Big Data

Required Install

- https://www.virtualbox.org/
- http://content.udacity-data.com/courses/ud617/Cloudera-Udacity-Training-VM-4.1.1.c.zip
- https://www.7-zip.org/download.html

Get Started Commands

- Créez un répertoire dans HDFS, appelé « myinput ». Pour cela, tapez : `hadoop fs -mkdir myinput`
- Pour copier le fichier purchases.txt dans HDFS sous le répertoire myinput, il s'agit e se placer dans le répertoire local data où se trouve le fichier, puis tapez la commande : `hadoop fs -put purchases.txt myinput/`
- Pour afficher le contenu du répertoire « myinput », la commande est : `hadoop fs -ls myinput`
- Pour visualiser les dernières lignes du fichier, tapez : `hadoop fs -tail myinput/purchases.txt`

Map Reduce

Hadoop computation model – Data stored in a distributed file system spanning many inexpensive computers – Bring function to the data – Distribute application to the compute resources where the data is stored • Scalable to thousands of nodes and petabytes of data

---

- Décompresser l'archive : `$ unzip td1_bigdata.zip`
- Changer le répertoire courant : `$ cd td1_bigdata`
- Compiler le programme Hadoop (le tout sur la même ligne) : `$ javac -classpath $(hadoop classpath)WCount*java`
- On va désormais packager le programme d'exemple au sein d'un fichier .jar. Créer l'arborescence liée au nom du package avec la commande : `$ mkdir -p istic/hadoop/wordcount`
- Et déplacer, par la suite, les fichiers compilés au sein de cette arborescence : `$ mv *.class istic/hadoop/wordcount`
- Générer le .jar : `$ jar -cvf istic_wcount.jar -C . istic`
- En préparation de l'exécution de notre programme Hadoop, nous allons maintenant déplacer le texte du poème sur HDFS. Exécuter la commande : `$ hadoop fs -put poeme.txt /`
- et vérifier sa présence avec la commande : `et vérifier sa présence avec la commande : $ hadoop fs -ls /`
- Enfin, exécuter notre programme Hadoop avec la commande : `$ hadoop jar istic_wcount.jar istic.hadoop.wordcount.WCount /poeme.txt /results`
- Vérifier la présence des fichiers de résultats dans le répertoire /results avec la commande : `$ hadoop fs -ls /results`
- 8. Enfin, afficher les résultats finaux avec la commande : `$ hadoop fs -cat /results/part-r-00000`

Manipulation Cloudera

- Depuis le terminal. Exécuter la commande suivante pour transformer le clavier en azerty `$ setxkbmap fr`
- Vérifier le bon fonctionnement de Hadoop en exécutant la commande: `$ hdfs dfsadmin -report`
- La commande vérifie le statut de HDFS. Elle devrait afficher: Live datanodes (1) :
- Vérifier la version d'Hadoop en exécutant la commande : `$ hadoop version`
- On va désormais packager le programme d'exemple au sein d'un fichier .jar. Créer l'arborescence liée au nom du package avec la commande: (Attention : cette commande ne pourra fonctionner que si vous la lancez depuis le même dossier contenant le fichier pom.xml: `$ cd WordCount/WordCount ) $ mvn clean package`
- En préparation de l'exécution du programme Hadoop, nous allons en premier temps créer dans le disque HDFS un dossier pour ce TP. Vous allez travailler dans la suite du TP dans un dossier nom.de.votre.dossier (à fixer) sur la racine du disque HDFS (Attention : le dossier nom.de.votre.dossier n'existe pas). On va créer dedans un dossier wordcount en exécutant la

commande suivante : `$ hadoop fs -mkdir -p /nom.de.votre.dossier/wordcount`

- Maintenant il faut déplacer le texte du poème du dossier data sur le disque HDFS dans le dossier créé. Exécuter la commande : `$ hadoop fs -put data/poeme.txt /nom.de.votre.dossier/wordcount`
- vérifier sa présence avec la commande : `$ hadoop fs -ls /nom.de.votre.dossier/wordcount`
- Enfin, exécuter votre programme Hadoop avec la commande ci-dessous sur la même ligne : (Attention : le jar créé est dans le dossier target: `$ cd target) $ hadoop jar WordCount-1.0-SNAPSHOT.jar WCount /nom.de.votre.dossier/wordcount/poeme.txt /nom.de.votre.dossier/resultat_wordcount`
- Vérifier la présence des fichiers de résultats dans le répertoire /resultat_wordcount avec la commande: `$ hadoop fs -ls /nom.de.votre.dossier/resultat_wordcount`
- Enfin, afficher les résultats finaux avec la commande: `$ hadoop fs -cat /nom.de.votre.dossier/resultat_wordcount/part-r-00000`

Spark

• Apache Spark is a fast and general engine for large-scale data processing.
• Spark has a variety of advantages including:
- Speed : Run programs faster than MapReduce in memory
- Easy to use : Write apps quickly with Java, Scala, Python, R
- Generality : Can combine SQL, streaming, and complex analytics
• Runs on variety of environments and can access diverse data sources Hadoop, Mesos, standalone, cloud...
HDFS, Cassandra, HBase, S3...

Spark API

Each Spark application consists of à driver program that execute the user's main function and initiates several parallel operations on the cluster. The main abstraction provided by Spark is an RDD (Resilient Distributed Dataset), which represents a collection of elements partitioned through the nodes of the cluster, and on which we can operate in parallel. RDDs are created from a file in HDFS for example, then transform it. Users can instruct Spark to save an RDD in memory, allowing it to be reused efficiently across multiple parallel operations.

> using **spark-shell**

- Créé un dossier nommé TPspark: `mkdir TPspark`
  - Se place dans le dossier TPspark : `cd TPspark`
  - wget permet de télécharger un fichier: `wget https://cedric.cnam.fr/vertigo/Cours/RCP216/sophie.txt`
  - Chargez ensuite le fichier sophie.txt dans HDFS : `Hadoop fs -put sophie.txt`
  - Maintenant lancez dans la fenêtre l'interpréteur de commandes spark-shell en entrant : `spark-shell`

Transformations and Actions

- charger le fichier text: `scala> var textsophie = sc.textFile ("sophie.txt")`
- Il est possible maintenant d'appliquer à ce RDD des actions, qui produisent des valeurs. Quelques actions simples : `scala> textsophie.count()` // Nombre de documents dans ce RDD `scala> textsophie.first()` // Premier document du RDD `scala> textsophie.collect()` // Récupération du RDD complet

Passons aux transformations. Elles prennent un (ou deux) DataFrame en entrée, produisent un DataFrame en sortie. Pour construire un Dataset comprenant seulement les lignes de texteSophie qui contiennent « poupée », et retourner un tableau avec ses 2 premières lignes on exécute cette commande : - `Scala> val lignesAvecPoupee = texteSophie.filter(line => line.contains("poupée"))` - `Scala> lignesAvecPoupee.take(2) // tableau avec les 2 premières lignes de ce Dataset`

Pour construire un Dataset comprenant les longueurs des lignes de texteSophie et retourner un tableau avec ses 5 premières lignes on exécute cette commande :

- `Scala> val longueursLignes = texteSophie.map(l => l.length)`
- `Scala> longueursLignes.take(5)` // tableau avec les 5 premières lignes de ce Dataset

On peut combiner une transformation et une action. En fait, avec Scala, on peut chaîner les opérations et ainsi définir très concisément le workflow. - `Scala > texteSophie.filter(line => line.contains("poupée")).count()`

Terms Counter

On crée un premier RDD constitué de tous les termes : - Scala> val termes = texteSophie.flatMap({ line => line.split(" ") }) // décompose une chaine de caractères - Scala > termes.collect()

On introduit la notion de comptage : chaque terme vaut 1. L'opérateur map produit un document en sortie pour chaque document en entrée. On peut s'en servir ici pour enrichir chaque terme avec son compteur initial. - scala> val termeUnit = termes.map({word => (word, 1)}) - scala> termeUnit.collect()

L'étape suivante regroupe les termes et effectue la somme de leurs compteurs : c'est un opérateur reduceByKey. - scala> val compteurTermes = termeUnit.reduceByKey({(a, b) => a + b}) - scala> compteurTermes.collect()

On peut faire toute en une fois `:paste <br> val compteurTermes = textsophie.flatMap({ line => line.split(" ") })<br> .map({ word => (word, 1) })<br> .reduceByKey({ (a, b) => a + b })<br> compteurTermes.collect()`

Finalement, si on souhaite conserver en mémoire le résultat final il suffit de faire:

- `scala> compteurTermes.persist()`

TEST Spark

Dans le but de tester l'exécution de spark, commencer par créer un fichier file1.txt dans votre noeud master, contenant le texte suivant :

Hello Spark Wordcount! Hello Hadoop Also :)

Charger ensuite ce fichier dans HDFS: - `hadoop fs -put file1.txt`

Maintenant taper la commande suivante: - `spark-shell`

Vous pourrez tester spark avec un code scala simple comme suit (à exécuter ligne par ligne): - `scala> val lines = sc.textFile("file1.txt")` - `scala> val words = lines.flatMap(_.split("\\s+"))` - `scala> val wc = words.map(w => (w, 1)).reduceByKey(_ + _)` - `scala> wc.saveAsTextFile("file1.count")`

Pour afficher le résultat, sortir de spark-shell en cliquant sur Ctrl-C. Télécharger ensuite le répertoire file1.count créé dans HDFS comme suit : - `hadoop fs -get file1.count`

Puis exécuter cette commande : - `cat file1.count/part-00000`

Actions SAMPLE

Action Dans les exemples précédents, count, first, take, collect et reduce sont des exemples d'actions. Voici ci-dessous quelques actions parmi les plus utilisées. - `reduce(func)` | Agréger les éléments du Dataset en utilisant la fonction func (qui prend 2 arguments et retourne 1 résultat). La fonction devrait être associative et commutative pour être correctement calculée en parallèle. - `collect()` | Retourner toutes les lignes du Dataset comme un tableau au programme driver. À utiliser seulement si le Dataset a un volume faible (par ex., après des opérations de type filter très sélectives).- count() | Retourner le nombre de lignes du Dataset. - `head(n)` | Retourner un tableau avec les n premières lignes du Dataset. - `take(n)` | Alias pour head(n). - `first()` | Retourner la première ligne du Dataset (équivalent à take(1) ou head()). - `show(n)` | Afficher les n premières lignes du Dataset sous forme de tableau. - `foreach(func)` | Appliquer la fonction func à toutes les lignes.

Transformation SAMPLE

- `map(func)` | Retourne un nouveau Dataset obtenu en appliquant la fonction func à chaque ligne du Dataset de départ.
- `filter(func)` | Retourne un nouveau Dataset obtenu en sélectionnant les lignes de la source pour lesquelles la fonction func retourne « vrai ».
- `flatMap(func)` | Similaire à map mais chaque ligne du Dataset source peut être transformée en 0 ou plusieurs lignes ; retourne une séquence (Seq) plutôt qu'une seule ligne. |- sample(withReplacement, | Retourne un Dataset contenant une fraction aléatoire fraction du Dataset auquel la transformation s'applique, avec ou sans remplacement, avec une seed pour le générateur aléatoire.
- `union(otherDataset)` | Retourne un Dataset qui est l'union des lignes du Dataset source et du Dataset argument (otherDataset).

- `intersection(otherDataset)` | Retourne un Dataset qui est l'intersection des lignes du Dataset source et du Dataset argument (otherDataset).
- `distinct()` | Retourne un Dataset qui est obtenu du Dataset source en éliminant les doublons des lignes.
- `groupByKey(func)` | Pour un Dataset, calcule la clé par func et retourne un KeyValueGroupedDataset.
- `join(otherDataset, joinExprs,` | Pour un Dataset jointure avec otherDataset de type joinType avec condition de jointure joinExprs.
- `crossJoin(otherDataset)` | Pour Dataset de type T et otherDataset de type U, retourne un Dataset de type (T, U) (produit cartésien).

Hbase

HBase is a columnar datastore, which means that the data is organized in columns as opposed to traditional rows where traditional RDMBS is based upon. - HBase has this concept of column families to store and retrieve data. HBase is great for large datasets, but, not ideal for transactional data processing. - This means if you have use cases where you rely on transactional processing, you would go with a different datastore that has the features that you need. The common use for HBase is if you need to perform random read/write access to your big data.

---

1. Lancez hbase shell à l'aide de la commande : `hbase shell`
2. Créez une table HBase avec trois familles de colonnes à l'aide de la commande create : `hbase(main):001:0> create 'table_1', 'column_family1', 'column_family2', 'column_family3'`
3. Vérifiez les propriétés par défaut utilisées pour les familles de colonnes à l'aide de la commande describe :
4. Spécifiez la compression pour une famille de colonnes dans la table. Pour HBase packagé avec BigInsights, seule la compression gzip peut être utilisée prête à l'emploi. Utilisez la commande alter. Pour modifier une table, elle doit d'abord être désactivée.

hbase(main):003:0> disable 'table_1' `hbase(main):004:0> alter 'table_1', {NAME => 'column_family1', COMPRESSION => 'GZ'}`

5. Spécifiez l'option IN_MEMORY pour une famille de colonnes qui sera fréquemment interrogée. `hbase(main):005:0> alter 'table_1', {NAME => 'column_family1', IN_MEMORY => 'true'}`
6. Spécifiez le nombre requis de versions comme 1 pour la famille de colonnes. `hbase(main):006:0> alter 'table_1', {NAME => 'column_family1', VERSIONS => 1}`
7. Exécutez de nouveau la description par rapport au tableau pour vérifier les modifications. Avant de pouvoir charger des données, vous devez activer la table. `hbase(main):007:0> enable 'table_1'` `hbase(main):008:0> describe 'table_1'`
8. Insérez les données à l'aide de la commande put. Chaque ligne peut avoir un ensemble différent de colonnes. Vous trouverez ci-dessous un ensemble de commandes put qui insèreront deux lignes avec un ensemble différent de noms de colonnes. La syntaxe de la commande put est put 'table' 'clé' 'famille:colonne' valeur.
```
hbase(main):009:0> put 'table_1', 'row1', 'column_family1:c11', 'r1v11'
hbase(main):010:0> put 'table_1', 'row1', 'column_family1:c12', 'r1v12'
hbase(main):011:0> put 'table_1', 'row1', 'column_family2:c21', 'r1v21'
hbase(main):012:0> put 'table_1', 'row1', 'column_family3:c31', 'r1v31'
hbase(main):013:0> put 'table_1', 'row2', 'column_family1:d11', 'r2v11'
hbase(main):014:0> put 'table_1', 'row2', 'column_family1:d12', 'r2v12'
hbase(main):015:0> put 'table_1', 'row2', 'column_family2:d21', 'r2v21'
```
9. Vérifiez que les deux lignes ont été insérées à l'aide de la commande count. La commande count fonctionne pour les petites tables. `hbase(main):016:0> count 'table_1'`
10. Pour afficher les données, vous pouvez utiliser get, qui renvoie les attributs d'une ligne spécifiée, ou scan qui permet l'itération sur plusieurs lignes. `hbase(main):017:0> get 'table_1', 'row1'`
11. Maintenant, exécutez la commande scan pour voir une liste de valeurs par ligne. `hbase(main):018:0> scan 'table_1'`
12. HBase n'a pas de commande ou d'API de mise à jour. Une mise à jour est identique à celle d'un autre ensemble de valeurs de colonne pour la même clé de ligne. Mettez à jour les valeurs dans : column_family1 (avec VERSIONS => 1) et

column_family2 (avec VERSIONS => 3).

```
hbase(main):019:0> put 'table_1', 'row1', 'column_family1:c11', 'n1v11'
hbase(main):020:0> put 'table_1', 'row2', 'column_family1:d11', 'n2v11'
hbase(main):021:0> put 'table_1', 'row2', 'column_family2:d21', 'n2v21'
```

La commade scan affichera les valeurs mises à jour des lignes. `hbase(main):022:0> scan 'table_1'`

13. Affichez les valeurs avec les différentes versions stockées pour column_family2.

```
hbase(main):023:0> scan 'table_1', {VERSIONS => 2}
```