

# **HIVE**



## **Le Data Warehouse de Hadoop**

# Hive: Introduction

---

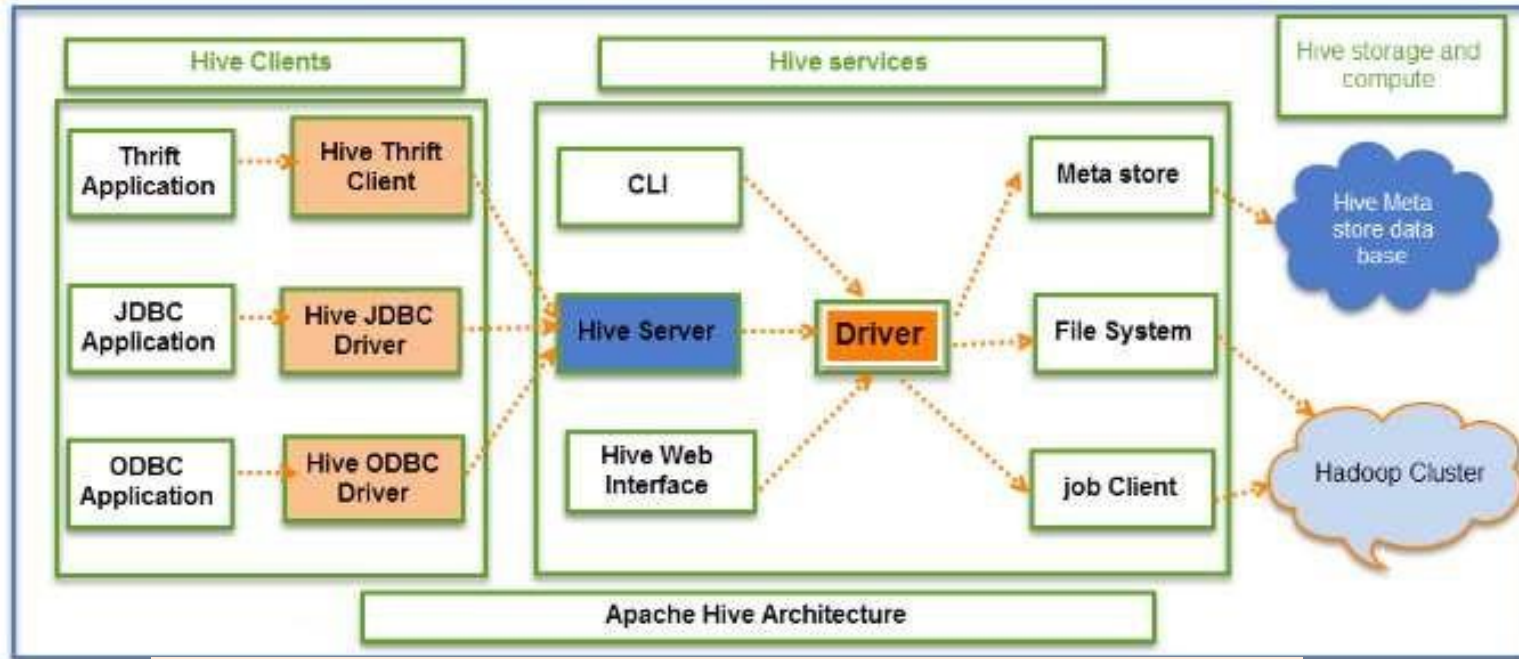
- **Hive** est un projet né au sein de **Facebook** et repris par la **fondation Apache**.
- Apache Hive représente une **surcouche analytique** à Hadoop
- Il représente aussi une **couche d'abstraction** aux données HDFS sous forme d'un **modèle tabulaire** (*lignes, colonnes*)
- Hive permet de traiter des données structurées dans Hadoop.
- Il est utilisé par différentes entreprises, par exemple, **Amazon** l'utilise dans **Amazon Elastic MapReduce** (*outil AWS de traitement et d'analyse de données*)
- Il a un langage de requête **HiveQL** ou **HQL** proche de SQL (*sélection, jointure, agrégation, union, sous-requêtes*)
- La principale différence entre **HiveQL** et **SQL** est qu'une requête Hive s'exécute sur Hadoop plutôt que sur une BD relationnelle.
- Hive prend en charge les fonctions et procédures **java/scala** définies par l'utilisateur pour étendre ses fonctionnalités.

# Hive: Introduction

---

- Une requête **HiveQL** est transformée en une série de jobs **MapReduce** exécutés sur le cluster Hadoop afin de récupérer un jeu de résultat tabulaire.
- **HiveQL** décharge l'utilisateur de la complexité de la programmation MapReduce. Il réutilise les concepts familiers de BD relationnelle (*tables, lignes, colonnes, schémas, etc*) pour faciliter l'apprentissage.
- Hive supporte plusieurs formats de fichiers: **TEXTFILE**, **SEQUENCEFILE**, **ORC**, **RCFILE** (*Record Columnar File*), **PARQUET**, ...etc.
- Les **schémas de BDs** sont gérés par le **MetaStore** qui englobe un service qui permet aux clients (*applications clientes*) d'avoir les **métadonnées** des tables déclarées dans Hive.
- Le **metastore** persiste ces métadonnées dans une BD **Apache Derby** (*SGBD développé en Java*).

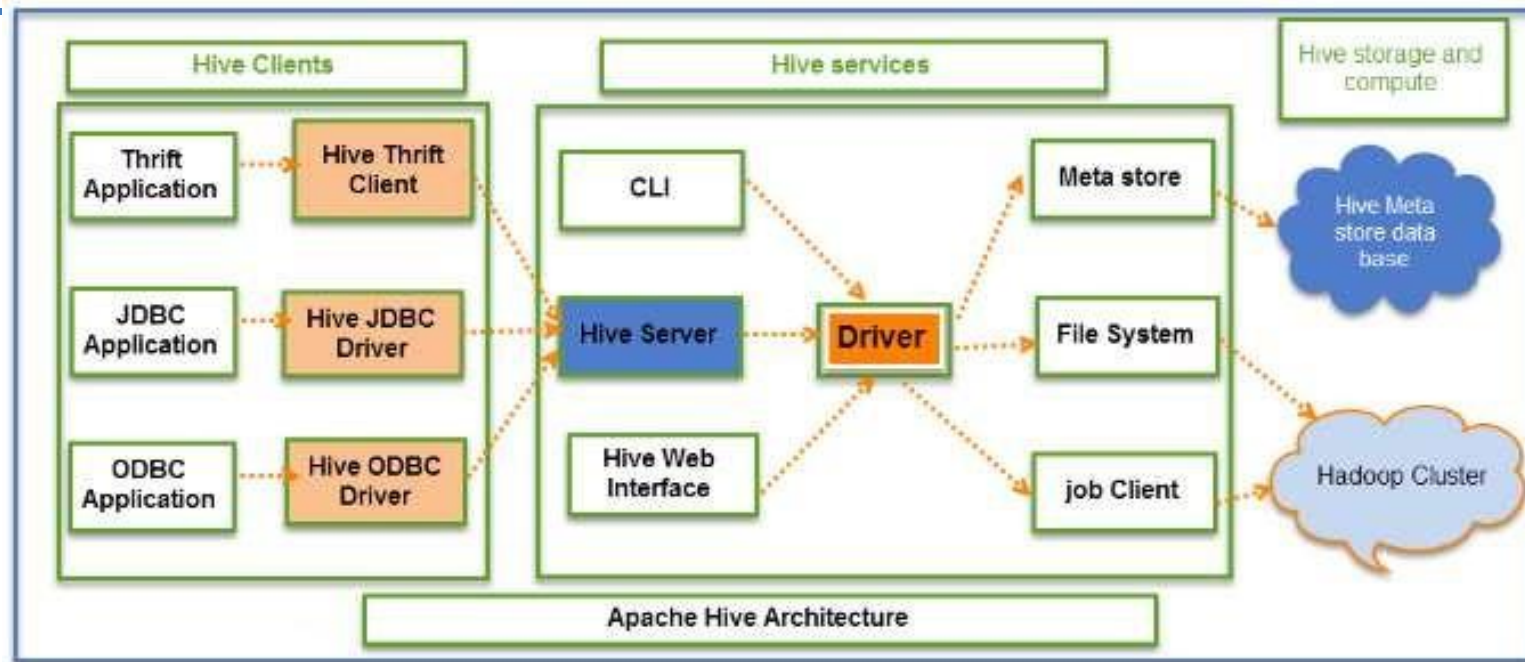
# Hive: Architecture



<https://www.guru99.com/introduction-hive.html>

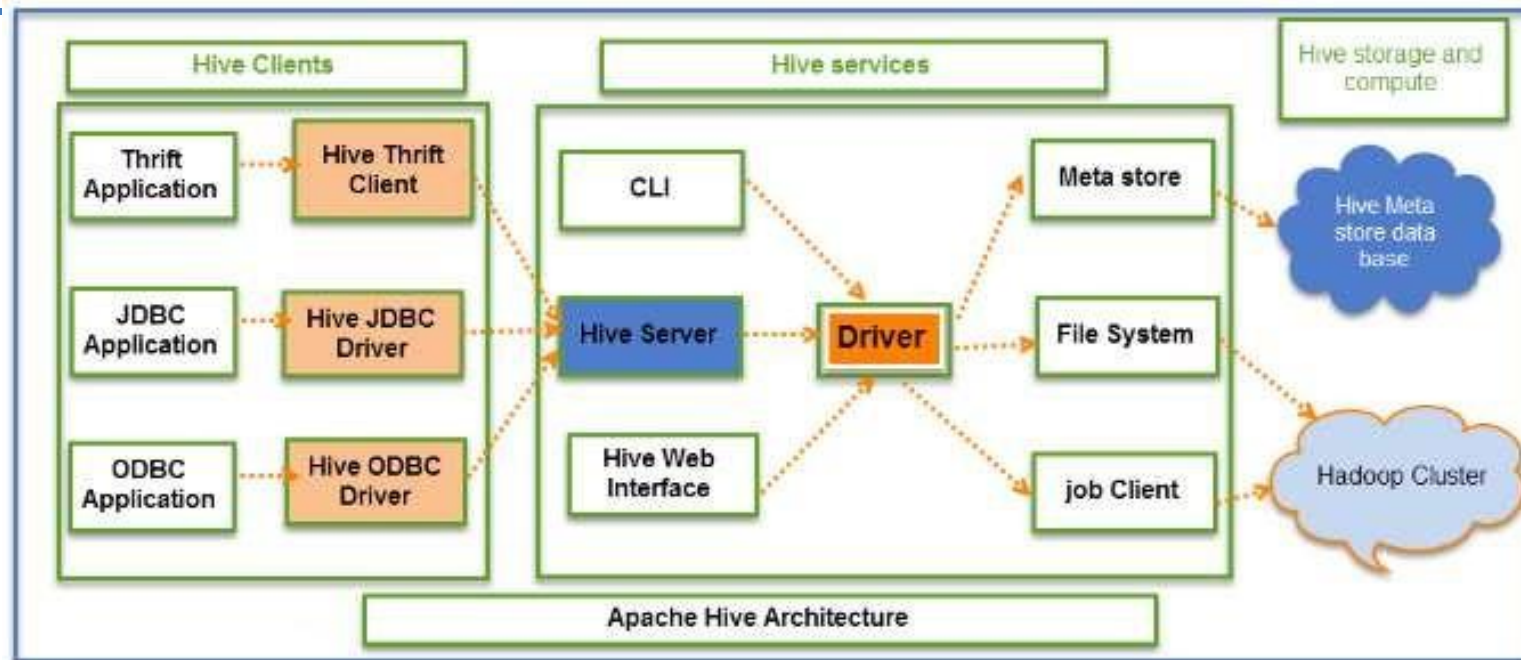
- **Clients Hive:** Hive supporte toute application (C++, Java, Python, ...) utilisant **JDBC**, **Thrift** ou **ODBC** drivers.
- **Services Hive:** Cette couche gère les interactions avec l'utilisateur via :
  - Une interface de commandes en ligne (**CLI**): C'est le shell Hive
  - Une **interface web de Hive**, dans Cloudera Quickstart: **Hue (Hadoop User Experience)**
  - **Hive Server** lorsqu'il s'agit d'applications (JDBC, ODBC ou Thrift).

# Hive: Architecture



- Les requêtes utilisateur sont reçues par le **Driver** qui inclut :
  - Un **compilateur (Compiler)** : vérifie la syntaxe et la sémantique de la requête en utilisant les schémas stockés dans le **Meta Store**.
  - Un **optimiseur (Optimizer)** qui génère le plan d'exécution optimal de la requête sous forme d'un DAG (Directed Acyclic Graph)
  - Un **moteur d'exécution (Execution Engine)** qui exécute le plan d'exécution sous forme d'une série de jobs MapReduce.

# Hive: Architecture



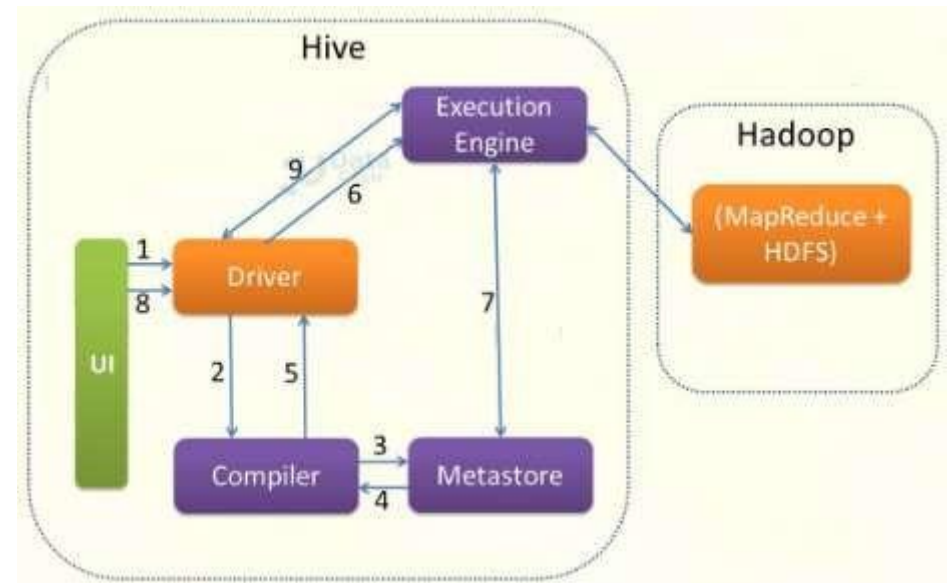
- **Stockage et traitement Hive:**

- Le **MetaStore** est le référentiel central des *métadonnées* Apache Hive. Il stocke les métadonnées des **tables** Hive (*schéma*, *emplacement des fichiers*, ...) et des **partitions** dans une BD relationnelle (*Mysql*, *Derby*, ...). Il consiste en deux unités fondamentales:
  - \* Un **service** qui fournit un accès métastore aux autres services Apache Hive.
  - \* **Stockage des métadonnées** Hive sur disque (*Mysql* ou *Derby*).
- Les données chargées dans les tables Hive et les résultats des requêtes seront stockés dans le cluster Hadoop (**HDFS**)



# Hive: Etapes Traitement des données dans Hive

1. La requête utilisateur est soumise au **Driver**
2. Le **Driver** envoie la requête au compilateur afin d'avoir le plan d'exécution.
- 3.4. Le **compilateur** interagit avec le **MetaStore** pour avoir les métadonnées des tables utilisées dans la requête.



5. Le **compilateur (+optimiseur)** crée le plan d'exécution (sous forme d'un **DAG(Directed Acyclic Graph)** où chaque arc représente un job map/reduce job, une opération sur HDFS ou une opération sur les métadonnées) et l'envoie au **Driver**.
6. Le **Driver** envoie le plan d'exécution au **Moteur d'exécution** qui va l'exécuter sur le cluster en tant qu'une série de jobs **MapReduce**. Les résultats seront stockés dans HDFS.
7. Le **Moteur d'exécution** interagit avec le **MetaStore** lorsqu'il s'agit d'une requête DDL
- 8-9. L'application cliente récupère les résultats depuis le **Driver**. Le **moteur d'exécution** communique avec les **Data Nodes (via le NameNode)** afin d'avoir les résultats qu'il passera ensuite à l'application cliente.

# Hive: Configuration

- Les paramètres de configuration de **Hive** sont dans le fichier **hive-site.xml** (*/etc/hive/conf/hive-site.xml*), par exemple:

```
<property>
  <name>javax.jdo.option.ConnectionURL</name>
  <value>jdbc:mysql://127.0.0.1/metastore?createDatabaseIfNotExist=true</value>
  <description>JDBC connect string for a JDBC metastore</description>
</property>
```

Chaîne de connexion JDBC à la BD du Metastore

```
<property>
  <name>javax.jdo.option.ConnectionDriverName</name>
  <value>com.mysql.jdbc.Driver</value>
  <description>Driver class name for a JDBC metastore</description>
</property>
```

Le classe du driver JDBC

```
<property>
  <name>javax.jdo.option.ConnectionUserName</name>
  <value>hive</value>
</property>
```

Utilisateur et mot de passe.

```
<property>
  <name>javax.jdo.option.ConnectionPassword</name>
  <value>cloudera</value>
</property>
```

```
<property>
  <name>hive.metastore.uris</name>
  <value>thrift://127.0.0.1:9083</value>
  <description>IP address (or fully-qualified domain name) and port of the metastore host</description>
</property>
```

**HiveServer/ Driver** communique avec le **Metastore Service** distant via l'API Thrift



# Hive en ligne de commandes

- La commande **hive**:

```
[cloudera@quickstart ~]$ hive
```

```
Logging initialized using configuration in file:/etc/hive/conf.dist/hive-log4j.properties
```

```
WARNING: Hive CLI is deprecated and migration to Beeline is recommended.
```

```
hive> █
```

- La commande **beeline** :

```
[cloudera@quickstart ~]$ beeline
```

```
Beeline version 1.1.0-cdh5.13.0 by Apache Hive
```

```
beeline> !connect jdbc:hive2://localhost:10000
```

```
scan complete in 3ms
```

```
Connecting to jdbc:hive2://localhost:10000
```

```
Enter username for jdbc:hive2://localhost:10000: hive
```

```
Enter password for jdbc:hive2://localhost:10000: *****
```

```
Connected to: Apache Hive (version 1.1.0-cdh5.13.0)
```

```
Driver: Hive JDBC (version 1.1.0-cdh5.13.0)
```

```
Transaction isolation: TRANSACTION_REPEATABLE_READ
```

```
0: jdbc:hive2://localhost:10000> █
```

- La BD utilisée par défaut est **default**

# Hive en ligne de commandes

- **hive** est un client basé sur **Apache-Thrift** et **Beeline** est un client **JDBC**.
- La commande **hive** se connecte **directement** aux pilotes Hive (**Driver**), ce qui nécessite l'installation de la bibliothèque Hive **sur le poste client**.
- **beeline** se connecte à **hiveserver** via des **connexions JDBC** sans installer les bibliothèques Hive sur le poste client. On peut donc exécuter **beeline** à distance depuis l'extérieur du cluster.

Opération	beeline	hive
Se connecter au hiveserver	<b>!connect</b> <i>jdbc_url</i>	
Lister les tables	<b>!table</b> <b>show tables ;</b>	<b>show tables ;</b>
Lister les colonnes	<b>!column</b> <i>table_name</i> <b>desc table_name ;</b>	<b>desc table_name ;</b>
Exécuter une requête	<code>select * from table_name;</code>	<code>select * from table_name ;</code>
Sauvegarder les commandes et leurs résultats dans un fichier.	<b>!record</b> <i>result_file.txt</i> <b>!record</b>	
Exécuter une commande shell	<b>!sh</b> <code>ls</code>	<b>!ls ;</b>
Exécuter une commande dfs	<b>dfs</b> <code>-ls ;</code>	<b>dfs -ls ;</b>
Exécuter un script hql	<b>!run</b> <i>hql_query_file.hql</i>	<b>source</b> <i>hql_query_file.hql ;</i>
Quitter le shell hive	<b>!quit</b>	<b>quit ;</b>
Choix d'une BD	<b>!connect</b> <i>jdbc_url</i>	<b>use</b> <i>database_name ;</i>

# Hive: Les types de données dans Hive

Primitive type	Description	Example
INT	It has 4 bytes, from -2,147,483,648 to 2,147,483,647.	10
BIGINT	It has 8 bytes, from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807. The postfix is L.	100L
FLOAT	This is a 4 byte single-precision floating-point number, from $1.40129846432481707e^{-45}$ to $3.40282346638528860e^{+38}$ (positive or negative). Scientific notation is not yet supported. It stores very close approximations of numeric values.	1.2345679
DOUBLE	This is an 8 byte double-precision floating-point number, from $4.94065645841246544e^{-324d}$ to $1.79769313486231570e^{+308d}$ (positive or negative). Scientific notation is not yet supported. It stores very close approximations of numeric values.	1.2345678901234567
BINARY	This was introduced in Hive 0.8.0 and only supports CAST to STRING and vice versa.	1011
BOOLEAN	This is a TRUE or FALSE value.	TRUE



# Hive: Les types de données dans Hive

STRING	This includes characters expressed with either single quotes ( ' ) or double quotes ( " ). Hive uses C-style escaping within the strings. The max size is around 2 G.	'Books ' or "Books "
CHAR	This is available starting with Hive 0.13.0. Most UDF will work for this type after Hive 0.14.0. The maximum length is fixed at 255.	'US ' or "US "
VARCHAR	This is available starting with Hive 0.12.0. Most UDF will work for this type after Hive 0.14.0. The maximum length is fixed at 65, 355. If a string value being converted/assigned to a varchar value exceeds the length specified, the string is silently truncated.	'Books ' or "Books "
DATE	This describes a specific year, month, and day in the format of YYYY-MM-DD. It is available starting with Hive 0.12.0. The range of dates is from 0000-01-01 to 9999-12-31.	2013-01-01
TIMESTAMP	This describes a specific year, month, day, hour, minute, second, and millisecond in the format of YYYY-MM-DD HH:MM:SS [ .fff . . . ]. It is available starting with Hive 0.8.0.	2013-01-01 12:00:01.345

# Hive: Les types de données dans Hive

Complex type	Description	Example
ARRAY	This is a list of items of the same type, such as [val1, val2, and so on]. You can access the value using <code>array_name[index]</code> , for example, <code>fruit[0]="apple"</code> . Index starts from 0.	<code>["apple", "orange", "mango"]</code>
MAP	This is a set of key-value pairs, such as {key1, val1, key2, val2, and so on}. You can access the value using <code>map_name[key]</code> for example, <code>fruit[1]="apple"</code> .	<code>{1: "apple", 2: "orange"}</code>
STRUCT	This is a user-defined structure of any type of field, such as {val1, val2, val3, and so on}. By default, STRUCT field names will be col1, col2, and so on. You can access the value using <code>structs_name.column_name</code> , for example, <code>fruit.col1=1</code> .	<code>{1, "apple"}</code>
NAMED STRUCT	This is a user-defined structure of any number of typed fields, such as {name1, val1, name2, val2, and so on}. You can access the value using <code>structs_name.column_name</code> , for example, <code>fruit.apple="gala"</code> .	<code>{"apple": "gala", "weight kg": 1}</code>
UNION	This is a structure that has exactly any one of the specified data types. It is available starting with Hive 0.7.0. It is not commonly used.	<code>{2: ["apple", "orange"]}</code>

# Hive: Créer une BD

- Une BD dans Hive décrit une collection de tables.
- On crée une BD avec: **CREATE DATABASE** [IF NOT EXISTS] *db\_name* ;
- Lorsqu'on crée une nouvelle BD, Hive lui crée un nouveau sous-répertoire dans **/user/hive/warehouse**.

Si la BD existe, on n'aura pas un message d'erreur.

Création de: **/user/hive/warehouse/emi19.db**

- Exemple: hive> **CREATE DATABASE** emi19;
- **DATABASE** a l'alias **SCHEMA** dans HQL:

**CREATE SCHEMA** [IF NOT EXISTS] *db\_name* ;

- Exemple:

hive> **CREATE SCHEMA IF NOT EXISTS** emi19;

- Créer une BD en précisant le chemin du dossier et de la description de la BD:

hive> **CREATE DATABASE IF NOT EXISTS** *db\_name*

hive > **COMMENT** ' *description* '

hive > **LOCATION** ' *hdfs\_directory* ' ;



# Hive: Description et Suppression de BD

- Afficher les BDs existantes: **SHOW DATABASES ;**
- Récupérer les meta-données d'une BD: **DESCRIBE DATABASE** *nom\_bd* ;
- Choisir la BD à utiliser: **USE** *nom\_bd* ;
- Connaitre la BD courante: **SELECT current\_database() ;**
- Supprimer une BD:
  - DROP DATABASE IF EXISTS** *nom\_bd* ; -- échec si la BB n'est pas vide
  - DROP DATABASE IF EXISTS** *nom\_bd* **CASCADE** ; --supprime la BD et les tables
- Afficher la liste des tables de la BD courante: **SHOW TABLES;**
- Afficher l'instruction de création d'une table existante: **SHOW CREATE TABLE** *nom\_table* ;
- Afficher la structure d'une table :
  - DESCRIBE | DESC** [*nom\_bd.*]*nom\_table* ;
  - DESCRIBE | DESC FORMATTED** [*nom\_bd.*]*nom\_table* ; -- pour avoir plus de détails
- Supprimer une table: **DROP TABLE** *nom\_table* ;
- Vider une table: **TRUNCATE TABLE** *nom\_table* ; -- juste pour une table interne.

# HIVE: Les tables

---

- La BD **default** est utilisée par défaut lorsqu'on se connecte à hive.
- Chaque table est **mappée** sur un répertoire qui se trouve par défaut sous **/user/hive/warehouse** dans HDFS. Par exemples:

**/user/hive/warehouse/empl** est créé pour la table **empl** de la BD **default**

**/user/hive/warehouse/emi19.db/empl** est créé pour la table **empl** de la BD **emi19**

- Toutes les données de la table seront stockées dans ce répertoire
- Ce type de table s'appelle une table **interne** ou une table **gérée (Managed Table)**
- Lorsque les données sont **déjà** stockées dans HDFS, une table **externe** peut être créée pour décrire ces données (*pointer sur ces données*).
- Lorsqu'une table **interne** est supprimée, ses **données** et ses **métadonnées** sont supprimées ensemble.
- Lorsque la table **externe** est supprimée, seules ses **métadonnées** sont supprimées. Les **données ne sont pas supprimées**. Ceci est très utile lorsque ces données sont partagées avec d'autres outils (**Pig**, ...).

# Hive: Création de table

- **CREATE** [EXTERNAL] **TABLE** [IF NOT EXISTS] [*db\_name.*] *table\_name*  
( *col\_name* *data\_type* [COMMENT '*col\_comment*' ] , ... )  
[COMMENT '*table\_comment*' ]  
[**ROW FORMAT** *row\_format*]  
[STORED AS *file\_format* ]  
[PARTITIONED BY (*col\_name* *data\_type* )]
- **EXTERNAL**: si on supprime la table, ses données ne seront pas supprimées de HDFS.
- **IF NOT EXISTS**: la table sera créée si elle n'existe pas.
- **COMMENT**: on associe un commentaire à la table ou à une colonne.
- **ROW FORMAT**: on précise le format de chaque ligne Par défaut, les champs sont délimités par (^A) '\001'  
Exemple: **DELIMITED FIELDS TERMINATED BY '|' LINES TERMINATED BY '\n'**
- **STORED AS**: on indique le format du fichier de données relatif à la table.  
Exemple: **TEXTFILE, SEQUENCEFILE, PARQUET, AVRO, RCFILE, ORC**
- **PARTITIONED BY**: on partitionne les fichiers de données de la table selon une ou plusieurs colonnes.

# Hive: Création de table interne

- **Exemple 1:** Soit **temp.csv** un fichier contenant des mesures de températures

Use **emi19**;

**CREATE TABLE IF NOT EXISTS** temperature

( Year int **COMMENT** 'année de mesure',  
T int **COMMENT** 'température enregistrée',  
C int **COMMENT** 'Catégorie'  
)

**COMMENT** 'table des températures enregistrées par année et catégorie'

**ROW FORMAT DELIMITED**

**FIELDS TERMINATED BY '\;**

**STORED AS TEXTFILE;**

```
2011;41;3  
2011;20;3  
2012;31;3  
2012;34;3  
2012;35;3
```

```
hive> describe temperature;  
OK  
year          int      année de mesure  
t             int      température enregistrée  
c             int      Catégorie
```

# Hive: Création de table interne

```
hive> desc formatted temperature;
```

```
OK
```

# col_name	data_type	comment
year	int	année de mesure
t	int	température enregistrée
c	int	Catégorie

```
# Detailed Table Information
```

```
Database:      [ ems19 ]
```

```
Owner:         cloudera
```

```
CreateTime:    Sun Mar 31 04:13:39 PDT 2019
```

```
LastAccessTime: UNKNOWN
```

```
Protect Mode:  None
```

```
Retention:     0
```

```
Location:      hdfs://quickstart.cloudera:8020/user/hive/warehouse/ems19.db/temperature
```

```
Table Type:    [ MANAGED_TABLE ]
```

```
Table Parameters:
```

```
  COLUMN_STATS_ACCURATE  true
```

```
  comment                table des températures enregistrées par année et catégorie
```

# Hive: Création de table interne

**Exemple 2:** Le fichier **empl.txt** contient des informations sur des employés:

Amini Saad|Morocco,Algeria|Male,30|DB:80|Product:Developer Lead  
Salhi Anas|Morocco|Male,35|Perl:85|Product:Lead,Test:Lead  
Faridi Amal|Egypte|Female,27|Python:80|Test:Lead,COE:Architect  
Bellal Hafsa|Tunisia|Female,57|Sales:89,HR:94|Sales:Lead

```
CREATE TABLE IF NOT EXISTS empl
( nom          STRING,
  lieu_t       ARRAY<STRING> COMMENT 'lieu de travail',
  sexe_age     STRUCT<sexe:STRING,age:INT>,
  comp_score   MAP<STRING,INT>  COMMENT 'compétence et score',
  depart_role  MAP<STRING,STRING> COMMENT 'département et rôles'
) ROW FORMAT DELIMITED FIELDS TERMINATED BY '|'
  COLLECTION ITEMS TERMINATED BY ','
  MAP KEYS TERMINATED BY ':'
  STORED AS TEXTFILE;
```



# Hive: Création de table interne

```
hive> desc formatted empl;
```

```
OK
```

# col_name	data_type	comment
nom	string	
lieu_t	array<string>	lieu de travail
sexe_age	struct<sexe:string,age:int>	
comp_score	map<string,int>	compétence et score
depart_role	map<string,string>	département et rôles

```
# Detailed Table Information
```

```
Database:      emsi19
Owner:         cloudera
CreateTime:    Sun Mar 31 04:46:30 PDT 2019
LastAccessTime: UNKNOWN
Protect Mode:  None
```

```
Retention:     0
Location:      hdfs://quickstart.cloudera:8020/user/hive/warehouse/emsi19.db/empl
Table Type:    MANAGED_TABLE
```

```
Table Parameters:
```

```
    transient_lastDdlTime    1554032790
```

```
.....
```

```
.....
```

```
Storage Desc Params:
```

collection.delim	,
field.delim	
mapkey.delim	:
serialization.format	

# Hive: Chargement de données dans une table interne

**LOAD DATA [LOCAL] INPATH 'filepath' [OVERWRITE] INTO TABLE tablename ;**

## Exemple:

**LOAD DATA INPATH '/user/cloudera/hive\_lab/data/empl.txt' INTO TABLE empl;**

→ Le fichier HDFS sera **supprimé** après le chargement (**il est déplacé**)

N.b: On peut copier (**hdfs dfs -cp**) le fichier **empl.txt** directement dans le dossier de la table **/user/warehouse/emsi19/empl**

**LOAD DATA LOCAL INPATH '/home/cloudera/tp/hive/empl.txt' INTO TABLE empl;**

**LOAD DATA LOCAL INPATH '/home/cloudera/tp/hive/empl.txt' OVERWRITE INTO TABLE empl;**

→ La table (**interne ou externe**) sera **vidée** avant le chargement

# Hive: Création de table externe

- **Exemple 1:** Soit **temp.csv** un fichier contenant des mesures de températures

Use **emi19**;

**CREATE EXTERNAL TABLE IF NOT EXISTS** temperature

( Year int **COMMENT** 'année de mesure',  
T int **COMMENT** 'température enregistrée',  
C int **COMMENT** 'Catégorie'  
)

**COMMENT** 'table des températures enregistrées par année et catégorie'

**ROW FORMAT DELIMITED**

**FIELDS TERMINATED BY '\';**

**LOCATION** '/user/cloudera/hive/data/db' ; -- dossier contenant les fichiers de données  
-- s'il n'existe pas, il sera créé.

-- Si la clause Location n'est pas indiquée, les fichiers de données seront stockés dans  
/user/hive/warehouse/**emi19**/temperature

# Hive: chargement de données dans une table externe

---

## Shell Linux

```
hdfs dfs -put /home/cloudera/tp/hive/empl.txt /user/cloudera/hive/data/db
```

## Shell Hive

```
LOAD DATA LOCAL INPATH '/home/cloudera/tp/hive/empl.txt' INTO TABLE empl;
```

# Hive: Atelier 1

1. Dans un terminal, lancer la commande: **hive**
2. Créer la BD *analyse*: **hive> CREATE DATABASE analyse**
3. Lister le contenu du dossier HDFS : */user/hive/warehouse*
4. Utiliser de la BD *analyse* : **hive> Use analyse**
5. Créer la table *vol1* (*year, month, day, fl, dep, arr, distance*)

**hive> CREATE TABLE vol1**

**( year INT, month INT, day INT, fl STRING, dep STRING, arr STRING, distance INT )**  
**ROW FORMAT DELIMITED FIELDS TERMINATED BY '\';**  
**STORED AS TEXTFILE;**

6. Afficher la liste des tables de la BD courante.
7. Consulter le **Metastore** pour avoir le schéma de la table *vol1*: **hive> DESCRIBE vol1 ;**
8. Charger le fichier **local vol.csv** dans la table *vol1* en utilisant la commande **LOAD**
9. Consulter de la table: **hive> SELECT year, dep, COUNT(fl)**  
**FROM vol1**  
**GROUP BY dep, year;**

**N.B: Remarquez les jobs Map-Reduce créés.**

# Hive: Atelier 1

---

10. Créer la table **externe vol2** (*year, month, day, fl, dep, arr, distance*) en indiquant son dossier HDFS de données qu'il faut créer, par exemple: */user/cloudera/hive/data/db*
11. Copier le fichier local **vol.csv** dans le dossier HDFS: */user/cloudera/hive/data/db*
12. Effectuer une requête HQL sur la table **vol2**.
13. Charger le fichier local **vol.csv** dans la table **vol2** avec **LOAD** et sans l'option **overwrite**.
14. Lister le contenu du dossier HDFS: */user/cloudera/hive/data/db*
15. Afficher les métadonnées **détaillées** de la table **vol2**.
16. Exécuter séparément les deux requêtes:

```
SELECT * FROM vol2;
SELECT year, dep, COUNT(fl)
FROM vol1
GROUP BY dep, year;
```

**Quelle est la différence entre les deux lors de leur exécution?**




# Hive: Atelier 1

---

17. Créer la table **interne vol3** (*year, month, day, fl, dep, arr, distance*).
18. Charger le contenu de la table **vol2** dans la table **vol3**.
19. Effectuer une requête HQL (Select ....) sur la table **vol2**.  
Qu'obtient-on? Pourquoi?
20. Lister le contenu du dossier de la table **vol2**  
Comment il est? Pourquoi?


# Hive: Gérer les données avec Apache Hue (Hadoop User Experience)

- Interface web donnant accès aux données stockées dans HDFS.
- Hue est configuré pour répondre sur le port **8888**.
- Hue permet de visualiser l'arborescence HDFS et le **metastore** de Hive.
- L'URL sur la VM Cloudera: <http://10.0.2.15:8888>
- Login/mot de passe = cloudera/cloudera
- Le raccourci  /Files permet de visualiser l'arborescence HDFS

## File Browser

Search for file name		⚙ Actions ▾	✕ Move to trash ▾				⬆ Upload ▾	⬆ New ▾
🏠 Home / user / cloudera								▼ History
<input type="checkbox"/>	Name	Size	User	Group	Permissions	Date		
<input type="checkbox"/>	📁 ↕		hdfs	supergroup	drwxr-xr-x	March 14, 2018 08:18 AM		
<input type="checkbox"/>	📁 .		cloudera	cloudera	drwxr-xr-x	April 14, 2018 12:48 PM		
<input type="checkbox"/>	📄 FL_insurance.csv	3.9 MB	cloudera	cloudera	-rw-r--r--	March 28, 2018 05:18 PM		
<input type="checkbox"/>	📁 tp		cloudera	cloudera	drwxr-xr-x	March 31, 2018 05:34 PM		
<input type="checkbox"/>	📄 vol.csv	2.9 KB	cloudera	cloudera	-rw-r--r--	April 14, 2018 12:48 PM		
<input type="checkbox"/>	📁 wordcount		cloudera	cloudera	drwxr-xr-x	April 08, 2018 10:28 AM		

# Hive: Gérer les données avec Apache Hue (Hadoop User Experience)

- L'éditeur Hive est accessible via l'URL: <http://10.0.2.15:8888/hue>
- Cliquer sur le lien  (à gauche de l'interface Hue) plusieurs fois jusqu'à avoir **Sources**
- Cliquer sur **Hive** pour avoir accès aux BD déjà créées.
- Cliquer sur la BD **analyse** pour avoir ses tables
- Cliquer sur le raccourci **+** pour créer une nouvelle table.
- Remplir les champs comme suit puis cliquer sur **Next**:



## Source

Type

File

Path

/user/cloudera/vol.csv

..



## Format

Field Separator

;

Record Separator

New line

Quote Character

Double Quote

☐ Has Header

## Preview

field_1	field_2	field_3	field_4	field_5	field_6	field_7
2008	1	3	N772SW	IAD	TPA	810
2008	1	3	N428WN	IND	BWI	515

# Hive: Gérer les données avec Apache Hue (Hadoop User Experience)

- Remplir les champs comme suit

**Name:** analyse.vol

**Format:** Text

**Fields:** year:int , month:int, day:int, flight:string, depart:string,  
destination:string, distance:int

- Cliquer sur **Submit**
- Les détails de la table créée seront affichés: [Databases](#) > [default](#) > [vol](#)
- Choisir: **Query** / **Editor** / **Hive**
- Saisir la requête suivante puis cliquer sur le petit triangle bleu pour l'exécuter:

```
1 SELECT flight, depart, destination, distance
2 FROM vol
3 WHERE depart like 'I%'
4 ORDER BY depart
```



- Refaire la même chose pour:

```
1 SELECT depart, count(depart) as Nb_Vol
2 FROM vol
3 WHERE depart like 'I%'
4 GROUP BY depart
5 ORDER BY depart
```

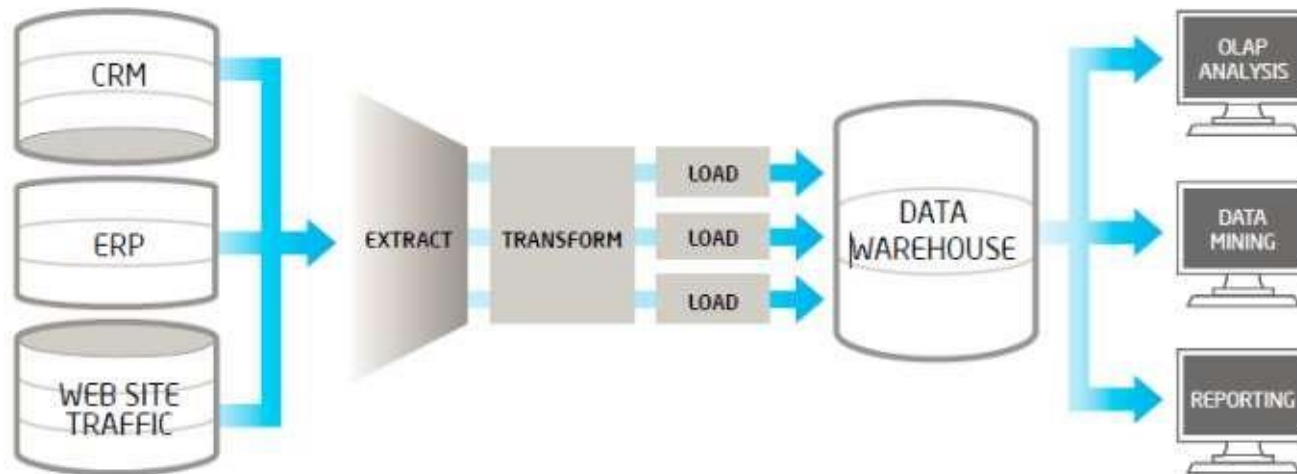
# Hive: Format de fichiers

---

- Les formats de représentation de données utilisés en Big Data sont très variés:
  - Formats de stockage **en lignes** ou enregistrements:
    - Fichiers plats / fichiers texte
    - CSV et fichiers délimités
    - JSON
    - **SequenceFile**
    - **Avro**
    - Autres formats: XML, YAML
  - Formats de stockage **orientés colonnes**:
    - **RC** (*Record Columnar File*) / **ORC** (*Optimized Row Columnar File*)
    - **Parquet**
- Bibliothèques de compression des données

# Fichier Plat/Fichier texte

- Le processus **ETL** (Extract-Transform-Load) traditionnel extrait les données de plusieurs sources, les nettoie, les formate et les charge dans un entrepôt de données pour des besoins d'analyse.



- Les fichiers plats ou texte doivent être structurés et traités sous forme de champs (attributs):
  - ☐ Les champs peuvent être à des positions fixées dans des enregistrements
  - ☐ Ou, l'analyse textuelle peut être nécessaire pour extraire le sens.
- Généralement en Big Data, on charge les données dans leur état brut.



# CSV et fichiers délimités

---

- Format utilisé dans les tableurs (Excel, ...):
  - ☐ Chaque ligne correspond à un enregistrement.
  - ☐ Les colonnes d'une ligne sont séparées par une virgule (ou autre délimiteur: tabulation, |, ; , ...)
  - ☐ On peut avoir une ligne d'en-tête avec des noms de colonnes
- Problèmes:
  - ☐ Des caractères d'échappement peuvent être présents (backslash = \)
  - ☐ Windows et Linux utilisent différents caractères de fin de ligne
- Python a une bibliothèque standard qui inclut un package CSV
- Les capacités des formats CSV sont limitées:
  - ☐ **Suppose que chaque enregistrement a un nombre fixe d'attributs**
  - ☐ **Pas facile de représenter des ensembles, des listes, des maps ou des structures de données plus complexes**

# JSON: JavaScript Object Notation

- JSON est un format de sérialisation d'objets en texte brut qui peut représenter des données assez complexes pouvant être transférées entre utilisateur/programme ou programme/programme
- Souvent appelé le langage du Web 2.0
- Deux structures de base:
  - *Enregistrements* constitués de map (paires *clé/valeur*), entre accolades:  
`{nom: "John", âge: 25 ans}`
  - Les *listes (tableaux)* sont entre crochets: `[. . . ]`
- Les enregistrements et les tableaux peuvent être imbriqués  
`{ "firstName": "John", "lastName": "Smith", "phoneNumbers": [ {"type": "home", "number": "212 ....4"}, {"type": "office", "number": "646 555-4567" } ] ....}`
- Les bibliothèques supportant JSON sont disponibles dans R, Python, ...etc
- Il y a beaucoup d'API qui renvoient des données JSON: Google Geocoder, Twitter, Yahoo Answers, ... etc.

# SequenceFile

---

- Représentation **binaire** de pairs *clé/valeur*
- **Orienté ligne** (enregistrements)
- Utilisé pour le transfert de données entre les jobs Map et Reduce.
- Permet de spliter les données, pour les job Map, même compressées.
- La lecture de données dans le format **SequenceFile** est plus performante que celle des fichiers textes plats (*pas besoin de structuration d'enregistrements*)
- **Ce format a une implémentation uniquement en Java où plusieurs classes permettent de lire, écrire ou trier.**
- Il y a trois choix de compression de fichier **SequenceFile**:
  - 1.NONE**: Enregistrements *clé/valeur* non compressés.
  - 2.RECORD**: Enregistrements *clé/valeur* compressés: seules les valeurs sont compressées
  - 3.BLOCK**: Bloc d'enregistrements *clé/valeur* compressés: les clés et les valeurs sont collectées dans des «blocs» séparément et compressées. La taille du 'bloc' est configurable par **io.seqfile.compress.blocksize** dans **core-site.xml** (valeur par défaut: 1000000 Octets)



- Le format **Avro** permet de stocker les fichiers sous format **binaire** permettant l'interopérabilité d'applications écrites en différents langages de programmation · **Avro** est **indépendant** des langages de programmations.
- **Avro** est **orienté ligne** (*enregistrement*) dont le schéma est codé en **JSON** dans le fichier lui-même.
- Les blocs de fichiers Avro peuvent être compressés et splitables.
- **Avro** **supporte l'évolution de schéma (versioning)** : par exemple, si des colonnes sont ajoutées ou supprimées d'une table, les fichiers de données précédemment importés peuvent être traités avec les nouveaux.

# Record Columnar File (RC)

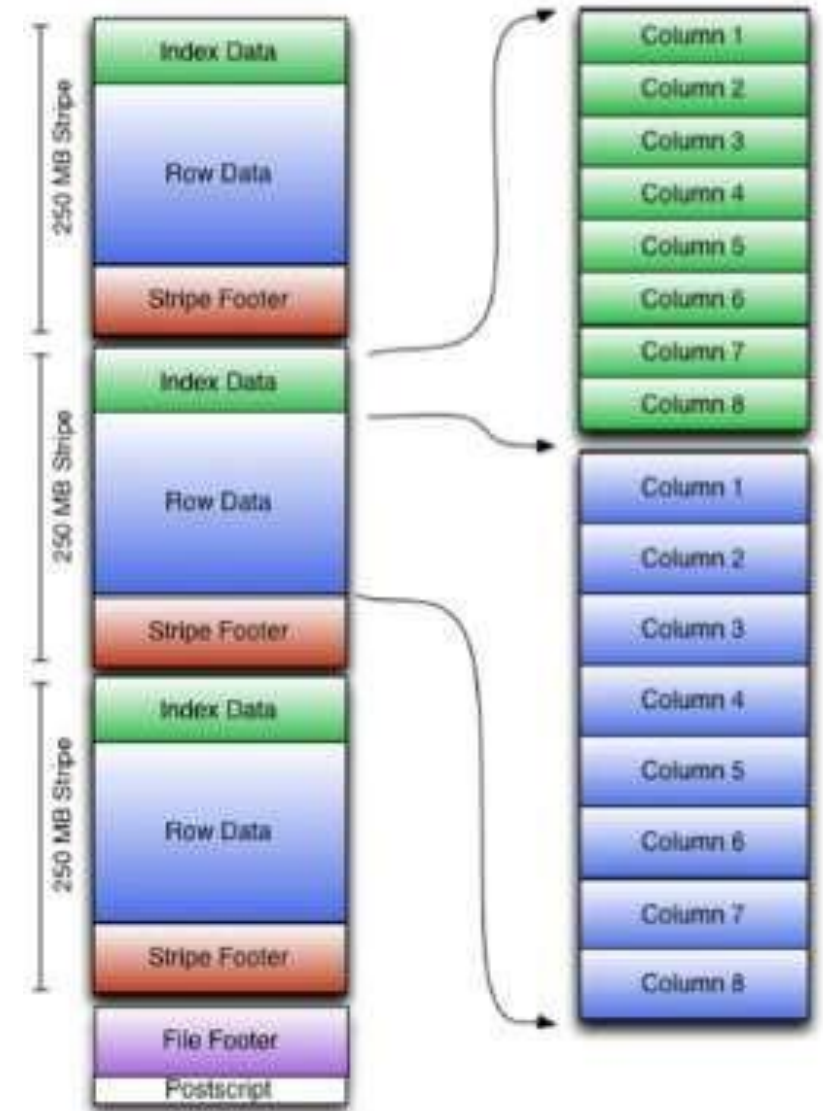
---

- **RCFile** (**R**ecord **C**olumnar **F**ile) est le premier format "**colonne**" adopté par Hadoop et utilisé initialement dans **Hive**.
- **Objectif**: chargement rapide des données, exécution rapide des requêtes et une utilisation très efficace de l'espace de stockage.
- Colonnes stockées séparément en format binaire.
- Lecture et décompression des colonnes désirées.
- RC applique une compression par colonne sur des groupes de lignes.
- Le format RC est un format tabulaire constitué de lignes et de colonnes.
- Les tables sont stockées dans HDFS: des groupes de lignes sont stockés dans des blocs HDFS.

# Optimized Row Columnar (ORC)

Le format ORC (**O**ptimized **R**ow **C**olumnar) désigné comme successeur du format RC en proposant une approche optimisée de compression plus efficace:

- ❑ Un fichier ORC contient des **bandes (stripes)** de données de ligne
- ❑ Taille de **bandes** par défaut 256 Mo et peut être plus grande.
- ❑ Les données d'**index** se composent de valeurs minimales et maximales pour chaque colonne, ainsi que les positions de ligne dans chaque colonne.
- ❑ L'indexation permet de sauter des blocs entiers de rangées ne répondant pas à une requête.
- ❑ Le **pied** de bande contient l'encodage de chaque colonne.
- ❑ À la **fin** du fichier, un **postscript** contient les paramètres de compression.
- ❑ Algorithmes (codecs: Snappy, GZIP, BZIP2, ...) de compression **spécifiques pour différents types de colonnes**.





# Parquet

---

- Apache **Parquet** est un format de stockage en **colonnes** compressées (développé par **Cloudera** et **Twitter**)
- Il a les mêmes caractéristiques que ORC:
  - ❑ Groupement de lignes avec un stockage et compression par colonne
  - ❑ Supporte les évolutions de schéma.
- Il est compatible avec toutes les interfaces **MapReduce** comme **Java**, **Hive**, **Pig** et d'autres moteurs d'exécution comme **Impala** ou **Spark**.
- **Parquet** supporte des structures de données complexes ainsi que la lecture et l'écriture par les **APIs d'Avro**.
- Fournit l'un des meilleurs résultats dans divers tests de **performance** de référence
- Le stockage en colonnes offre les avantages suivants:
  - la compression est plus efficace car les données de colonne sont du même type.
  - Le traitement des requêtes est plus efficace car les colonnes sont stockées ensemble.

# HIVE: Table au format Avro *(avant la version 1.0)*

- Création d'un fichier JSON sur HDFS contenant le schéma de la table:

```
{ "type" : "record",  
  "name" : "nom_schema",  
  "namespace" : "namespace_schema",  
  "fields" : [  
    { "name" : "nom_champ",  
      "type" : "type_champ",  
      "default" : "val_défaut|NONE",  
      "doc" : "comment" },  
    ...  
  ]  
}
```

**Exemple:** Le fichier HDFS ' /user/cloudera/hive\_lab/schema\_avro/temperature.avsc '

```
{ "type" : "record",  
  "name" : "temperature",  
  "namespace" : "avro.google.com",  
  "fields" : [{ "name" : "year", "type" : "int", "default" : "1800", "doc" : "année d'enregistrement"},  
    { "name" : "T", "type" : "int", "default" : "999", "doc" : "température enregistrée"},  
    { "name" : "C", "type" : "int", "default" : "-1", "doc" : "catégorie"}  
  ]  
}
```

## HIVE: Table au format Avro (*avant la version 1.0*)

Classe	Description	Utilisée dans la clause
org.apache.hadoop.hive.serde2.avro. <b>AvroSerDe</b>	La classe Hive SerDe pour le stockage de données au format Avro	ROW FORMAT SERDE
org.apache.hadoop.hive ql.io.avro. <b>AvroContainerInputFormat</b>	Format de données en entrée d'un conteneur Avro.	STORED AS
org.apache.hadoop.hive ql.io.avro. <b>AvroContainerOutputFormat</b>	Format de données de sortie d'un conteneur Avro.	STORED AS

- La clause **TBLPROPERTIES** de la commande **CREATE EXTERNAL TABLE** permet de spécifier le **schéma** de la table si les données de la table sont stockées au format Avro (ou autre: ORC, ....).

**CREATE EXTERNAL TABLE IF NOT EXISTS** *nom\_table*

**ROW FORMAT SERDE** 'org.apache.hadoop.hive.serde2.avro.AvroSerDe'

**STORED AS INPUTFORMAT** 'org.apache.hadoop.hive ql.io.avro.AvroContainerInputFormat'

**OUTPUTFORMAT** 'org.apache.hadoop.hive ql.io.avro.AvroContainerOutputFormat'

**TBLPROPERTIES** ('**avro.schema.url**'='hdfs\_directory/fichier\_schéma.avsc');

# HIVE: Table au format Avro (*avant la version 1.0*)

---

- Exemple:

**CREATE EXTERNAL TABLE IF NOT EXISTS** temperature\_avro

**ROW FORMAT SERDE** 'org.apache.hadoop.hive.serde2.avro.AvroSerDe'

**STORED AS INPUTFORMAT** 'org.apache.hadoop.hive ql.io.avro.AvroContainerInputFormat'

**OUTPUTFORMAT** 'org.apache.hadoop.hive ql.io.avro.AvroContainerOutputFormat'

**LOCATION** '/user/cloudera/hive\_lab/data/db\_avro'

**TBLPROPERTIES** ('avro.schema.url'='/user/cloudera/hive\_lab/schema\_avro/temperature.avsc') ;

# HIVE: Table au format Avro

---

- A partir de la version Hive 1.0:

**CREATE EXTERNAL TABLE** *nom\_table* (.....)

**STORED AS AVRO**

**LOCATION** '*hdfs\_directory*' ;

- Chargement de données (fichiers) au format Avro avec l'instruction **LOAD DATA**.

**N.B:** Les fichiers Avro chargés doivent avoir le schéma de la table.

- Importation de données dans une table au format **Avro** à partir d'une autre table:

**INSERT OVERWRITE TABLE** *table\_destination* **SELECT \* FROM** [*db\_name.*]*table\_source* ;

**Exemple:** **INSERT OVERWRITE TABLE** *temperature\_avro* **SELECT \* FROM** *temperature* ;

→ *Un fichier binaire sera créé dans le dossier de données de la table dont le schéma sera stockée en format JSON dans l'entête du fichier.*

# HIVE: Table au format Parquet

---

- Création d'une table externe au format Parquet

```
CREATE EXTERNAL TABLE [IF NOT EXISTS] nom_table (.....)  
ROW FORMAT SERDE 'parquet.hive.serde.ParquetHiveSerDe'  
STORED AS PARQUET  
LOCATION 'hdfs_directory' ;
```

## Exemple:

```
CREATE EXTERNAL TABLE IF NOT EXISTS temperature_parquet  
    (  
        Year int,  
        T int,  
        C int  
    )  
ROW FORMAT SERDE 'parquet.hive.serde.ParquetHiveSerDe'  
STORED AS PARQUET  
LOCATION '/user/cloudera/hive_lab/data/db_parquet'
```

- On importe des données dans la table avec **INSERT OVERWRITE TABLE**



# Hive: Atelier 2

---

1. Créer les dossiers HDFS suivants:

`/user/cloudera/hive_lab/data/bank_txt`

`/user/cloudera/hive_lab/data/bank_avro`

`/user/cloudera/hive_lab/data/bank_parquet`

**N.B:** Utiliser **mkdir** avec l'option **p** pour créer tous les dossiers avec une seule commande

**hdfs dfs -mkdir -p /user/cloudera/hive\_lab/data/{bank\_txt,bank\_avro,bank\_parquet}**

2. Copier le fichier local **bank-data.csv** dans le dossier HDFS `/user/cloudera/hive_lab/data/bank_txt`

3. Créer la table externe **bank\_txt** stockée au format texte et dont:

- les données sont dans le dossier HDFS `/user/cloudera/hive_lab/data/bank_txt`
- les colonnes contiennent: *id client (entier), âge (entier), sexe (texte), région (texte), revenu (, marié, enfants, voiture, hypothèque.*

**N.B:** Visualiser le contenu du fichier CSV pour avoir le type de chaque colonne.

4. Consulter la table créée par une requête HQL.

## Hive: Atelier 2

---

5. Créer la table externe **bank\_avro** stockée au format **Avro** et dont le dossier de données est **/user/cloudera/hive\_lab/data/bank\_avro**
6. Importer dans la table **bank\_avro** les données de la table **bank\_txt**.
7. Lister ( avec **ls**) les fichiers du dossier HDFS **/user/cloudera/hive\_lab/data/bank\_avro**
8. Afficher ( avec **cat** ) le contenu des fichiers de **/user/cloudera/hive\_lab/data/bank\_avro**
9. Créer la table externe **bank\_parquet** stockée au format **Parquet** et dont le dossier de données est **/user/cloudera/hive\_lab/data/bank\_parquet**
10. Importer dans la table **bank\_parquet** les données de la table **bank\_txt**.
11. Lister ( avec **ls**) les fichiers du dossier HDFS **/user/cloudera/hive\_lab/data/bank\_parquet**
12. Afficher ( avec **cat** ) le contenu des fichiers de **/user/cloudera/hive\_lab/data/bank\_parquet**
13. Ecrire des requêtes afin d'avoir à partir des **trois tables** :
  - a) La moyenne de revenu par région.
  - b) Le pourcentage des clients mariés et ayant une hypothèque.
  - c) Le nombre de clients mariés par région et ayant des enfants et une hypothèque
  - d) Les identifiants des clients ayant le revenu maximal.

# Hive: Table partitionnée

- Par défaut, une requête HQL analyse **toute la table**. Cela ralentit les performances lors de l'interrogation d'une grande table.
- Ce problème pourrait être résolu en créant des **partitions** similaires à celles dans les SGBDR.
- Dans Hive, le **partitionnement** est effectué par une ou plusieurs colonnes.
- Chaque **partition** est mappée vers un **sous-répertoire** du répertoire de la table dans HDFS.
- Lorsque la table est interrogée, seules les partitions requises (*répertoires*) des données de la table sont lues afin de réduire le temps d'exécution de la requête et améliorer les performances.
- Chaque partition a son propre stockage (*répertoire*) et sa propre sérialisation (*format*)
- **Exemple:**

```
CREATE EXTERNAL TABLE temperature_partition
(
    -- colonnes régulières
    T int, C int
)
    -- colonnes de partitionnement
PARTITIONED BY (Year int)
ROW FORMAT DELIMITED FIELDS TERMINATED BY '\;'
STORED AS TEXTFILE
LOCATION '/user/cloudera/hive_lab/data/temperature_partition' ;
```

# Hive: Table partitionnée

- La description de la table partitionnée fournit des informations sur le partitionnement effectuée

Exemple:

```
hive> desc temperature_partition;
OK
t                int
c                int
year             int
# Partition Information
# col_name      data_type      comment
year            int
```

- Par défaut, le partitionnement dynamique n'est pas activé.
- On utilise l'instruction **ALTER TABLE ... ADD PARTITION** pour ajouter des partitions **statiques** (*manuellement*) à une table.

Exemple:

**ALTER TABLE** temperature\_partition

**ADD PARTITION** (year=2011) **PARTITION** (year=2013);

```
hive> show partitions temperature_partition;
OK
year=2011
year=2013
```

```
[cloudera@quickstart ~]$ hdfs dfs -ls /user/cloudera/hive_lab/data/temperature_partition
Found 2 items
drwxr-xr-x - cloudera cloudera      0 2019-04-16 08:10 /user/cloudera/hive_lab/data/temperature_partition/year=2011
drwxr-xr-x - cloudera cloudera      0 2019-04-16 08:10 /user/cloudera/hive_lab/data/temperature_partition/year=2013
```

- La commande **SHOW PARTITIONS** permet d'avoir la liste des partitions actuelles dans une table partitionnée. Exemple: **SHOW PARTITIONS** temperature\_partition;

# Hive: Table partitionnée

- On utilise l'instruction **ALTER TABLE ... DROP PARTITION** pour supprimer des partitions.

Exemple: **ALTER TABLE** temperature\_partition

**DROP IF EXISTS PARTITION (year=2011);**

- On utilise l'instruction **TRUNCATE TABLE ... PARTITION** pour vider une partition d'une **table interne**.

Exemple: **TRUNCATE TABLE** temperature\_partition **PARTITION (year=2011);**

- On utilise l'instruction **hdfs dfs -rm** pour vider une partition d'une **table externe**.

Exemple:

**hdfs dfs -rm -f /user/cloudera/hive\_lab/data/temperature\_partition/year=2011/\***

- On peut activer le partitionnement **dynamique** afin que les partitions soient créées de manière dynamique à partir de valeurs de données insérées. **[utiliser Hive Shell ou hive-site.xml ]**

**hive> SET hive.exec.dynamic.partition=true;**

**hive> SET hive.exec.dynamic.partition.mode=nonstrict;**

# Hive: Table partitionnée

- On importe les données dans la table partitionnée dont les partitions seront créées dynamiquement.

Exemple: **INSERT OVERWRITE TABLE** temperature\_partition

**PARTITION** (year)

**SELECT T, C, Year FROM temperature;** *--Attention à l'ordre des colonnes*

*N.B: la partition year=\_\_HIVE\_DEFAULT\_PARTITION\_\_ sera créée pour les enregistrement ayant year=NULL*

*Création dynamiques des partitions selon les données importées*

```
hive> show partitions temperature_partition;  
OK  
year=2011  
year=2012  
year=2013  
year=2014  
year=2015
```

```
[cloudera@quickstart hive]$ hdfs dfs -ls /user/cloudera/hive_lab/data/temperature_partition
```

Found 5 items

```
drwxr-xr-x - cloudera cloudera 0 2019-04-14 10:21 /user/cloudera/hive_lab/data/temperature_partition/year=2011  
drwxr-xr-x - cloudera cloudera 0 2019-04-14 10:21 /user/cloudera/hive_lab/data/temperature_partition/year=2012  
drwxr-xr-x - cloudera cloudera 0 2019-04-14 10:21 /user/cloudera/hive_lab/data/temperature_partition/year=2013  
drwxr-xr-x - cloudera cloudera 0 2019-04-14 10:21 /user/cloudera/hive_lab/data/temperature_partition/year=2014  
drwxr-xr-x - cloudera cloudera 0 2019-04-14 10:21 /user/cloudera/hive_lab/data/temperature_partition/year=2015
```

- On peut charger des fichiers dans une partition spécifique d'une table partitionnée.

**LOAD DATA INPATH** '/user/cloudera/data/2012.txt' **INTO TABLE** temperature\_partition  
**PARTITION**(year=2012);

*Attention à l'ordre des colonnes dans 2012.txt*



# Hive: Table partitionnée

Modifier les propriétés d'une partition:

**ALTER TABLE** temperature\_partition **PARTITION** (year=2011) **SET FILEFORMAT** ORC;

Les anciennes données avec l'ancien format ne sont plus accessible avec Select.

**Exemple:** Select \* from temperature\_partition ; → *exception : Malformed ORC file..... Invalid postscript.)*

**ALTER TABLE** temperature\_partition **PARTITION** (year=2011) **SET LOCATION**

'/user/hive\_lab/data';

**ALTER TABLE** temperature\_partition **PARTITION** (year=2011) **ENABLE NO\_DROP**;

Empêcher la suppression d'une partition

**ALTER TABLE** temperature\_partition **PARTITION** (year=2011) **DISABLE NO\_DROP**;

**ALTER TABLE** temperature\_partition **PARTITION** (year=2011) **ENABLE OFFLINE**;

L'accès aux données d'une partition **offline** générera une exception.

**Exemple:** Select \* From temperature\_partition ; → *exception : Query against an offline table or partition  
Table temperature\_partition Partition year=2011*

**ALTER TABLE** temperature\_partition **PARTITION** (year=2011) **DISABLE OFFLINE**;

**ALTER TABLE** temperature\_partition **PARTITION** (year=2011) **CONCATENATE**;

La fusion est possible pour les partitions ayant le format **RCFILE** ou **ORCFIL**.

# Hive: Table partitionnée

- On peut partitionner une table suivant plusieurs colonnes ce qui engendrera la création de plusieurs **dossiers** et **sous-dossiers**.

## Exemple:

```
CREATE EXTERNAL TABLE temperature_partition2 ( T int )
```

```
PARTITIONED BY (Year int, C int)
```

```
ROW FORMAT DELIMITED FIELDS TERMINATED BY '\;' STORED AS TEXTFILE
```

```
LOCATION '/user/cloudera/hive_lab/data/temperature_partition2' ;
```

```
INSERT OVERWRITE TABLE temperature_partition2 PARTITION (Year, C)
```

```
SELECT T, Year, C FROM temperature; --Attention à l'ordre des colonnes
```

```
hive> show partitions temperature_partition2;  
OK  
year=2011/c=1  
year=2011/c=3  
year=2011/c=4  
year=2011/c=9  
year=2011/c=99  
year=2012/c=1  
year=2012/c=3  
year=2012/c=4  
year=2012/c=9  
.....
```

# Hive: Table partitionnée

- On aura un partitionnement de premier niveau effectué par **Year** et un deuxième effectué par **C**.

**hdfs dfs -ls -R /user/cloudera/hive\_lab/data/temperature\_partition2**

```
/user/cloudera/hive_lab/data/temperature_partition2/year=2011
/user/cloudera/hive_lab/data/temperature_partition2/year=2011/c=1
/user/cloudera/hive_lab/data/temperature_partition2/year=2011/c=1/000000_0
/user/cloudera/hive_lab/data/temperature_partition2/year=2011/c=3
/user/cloudera/hive_lab/data/temperature_partition2/year=2011/c=3/000000_0
/user/cloudera/hive_lab/data/temperature_partition2/year=2011/c=4
/user/cloudera/hive_lab/data/temperature_partition2/year=2011/c=4/000000_0
/user/cloudera/hive_lab/data/temperature_partition2/year=2011/c=9
/user/cloudera/hive_lab/data/temperature_partition2/year=2011/c=9/000000_0
/user/cloudera/hive_lab/data/temperature_partition2/year=2011/c=99
/user/cloudera/hive_lab/data/temperature_partition2/year=2011/c=99/000000_0
/user/cloudera/hive_lab/data/temperature_partition2/year=2012
/user/cloudera/hive_lab/data/temperature_partition2/year=2012/c=1
/user/cloudera/hive_lab/data/temperature_partition2/year=2012/c=1/000000_0
/user/cloudera/hive_lab/data/temperature_partition2/year=2012/c=3
/user/cloudera/hive_lab/data/temperature_partition2/year=2012/c=3/000000_0
/user/cloudera/hive_lab/data/temperature_partition2/year=2012/c=4
/user/cloudera/hive_lab/data/temperature_partition2/year=2012/c=4/000000_0
/user/cloudera/hive_lab/data/temperature_partition2/year=2012/c=9
/user/cloudera/hive_lab/data/temperature_partition2/year=2012/c=9/000000_0
```

# Hive: Table partitionnée

---

Modifier les propriétés d'une partition multi-colonnes:

**ALTER TABLE** temperature\_partition2 **PARTITION** (year=2011, c=3) **SET FILEFORMAT** AVRO;

La sous-partition c=3 de la partition year=2011 aura le format AVRO

**ALTER TABLE** temperature\_partition2 **PARTITION** (year=2011,c=3) **ENABLE OFFLINE**;

La sous-partition c=3 de la partition year=2011 sera offline.

**ALTER TABLE** temperature\_partition2 **PARTITION** (c=3) **ENABLE OFFLINE**;

La sous-partition c=3 de **toutes** les partitions year sera offline.

## Hive: Bucketed Table

---

- Le partitionnement fournit un moyen pour séparer les données d'une table Hive en plusieurs fichiers/répertoires. Cependant, cela ne donne de bonnes performances que dans peu de scénarios, comme:
  - Quand le nombre de partitions est limité.
  - Ou lorsque les partitions sont de taille relativement égale.
- Un grand nombre de partitions engendre un grand nombre de fichiers HDFS ce qui affecte considérablement les performances du **Namenode**.
- Hive propose le concept **Bucketing** qui est une autre technique pour décomposer les données de tables en parties plus faciles à gérer.
- Contrairement aux partitions, un **Bucket** (*compartiment*) correspond à un **fichier** HDFS.
- Le nombre de **Buckets** est défini par l'utilisateur lors de la création de la table
- Le **Bucketing** est effectué selon une **colonne** (ou plusieurs) dont la valeur sera **hachée** afin de déterminer le Bucket dans lequel chaque enregistrement doit être stocké.
- La meilleur façon de choisir les colonnes de Bucketing est d'identifier les colonnes les plus utilisés pour filtrer les enregistrement ou dans les jointures selon la logique métier.

## Hive: Bucketed Table

---

- Exemple: Si on a un **Bucketing** par *idclient*, les enregistrements ayant le même *idclient* seront toujours stockés dans le même Bucket.
- La clause **CLUSTERED BY** permet de préciser les colonnes de Bucketing et le nombre de Buckets: **CLUSTERED BY** (*nom\_colonne*, ...) [**SORTED BY** (*coll* [ASC|DESC], ...)]  
**INTO** *nb\_buckets* **BUCKETS**
- Exemple:

```
CREATE EXTERNAL TABLE IF NOT EXISTS temperature_buckets
(
    Year int,
    T int,
    C int
)
CLUSTERED BY (year) INTO 2 BUCKETS
ROW FORMAT DELIMITED FIELDS TERMINATED BY '\;'
STORED AS TEXTFILE
LOCATION '/user/cloudera/hive_lab/data/temperature_buckets' ;
```



# Hive: Bucketed Table

- Activer le Bucketing avant l'importation des données : **set hive.enforce.bucketing = true;**
- Importation de données:

**INSERT OVERWRITE TABLE** temperature\_buckets **SELECT \* FROM** temperature;

```
[cloudera@quickstart ~]$ hdfs dfs -ls /user/cloudera/hive_lab/data/temperature_buckets
Found 2 items
-rwxr-xr-x 1 cloudera cloudera 14948 2019-04-21 11:16 /user/cloudera/hive_lab/data/temperature_buckets/000000_0
-rwxr-xr-x 1 cloudera cloudera 45865 2019-04-21 11:16 /user/cloudera/hive_lab/data/temperature_buckets/000001_0
```

- Le compartiment (Bucket) de chaque enregistrement est déterminé par :

**Hash**(*val\_colonne(s)*) MOD *Nbre\_Buckets*

**Exemple:** Si on a Nbre\_Buckets =4, on aura quatre fichiers créés **000000\_0** , **000001\_0**, **000002\_0** et **000003\_0**

Hash(colonne(s)) MOD 4 =

0	□	stockage dans le fichier	000000_0
1	□	stockage dans le fichier	000001_0
2	□	stockage dans le fichier	000002_0
3	□	stockage dans le fichier	000003_0

## Hive: Bucketed Table

---

- Pour pouvoir réaliser les opérations **Update** et **Delete** sur une table, elle doit être une **Bucketed** table au format **ORCFIELD**.
- Les paramètres suivants doivent être positionnés dans **hive-site.xml** ou via le shell Hive:

`SET hive.support.concurrency = true;`

`SET hive.enforce.bucketing = true;`

`SET hive.exec.dynamic.partition.mode = nonstrict;`

`SET hive.txn.manager = org.apache.hadoop.hive.ql.lockmgr.DbTxnManager;`

`SET hive.compactor.initiator.on = true;`

`SET hive.compactor.worker.threads = 1;`

# Hive: Atelier 3

---

1. Créer les dossiers HDFS suivants:

`/user/cloudera/hive_lab/data/bank_partition`

`/user/cloudera/hive_lab/data/bank_bucket`

`/user/cloudera/hive_lab/data/bank_part_bucket`

**N.B:** Utiliser **mkdir** avec l'option **p** pour créer tous les dossiers et sous-dossiers avec une seule commande

```
hdfs dfs -mkdir -p /user/cloudera/hive_lab/data/{bank_partition,bank_bucked,bank_part_bucket}
```

2. Créer la table interne temporaire **bank\_temp** stockée au format texte.

```
CREATE TEMPORARY TABLE bank_temp (id string, age int, sexe string, region  
string, revenu double, marie string, enfant int, voiture string, hypothec string)
```

```
ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t'
```

```
STORED AS TEXTFILE;
```

3. Charger le fichier local **bank-data.csv** dans la table **bank\_temp**.

Interroger la table **bank\_temp** par une requête HQL

## Hive: Atelier 3

4. Créer la table externe **bank\_partition** partitionnée par **région** et dont le dossier de données est **/user/cloudera/hive\_lab/data/bank\_partition**
5. Créer la table externe **bank\_bucket** ayant deux **Buckets** par la colonne **âge** triée dans l'ordre croissant. Le dossier de données est **/user/cloudera/hive\_lab/data/bank\_bucket**
6. Activer le partitionnement et le Bucketing automatique
7. Importer dans les table **bank\_partition** et **bank\_bucket** les données de la table **bank\_temp**.

```
FROM bank_temp
```

```
INSERT OVERWRITE TABLE bank_partition Partition(region) SELECT
```

```
id,age,sexe,revenu,marie,enfant,voiture,hypothèque,region
```

```
INSERT OVERWRITE TABLE bank_bucket SELECT * ;
```

8. Lister (avec **ls -R**) le contenu du dossier HDFS **/user/cloudera/hive\_lab/data/bank\_partition**
9. Mettre hors ligne la partition **region=TOWN** de la table **bank\_partition**
10. Tester la requête: `select * from bank_partition;`
11. Rendre en ligne la partition **region=TOWN** de la table **bank\_partition**

## Hive: Atelier 3

---

12. Lister (avec **ls**) le contenu du dossier HDFS **/user/cloudera/hive\_lab/data/bank\_bucket**

13. Afficher chacun des deux fichiers de ce dossier.

Y a-t-il une valeur de la colonne **âge** qui apparaît dans les deux fichiers?

14. Créer la table externe **bank\_part\_bucket** partitionnée par **région** et chaque partition a **deux** Buckets par la colonne **âge** et triés par **âge** et **revenu**. Le dossier de données est **/user/cloudera/hive\_lab/data/bank\_part\_bucket**

```
CREATE EXTERNAL TABLE nom_table (.....)
PARTITIONED BY (....)
CLUSTERED BY (....) SORTED BY (...) INTO ... BUCKETS
ROW FORMAT ....
STORED AS ...
LOCATION ....;
```

15. Importer dans la table **bank\_part\_bucket** les données de la table **bank\_temp**.

16. Lister (avec **ls -R**) le contenu du dossier HDFS **/user/cloudera/hive\_lab/data/bank\_part\_bucket**

17. Visualiser le contenu de quelques fichiers de données.