Chapitre 5 :
Les bases de Données NoSQL

5. 1. Introduction

L'essor des très grandes plateformes et applications Web (Google, Facebook, Twitter, LinkedIn, Amazon, …) et le volume considérable de données à gérer par ces applications nécessitant une distribution des données et leur traitement sur de nombreux serveurs : « Data Centers » ; ces données étant souvent associées à des objets complexes et hétérogène ont montré les limites des SGBD traditionnels (relationnels et transactionnels) basés sur SQL.

Ainsi, de nouvelles approches de stockage et de gestion des données sont apparues, permettant une meilleure scalabilité dans des contextes fortement distribués et offrant une gestion d'objets complexes et hétérogènes sans avoir à déclarer au préalable l'ensemble des champs représentant un objet.

Ces approches regroupées derrière le terme NoSQL (proposé par Carl Strozzi) ne se substituent pas aux SGBD Relationnels mais les complètent en comblant leurs faiblesses (Not Only SQL).

5.2. Limites des SGBD relationnels

Les SGBD Relationnels offrent un système de jointure entre les tables permettant de construire des requêtes complexes impliquant plusieurs entités ainsi qu'un système d'intégrité référentielle permettant de s'assurer que les liens entre les entités sont valides. Dans un contexte fortement distribué, ces mécanismes ont un coût considérable. En effet, avec la plupart des SGBD relationnels, les données d'une BD liées entre elles sont placées sur le même nœud du serveur. Si le nombre de liens important, il est de plus en plus difficile de placer les données sur des noeuds différents.

D'autre part, les SGBD Relationnels sont généralement transactionnels et la gestion de transactions respectant les contraintes ACID (Atomicity, Consistency, Isolation, Durability). Dans un contexte fortement distribué, cette gestion a un coût considérable car il est nécessaire de distribuer les traitements de données entre différents serveurs, il devient alors difficile de maintenir les contraintes ACID à l'échelle du système distribué entier tout en maintenant des performances correctes. De ce fait, la plupart des SGBD « NoSQL » relâchent les contraintes ACID, ou même ne proposent pas de gestion de transactions.

5.3. Caractéristiques générales des BD NoSQL

Les BD NoSQL :

- Adoptent une représentation de données non relationnelle
- ne remplacent pas les BD relationnelles mais sont une alternative, un complément apportant des solutions plus intéressantes dans certains contextes
- apportent une plus grande performance dans le contexte des applications Web avec des volumétries de données exponentielle
- utilisent une très forte distribution de ces données et des traitements associés sur de nombreux serveurs

- font un compromis sur le caractère « ACID » des SGBDR pour plus de scalabilité horizontale et d'évolutivité.

L'adoption croissante des bases NoSQL par des grands acteurs du Web (Google, faceBook, …) favorise la multiplication des offres de systèmes NoSQL.

5.4. Fondements des systèmes NoSQL

Les systèmes NoSQL se fondent sur les éléments suivants :

5.4.1. Le Sharding

Le « Sharding » est un ensemble de techniques qui permet de répartir les données sur plusieurs machines pour assurer la scalabilité de l'architecture. C'est un mécanisme de partitionnement horizontal (par tuples) des données dans lequel les objets-données sont stockées sur des nœuds serveurs différents en fonction d'une clé (ex : fonction de hachage).

Certains systèmes utilisent aussi un partitionnement vertical (par colonnes) dans lequel des parties d'un enregistrement sont stockées sur différents serveurs.

5.4.2. Le « Consistent hashing»

Le « Consistent hashing» est un mécanisme de partitionnement (horizontal) dans lequel les objet-données sont stockés sur des nœuds-serveurs différents en utilisant la même fonction de hachage à la fois pour le hachage des objets et le hachage des nœuds.

5.4.3. Le « Map Reduce »

Le « Map Reduce » est un modèle de programmation parallèle permettant de paralléliser tout un ensemble de tâches à effectuer sur un ensemble de données,

5.4.4. Le MVCC

Le « MVCC » (Contrôle de Concurrence Multi-Version) est un mécanisme permettant d'assurer le contrôle de concurrence,

5.4.5. Le « Vector – Clock »

Le « Vector-Clock» ou horloges vectorielles permet des mises à jours concurrentes en datant les données par des vecteurs d'horloge.

4.5. Typologie des BD NoSQL

Selon la représentation adopté pour les données, différents types de BD NoSQL existent :

« Clé-valeur / Key-value » : basique, chaque objet est identifié par une clé unique constituant la seule manière de le requêter.

- Voldemort, Redis, Riak, …

« Colonne / Column » : permet de disposer d'un très grand nombre de valeurs sur une même ligne, de stocker des relations « one-to-many », d'effectuer des requêtes par clé (adaptés au stockage de listes : messages, posts, commentaires, ...)

- HBase, Cassandra, Hypertable, …

« Document » : pour la gestion de collections de documents, composés chacun de champs et de valeurs associées, valeurs pouvant être requêtées (adaptées au stockage de profils utilisateur).

- MongoDBn, CouchDB, Couchbase, …

« Graphe » : pour gérer des relations multiples entre les objets (adaptés au données issues de réseaux sociaux, …)

- Neo4j, OrientDB, …

5.5. HBase

HBase est un système de stockage efficace pour des données très volumineuses. Il permet d'accéder aux données très rapidement même quand elles sont gigantesques. Une variante de HBase est notamment utilisée par FaceBook pour stocker tous les messages SMS, email et chat.

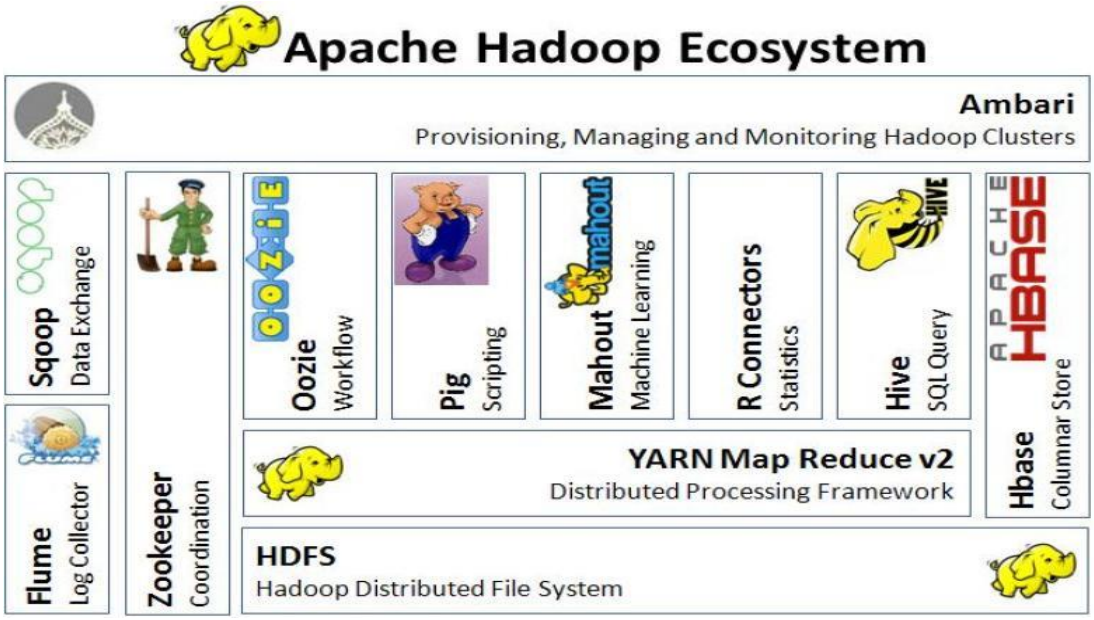

Fig.5.1. HBase dans l'écosystème Hadoop

5.5.1. Présentation

HBase mémorise des n-uplets constitués de colonnes (champs). Les n-uplets sont identifiés par une **clé**. À l'affichage, les colonnes d'un même n-uplet sont affichées successivement : Clés, Colonnes et Valeurs.

isbn7615	colonne=auteur	valeur="Jules Verne"
isbn7615	colonne=titre	valeur="De la Terre à la Lune"
isbn7892	colonne=auteur	valeur="Jules Verne"
isbn7892	colonne=titre	valeur="Autour de la Lune"

Pour obtenir une grande efficacité, les données des tables HBase sont séparées en *régions*. Une région contient un certain nombre de n-uplets contigus (un intervalle de clés successives).

Une nouvelle table est mise initialement dans une seule région. Lorsqu'elle dépasse une certaine limite, elle se fait couper en deux régions au milieu de ses clés. Et ainsi de suite si les régions deviennent trop grosses.

Chaque région est gérée par un Serveur de Région (*Region Server*). Ces serveurs sont distribués sur le cluster, ex: un par machine. Un même serveur de région peut s'occuper de plusieurs régions de la même table. Au final, les données sont stockées sur HDFS.

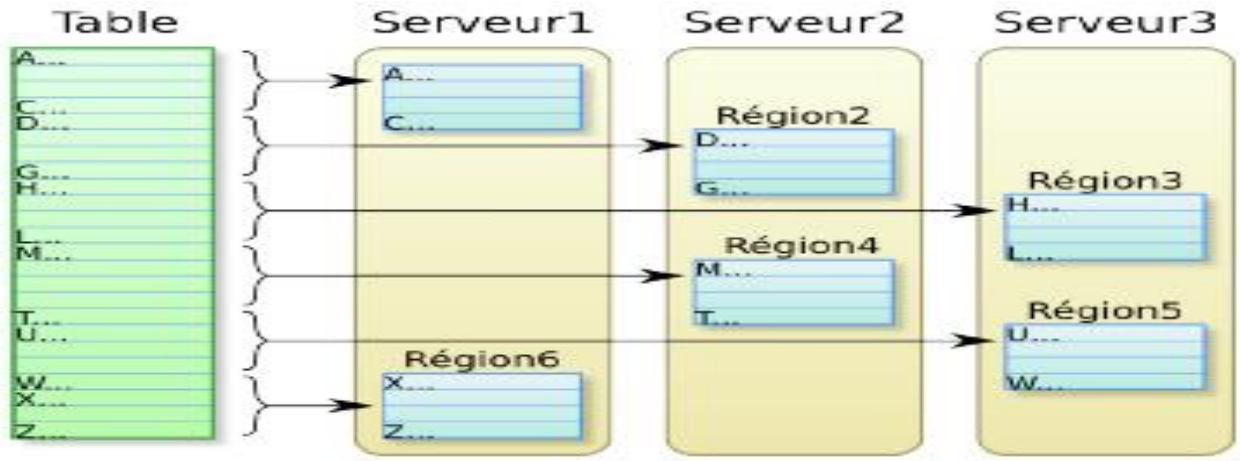

Fig. 5.2. Tables et régions de HBase

5.5.2. Structure de données

Au plus haut niveau, une table HBase est un dictionnaire <clé, n-uplet> trié sur les **clés**,

- Chaque **n-uplet** est une liste de *familles*,
- Une **famille** est un dictionnaire <nomcolonne, cellule> trié sur les noms de colonnes (aussi appelées *qualifier*),
- Une **cellule** est une liste de (quelques) paires <valeur, date>.
- La date, un *timestamp* permet d'identifier la version de la valeur.

Donc finalement, pour obtenir une valeur isolée, il faut fournir un quadruplet :

(clé, nomfamille, nomcolonne, date)

Si la date est omise, HBase retourne la valeur la plus récente.

5.5.3. Nature des clés

Les familles et colonnes constituent un n-uplet. Chacun est identifié par une clé.

Les clés HBase sont constituées de n'importe quel tableau d'octets : chaîne, nombre. . . En fait, c'est un point assez gênant quand on programme en Java avec HBase, on doit tout transformer en tableaux d'octets : clés et valeurs. En Java, ça se fait par :

final byte[] octets = Bytes.toBytes(donnée);

Si on utilise le shell de HBase, alors la conversion des chaînes en octets et inversement est faite implicitement.

Les n-uplets sont classés par ordre des clés et cet ordre est celui des octets. C'est donc l'ordre lexicographique pour des chaînes et l'ordre des octets internes pour les nombres. Ces derniers sont donc mal classés à cause de la représentation interne car le bit de poids fort vaut 1 pour les nombres

négatifs ; -10 est rangé après 3.

Par exemple, si les clés sont composées de "client" concaténée à un numéro, le classement sera :

client1

client10

client11

client2

Client3

Il faudrait écrire tous les numéros sur le même nombre de chiffres.

Pour retrouver rapidement une valeur, il faut bien choisir les clés. Il est important que des données connexes aient une clé très similaire.

Par exemple, on veut stocker des pages web. Si on indexe sur leur domaine, les pages vont être rangées n'importe comment. La technique consiste à inverser le domaine, comme un package Java.

URL	URL inversé
elearning.univ-annaba.dz	com.elwatan.sport
sport.elwatan.com	com.elwatan.www
www.elwatan.com	dz.poste.www
www.ncdc.noaa.gov	dz.univ-annaba.elearning
www.poste.dz	gov.noaa.ncdc.www

Même chose pour les dates : AAAAMMJJ

5.5.4. Exemple

On veut enregistrer les coordonnées et les achats de clients. On va construire une table contenant trois familles:

- La famille personne contiendra les informations de base:
  - colonnes personne:nom et personne:prenom
- La famille coords contiendra l'adresse :
  - colonnes coords:rue, coords:ville, coords:cp, coords:pays
- La famille achats contiendra les achats effectués :
  - colonnes achats:date, achats:montant, achats:idfacture

HBase autorise à dé-normaliser un schéma (redondance dans les informations) afin d'accéder aux données plus rapidement.

5.5.5. Les commandes de base

5.5.5.1. Création d'une table

- create 'NOMTABLE', 'FAMILLE1', 'FAMILLE2'...

D'autres propriétés sont possibles, par exemple VERSIONS pour indiquer le nombre de versions à garder.

Remarques :

- Les familles doivent être définies lors de la création. C'est coûteux de créer une famille ultérieurement.

On ne définit que les noms des familles, pas les colonnes. Les colonnes sont créées dynamiquement.

5.5.5.2. Destruction d'une table

C'est en deux temps, il faut d'abord désactiver la table, puis la supprimer :

1. disable 'NOMTABLE'

2. drop 'NOMTABLE'

Désactiver la table permet de bloquer toutes les requêtes.

5.5.5.3. Ajout et suppression de n-uplet

- **Ajout de cellules**

Un n-uplet est composé de plusieurs colonnes. L'insertion d'un n-uplet se fait colonne par colonne.

On indique la famille de la colonne. Les colonnes peuvent être créées à volonté.

put 'NOMTABLE', 'CLE', 'FAM:COLONNE', 'VALEUR'

- **Suppression de cellules**

Il y a plusieurs variantes selon ce qu'on veut supprimer, seulement une valeur, une cellule, ou tout un n-uplet :

deleteall 'NOMTABLE', 'CLE', 'FAM:COLONNE', TIMESTAMP

deleteall 'NOMTABLE', 'CLE', 'FAM:COLONNE'

deleteall 'NOMTABLE', 'CLE'

5.5.5.4. Affichage de n-uplet

La commande get affiche les valeurs désignées par une seule clé. On peut spécifier le nom de la colonne avec sa famille et éventuellement le timestamp.

get 'NOMTABLE', 'CLE'

get 'NOMTABLE', 'CLE', 'FAM:COLONNE'

get 'NOMTABLE', 'CLE', 'FAM:COLONNE', TIMESTAMP

La première variante affiche toutes les colonnes ayant cette clé. La deuxième affiche toutes les valeurs avec leur timestamp.

5.5.5.5. Recherche de n-uplet

La commande scan affiche les n-uplets sélectionnés par les conditions.

scan 'NOMTABLE', {CONDITIONS}

Parmi les conditions possibles :

COLUMNS=>['FAM:COLONNE',...] pour sélectionner certaines colonnes.

STARTROW=>'CLE1', STOPROW=>'CLE2' pour sélectionner les n-uplets de [CLE1, CLE2[.

Ou alors (exclusif), une condition basée sur un filtre :

FILTER=>"PrefixFilter('binary:client')"

5.5.5.6. Les filtres

L'ensemble des filtres d'un scan doit être placé entre "...".

Plusieurs filtres peuvent être combinés avec AND, OR et les parenthèses.

Exemple :

{ FILTER => "PrefixFilter('client') AND ColumnPrefixFilter('achat')" }

- PrefixFilter('chaîne') : accepte les valeurs dont la clé commence par la chaîne

ColumnPrefixFilter('chaîne') : accepte les valeurs dont la colonne commence par la chaîne.

Ensuite, on a plusieurs filtres qui comparent quelque chose à une valeur constante. La syntaxe générale est du type :

MachinFiter(OPCMP, VAL), . . . avec *OPCMP VAL* définies ainsi :

- OPCMP doit être l'un des opérateurs <, <=, =, !=, > ou >= (sans mettre de quotes autour)
- VAL est une constante qui doit valoir :
  - 'binary:chaine' pour une chaîne telle quelle
  - 'substring:chaine' pour une sous-chaîne
  - …

Exemple :

{ FILTER => "ValueFilter(=,'substring:univ-annaba')"}

Plusieurs filtres questionnent la clé, famille ou colonne d'une valeur :

- RowFilter(OPCMP, VAL)
- FamilyFilter(OPCMP, VAL)
- QualifierFilter(OPCMP, VAL)

accepte les n-uplet dont la clé, famille, colonne correspond à la constante

- SingleColumnValueFilter('fam','col',OPCMP,VAL)

garde les n-uplets dont la colonne 'fam:col' correspond à la constante. Ce filtre est utile pour garder des n-uplets dont l'un des champs possède une valeur qui nous intéresse.

- ValueFilter(OPCMP, VAL)

accepte les valeurs qui correspondent à la constante

Références et Webographie

R. Bruchez, Les bases de données NoSQL: Comprendre et mettre en œuvre, Eyrolles, 2013.

T. White, Hadoop : The Definitive Guide, O'Reilly, 2012.

https://openclassrooms.com/courses/realisez-des-calculs-distribues-sur-des-donnees-massives

http://bradhedlund.com/2011/09/10/understanding-hadoop-clusters-and-the-network/

http://hadoop.apache.org/docs/r1.2.1/hdfs_design.html