Maîtrisez les bases de données NoSQL

15 hours Medium License

Last updated on 6/4/21

Passez à l'échelle

Oups, une erreur s'est glissé dans la vidéo. 2 milliards se note bien 2^31 et non 2^64. Mais nous sommes sûr que votre vigilance n'aura pas été trompée !

Les origines de la distribution de données et le sharding

Nous avons parlé de stockage de données et de gestion de la cohérence permettant de faire la différence entre les solutions NoSQL. Toutefois, nous n'avons pas étudié le fonctionnement des différentes solutions. Mais tout d'abord, remettons dans le contexte le fonctionnement d'une base de données, puis voyons ensemble des caractéristiques des bases NoSQL : l'**élasticité** et le **sharding**.

Le but d'une base de données est de fournir efficacement un résultat à chaque requête. Pour cela plusieurs indexes permettent d'accéder directement à l'information recherchée. L'index d'une base de données est comme celui que vous trouvez à la fin d'un livre : lorsque vous cherchez un mot dans celui-ci, vous vous référez à l'index pour connaître les pages contenant ce mot. Ainsi, il est inutile de parcourir toutes les pages pour trouver ce mot. Nous utiliserons cette analogie pour présenter les techniques suivantes.

Les arbres

L'index le plus utilisé est l'arbre B (ou BTree), il adopte une structure arborescente pour chercher toute valeur indexée. Les feuilles de cet arbre contiennent des liens vers les pages contenant la valeur. Dans l'index d'un livre, les mots sont triés par ordre alphabétique. Dans un arbre, c'est la même chose en utilisant une recherche par dichotomie (plus petit/plus grand). Dans le cadre du NoSQL, l'arbre B va encore plus loin car les données sont triées : c'est un arbre B non-dense (ou **clustered index**). Notre livre devient alors un dictionnaire ! L'index est intégré directement au livre pour permettre de trouver directement la page de donnée.

Le hachage

Le but d'une table de hachage est de placer les données par paquets, et à chaque donnée correspond un seul paquet de destination. Pour placer cette donnée, une **fonction de hachage** unique détermine le paquet. Pour reprendre notre analogie, nous parlerons cette fois de bibliothèque : les rangées A à C stockeront les livres dont le nom de l'auteur commence entre la lettre A et C. Pour retrouver un livre d'un auteur, il suffit d'aller dans la bonne rangée, même s'il faut ensuite examiner un certain nombre de livres pour le trouver.

Created by

CentraleSupélec

Grande École d'ingénieurs : cycle ingénieur, Master et École Doctorale, Mastère Spécialisé et formation continue

OpenClassrooms, Leading E-Learning Platform in Europe

La distribution

Les bases de données distribuées existent depuis de nombreuses années ; le but de la distribution est de soulager le serveur central en répartissant les données sur plusieurs serveurs. Ce serveur central s'occupe ainsi de répartir la charge (données et requêtes), de fusionner le résultat et de gérer la cohérence des données. Cela vous fait penser au NoSQL ? Ne vous fiez pas aux apparences ! Les bases de données distribuées ne sont pas tolérantes aux pannes, la disponibilité est problématique en termes de synchronisation multi-serveurs, et il n'y a pas de techniques pour permettre l'élasticité de la base de données.

L'élasticité

L'*élasticité*, nous n'en avons pas encore parlé. C'est la capacité du système à s'adapter automatiquement en fonction du nombre de serveurs qu'il dispose et de la quantité de données à répartir. Par exemple, vous avez 1To de données sur 1000 serveurs (1Go par serveur), lors d'un pic d'activité (soldes, période de Noël...), il serait utile de rajouter 1000 serveurs pour répartir la charge de calcul avec 500 Mo par serveur. L'élasticité va permettre de **répartir** uniformément les données sur les 2000 serveurs (déplacement de la moitié des données), et inversement lorsque le pic d'activité s'achève. De même, si le volume de données augmente et atteint 2To, l'élasticité garantira une répartition uniforme de 2Go par serveur.

Le sharding

Le *sharding* est une technique permettant de distribuer des **chunks** (morceaux de fichiers) sur un ensemble de serveurs, avec la capacité de gérer l'élasticité (serveurs/données) et la tolérance aux pannes. Trois familles de distribution pour le NoSQL existent : *HDFS* (basé sur la distribution), *le clustered index* (basé sur le BTree) et le *consistent hashing* (basé sur les tables de hachage). Regardons comment elles fonctionnent, elles nous permettront d'orienter nos choix d'architecture.

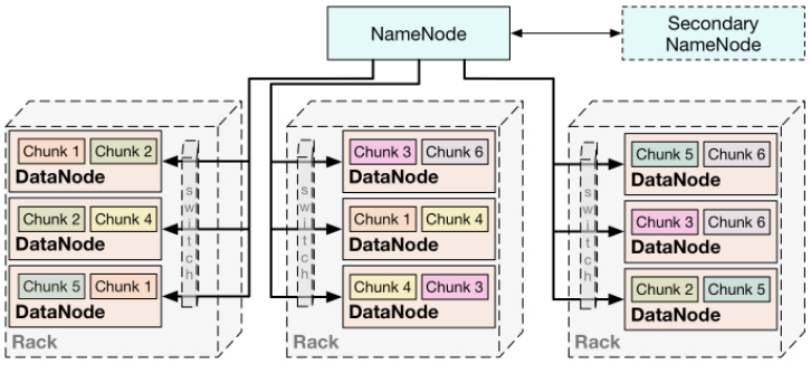
HDFS

HDFS (*Hadoop Distributed File System*) est une technique de distribution de fichiers volumineux. Chaque fichier sera découpé en "chunk" de 64Mo pour être distribué sur le réseau. Chaque serveur de ce réseau est un *datanode* contenant plusieurs chunks. La répartition de ces chunks est définie par le serveur central, le *namenode*.

La ressemblance avec une base de données distribuée classique est frappante. Toutefois, on peut constater quelques subtilités. Chaque chunk est répliqué sur 3 serveurs distincts ; de fait, si un serveur tombe en panne, la donnée peut tout de même être récupérée. Nous pouvons constater également que le **namenode** est lui-même répliqué avec un s*econdary namenode*. Celui-ci permet au serveur central de redémarrer rapidement en cas de panne.

L'autre différence est qu'aucun traitement n'est effectué sur le namenode. Son rôle est de router et d'administrer les datanodes. Les traitements se font donc sur les datanodes, ainsi que la répartition des résultats (produit par Map/Reduce).

La distribution des chunks est maitrisée par le namenode, répartissant ceux-ci en fonction de la structure physique du réseau, minimisant ainsi les problèmes de pannes (serveurs sur un même switch ou rack). On dit que le namenode est **rack-aware**.



Sharding sous HDFS

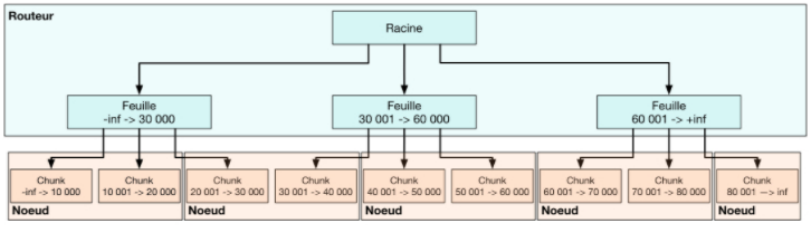
L'**élasticité** se fait assez naturellement. Lorsque l'on rajoute un datanode, il récupère des chunks de ses voisins pour répartir la charge. Lorsqu'un datanode disparaît, les chunks associés sont récupérés sur les replicas pour être répartis sur d'autres nœuds. Toutefois, la synchronisation des chunks en fait une solution sensible aux mises à jour.

L'avantage de cette solution est une forte puissance de calcul et une bonne tolérance aux pannes. On retrouve de nombreuses solutions NoSQL reposant sur cette architecture, notamment celles orientées colonnes : *HBase*, *PigLatin*, *Spark*.

Pour en apprendre plus sur HDFS, n'hésitez pas à consulter **ce cours sur Openclassrooms** !

Un arbre distribué

La seconde famille de distribution de données repose sur le traditionnel BTree non-dense (ou **clustered index**) : il s'agit de l'arbre dont les données sont triées. Un serveur central s'occupe de l'arborescence de cet arbre, et les feuilles (les données) sont prises en charge par les nœuds du cluster.



Sharding arborescent

Le serveur central agit comme "**routeur**" en donnant accès au serveur contenant la donnée requise. La gestion de l'arborescence lui permet de connaître la répartition des valeurs par nœud et, de fait, d'agir sur l'élasticité en modifiant les bornes de chaque nœud. Ceci permet de préserver en permanence le tri des données. Afin de garantir les pannes, le routeur est *répliqué* et oblige la synchronisation de l'arborescence.

Les nœuds contenant les données s'occupent des traitements (Map/Reduce), mais gèrent également la *réplication* des données. Contrairement à HDFS, la réplication est maintenue par le nœud lui-même, à cause de l'arborescence imposée par l'arbre. Nous pourrons voir dans le chapitre sur **MongoDB** la technique du *ReplicaSet* permettant d'effectuer cette réplication.

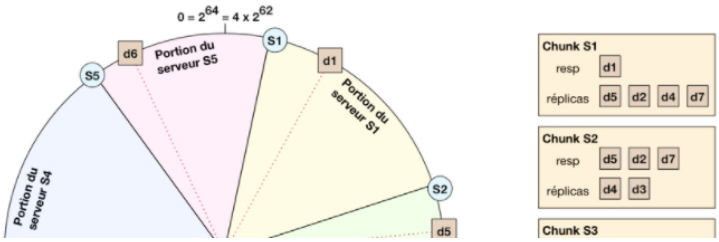
Les avantages de cette technique sont :

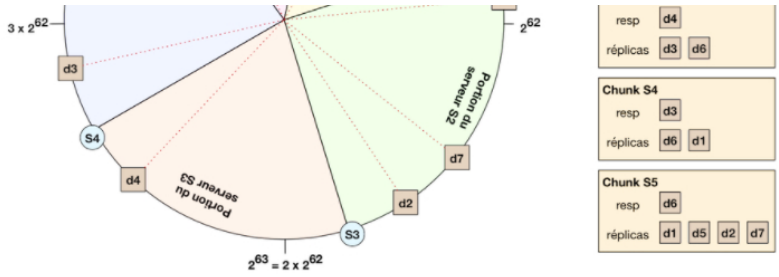
1. rassembler les données ayant des valeurs similaires (tri), et de fait toute opération sur ces données rassemblée (regroupement/reduce)
2. faciliter la gestion de la cohérence des données en verrouillant au niveau du nœud l'information et de synchroniser lui-même ses données avec ses replicas.

Une table de hachage distribuée (DHT)

Avec da dernière famille, basée sur les tables de hachage, on distribue l'intégralité des informations dont on dispose : à la fois la donnée et la table de hachage. C'est ce que l'on appelle le **Consistent Hashing**. La particularité de cette technique est que chaque nœud est à la fois client et serveur. Imaginez un anneau contenant 2^{64} serveurs (soit $1,8 \times 10^{19}$ serveurs !), la *fonction de hachage* place chaque donnée sur cet *anneau*, considéré comme un angle dans un cercle trigonométrique. Un nœud s'occupe de toutes les données dont l'angle est compris entre lui-même et le nœud suivant (*portion du cercle*), un peu comme une part de gâteau.

Bien sûr, vous n'avez pas à allouer 2^{64} serveurs pour faire fonctionner ce réseau. Cet anneau est virtuel et chaque serveur physique est placé sur cet anneau et prend en charge de la portion du cercle allant jusqu'au prochain serveur physique. Comme on peut le voir sur la figure, les 5 serveurs se partagent le gâteau. Les portions ne sont pas forcément de même taille mais doivent équilibrer une certaine charge pour constituer les *chunks*. Les données de ces chunks sont placées grâce à sa valeur de hachage (fonction donnant un nombre entre 0 et 2^{64}, forcément). Cette valeur de hachage peut être considérée comme un angle dans ce cercle, et le nœud ayant la portion contenant cet angle est responsable de la donnée.





Sharding avec une DHT

L'*élasticité* est simple à obtenir : il suffit d'allouer un serveur à une place de l'anneau ayant une trop grande quantité de données. Dans notre exemple, il suffirait d'ajouter un serveur entre S2 et S3 (en gros entre d5 et d7 ou d2 et d7) pour équilibrer la charge. En ce qui concerne la réplication, chaque donnée est répliquée sur trois serveurs physiques contigus sur l'anneau. Notre exemple montre pour chaque chunk les réplicas associés (d4 dont est responsable S3, est répliqué sur les serveurs S2 et S1). Ainsi, lorsqu'un nœud disparaît, le nœud précédent devient le responsable (déjà répliqué) ; il suffit alors de répliquer la donnée une nouvelle fois.

Pour se déplacer dans l'anneau, on ne va clairement pas faire des sauts de serveur en serveur, ce serait interminable ! On va effectuer des **sauts** plus grands, mais dépendant de l'endroit où est située la donnée recherchée. Dans la figure ci-dessus, supposons que le serveur S1 cherche la donnée d3. S1 va procéder par **dichotomie** par rapport au cercle, la donnée est-elle plus loin que le côté opposé du cercle ? En effet, donc on va dans le nœud qui a la charge de cette portion : S3. Du coup, S3 fait lui-même la recherche. Est-ce plus de la moitié ? non. Plus que le quart ? non plus. Plus que le ⅛ ? oui ! On va sur le serveur en question : S4. La donnée est sur S4, on est bon ! Cette dichotomie permet de garantir un temps de réponse acceptable pour toute requête.

Les avantages pour cette technique sont :

1. une architecture totalement distribuée capable de s'**auto-gérer**
2. une **élasticité** simplifiée, même s'il est difficile d'identifier les zones en surcharge (pas de structure de contrôle).

Par contre, cette technique ne peut bénéficier des avantages du HDFS, comme la tolérance à différents types de pannes (c'est un réseau logique et non physique) ou la distribution du BTree (pas de tri).

I FINISHED THIS CHAPTER. ONTO THE NEXT!

MAITRISEZ LE THÉORÈME DE CAP

CHOISISSEZ VOTRE BASE DE DONNÉES NOSQL

Teachers

Régis Behmo

Expert en machine learning, développeur fullstack, grimpeur invétéré et gros, très gros amateur de nouilles chinoises.

Nicolas Travers

Maître de Conférences en informatique au CNAM Spécialités : Bases de données, Optimisation BDD, SQL, NoSQL