Chapitre 6

Bases d'algorithmique distribuée Map-Reduce pour l'analyse de données

Ce chapitre présente la conception d'algorithmes distribués selon le paradigme *Map-Reduce* dans un environnement qui implante déjà un système de fichiers distribué et un *middleware* distribué adapté aux architectures logicielles *Map-Reduce* (comme *Hadoop*). Divers schémas *Map-Reduce* élémentaires sont étudiés, avec un objectif de minimisation du volume de données échangées et ensuite de minimisation des calculs.

6.1 Etapes de l'approche algorithmique

Concevoir des solutions algorithmiques selon le paradigme *Map-Reduce* permet de traiter des problèmes à grande échelle sur des architectures distribuées. Néanmoins cette démarche algorithmique reste originale, et concevoir des algorithmes *Map-Reduce* efficace n'est pas immédiat. Ce chapitre présente quelques patrons de conception *Map-Reduce* optimisés et facilement réutilisables, largement inspirés de [14]. Ces patrons visent 4 objectifs principaux :

1. Identifier des tâches *Mapper* et *Reducer* permettant de résoudre des problèmes d'analyse de données. On cherche à concevoir le plus possible des algorithmes *Map-Reduce* en une seule passe, car enchaîner les opérations *Map-Reduce* à souvent un coût élevé en écriture-relecture de fichiers temporaires et en lancement de tâches.

2. Définir des routines *Combiner* quand elles permettent de réduire les volumes de données en sortie des *Mappers*, afin d'éviter : (1) d'importants stockages temporaires sur disques, (2) de volumineux routages de données à travers le réseau d'interconnexion lors du *Shuffle & Sort*, et (3) des traitements de très grandes listes de valeurs dans les *Reducer* pouvant saturer leurs mémoires. Les *Combiners* peuvent être très utiles, mais tous les algorithmes ne s'y prêtent pas.

3. Identifier le bon nombre de *Reducers* à déployer pour paralléliser les derniers traitements tout en permettant un équilibrage de leur charge de calcul.

4. Définir un *Partitioner* avec de nouvelles règles de distribution des clés aux *Reducers* pour améliorer la répartition de charge, mais seulement si l'on possède des informations suffisantes sur la variétés des clés et sur la distribution des paires clé-valeur.

5. Si besoin, redéfinir les fonctions de *keyComparator* et de *groupComparator* pour réaliser des opérations à l'occasion du tri des clés fait systématiquement par *Hadoop*.

6.2 Patrons de récapitulation (*summarization*)

Les patrons de récapitulation (ou de *summarization*), agrègent/accumulent des informations concernant des données de même clé. Le *Word counter* (sorte de *Hello World* de l'analyse de données) en fait partie : il accumule des jetons de présence de mots dans un ensemble de fichiers, et permet finalement de compter le nombre d'occurrences de chaque mot. Mais ces patrons permettent aussi de réaliser des calculs plus complexes, comme des moyennes, des valeurs médianes, des écarts types...

6.2.1 Comptage d'occurrences de termes

La figure 6.1 montre un enchaînement d'operations *Map* et *Reduce*, avec des passages possibles mais non garantis dans un *Combiner*, pour au final compter des nombres d'occurrences de termes (simples mots, ou logins, ou adresses IP...). Les *Mappers* se chargent de lire les fichiers d'entrées segmentés en blocs : chaque tâche *Mapper* lit et traite un *bloc* de fichier (typiquement $64Mo$ de données). Chaque bloc est lu pour former une succession de paires clé-valeur d'entrées. Habituellement, si on cherche à lire un fichier texte on récupère des paires constituées chacune d'une ligne de texte (terminée par un retour à la ligne) en guise de valeur, et de la position du début de cette ligne dans le fichier (son *offset*) en guise de clé.

Comme le montre le haut de la figure 6.1, à l'intérieur d'un *Mapper* la fonction *map* est appelée sur chaque paire clé-valeur, et identifie la suite de termes contenus dans la valeur (elle sépare les mots de la ligne de texte constituant cette valeur). Pour chaque occurrence d'un terme, la fonction *map* écrit alors en sortie une paire clé-valeur composée du terme comme clé, et du nombre 1 comme valeur (signifiant "une occurrence du terme a été détectée"). Notons que le premier *for* du code du *Mapper* de la figure 6.1 est en grisé, car si on considère que la fonction *map* est automatiquement appelée pour chaque paire clé-valeur d'entrée, alors ce *for* est implicite.

Il est possible de ne pas implanter de *Combiner* et de laisser les *Mappers* déverser toutes leurs paires clé-valeur de sorties directement dans le mécanisme de *Shuffle & Sort*. Les *Reducers* reçoivent alors des paires constituées d'un terme comme clé, et d'une liste de toutes les valeurs associées à ce terme, c'est-à-dire une liste de 1 ($[1, 1, 1...]$). Chacune de ces paires est envoyée en entrée de la fonction *reduce* (voir le bas de la figure 6.1), qui somme les 1 de la liste de valeurs et calcule ainsi le nombre d'occurrences de la clé (du terme) dans toutes les données analysées.

Cette solution simple suffit pour compter les occurrences des termes d'un ensemble de textes, mais elle engendre un très gros trafic sur le réseau à l'occasion du *Shuffle & Sort*, car chaque occurrence de terme génère une paire < *terme, 1* >. Une optimisation très intéressante ici consiste à implanter un *Combiner* qui va jouer le rôle d'un *Reducer* local à la sortie de chaque *Mapper*, voir le premier cadre orange de la figure 6.1 (celui qui est barré). Dès lors, si chaque terme est statistiquement souvent présent dans les textes analysés, de nombreuses paire < *terme, 1* > vont se réduire en une seule paire < *terme, n* > (avec $n > 1$), ce qui allégera le trafic dans l'étape de *Shuffle & Sort*.

Mais le déclenchement du *Combiner* est entièrement contrôlé par *Hadoop*. Il décide de l'appliquer ou non, et décide du nombre de paires de sorties sur lesquelles il l'applique. Par exemple, si un *Mapper* génère trop peu de paires de sorties, *Hadoop* ne lancera pas du tout son *Combiner*. Au contraire, si un *Mapper* génère 10005 de paires de sorties, *Hadoop* peut décider de lancer progressivement son *Combiner* sur 10 tranches de 1000 paires de sorties, et de ne pas le lancer sur les 5 dernières. En conséquence, les *Reducers* peuvent recevoir des paires provenant des *Combiners*

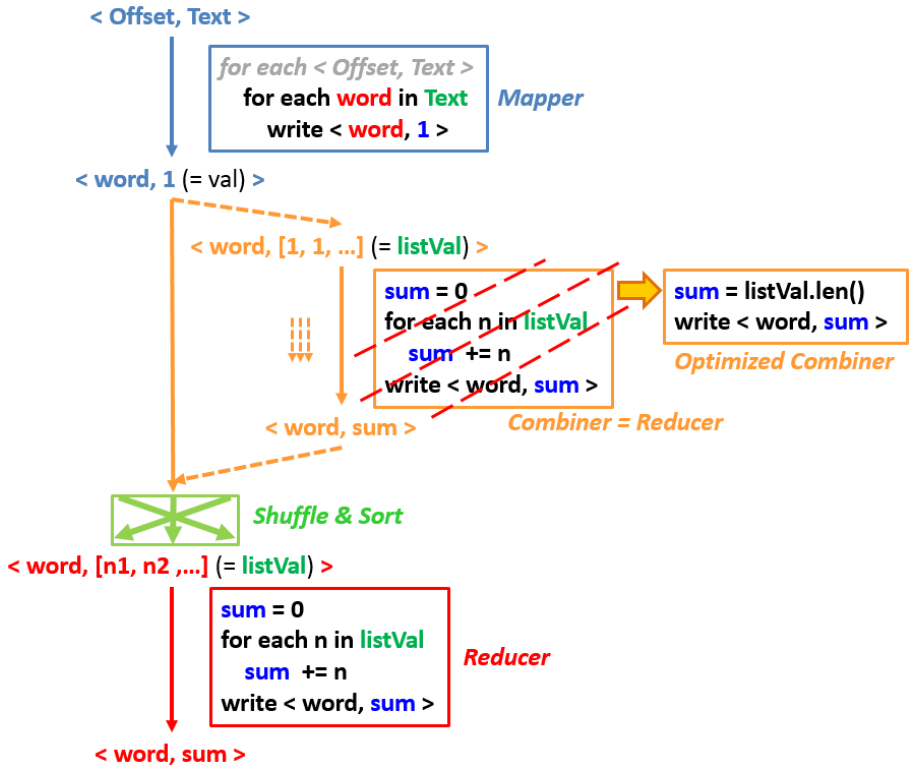

FIGURE 6.1 – Patron *Map-Reduce* de comptage d'occurrences de termes

ou directement des *Mappers*. **Les formats des paires de sorties des *Combiners* et des *Mappers* doivent donc être identiques**.

Le *Combiner* peut exécuter le même code que le *Reducer*, ou un code optimisé, ou encore un code totalement différent. Le cadre orange le plus à droite de la figure 6.1 permet au *Combiner* de réduire le nombre de paires de sorties du *Mapper* en considérant que pour chaque clé il reçoit bien une liste de "1" ($[1, 1, 1...]$). Dès lors la somme des valeurs est égale à la longueur de la liste, que l'on peut éviter de calculer en interrogeant la liste. En revanche, les *Reducers* doivent continuer à additionner les valeurs de leurs listes, car suite aux actions du *Combiner* elles ne contiennent pas que des "1".

La figure 6.2 résume les transformations de paires clé-valeur dans la chaîne *Map-Reduce* de ce compteur d'occurrences de termes. On y voit un processus *Mapper* produire 7 paires, dont les trois premières sont réduites à deux par un premier appel au *Combiner*, puis les deux suivantes sont réduites à une seule par un deuxième appel au *Combiner*, et enfin les deux dernières paires ne sont pas traitées par le *Combiner*. Le *Shuffle & Sort* reçoit donc 2 paires de sorties du *Mapper* et 3 paires de sorties de son *Combiner*, plus d'autres paires provenant d'autres *Mappers* (et de leurs *Combiners*). En bas à droite de la figure 6.2 on peut visualiser des paires d'entrées de *Reducers* contenant chacune une clé (un terme) et une liste de valeurs (une liste de nombres d'occurrences), qui sont finalement réduites en paires clé-valeurs finales (un terme et son nombre total d'occurrences dans les fichiers analysés).

Ce patron se déploie avec un grand nombre de tâches *Mapper*, une par bloc de fichier d'entrée (mais dont on peut limiter le nombre simultanément actifs sur un même nœud, voir section 5.2.4), et avec un nombre de *Reducers* dont l'optimum dépend du contenu des fichiers analysés ! Le nombre de termes différents rencontrés dans les fichiers analysés impose le nombre de paires < *clé - liste de valeurs* > traitées par les *Reducers*, puisque chaque terme correspondra finalement à

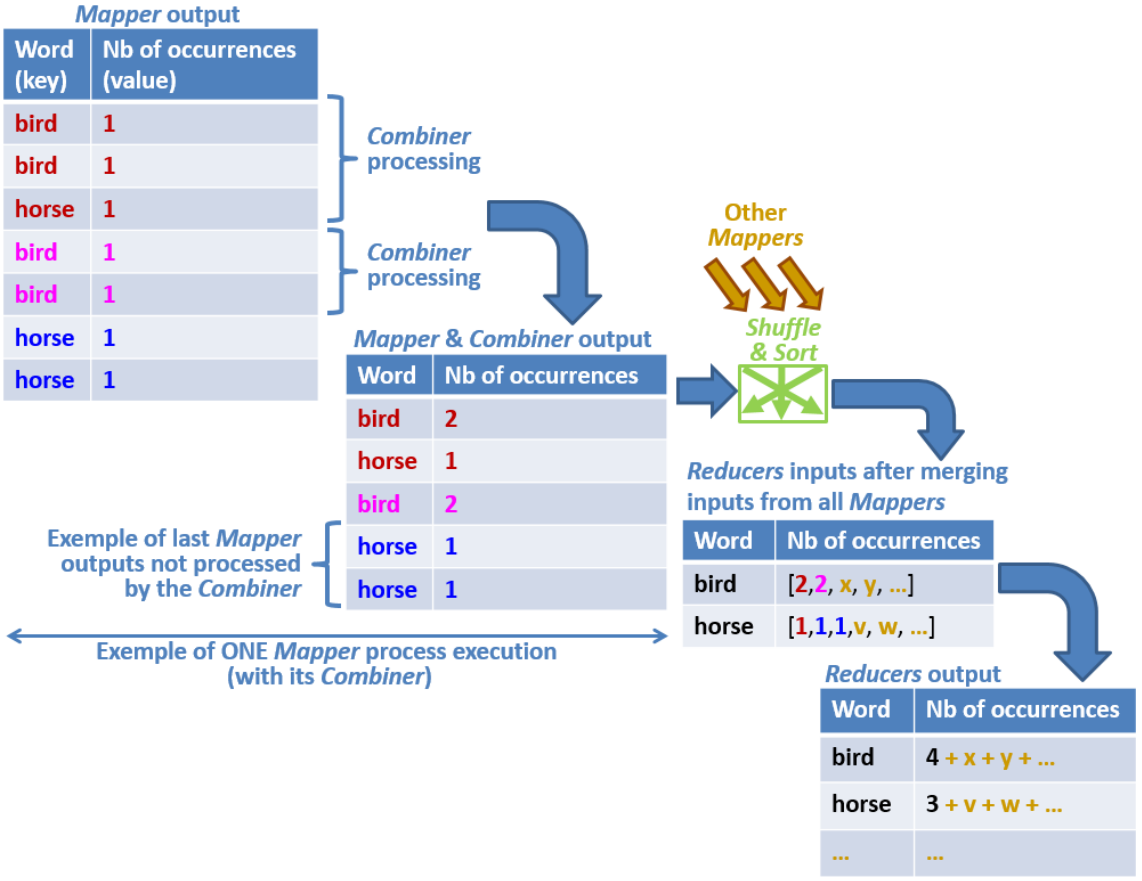

FIGURE 6.2 – Enchaînement des transformations de paires clé-valeurs du patron de comptage d'occurrences de termes

une clé. Il est donc inutile de déployer trop de *Reducers*, qui traiteraient chacun peu de termes et de listes d'occurrences, car ils auraient trop peu de travail pour rentabiliser leur création. Il semble donc logique de considérer que l'on créera beaucoup plus de *Mappers* que de *Reducers*, comme illustré sur la figure 6.3.

Enfin, si l'on connaît la variété des termes rencontrés on peut optimiser le *Partitioner* pour équilibrer au mieux la charge de travail des *Reducers*. Mais cela demande des connaissances a priori sur les résultats attendu de l'étude, ce qui est possible quand on refait régulièrement la même analyse sur des données simplement actualisées.

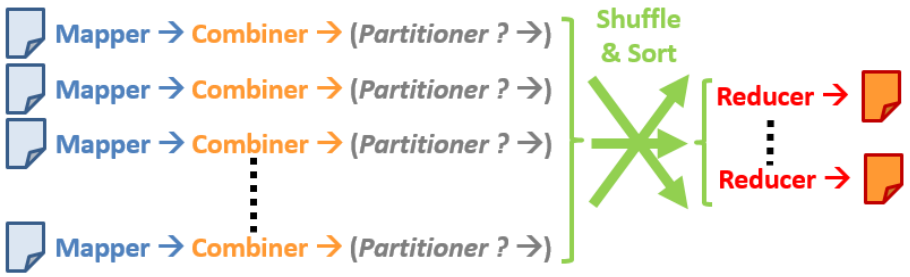

FIGURE 6.3 – Schéma de déploiement du patron de comptage d'occurrences de termes

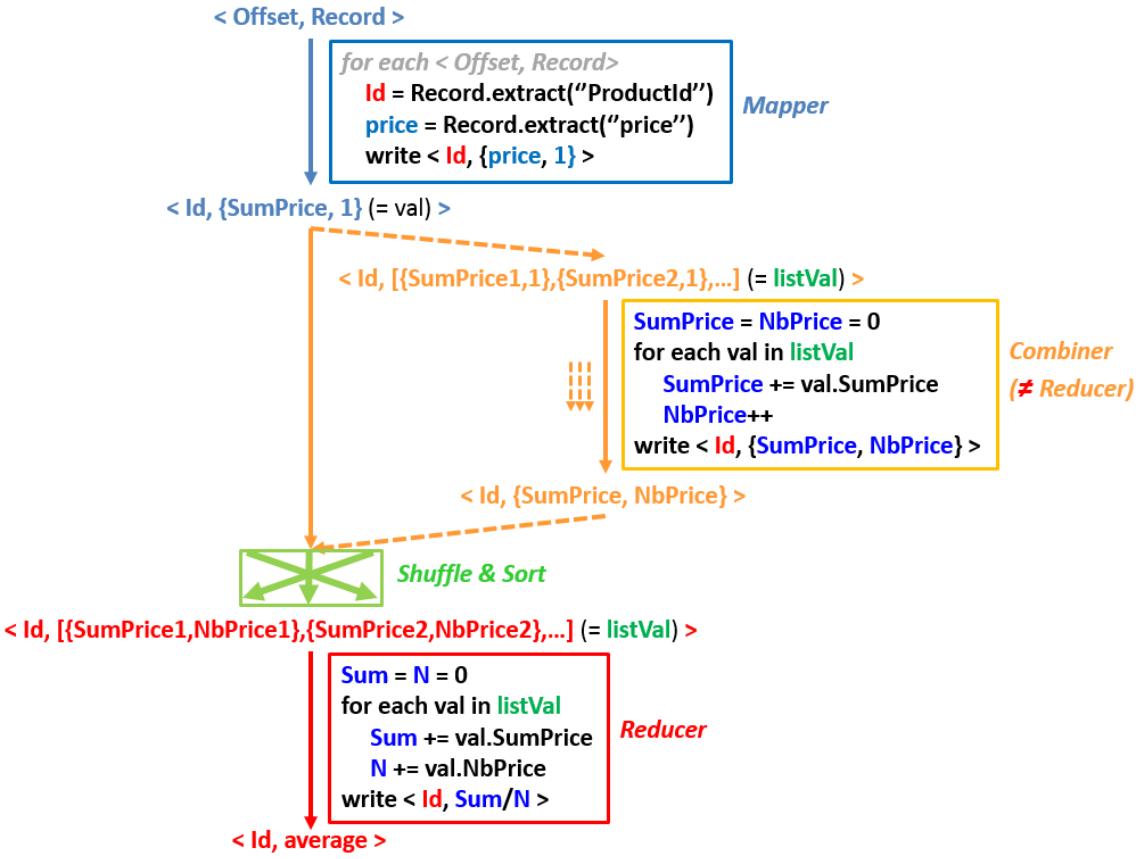

FIGURE 6.4 – Patron *Map-Reduce* de calcul de moyennes

6.2.2 Calcul d'une moyenne avec optimisation des communications

La figure 6.4 décrit un patron *Map-Reduce* un peu plus compliqué que le précédent, qui calcule des moyennes. En l'occurrence il s'agit de calculer le prix de vente moyen de chaque produit vendu par une société. Chaque produit est identifié par une référence unique (*Id*), et le prix d'un même produit varie en fonction de la date de la transaction, du volume acheté, de la fidélité du client... Au final on souhaite connaître le prix de vente moyen de chaque référence.

Un algorithme *Map-Reduce* simple comprendrait seulement des *Mappers* et des *Reducers*. Les *Mappers* généreraient des paires *< Id, OnePrice >*, et chaque *Reducer* traiterait des paires ayant un *Id* comme clé, et une liste de prix unitaires *[OnePrice1, OnePrice2, ...]* comme liste de valeurs. Le *Reducer* n'aurait alors plus qu'à calculer la moyenne de cette liste de prix pour obtenir le prix moyen du produit (de référence *Id*). Mais cela provoquerait un gros trafic lors du *Shuffle & Sort*, et comme tous les calculs se feraient dans les *Reducers*, la mémoire d'un *Reducer* pourrait saturer si un des produits avait été vendu un très grand nombre de fois (liste de prix trop volumineuse). Pour éviter ces écueils on souhaite vivement implanter un *Combiner* à la sortie des *Mappers*, afin de calculer des résultats partiels et de réduire le volume de données routées vers les *Reducers*.

Malheureusement, le calcul d'une moyenne n'est pas complètement associatif : on ne peut pas calculer des moyennes partielles dans les appels au *Combiner*, puis faire simplement des moyennes de moyennes dans les *Reducers*. Il faut associer chaque moyenne à son poids (le nombre de valeurs initiales qu'elle représente) et calculer des moyennes pondérées dans les *Reducers*. Il faut donc communiquer chaque moyenne partielle avec son poids à la sortie de chaque appel au *Combiner*, afin de pouvoir faire les calculs des moyennes finales dans les *Reducers* : les paires de sortie du

Combiner doivent avoir des doublets *{MoyenneParielle, Poids}* comme valeur. Mais cela signifie de faire des divisions lors des calculs de moyennes partielles dans les appels au *Combiner*, puis des multiplications par les poids et à nouveau des divisions dans les calculs de moyennes finales des *Reducers*. Cette solutions alourdit donc les calculs avec de nombreuses multiplications et divisions.

Une solution plus simple et plus élégante est décrite sur la figure 6.4. Les *Mappers* génèrent des paires dont la clé est la référence du produit (*Id*), et dont la valeur est une valeur composite : un doublet *{price,1}* qui représente un prix et son poids unitaire ("le poids d'un seul prix"), voir le haut de la figure 6.4.

$Write < Id, \{price, 1\} >$

Cette démarche alourdit dans un premier temps le volume des données générées par les *Mappers* en utilisant des valeurs composites. Mais il est maintenant possible d'implanter un *Combiner* qui somme les prix de ventes d'un même produit ainsi que leurs poids unitaires, et qui génère des paires < *Id, {*$SumPrice_{Id}$*,* $NbPrice_{Id}$*}* > (voir le milieu de la figure 6.4).

$Sum_{Id} = \sum_{j=0}^{j<n_{Id}} v_{Id}^{j}$

avec n_{Id} : le nombre de valeurs (composites) de la liste traitée par l'appel au *Combiner*

$Write < Id, \{Sum_{Id}, n_{Id}\} >$

Cette action des *Combiner* permet de réduire fortement le nombre de paires routées vers les *Reducers* lors du *Shuffle & Sort*. Ensuite les *Reducers* reçoivent des paires ayant toujours une référence de produit comme clé, et une liste de doublets *[{*$SumPrice_{Id}^{0}$*,* $NbPrice_{Id}^{0}$*}, {*$SumPrice_{Id}^{1}$*,* $NbPrice_{Id}^{1}$*},...]* comme liste de valeurs. Ils additionnent alors les sommes partielles reçus ainsi que leurs poids, calculent un prix moyen pour chaque référence de produit, et écrivent finalement une paire < *Id,* $Average_{Id}$ > pour chaque référence de produit (voir le bas de la figure 6.4).

$$Average_{Id} = \frac{\sum_{k=0}^{k<NSum_{Id}} Sum_{Id}^{k}.n_{Id}^{k}}{\sum_{k=0}^{k<NSum_{Id}} n_{Id}^{k}}$$

avec $NSum_{Id}$: le nombre de valeurs (composites) de la liste traitée par l'appel au *Reducer*

$Write < Id, Average_{Id} >$

On notera que certaines paires routées vers les *Reducers* pourraient provenir directement des *Mappers*, mais que leur format et leur sémantique sont totalement compatibles avec ceux des paires de sortie du*Combiner*.

Dans ce patron, le *Combiner* et le *Reducer* ont des codes un peu plus différents que dans le patron précédent de comptage d'occurrences. Cette fois-ci les formats des paires de sorties écrites par le *Combiner* et le *Reducer* sont différents. En revanche, le déploiement de ce patron est le même que celui du compteur d'occurrences (voir figure 6.3). On installe autant de *Mappers* que de blocs de fichiers à lire, et un nombre de *Reducers* plus faible mais à augmenter avec le nombre de produits rencontrés (c'est-à-dire avec le nombre de clés différentes).

6.2.3 Réalisation d'un index inversé

Les index inversés sont des outils indispensables aujourd'hui. Sans index inversé, une requête à un moteur de recherche se traduirait par une série de requêtes envoyées à travers le monde, pour analyser chaque document accessible et rechercher quelques mots-clés. Avec un index inversé on se contente d'interroger cet index pour obtenir la liste des références sur les documents contenant les termes recherchés. L'index inversé est mis à jour de temps en temps/régulièrement/en permanence, mais indépendamment du traitement des requêtes. On ne travaille donc pas sur des données

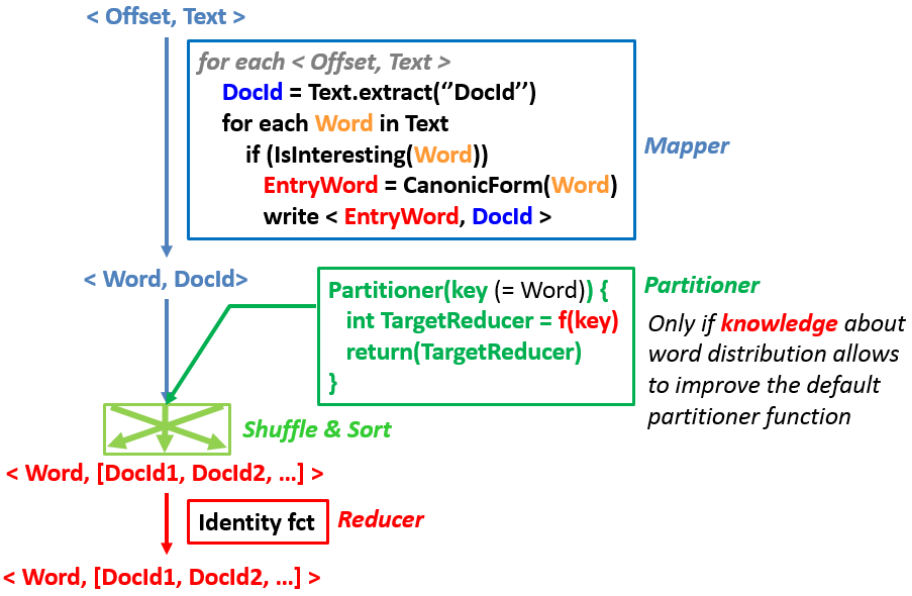

FIGURE 6.5 – Patron *Map-Reduce* de réalisation d'un index inversé

totalement à jour pour satisfaire les requêtes, mais cela est inévitable en *Big Data*, et surtout en *web-scale*.

La figure 6.5 décrit le patron *Map-Reduce* utilisé pour réaliser un index inversé. Les *Mappers* se chargent de lire et d'analyser les fichiers d'entrées. Ils dressent la liste de tous les *mots* contenus dans chaque fichier, et génèrent des paires clé-valeur constituées d'un mot reconnu (la clé), et de la référence du document (la valeur). Habituellement on réalise un code *Mapper* intelligent, qui ne génère pas une nouvelle clé à chaque mot trouvé. Il commence par éliminer certains mots considérés non-pertinents, comme les articles et les déterminants, ou des mots que l'utilisateur veut explicitement ne pas considérer. Pour réaliser des index inversés focalisés sur un ensemble de termes limités, le *Mapper* peut aussi rejeter tout mot qui n'est pas dans un ensemble prédéterminé. Tous ces filtrages sont illustrés par le test *if (IsInteresting(Word))* du code *Mapper* de la figure 6.5. Une fois un mot identifié et jugé pertinent, le *Mapper* peut en extraire une forme canonique, par exemple en le considérant au singulier, au masculin, à l'infinitif..., comme illustré par la fonction *CanonicForm(Word)* du code *Mapper* de la figure 6.5. Enfin, un *Mapper* peut aussi stocker en mémoire tous les mots trouvés dans son bloc de fichier pour éliminer les doublons (ne pas re-stocker un mot déjà trouvé dans le fichier en cours d'analyse). Cette démarche alourdit toutefois les *Mappers*, et n'est pas illustrée sur la figure 6.5.

Ensuite, le *Shuffle & Sort* et les *Reducers* sont utiles pour agréger les résultats, et associer chaque clé, donc chaque mot, à une liste de références de documents. Mais le *Reducer* n'a pas forcément de traitement à faire dans sa fonction *reduce*, qui peut être une fonction *Identity* (voir 6.5). Eventuellement, ils peuvent effectuer un changement de format : stocker la liste des références d'un mot dans une *String* pour faciliter des manipulations ultérieures, ou bien poursuivre l'élimination des doublons en éliminant les références multiples à un même mot apparu dans deux blocs distincts d'un même fichier.

En fait, si les documents analysés ne sont pas trop gros, s'ils sont plus petits qu'un bloc de fichier d'HDFS, alors chaque document sera traité intégralement par un même *Mapper*. L'élimination des doublons peut alors se faire entièrement dans les *Mappers*, et toutes les paires de sorties d'un même *Mapper* auront donc des clés différentes. Comme la plupart des documents analysés par un moteur de recherche ne sont pas très gros (ils sont très nombreux, mais pas très gros), cette

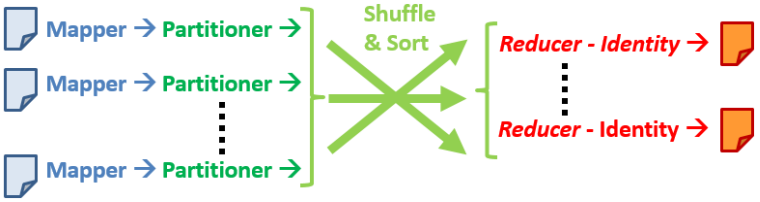

FIGURE 6.6 – Schéma de déploiement du patron *Map-Reduce* d'index inversé

hypothèse sera souvent vérifiée. Dès lors, un *Combiner* serait sans aucun effet : il n'y aura pas de liste de valeurs à réduire. C'est pourquoi il n'y a pas de *Combiner* sur la figure 6.5.

En revanche, les clés (les mots identifiés) sont source d'optimisation si on connaît leur nombre et leur distribution :

- La réalisation d'un index inversée peut se prêter à l'écriture d'un *Partitioner* dédié et optimisé pour le problème, par exemple, en fonction de la langue des documents. Une bonne fonction de hachage pour le dictionnaire français, ne sera pas forcément aussi efficace sur un dictionnaire anglais ou allemand. On peut également construire une fonction de partitionnement équilibrant la charge des *Reducers* si l'on connaît une estimation de la distribution des termes recherchés dans les documents analysés.

- Le nombre de *Reducers* peut être important si le nombre de mots identifiés l'est aussi. Par exemple, si l'on cherche à indexer tout les mots d'une langue naturelle (sauf les articles et déterminants) dans un énorme corpus de documents, alors on peut utiliser un grand nombre de *Reducers*.

La figure 6.6 illustre le déploiement d'une architecture *Map-Reduce* réalisant un index inversé. On y voit un ensemble de tâches *Mapper*, sans *Combiner*, déversant leurs paires clé-valeur dans un *Shuffle & Sort* avec l'aide d'un *Partitioner* optimisé, pour alimenter des *Reducers*. Ces derniers réalisent alors automatiquement une agrégation des sorties des différents *Mappers* et se contentent d'exécuter une fonction identité.

6.2.4 Optimisation d'un calcul de médiane et d'écart type

Le calcul de valeurs médianes et d'écarts types posent un problème algorithmique plus complexe en *Map-Reduce*, car on ne peut pas combiner linéairement des résultats intermédiaires. Il est plus compliqué de décomposer le travail en tâches *Mappers* et *Reducers*, et de trouver ensuite un moyen d'optimiser la phase de *Shuffle & Sort*.

Voir TD.

6.3 Patrons de *Filtrage*

Les patrons de filtrages sont a priori plus simples à concevoir. Ils consistent à éliminer certains enregistrements d'entrée (des fichiers ou des parties de fichiers), ou à n'accepter que ceux satisfaisant un certain critère. Ces patrons font surtout intervenir les *Mappers*, et peuvent très bien ne pas avoir besoin des étapes de *Shuffle & Sort* et de *Reducer*.

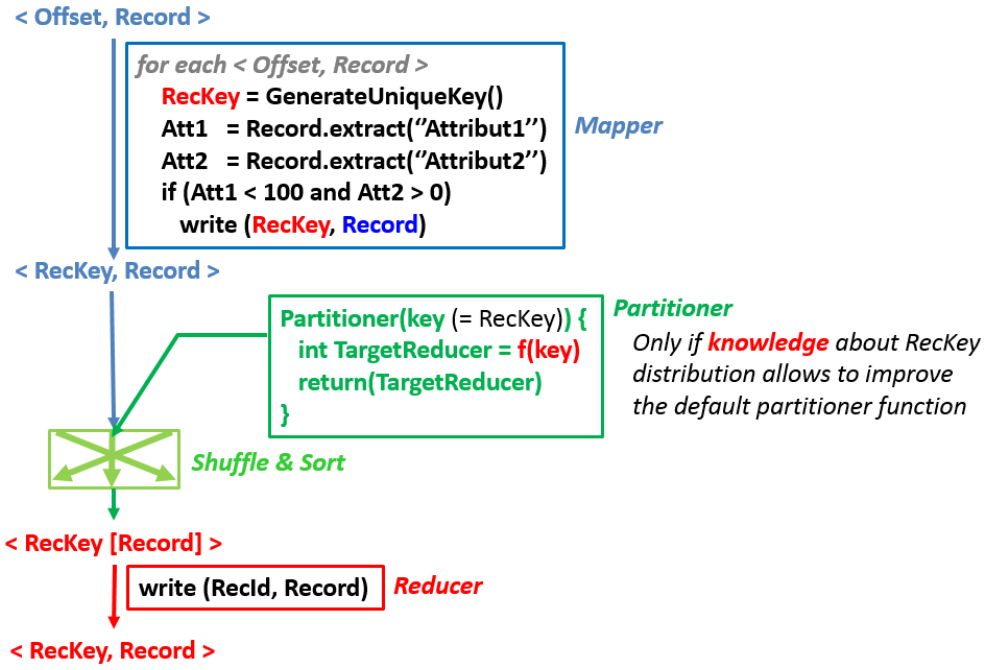

FIGURE 6.7 – Patron *Map-Reduce* de filtrage accepter/rejeter

6.3.1 Filtrage *accepter/rejeter*

Beaucoup de filtrages *accepter/rejeter* implantés avec le paradigme *Map-Reduce* reposent essentiellement sur les tâches *Mappers*. Chaque *Mapper* se charge de lire une suite d'enregistrements dans un document et d'appliquer sur chacun une série de tests sur ses attributs, suite à ces tests chaque enregistrement est accepté ou rejeté. En cas d'acceptation de l'enregistrement, le *Mapper* génère une paire clé-valeur, dont la valeur est l'enregistrement lui-même, et dont la clé doit être une clé unique (associée à cet enregistrement). En cas de rejet, le *Mapper* ne fait rien et passe à l'enregistrement suivant. Le haut de la figure 6.7 illustre un tel schéma de calcul, où deux attributs sont extraits de chaque enregistrement et où on vérifie la condition *(attribut1 < 100 and attribut2 > 0)* pour accepter l'enregistrement et générer la paire de sortie du *Mapper*.

Comme indiqué précédemment, on génère habituellement une clé unique par enregistrement. Celle-ci peut être extraite de l'enregistrement si ce-dernier possède un identificateur unique (numéro absolu de message, référence universelle de document...), ou bien générée par le *Mapper* si besoin (ce qui est le cas dans l'exemple de la figure 6.7).

Les *Reducers* recevront alors en entrée des paires de type < *clé-unique, [Record-unique]* >, et auront un comportement proche de l'identité. Ils ne feront éventuellement qu'une remise en forme de la paire d'entrée en supprimant la structure de liste entourant son unique enregistrement (voir le bas de la figure 6.7). Une autre solution est d'interrompre la chaîne *Map-Reduce* à la fin des *Mappers*, ce qui est possible en *Hadoop* en fixant le nombre de *Reducers* à 0.

Aucun *Combiner* ne peut être implanté car il n'y a pas d'opération de réduction associée à ce patron de filtrage. En revanche, si on n'interrompt pas la chaîne de *Map-Reduce* et que l'on implante des *Reducers*, il est pertinent d'optimiser la distribution des paires clé-valeur vers les *Reducers*. En effet, si le filtre n'est pas trop sélectif de nombreuses paires clé-valeur seront générées. Dès lors il est intéressant de déployer de nombreux *Reducers* pour équilibrer la charge d'écriture des paires clé-valeurs de sortie dans des fichiers, et un *Partitioner* spécifique répartissant bien les nombreuses clés sur les différents *Reducers* sera efficace (comme illustré sur la figure 6.7). Cela re-

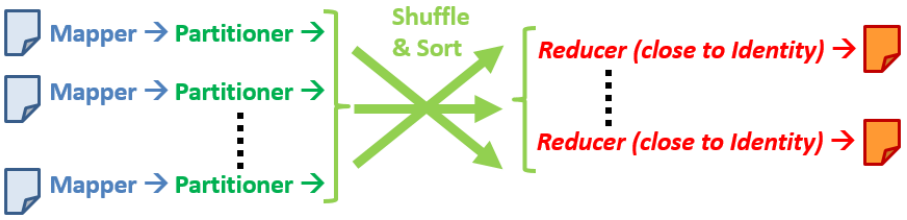

FIGURE 6.8 – Schéma de déploiement du patron de filtrage accepter/rejeter

vient à concevoir de concert la fonction de génération des clés uniques (utilisée dans les *Mappers*) et la fonction de calcul du *Reducer cible* en fonction de ces clés (utilisée dans le *Partitioner*).

La figure 6.8 montre le déploiement de ce patron de filtrage dans le cas où la chaîne de traitement n'est pas interrompue, et où un *Partitioner* a été implanté.

Si au contraire on souhaite regrouper dans un même fichier plusieurs enregistrements acceptés par le filtre, et partageant une même caractéristique, alors les *Mappers* vont générer des clés communes à ces enregistrements. Les *Reducers* vont ensuite recevoir des listes d'enregistrements associés à une même clé, et pouvoir écrire dans un même fichier tous les enregistrements acceptés et partageant une caractéristique commune. On aura ainsi réalisé en un seul traitement *Map-Reduce* le filtrage et le regroupement des données filtrées.

Enfin, si l'on compare ce filtrage *Map-Reduce* à un filtrage en SQL dans une BdD traditionnelle, alors la requête SQL la plus proche serait du type :

SELECT * FROM table WHERE condition

Remarque : Certains filtres sont conçus pour éliminer les documents indésirables (*Bloom Filtering*), plutôt que pour accepter certains documents. Ce sont des filtres qui garantissent qu'il n'y aura pas de faux négatifs (tous les documents rejetés doivent bien l'être), mais qui peuvent générer quelques faux positifs (certains documents sont conservés inutilement). On parle alors plutôt d'action *conserver/rejetter* plutôt que d'action *accepter/rejetter*.

6.3.2 Filtrage *Top Ten / Top K*

Un filtrage *Top K* consiste à filtrer des données initiales, comme précédemment, mais aussi à les ordonner selon un critère, et à ne retenir ensuite que les K meilleures données (avec $K = 10$ on obtient un *Top Ten*). La requête SQL la plus proche serait du type :

SELECT * FROM table WHERE condition ORDER BY attribut DESC LIMIT 10

Ce patron utilise n *Mappers* mais un seul *Reducer*, comme indiqué sur la figure 6.9. Chaque *Mapper* réalise le filtrage mais aussi un classement local des données qu'il a acceptées, et écrit alors K paires clé-valeur avec une clé unique sans importance (une *null-key* par exemple) et avec ses K meilleurs enregistrements comme valeurs (comme illustré sur la figure 6.10). L'unique *Reducer* reçoit ensuite une paire composée de la *null-key* et de la liste des $n \times K$ enregistrements écrites par les n *Mappers*. Il classe cette liste de $n \times K$ enregistrements, et identifie les K meilleurs enregistrements parmi tous ceux retenus après le filtrage des *Mappers*. Finalement, le *Reducer* écrit K paires de sorties composées des K meilleurs enregistrement avec leur rang en guise de clé (voir le bas de la figure 6.10).

Il n'y a pas d'optimisation possible de la chaîne *Map-Reduce* dans ce patron. Tout d'abord il n'y a pas de place pour un *Combiner* travaillant séparément sur les sorties de chaque *Mapper*, qui sont déjà réduites aux K meilleurs enregistrements locaux. Ensuite il n'y a qu'un seul *Reducer*, donc il n'y a pas d'optimisation possible de la distribution des clés par un *Partitioner*.

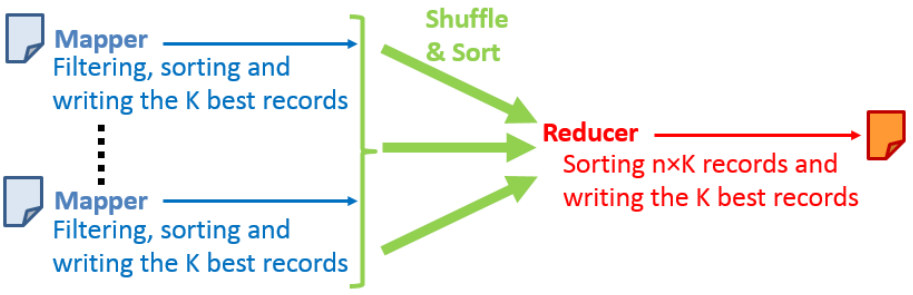

FIGURE 6.9 – Principe et schéma de déploiement du patron *Map-Reduce* de filtrage *Top Ten*

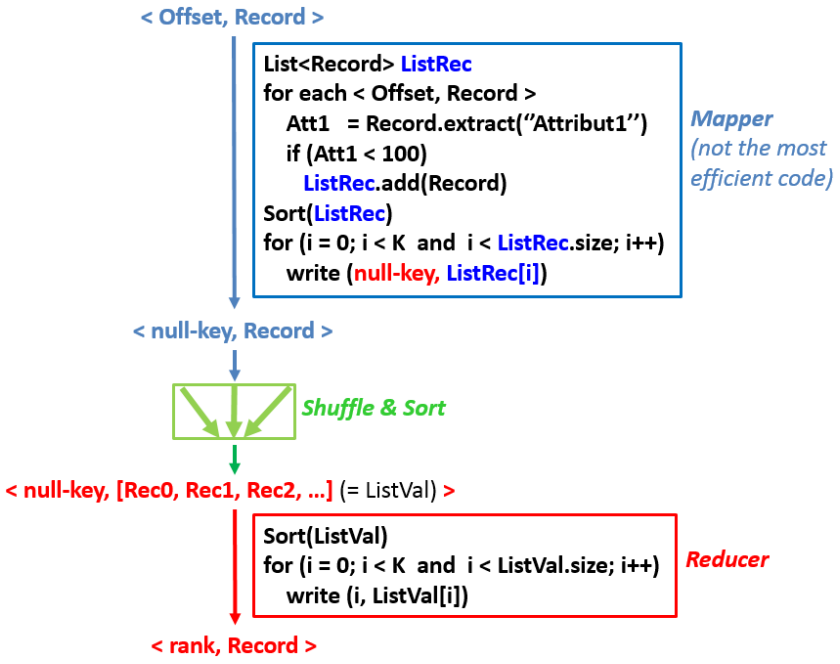

FIGURE 6.10 – Patron *Map-Reduce* de filtrage *Top Ten*

Le point faible de ce patron est son *Reducer* unique. Si le nombre de données, donc de blocs HDFS, donc de *Mappers* est important, alors le nombre de paires envoyées à l'unique *Reducer* le sera aussi ($n \times K$). Cela peut surcharger la mémoire du nœud qui héberge le *Reducer*, surtout si on ne cherche pas particulièrement les 10 premières valeurs mais plutôt les 50% meilleures, avec un critère de filtrage peu sélectif ! Ce sera alors au *Reducer* de trier toutes les valeurs retenues et transmises par les *Mappers*, et ce tri séquentiel final deviendra long et gourmand en mémoire.

6.3.3 Filtrage dé-duplicateur (*Distinct Filter*)

Si besoin le patron de *dé-duplication* peut tout d'abord réaliser un filtrage. Mais il va surtout garantir que tous les enregistrements seront uniques : les doublons seront éliminés, et chaque enregistrement ne sera présent qu'une seule fois. La présence de doublons est fréquente dans les masses de données, et des opérations de simplification (comme la suppression de certains attributs) peuvent créer de nouveaux doublons. Bien sur, dans certaines applications il peut être nécessaire de conserver et de compter les réplications, mais elles sont souvent inutiles. Par exemple, si on cherche des informations sur les profils des internautes ayant accédé à un site web, il est pratique de pouvoir retrouver l'ensemble des traces d'un même internaute et de ne le comptabiliser qu'une

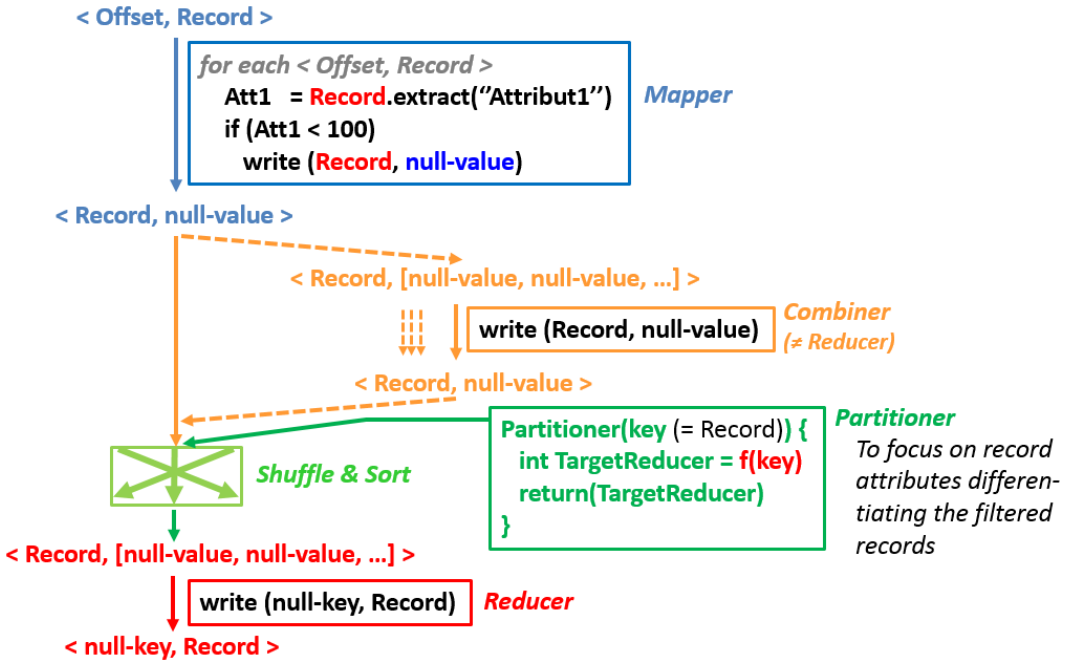

FIGURE 6.11 – Patron *Map-Reduce* de filtrage et de dé-duplication

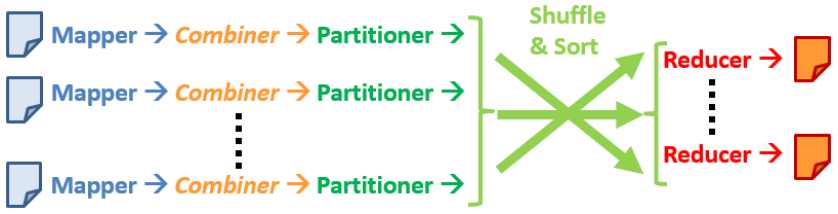

FIGURE 6.12 – Schéma de déploiement du patron de filtrage et de dé-duplication

seule fois. La requête SQL la plus proche serait :

SELECT DISTINCT * FROM table

La figure 6.11 montre les enchaînements d'opérations permettant un filtrage et une dé-duplication dans le paradigme *Map-Reduce*. **L'originalité de ce patron réside dans le fait que les *Mappers* écrivent des paires clé-valeur où les enregistrements sont les clés** (voir le haut de la figure 6.11). Les valeurs peuvent d'ailleurs être nulles (*null-value*). Dès lors, le mécanisme de regroupement des paires clé-valeur du *Map-Reduce* va réaliser automatiquement la dé-duplication. Les *Reducers* recevront des paires où la clé sera un enregistrement unique et où la valeur sera une liste de *null-value*. Ils pourront alors écrire dans des fichiers de sorties des paires clé-valeur où les enregistrements seront redevenus les valeurs (voir le bas de la figure 6.11). Les *Reducers* feront donc beaucoup d'écritures sur disques et peu de traitements.

Etudions maintenant les diverses optimisations possibles au sein de ce patron :

– Un grand volume de données va produire un grand nombre d'enregistrements différents et donc un grand nombre de clés différentes en entrée des *Reducers*. Par conséquent, il semble logique de déployer un nombre important de *Reducers* pour répartir la charge d'écriture des enregistrements dans les fichiers de sorties.

– Un *Combiner* peut supprimer des doublons dès la sortie de chaque *Mapper*, et réduire ainsi

significativement le traffic du *Shuffle & Sort* s'il y a beaucoup de doublons. Cependant, si très peu de doublons existent, le *Combiner* ne servira à rien.

– Un *Partitioner* peut aussi avoir son utilité car les clés sont des enregistrements complexes qui peuvent être volumineux et ne se différencier que par quelques attributs. Il faut donc étudier la nature des enregistrements pour décider si un *Partitioner* spécifique peut mener à une meilleure distribution des paires sur les différents *Reducers*.

Le schéma générique de déploiement de ce patron de dé-duplication est donc celui de la figure 6.12. Le *Combiner* et le *Partitioner* sont laissés à la discrétion du développeur selon les caractéristiques des données.

6.4 Patrons de restructuration des données

6.4.1 Transformation de structures quelconques vers une structure hiérarchique

Le cas d'utilisation classique de ce patron consiste à lire des données dans un format structuré mais non hiérarchique, par exemple dans plusieurs tables d'une BdD Relationnelle, et à les transformer en documents structurés hiérarchiques, par exemple au format XML ou JSON, pour une entrée en BdD *NoSQL* orientée documents comme MongoDB (voir section 9.3).

On verra au chapitre 8 que les BdD NoSQL *ne sont pas construites pour faciliter des jointures lors de leur interrogation, car leur aspect distribuée et large échelle rendrait les jointures coûteuses. L'approche retenue consiste à réaliser la jointure lors de l'écriture des documents, c'est-à-dire à stocker des documents structurés contenant toutes les informations accédées habituellement, et évitant ainsi de calculer des jointures à chaque lecture. Evidemment, si on interroge la BdD* NoSQL *pour faire d'autres recoupements d'informations, alors il faut accéder à de nombreux documents et les croiser par des algorithmes de* Map-Reduce*. L'organisation des données stockées dans une BdD* NoSQL *est donc critique pour permettre une interrogation efficace "la plupart du temps".*

Pour créer une structure hiérarchique adaptée aux interrogations les plus fréquentes on commence par identifier les tables d'une BdD relationnelle qui sont souvent associées à travers une jointure, puis par identifier l'attribut qui sert à exprimer la condition de la jointure. Par exemple, considérons des publications déposées sur un site d'archivage de travaux de recherche, où chaque publication peut faire l'objet de commentaires de la part de lecteurs (comme sur le portail *Researchgate*).

– Dans une BdD relationnelle les meta-données des publications (comme les titres, auteurs, nombre de pages et *Digital Object Identifier* (DOI)) sont stockées dans une table, alors que les informations sur les commentaires (textes, auteurs et dates des commentaires, ainsi que les DOI des publications visées) sont typiquement stockées dans une autre table relationnelle. Le DOI est un numéro d'identification unique d'une publication, qui peut exprimer naturellement la condition de jointure entre ces deux tables, et permettre ainsi de retrouver les commentaires qui vont avec une publication donnée.

– Un stockage en BdD *NoSQL* orientée document se fera au contraire en regroupant dans un même document hiérarchique les meta-données sur une publication et les commentaires émis sur cette publications. Le DOI devient alors naturellement la valeur à la racine du document hiérarchique stocké dans la BdD *NoSQL*.

Le patron *Structured-to-Hierarchical* facilite ces transformations de plusieurs structures quelconques vers une seule structure hiérarchique. Il s'appuie notamment sur la capacité d'*Hadoop*

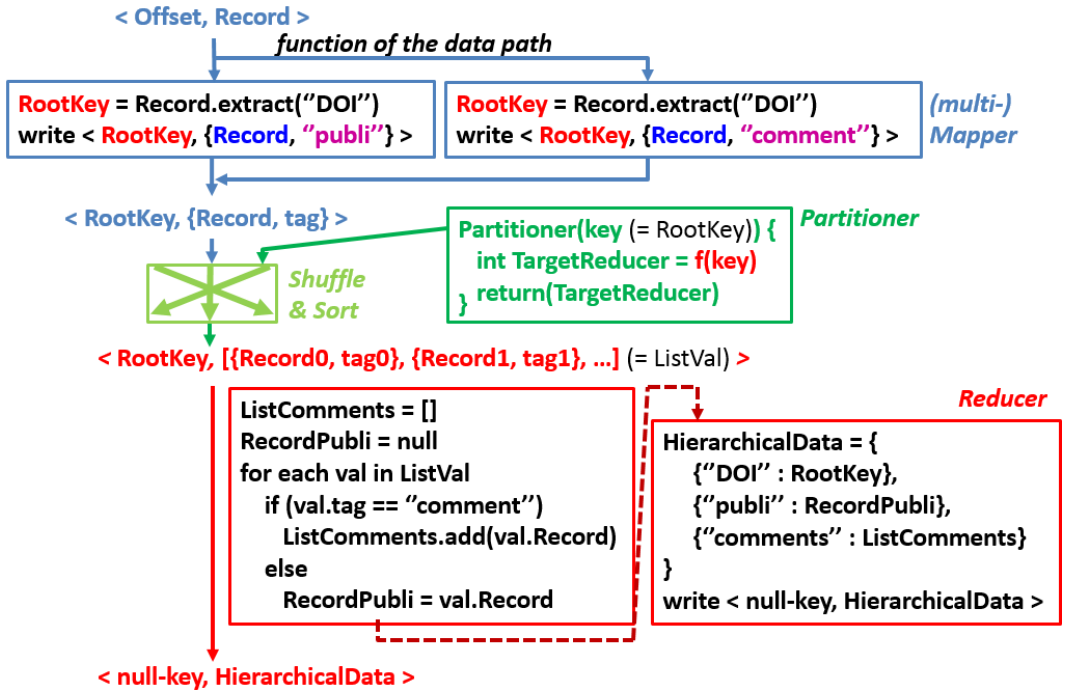

FIGURE 6.13 – Patron *Map-Reduce* de restructuration de données *Structured-to-Hierarchical*

d'utiliser plusieurs types de *Mappers* dans une même opération de *Map-Reduce*, en s'aiguillant sur le chemin d'accès aux données : un fichier de données sera traité par une fonction *map()* ou une autre selon son *path*, comme illustré sur la figure 6.14. Il est aussi possible que les données structurées initiales contiennent un entête permettant de connaître leur type et d'utiliser la bonne fonction *map()*. Dans tous les cas, le *Mapper* recherchera dans la donnée structurée l'attribut destiné à devenir la valeur racine de la future donnée hiérarchique, et générera une paire clé-valeur avec une valeur composite (voir le haut de la figure 6.13) :

- La clé sera constituée de la valeur racine. Dans notre exemple il s'agirait du DOI de la publication.
- La valeur composite sera constituée de la donnée structurée initiale (éventuellement modifiée et enrichie), et d'un identifiant du type de cette donnée en fonction de son *path*. Dans notre exemple ce pourrait être : *{un-enregistrement-publication, "publi"}*, ou bien : *{un-enregistrement-commentaire, "comment"}*.

Chaque *Reducer* reçoit ensuite toutes les paires clé-valeur ayant comme clé la même valeur racine et comme valeur la liste de toutes les données structurées liées à cette valeur racine (voir le bas de la figure 6.13). Dans notre exemple, il s'agirait d'une publication et de tous les commentaires associés, tous étiquetés avec le DOI de la publication. Chaque *Reducer* peut alors construire une donnée hiérarchique rassemblant toutes ces informations. La figure 6.13 illustre cet enchaînement de formats clé-valeur, avec la construction d'une donnée hiérarchique inspirée d'un format JSON (voir la section 8.4).

Les optimisations possibles de ce patron en terme de nombre de *Reducers*, de *Combiner* et de *Partitioner* sont les suivantes :

- La quantité de données ré-écrites au final dans les données hiérarchiques sera à peu près la même que la quantité de données initiales, car il n'y a pas de raison de perdre de l'information dans la transformation. Un grand nombre de *Reducers* sont souhaitables pour répartir

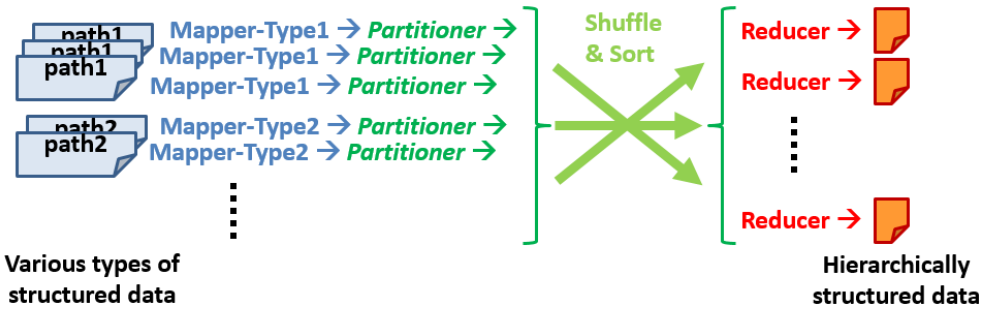

FIGURE 6.14 – Déploiement du patron de restructuration *Structured-to-Hierarchical*

sur de nombreuses machines la charge d'écriture des données hiérarchiques.

– Le développement d'un *Combiner* est inutile dans ce patron, car il n'aurait connaissance que d'une partie des données associées à une même valeur racine (celles traitées par son *Mapper*) et ne pourra pas construire les données hiérarchiques finales. De plus, même si dans certains cas il pouvait commencer à construite ces données, cela ne réduirait pas le volume des données routées vers les *Reducers*.

– En revanche, on peut trouver un intérêt à concevoir un *Partitioner* optimisant le répartition de charge entre les *Reducers*, si l'on a des connaissances sur la variété des valeurs racines et la distribution des volumes de données associées à ces racines.

La figure 6.14 illustre le déploiement du patron de restructuration *Structured-to-Hierarchical*. Le nombre de *Reducers* de ce patron peut donc être très important, mais l'implantation d'un *Partitioner* est laissé à la discrétion du développeur, fonction de sa connaissance des *RootKeys* et des volumes de données associées.

6.4.2 Partitionnement et *binning* d'ensembles de données

L'objectif est ici de partionner un ensemble initial de données selon un critère et une partition prédéfinis. Par exemple, on peut partitioner des *logs* selon l'année de leur apparition. Ce patron propose une solution utilisant essentiellement les *Mappers* et le *Partitioner*. Les *Reducers* sont indispensables pour écrire les résultats sur disques mais ont une action proche de l'identité.

Première solution

La figure 6.15 illustre le fonctionnement de ce patron sur un cas de partitionnement d'un grand ensemble de *logs*. Ils ont été collectés pendant un nombre connu d'années (n), et doivent être partitionnés selon leur année d'apparition, donc en n sous-ensembles.

Chaque *Mapper* traite un ensemble de *logs*, et extrait pour chaque *log* son année d'apparition. Puis, il génére simplement une nouvelle paire clé-valeur avec l'attribut de partitionnement en clé, et la donnée initiale en valeur (voir le haut de la figure 6.15). Le *Partitioner* est le véritable cœur de ce patron. Il est en charge de désigner le *Reducer* vers lequel orienter chaque paire clé-valeur, c'est donc lui qui réalise réellement le partitionnement. La figure 6.15 décrit le cas le plus simple où il y a autant de *Reducers* que de sous-ensembles, et où le *Partitioner* va juste comparer la clé (l'attribut de partitionnement) aux différents seuils de la partition pour identifier l'indice du sous-ensemble et donc le *Reducer* adapté. Ainsi, non seulement on aura bien un sous-ensemble par *Reducer*, mais le sous-ensemble de données i sera géré par le *Reducer* i et sera donc stocké dans le fichier de sortie i (on saura ainsi facilement retrouver les données après partitionnement).

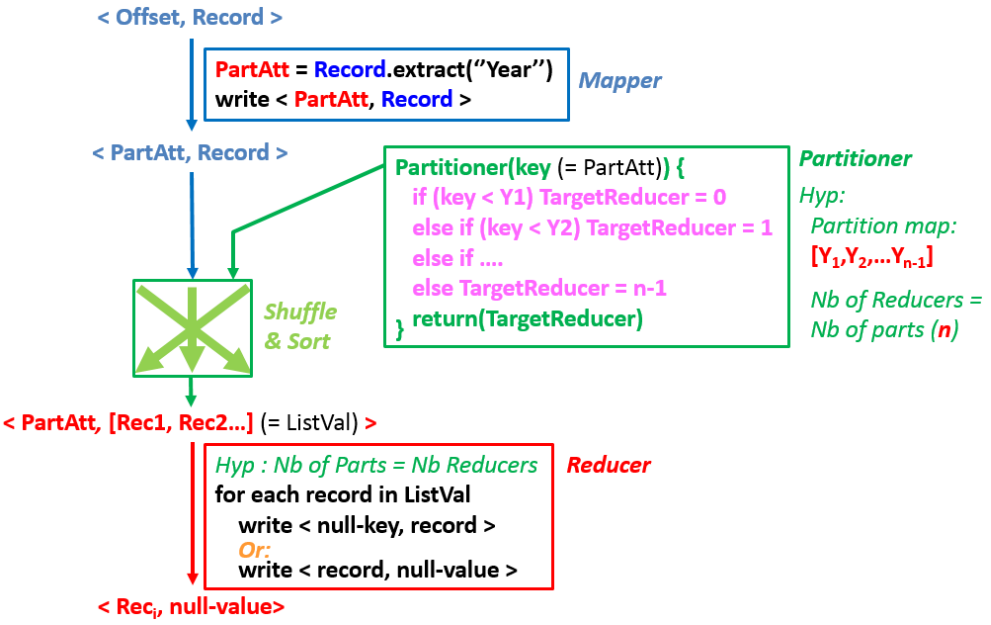

FIGURE 6.15 – Patron *Map-Reduce* de partitionnement par année d'un ensemble de logs enregistrés pendant un nombre connu d'années

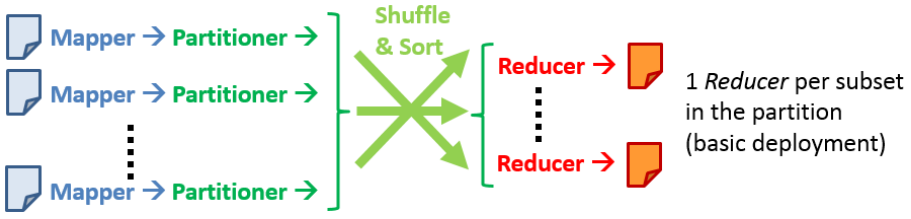

FIGURE 6.16 – Déploiement du patron de partitionnement d'un ensemble de données

Les *Reducers* se contentent ensuite d'écrire dans des fichiers différents les données initiales qui leurs parviennent, à raison d'un fichier par *Reducer* donc par sous-ensemble. Chaque *Reducer* écrit donc un ensemble de paires clé-valeur : une par enregistrement reçu (voir le bas de la figure 6.15). Les paires de sorties peuvent avoir une clé nulle ou une valeur nulle.

Evidemment, si l'on sait qu'un sous-ensemble va être beaucoup plus volumineux que les autres, il est possible de prévoir plusieurs *Reducers* pour lui afin d'équilibrer la charge d'écriture des données sur disque. On doit alors implanter un *Partitioner* qui répartisse les paires de ce sous-ensemble sur plusieurs *Reducers*, par exemple de manière aléatoire ou en *Round Robin*.

En revanche, il n'est pas possible d'implanter un *Combiner*, car il n'y a pas de réduction possible des données : toute donnée initiale doit être routée vers un *Reducer* et ré-écrite sur disque. Il n'y a donc pas de réduction de trafic possible dans le *Shuffle & Sort*, et il n'y a pas non plus de traitement à alléger dans les *Reducers*.

Au final, le déploiement de ce patron est assez simple, comme le montre la figure 6.16.

Deuxième solution

La figure 6.17 décrit une solution où les *Mappers* font l'extraction de l'attribut de l'enregistrement servant de critère de partitionnement, comme précédemment, mais en calculent aussi le sous-ensemble de la partition dans lequel se range l'enregistrement. Les paires clé-valeur de sor-

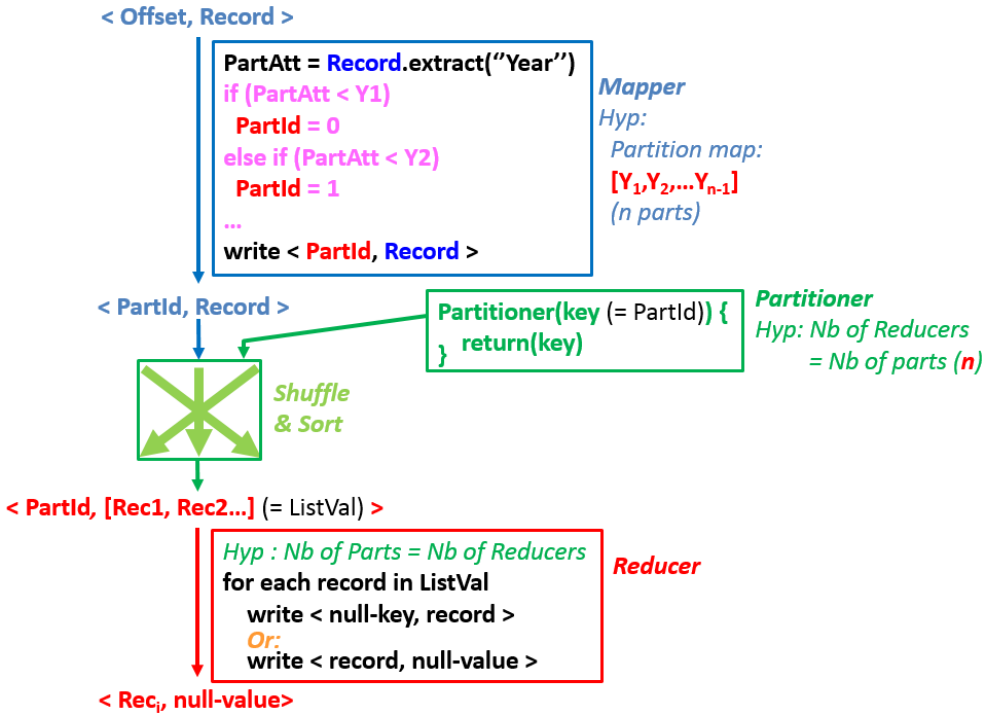

FIGURE 6.17 – Version du partitionnement par année d'un ensemble de logs avec la répartition en sous-ensembles faite dans les *Mappers*

tie des *Mappers* ont donc directement comme clé des identificateurs de sous-ensembles (de 0 à $n-1$) au lieu d'attributs d'enregistrements. Dans notre exemple de la figure 6.17, les *Mappers* font maintenant un travail que faisaient le *Partitioner* dans la première solution, et le *Partitioner* est beaucoup plus simple et ne fait plus qu'une traduction souvent immédiate d'un identificateur de sous-ensemble vers un numéro de *Reducer*. Le déploiement de cette deuxième solution reste en revanche inchangé (voir figure 6.16).

Variante avec le patron de *binning* (*mise en récipients*)

Une variante de ce patron de partitionnement est celui de *binning* (ou de *mise en récipients*...). Il répartit toujours des données d'un ensemble en plusieurs sous-ensembles, mais peut mettre une même donnée dans plusieurs sous-ensembles (récipients) à la fois : ce n'est plus une partition stricte. Par exemple, un enregistrement avec un attribut valant exactement une valeur seuil entre deux sous-ensembles, pourrait être classé dans les deux sous-ensembles. La figure 6.18 montre une adaptation du *Mapper* de la deuxième version du patron de partitionnement (celle où les *Mappers* font le partitionnement). Maintenant les *Mappers* peuvent générer plusieurs paires clé-valeur de sorties pour une même paire d'entrée.

Le déploiement du patron de *binning* est encore le même que pour le patron de partitionnement (voir figure 6.16).

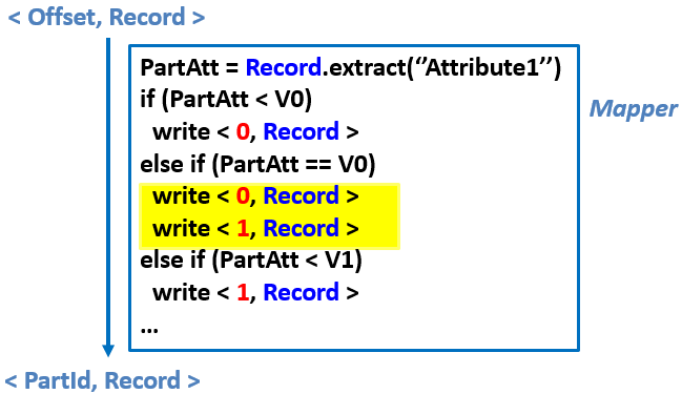

FIGURE 6.18 – Exemple de *Mapper* pour le patron de *binning*, où un enregistrement peut être rangé dans plusieurs sous-ensembles

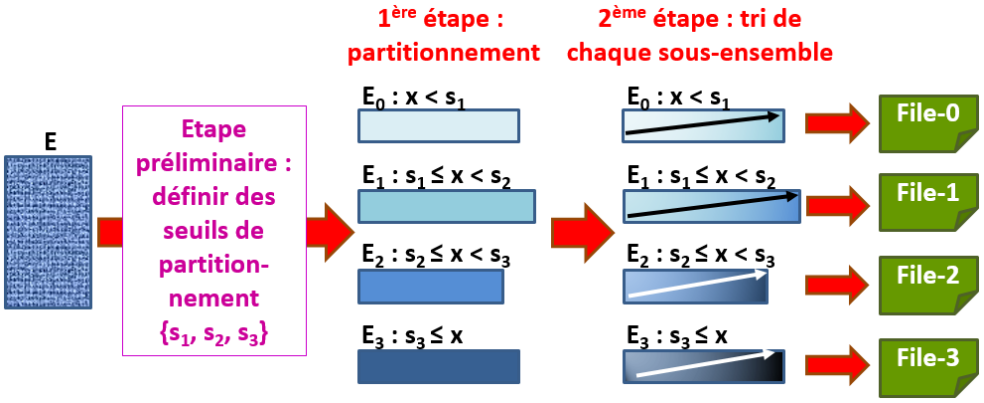

FIGURE 6.19 – Principe d'un tri total dans le paradigme *Map-Reduce*

6.5 Patrons de tris de données

6.5.1 Principes de tri total en *Map-Reduce*

Le paradigme *Map-Reduce* impose un schéma de calcul distribué qui contraint les échanges de données entre tâches, et ce schéma ne permet pas d'implanter les tris parallèles classiques du HPC. Mais une stratégie de tri adapté au paradigme *Map-Reduce* a émergé, en une étape préliminaire et deux étapes de traitements, comme illustré sur la figure 6.19.

L'étape préliminaire consiste à identifier un partitionnement équilibré de l'ensemble E des données d'entrée.

Cela revient à identifier $n_{SE} - 1$ (S_1, S_2, ...$S_{n_{SE}-1}$) seuils pour une répartition des données initiales en n_{SE} sous-ensembles ordonnés de cardinalités à peu près égales. Par exemple, les données insérées dans E_0 seront toutes inférieures au seuil s_1, les données insérées dans E_1 seront toutes supérieures ou égales à s_1 et inférieures à s_2... Obtenir une partition menant à n_{SE} sous-ensembles équilibrés n'est évidemment pas trivial, et nécessite une connaissance a priori sur la distribution des données à trier ou bien un traitement préliminaire (voir section 6.7). Mais si l'on travaille sur de gros ensembles de données, il est probable que leur distribution ne varie pas rapidement de manière significative (comme par exemple la pyramide des ages d'un pays). Cette étape préliminaire n'est donc pas à refaire systématiquement.

La première étape de traitement consiste à partitioner les données d'entrée en suivant le parti-

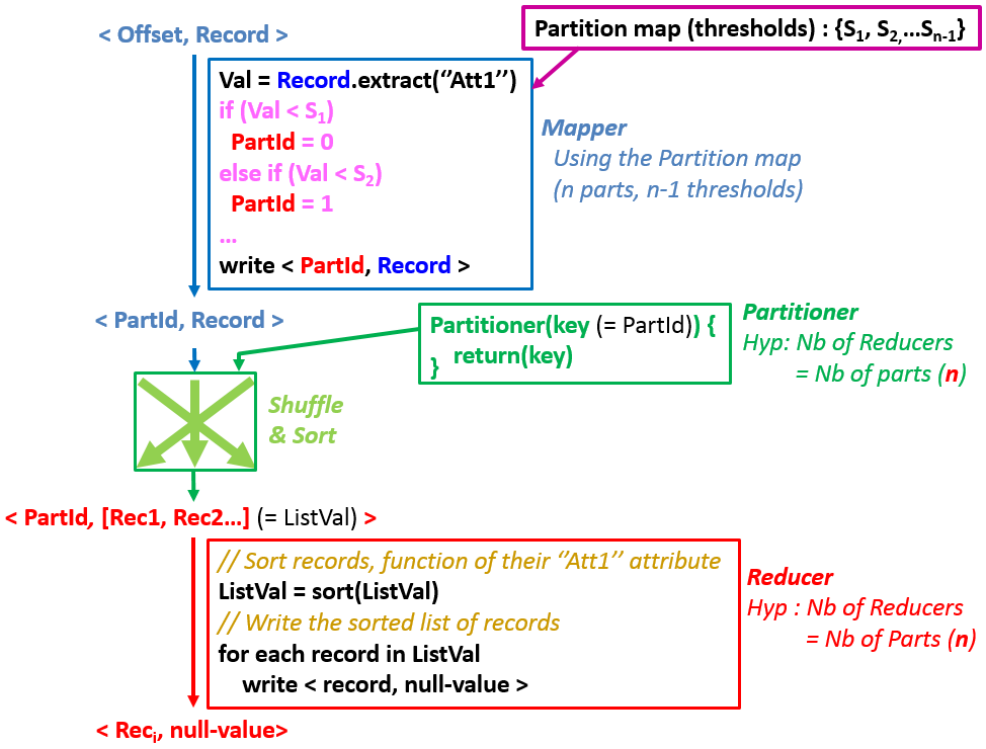

FIGURE 6.20 – Patron de tri *Map-Reduce* avec tri final dans les *Reducers*

tionnement identifié à l'étape préliminaire.

Cette étape incombe surtout aux *Mappers* et au *Partitioner*, comme étudié dans le patron de partitionnement (voir la section 6.4.2). Cependant, nous souhaitons au final trier nos données. Il est donc important que le sous-ensemble E_i soit envoyé vers le *Reducer* de rang i, afin de se retrouver finalement dans le fichier de rang i. Il nous faut donc :

1. déployer strictement autant de *Reducers* qu'il y a de sous-ensembles dans la partition,
2. maîtriser l'affectation des clés sur les *Reducers* (clé du sous-ensemble i envoyée vers le *Reducer* i).

Or, ce dernier point n'est pas possible avec le *Partitioner* par défaut, on ne saurait pas dans quel fichier serait finalement stocké le sous-ensemble de données i. Une ré-écriture du *Partitioner* est donc nécessaire, en plus du développement des *Mappers*.

La deuxième étape de traitement consiste à ordonner chaque sous-ensemble de la partition.

Cette seconde étape concerne a priori les *Reducers*, qui reçoivent un ou plusieurs sous-ensembles en intégralité et peuvent réordonner chacun d'eux localement. Si les ensembles traités sont volumineux, il serait préférable de n'avoir qu'une seule tâche *Reducer* par nœud, de manière à ce qu'une machine ait besoin de charger en mémoire et de trier un seul sous-ensemble à la fois. Enfin, comme cela a déjà été évoqué, chaque *Reducer* stockera finalement son sous-ensemble trié dans un fichier, et le nom du fichier sera indexé avec le rang du *Reducer*. Grace à la réécriture du *Partitioner* intervenu lors de la première étape, le rang du *Reducer* sera aussi celui du sous-ensemble traité (voir la partie droite de la figure 6.19).

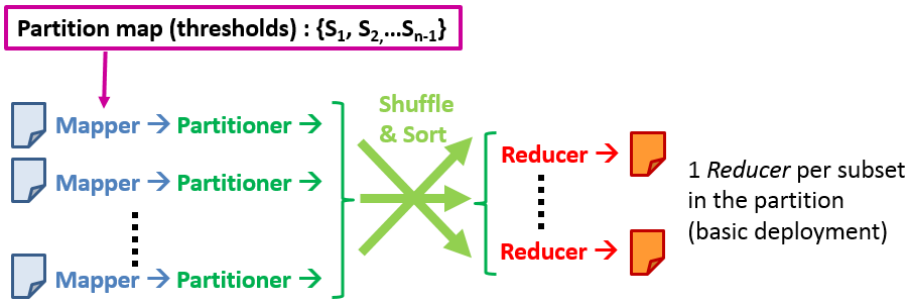

FIGURE 6.21 – Déploiement du tri *Map-Reduce* avec tri final dans les *Reducers*

6.5.2 Tri total avec tri final dans les *Reducers*

Le patron de tri décrit dans cette section implante scrupuleusement le principe général décrit à la section 6.5.1. Pour cela, il reprend le patron de partitionnement décrit dans la section 6.4.2 (deuxième solution) et l'enrichit d'un tri local dans les *Reducers*.

Algorithmes des *Mappers* et du *Partitioner*

La figure 6.20 montre le schéma du patron de tri complet. On suppose pour l'instant que l'on dispose d'une carte de partitionnement (en haut à droite de la figure 6.20), composé de seuils de passages d'un sous-ensemble à un autre. Les *Mappers* extraient les valeurs de l'attribut servant à trier les enregistrements (*Att1* dans l'exemple de la figure 6.20). Puis, connaissant la carte de partitionnement ils en déduisent l'identificateur du sous-ensemble auquel appartient chaque enregistrement. Ils génèrent alors des paires dans lesquelles la clé est l'identificateur d'un sous-ensemble de la partition, et la valeur est un enregistrement destiné à ce sous-ensemble. Dans notre exemple, un sous-ensemble est entièrement identifié par son indice (commençant à partir de 0), et les *Mappers* produisent des clés < *PartId, Record* >.

Le *Partitioner* a été ré-écrit pour aiguiller les paires du premier sous-ensemble (de clé 0) sur le premier *Reducer* (de rang 0), et finalement aiguiller les paires du sous-ensemble i sur le *Reducer* i (voir le pseudo-code sur la figure 6.20).

Algorithme des *Reducers*

Chaque *Reducer* reçoit des paires *clé - liste de valeurs*, dont chacune rassemble toutes les valeurs appartenant à un même sous-ensemble. Dans notre cas, un *Reducer* reçoit des paires $< PartId, [Rec_1, Rec_2...] >$ et trie localement ces listes d'enregistrements :

ListVal = sort(ListVal), sur l'exemple de la figure 6.20.

Ce tri se base évidemment sur l'attribut servant à trier les enregistrement, comme dans les *Mappers* (*Att1* dans l'exemple de la figure 6.20). Chaque *Reducer* écrit ensuite dans son fichier de sortie une succession ordonnée de paires $< PartId, Rec' >$ ou de paires $< Rec', null - value >$, selon les besoins de la suite de l'application (voir le bas de la figure 6.20).

Déploiement et optimisations

La figure 6.21 résume le déploiement de ce patron sur plusieurs nœuds de données et de traitements. Comme dans le patron de partitionnement, il n'est pas utile de créer des *Combiners* car on ne peut pas diminuer le volume de données routées : toutes les données doivent bien être acheminées vers les *Reducers*.

Au final, ce patron de tri fonctionne bien mais **il n'exploite pas toutes les possibilités du tri de clés déjà implanté dans le *Shuffle & Sort***, que nous allons étudier dans la suite de ce chapitre.

6.5.3 Reconfiguration du *Shuffle & Sort* pour optimisation

Trier des données en suivant le paradigme *Map-Reduce* peut se faire en tirant partie de certaines fonctionnalités de l'étape *Shuffle & Sort*. En fait, on peut distinguer 3 mécanismes reconfigurables dans le *Shuffle & Sort* d'*Hadoop*, correspondant à 3 sous-étapes : le *Partitioner*, le *keyComparator* et le *groupComparator*. Ces mécanismes sont facilement adaptables aux spécificités du problème traité en redéfinissant certaines classes ou fonctions Java.

Utilisation de clés composites

Les clés composites sont des clés structurées qui contiennent plusieurs attributs. Leur utilisation **permet d'augmenter la quantité et la variété des informations portées par une clé**, et de faire fonctionner plus finement la chaîne *Map-Reduce* d'*Hadoop*. Mais il est nécessaire que chaque mécanisme de chaque sous-étape du *Shuffle & Sort* n'exploite que les parties de la clé qui le concernent. La reconfiguration des 3 mécanismes du *Shuffle & Sort* permet cette sélectivité, et permet d'exploiter pleinement la richesse des *clés composites*.

Reconfiguration du *Partitioner*

Ce composant permet de **redéfinir l'affectation des clés de sorties des *Mappers* vers les *Reducers***, en calculant le numéro du *Reducer* vers lequel router une paire clé-valeur. Nous avons déjà suggéré de le modifier dans plusieurs patrons précédents. Notons que le calcul du numéro du *Reducer* cible peut se faire en utilisant tout ou partie de la clé s'il s'agit d'une clé composite.

Techniquement, en *Hadoop* on doit définir une classe fille de partitionnement :

```
public static class MyPartitioner
  extends Partitioner<keyClass,valueClass> {
  ...
  public int getPartition(keyClass key,
                          valueClass val,
                          int numPartitions) {
    ...
    // Must return a Reducer index [0 ... NbReducers-1]
    int index = ...
    return(index)
  }
}
```

et imposer cette classe comme nouveau partitionneur avant de lancer le *job* de *Map-Reduce* :

```
job.setPartitionerClass{MyPartitioner.class}
```

Reconfiguration du *keyComparator*

Ce composant permet de **modifier l'ordre des clés présentées à un même *Reducer***, en définissant une nouvelle fonction de comparaison de 2 clés.

Les clés envoyées à un même *Reducer* depuis divers *Mappers* sont ordonnées entre la sortie du *Shuffle & Sort* et le début du *Reducer*. Mais l'opération de comparaison des clés peut être redéfinie afin de changer l'ordre dans lequel les paires clé-valeur sont finalement présentées au *Reducer*. Ces paires seront ensuite regroupées en paires *clé - liste de valeurs* (voir le *groupComparator* dans

la section suivante), en respectant cet ordre. Donc, le *keyComparator* permet de contrôler l'ordre de présentation des paires *clé - liste de valeurs* en collaboration avec le *groupComparator*, et aussi l'ordre des valeurs à l'intérieur des *listes de valeurs*.

En cas d'utilisation de clés composites, la nouvelle fonction de comparaison des clés peut utiliser tout ou partie des clés, et peut ne pas utiliser les mêmes parties que le *Partitioner*.

La démarche technique en *Hadoop* et similaire à la précédente. On doit définir une classe fille de comparaison de clés :

```
public static class MyKeyComparator
  extends WritableComparator {
  ...
  public int compare(WritableComparable w1,
                     WritableComparable w2) {
    ...
    // Must return -1/0/+1,
    // considering w1 < w2, or w1 = w2, or w1 > w2
    int ComparisonResult = ...
    return(ComparisonResult)
  }
}
```

et imposer cette classe pour le tri des clés, avant de lancer le *job* de *Map-Reduce* :

```
job.setSortComparatorClass{MyKeyComparator.class}
```

Reconfiguration du *groupComparator*

Ce composant permet de **modifier la constitution des paires *clé - liste de valeurs* fournies en entrée de la fonction *reduce()***. Les paires clé-valeur d'un *Reducer* sont regroupées quand elles ont des clés considérées identiques, pour former des paires *clé-liste de valeurs*. Mais on peut contrôler ce regroupement en modifiant la fonction de comparaison des clés qui est utilisée : on peut ainsi décider sur quels critères deux clés seront considérés identiques. En revanche, l'ordre des valeurs de la *liste de valeurs* respectera celui des paires clé-valeur obtenu par le *keyComparator* à l'étape précédente.

Notons que la fonction de comparaison de clés de l'étape de regroupement des paires clé-valeur est distincte de celle de l'étape d'ordonnancement de ces paires (étape précédente). On peut donc appliquer des critères et algorithmes de comparaison différents, et notamment utiliser des parties différentes des clés composites.

En *Hadoop* on doit à nouveau définir une classe fille et une fonction de comparaison de 2 clés, dont la signature et le squelette sont très similaire à ceux du *keyComparator* :

```
public static class MyGroupComparator
  extends WritableComparator {
  ...
  public int compare(WritableComparable w1,
                     WritableComparable w2) {
    ...
    // Must return -1/0/+1,
    // considering w1 < w2, or w1 = w2, or w1 > w2
    int ComparisonResult = ...
    return(ComparisonResult)
  }
}
```

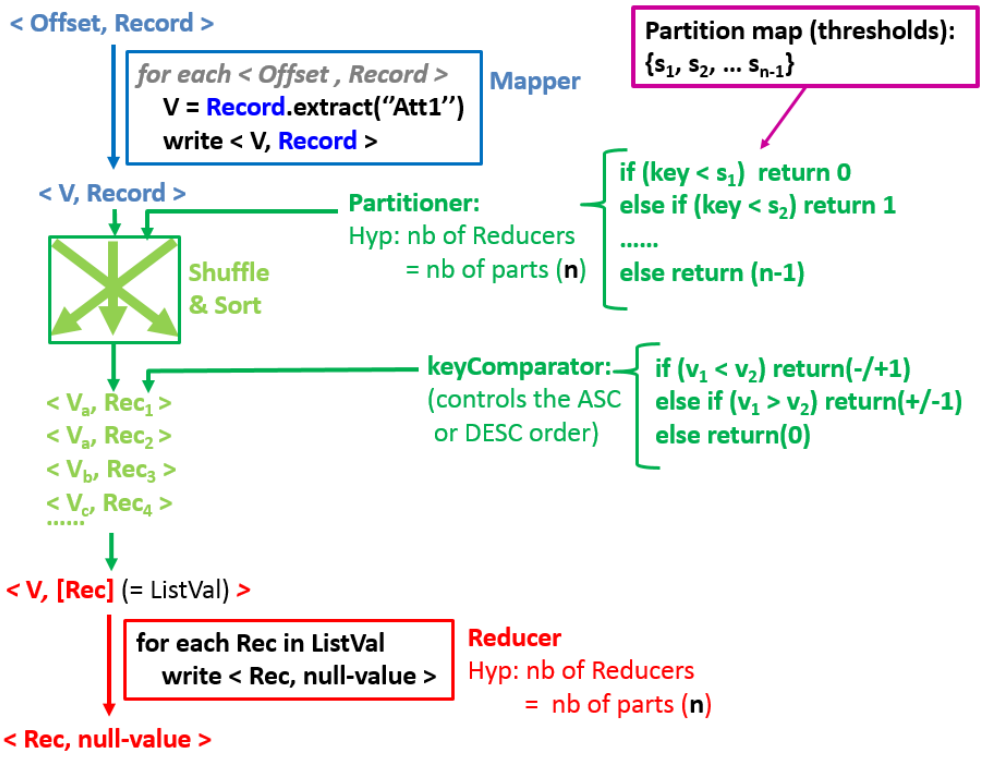

FIGURE 6.22 – Patron de tri *Map-Reduce* optimisé, sur des clés simples et continues

et imposer cette classe pour le regroupement des clés considérées identiques, avant de lancer le *job* de *Map-Reduce* :

```
job.setGroupingComparatorClass{MyGroupComparator.class}
```

Il existe des versions par défaut de ces 3 mécanismes, qui ont été tout à fait adaptées aux besoins des patrons précédents. Cependant, on peut les modifier et les utiliser sur des clés simples ou composites pour réaliser des tris complets profitant des tris de clés du *Shuffle & Sort*, et évitant ainsi de refaire des tris dans les *Reducers*. Nous allons étudier deux solutions de ce type dans les sections suivantes.

6.5.4 Tri optimisé par le *Shuffle & Sort* sur clés simples et continues

Cette version de tri complet dans le paradigme *Map-Reduce* exploite les opérations de partitionnement et de comparaison des clés du *Shuffle & Sort*, afin de ne pas refaire de tris locaux dans les fonctions *reduce()*. Cet algorithme de tri permet donc une réduction du nombre d'opérations en échange d'un éloignement du pur paradigme de programmation *Map-Reduce* (car il nécessite beaucoup plus que de programmer une fonction *map()* et une fonction *reduce()*.

Algorithmes du *Mapper* et du *partitioner*

La figure 6.22 présente le schéma de ce patron de tri. La première partie de ce patron (*Mappers* et *Partitioner*) reprend cette fois-ci la première solution du patron de partitionnement, vu à la section 6.4.2. On y retrouve (en haut à droite) une carte de partitionnement supposée pré-existante, et des *Mappers* qui extraient de chaque enregistrement la valeur de l'attribut à pendre en compte

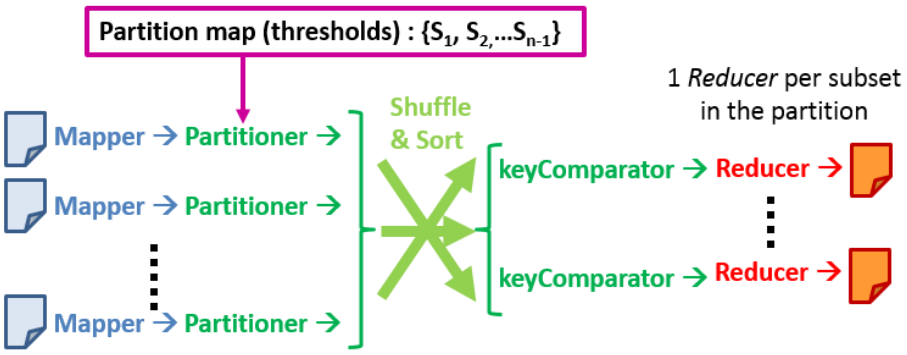

FIGURE 6.23 – Déploiement d'un tri *Map-Reduce* optimisé, sur des clés simples et continues

pour le tri. Cependant, les *Mappers* ne réalisent plus de partitionnement des enregistrements, ils se contentent de générer des paires clé-valeur où la clé est la valeur servant à réaliser le tri, et la valeur de la paire est l'enregistrement associé (voir le haut de la figure 6.22).

C'est ensuite le *Partitioner* qui réalise vraiment le partitionnement : il teste la clé de chaque paire, identifie dans quel sous-ensemble du partitionnement se range l'enregistrement, et en déduit le numéro de *Reducer* auquel la paire sera routée. Le *Partitioner* est donc maintenant le seul composant à devoir connaître la carte de partitionnement. Une caractéristique importante de ce patron est que l'on doit créer exactement autant de *Reducers* qu'il y a de sous-ensembles dans la partition.

Configuration du tri des clés du *Shuffle & Sort*

En sortie du *Shuffle & Sort*, le *keyComparator* (voir section 6.5.3) permet d'ordonner selon leurs clés toutes les paires arrivées sur le *Reducer*. Si l'on suppose que les valeurs utilisées comme clés sont de type standard, et doivent être ordonnées selon un ordre numérique ou lexicographique, alors il n'est pas utile de redéfinir le *keyComparator*. Toutefois, si l'on veut pouvoir contrôler l'ordre ascendant ou descendant du tri, ou classer des données de type non standard, alors il est nécessaire de redéfinir le *keyComparator* (voir son pseudo code sur la figure 6.22).

Dans l'exemple de la figure 6.22 les valeurs servant à ordonner les enregistrements sont supposées être réelles (valeurs *continues*). En conséquence, chaque paire arrivée sur un *Reducer* aura une clé unique, sauf en cas de rares valeurs réelles strictement identiques. Un *Reducer* recevra donc une grande quantité de paires clé-valeur appartenant au même sous-ensemble de la partition et ordonnées selon leur valeur de tri, mais quasiment toutes avec des clés différentes.

Algorithme du *Reducer*

Chaque *Reducer* sera appelé un grand nombre de fois avec une succession ordonnée de paires *clé - liste d'UNE valeur* (et quelques rares fois avec une liste de plusieurs valeurs). Chaque *Reducer* doit alors écrire successivement les valeurs de toutes ces paires dans *le même fichier de sortie*, et dans un format adapté à la suite de l'application. Par exemple, en écrivant des paires < *Rec, null-value* > (voir le bas de la figure 6.22).

Bilan et déploiement

Le déploiement de ce patron de tri utilisant les mécanismes du *Shuffle & Sort*, et évitant un tri local dans chaque *Reducer*, est présenté sur la figure 6.23. Comparé au patron précédent (figure 6.21) il comporte la définition d'un composant de plus (le *keyComparator*) qui évite de refaire des

tris dans les *Reducers*. On note que le nombre de *Reducers* doit toujours être strictement égal au nombre de sous-ensembles de la partition.

Sa faiblesse principale est de ne trier les enregistrements que selon une seule valeur (un seul attribut). Dans la section suivante nous utiliserons des clés composites pour trier des enregistrement selon deux critères emboîtés.

6.5.5 Tri optimisé par le *Shuffle & Sort* sur clés composites

Objectif et clés composites

On a tendance à trier des données structurées (enregistrements) contenant n attributs selon k de ces attributs (avec $k < n$). Par exemple, on classe des étudiants de 3A par ordre lexicographique croissant de leur spécialité de 3A, et pour une spécialité donnée on les trie par moyenne décroissante. Le nom de la spécialité de 3A suivie et la moyenne obtenue sont supposés être des attributs de chaque enregistrement représentant un étudiant. Une requête SQL correspondante pourrait être :

SELECT * FROM etudiant ORDER BY specialite, moyenne DESC

Nous allons réaliser ce tri dans le paradigme *Map-Reduce* en continuant à exploiter le tri implicite des clés présentées aux *Reducers*, et en introduisant des *clés composites*. Une clé composite rassemble plusieurs attributs extraits des données, qui sont souvent des critères de tri. Dans notre exemple, nous fabriquerons des clés composites à partir du nom de la spécialité de 3A et de la moyenne de chaque étudiant.

Après avoir trié les enregistrements selon nos deux critères emboîtés, nous calculerons pour chaque option de 3A la valeur médiane des moyennes des étudiants, et nous enrichirons chaque enregistrement d'étudiant avec un nouvel attribut "*OptionMedian*". Enfin, on écrira les enregistrements ordonnés et enrichis dans des fichiers de sorties.

Algorithmes des *Mappers* et du *Partitioner*

La figure 6.24 illustre l'enchaînement des étapes d'un tri complet avec des clés composites. Ici on considère que deux attributs (k_0 et k_1) sont utiles pour trier des enregistrements, qui contiennent entres autres ces deux attributs. Dans notre exemple, k_0 est un nom de spécialité, donc une chaîne de caractères, et k_1 est une moyenne, donc un *float* ou un *double*, et les *valeurs* à trier sont des données structurées définissant des étudiants. Les *Mappers* vont extraire pour chaque donnée d'entrée les deux attributs clés, et les assembler au sein d'un doublet $\{k_0, k_1\}$ qui constituera une *clé composite* (voir le haut de la figure 6.24). Chaque clé sera associée à son enregistrement initial qui décrit un étudiant (nom, prénom, date de naissance...).

Le *Partitioner* répartira les données selon la carte de partitionnement pré-établie, qui vise à créer q sous-ensembles d'étudiants, à la fois équilibrés au mieux, et où tous les étudiants d'une même spécialité se retrouvent dans le même *Reducer*. Les noms des spécialités sont supposés triés par ordre lexicographique croissant dans la carte de partitionnement, et les plus petits noms seront envoyés sur le *Reducer* de plus petit rang. Le *Partitioner* doit donc être programmé pour ne prendre en compte que le premier attribut de la clé composite au moment de choisir un *Reducer* (voir la figure 6.24).

Configuration du *Shuffle & Sort* et algorithme des *Reducers*

Après la redistribution et le routage des paires de sorties des *Mappers* vers les *Reducers*, ces paires sont ordonnées à l'entrée de chaque *Reducer* en utilisant la fonction de comparaison du

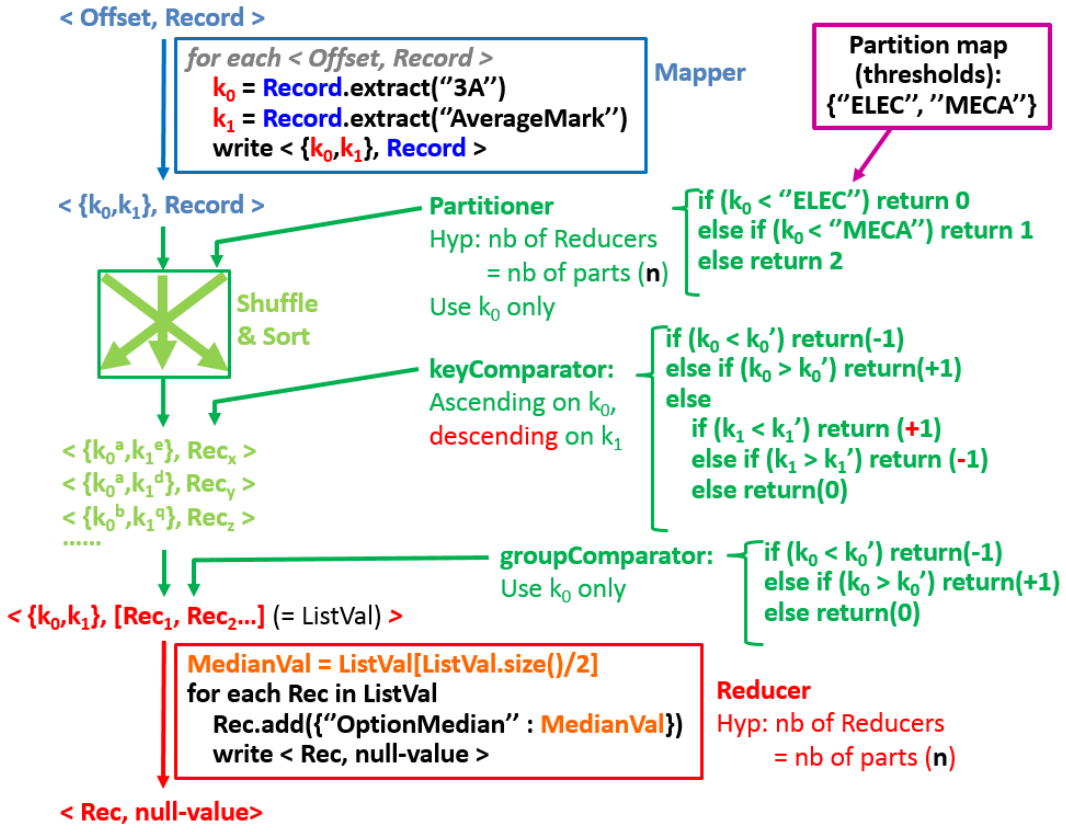

FIGURE 6.24 – Patron de tri *Map-Reduce* optimisé, avec clés composites

keyComparator. Pour répondre à notre objectif, celle-ci doit utiliser les deux attributs de la clé composite, et doit trier les clés dans le sens croissant sur le premier attribut, puis dans le sens décroissant sur le second attribut. Le pseudo-code du *keyComparator* de la figure 6.24 implante ce tri à deux critères emboîtés.

Le *groupComparator* spécifié sur la figure 6.24 teste l'égalité des clés seulement sur leur attribut k_0 (le nom de l'option de 3A). Il va donc provoquer le regroupement dans une paire *clé - liste de valeurs* de tous les enregistrements d'étudiants inscrits dans une même option, tout en gardant l'ordre dans lequel le *keyComparator* les aura classé (en ordre décroissant de leurs moyennes générales).

Pour chaque option qu'il traite, un *Reducer* va donc pouvoir calculer très simplement la valeur médiane de la moyenne générale des étudiants. Il lui suffit de récupérer la moyenne de l'enregistrement situé au milieu de la liste de valeurs reçue en entrée (voir la première ligne du pseudo-code du *Reducer* sur la figure 6.24). Ensuite, le *Reducer* parcourt sa liste de valeurs, et enrichit chaque enregistrement avec la médiane récupérée avant d'écrire le nouvel enregistrement dans un fichier de sortie. Les nouveaux enregistrements se retrouvent bien écrits dans l'ordre obtenu par le *keyComparator*.

Exemple de déroulement du tri optimisé

La figure 6.25 montre le déroulement du tri optimisé d'étudiants, selon leur option de 3A (par ordre croissant) puis selon leur moyenne générale (par ordre décroissant). Les 3 mécanismes reconfigurables du *Shuffle & Sort* ont été exploités, et une valeur médiane est extraite et ajoutée aux enregistrements avant écriture ordonnée dans des fichiers. On retrouve évidemment les opérations

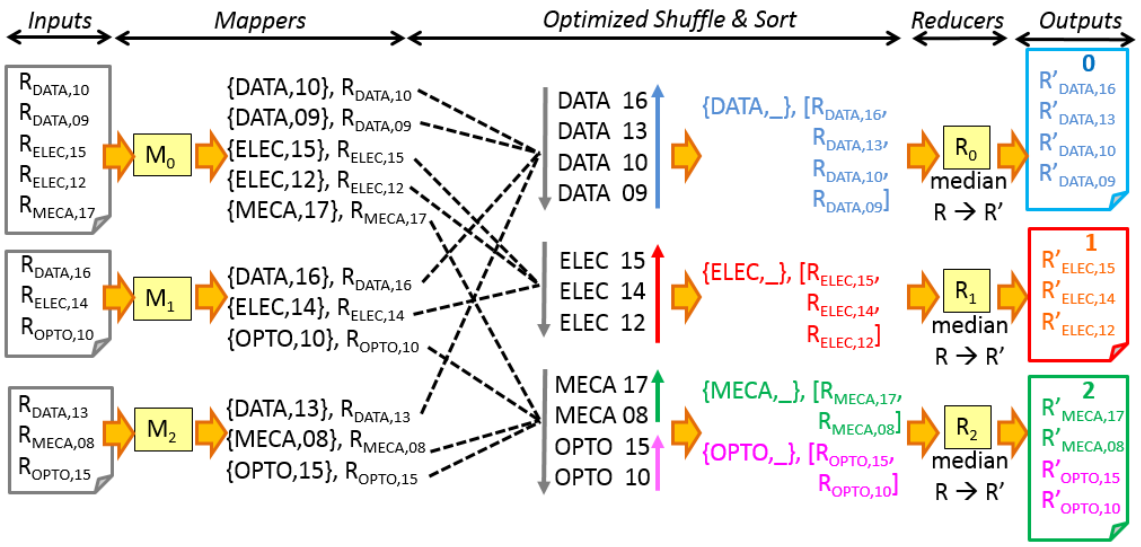

FIGURE 6.25 – Déroulement du tri optimisé par les 3 niveaux de configuration du *Shuffle & Sort*

décrites précédemment dans le patron de tri :

- Chaque *Mappers* lit un fichier d'entrée et extrait de chaque enregistrement un nom d'option de 3A et une moyenne générale. Puis pour chaque enregistrement il génère une paire clé-valeur, avec une clé composite composée des deux valeurs extraites, et avec une valeur constituée de l'enregistrement lu.

- Le *Shuffle & Sort* exécute ses 3 sous-étapes :
  - il partitionne les paires de sorties des *Mappers* selon le premier attribut seulement de la clé composite (le nom de l'option de 3A) et selon une carte de partitionnement pré-établie (action du *Partitioner*),
  - il tri les paires clé-valeur sur chaque *Reducer* selon les deux attributs de leurs clés composites, avec un tri ascendant sur le premier attribut et un tri descendant sur le second (action du *keyComparator*),
  - il regroupe les paires clé-valeur selon le premier attribut seulement de leur clé composite pour créer des paires *clé - liste de valeurs*, qui regroupent tous les étudiants d'une même option et conservent l'ordre obtenu à la sous-étape précédente (action du *groupComparator*).

- Chaque *Reducer* reçoit une paire *clé composite - liste ordonnée d'enregistrements* par option de 3A dont il est en charge, en extrait la valeur médiane de la moyenne générale des étudiants de l'option, enrichit l'enregistrement de chaque étudiant avec cette médiane ($R \rightarrow R'$), et écrit sur disque la liste ordonnée des enregistrements enrichis.

Bilan et déploiement

Les clés composites sont la solution classique pour appliquer sur des données structurées plusieurs critères de tris emboîtés. Il est cependant nécessaire de réécrire les fonctions de partitionnement, de comparaison de clés et éventuellement de comparaison de groupes pour tirer pleinement partie des mécanismes de tris implicites du *Shuffle & Sort* d'*Hadoop*. On évite ainsi de répéter ces opérations de tri dans un tri local à chaque *Reducer*.

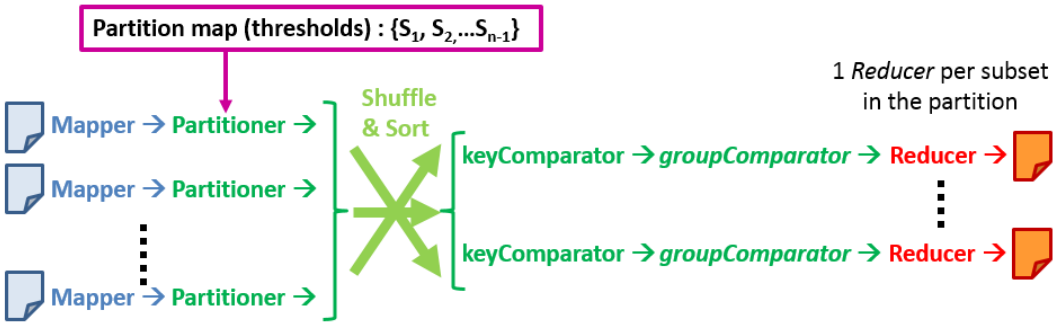

FIGURE 6.26 – Déploiement d'un tri complet en *Map-Reduce* optimisé par clés composites et par tous les mécanismes du *Shuffle & Sort*

La figure 6.26 résume le déploiement de ce patron de tri. Tous les composants de la chaîne *Map-Reduce* d'*Hadoop* y sont présents. Cet exemple de tri est le plus complet que nous avons étudié jusqu'ici.

De nombreux patrons *Map-Reduce* réalisent un partitionnement, et supposent qu'une carte de partitionnement a été pré-établie. Toutefois, sans connaissance a priori sur les données ni expérience préalable sur un jeu de données similaire, il est difficile d'établir une carte pertinente. Heureusement, *Hadoop* nous propose quelques outils d'*échantillonage* pour nous aider, voir la section 6.7.

6.6 Patrons de jointure

Pouvoir réaliser une jointure en *Map-Reduce* est très important, car les BdD *NoSQL* ont souvent besoin de faire des *Join* grace à une couche externe, et notamment en *Map-Reduce*. Le coût de l'opération reste cependant élevé.

6.6.1 Equi-jointure générique

La figure 6.27 résume l'enchaînement des opérations d'une équi-jointure en *Map-Reduce*. La première partie (*Mapper* et *Partitioner*) ressemble au patron de restructuration de données structurées en données structurées hiérarchiques (voir section 6.4.1). La deuxième partie (*Reducer*, après le *Shuffle & Sort*) est spécifique au traitement du *Join*.

Chaque *Mapper* lit des blocs de la table A ou de la table B, sur lesquelles on veut réaliser un *Join*. Pour chaque enregistrement lu, un *Mapper* recherche la valeur de l'attribut qui servira de clé commune pour l'équi-jointure entre les deux tables (désigné par *"Att3"* pour *"A"* et *"AttZ"* pour *"B"* sur la figure 6.27). Si besoin, les *Mappers* peuvent pratiquer un filtrage au passage, c'est-à-dire *ignorer* des enregistrements ne satisfaisant pas certains critères. Ce qui revient à augmenter les conditions de la jointure (test sur les valeurs des attributs *"Att1"* et *"AttX"* sur la figure 6.27). Lorsqu'un enregistrement est accepté, son *Mapper* va générer une paire clé-valeur dont la clé sera la valeur de l'attribut d'équi-jointure, et dont la valeur sera un couple d'information. Ce couple comprendra l'enregistrement lui même, et un *tag* codant son type/sa provenance (*"A"* ou *"B"* sur la figure 6.27).

Tous les enregistrements ayant la même valeur d'attribut d'équi-jointure seront donc stockés dans des paires ayant la même clé, qui seront routées vers le même *Reducer* par l'étape de *Shuffle & Sort*. Un *Reducer* verra donc arriver sur lui tous les enregistrements des tables A et B de même clé,

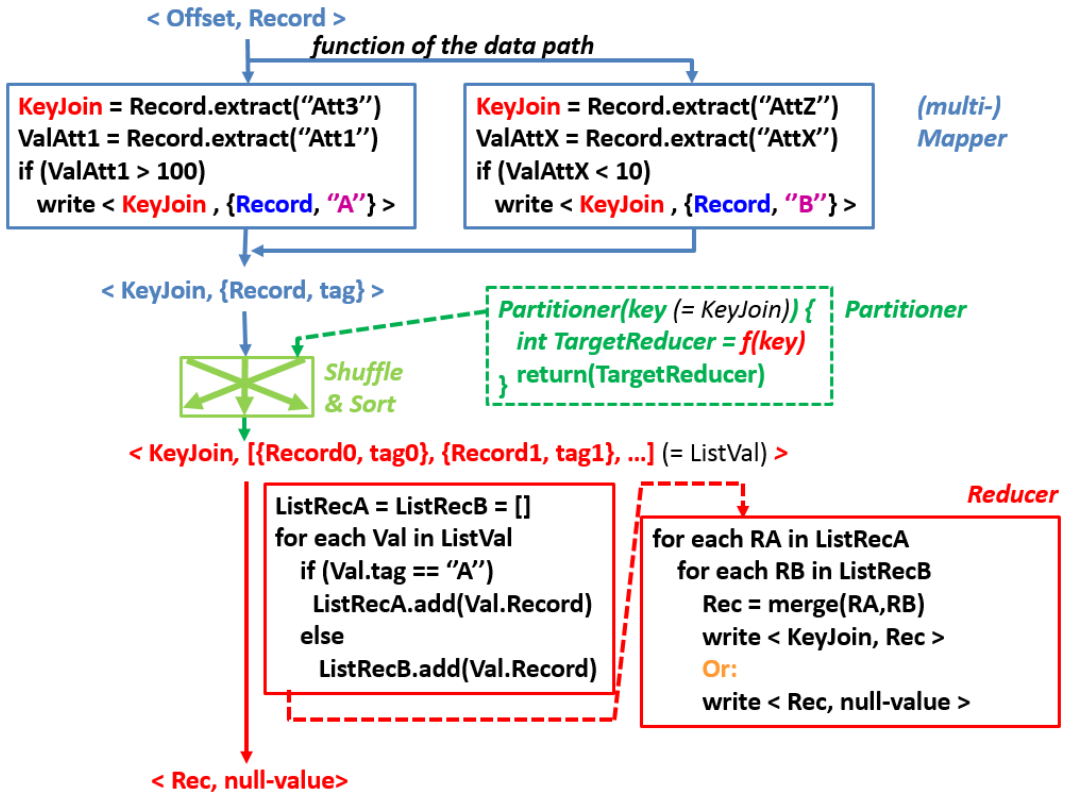

FIGURE 6.27 – Patron de *Join* générique en *Map-Reduce*

avec lesquels il devra construire les lignes de la table de jointure qui sont associées à cette clé. C'est donc le *Reducer* qui *fait le Join*. Le *Reducer* reçoit en fait en entrée une paire *clé - liste de valeurs* et la redécompose en une liste d'enregistrements venant de la table A et une liste d'enregistrements venant de la table B. Ensuite le *Reducer* va constituer des paire clé-valeur composé de la clé reçue, et d'enregistrements venant de la fusion d'un enregistrement de la table A et d'un de la table B (opération *merge(RA,RB)* dans le pseudo-code du *Reducer*). Toutes les combinaisons seront produites afin de générer l'ensemble des lignes de la table d'équi-jointure (voir le pseudo-code du *Reducer* sur la figure 6.27). Il ne restera plus au *Reducer* qu'à écrire dans un fichier de sortie des paires clé-valeur, ou simplement des paires < *enregistrement de jointure, null-value* >, selon les besoins de la suite de l'application (voir le bas de la figure 6.27).

La figure 6.28 illustre ce fonctionnement sur un petit exemple ou une clé se retrouve associée à 3 enregistrements venant de la table A et à 2 enregistrements venant de la table B. Au final, $6 = 3 \times 2$ paires de sorties sont générées par le *Reducer*, avant qu'il ne passe à sa paire *clé - liste de valeurs* suivante.

Il n'y a pas matière à écrire un *Combiner* pour ce patron, car il ne réduira pas le volume routé dans le *Shuffle & Sort*, tous les enregistrements écrit par les *Mappers* devant être routés sans perte vers les *Reducers*. Si l'on possède des informations pertinentes sur la distribution des clés de jointure et les volumes d'enregistrements associés on peut écrire un *Partitioner* spécifique optimisant la répartition des données sur les *Reducers*. Mais ce n'est pas habituel lorsque l'on réalise simplement un *Join* dans une BdD. Le *Partitioner* par défaut pourrait donc être un bon choix.

En revanche, il est important de déployer de nombreux *Reducers* sur de nombreuses machines, car chaque *Reducer* aura un travail non négligeable, et devra construire en mémoire plusieurs listes conséquentes. D'autre part, le volume de données routé est également conséquent, car finalement

Exemple de paire *clé – liste de valeurs* reçue par un *Reducer* :

< KeyJoin = ''Dupont'', [{R_1,''A''}, {R_2,''A''}, {R'_1,''B''}, {R'_2,''B''}, {R_3,''A''}] >

Actions du *Reducer* :

Construction de deux listes d'enregistrements :

→ ListRecA = (R_1, R_2, R_3)
ListRecB = (R'_1, R'_2)

Génération de paires *clé – valeur de jointure*

→ < ''Dupont'', R_1 U R'_1 >
< ''Dupont'', R_1 U R'_2 >
< ''Dupont'', R_2 U R'_1 >
< ''Dupont'', R_2 U R'_2 >
< ''Dupont'', R_3 U R'_1 >
< ''Dupont'', R_3 U R'_2 >

Or : → < R_1 U R'_1, null-value >
< R_1 U R'_2, null-value >
< R_2 U R'_1, null-value >
< R_2 U R'_2, null-value >
< R_3 U R'_1, null-value >
< R_3 U R'_2, null-value >

FIGURE 6.28 – Action d'un *Reducer* du *Join* générique en *Map-Reduce*

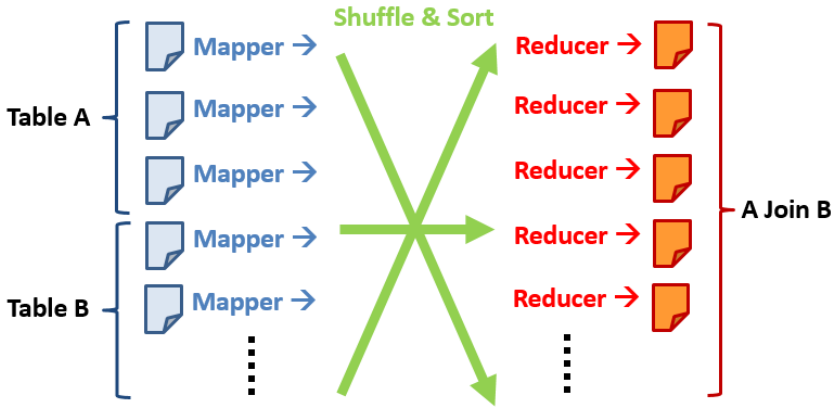

FIGURE 6.29 – Déploiement du patron de Join générique en *Map-Reduce*

les deux tables A et B devront être entièrement routées par le *Shuffle & Sort* (à moins qu'un filtrage sévère soit ajouté dans les *Mappers*). Une jointure est donc une opération qui peut stresser fortement un *cluster Hadoop*, à la fois au niveau des traitements (*Reducers*) et au niveau des communications (*Shuffle & Sort*).

La figure 6.29 résume le schéma de déploiement de ce patron de jointure. Des *Mappers* et des *Reducers* sont implantés et déployés, sans autres compléments ou optimisation de la chaîne *Map-Reduce*. Certains *Mappers* lisent et traitent le fichier distribué de la table A, pendant que d'autres *Mappers* s'occupent du fichier distribué de la table B. Selon le code implanté dans les *Reducers*, on peut réaliser une jointure *interne*, ou *externe*, ou *externe gauche* ou *externe droite*, ce patron de jointure est assez générique pour cela.

6.6.2 Equi-jointure de tables de grande taille

On considère deux grandes tables d'une BdD *NoSQL*, c'est-à-dire stockées sous la forme de deux gros fichiers *HDFS* contenant des données structurées sous forme d'ensembles de *clé-valeur*. On désire faire une jointure selon un attribut commun à chaque table (ex : *UserID*), qui servira de clé d'équi-jointure. La solution générique vue précédemment peut bien sur fonctionner ici aussi.

Mais on cherche une solution plus rapide en considérant qu'un pré-traitement sera effectué préalablement sur les deux tables pour les préparer à une jointure selon une clé fixée. Evidemment, on a intérêt à conserver les tables pré-traitées pour être prêt à faire de nouvelles jointures dans le futur.

Voir exercice 6.9.3.

6.7 Outils d'aide au partitionnement

Réaliser un bon partitionnement des données en un nombre fixé de sous-ensembles est un problème qui apparaît comme un pré-requis à de nombreuses applications *Map-Reduce*, comme les tris (voir section 6.5) et les jointures sur de gros ensembles de données (voir section 6.6.2).

Pour nous aider *Hadoop* fournit des codes de *Partitioners* et de *Samplers* prêts à l'emploi, qui simplifient l'optimisation de la répartition des clés de sorties des *Mappers* vers les différents *Reducers*. Néanmoins, le mécanisme de création d'une carte de partitionnement proposé par *Hadoop* sort du paradigme *Map-Reduce*.

6.7.1 Un partitionneur prédéfini : le *TotalOrderPartitioner*

La classe *TotalOrderPartitioner* est une classe Java qui s'adapte à la configuration du *job Map-Reduce* qui l'utilisera. Cette solution permet de disposer facilement d'un *Partitioner* qui répartit les clés selon une carte de partitionnement prédéfinie. Lors de la configuration du *job Map-Reduce*, on doit notamment spécifier deux informations :

- Le nombre de *Reducers*, car le *Partitioner* aura besoin de cette information : pour n *Reducers* il y aura n sous-ensembles de données, et donc il devra y avoir $n - 1$ seuils de partitionnement dans la carte de partitionnement spécifiée.
- Le chemin d'accès au fichier contenant la carte de partitionnement : on indique ce chemin d'accès au *TotalOrderPartitioner*, qui le stocke au passage dans la configuration du *job Map-Reduce*.

Le code ci-après donne un exemple de configuration d'un *job Map-Reduce* avec le *TotalOrderPartitioner*, puis montre comment configurer ce partitionneur avec un partitionnement prédéfini, avant d'exécuter le *job Map-Reduce*.

```
// Create the 'Map-Reduce job'
Job MRJob = new Job(getConf());

// Configure the 'Map-Reduce job'
// - Set the Mapper and Reducer to use
MRJob.setMapperClass(MyMapper.class)
MRJob.setReducerClass(Myreducer.class)

// - Set the number of Reducers
MRJob.setNumReduceTasks(REDUCE_TASKS);
// - Set the Partitioner to use: 'TotalOrderPartitioner'
MRJob.setPartitionerClass(TotalOrderPartitioner.class);
// - Set the partition map used by the 'TotalOrderPartitioner'
//   (set the partition file path)
TotalOrderPartitioner.setPartitionFile(MRJob.getConfiguration(),
                                       partitionPath);

// Run the Map-Reduce job, using the TotalOrderPartitioner
System.exit(MRJob.waitForCompletion(true) ? 0 : 1);
```

Cette démarche évite donc au développeur applicatif d'avoir à écrire son propre partitionneur. En revanche, il est toujours nécessaire de disposer d'une carte de partitionnement. Les sections suivantes sont consacrées à sa création.

6.7.2 Configuration d'un partitionnement par échantillonage

Dans une application *Map-Reduce* on cherche à éviter qu'un *Reducer* soit amené à stocker une trop grande partie des données. Il pourrait alors constituer une source de ralentissement ou de débordement mémoire, qui empêcherait un *passage à l'échelle*. L'objectif est donc d'établir une carte de partitionnement *équilibrée* des données en n sous-ensembles à destination de n *Reducers*. En l'absence de connaissance a priori sur les données, ou d'expérience sur des jeux de données similaires, on doit étudier la distribution des données d'entrées et le filtrage fait par les *Mappers* pour appréhender la distribution des paires clé-valeur en sortie des *Mappers*. On peut alors en déduire une carte de partitionnement équilibrée.

Mais il faut éviter d'analyser toutes les données d'entrée, surtout avec de grosses volumétries. Cela reviendrait à lancer un traitement *Map-Reduce* complet pour optimiser le partitionnement d'un autre traitement *Map-Reduce*. On se contente en général de *prélever un échantillon représentatif* des données d'entrée, puis de les transformer comme le ferait un *Mapper*, pour analyser ensuite la distribution des paires clé-valeur obtenues.

Hadoop nous fournit des *outils d'échantillonnage* de grands ensembles de données, qui nous permettent de configurer un partitionnement. Toutefois l'utilisation de ces outils sort du strict paradigme *Map-Reduce*, comme on le voit dans la section suivante.

6.7.3 Des échantillonneurs prédéfinis (*Samplers*)

Hadoop met à la disposition de l'utilisateur des classes d'échantillonnage (*Samplers*). On les utilise en exécutant une routine prédéfinie sur le poste client, qui accéde au système de fichier HDFS d'*Hadoop*, réalise un échantillonage des données avec une des classes *Sampler*, et en déduit rapidement une carte de partitionnement équilibrée. Le développeur peut choisir une classe *Sampler* qui échantillonne aléatoirement des données, ou à intervalle régulier, ou encore qui retient les m premières données. Une connaissance sur les données stockées et leur type de distribution permet d'utiliser le *Sampler* les plus adapté et le plus rapide, sinon le *RandomSampler* devrait donner des résultats corrects dans tous les cas.

Néanmoins, il faut commencer par définir en partie le *job Map-Reduce* sur lequel on appliquera par la suite le partitionnement mis au point par le *Sampler*. Ce dernier utilisera la configuration de ce *job* pour y lire des informations utiles à son propre travail, comme les formats des clés et valeurs de sorties des *Mappers*, le nombre de *Reducers* prévus, le fichier où écrire la carte de partitionnement calculée... Enfin, le véritable *job Map-Reduce* sera exécuté avec le *Partitioner* spécifié et la carte de partitionnement établie par le *Sampler*. Le code qui suit illustre ces différentes étapes.

```
// Set the partition file path
Path partitionOutputPath = ...

// Define and start to configure a new Map-Reduce job
Job MRJob = ...
MRJob.setXXXX...
MRJob.setYYYY...
...
// - Set the number of Reducers (required before to run the sampler)
MRJob.setNumReduceTasks(REDUCE_TASKS);
```

```
// - Set the partition output path to use for the Map-Reduce job
//   (do it across a Partitioner configuration)
TotalOrderPartitioner.setPartitionFile(MRJob.getConfiguration(),
                                       partitionOutputPath);

// Choice and configuration of a sampler (RandomSampler)
// - (0.1,10000,10) : 10% of chance to choose a data
//                    10000: max number of choosen data
//                    10: max number of split analyzed
InputSampler.Sampler mySampler =
                  new InputSampler.RandomSampler(0.1,10000,10);

// Run the Sampler that builds and writes the partition map:
// - MRJob is the Map-Reduce job requiring the partition map
// - Its reference allows to get the number of planned Reducers,
//   the input data path, the partition output path...
InputSampler.WritePartitionFile(MRJob, mySampler);

// Complete the Map-Reduce job configuration
MRJob.setPartitionerClass(TotalOrderPartitioner.class);
MRJob.setMapperClass(MyMapper.class);
MRJob.setReducerClass(MyReducer.class);
...

// Run the Map-Reduce job, using the Partitioner that uses the
// partition map computed by the sampler.
System.exit(MRJob.waitForCompletion(true) ? 0 : 1);
```

Le calcul de la carte de partitionnement apparaît donc comme une simple étape préliminaire à l'exécution d'une application *Map-Reduce*. Mais elle présente quelques inconvénients :

- Elle ne suit pas le paradigme *Map-Reduce*, car elle ajoute une opération sur le poste client et en amont des opérations *map()*.
- S'il s'avère nécessaire de traiter une grand nombre d'échantillons pour avoir une carte de partitionnement correcte, alors la calculer sur le poste client peut devenir rédhibitoire (à cause des temps de calcul, et des temps d'accès au HDFS d'*Hadoop*).

Donc sur de grosses volumétries, il peut être finalement pertinent de réaliser une application *Map-Reduce* entière qui ne fasse qu'échantilloner et calculer une carte de partitionnement pour la véritable application *Map-Reduce* ! Même si une telle exécution entraîne un surcoût de temps d'exécution non négligeable, elle peut être acceptable si la carte de partitionnement obtenue est de meilleure qualité et peut être réutilisée dans d'autres traitements, sur des données similaires.

6.8 Bilan de la généricité du paradigme *Map-Reduce*

Le paradigme *Map-Reduce* semble au début assez contraint, et pourrait passer pour un simple enchaînement de deux étapes SPMD (les *Mappers* et les *Reducers*), liés par un schéma spécifique d'échange de données. Mais en fait, la phase de *map* peut être réalisée avec un unique programme de *Mapper* ou avec plusieurs (section 6.4.1) et dépasse donc le cadre de la programmation parallèle SPMD (grand classique du parallélisme HPC). D'autre part, la phase de *Shuffle & Sort* ressemble à un *all-to-all* bien qu'elle aille toujours des *Mappers* vers les *Reducers*, et est fortement adaptable à trois niveaux qui permettent de contrôler la redistribution des données. **C'est donc une phase de redistribution et de routage finalement assez générique.**

Enfin, les trois phases de *map*, *shuffle & sort* et de *reduce* sont en fait pipelinées pour recouvrir les traitements, les communications et les E/S dans les fichiers temporaires. **Il s'agit donc d'une mise en œuvre relativement optimisée.**

Ce paradigme de programmation publié par *Google* en 2009, et implanté dans *Hadoop* peu après par Doug Cutting, est donc beaucoup plus générique et optimisé qu'il n'y parait en première lecture. Comparé à des paradigmes de calcul distribué plus classiques en HPC, il ne permet pas d'implanter des schémas de parallélisation quelconques avec un minimum de communications, et est implanté initialement en Java qui n'est pas un langage rapide. En revanche, il permet de masquer aux développeurs les aspects de communications et de gestion des tâches, et de les focaliser sur l'implantation des traitements.

Cette approche a permis d'élargir la communauté des développeurs *Big Data*, à l'opposé des environnements HPC qui ciblent des experts peu nombreux, capables d'écrire des codes distribués et optimisés presque toujours sur mesure.

6.9 Exercices

6.9.1 Analyse de graphe

On considère un graphe de nœuds reliés par des arêtes orientées. La plus simple représentation du graphe est un ensemble de paires clé-valeur de la forme $< X_i, X_j >$, signifiant que le nœud d'identifiant X_i est relié vers le nœud d'identifiant X_j : $X_i \rightarrow X_j$. On adopte les définitions suivantes :

- Le nombre de chemins de sorties à distance 1 d'un nœud est le nombre de liaisons $NbOut_{d1}$ qui sortent de ce nœud pour rejoindre d'autres nœuds. Si on a $X_i \rightarrow X_a$ et $X_i \rightarrow X_b$, alors $NbOut^i_{d1} = 2$.
- De même, le nombre de chemins d'entrées à distance 1 d'un nœud est le nombre de liaisons $NbIn_{d1}$ qui partent d'autres nœuds pour entrer dans ce nœud.
- Le nombre de chemins de sorties à distance 2 d'un nœud est le nombre de chemins différents que l'on peut suivre en sortant de ce nœud et en traversant un nœud intermédiaire. On le note $NbOut_{d2}$.
  Dans le graphe de la figure 6.30, les chemins sortant à distance 2 du nœud 0 sont au nombre de 4 : $0 \rightarrow 1 \rightarrow 2$, $0 \rightarrow 3 \rightarrow 1$, $0 \rightarrow 3 \rightarrow 4$ et $0 \rightarrow 5 \rightarrow 0$.
- Le nombre de chemins d'entrées à distance 2 ($NbIn_{d2}$) d'un nœud est le nombre de chemins différents que l'on peut suivre pour arriver à ce nœud, en traversant un nœud intermédiaire. Dans le graphe de la figure 6.30, les chemins entrant à distance 2 du nœud 0 sont au nombre de 2 : $6 \rightarrow 5 \rightarrow 0$ et $0 \rightarrow 5 \rightarrow 0$. Ce dernier chemin est donc comptabilisé à la fois comme un chemin à distance 2 entrant et sortant. En fait, cette définition va simplifier les traitements de l'exercice.
- Enfin, on considère enfin qu'il n'y a PAS de connexion réflexive d'un nœud sur lui-même.

Question 1 : Calcul du nombre de chemins de sorties à distance 1 de chaque nœud

A partir des paires $< X_i, X_j >$, trouvez une solution en UN *Map-Reduce* permettant de générer pour chaque nœud une paire $< X_a, NbOut^a_{d1} >$. Ecrivez le pseudo-code de votre solution.

Expliquez rapidement pourquoi vous choisissez d'implanter ou non un *Combiner*.

Question 2 : Calcul du nombre de chemins d'entrées à distance 1 de chaque nœud

A partir des paires $< X_i, X_j >$, trouvez une solution en UN *Map-Reduce* permettant de générer pour chaque nœud une paire $< X_a, NbIn^a_{d1} >$. Ecrivez le pseudo-code de votre solution.

Expliquez rapidement pourquoi vous choisissez d'implanter ou non un *Combiner*.

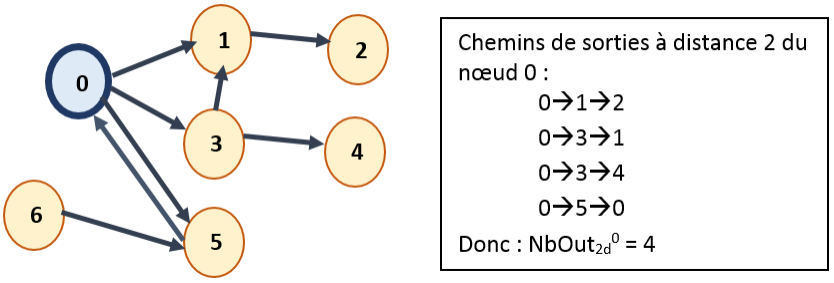

FIGURE 6.30 – Exemple de graphe et de chemins à distance 2

Question 3 : Calcul des nombres de chemins d'entrées et de sorties à distance 1 de chaque nœud

A partir des paires $< X_i, X_j >$, trouvez une solution en UN *Map-Reduce* permettant de générer pour chaque nœud une paire $< X_a, NbIn^a_{d1}, NbOut^a_{d1} >$. Ecrivez le pseudo-code de votre solution.

Expliquez rapidement pourquoi vous choisissez d'implanter ou non un *Combiner*.

Question 4 : Calcul des nombres de chemins d'entrées et de sorties à distance 1 du nœud d'arrivée de chaque connexion

Question un peu difficile.

A partir des paires $< X_i, X_j >$, trouvez une solution en UN *Map-Reduce* permettant de générer pour chaque connexion une paire $< X_i, X_j, NbIn^j_{d1}, NbOut^j_{d1} >$. Cette démarche correspond à un enrichissement de la représentation initiale du graphe. Ecrivez le pseudo-code de votre solution.

Expliquez rapidement pourquoi vous choisissez d'implanter ou non un *Combiner*.

Question 5 : Calcul des nombres de chemins de sorties à distance 1 et à distance 2 de chaque nœud

Repartez des *paires enrichies* obtenues à la question précédente $< X_i, X_j, NbIn^j_{d1}, NbOut^j_{d1} >$, et trouvez une solution en UN *Map-Reduce* permettant de générer pour chaque nœud une paire $< X_a, NbOut^a_{d1}, NbOut^a_{d2} >$. Ecrivez le pseudo-code de votre solution.

Expliquez rapidement pourquoi vous choisissez d'implanter ou non un *Combiner*.

Question 6 : Enchaînement de 2 Map-Reduce

Repartez des paires $< X_i, X_j >$ initiales, afin de générer en DEUX *Map-Reduce* l'ensemble des paires $< X_k, NbIn^k_{d2}, NbOut^k_{d2} >$.

Vous pouvez construire entièrement votre propre solution, ou bien suivre la démarche passant par un ensemble de paires intermédiaires :

$< X_a, direction, X_b, NbIn^b_{d1}, NbOut^b_{d1} >$ avec $direction$ = "out" ou "in"

Si $direction$ = "out", la paire signifie : $X_a \rightarrow X_b$

Si $direction$ = "in", la paire signifie : $X_a \leftarrow X_b$

Question 6.a : Concevez un premier *Map-Reduce* qui vous fera passer des paires initiales aux paires intermédiaires (*question difficile*) :

$$\begin{aligned} < X_i, X_j > \quad &\rightarrow \quad < X_i, "out", X_j, NbIn^j_{d1}, NbOut^j_{d1} > \\ et \quad &\rightarrow \quad < X_j, "in", X_i, NbIn^i_{d1}, NbOut^i_{d1} > \end{aligned}$$

Rmq : n connexions sont représentées par n paires initiales, mais donneront $2 \times n$ paires intermédiaires.

Question 6.b : Concevez un second Map-Reduce qui vous fera passer des paires intermédiaires aux paires recherchées :

$$< X_a, direction, X_b, NbIn^b_{d1}, NbOut^b_{d1} > \quad \rightarrow \quad < X_k, NbIn^k_{d2}, NbOut^k_{d2} >$$

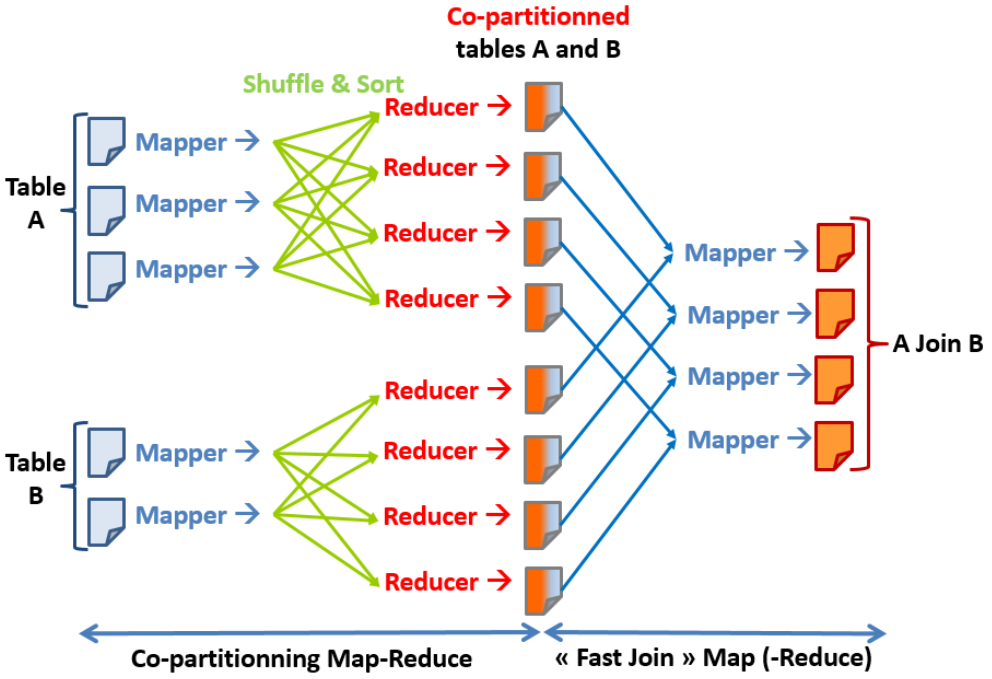

FIGURE 6.31 – Exemple de jointure avec une étape préparatoire de co-partitionnement

6.9.2 Produit de matrices creuses

On considère deux matrices A et B, carrées et creuses, c'est-à-dire comportant beaucoup de valeurs nulles (0.0). On suppose que la longueur du côté des matrices est connue, et notée n.

Les matrices étant creuses, elles sont stockées sous la forme d'ensembles de triplets $(i, j, val_{i,j})$ où $val_{i,j} \neq 0$. Ce format est plus économique qu'un stockage sous forme de tableau dès qu'il y a principalement des valeurs nulles dans les matrices, et reste générique (adapté à tout type de matrices creuses).

Question 1 : Proposez un algorithme *Map-Reduce* pour calculer la matrice carrée $C = A \times B$, de $n \times n$ éléments, où chaque appel à la fonction *reduce()* permet de calculer un élément $C_{i,j}$.

Question 2 : Proposez un algorithme *Map-Reduce* où chaque appel à la fonction *reduce()* permet de calculer tous les éléments d'une ligne de C : $C_{i,*}$.

6.9.3 Jointure de grandes tables

On considère deux grandes tables d'une BdD NoSQL, stockées dans deux gros fichiers HDFS contenant des données structurées sous forme d'ensembles de paires *clé - valeur structurée*. On désire réaliser une jointure selon un attribut commun à chaque table (ex : "UserID"), qui servira de clé d'équi-jointure, et on cherche une solution plus rapide que celle générique vu à la section 6.6.1.

Pour cela, on suppose qu'un prétraitement de *co-partitionnement* peut être effectué préalablement sur toutes les tables concernées :

– on définit un partitionnement des valeurs de la clé de jointure en n intervalles, menant à n sous-ensembles de données de tailles voisines (ce qui suppose une connaissance a priori sur la distribution de la clé de jointure),

– chaque table est ensuite partitionnée en n sous-tables, et stockée dans des fichiers différents de numéros croissants (ex : A_0, $A_1 \ldots A_{n-1}$, et B_0, $B_1 \ldots B_{n-1}$),

Puis une étape de *fast join* peut être réalisée en tirant avantage de l'existence d'un partitionnement identique des tables. Cette dernière étape se réduit alors à un simple *Map*.

La figure 6.31 illustre cette stratégie qui suppose que l'on peut *préparer* plusieurs grandes tables, sur lesquelles on effectue régulièrement des jointures selon une clé donnée.

Question 1 : Quelles paires *clé-valeur* proposez-vous en sortie du *Map-Reduce* de *co-partitionning* ?

Question 2 : Ecrivez le pseudo-code *Map-Reduce* de l'étape de *co-partitionning*.

Question 3 : Concevez enfin le pseudo-code *Map-Reduce* réalisant un *fast join*, à partir des fichiers de co-partitionnement de la question 2.

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.