

BigData - Semaine 8

Pierre Nerzic

février-mars 2019

Le cours de cette semaine présente HBase et Hive.

HBase est un SGBD non relationnel, orienté colonne adapté au stockage et à l'accès rapide à des mégadonnées.

Hive est une surcouche de HBase afin d'offrir des fonctionnalités similaires à SQL.

Introduction

Présentation de 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, voir [cette page](#).

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"

Structure interne

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).

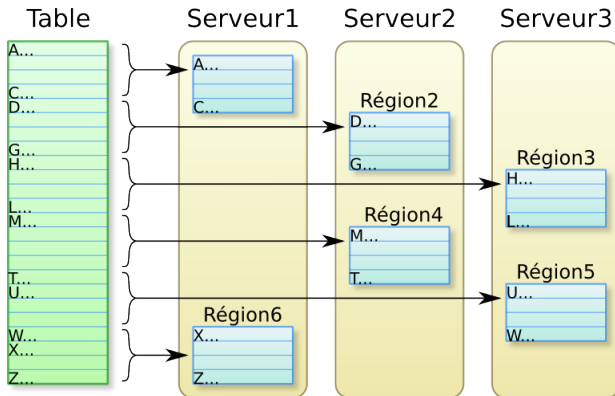
Une nouvelle table est mise initialement dans une seule région. Lorsqu'elle dépasse une certaine limite, elle se fait couper en deux régions au milieu de ses clés. Et ainsi de suite si les régions deviennent trop grosses.

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.

Tables et régions

Une table est découpée en régions faisant à peu près la même taille. Le découpage est basé sur les clés. Chaque région est gérée par un Serveur de région. Un même serveur peut gérer plusieurs régions.



Différences entre HBase et SQL

Voici quelques caractéristiques de HBase :

- Les n-uplets sont classés selon leur clé, dans l'ordre alphabétique. Cette particularité est extrêmement importante pour la recherche d'informations. On est amené à définir les clés de façon à rapprocher les données connexes.
- Les n-uplets de HBase peuvent être incomplets. Les colonnes ne sont pas forcément remplies pour chaque n-uplet, au point qu'on peut même avoir des colonnes différentes pour les n-uplets. Ce ne sont pas des valeurs `null`, mais des colonnes carrément absentes. On qualifie ça de « matrice creuse » (*sparse data*).
- Les colonnes appelées *qualifiers* sont groupées en *familles*.
- Les valeurs, appelées *cellules* sont enregistrées en un certain nombre de versions, avec une date appelée *timestamp*.

Structure des 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.

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.

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);
```

Voir page 33 pour tous les détails.

Si on utilise le shell de HBase, alors la conversion des chaînes en octets et inversement est faite implicitement, il n'y a pas à s'en soucier.

Ordre des clés

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.

Choix des clés

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
info.iut-lannion.fr	com.alien.monster
monster.alien.com	com.alien.www
mp.iut-lannion.fr	fr.iut-lannion.info
www.alien.com	fr.iut-lannion.mp
www.iut-lannion.fr	fr.iut-lannion.www

Même chose pour les dates : AAAAMMJJ

Éviter le hotspotting

Il faut également concevoir les clés de manière à éviter l'accumulation de trafic sur une seule région. On appelle ça un point chaud (*hotspotting*). Ça arrive si les clients ont tendance à manipuler souvent les mêmes n-uplets.

Paradoxalement, ça peut être provoqué par le classement des clés pour l'efficacité comme dans le transparent précédent.

Ça vient aussi du fait qu'il n'y a qu'un seul serveur par région¹. Il n'y a donc pas de parallélisme possible.

Pour résoudre ce problème, on peut disperser les clés en rajoutant du « sel », c'est à dire un bidule plus ou moins aléatoire placé au début de la clé, de manière à écarter celles qui sont trop fréquemment demandées : un timestamp, un hash du début de la clé. . .

¹Une évolution de HBase est demandée pour permettre plusieurs serveurs sur une même région.

Travail avec HBase

Shell de HBase

HBase offre plusieurs mécanismes pour travailler, dont :

- Un shell où on tape des commandes,
- Une API à utiliser dans des programmes Java, voir plus loin,
- Une API Python appelée HappyBase.

Il y a aussi une page Web dynamique qui affiche l'état du service et permet de voir les tables.

On va d'abord voir le shell HBase. On le lance en tapant :

```
hbase shell
```

Il faut savoir que c'est le langage Ruby qui sert de shell. Les commandes sont écrites dans la syntaxe de ce langage.

Commandes HBase de base

Voici les premières commandes :

- `status` affiche l'état du service HBase
- `version` affiche la version de HBase
- `list` affiche les noms des tables existantes
- `describe 'table'` décrit la table dont on donne le nom.

NB: ne pas mettre de ; à la fin des commandes.

Attention à bien mettre les noms de tables, de familles et de colonnes entre '...'

Les commandes suivantes sont les opérations **CRUD** : créer, lire, modifier, supprimer.

Création d'une table

Il y a deux syntaxes :

- `create 'NOMTABLE', 'FAMILLE1', 'FAMILLE2'...`
- `create 'NOMTABLE', {NAME=>'FAMILLE1'},
{NAME=>'FAMILLE2'}...`

La seconde syntaxe est celle de Ruby. On spécifie les familles par un dictionnaire {propriété=>valeur}. 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.

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.

Ajout et suppression de n-uplets

■ 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'
```

Affichage de n-uplets

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.

Recherche de n-uplets

La commande `scan` affiche les n-uplets sélectionnés par les conditions. La difficulté, c'est d'écrire les conditions en Ruby.

```
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')"`

Il existe de nombreux filtres, voir [la doc](#).

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.

Filtres, suite

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:chaîne'` pour une chaîne telle quelle
 - `'substring:chaîne'` pour une sous-chaîne
 - `'regexstring:motif'` pour un motif egrep, voir [la doc](#).

Exemple :

```
{ FILTER => "ValueFilter(=,'substring:iut-lannion')"
```

Filtres, suite

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

Comptage de n-uplets

Voici comment compter les n-uplets d'une table, en configurant le cache pour en prendre 1000 à la fois

```
count 'NOMTABLE', CACHE => 1000
```

C'est tout ?

Oui, HBase n'est qu'un stockage de mégadonnées. Il n'a pas de dispositif d'interrogations sophistiqué (pas de requêtes imbriquées, d'agrégation, etc.)

Pour des requêtes SQL sophistiquées, il faut faire appel à Hive. Hive est un SGBD qui offre un langage ressemblant à SQL et qui s'appuie sur HBase.

API Java de HBASE

Introduction

On peut écrire des programmes Java qui accèdent aux tables HBase. Il y a un petit nombre de classes et de méthodes à connaître pour démarrer.

Nous allons voir comment :

- créer une table
- ajouter des cellules
- récupérer une cellule
- parcourir les cellules

Noter qu'on se bat contre le temps, HBase évolue très vite et de nombreux aspects deviennent rapidement obsolètes (*deprecated*). L'API était en 0.98.12 cette année (hadoop 2.8.4), maintenant c'est la 2.0 et la 3.0 est déjà annoncée.

Imports communs

Pour commencer, les classes à importer :



```
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.hbase.*;
import org.apache.hadoop.hbase.client.*;
import org.apache.hadoop.hbase.util.*;
```

Ensuite, chaque programme contient ces lignes (API 0.98) qui établissent une connexion avec le serveur HBase :



```
Configuration config = HBaseConfiguration.create();
HBaseAdmin admin = new HBaseAdmin(config);
try {
    ...
} finally {
    admin.close();
}
```

Création d'une table


Voici une fonction qui crée une table :



```
static void CreerTable(HBaseAdmin admin,
                      String nomtable, String... familles)
    throws IOException
{
    TableName tn = TableName.valueOf(nomtable);
    HTableDescriptor htd = new HTableDescriptor(tn);
    for (String famille: familles) {
        htd.addFamily(new HColumnDescriptor(famille));
    }
    admin.createTable(htd);
}
```


Notez que j'ai fourni le `HBaseAdmin` en paramètre et que les familles sont sous forme de *varargs*.

Suppression d'une table

Voici comment on supprime une table, en vérifiant au préalable si elle existe : 

```
static void SupprimerTable(HBaseAdmin admin, String nomtable)
{
    TableName tn = TableName.valueOf(nomtable);
    if (admin.tableExists(tn)) {
        admin.disableTable(tn);
        admin.deleteTable(tn);
    }
}
```

Manipulation d'une table

Dans les fonctions suivantes, on va modifier les données d'une table existante. Pour cela, il faut récupérer un objet `HTable` représentant la table. Il est important de libérer cet objet dès qu'il ne sert plus. Voici comment faire en API 0.98 : 

```
static void OperationSurTable(Configuration config,
                             String nomtable, ...)
{
    HTable table = new HTable(config, nomtable);
    try {
        ... opérations sur le contenu de la table ...
    } finally {
        table.close();
    }
}
```

Prévoir aussi l'arrivée de `IOException` à tout moment.

Insertion d'une valeur

L'insertion d'une valeur consiste à créer une instance de la classe Put. Cet objet spécifie la valeur à insérer :

- identifiant du n-uplet auquel elle appartient
- nom de la famille
- nom de la colonne
- valeur
- en option, le timestamp à lui affecter.

Toutes les données concernées doivent être converties en tableaux d'octets.

Transformation en tableaux d'octets

HBase stocke des données binaires quelconques : chaînes, nombres, images jpg, etc. Il faut seulement les convertir en `byte []`.

Convertir une donnée en octets se fait quelque soit son type par :

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

Dans le cas de clés de n-uplets de type nombre (int, long, float et double), le classement des clés sera fantaisiste à cause de la représentation interne, voir [cette page](#). Du fait que le signe du nombre soit en tête et vaut 1 pour les nombres négatifs, 0 pour les nombres positifs, un nombre négatif sera considéré comme plus grand qu'un positif.

Il est possible d'y remédier en trafiquant le tableau d'octets afin d'inverser le signe mais c'est hors sujet.

Transformation inverse

La récupération des données à partir des octets n'est pas uniforme. Il faut impérativement connaître le type de la donnée pour la récupérer correctement. Il existe plusieurs fonctions, voir [la doc](#) :

```
String chaine = Bytes.toString(octets);  
Double nombre = Bytes.toDouble(octets);  
Long entier = Bytes.toLong(octets);
```

Dans certains cas, HBase nous retourne un grand tableau d'octets dans lequel nous devons piocher ceux qui nous intéressent. Nous avons donc trois informations : le tableau, l'offset du premier octet utile et le nombre d'octets. Il faut alors faire ainsi :

```
Double nombre = Bytes.toDouble(octets, debut, taille);
```

Insertion d'une valeur, fonction



```
static void AjouterValeur(Configuration config,
    String nomtable, String id,
    String fam, String col, String val)
{
    HTable table = new HTable(config, nomtable);
    try {
        // construire un Put
        final byte[] rawid = Bytes.toBytes(id);
        Put action = new Put(rawid);
        final byte[] rawfam = Bytes.toBytes(fam);
        final byte[] rawcol = Bytes.toBytes(col);
        final byte[] rawval = Bytes.toBytes(val);
        action.add(rawfam, rawcol, rawval);
        // effectuer l'ajout dans la table
        table.put(action);
    } finally { table.close(); }
```

Insertion d'une valeur, critique

- Le problème de la fonction précédente, c'est qu'on a ajouté une valeur de type chaîne. Il faut écrire une autre fonction pour ajouter un entier, un réel, etc.
Il faudrait réfléchir à une fonction un peu plus générale, à laquelle on peut fournir une donnée quelconque et qui l'ajoute correctement en binaire. C'est pas tout à fait trivial, car la méthode `Bytes.toBytes` n'accepte pas le type `Object` en paramètre.
- Il ne faut pas insérer de nombreuses valeurs une par une avec la méthode `table.put`. Utiliser la surcharge `table.put(ArrayList<Put> liste)`.


Extraire une valeur

La récupération d'une cellule fait appel à un `Get`. Il se construit avec l'identifiant du n-uplet voulu. Ensuite, on applique ce `Get` à la table. Elle retourne un `Result` contenant les cellules du n-uplet.


```
static void AfficherNuplet(Configuration config,
                          String nomtable, String id)
{
    final byte[] rawid = Bytes.toBytes(id);
    Get action = new Get(rawid);
    // appliquer le get à la table
    HTable table = new HTable(config, nomtable);
    try {
        Result result = table.get(action);
        AfficherResult(result);
    } finally { table.close(); }
}
```

Résultat d'un Get

Un **Result** est une sorte de dictionnaire (famille,colonne)→valeur


- Sa méthode `getValue(famille, colonne)` retourne les octets de la valeur désignée, s'il y en a une : 

```
byte[] octets = result.getValue(rawfam, rawcol);
```

- On peut parcourir toutes les cellules par une boucle : 


```
void AfficherResult(Result result)
{
    for (Cell cell: result.listCells()) {
        AfficherCell(cell);
    }
}
```

Affichage d'une cellule

C'est un peu lourdingue car il faut extraire les données de tableaux d'octets avec offset et taille, et attention si le type n'est pas une chaîne. 

```
static void AfficherCell(Cell cell)
{
    // extraire la famille
    String fam = Bytes.toString(cell.getFamilyArray(),
        cell.getFamilyOffset(), cell.getFamilyLength());
    // extraire la colonne (qualifier)
    String col = Bytes.toString(cell.getQualifierArray(),
        cell.getQualifierOffset(), cell.getQualifierLength());
    // extraire la valeur (PB si c'est pas un String)
    String val = Bytes.toString(cell.getValueArray(),
        cell.getValueOffset(), cell.getValueLength());
    System.out.println(fam+": "+col+" = "+val);
}
```

Parcours des n-uplets d'une table

Réaliser un Scan par programme n'est pas très compliqué. Il faut fournir la clé de départ et celle d'arrêt (ou alors le scan se fait sur toute la table). On reçoit une énumération de Result. 

```
static void ParcourirTable(Configuration config,
                          String nomtable, String start, String stop)
{
    final byte[] rawstart = Bytes.toBytes(start);
    final byte[] rawstop = Bytes.toBytes(stop);
    Scan action = new Scan(rawstart, rawstop);

    HTable table = new HTable(config, nomtable);
    ResultScanner results = table.getScanner(action);
    for (Result result: results) {
        AfficherResult(result);
    }
}
```


Paramétrage d'un Scan

Il est possible de filtrer le scan pour se limiter à :

- certaines familles et colonnes (on peut en demander plusieurs à la fois), sinon par défaut, c'est toutes les colonnes.

```
action.add(rawfam, rawcol);
```

- rajouter des filtres sur les valeurs, par exemple ci-dessous, on cherche les colonnes supérieures ou égales à une limite.

NB: Il faut utiliser la classe CompareOp et BinaryComparator

```
BinaryComparator binlimite = new BinaryComparator(rawlimite);  
SingleColumnValueFilter filtre = new SingleColumnValueFilter(  
    rawfam, rawcol, CompareOp.GREATER_OR_EQUAL, binlimite);  
action.setFilter(filtre);
```

Filtrage d'un Scan

Pour définir une condition complexe, il faut construire un `FilterList` qu'on attribue au `Scan`. Un `FilterList` représente soit un «et» (`MUST_PASS_ALL`), soit un «ou» (`MUST_PASS_ONE`). Les `FilterList` peuvent être imbriqués pour faire des combinaisons complexes.

```
SingleColumnValueFilter filtre1 = ...
QualifierFilter filtre2 = ...

FilterList conjonction = new FilterList(
    FilterList.Operator.MUST_PASS_ALL, filtre1, filtre2);
action.setFilter(conjonction);
```

Attention aux comparaisons de nombres. Elles sont basées sur la comparaison des octets internes, or généralement, les nombres négatifs ont un poids fort plus grand que celui des nombres positifs.

Hive

Présentation rapide

Hive simplifie le travail avec une base de données comme HBase ou des fichiers CSV. Hive permet d'écrire des requêtes dans un langage inspiré de SQL et appelé HiveQL. Ces requêtes sont transformées en jobs MapReduce.

Pour travailler, il suffit définir un schéma qui est associé aux données. Ce schéma donne les noms et types des colonnes, et structure les informations en tables exploitables par HiveQL.

Définition d'un schéma

Le schéma d'une table est également appelé méta-données (c'est à dire informations sur les données). Les métadonnées sont stockées dans une base de données MySQL, appelée *metastore*.

Voici la définition d'une table avec son schéma :

```
CREATE TABLE releves (  
    idreleve STRING,  
    annee INT, ...  
    temperature FLOAT, quality BYTE,  
    ...)  
ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t';
```

Le début est classique, sauf les contraintes d'intégrité : il n'y en a pas. La fin de la requête indique que les données sont dans un fichier CSV. Voyons d'abord les types des colonnes.

Types HiveQL

Hive définit les types suivants :

- BIGINT (8 octets), INT (4), SMALLINT (2), BYTE (1 octet)
- FLOAT et DOUBLE
- BOOLEAN valant TRUE ou FALSE
- STRING, on peut spécifier le codage (UTF8 ou autre)
- TIMESTAMP exprimé en nombre de secondes.nanosecondes depuis le 01/01/1970 UTC
- données structurées comme avec Pig :
 - ARRAY<type> indique qu'il y a une liste de *type*
 - STRUCT<nom1:type1, nom2:type2...> pour une structure regroupant plusieurs valeurs
 - MAP<typecle, typeval> pour une suite de paires clé,valeur

Séparations des champs pour la lecture

La création d'une table se fait ainsi :

```
CREATE TABLE nom (schéma) ROW FORMAT DELIMITED descr du format
```

Les directives situées après le schéma indiquent la manière dont les données sont stockées dans le fichier CSV. Ce sont :

- `FIELDS TERMINATED BY ';' :` il y a un ; pour séparer les champs
- `COLLECTION ITEMS TERMINATED BY ',' :` il y a un , entre les éléments d'un ARRAY
- `MAP KEYS TERMINATED BY ':' :` il y a un : entre les clés et les valeurs d'un MAP
- `LINES TERMINATED BY '\n' :` il y a un \n en fin de ligne
- `STORED AS TEXTFILE :` c'est un CSV.

Chargement des données

Voici comment charger un fichier CSV qui se trouve sur HDFS, dans la table :

```
LOAD DATA INPATH '/share/noaa/data/186293'  
OVERWRITE INTO TABLE releves;
```

NB: le problème est que Hive **déplace** le fichier CSV dans ses propres dossiers, afin de ne pas dupliquer les données. Sinon, on peut écrire `CREATE EXTERNAL TABLE ...` pour empêcher Hive de capturer le fichier.

On peut aussi charger un fichier local (pas HDFS) :

```
LOAD DATA LOCAL INPATH 'stations.csv'  
OVERWRITE INTO TABLE stations;
```

Le fichier est alors copié sur HDFS dans les dossiers de Hive.

Liens entre HBase et Hive

Il est également possible d'employer une table HBase.

Cela fait appel à la notion de gestionnaire de stockage (*Storage Handler*). C'est simplement une classe générale qui gère la lecture des données. Elle a des sous-classes pour différents systèmes, dont `HBaseStorageHandler` pour HBase.

```
CREATE TABLE stations(idst INT, name STRING, ..., lat FLOAT,...)
STORED BY 'org.apache.hadoop.hive.hbase.HBaseStorageHandler'
WITH
SERDEPROPERTIES("hbase.columns.mapping" = ":dst,data:name,...")
TBLPROPERTIES("hbase.table.name" = "stations");
```

La clause `SERDEPROPERTIES` (serialisation/désérialisation) associe les noms des colonnes HBase à ceux de la table Hive.

Requêtes HiveQL

Comme avec les SGBD conventionnels, il y a un shell lancé par la commande `hive`. C'est là qu'on tape les requêtes SQL. Ce sont principalement des `SELECT`. Toutes les clauses que vous connaissez sont disponibles : `FROM`, `JOIN`, `WHERE`, `GROUP BY`, `HAVING`, `ORDER BY`, `LIMIT`.

Il y en a d'autres pour optimiser le travail MapReduce sous-jacent, par exemple quand vous voulez classer sur une colonne, il faut écrire :

```
SELECT... DISTRIBUTE BY colonne SORT BY colonne;
```

La directive envoie les n-uplets concernés sur une seule machine afin de les comparer plus rapidement pour établir le classement.

Autres directives

Il est également possible d'exporter des résultats dans un dossier :

```
INSERT OVERWRITE LOCAL DIRECTORY '/tmp/meteo/chaud'  
SELECT annee,mois,jour,temperature  
FROM releves  
WHERE temperature > 40.0;
```

Parmi les quelques autres commandes, il y a :

- `SHOW TABLES;` pour afficher la liste des tables (elles sont dans le metastore).
- `DESCRIBE EXTENDED table;` affiche le schéma de la table