BigData - Semaine 5

Pierre Nerzic

février-mars 2019

BY-NC-SA

Le cours de cette semaine présente le SGBD Cassandra, conçu pour le stockage de mégadonnées sous forme de tables ressemblant à celle de SQL.

Ce qui caractérise Cassandra, c'est la distribution des n-uplets sur des machines organisées en anneau et un langage ressemblant à SQL pour les interroger.

Un plugin permet de traiter les données Cassandra à l'aide de Spark.

Pierre Nerzic
BY-NC-SA

Cassandra

Pierre Nerzic
BY-NC-SA

Présentation rapide

Cassandra est totalement indépendant de Hadoop. En général, ces deux-là s'excluent car chacun réquisitionne toute la mémoire et la capacité de calcul.

Cassandra gère lui-même la distribution et la réplication des données. Les requêtes distribuées sont extrêmement rapides.

Cassandra offre un langage d'interrogation appelé CQL très similaire à SQL, mais beaucoup plus limité et certains aspects sont très spécifiques.

Cassandra est issu d'un projet FaceBook, rendu libre en 2008 sous licence Apache. Une version professionnelle est développée par DataStax.

NB: tout ne sera pas expliqué dans ce cours, il faudrait cinq fois plus de temps.

BY-NC-SA

Modèle de fonctionnement

Cassandra est **distribué**, c'est à dire que :

1. Les données sont disposées sur plusieurs machines, avec ou sans réplication (certaines machines ont des données en commun).
2. Les traitements sont effectués simultanément sur ces machines, selon les données qu'elles ont, par rapport à ce qu'il faut faire.

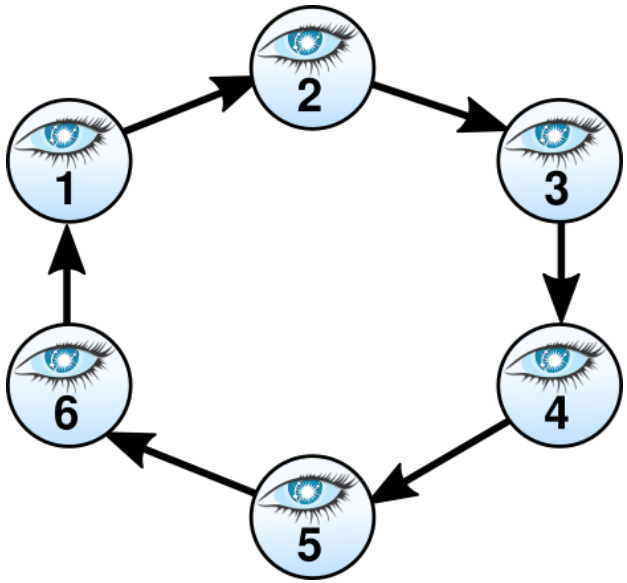
Cassandra est également **décentralisé**, c'est à dire qu'aucune machine n'a un rôle particulier.

- Dans Hadoop, les machines n'ont pas les mêmes rôles : namenode, datanode, nodemanager...
- Au contraire, dans Cassandra, les machines ont toutes le même rôle : stocker les données et calculer les requêtes. On peut contacter n'importe laquelle pour toute requête.

BY-NC-SA

Structure du cluster et données

Les machines, appelées *nodes*, sont organisées en un « anneau » (*ring*) : chacune est reliée à une précédente et une suivante, le tout formant une boucle.

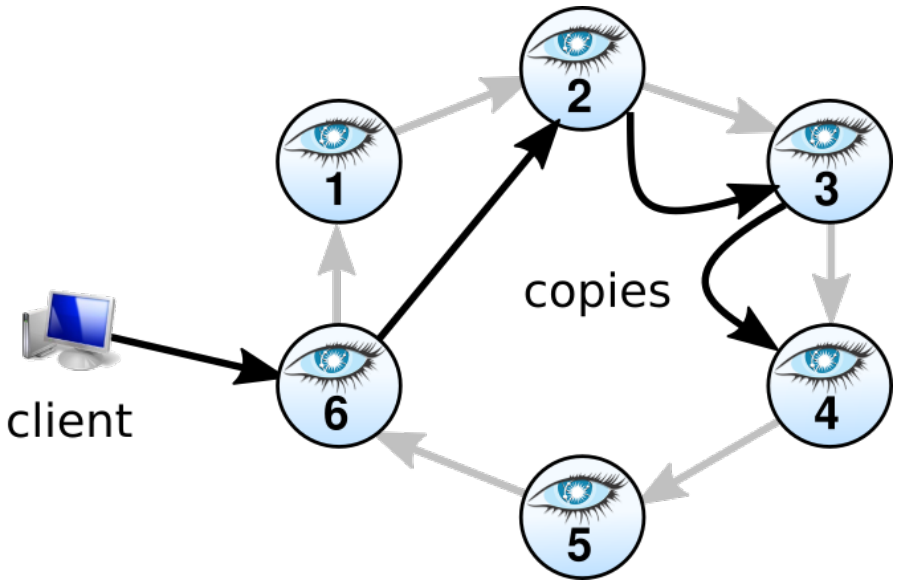


L'anneau est construit automatiquement, par découverte de proche en proche à partir de quelques machines initiales.

Pierre Nerzic
BY-NC-SA

Communication entre machines

Les communications sont gérées d'une manière appelée *Gossip* (rumeur) : les informations vont d'un nœud à l'autre.



Un client dépose des données sur l'un des nœuds, elles sont dupliquées et envoyées aux nœuds concernés (voir plus loin).

BY-NC-SA

Cohérence des données

Les mises à jour des données sont donc effectuées de proche en proche, et de manière non synchronisée.

Ce modèle sans arbitre central pose un problème pour définir la cohérence des données (*consistency*). À un moment donné, il est possible que les machines n'aient pas toutes les mêmes valeurs dans les tables, le temps que les mises à jour se propagent.

BY-NC-SA

Théorème CAP

En fait, c'est un problème de fond. Dans tout système de données, il est impossible d'avoir simultanément :

- la cohérence (**C**onsistency) : les clients lisent tous la même valeur au même moment,
- la disponibilité (**A**vailability) : les données peuvent être lues et écrites à tout moment,
- la résistance aux **P**artitions : les données peuvent être distribuées et résister à des pannes de communication entre machines.

Chaque système privilégie deux de ces critères. Par exemple les SGBD SQL ne garantissent que le couple CA, Cassandra privilégie AP, et HBase uniquement CP.

BY-NC-SA

Théorème CAP

Concernant la consistance, Cassandra propose trois niveaux :

Consistance stricte Celle des SGBD relationnels. Toute lecture de données doit retourner la dernière valeurs écrite.

Consistance causale Si une donnée est modifiée plusieurs fois en séquence par le même agent, il faut obliger les lectures à retourner la dernière valeur. Par contre si elle est modifiée simultanément par des agents différents, alors il se peut que certains ne relisent pas la même valeur que les autres.

Consistance finale Les mises à jour sont faites progressivement, en arrière-plan. Les données ne sont cohérentes qu'au bout d'un certain temps.

Cela se choisit lors de la création des tables.

BY-NC-SA

Modèle de données

Vocabulaire de Cassandra concernant les données :

Espace de clés C'est l'équivalent d'une base de données dans le monde SQL. Un *keyspace* peut contenir plusieurs tables.

Table Une table regroupe des colonnes. Comme en SQL, il y a une clé primaire et les colonnes peuvent être indexées pour des recherches plus rapides.

Colonne Elle contiennent les attributs des n-uplets. Ce sont des paires (clé, valeur), la clé est l'identifiant d'un n-uplet.

Partition C'est l'équivalent d'un n-uplet, une liste de (nom de colonne, valeur). Les n-uplets sont identifiés par la clé primaire.

BY-NC-SA

Stockage des données

Chaque nœud Cassandra stocke une partie des n-uplets d'une table. Ce n'est pas fait n'importe comment :

- La clé primaire des n-uplet est transformée en un nombre appelé *token* par un hachage. Plusieurs algorithmes de hachage sont disponibles : murmur, md5, lexicographique.
- Chaque machine stocke un intervalle de ces hachages (de tel token à tel token) et les intervalles pris en charge se suivent en croissant, selon l'ordre de l'anneau. Ça forme une partition mathématique régulière des tokens.

Ainsi tous les tokens possibles sont quelque part dans l'anneau, et on sait très rapidement à quelle machine s'adresser quand on demande un n-uplet.

BY-NC-SA

Réplication et redistribution des données

En général, pour la fiabilité, les données sont dupliquées N fois, par exemple 3 fois.

Les données d'une machine sont automatiquement recopiées sur les $N-1$ suivantes dans l'ordre de l'anneau, par le *gossip*.

Si une machine devient inaccessible, ou si on rajoute une nouvelle machine dans l'anneau, c'est assez compliqué. La partition des tokens est bouleversée. Normalement, il doit y avoir le même nombre de tokens sur chaque machine. Il faut donc redistribuer les tokens sur les machines voisines.

Pour faire tout cela plus facilement et ne pas surcharger le réseau, Cassandra définit des nœuds virtuels : une machine physique contient plusieurs machines virtuelles, en fonction de leur puissance de calcul, chacune stockant un intervalle de tokens.

BY-NC-SA

Stockage des données

Initialement, les nouveaux n-uplets sont stockés en mémoire, dans une sorte de *cache*. Comme avec un système de fichiers, Cassandra enregistre un journal pour la fiabilité. Ça permet de modifier les données récentes très rapidement.

Les données stables sont placées dans une structure appelée SSTable *sorted string table*. C'est là qu'elles résident, une fois consolidées.

Lorsqu'on supprime un n-uplet, il est simplement marqué comme supprimé (*tombstone*). La suppression effective se fera ultérieurement lors d'un processus appelé compactage (*compaction*). Ce processus reconstruit de nouvelles SSTables à partir des anciennes.

BY-NC-SA

Informations sur le cluster

La commande `nodetool` permet de gérer le cluster. On lui fournit un argument qui indique quoi faire :

- Pour afficher des informations générales sur le cluster : 

```
nodetool info
nodetool describecluster
```

- Pour afficher l'état de chaque machine dans le cluster : 

```
nodetool status
```

- Pour afficher la liste (énorme et incompréhensible) des intervalles de tokens machine par machine : 

```
nodetool ring
```

 BY-NC-SA

Connexion au shell Cassandra CQL

Ce shell permet de manipuler et interroger la base de données dans un langage ressemblant à SQL.

Il faut fournir le nom ou le n°IP d'une des machines du cluster. Par exemple, à l'IUT, il suffira de taper :

```
prompt$ cqlsh master
```

Cela ouvre un shell CQL dans lequel on tape les requêtes (`^D` ou `exit` pour quitter).

La première à connaître est `HELP`. On peut lui ajouter un paramètre, par exemple `HELP DESCRIBE`.

BY-NC-SA

Premières commandes

- Création d'un espace de clés (ensemble de tables)

```
CREATE KEYSPACE [IF NOT EXISTS] nomkeyspace
    WITH REPLICATION = {
        'class': 'SimpleStrategy',
        'replication_factor': 2
    };
```

La stratégie `SimpleStrategy` convient pour les clusters locaux. Ici, les données seront répliquées en 2 exemplaires.

- Suppression d'un keyspace et de tout son contenu

```
DROP KEYSPACE nomkeyspace;
```


BY-NC-SA

Affichage d'informations

- Liste des keyspaces existants

```
DESCRIBE KEYSPACES;
```

- Structure d'un keyspace : cela affiche toutes les commandes servant à le reconstruire ainsi que ses tables

```
DESCRIBE KEYSPACE nomkeyspace;
```

- Sélection d'un keyspace pour travailler

```
USE nomkeyspace;
```

Au lieu de changer de keyspace, on peut aussi préfixer toutes les tables par `nomkeyspace.nomtable`

BY-NC-SA

Premières commandes, suite

- Création d'une table

```
CREATE TABLE [IF NOT EXISTS] nomtable ( def colonnes );
```

On peut préfixer le nom de la table par `nomkeyspace.` si on est hors keyspace ou qu'on veut en désigner un autre.

Les définitions de colonnes sont comme en SQL : *nom type*. Les types sont `boolean`, `int`, `float`, `varchar`, `text`, `blob`, `timestamp`, etc.

Il y a des types spéciaux, comme `counter`, `list`, `set`, `map`, etc.

Voir la documentation.

BY-NC-SA

Identification des n-uplets

Soit une table représentant des clients :

```
CREATE TABLE clients (
    idclient INT,          -- n° du client
    departement INT,       -- n° du département, ex: 22, 29, 35...
    nom TEXT, ...          -- coordonnées du client...
    PRIMARY KEY (...)
```

On a plusieurs possibilités pour la contrainte `PRIMARY KEY` :

- `PRIMARY KEY (idclient)` : les n-uplets sont identifiés par le n° client, c'est la « *row key* »
- `PRIMARY KEY (departement, idclient)` : la clé est *composite*, `departement` sert de « *clé de partition* ». Tous les clients du même département seront sur la même machine et ils seront classés par `idclient`.

BY-NC-SA

Création d'un index secondaire

La clé est un index primaire. On rajoute un index secondaire par :

```
CREATE INDEX ON table ( nomcolonne );
```

L'index s'appelle généralement `table_nomcolonne_idx`.

Pour supprimer un index :

```
DROP INDEX table_nomcolonne_idx
```

Il n'est pas du tout recommandé de construire un index lorsque :

- les données sont extrêmement différentes (ex: une adresse mail)
- les données sont très peu différentes (ex: une année)

Il vaut mieux dénormaliser le schéma, construire une autre table ayant une clé primaire adaptée.

BY-NC-SA

Insertion de données

C'est un peu comme en SQL et avec d'autres possibilités :

```
INSERT INTO nomtable (nomscolonnes...) VALUES (valeurs...);
INSERT INTO nomtable JSON 'données json';
```

Contrairement à SQL, les noms des colonnes concernées sont obligatoires, mais toutes les colonnes n'ont pas obligation d'y être, les absentes seront affectées avec `null`.

Exemples :

```
INSERT INTO clients
    (idclient, departement, nom) VALUES (1, 22, 'pierre');
INSERT INTO clients
    (idclient, nom) VALUES (2, 'paul');
INSERT INTO clients JSON '{"id":3, "nom":"jacques"}';
```


BY-NC-SA

Insertion par fichier CSV

Il est possible de stocker les données dans un fichier CSV et de les injecter dans une table par :

```
COPY nomtable(nomscolonnes...) FROM 'fichier.csv'
    WITH DELIMITER=';' AND HEADER=TRUE;
```

On doit mettre les noms des colonnes dans le même ordre que le fichier CSV.

Une table peut être enregistrée dans un fichier CSV par :

```
COPY nomtable(nomscolonnes...) TO 'fichier.csv';
```

Le fichier sera créé/écrasé avec les données indiquées.

Pierre Nerzic
BY-NC-SA

Sélection de données

C'est comme en SQL :

```
SELECT nomscolonnes... FROM table
    [WHERE condition]
    [LIMIT nombre]
    [ALLOW FILTERING];
```

Les colonnes peuvent utiliser la notation `*`.

Il y a une très forte limite sur la clause `WHERE` : elle doit sélectionner des n-uplets contigus dans un index. Donc c'est limité aux conditions sur les clés primaires ou secondaires. On peut ajouter les mots-clés `ALLOW FILTERING` pour rompre cette contrainte mais ce n'est pas recommandé car ça oblige à traiter tous les n-uplets.

BY-NC-SA

Agrégation

On peut faire ceci comme en SQL :

```
SELECT nomscolonnes... FROM table
    ...
    GROUP BY clé de partition;
```

Les colonnes peuvent faire appel aux fonctions d'agrégation `COUNT`, `MIN`, `MAX`, `AVG`, `SUM` ainsi que des clauses `GROUP BY`. Mais dans ce cas, il est impératif que la colonne groupée soit une clé de partition (la première dans la clé primaire).

Vous voyez que le schéma des tables doit être conçu en fonction des requêtes et non pas en fonction des dépendances fonctionnelles entre les données. Cela implique de *dénormaliser* le schéma.

BY-NC-SA

Autres requêtes

Oubliez les jointures, ça n'est pas possible. Cassandra, comme les autres systèmes, est destiné à stocker d'énormes volumes de données et à y accéder rapidement. Les possibilités de traitement sont très limitées. Pour en faire davantage, il faut programmer avec les API de Cassandra, ou avec SparkSQL, voir plus loin.

CQL offre d'autres possibilités : création de *triggers* et de fonctions, mais ça nous entraînerait trop loin.

BY-NC-SA

Mise à jour de n-uplets

Comme avec SQL, on peut mettre à jour ou supprimer des valeurs :

```
UPDATE nomtable SET nomcolonnes=valeur WHERE condition;
DELETE FROM nomtable WHERE condition;
```

Il faut savoir que les valeurs supprimées sont marquées *mortes*, elles créent seulement une *tombstone*, et seront réellement supprimées un peu plus tard.

Il en va de même avec les mises à jour, elles sont associées à un *timestamp* qui permet de savoir laquelle est la plus récente sur un nœud.

BY-NC-SA

Injection de données

Pierre Nerzic
BY-NC-SA

Présentation

On s'intéresse au remplissage de tables Cassandra par des fichiers extérieurs (HDFS ou autres).

Ce n'est pas un problème avec une application neuve, remplissant ses tables au fur et à mesure. C'est un problème si on dispose déjà des données sous une autre forme et qu'on veut les placer dans Cassandra. Les requêtes `COPY FROM` sont très lentes.

La technique proposée consiste à créer directement des structures de données internes de Cassandra, des `SSTables`, puis d'utiliser un outil du SDK, `sstableloader`, voir cette page pour déplacer ces tables dans les dossiers internes. On appelle cela du *bulk loading*.

BY-NC-SA

Étapes

On suppose que la table et son keyspace sont déjà créés.

1. Dans un premier temps, il faut programmer en Java un lecteur pour les données dont on dispose.
   a. D'abord, il faut préparer deux chaînes : le schéma de la table et la requête CQL d'insertion.
   b. On doit créer un écrivain de `SSTable`. C'est une instance de `CQLSSTableWriter` prenant le schéma et la requête.
   c. Ensuite, on lit chaque n-uplet des données et on le fournit à l'écrivain.

   Il y a une limite RAM au nombre de n-uplets pouvant être écrit ensemble, donc il faut périodiquement changer d'écrivain.
2. Ensuite, on lance `sstableloader` sur ces tables.

Consulter ce blog pour un exemple complet.

BY-NC-SA

Définition du schéma et de la requête d'insertion

Voici comment définir les deux chaînes :

```
String schema = "CREATE TABLE ks.table (...)");
String insert = "INSERT INTO ks.table (...) VALUES (?,?,?...)";
```

Constatez que la requête d'insertion est une requête préparée. Chaque ? sera remplacé par une valeur lors de la lecture des données, mais ça sera fait automatiquement par l'écrivain de `SSTable`.

BY-NC-SA

Création de l'écrivain de SSTable

Ensuite, on initialise l'écrivain à l'aide d'un *builder* :

```
int num = 0;
String outDir = String.format("/tmp/cass/%03d/ks/table",num++);
CQLSSTableWriter.Builder builder = CQLSSTableWriter.builder();
builder.inDirectory(new File(outDir))
       .forTable(schema)
       .using(insert)
       .withPartitioner(new Murmur3Partitioner());
CQLSSTableWriter writer = builder.build();
```

Le dossier destination des SSTable, outDir est à définir là il y a beaucoup de place libre. D'autre part, son chemin est structuré ainsi : dossier/n°/ks/table. Cet écrivain devra être recréé en incrémentant le numéro tous les quelques dizaines de milliers de n-uplets.

BY-NC-SA

Écriture de n-uplets

Voici le principe, extraire les colonnes puis les écrire :

```
// extraire les colonnes des données
String[] champs = ligne.split(";");
Integer col1 = Integer.parseInt(champs[0]);
Float   col2 = Float.parseFloat(champs[1]);
String  col3 = champs[2];

// écrire un n-uplet
writer.addRow(col1, col2, col3);
```

NB: les colonnes doivent être des objets correspondant au schéma. La méthode `addRow` remplit les paramètres de la requête préparée.

Il existe des variantes de `addRow`, consulter les sources, mais le problème principal est le nombre des allocations mémoire qu'elle fait à chaque appel.

BY-NC-SA

Algorithme général

Voici le programme général :

```
import org.apache.cassandra.config.Config;

public static void main(String[] args)
{
    Config.setClientMode(true);
    // ouvrir le fichier de données
    BufferedReader br = new BufferedReader(new InputStreamReader
    String ligne;  long numligne = 0;
    while ((ligne = br.readLine()) != null) {
        // créer l'écrivain si besoin
        if (numligne++ % 50000 == 0) {...}
        // extraire les colonnes des données...
        // écrire le n-uplet...
    }
}
```


BY-NC-SA

Envoi des tables à Cassandra

Pour finir, il reste à placer les SSTables obtenues dans Cassandra. On peut utiliser ce petit script bash : 

```
for d in /tmp/cass/*
do
    sstableloader -d master $d/ks/table
done
```

`master` étant le nom de l'une des machines du cluster. On doit retrouver le *keyspace* et le nom de la table dans le chemin fourni à `sstableloader`.

On va maintenant voir comment utiliser une table Cassandra avec Spark.

BY-NC-SA

SparkSQL sur Cassandra

BY-NC-SA

Présentation

Le cours précédent avait présenté les concepts de *DataFrame*. C'est à dire l'association entre un RDD et un schéma. Cette association est automatique quand on utilise Cassandra en tant que source de données.

Pour cela, il suffit d'importer un *plugin* établissant le lien entre pySpark et Cassandra. Il s'appelle `pyspark-cassandra`. Il fonctionne actuellement très bien, mais hélas, il n'est pas « officiel » et donc pourrait devenir obsolète.

Ensuite, on ouvre une table Cassandra et on obtient un DataFrame sur lequel on peut faire tout calcul Spark souhaité.

BY-NC-SA

Début d'un script

Un script pySpark doit commencer par ces lignes : 

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

from pyspark import SparkConf
from pyspark_cassandra import CassandraSparkContext

# contexte d'exécution pour spark-submit
appName = "MonApplicationSparkCassandra"
conf = SparkConf()  \
       .setAppName(appName) \
       .setMaster("spark://master:7077") \
       .set("spark.cassandra.connection.host", "master")
csc = CassandraSparkContext(conf=conf)
```

 BY-NC-SA

Ouverture d'une table Cassandra

Comment faire plus simple que ceci pour ouvrir une table appelée `clients` dans un keyspace `ks` ?

```
clients = csc.cassandraTable("ks", "clients")
```

Toutes les requêtes Spark sont possibles, voici quelques exemples :

```
print clients.count()
print client.map(lambda client: client.age).filter(None).mean()
```

Les n-uplets sont également vus comme des dictionnaires Python, on peut accèder aux colonnes par la notation `nuplet['nomcol']`

BY-NC-SA

Lancement d'un script

Le lancement est un peu plus compliqué, mais il suffit de faire un script shell :

```
spark-submit \
  --py-files /usr/lib/spark/jars/pyspark-cassandra-0.7.0.jar \
  script.py
```

Si on doit ajouter d'autres scripts Python, tels qu'une classe pour traiter les données, il faut les ajouter après le jar, avec une virgule pour séparer :

```
spark-submit \
  --py-files \
    /usr/lib/spark/jars/pyspark-cassandra-0.7.0.jar,client.py \
  script.py
```


BY-NC-SA