Chapitre 9

Technologie des moteurs de BdD *NoSQL*

L'implantation des bases de données *NoSQL* pose des problèmes techniques variés, et certaines bases privilégient la capacité à satisfaire un requêtage riche, d'autres d'être capable de traiter une énorme volumétrie, ou encore de traiter des flux de données temps-réel... D'autres problématiques sont en revanche communes à toutes les bases *NoSQL*, comme l'indexage des données distribuées et la gestion de leur réplication afin d'assurer une tolérance aux pannes. Ce chapitre liste les principaux défis et choix technologiques présents dans quelques bases de données *NoSQL*.

9.1 Principes de fonctionnement et terminologie

Dans la majorité des systèmes distribués de stockage d'information on retrouve des concepts de *sharding* pour distribuer automatiquement les données sur différents nœuds, de *réplication* de chaque donnée en quelques exemplaires sur plusieurs nœuds, et dans le cas des BdD NoSQL on retrouve assez souvent un *mécanisme de Map-Reduce* permettant d'écrire des requêtes complexes. Tous ces concepts sont habituellement mis en œuvre sous forme de services se référençant les uns les autres, et formant finalement un *graphe de services*.

Une stratégie de *déploiement* de ce graphe de services sur un réseau de serveurs physiques (réseau de PC) doit ensuite être définie et mise en œuvre. Une stratégie basée sur une forte redondance sera plus robuste aux pannes, mais demandera plus de ressources physiques.

9.1.1 Distribution automatique des données (*sharding*)

Dans une BdD distribuée la stratégie de distribution des données sur les différents machines a pour premier but de permettre un *passage à l'échelle* (voir section 3.6). C'est-à-dire de permettre d'écrire et de lire de plus gros volumes de données en utilisant plus d'espace disque, plus de bande passante disque, plus de mémoire et plus de processeurs. Si l'on peut toujours utiliser plus de ressources quand le volume de données augmente, alors on peut (1) arriver à traiter le problème (!), et (2) espérer le traiter dans le même temps qu'un problème plus petit sur moins de ressources. On retrouve là les concepts de *passage à l'échelle* étudiés dans la section 3.6, mais aussi de *scale out* : pouvoir augmenter le nombre de machines autant que nécessaire pour traiter de plus gros problèmes, plutôt que de chercher à utiliser une machine plus grosse (ce qui coûte plus cher et rencontre forcément une limite).

Le mécanisme de répartition et de distribution des données sur un ensemble de machines (de *nœuds*) est donc très important. On souhaite que cette distribution soit possible, pour être capable de passer à l'échelle, et soit automatisée, pour que le simple utilisateur de la BdD ne perçoive pas les aspects techniques du stockage sous-jacent. Mais on souhaite également que les performances globales augmentent et surtout qu'elles ne diminuent pas : qu'il ne soit pas trop souvent nécessaire

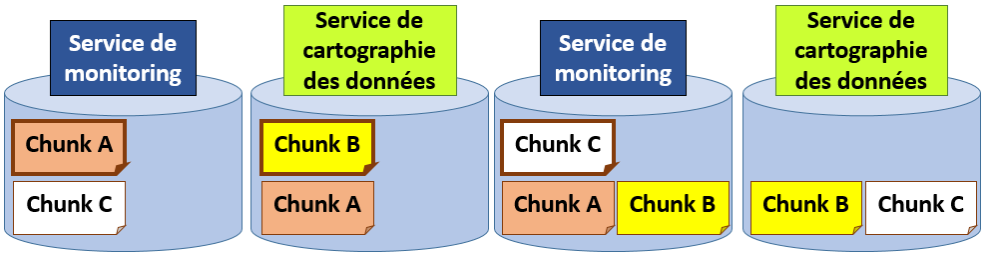

FIGURE 9.1 – Exemple de déploiement économique d'un système fortement répliqué

d'échanger des messages entre les machines pour répondre aux requêtes des utilisateurs. Dans le cas des BdD relationnelles on cherche par exemple à garder sur un même nœud les tables que l'on croise le plus souvent par des jointures. Dans le cas des BdD *NoSQL* leurs mécanismes de distribution des données visent plutôt à équilibrer la charge des nœuds de données, car elles sont conçues pour éviter d'avoir à réaliser des jointures. Cette opération critique de répartition/distribution automatisée et optimisée est appelée *sharding*.

Le *sharding* peut reposer sur l'action d'une (seule) machine maître ou bien être réalisé dans une architecture totalement distribuée sans maître. Certaines BdD *NoSQL* spécifiques ne pratiquent pas le *sharding* sur un ensemble de nœuds, mais la tendance est toutefois d'implanter un *sharding* natif et sous le contrôle d'un mécanisme redondé et tolérant aux pannes.

9.1.2 Réplication automatique des données

Deux grandes stratégies existent afin de ne jamais perdre de données stockées ni d'être simplement dans l'incapacité d'accéder aux supports de stockage, c'est-à-dire pour être tolérant aux pannes. L'une consiste à utiliser du matériel haut de gamme, cher et réalisant un stockage redondé au plus bas niveau (disques multiples en parallèle, bit de parité, technologies RAID...). L'autre utilise au contraire des nœuds de stockage bon marché pour y répliquer les données sur ces différentes machines, et gérer cette réplication au niveau de l'application ou du middleware (voir figure 9.1). Le Big Data est orienté vers cette deuxième solution, et les BdD *NoSQL*, ainsi que les systèmes de fichiers distribués comme le HDFS d'Hadoop, gèrent eux-mêmes la réplication des données.

Le facteur de réplication standard du *Big Data* est de 3. Chaque données est tripliquée et stockée sur 3 machines différentes, si possible sur différents sites (ou au moins dans différentes baies éloignées les unes des autres). En fait les fichiers sont décomposés en sous-parties contiguës : des *chunks*, qui sont répliqués. Un fichier réel sur disque est donc souvent un simple *réplicat* d'un *chunk*. Au cas où un *chunk* est inaccessible, un de ses réplicats est accédé. Si le *chunk* inaccessible le reste longuement, une autre copie est faite dans un nouveau réplicat.

9.1.3 Réplication de services dans une BdD *NoSQL*

Toute solution basée sur une distribution et une réplication de données sur un ensemble de nœuds nécessite un *monitoring* du système pour détecter et isoler les parties qui tombent en panne, et une cartographie de la distribution et réplication des fichiers pour être capable de les retrouver et de savoir quels nœuds interroger. Pour assurer ces fonctionnalités on trouve habituellement un *service de monitoring* et un *service de cartographie*, installés sur des nœuds du système.

Sans ces services rien ne fonctionne, et on ne peut plus exploiter la BdD. Par conséquent il est nécessaire de redonder ces deux services sur plusieurs nœuds (voir la figure 9.1), comme on l'a fait avec les données.

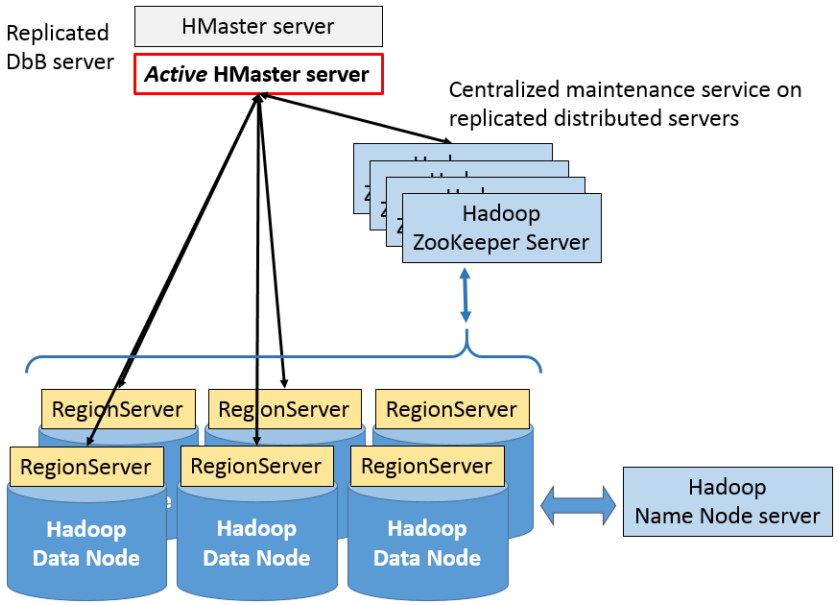

FIGURE 9.2 – Architecture d'une BdD HBase, bâtie au dessus d'Hadoop

9.1.4 Déploiement robuste et économique

Répliquer n fois les données et les nombreux services d'une BdD *NoSQL* augmente considérablement le coût de l'infrastructure. Une stratégie classique de tolérance aux pannes par réplication avec une augmentation limitée des coûts, consiste à utiliser chaque nœud plusieurs fois pour stocker des réplicats différents et/ou pour héberger des services différents. Il faut simplement éviter de stocker des réplicats redondants ou des services redondants sur le même nœud. La figure 9.1 donne un exemple de déploiement à la fois robuste et économique, où aucun *chunk* de données ni aucun service ne disparaît complètement si un nœud devient inaccessible.

9.2 Architecture de l'environnement *HBase*

9.2.1 Principales caractéristiques

HBase est une BdD *NoSQL orientée colonnes*, basée sur le système de fichiers distribué et le mécanisme de Map-Reduce d'Hadoop (et très proche de la BdD *Big Table* de Google).

Une ligne d'une table est indexée par une clé unique, pendant que les colonnes sont regroupées en *familles*. Les colonnes d'une même famille sont gérées ensembles et sont stockées sur disque dans un même fichier appelé un *Hfile*. Il est également possible de spécifier des options de compression à toute une famille de colonnes. Chaque colonne est de plus associée à une estampille (un *timestamp*), qui permet de la *versionner*, et il est possible de spécifier un nombre maximum de versions à sauvegarder pour chaque colonne (les plus anciennes étant automatiquement oubliées). Les estampilles peuvent aussi servir à assurer une cohérence d'ensemble de la famille de colonnes ou de la BdD.

9.2.2 Eléments d'architecture

La gestion distribuées des différents Hfiles est assurée par le système de fichiers distribué d'Hadoop (HDFS), mais la distribution des données (ou *sharding*, voir section 9.1.1) est assurée par des *serveurs de régions* au dessus de chaque nœud de données Hadoop (voir figure 9.2). Une

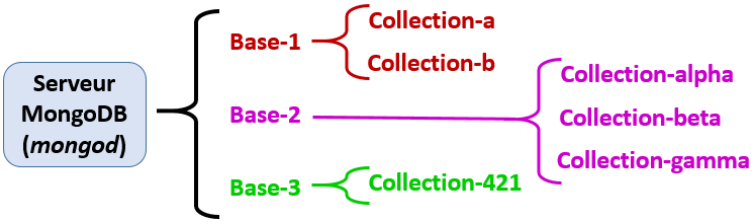

FIGURE 9.3 – Un serveur MongoDB gère des BdD constituées de *collections*

table est au départ contenue dans une seule *région*, puis quand elle grandit et dépasse un seuil elle est automatiquement scindée en deux sous-régions de même taille sur le même serveur de région, chaque sous-région gérant des lignes de clés contiguës. Cette séparation est alors signalée au serveur de la BdD (le HMaster) qui peut demander le déplacement d'une des deux sous-régions (ou portion de table) sur un autre serveur de régions. Le mécanisme se reproduit ensuite sur chaque sous-région qui grandit.

La sauvegarde des données écrites ou modifiées dans une famille de colonnes se fait par stockage dans un buffer mémoire de toutes les modifications effectuées. Ce buffer de modification est sérialisé sur disque seulement quand il atteint une taille fixée, et cette sérialisation se fait en une seule écriture dans un Hfile sur HDFS. L'application de toutes les modifications stockées dans divers Hfiles et visant une famille de colonnes, se fait ultérieurement lors d'une *phase de compactage* des Hfiles. Ces mécanismes peuvent parfois remanier profondément le stockage des données, engendrer de nombreuses IO, et prendre du temps. Ils font donc l'objet de nombreuses optimisations au sein du moteur de la base.

Enfin, le serveur de la BdD (le HMaster) est répliqué, et l'une de ses instances devient le serveur actif. En cas de défaillance, une autre instance est promue active. Le monitoring de l'ensemble et la détection des défaillance sont assurées par un service centralisé de maintenance du système de fichiers distribué d'Hadoop : le *service ZooKeeper*, qui est lui aussi implanté par un *ensemble de serveurs ZooKeeper* (voir figure 9.2). L'ensemble des serveurs de la BdD signalent régulièrement au *ZooKeeper* qu'ils sont bien en vie par envois de messages de *heartbeats*. En l'absence de réception de *heartbeats* d'un serveur, le *ZooKeeper* en déduit une défaillance, et peut la signaler ou entreprendre une compensation. En cas de redémarrage après une coupure de courant, c'est par exemple ce service qui assure l'élection d'un serveur HMaster pour être le nouveau serveur actif.

9.3 Architecture et mécanismes de l'environnement MongoDB

MongoDB est une BdD *NoSQL* qui ne repose pas sur Hadoop. En conséquence les mécanisme de système de fichiers distribué, de répartition automatique des données, de réplication et de requêtage par *Map-Reduce* sont propres à MongoDB. La configuration d'une BdD MongoDB distribuée peut donc apparaître fastidieuse car tout doit être mis en place, au lieu d'avoir été fait auparavant dans une installation d'Hadoop.

9.3.1 Principales caractéristiques

MongoDB est une BdD *NoSQL orientée documents*, c'est-à-dire stockant des données structurées au format JSON (voir section 8.4), mais stockées sur disque au format binaire BSON pour plus d'efficacité. De même, MongoDB est un des rares environnements *NoSQL* développé dès le départ en C++, ce qui le rend très efficace.

Un serveur MongoDB (*mongod*), gère plusieurs BdD, et chaque base contient des *collections* (voir figure 9.3). Une collection est à peu près le pendant d'une table relationnelle dans une BdD

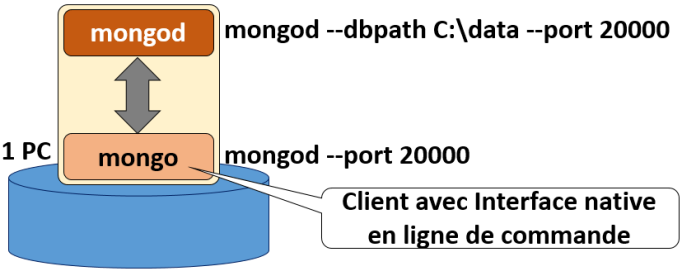

FIGURE 9.4 – Architecture MongoDB minimale, sur un seul PC

relationnelle. Mais les BdD et les collections de MongoDB peuvent être utilisées sans avoir été explicitement créées auparavant, et le seront réellement dès qu'on y aura stocké des données. Il s'agit d'une gestion *paresseuse* des BdD et de leurs collections. Les données entrées dans les BdD du serveur MongoDB seront sauvées sur disque de manière atomique document par document : un document est modifié de manière atomique et reste cohérent, en revanche une séquence de mises-à-jour sur une BdD ne sera pas accomplie de manière atomique, afin de ne pas bloquer longuement l'utilisation de toute la BdD. La BdD peut donc se retrouver momentanément non cohérente .

MongoDB possède plusieurs mécanismes d'indexation des données, afin d'augmenter ses performances. Certains index sont spécialisés pour répertorier des données non contiguës correspondant à une extraction conditionnelle d'une BdD, d'autres sont au contraire spécialisés pour répertorier des données contiguës correspondant à des extractions de colonnes complètes, d'autres encore sont spécialisés dans l'identification de données textuelles et facilitent l'analyse de documents textuels. En fait, une indexation automatique se fait en MongoDB, mais l'utilisateur peut créer explicitement des index et ainsi accélérer ses principales requêtes ou diminuer leur encombrement mémoire.

Si on considère une installation sur un simple PC, sans distribution ni réplication des données, alors elle se réduit à un serveur MongoDB (*mongod*) et une application client, qui peut être une simple interface textuelle native (*mongo*) (voir la figure 9.4). Mais si on considère une installation sur un cluster de PC, MongoDB possède des fonctionnalités de réplication et de distribution des collections (*sharding*) qu'il faut exploiter. Le nombre de réplicats peut être fixé par l'utilisateur, en fonction par exemple de la fiabilité du matériel sous-jacent, mais les opérations de réplication des données et de gestion des réplicats sont gérées par MongoDB. Le *sharding* de MongoDB est basé sur des *clés de sharding* associées à des index, qui permettent de segmenter des documents, de les distribuer et d'équilibrer la charge automatiquement. A travers ces *clés de sharding* et leurs index, l'utilisateur à le choix entre deux politiques de *sharding* prédéfinies : l'une favorable à des accès à des données contiguës en clés de *sharding*, l'autre favorable à des accès plutôt aléatoires. Mais l'utilisateur peut aussi définir sa propre politique de *sharding*, favorable à ses données et à ses requêtes (par exemple avec un découpage et des regroupement basés sur des aspects géolocalisés des données).

Une configuration de *production* cumule la *réplication* et le *sharding*, permettant un passage à l'échelle tolérant aux pannes. Toutefois, MongoDB n'étant pas basé sur Hadoop, le déploiement d'une architecture MongoDB de production sur un cluster de PC doit être faite explicitement et dans les détails.

9.3.2 Eléments d'architecture

La figure 9.5 montre l'architecture générale d'une BdD MongoDB distribuée, avec chaque service déployé (luxueusement) sur un nœud propre. Sur sa partie haute, cet exemple contient n *shard servers* (ou "*shards*"). Une *collection* MongoDB (équivalent d'une table relationnelle) peut

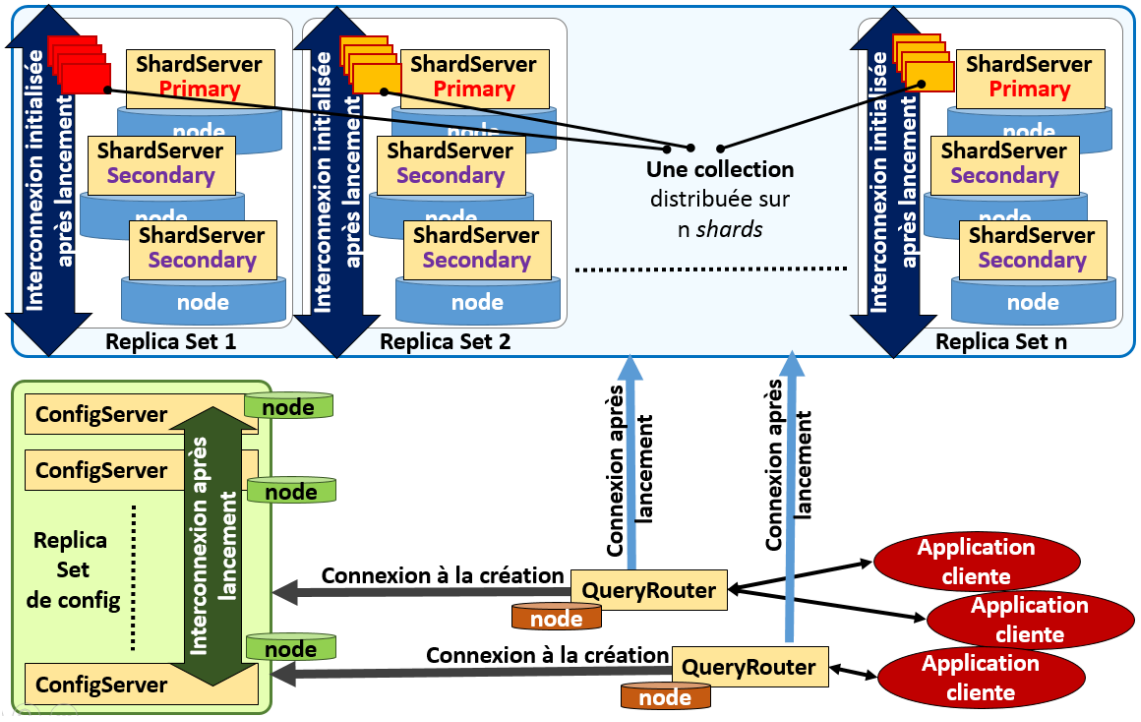

FIGURE 9.5 – Architecture générique d'une BdD MongoDB distribuée où chaque service est installé sur un nœud propre

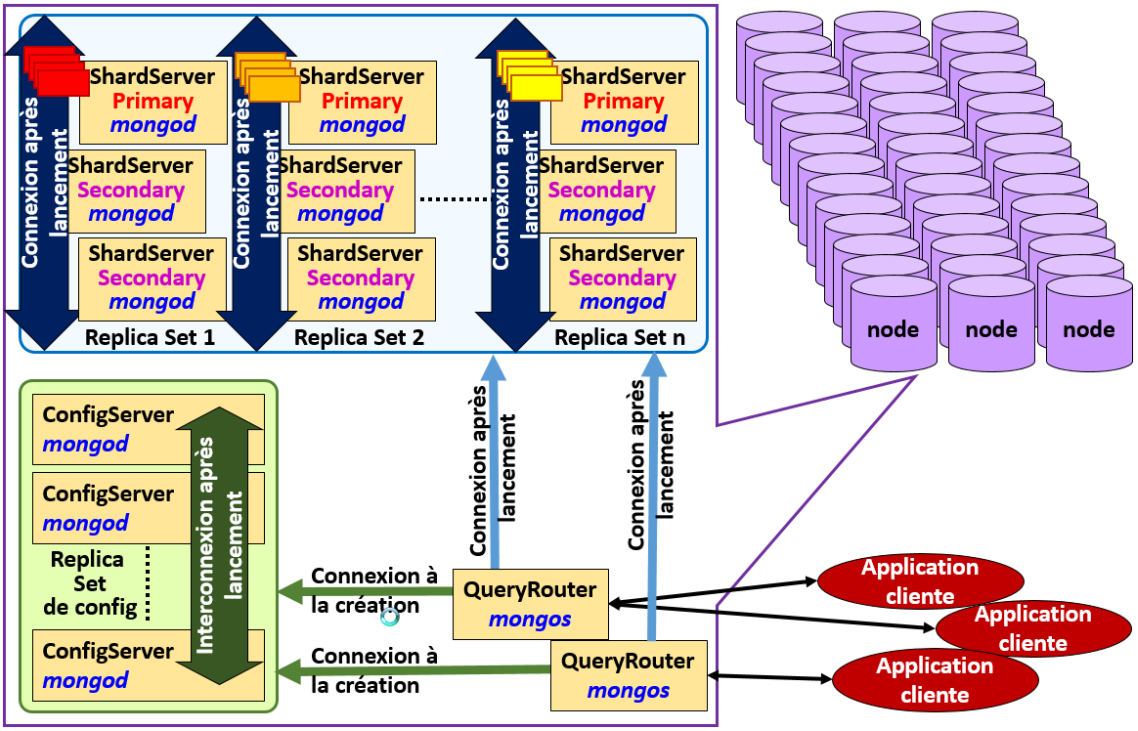

FIGURE 9.6 – Architecture générique d'une BdD MongoDB distribuée où un ensemble de nœuds accueille l'ensemble des services, et où des services non-redondants peuvent partager le même nœud

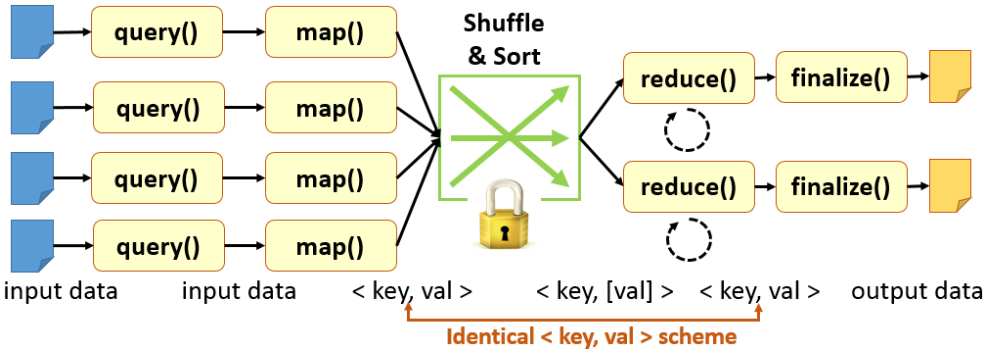

FIGURE 9.7 – Etapes du *mapReduce* de MongoDB

ainsi être distribuée sur n serveurs de stockage pour répartir la charge d'écriture et de lecture des données, et permettre un passage à l'échelle par *scale out*.

Chaque *shard* de la figure 9.5 est constitué d'un ensemble de 3 services formant un *replica set* installés sur 3 nœuds. Chaque *replica set* comprend un service primaire et deux services secondaires. En cas de défaillance ou de simple inaccessibilité du service primaire, un service secondaire est automatiquement élu service primaire, ce qui rend le *replica set* résistant aux pannes (à moins de perdre d'un seul coup la totalité d'un *replica set*). L'utilisateur peut cependant spécifier des lectures explicites à des réplicats secondaires, même s'ils ne sont potentiellement pas complètement à jour, et réaliser ainsi des accès plus rapides en visant des réplicats plus proche de son application cliente. Notons qu'une fois déployé les services d'un *replica set*, il est encore nécessaire de les initialiser pour qu'ils se connaissent mutuellement et forment réellement un *replica set* actif.

La partie gauche de la figure 9.5 décrit un *replicat set* de services de configuration, dont le rôle est de maintenir à jour une vision de la répartition des données sur les différents *shards*. Il est important que ces services soient redondés et toujours accessibles, ce qui est accompli par le mécanisme de *replicat set* déployé sur 3 nœuds différents. Comme précédemment, une fois installé, ce *replicat set* doit être configuré pour devenir actif.

Enfin, sur la partie basse et centrale de la figure 9.5 se trouvent 2 services de *routage de requêtes*. Ils sont associés aux services de configuration lors de leurs créations, puis aux services de *sharding* lors de leur initialisation. Ces routeurs de requêtes agissent comme des *proxy* : des services proches des applications clientes, qui leurs facilitent l'accès à la BdD en masquant son aspect distribuée. Grace aux *routeurs de requêtes*, les applications clientes accèdent donc aussi facilement à une BdD *shardée* que s'il devaient se connecter directement à un serveur MongoDB non distribué et sans réplication sur un simple PC (figure 9.4).

La figure 9.6 montre la même architecture avec uniquement les services sur la partie gauche, et une réserve de nœuds génériques sur la partie droite, prêt à les accueillir. Il est tout à fait possible de déployer tous les services d'une architecture MongoDB distribuée sur un ensemble de nœuds en installant plusieurs services sur chaque nœud, du moment que des services redondants ne partagent pas le même nœud (voir section 9.1.4). On peut donc réaliser un déploiement plus économique d'une architecture MongoDB distribuée, mais il faut implanter le programme qui réalise ce déploiement automatiquement, car pour des architectures de grandes tailles il est impossible de le faire manuellement. On peut assez facilement écrire un tel programme en Python.

9.3.3 Fonctionnement du *"mapReduce"*

La figure 9.7 montre les étapes du mécanisme de *Map-Reduce* inclut dans *MongoDB*. Il est en fait appelé *"mapReduce"* dans la terminologie de *MongoDB*, et présente quelques différences

importantes avec le mécanisme d'*Hadoop* étudié au chapitre 6.

L'étape de *map* est précédée par l'exécution d'une *query* dans le langage natif de MongoDB, c'est-à-dire par l'exécution d'une commande *find(...)*. Habituellement ce sont justement les processus *Mapper* qui réalisent des sélections et projections des données, une *query* en entrée n'est donc pas nécessaire. Mais une *query* dans le langage natif de la base MongoDB sera plus rapide que l'interprétation d'un code utilisateur écrit en *JavaScript* dans une fonction *map*, et sera également plus simple à exprimer.

Ensuite, les étapes de *map* est de *Shuffle & Sort* sont similaires à celles du *Map-Reduce* d'*Hadoop*. Toutefois, le *Shuffle & Sort* n'est pas modifiable, contrairement à celui d'*Hadoop*, ce qui est illustré sur la figure 9.7 par la présence d'un cadenas. On note également qu'il n'y a pas de *Combiner* dans cette suite *mapReduce* de MongoDB, qui commence à apparaître plus simple et moins optimisable que celle d'*Hadoop*.

Mais c'est l'étape de *reduce* qui concentre les plus grosses différences entre *Hadoop* et MongoDB, et qui provoque des divergences de fonctionnement ou de conception des algorithmes de *Map-Reduce*. La fonction *reduce* est toujours appelée pour traiter des paires *clé - listes de valeurs*, où la liste des valeurs regroupe toutes les valeurs associées à une même clé. Cette liste peut donc devenir énorme (s'il y a accumulation de valeurs sur une même clé), et provoquer des débordements mémoire dans la tâche *Reducer* si elle reçoit et traite cette liste en une seule fois. MongoDB évite cet écueil en traitant les listes de valeurs par parties : si la liste est importante la fonction *reduce* sera appelée plusieurs fois sur des portions de la liste. Mais pour que l'opération *reduce* soit toujours globale sur toute la liste, à chaque appel la fonction *reduce* traitera une nouvelle partie de la liste et des résultats des traitements des parties précédentes. Cette façon de réaliser l'étape de *Reduce* évite les débordements mémoire, mais impose :

- de concevoir des algorithmes de *Reducer* commutatifs, associatifs et idempotents, c'est-à-dire vérifiant :
$$reduce(reduce(x)) = reduce(x),$$
- de générer des paires *clé-valeur* de sortie des *Reducers* qui soit au même format qu'en sortie des *Mappers* (comme indiqué au bas de la figure 9.7).

Cette stratégie contraint fortement les algorithmes *mapReduce* et peut nécessiter d'enchaîner plusieurs *mapReduce* de MongoDB, là où *Hadoop* ne ferait qu'un seul *Map-Reduce*.

Enfin, une étape de post-traitement appelant une fonction *finalize* fait suite à chaque exécution complète d'un *Reducer*. Elle permet notamment de générer des données de sortie au format voulu, et non pas obligatoirement au format de sortie des *Mappers*.

La suite *mapReduce* de MongoDB est donc algorithmiquement plus contrainte que la suite *Map-Reduce* d'*Hadoop*, et s'avère moins générique. En revanche, une fois une solution trouvée (éventuellement en enchaînant plusieurs *mapReduce*) elle peut être plus simple à implanter. Pour terminer, notons qu'une fois un traitement *mapReduce* achevé, il est possible de lui faire conserver ou non ses fichiers de sortie, et il est possible de lui faire générer une collection *shardé*.

9.4 Environnements *NoSQL* de plus haut niveau d'abstraction

9.4.1 *Pig*

Objectifs et principes

Pig est un environnement d'exploitation de BdD *NoSQL* bâti au dessus de *Map-Reduce* et d'*Hadoop*. Il possède un plus haut niveau d'abstraction que *Map-Reduce*, et se veut donc plus simple à utiliser, par un public plus large, pour réaliser de l'analyse de données. Il est né en 2006 au

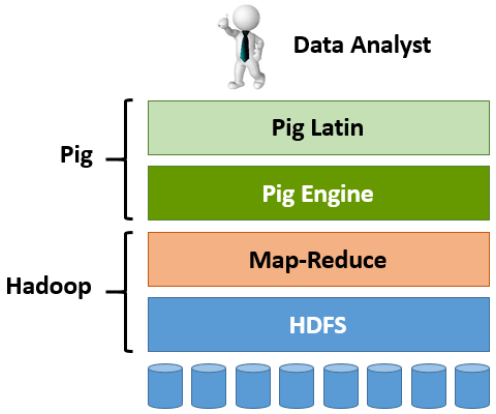

FIGURE 9.8 – Pile logicielle de *Pig*

sein de *Yahoo !*, qui voulait simplifier le travail de ses équipes de fouille d'énormes BdD (*mining*). En fait, les *data analysts* éprouvent des difficultés à exploiter directement le paradigme *Map-Reduce*, et encore plus à écrire des codes *Map-Reduce* dans l'environnement *Hadoop*. *Pig* est une tentative de réponse à ce problème par *Yahoo !*, qui est ensuite devenu un projet OpenSource en 2007.

On retrouve dans *Pig* des commandes comme *FILTER*, *GROUP BY*, *JOIN* et *SORT*, qui correspondent à des programmes complets dans le paradigme *Map-Reduce* (voir chapitre 6). Ces commandes *Pig* sont beaucoup plus simples à utiliser, mais les applications *Pig* ne sont performantes que sur de grosses volumétries et en tâche de fond. *Pig* n'est pas adapté à du traitement temps-réel sur de petits volumes de données.

La figure 9.8 montre la pile logicielle de *Pig*. Elle se repose sur celle d'*Hadoop*, et se décompose en deux couches :

- Une couche de programmation avec le langage de script *Pig Latin*, qui compile ses codes pour obtenir des *plans d'exécutions* sous forme de graphes acycliques (*DAG*) d'opérations *Map-Reduce*. Cette compilation tente d'exprimer un maximum de parallélisme dans le DAG,
- Une couche d'exécution des *DAG Map-Reduce* générés, appelée le *Pig Engine* que l'on peut voir comme un *Middleware*). Le *Pig Engine* fixe le nombre de *Reducers* au mieux pour obtenir le maximum de parallélisme à l'exécution, mais il est possible de le guider.

Langage *Pig Latin* et exemples d'utilisations

Un code *Pig Latin* exprime un flot de transformations de données et le traduit en un graphe acyclique d'opérations *Map-Reduce*. Les mécanismes de cette traduction internes à *Pig* ne sont pas étudiés dans ce cours, mais nous allons introduire rapidement la capacité d'expression du langage *Pig Latin*. Il y a en fait 3 façons d'exploiter *Pig*, en interprétant globalement un fichier de script (*pig myscript.pig*), en utilisant l'interpréteur de commandes de *Pig* nommé *grunt* (voir les exemples ci-après), ou à travers un langage comme Java ou Python en créant et en dialoguant avec un objet serveur *Pig*.

La figure 9.9 illustre un filtrage de données où le quatrième attribut est ignoré et où le troisième est incrémenté de 1. La combinaison du *FOREACH* et du *GENERATE* permet de créer facilement une nouvelle donnée structurée à partir de la donnée initiale. La figure 9.10 montre un exemple de jointure entre les données structurées *Véhicule* et *Constructeur* pour aboutir à la donnée *Voiture*. La jointure se fait très simplement en précisant le nom de chaque donnée et le rang des attributs

```
grunt> DUMP ETUDIANT;
(Dupond, Alphonse, 10, Metz)
(Dupond, Grégoire, 9, Rennes)
(Durant, Hubert, 12, Gif)
(Talon, Achille, 15, Gif)

grunt> ETUDIANT2 = FOREACH ETUDIANT GENERATE $0, $1, $2+1 ;

grunt>DUMP ETUDIANT2;
(Dupond, Alphonse, 11)
(Dupond, Grégoire, 10)
(Durant, Hubert, 13)
(Talon, Achille, 16)
```

FIGURE 9.9 – Exemple de filtrage de données en *Pig Latin*

```
grunt> DUMP VEHICULE;
(AA 123 AB, bleue, Twingo)
(AZ 451 GT, noire, C3)
(BC 634 FY, jaune, Twingo)
(DE 398 AA, blanche, DS4)

grunt>DUMP CONSTRUCTEUR;
(Twingo, Renault)
(C3, Citroen)
(DS4, Citroen)

grunt> VOITURE = JOIN VEHICULE BY $2, CONSTRUCTEUR BY $0;

grunt>DUMP VOITURE;
(AA 123 AB, bleue, Twingo, Twingo, Renault)
(AZ 451 GT, noire, C3, C3, Citroen)
(BC 634 FY, jaune, Twingo, Twingo, Renault)
(DE 398 AA, blanche, DS4, DS4, Citroen)
```

FIGURE 9.10 – Exemple de jointure en *Pig Latin*

```
grunt> VOITURE2 = FOREACH VOITURE GENERATE $0, $1, $2, $4 ;

grunt> VOITURE3 = ORDER VOITURE2 BY $2, $0 DESC ;
(AZ 451 GT, noire, C3, Citroen)
(DE 398 AA, blanche, DS4, Citroen)
(BC 634 FY, jaune, Twingo, Renault)
(AA 123 AB, bleue, Twingo, Renault)
```

FIGURE 9.11 – Exemple de tri en *Pig Latin*

servant de clés de jointure dans un opérateur *JOIN*. Enfin, la figure 9.11 montre la réalisation d'un filtrage pour éliminer un attribut doublon dans le résultat de la jointure précédente, puis surtout la réalisation d'un tri des voitures de la donnée *Voiture2*. L'opération *ORDER* réalise un tri par ordre croissant sur le modèle de la voiture, puis pour un modèle donné un tri par ordre décroissant sur l'immatriculation.

Ces deux derniers exemple d'utilisation de *Pig* montre sa simplicité d'utilisation vis-à-vis de développements directement dans le paradigme *Map-Reduce*. Les versions *Map-Reduce* de la jointure et du tri se sont révélées beaucoup plus complexes (voir les sections 6.6 et 6.5).

Toutefois, SQL est un langage déclaratif : on se contente de décrire le résultat attendu, alors que *Pig Latin* reste un langage procédural : on spécifie comment sont transformées les données pour arriver au résultat voulu. C'est donc un langage qui reste plus compliqué à utiliser que SQL, mais qui permet de spécifier quelle fonction appeler (par exemple en ré-écrivant l'opération de jointure) pour être plus efficace selon le type ou le volume des données d'entrée.

Optimisation et parallélisme

Pig est appréciable par sa simplicité et sa généricité, mais reste un outil d'analyse de gros volumes de données à réaliser en *batch*. Une démarche d'optimisation des scripts est possible mais demande des compétences en *Hadoop*.

Comme nous l'avons évoqué précédemment, *Pig Latin* génère des DAG d'opérations *Map-Reduce*, et tente de maximiser le parallélisme. Pour cela il crée des *Reducers* à raison d'un par Go de données initiales, avec un maximum de 999 *Reducers*. Ce comportement par défaut est modifiable, mais il est surtout intéressant de modifier le nombre de *Mappers* activés simultanément sur chaque nœud et le nombre total de *Reducers*, en fonction de la taille des données traitées. Pas assez de *Mappers* activables en même temps, ou pas assez de *Reducers* sous-exploitera les ressources matérielles disponibles, mais trop de *Mappers*, et surtout trop de *Reducers* mènera à des fichiers de sorties très petits, et peut être même vides. L'optimum de performances se situe entre les deux mais pourra difficilement être trouvé automatiquement par *Pig*.

Il est possible d'optimiser le fonctionnement de *Pig* en analysant les traces de l'exécution précédente. Par exemple, on étudie notamment la taille des fichiers de sortie par *Reducer*, et les temps d'exécution des tâches *Mappers* et *Reducers*. Des fichiers de sorties de petite taille (voire nulle), et des temps d'exécution de quelques secondes sont signes de tâches sous-alimentées en données, donc de trop de tâches *Mappers* ou *Reducers*. On doit alors contraindre le nombre de processus créés et la simultanéité de leurs exécutions pour améliorer les performances. Mais cela demande une connaissance du fonctionnement d'*Hadoop*, alors que l'on cherche justement à masquer son fonctionnement à l'utilisateur de *Pig* !

On peut voir ici une analogie avec les approches HPC où les optimisations de codes demandent de connaître le fonctionnement du compilateur, des bibliothèques utilisées et du *hardware* sous-jacent. Plus généralement, une démarche d'optimisation demande (souvent) de connaître le fonctionnement des couches inférieures, et n'est donc pas accessible à un développeur de trop haut niveau.

9.4.2 Hive

Objectifs et principes

La conception de *Hive* a démarré au sein de *Facebook* en 2005, avant de devenir un projet OpenSource à partir de septembre 2008. L'objectif de *Facebook* était de donner un outil à ses nombreux développeurs très expérimentés en *SQL* mais moins en *Java*, et donc peu enclin à développer du code *Map-Reduce* directement sur *Hadoop*. *Hive* propose un langage *HiveQL* qui permet d'écrire des requêtes proches de *SQL* (avec une forte influence de *MySQL*).

Contrairement à *Pig*, *Hive* propose donc un langage déclaratif qui vise à se rapprocher de *SQL*. Mais comme *Pig*, il est fait pour traiter de gros volumes de données en mode *batch* et n'est pas efficace pour traiter de petits volumes en temps réel. Les temps de latence des traitements *Hive* peuvent être de plusieurs minutes.

Structuration des données et exemples d'utilisations

Les données manipulées par *HiveQL* sont structurées en tables où les lignes ont des structures identiques, et chaque attribut peut avoir un type primitif (scalaire) classique, ou un type complexe : un tableau, une *map* ou une structure. Ce dernier point constitue un enrichissement comparé à *SQL* où les attributs doivent être atomiques pour obtenir un schéma de base efficace. De plus, *HiveQL* n'impose pas de définir un schéma de BdD, ni de vérifier a priori que toute donnée voulant entrer dans la base vérifie ce schéma, contrairement à ce que ferait un SGDB relationnel. *Hive* pratique

```
// Create new table from result of a SELECT command
hive> INSERT OVERWRITE TABLE user_active
SELECT user.*
FROM user
WHERE user.active = 1;

// Projection on one index of an array attribute
hive> SELECT pv.friends[2]
FROM page_views pv;

// Join 2 tables and write in a new one
hive> INSERT OVERWRITE TABLE pv_users
SELECT pv.*, u.gender, u.age
FROM user u JOIN page_view pv ON (pv.userid = u.id)
WHERE pv.date = '2008-03-03';
```

FIGURE 9.12 – Exemple de script *HiveQL*

une vérification paresseuse et vérifie le respect du schéma lors de la lecture d'une donnée (lors de son utilisation), pas lors de son chargement dans la base. Cette démarche permet notamment de charger des données et de se laisser le temps de choisir le schéma à adopter (en fonction des valeurs d'autres données par exemple). La lecture des données est moins rapide puisque des vérifications doivent y être faites, mais est plus souple et conforme à l'approche peu contraignante du *NoSQL*.

L'interpréteur de commandes de *Hive* permet une utilisation interactive de *HiveQL*, toutefois il est possible de l'utiliser en mode non-interactif sur un fichier de script *HiveQL*. Dans les deux cas, cet interpréteur de commandes peut accéder à *HDFS* (et à l'OS sous-jacent) pour lire et écrire des fichiers *Hadoop*, et pour traduire les requêtes *HiveQL* en suites d'opérations *Map-Reduce* (comme le fait *Pig*). Le langage *HiveQL* vise donc à se rapprocher de la syntaxe et de la sémantique de *SQL*, mais suit bien une approche *NoSQL* au dessus d'*Hadoop* et du paradigme *Map-Reduce*.

HiveQL connaît cependant des limitations que ne connaît pas *SQL*, notamment dans l'écriture de requêtes comprenant des sous-requêtes. La traduction en suite d'opérations *Map-Reduce* de sous-requêtes complexes n'est pas toujours possible en *Hive*.

La figure 9.12 montre trois exemples de codes *Hive*, provenant du tutoriel *HiveQL* de la fondation *Apache* [1]. Le premier est un simple *SELECT* par sélection sur la valeur d'un attribut, et dont le résultat sert à créer une nouvelle table (et éventuellement à écraser une table précédente du même nom). Le deuxième est une projection sur l'indice 2 d'un attribut tableau, illustrant une utilisation très facile des attributs de type complexes. Enfin, le troisième exemple illustre une équi-jointure entre deux tables, suivi d'un stockage du résultat dans une nouvelle table (ou avec écrasement des valeurs précédentes). Bien que tout le langage *SQL* ne soit pas reproduit dans *Hive*, ce dernier offre tout de même un bon niveau d'abstraction proche de *SQL* tout en profitant des capacités de stockage et de la souplesse de l'approche *NoSQL*.

Stockage des données

Hive a été conçu pour permettre de stocker les données sous forme de tables, de préciser un partitionnement de chaque table en sous-ensembles selon un attribut clé, et un regroupement des données selon un autre attribut au sein d'un sous-ensemble. Il est également possible de définir des fonctions de sérialisation et de désérialisation pour chaque attribut et pour chaque ligne. On peut ainsi optimiser les stockage des données de *Hive*.

1. https://cwiki.apache.org/confluence/display/Hive/Tutorial

9.5 Exercices

9.5.1 Architecture d'une base *NoSQL*

1. Qu'est-ce que le *sharding* ?
2. Combien de processus/composants différents peut-on identifier dans une BdD MongoDB (distribuée et *shardée*) ? Quel est le rôle de chacun ?

9.5.2 Comparaison des mécanismes de *Map-Reduce*

1. Comparer les mécanismes de *Map-Reduce* de Hadoop et de MongoDB. Quels avantages et inconvénients identifiez-vous pour chaque solution ?
2. Etendez la comparaison aux différents mécanismes de *reduce* de Spark.
3. Est-ce que l'exploitation d'une BdD *NoSQL* à partir de mécanismes *Map-Reduce* a été satisfaisante pour les utulisateurs ? D'autres mécanismes ont-ils vu le jour ?

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.