# 1. HBase en ligne de commande

On va découvrir la base de données NoSQL HBase à l'aide de son shell. Vous allez être dérouté(e) par la syntaxe employée ; c'est celle de Ruby.

Commencez par créer un dossier TP6.

## 1.1. Premières commandes

### 1.1.1. Commandes de base

- Connectez-vous sur la machine `hadoop` par `ssh`.
- Lancez le shell de HBase en tapant `hbase shell`
- Tapez ces commandes :
  - `status`
  - `version`
  - `whoami`
  - `list`
  - `exit`

### 1.1.2. Création d'une table

La première manipulation consiste à créer une table, ajouter des données et supprimer cette table. Alors le plus gros problème que nous avons à résoudre est le nom de la table. En effet tel qu'il est configuré à l'IUT, HBase ne crée pas de frontières entre les utilisateurs. Chacun peut utiliser les données créées par d'autres utilisateurs. Vous allez tous travailler ensemble, alors il faudra que chacun crée des tables à son propre nom. Donc dans la suite, vous devrez systématiquement mettre votre login en préfixe de toutes les tables. Par exemple, je dois remplacer `bibliotheque` par `nerzicpiEcrivains`.

Pour être sûr que vous allez bien faire cela, voici un script à lancer, il lance les commandes dans HBase en ayant affecté une variable Ruby avec le nom de votre table :

```
#!/bin/sh
cat <<CMDES | hbase shell
bibliotheque='${LOGNAME}Bibliotheque'

create bibliotheque,
    { NAME=>'auteur', VERSIONS=>2 }, { NAME=>'livre', VERSIONS=>3 }

list
describe bibliotheque
CMDES
```

La première ligne définit une variable Ruby bibliotheque valant le nom de la table. Ici, on a indiqué combien de versions on voulait garder pour chaque valeur, mais quand on n'en veut qu'une, on met directement le nom de la famille : `create bibliotheque, 'auteur', 'livre'`.

Cette commande crée une table avec deux familles. Les familles sont comme des *namespaces* pour les colonnes. Également, on peut spécifier le nombre de versions à mémoriser par famille.

Allez voir dans la page HBase, la liste des tables créées. Vous pouvez cliquer sur la table pour en avoir plus de détails techniques. Vous verrez que votre table est prise en charge par l'un des *Region Servers*.

### 1.1.3. Ajout de valeurs

Voici la suite des commandes avec le rajout de données. L'ajout se fait valeur par valeur, et non pas par n-uplets entiers. La syntaxe de la commande `put` est `put 'table' 'clé' 'famille:colonne' valeur`. Il faut mettre des `'...'` pour toutes les chaînes.

Voici les commandes : 

```
#!/bin/sh
cat <<CMDES | hbase shell
bibliotheque='${LOGNAME}Bibliotheque'

put bibliotheque, 'vhugo', 'auteur:nom', 'Hugo'
put bibliotheque, 'vhugo', 'auteur:prenom', 'Victor'
put bibliotheque, 'vhugo', 'livre:titre', 'La Légende des siècles'
put bibliotheque, 'vhugo', 'livre:categ', 'Poèmes'
put bibliotheque, 'vhugo', 'livre:date', 1855
put bibliotheque, 'vhugo', 'livre:date', 1877
put bibliotheque, 'vhugo', 'livre:date', 1883

put bibliotheque, 'jverne', 'auteur:prenom', 'Jules'
put bibliotheque, 'jverne', 'auteur:nom', 'Verne'
put bibliotheque, 'jverne', 'livre:editeur', 'Hetzel'
put bibliotheque, 'jverne', 'livre:titre', 'Face au drapeau'
put bibliotheque, 'jverne', 'livre:date', 1896
CMDES
```

Vous constaterez que l'ajout des données est très rapide. Seul le lancement du shell HBase est lent.

Toutes ces instructions n'ont créé que deux n-uplets en tout. Le premier est identifié par `'vhugo'` et le second par `'jverne'` (identifiants très mal choisis s'il fallait ajouter d'autres livres de ces auteurs). Vous voyez aussi que ces n-uplets n'ont pas tous les mêmes colonnes.

### 1.1.4. Comptage des valeurs

HBase ne dispose pas de nombreuses fonctions. C'est une énorme table de hachage de paires <*clé*, *valeur*>, extrêmement efficace, mais c'est tout. On peut seulement les compter et faire des recherches dessus. Voici comment compter les n-uplets : 

```
#!/bin/sh
cat <<CMDES | hbase shell
bibliotheque='${LOGNAME}Bibliotheque'
count bibliotheque
CMDES
```

Quand la table est énorme, il faut spécifier une taille de cache : 

```
#!/bin/sh
cat <<CMDES | hbase shell
bibliotheque='${LOGNAME}Bibliotheque'
count bibliotheque, CACHE=>1000
CMDES
```

### 1.1.5. Récupération de valeurs

On passe maintenant à l'affichage : 

```
#!/bin/sh
cat <<CMDES | hbase shell
bibliotheque='${LOGNAME}Bibliotheque'

get bibliotheque, 'vhugo'
get bibliotheque, 'vhugo', 'auteur'
get bibliotheque, 'vhugo', 'auteur:prenom'
get bibliotheque, 'jverne', {COLUMN=>'livre'}
get bibliotheque, 'jverne', {COLUMN=>'livre:titre'}
get bibliotheque, 'jverne',
        {COLUMN=>['livre:titre', 'livre:date', 'livre:editeur']}
get bibliotheque, 'jverne', {FILTER=>"ValueFilter(=, 'binary:Jules')"}
CMDES
```

On doit fournit l'identifiant du n-uplet à la commande `get`. Ça affiche toutes les colonnes du n-uplet. Ensuite, on peut rajouter des propriétés telles que le nom de la famille, le nom de la colonne (avec deux syntaxes possibles) ou des filtres. Ce filtre signifie : quelle est la colonne dont la valeur vaut 'Jules'. Par contre, les filtres sont particulièrement difficile à écrire et ne marchent pas toujours bien, on se limitera à une simple comparaison comme ici.

### 1.1.6. Parcours des n-uplets

La commande `get` ne peut retourner qu'un seul n-uplet. Voici une autre façon de récupérer les informations, avec la commande `scan` : 

```
#!/bin/sh
cat <<CMDES | hbase shell
bibliotheque='${LOGNAME}Bibliotheque'

scan bibliotheque
scan bibliotheque, {COLUMNS=>['livre']}
scan bibliotheque, {COLUMNS=>['livre:date']}
CMDES
```

Chaque exemple du code précédent est de plus en plus spécifique. Le premier `scan` affiche toutes les données de la table. Le deuxième n'affiche que les données de la famille `livre`. Le troisième `scan` affiche seulement les valeurs `date` de la famille `livre` présentes dans la table.

Il est possible d'exprimer des conditions plus fines : 

```
#!/bin/sh
cat <<CMDES | hbase shell
bibliotheque='${LOGNAME}Bibliotheque'

scan bibliotheque, { STARTROW=>'a', STOPROW=>'n', COLUMNS=>'auteur'}
scan bibliotheque,
    {FILTER=>"RowFilter(>=, 'binary:a') AND RowFilter(<, 'binary:n') AND
              FamilyFilter (=, 'binary:auteur')"}
scan bibliotheque,
    {FILTER=>"FamilyFilter (=, 'binary:auteur') AND
              QualifierFilter (=, 'binary:prenom')"}
scan bibliotheque,
    {FILTER=>"SingleColumnValueFilter('livre', 'titre', =, 'binary:Face au drapeau')"}
scan bibliotheque,
    {FILTER=>"SingleColumnValueFilter('livre', 'date', <=, 'binary:1890')"}
scan bibliotheque,
    {FILTER=>"PrefixFilter('jv') AND ValueFilter(=,'regexstring:[A-Z]([a-z]+ ){2,}')"}
CMDES
```

Le premier `scan` parcourt les n-uplets par clés comprises entre a et n et les champs de la famille `auteur`. Le deuxième fait exactement pareil, mais avec un filtre. Le troisième `scan` affiche les valeurs `auteur:prenom`. Le quatrième `scan` cherche les colonnes dont la valeur vaut le titre indiqué. Le cinquième affiche les n-uplets dont la (dernière version de la) colonne `livre:date` est inférieure ou égale à 1890. Le dernier filtre est plus complexe, il cherche les n-uplets dont la clé commence par "jv" et dont l'une des valeurs correspond à l'expression régulière, à vous de vous souvenir comment on les écrit. La liste des jokers est sur cette page.

- Notez que tous ces `scans` retournent à chaque fois le n-uplet complet, toutes ses colonnes, puisqu'on ne les a pas limitées avec `COLUMNS`.
- Attention à bien orthographier les champs, sinon tous les n-uplets sont sélectionnés.
- Ne pas mélanger une condition type `FILTER` avec une condition `COLUMNS`, ça ne marche pas du tout.
- Ne pas oublier `binary:` devant les constantes à comparer.

### 1.1.7. Mise à jour d'une valeur

On va s'intéresser aux versions des données. La table a été créée pour garder 2 versions des valeurs de la famille `auteur` et 3 de la famille `livre`.

```
#!/bin/sh
cat <<CMDES | hbase shell
bibliotheque='${LOGNAME}Bibliotheque'

put bibliotheque, 'vhugo', 'auteur:nom', 'HAGO'
put bibliotheque, 'vhugo', 'auteur:nom', 'HUGO'
put bibliotheque, 'vhugo', 'auteur:prenom', 'Victor Marie'
put bibliotheque, 'vhugo', 'auteur:nom', 'Hugo'
```

```
get bibliotheque, 'vhugo', 'auteur'
get bibliotheque, 'vhugo', {COLUMNS=>'auteur'}
get bibliotheque, 'vhugo', {COLUMNS=>'auteur', VERSIONS=>10}
CMDES
```

Les versions sont indiquées avec leur *timestamp*. Malheureusement, il ne semble pas possible de le manipuler sous forme d'une date lisible.

### 1.1.8. Suppression d'une valeur ou d'une colonne

Il y a plusieurs types d'informations qui peuvent être supprimées : valeur, colonne et famille entière. Si on supprime une valeur, ça supprime aussi les valeurs plus anciennes.

Dans l'exemple suivant, vous allez devoir saisir le *timestamp* de la valeur `HUGO` pour le nom. Recherchez-la dans le dernier get.

```
#!/bin/sh
read -p 'Saisissez le timestamp de vhugo auteur:nom à supprimer :' TIMESTAMP
cat <<CMDES | hbase shell
bibliotheque='${LOGNAME}Bibliotheque'

deleteall bibliotheque, 'vhugo', 'auteur:nom', ${TIMESTAMP}
deleteall bibliotheque, 'vhugo', 'auteur:prenom'
deleteall bibliotheque, 'jverne'

scan bibliotheque, {VERSIONS=>10}
CMDES
```

Le premier `deleteall` a supprimé la valeur `auteur:nom=HUGO`, mais pas l'autre (sauf si vous vous êtes trompé de *timestamp*). Le deuxième a supprimé puis toutes les valeurs pour la colonne `prenom`. Le dernier `deleteall` a supprimé le n-uplet entier.

Il y a une autre commande, `delete`, mais au lieu de supprimer une information, elle place une valeur spéciale la marquant en tant qu'effacée.

### 1.1.9. Suppression d'une table

Supprimer une table demande de d'abord la désactiver sur les serveurs de région.

```
#!/bin/sh
cat <<CMDES | hbase shell
bibliotheque='${LOGNAME}Bibliotheque'
disable bibliotheque
drop bibliotheque
CMDES
```

## 1.2. Exercices à faire

Vous allez travailler sur le fichier des arbres, préalablement inséré dans HBase.

### 1.2.1. Insertion des données

Le but est d'insérer le fichier `/share/paris/arbres.csv` sous forme d'une table HBase puis de faire quelques interrogations simples. C'est un fichier CSV. Sa première ligne donne les noms des champs :

```
GEOPOINT;ARRONDISSEMENT;GENRE;ESPECE;FAMILLE;ANNEE PLANTATION;HAUTEUR
CIRCONFERENCE;ADRESSE;NOM COMMUN;VARIETE;OBJECTID;NOM_EV
```

Le champ `OBJECTID`:12 est l'identifiant des n-uplets. On va regrouper les autres informations en trois familles (l'indice du champ est mis après le nom) :

- `GENRE`:3, `ESPECE`:4, `FAMILLE`:5, `NOM COMMUN`:10, `VARIETE`:11 dans la famille `genre`
- `ANNEE PLANTATION`:6, `HAUTEUR`:7, `CIRCONFERENCE`:8 dans la famille `infos`
- `GEOPOINT`:1, `ARRONDISSEMENT`:2, `ADRESSE`:9, `NOM_EV`:13 dans la famille `adresse`

L'idée est, pour ne pas galérer d'écrire un programme qui va insérer les données. Comme le fichier est court, on peut employer un langage de scripts comme bash, awk ou Python. Son principe est de lire le fichier ligne par ligne en ignorant la première. Chaque ligne est découpée en mots. Les mots sont regroupés par familles et insérés dans la table. Le plus simple et le plus facilement configurable est en Python, `arbres2hbase.py` :

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

# traite le fichier /share/paris/arbres.csv et affiche les commandes HBase à faire
# le fichier doit arriver sur stdin, les commandes sortent sur stdout
# donc, lancer par
#       hdfs dfs -cat /share/paris/arbres.csv | python ./arbres2hbase.py | hbase shell

# dictionnaire des noms et indices des colonnes par famille
familles = {
    'genre':   {'genre':3, 'espece':4, 'famille':5, 'nom_commun':10, 'variete':11},
    'infos':   {'annee_plantation':6, 'hauteur':7, 'circonference':8},
    'adresse': {'geopoint':1, 'arrondissement':2, 'adresse':9, 'nom_ev':13}}


from sys import stdin
from os import getenv

# définir la variable arbres avec le LOGNAME
print "arbres='%sArbres'" % getenv('LOGNAME')

# créer la table
print "create arbres, %s" % (", ".join(["'%s'"%famille for famille in familles]))

for ligne in stdin:
    ligne = ligne.strip()
```

```
    if not ligne.startswith('('): continue
    mots = ligne.split(';')

    # identifiant
    id = "arbre-%02d" % int(mots[12 - 1])

    # produire les cellules
    for famille in familles:
        for colonne in familles[famille]:
            numero = familles[famille][colonne] - 1
            valeur = mots[numero].replace("'","_")
            if not valeur: continue
            print "put arbres, '%s', '%s:%s', '%s'" % (id, famille, colonne, valeur)

    # affichage de la table à la fin
    print "scan arbres"
```

Si vous avez fait *enregistrer sous...*, vous devrez supprimer le BOM unicode avec geany. N'oubliez pas de le rendre exécutable. Ce script produit des lignes qu'il suffit d'envoyer dans HBase shell par cette simple commande (avant de la lancer en entier, visualisez ce que chaque commande fait) :

```
hdfs dfs -cat /share/paris/arbres.csv | python ./arbres2hbase.py | hbase shell
```

Ne recopiez pas les lignes produites dans `tp6_partie1.sh`, mais seulement cette commande, mais mettez-la en commentaire pour les essais.

Ce script met quelques secondes à s'exécuter. Vous imaginez ce que ça serait dans le cas d'un énorme fichier. Il existe des solutions, Avro et Parquet, que nous n'avons pas le temps de voir dans ce cours qui permettent de lire ce genre de fichiers très rapidement.

### 1.2.2. Travail à rendre

Vous écrirez les commandes à faire dans un script `tp6_partie1.sh` comme ceux des essais précédents. La dernière commande de ce script devra supprimer la table afin que rien ne reste sur le serveur.

```
#!/bin/sh

# insertion des données (à ne faire qu'une fois, commenter après)
#hdfs dfs -cat /share/paris/arbres.csv | ./arbres2hbase.py | hbase shell

# réponses pour les exercices sur HBase
cat <<CMDES | hbase shell
arbres='${LOGNAME}Arbres'

# question 1
get arbres, '...
```

```
... vos commandes pour le shell HBase ...

CMDES
```

#### 1.2.3. Recherche d'informations

Voici quelques requêtes à coder en HBase

- Afficher le genre de l'arbre `arbre-82`.
- Afficher les valeurs de la famille `infos` de l'arbre `arbre-10`.
- Afficher l'année de plantation des arbres dont le champ `nom_ev` vaut « Parc Montsouris ».
- Afficher la hauteur des arbres dont le genre est "Quercus".
- Afficher les noms communs des arbres du 13e arrondissement.
- Afficher la hauteur des arbres plantés avant l'année 1800. C'est un problème difficile à cause des valeurs absentes. Pour ne pas produire les valeurs absentes, il faut rajouter `, true, true` en paramètres supplémentaires de `SingleColumnValueFilter`.

# 2. Fonctions de l'API HBase

On va maintenant étudier l'API qui permet de programmer HBase en Java. Préparez un nouveau projet Eclipse Java appelé `DemoHBase` dans le dossier TP6.

Exécutez les commandes suivantes afin de copier les archives nécessaires à la compilation.

```
mkdir -p ~/lib/hbase
cp /usr/lib/hbase/hbase-common-0.98.12-hadoop2.jar ~/lib/hbase
cp /usr/lib/hbase/hbase-client-0.98.12-hadoop2.jar ~/lib/hbase
cp /usr/lib/hadoop/client/commons-logging.jar ~/lib/hbase
cp /usr/lib/hadoop/hadoop-common-2.8.4.jar ~/lib/hbase
cp /usr/lib/hadoop/client/hadoop-common-2.8.4.jar ~/lib/hbase
```

Dans Eclipse, modifiez les propriétés du projet : clic droit sur le projet, item `Properties`, onglet `Java Build Path`, sous-onglet `Libraries`, cliquez sur le bouton `Add External Jars...`, allez dans le dossier `lib/hbase` de votre compte, sélectionnez tous les éléments et validez. Fermez le dialogue. Pour info, ces chemins sont enregistrés dans un fichier caché appelé `.classpath` dans le projet.

Comme avec YARN, vous ne pourrez qu'éditer le programme avec Eclipse. Pour l'exécution, il suffira de taper `make` dans le dossier du projet, en étant sur la machine `hadoop`.

## 2.1. Découverte de l'API HBase

Téléchargez le projet DemoHBase.zip et allez l'extraire dans le projet Eclipse. Il comprend un source et un Makefile. C'est le programme montré dans le cours de la semaine 7.

Votre travail consiste dans un premier temps à regarder comment il fonctionne. Regardez tous les appels aux méthodes HBase. Vous voyez qu'il y a une anomalie d'affichage des valeurs entières. C'est parce que les entiers sont stockés sous forme binaire. Il faudrait le savoir pour les décoder

correctement lors de l'affichage. Décommentez ce qui permet d'afficher le bon résultat dans la méthode `AfficherCell`.

Dans un second temps, mais c'est seulement une idée de projet, il faudrait réorganiser cet exemple en deux classes. Il y aurait la classe principale, celle qui contient `public static void main()` et une classe appelée `TableHBase` qui contiendrait des méthodes générales afin de facilement programmer le patron de conception *Active Records*. La classe `TableHBase` serait une superclasse pour différentes sous-classes, par exemple `Annuaire`. L'objectif est, fonction du temps restant, d'y réfléchir. Vous pouvez par exemple y réfléchir pour la réalisation du projet suivant, le raccourcisseur d'URL.

## 2.2. Raccourcisseur d'URL

On va créer un service de mémorisation d'URL. Soit un URL assez long comme celui de cet énoncé de TP. On se propose de le simplifier en, par exemple : http://co.urt/I8hCz7 (il faudrait acheter ce domaine). Ce mécanisme est déjà proposé par plusieurs entreprises : [https://goo.gl/], [https://bitly.com/], [http://ow.ly/url/shorten-url], avec des services supplémentaires (vérification périodique que le lien est vivant...).

Le principe est de gérer une table HBase pour contenir les associations entre l'URL court et l'URL long. Lorsqu'un utilisateur crée un lien, on ajoute une nouvelle paire dans la table. Lorsqu'un utilisateur demande le lien, on va le chercher avec un simple get. Il y a deux aspects : génération d'un URL court et transformation d'un URL court en URL long.

Pour créer un URL court, une idée simple est d'utiliser un entier séquentiel. 1 pour le premier lien, 2 pour le deuxième et ainsi de suite. Le problème, c'est qu'on veut éviter que n'importe qui puisse facilement consulter les liens enregistrés. Donc on va les brouiller, les transformer en chaîne comme « I8hCz7 ».

Téléchargez le fichier Hashids.java qui provient du projet hashids.org et placez-le dans un sous-dossier `HashidsJava` de vos sources. C'est un mécanisme pour transformer un entier en code brouillé. Pour générer une clé de 6 caractères à partir de ce numéro, il suffit de :

```
long numero;
Hashids hashids = new Hashids("un texte quelconque secret mais constant", 6);
String cle = hashids.encrypt(numero);
```

Ensuite, on doit seulement ajouter une paire (clé, URLlong) dans la table.

Votre travail consiste à programmer deux programmes Java, probablement deux projets. Le premier permet d'ajouter un nouvel URL long et de récupérer le code de l'URL court. Le second permet de faire l'inverse : fournir l'URL long à partir du code court. Les deux stockent toutes leurs informations dans HBase. Vous devrez stocker le compteur dans une table afin de pouvoir l'incrémenter d'un appel à l'autre.

# 3. Travail à rendre

Supprimez les dossiers bin de vos projets Java.

Compresser le dossier tp6 en `tp6.tar.gz`. Déposez l'archive sur Moodle.