Une fois le premier bloc lu, si les opérations de lecture continuent de la part du client, le lecteur spécialisé ira interroger un nœud contenant le second bloc (étape 5) et ainsi de suite.

Finalement, le client demandera à fermer le fichier ouvert en lecture auprès du lecteur spécialisé (étape 6, opération *close*). A noter que le lecteur spécialisé vérifie l'intégrité des données lues (calculs de *checksum*) et signale toute anomalie au *stub*, qui les retransmet au *NameNode*.

5.3.4 Mécanismes d'écriture d'HDFS

L'écriture d'un fichier en HDFS est plus complexe que sa lecture, et se trouve résumée sur les figures 5.8 et 5.9. Au début, le client crée un objet local *DistributedFileSystem* servant de *stub* ou *proxy* envers le système de fichier HDFS, comme c'était déjà le cas pour une opération de lecture (voir la section précédente). Dès lors, le client peut demander au *stub* de créer un nouveau fichier HDFS (étape 1, opération *create* sur la figure 5.8). Le *stub* s'adresse alors au *NameNode*

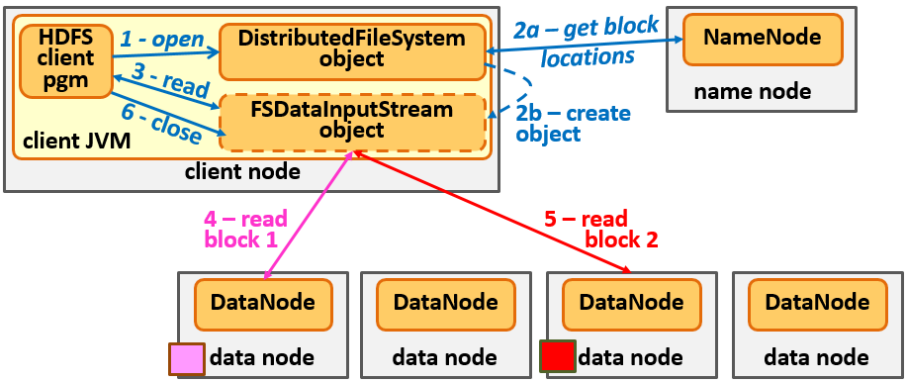

FIGURE 5.7 – Fonctionnement d'HDFS pour la lecture de données

d'HDFS pour obtenir le droit de créer un tel fichier (étape $2a$), et pouvoir créer en local un objet *FSDataOutputStream* (étape $2b$) qui jouera le rôle d'un écrivain spécialisé dans l'écriture du fichier visé. Par la suite, le client s'adresse localement à cet écrivain spécialisé pour lui demander d'écrire des données dans ce fichier (étape 3, opération *write*).

L'écrivain dialogue alors avec le *NameNode* d'HDFS pour savoir où créer un premier bloc et ses réplicats (étape $4a$), puis commence à écrire le premier réplicat du premier bloc sur un nœud de données (étape $4b$). Un mécanisme de *pipeline* se met alors en place : le nœud du premier réplicat retransmet ses données au nœud choisi pour héberger le deuxième réplicat (étape $4c$), qui les retransmet lui même au nœud choisi pour héberger le troisième réplicat (étape $4d$), et le processus se poursuit si plus de 3 réplicats par bloc sont spécifiés. Quand l'écriture sur le dernier réplicat est terminée, un message d'*acknowledge* remonte vers le nœud du réplicat précédent (étape $5a$), et ainsi de suite jusqu'à atteindre le nœud du premier réplicat (étapes $5b$). Ce nœud retourne alors un *acknowledge* à l'écrivain spécialisé (étape $5c$), et si tout s'est bien passé, l'écrivain enchaînera en traitant la demande suivante d'écriture de données. Quand le premier bloc est plein, l'écrivain demande à nouveau au *NameNode* d'HDFS un ensemble de nœuds où écrire un deuxième bloc et ses réplicats (étape $6a$), et le processus d'écriture pipelinée se reproduit (étapes $6b$ à $6d$, puis $7a$ à $7c$).

Une fois toutes les écritures de données terminées, le client demande à l'écrivain de refermer le nouveau fichier (étape 8, opération *close* sur la figure 5.9). Celui-ci en informe alors le *stub* (étape $9a$), qui en informe à son tour le *NameNode* d'HDFS (étape $9b$), qui met à jour ses metadonnées avec un nouveau fichier dans sa cartographie.

5.3.5 Accès séquentiel ou parallèle aux données stockées en HDFS ?

Les mécanismes de lecture et d'écriture de fichiers HDFS présentées dans les deux sections précédentes sont très séquentiels, menés par un programme client qui écrit ou lit des données dans le système HDFS les unes après les autres. Cette démarche permet par exemple d'importer ou d'exporter des données entre le système de fichier d'*Hadoop* et celui d'un simple PC, mais elle n'est pas adaptée pour exploiter intensivement des données déjà présentes dans des fichier HDFS.

Au contraire, une application *Map-Reduce* va déployer plusieurs processus sur différents nœuds de données et accéder en parallèle à divers blocs d'un même fichier HDFS (ou de plusieurs fichiers HDFS). L'architecture d'une application *Map-Reduce* est conçue pour tirer un maximum de performances d'un système de fichier distribué comme HDFS. Les sections 5.4 et 5.5 présentent le déploiement d'une telle application sur un système HDFS.

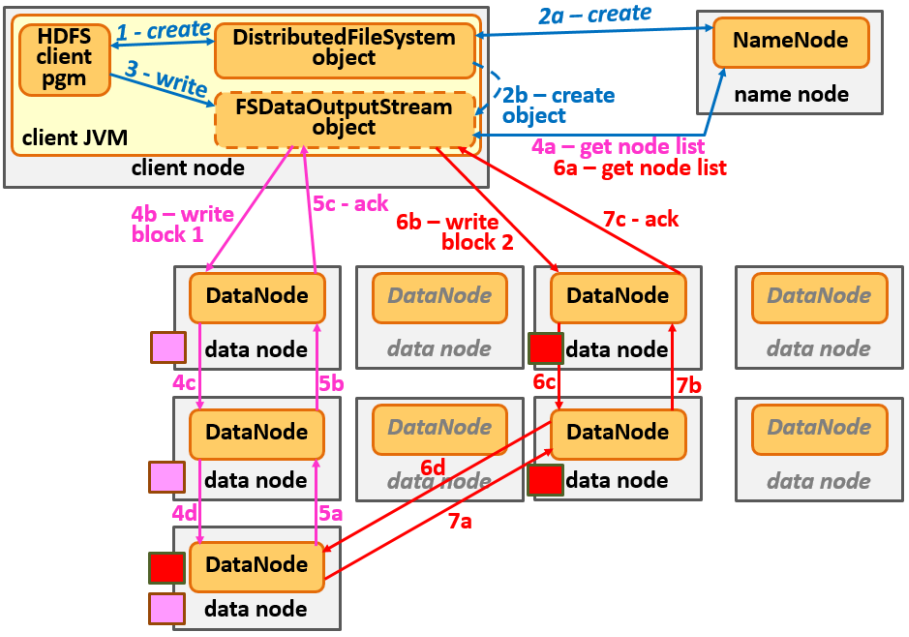

FIGURE 5.8 – Fonctionnement d'HDFS pour l'écriture de nouvelles données dans HDFS

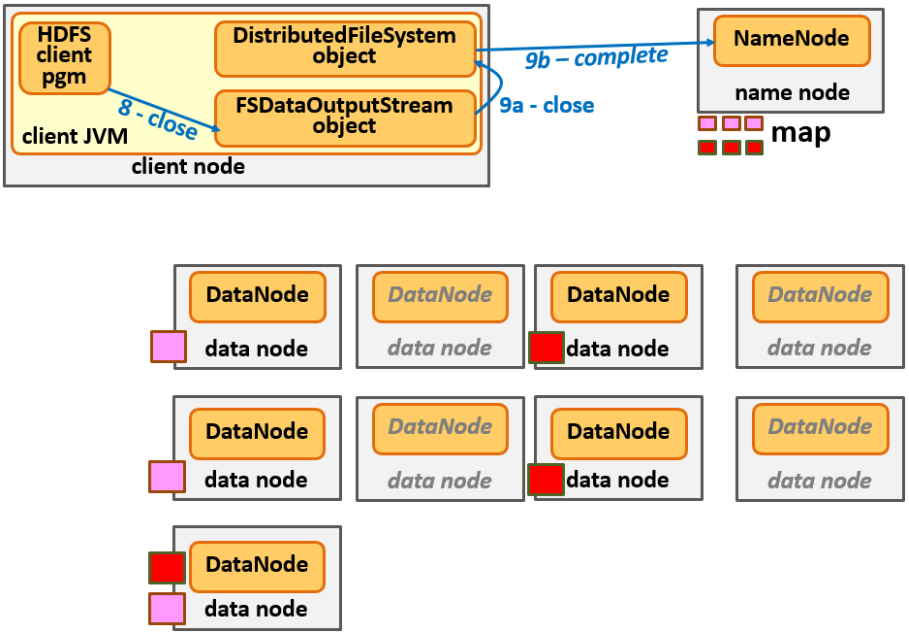

FIGURE 5.9 – Fonctionnement d'HDFS à la fin de l'écriture d'un nouveau fichier

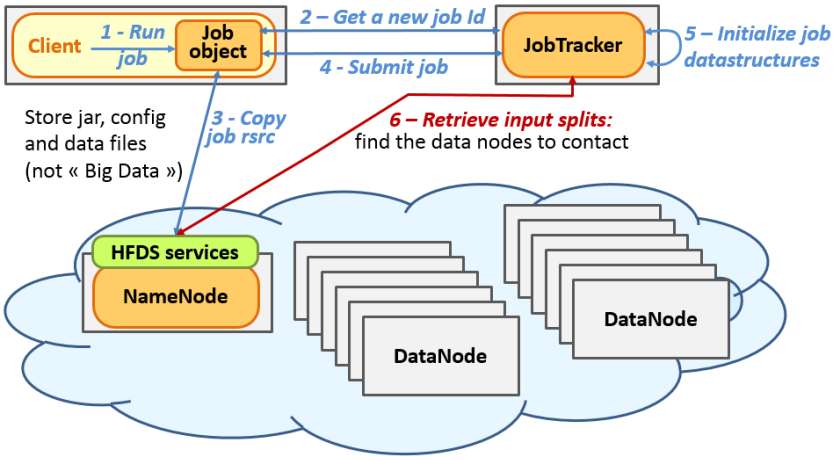

FIGURE 5.10 – Détail de la gestion d'un *Map-Reduce* sur *Hadoop* en version classique (*Map-Reduce 1*) - AVANT création des tâches *Mappers* et *Reducers*

5.4 Gestion de ressources d'Hadoop - version 1

Comme lors des lectures et écritures de fichiers HDFS, le programme client crée un objet local agissant comme un *stub* ou *proxy*. Cette fois-ci il s'agit d'un objet de la classe *Job*, à partir duquel le programme client configure, lance et attend la fin d'une opération complète de *Map-Reduce*. Le *stub* masque encore une fois la complexité réelle de l'interaction avec le système de fichiers HDFS, mais aussi avec le gestionnaire et allocateur de ressources qu'est le *JobTracker*. Ce dernier décide quels nœuds choisir (parmi ceux éligibles) pour héberger les processus *Mappers* et *Reducers*, et assure ensuite le monitoring et la gestion des processus lancés.

5.4.1 Mise en œuvre d'un *Map-Reduce* sur *Hadoop* en version 1

On peut décomposer la mise en œuvre d'un *Map-Reduce* en deux grandes étapes : avant la création des tâches *Mappers* et *Reducers* (figure 5.10), et à partir de leur création (figure 5.11).

La figure 5.10 montre l'enchaînement des opérations avant la création de tâches *Mappers* et *Reducers*. Comme précédemment l'application cliente crée et initialise un *stub* (l'objet *Job*) et lui demande d'exécuter une application *Map-Reduce* sur l'architecture distribuée *Hadoop* (étape 1). Le *stub* va alors s'adresser au service de gestion des *jobs* d'*Hadoop* : le *JobTracker*, et lui demander d'attribuer un identificateur à ce futur *job* (étape 2). Le *stub* va ensuite s'adresser au *NameNode* d'HDFS pour y stocker les diverses données de configuration du *job* (fichiers *jar* de codes Java, fichiers de configuration...) qui ne sont pas déjà dans HDFS (étape 3). Si l'accès à HDFS a réussi, le *stub* va recontacter le *JobTracker* pour lui demander cette fois-ci d'exécuter le *job* auquel il a précédemment accordé un identificateur (étape 4). La suite se passe alors sans le client ni le *stub*.

Le *JobTracker* initialise des structures de données de gestion du *job* (étape 5), puis interroge le *NameNode* d'HDFS pour connaître les nœuds de données sur lesquels lancer des tâches *Mappers* et *Reducers* (étape 6). A cet effet, la définition du *job* communiquée par le *stub* contient les identifiant des fichiers d'entrées utilisés, le *JobTracker* peut donc interroger le *NameNode* pour connaître la liste des nœuds de données hébergeant des réplicats des blocs de ces fichiers d'entrées (les "*splits*" de l'étape 6). A partir de cette liste, le *JobTracker* va pouvoir choisir les nœuds les plus adéquats et créer les tâches nécessaires au *Map-Reduce* sur ces nœuds.

La figure 5.11 illustre la création des tâches *Mappers* et *Reducers* et leur exécution. Pour

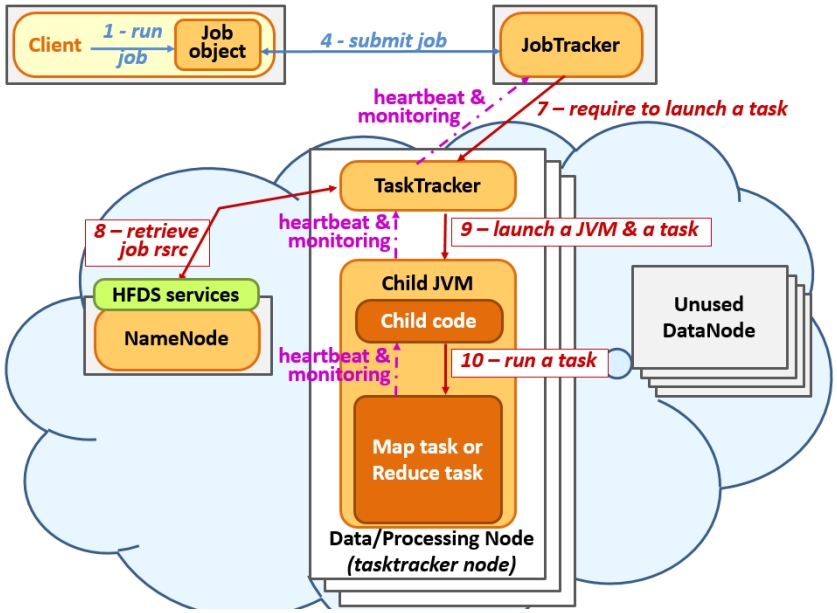

FIGURE 5.11 – Détail de la gestion d'un *Map-Reduce* sur *Hadoop* en version classique (*Map-Reduce 1*) - A PARTIR DE la création des tâches *Mappers* et *Reducers*

chaque nœud qu'il a identifié comme pertinent, le *JobTracker* contacte son service *TaskTracker* et lui demande de lancer une JVM pour héberger une tâche *Mapper* ou *Reducer* (étape 7). Sur chaque nœud concerné le *TaskTracker* s'adresse alors aux services d'HDFS pour récupérer les données de configuration du *Job* et les stocker dans le disque local (étape 8), puis démarre une JVM avec un code minimal (étape 9), qui va à son tour lancer une tâche *Mapper* ou *Reducer* (étape 10). Dès lors les nœuds de données utilisés sont devenus des *nœuds de données et de traitement*. Le nombre de *Reducers* est fixé par le client lors de la configuration du *job* ou estimé par *Hadoop* en fonction du volume de données d'entrée.

Une fois lancées, les tâches *Mappers* et *Reducer* s'auto-monitorent et envoient régulièrement (à intervalles de quelques secondes) des messages de progression et de *heartbeat* à leurs *TaskTracker*, qui envoient eux-mêmes de tels messages au *JobTracker* (voir le circuit violet sur la figure 5.11). Ainsi le *JobTracker* peut détecter une panne d'un des nœuds de données, décider de démarrer une tâche de remplacement sur un autre nœud, et fournir des informations sur la progression du *job Map-Reduce* au *stub* et au client.

Notons que de nombreux aspects du fonctionnement d'*Hadoop* sont finalement reconfigurables par l'utilisateur. Par exemple :

– Le client et son *stub* soumettent des demandes d'exécutions de *jobs* au *JobTracker*, qui les insère dans une queue et choisit lequel exécuter selon une politique d'ordonnancement des *jobs*. Cette politique peut être reconfigurée par l'utilisateur.

– Le lancement systématique d'une JVM pour encapsuler chaque tâche, à pour but de rendre le système global plus robuste en évitant que le *crash* d'une tâche ne se propage au reste de l'application. Mais ce fonctionnement aussi est reconfigurable par l'utilisateur, qui peut préciser le nombre maximal de réutilisations d'une même JVM (par défaut une JVM ne peut servir qu'à une seule tâche).

5.4.2 Limitation du *Map-Reduce* d'*Hadoop* version 1

Malgré ses aspects distribués et sa souplesse de configuration l'architecture logicielle de la première version d'*Hadoop* présente une faiblesse qui limite son passage à l'échelle : le *JobTracker* doit gérer tous les *jobs* en cours d'exécution, en plus de choisir les ressources des nouveaux *jobs* soumis. Le *JobTracker* maintient ainsi un lien avec chaque *TaskTracker* de chaque *job* en cours, et reçoit et traite notamment tous les messages de *heartbeat*.

Au final, le *JobTracker* constitue un goulot d'étranglement quand la taille du cluster et le nombre de *jobs* augmentent. Une seconde version d'*Hadoop* a donc été développée avec une couche plus évoluée de gestion des applications distribuées (voir section suivante).

5.5 Gestion de ressources d'Hadoop - version 2 (YARN)

5.5.1 Objectifs et principaux changements

Le premier environnement d'exécution d'applications *Map-Reduce* est apparu limité quand les clusters *Hadoop* ont commencé à grossir et à atteindre plusieurs milliers de nœuds. Il est alors devenu nécessaire de disposer d'un nouvel environnement qui permette :

- d'éviter un goulot d'étranglement au niveau du *JobTracker* qui gère toutes les tâches en cours d'exécution et centralise leurs informations de monitoring, afin de renforcer la capacité de passage à l'échelle,

- de supporter l'exécution d'autres types d'applications distribuées qu'uniquement des *Map-Reduce*, donc de pouvoir déployer des gestionnaires d'applications variés.

En 2010 *Yahoo !* a démarré le développement d'une nouvelle couche d'exécution et de contrôle des applications *Hadoop* au dessus d'HDFS. Cette version fut appelé *YARN* : *Yet Another Resource Negotiator*. Elle distingue clairement :

- les fonctionnalités de gestion et d'allocation de ressources de calculs, c'est-à-dire l'identification des nœuds de données qui hébergeront les traitements,

- les fonctionnalités de lancement et de monitoring des tâches d'une application distribuée, sur les ressources allouées.

Comparé au premier environnement (voir section 5.4), le *JobTracker* et les *TaskTrackers* disparaissent, au profit d'un *ResourceManager* et de plusieurs *MRAppManager* (si on considère des applications *Map-Reduce*).

Vu de l'application cliente, il existe toujours un objet *job* qui sert de *stub* (ou *proxy*) pour s'interfacer facilement avec HDFS et les mécanismes d'*Hadoop* de gestion des applications. La figure 5.12 nous montre que le *stub* se charge comme précédemment de stocker dans HDFS les données spécifiques à l'application *Map-Reduce* en s'adressant au *NameNode*, et de soumettre un *job* à *Hadoop* en s'adressant cette fois-ci au *ResourceManager*. Mais le *ResourceManager* ne déploie pas lui même les tâches *Mappers* et *Reducers* sur les nœuds de données, il sélectionne un nœud de données sur lequel il installe un *MRAppManager* pour gérer l'application. C'est bien le *ResourceManager* qui va choisir les nœuds de données sur lesquels installer des tâches de traitement, mais c'est le *MRAppManager* qui va déployer ces tâches et suivre leur exécution (centraliser leurs informations de monitoring et détecter leurs éventuelles défaillances).

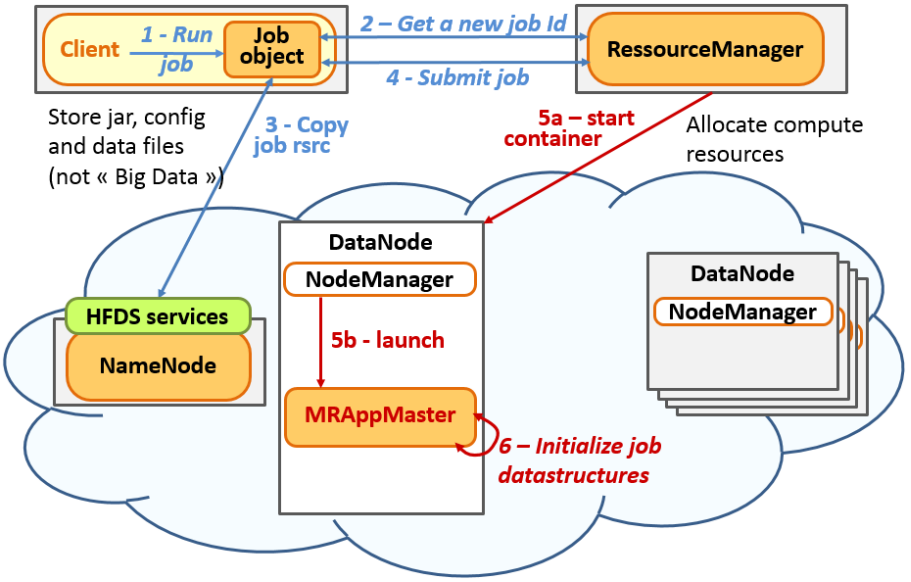

FIGURE 5.12 – Détail de la gestion d'un *Map-Reduce* sur *Hadoop* avec *YARN* (*Map-Reduce 2*) - AVANT création des tâches *Mappers* et *Reducers*

5.5.2 Mise en œuvre détaillée d'un *Map-Reduce* sur *Hadoop* version 2 (YARN)

La figure 5.12 montre le détail des premières étapes du lancement d'une application *Map-Reduce* sur *Hadoop* avec *YARN*. Les premières étapes (1 à 4) sont similaires à celles avec la première version de l'environnement d'exécution du *Map-Reduce* : le client demande à son *stub* d'exécuter son application *Map-Reduce* (étape 1), le *stub* demande alors l'identificateur d'un nouveau (futur) *job* au *ResourceManager* (étape 2), celui-ci vérifie que le *job* a été bien spécifié avant de lui accorder un identificateur, puis le *stub* stocke des données spécifiques à l'application dans le système de fichiers distribué HDFS (étape 3), et enfin le *stub* soumet le *job* au *ResourceManager* avec l'identificateur obtenu précédemment (étape 4).

Ensuite le *ResourceManager* choisit un nœud de données pour y lancer un service de gestion du *job* (étapes 5*a* et 5*b*). Dans un premier temps le *ResourceManager* s'adresse au service *NodeManager* du nœud visé, et lui demande de lancer localement un *container* incluant un *MRAppManager* (selon la terminologie de *YARN*). Le *MRAppManager* va alors compléter l'initialisation du *job* sur le nœud manager (étape 6). A ce stade, les tâches *Mappers* et *Reducers* n'existent pas encore, mais le *ResourceManager* et le *MRAppManager* vont maintenant collaborer pour les déployer.

La figure 5.13 montre que le *MRAppManager* commence par interroger le *NameNode* d'HDFS pour connaître les nœuds stockant les différents blocs des fichiers d'entrées du *job* (étape 7). Il communique alors cette liste de nœuds au *ResourceManager*, qui choisira les nœuds les plus adéquats pour héberger les tâches *Mappers* et *Reducers*, puis retournera cette nouvelle liste de nœuds choisis au *MRAppManager* (étape 8). Le *MRAppManager* va alors contacter le *NodeManager* de chaque nœud visé pour lui demander de lancer un *container* incluant une JVM avec un code *YarnChild* (étapes 9*a* et 9*b*), et c'est ce code qui démarrera finalement la tâche applicative (étape 10).

Une fois les tâches *Mappers* et *Reducers* lancées, elles envoient à leur *MRAppManager* des informations de progression et des *heartbeat* pour signaler qu'elles sont toujours en vie. Ces communications se font à intervalles de quelques secondes. Dès lors, le client peut interroger le gestionnaire de l'application par l'intermédiaire de son *stub* pour connaître la progression du *Map-Reduce* (voir le circuit en violet sur la figure 5.13). Comme prévu, toutes ces opérations de monitoring ne passent pas par le *ResourceManager*, qui peut donc se concentrer sur son rôle de gestionnaire et

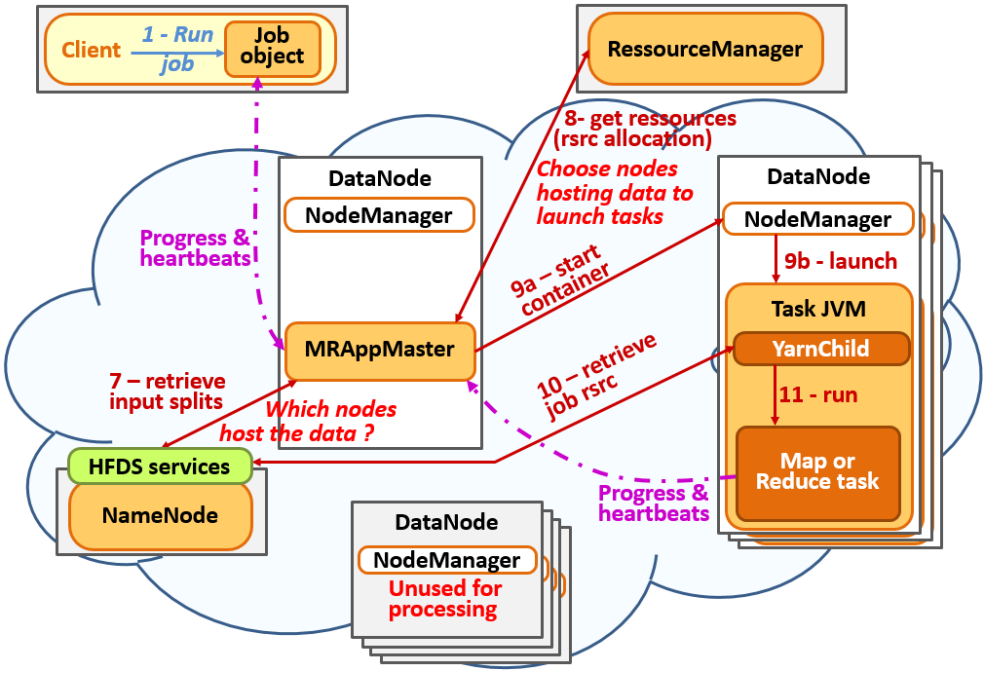

FIGURE 5.13 – Détail de la gestion d'un *Map-Reduce* sur *Hadoop* avec *YARN* (*Map-Reduce 2*) - A PARTIR de la création des tâches *Mappers* et *Reducers*

d'allocateur de ressources de données et de calculs à chaque fois qu'un nouveau *job* est soumis.

Le second environnement d'allocation de ressources et d'exécution d'applications d'*Hadoop* apparaît plus complexe que le premier. Mais grâce à *YARN* il est devenu possible de lancer plusieurs applications simultanées sur le cluster *Hadoop* sans créer un goulot d'étranglement, chacune ayant son propre gestionnaire (son *MRAppManager*). De plus il devient possible de lancer d'autres types d'applications distribuées, ne suivant pas un schéma *Map-Reduce*. Avec *YARN*, *Hadoop* est donc encore plus apte au passage à l'échelle.

L'architecture logicielle de *YARN* et de son *Map-Reduce* a toutefois un coût, et de nombreux utilisateurs ne sont pas satisfaits des performances d'*Hadoop*. D'autres environnements existent, qui peuvent réutiliser le système de fichiers distribués HDFS d'*Hadoop* mais qui redéfinissent la couche de gestion des applications pour obtenir plus de performances. Voir notamment *Spark* qui travaille en mémoire et évite au maximum de passer par des fichiers temporaires.

5.6 Exercices

5.6.1 Cluster Hadoop *vs* cluster HPC

Dans une architecture classique de centre de calcul HPC, on trouve des "baies de nœuds de calculs très fiables" qui lisent leurs données et écrivent leurs résultats finaux dans des "baies de disques également très fiables" accédées à travers un réseau rapide.

1. A l'opposé, quel type de matériel trouve-t-on dans une architecture distribuée de stockage et de traitement de données comme *Hadoop* (cluster *Hadoop*) ?

2. Quel est l'impact sur la politique de tolérance aux pannes d'*Hadoop* ?

5.6.2 Fonctionnement d'un cluster *Hadoop*

1. Que se passe-t-il lorsqu'un nœud de données qui n'accueille aucun calcul tombe en panne ?

2. Quels critères sont utilisés pour décider sur quels nœuds exécuter un traitement des données ?

Bibliographie

[1] G. Amdahl. Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities. In *AFIPS Conference Proceedings*, volume 30, pages 483–485, 1967.

[2] B. Azarmi. *Scalable Big Data Architecture*. Apress, 2016.

[3] R. Bruchez. *Les bases de données NoSQL et le Big Data*. Eyrolles, 2ème edition, 2016.

[4] R. Bruchez and M. Lutz. *Data science : fondamentaux et études de cas*. Eyrolles, 2015.

[5] B. Chapman, G. Jost, R. Van Der Pas, and D. Kuck. *Using OpenMP*. The MIT Press, 2008.

[6] K. Chodorow. *MongoDB, the Definitive Guide*. O'Reilly, 2ème edition, 2013.

[7] K. Dowd and Ch. Severance. *High Performance Computing*. O'Reilly, 2nd edition, 2008.

[8] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI*. The MIT Press, 1999.

[9] J.L. Gustafson. Reevaluating amdahl's law. *Communications of the ACM*, 31 :532–533, 1988.

[10] H. Karau, A. Konwinski, P.Wendell, and M.Zaharia. *Learning Spark*. O'Reilly, 1st edition, 2015.

[11] H. Karau and R. Warren. *High Performance Spark*. O'Reilly, 1st edition, 2017.

[12] M. Kirk. *Thoughtful Machine Learning with Python*. O'Reilly, 2017.

[13] P. Lemberger, M. Batty, M. Morel, and J-L. Raffaelli. *Big Data et Machine Learning*. Dunod, 2015.

[14] D. Miner and A. Shook. *MapReduce Design Patterns*. O'Reilly, 2013.

[15] T. White. *Hadoop. The definitive Guide*. O'Reilly, 3rd edition, 2013.

[16] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M.J. Franklin, S. Shenker, and I. Stoica. Resilient Distributed Datasets : A Fault-tolerant Abstraction for In-memory Cluster Computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, 2012.