INTRODUCCION

Hadoop → Framework opensource para almacenar datos y ejecutar aplicaciones en clusters de hardware basicos (Spark en memoria). Programado en Java

- Escalable horizontalmente
- Poco coste
- Tolerante a fallos

Se compone de:

- HDFS: Almacena los datos
- MapReduce: Procesa los datos
- Ecosistema de Herramientas

Un cluster es un conjunto de servidores (nodos) que trabajan juntos

- Maestros (gobiernan el cluster)
- Esclavos (procesan y almacenan la informacion)

Cada nodo tiene sus Demons

HDFS: Provee de almacenamiento redundante de grandes volumenes de datos transparente al usuario. Conviene usar cuando hay muchos datos

Usar HDFS en la linea de comandos:

- Copiar un fichero de disco local a HDFS
  hadoop fs -put /…/foo.txt /…/foo.txt

- Listar directorios
  hadoop fs -ls

- Listar directorio root
  hadoop fs -ls /

- mostrar contenido de un fichero
  hadoop fs -cat /user/fred/bar.txt

- Copiar fichero de HDFS al local
  hadoop fs -get /user/fred/bar.txt baz.txt

- Crear directorios en home
  hadoop fs -mkdir input

- Borar directorio Y TODO SU CONTENIDO
  hadoop fs -rm -r input_old

HIVE

¿Que es?

- Infraestructura para el **almacenaje** y **consulta** de datos basada en **hadoop**
- Diseñado para consultar y analizar **GRANDES** volumenes de datos
- Lenguaje **HQL**
- **Latencia muy alta** (al ser un sistema de procesamiento por lotes)

Tipos de dato (Tinyint, smallint, int, float, double, decimal, timestamp, date, string, varchar, char, boolean, arrays, maps, structs…)

Crear db: CREATE DATABASE my_db
USE my_db
DROP DATABASE my_db

Crear tabla: CREATE TABLE page_view (viewTime INT, userid BIGINT, page_url STRING, referrer_url STRING, friends ARRAY<BIGINT>, properties MAP <STRING, STRING>, ip STRING COMMNET 'IP Address')
COMMENT 'esta es la tabla'
PARTITIONED BY (dt STRING, country STRING)
CLUSTERED BY (userid) SORTER BY (viewTime) INTO 32 BUCKETS
ROW FORMAT DELIMITED
FIELDS TERMINATED BY '1'
COLLECTION ITEMS TERMINATED BY '2'
MAP KEYS TERMINATED BY '3'
LINES TERMINATED BY '4'
STORED AS SEQUENCEFILE;

Mostrar tablas: SHOW TABLES;
Mostrar tablas que comiencen por x: SHOW TABLES 'x.*';
Mostrar particiones: SHOW PARTITIONS page_view;
Mostrar informacion de tabla: Describe page_view;
Cambiar nombre de tabla: ALTER TABLE old RENAME TO new;
Modificar: ALTER TABLE old REPLACE COLUMNS (col1 TYPE…);
Añadir columnas: ALTER TABLE tab1 ADD COLUMNS (cq INT COMMENT 'a new int column', c2 STRING DEFAULT 'def val');
Borrar tabla: DROP TABLE pv_users;
Borrar particion: ALTER TABLE pv_users DROP PARTITION (ds = '2008-08-08');
Copiar datos de un fichero a una tabla: LOAD DATA LOCAL INPATH '/tmp/…' INTO TABLE page_view PARTITION(date='2008-06-08', country='US')
Mover datos de un fichero HDFS a una tabla: Igual pero sin LOCAL
Sobrescribir la tabla/ficheros en destino: LOAD DATA INPATH '/tmp/…' OVERWRITE INTO TABLE page_view PARTITION(...);

Exportar datos de una tabla a un fichero localINSERT OVERWRITE LOCAL DIRECTORY '/tmp/…'
SELECT … FROM … WHERE …

CONSULTAS = Como en SQL

PIG

Al igual que hive, PIG permite manejar grandes cantidades de datos gracias a su Paralelismo.
Pig Latin es un lenguaje de flujo de datos.

Cargar datos: allsales = LOAD 'sales' AS (name, price)
Con delimitadores:
allsales = LOAD 'sales.csv' USING PigStorage(',') AS (name, price);
allsales = LOAD 'sales.txt' USING PigStorage('|');
obtener salida: DUMP result;
almacenar datos en HDFS: STORE bigsales INTO 'myreport';
Con delimitadores:
STORE bigsales INTO 'myreport' USING PigStorage(',');

Tipos de datos:
Mejor especificar el tipo de datos:
allsales = LOAD 'sales' AS (name:chararray, price:int);

Filtrar registros invalidos:
hasprices = FILTER records by price IS NOT NULL;
Otro filtros:
bigsales = FILTER allsales BY price > 3000;
somesales = FILTER allsales BY name == 'Dieter' OR (price > 35 AND price < 40);
alices = FILTER allsales BY name == 'Alice';
a_names = FILTER allsales BY name MATCHES 'A.*';
Extraccion de columnas
twofields = FOREACH allsales GENERATE amountm trans_id;
crear campo basado en precio:
t = FOREACH allsales GENERATE price * 0,07 AS tax:float;
DISTINCT: unique_records = DISTINCT all_alices;

Funciones basicas:
UPPER(country)
TRIM(name)

RANDOM()
ROUND(price)
SUBSTRING(name, 0, 2)

IMPALA

¿Que es?
Es un motor SQL pensado para operar sobre grandes volumenes de datos. Corre sobre clusters Hadoop
Es mas rapido que Hive o Pig

SQOOP

Es una herramienta para transferir datos entre RDBMs y Hadoop (HDFS)

- Tranferir una o todas las tablas en una BBDD
- Transferir partes de una tabla

Se puede utilizar una interfaz JDBC
Los datos son importados a HDFS como text files o sequenceFiles (por defecto text files separados por comas)

Sintaxis básica:
sqoop tool-name [tool-option]

tool-name = import, import-all-tables, list-tables…
tool-options = –conect, –username, –password…

Ejemplo (importar una tabla empleados de una bbdd “personal” de los empleados con mas de 35 años
sqoop import –username user –password pass \
–connect jdbc:mysql://database.example.com/personal \
–table empleados –where “edad>35” –target-dir XXX

Otras cosas:
-p: Recoge la password
–password-alias: indica el fichero donde está almacenado la contraseña

Queries especificas con –query
sqoop import [% argumentos %] –query ‘SELECT a.*, b.* FROM a join b on (a.id == b.id) WHERE $CONDITIONS’ –target-dir /user/foo/joinresults

Exportar datos desde HDFS e insertarlos en una tabla en RDBMS
sqoop export [options]

Tambien es posible importar datos a hive en vez de HDFS
sqoop import –connect jdbc:mysql://db.foo.com/corp –table EMPLOYEES –hive-import
api completa: https://sqoop.apache.org/

FLUME

Flume es un sistema distribuido para recopilar, agregar y mover de forma eficiente grandes cantidades de datos de varias fuentes a un almacen de datos.

- Arquitectura sencilla
- Tolerante a fallos
- Transaccionalidad en la comunicación entre sus componentes

Se compone de:

- Event: Unidad de dato que se va propagando a traves de la arquitectura
- Source: origen de los datos
- Sink: destino de los datos
- Channel: buffer que conecta el source con el sink

Ejemplo flume:

```
#Definir Source, Channel y Sink
a1.sources = r1
a1.sinks = r1
a1.channels = c1

#Configuramos Source
a1.sources.r1.type = netcat
a1.soures.r1.bind = localhost
a1.sources.r1.port = 44444

#Configuramos en sink
a1.sinks.k1.type = logger

#Configuramos el channel
a1.channels.c1.type = memory
a1.channels.c1.capacity = 1000
```

```
a1.channels.c1.transactionCapacity = 100

#Unimos source y sink a traves del channel
a1.sources.r1.channels = c1
a1.sinks.k1.channel = c1
```

SINKS PREDEFINIDOS

- HDFS: almacena eventos en el sistema de ficheros de Hadoop. Permite almacenar ficheros en formato text y sequenceFile. Permite compresión.
- Hive: almacena eventos con formato texto o Json en tablas o particiones de Hive. Utiliza la transaccionalidad de Hive. (Not production ready)
- Logger: utiliza el nivel de log INFO para guardar los eventos
- Avro: almacenados sobre un host/port de Avro
- Thrift: almacenados sobre un host/port de Thrift (framework multilenguaje)
- IRC: utiliza el sistema de IRC channels
- FileRoll: almacena eventos en el sistema de ficheros local.
- Null: descarta los eventos.
- Hbase: almacena eventos en una base de datos Hbase. Se necesita utilizar un serializer de Hbase específico. Se puede tener autenticación mediante Kerberos.
- MorphlineSolr:transforma los eventos a partir de una configuración y los almacena en un motor de búsqueda Solr
- ElasticSearch: almacena los eventos en ElasticSearch
- Kite Dataset: permite almacenar eventos en Kite (hadoop layer for speedup dev.)
- Kafka: se puede publicar los eventos en un topic de Kafka
- Custom: se pueden construir sumideros específicos

CHANNELS PREDEFINIDOS

- Memoria: Los eventos se almacenan en una cola de memoria de tamaño predefinido. Útil cuando se queremos un throughput alto y podemos recuperar los eventos.
- JDBC. Los eventos son persistidos sobre una base de datos. Se necesita definir el driver, la url de conexión, …
- Kafka: los eventos son persistidos en un cluster de Kafka. Nos proporciona alta disponibilidad y replicación.
- File: los eventos son guardados en un fichero en el local system.
- Spillable Memory: Los eventos se guardan por defecto en una cola en memoria. Si el número de eventos almacenados sobrecarga la cola, éstos se pueden guardar en disco.
- Pseudo Transaction: utilizado para testing.
- Custom Channel: Implementación propia.

INTERCEPTORES PREDEFINIDOS

- Timestamp: inserta un timestamp en las cabecera de los eventos. •Host: añade el host o la IP al evento.
- Static: añade una cabecera 'fija' a los eventos.
- UUID: añade un identificador único a la cabecera.
- Morphline: permite hacer un transformación predefinida en un fichero de configuración
- Search&Replace: busca una cadena en el evento y la reemplaza por otra cadena.
- Regex: utiliza una expresión regular sobre la que matchear.

SPARK

Spark es mucho mas rapido, productivo y experimental que Hadoop
Tambien es mas expresivo y sencillo, y está muy demandado.

Spark sigue siendo tolerante a fallos y escalable.
Almacena los datasets en memoria, mientras que Hadoop lo hace en disco.

RDDs

RDD: Resilient Distributed Dataset
- Resiliente: Podemos recuperar los datos en memoria si se pierden
- Inmutable: No podemos realizar modificaciones sobre el mismo RDD
- Distribuido: Cada RDD se divide en multiples particiones.

Existen 2 tipos de operaciones sobre los RDD:
- Transformaciones: operaciones que crean un nuevo RDD. Spark no proceso las transformaciones hasta que no se ejecuta una accion sobre ellos y el RDD resultado no es inmediatamente computado.
- Acciones: Operaciones que devuelven un resultado. No son perezosas y el resultado no es inmediatamente computado.

Spark Context para comunicarse con el cluster

Parallelize: Convierte Scala Collection (sc) en un RDD
TextFile: Lee un fichero y lo transforma en un RDD de tipo String

Transformaciones:
- **map():** aplica una funcion sobre cada linea del RDD

- **filter():** Toma una funcion y la aplicado a los elementos del RDD que cumplen el filtro. Como el “where” en SQL
- **flatMap():** Es como el map (se aplica a cada elemento del RDD) pero en lugar de devolver un solo resultado compuesto por el resultado de aplicar dicha funcion a cada elemento, se itera sobre cada elemento inicial y se devuelve un valor por cada iteracion. (Ej: Teniendo como interentrada una frase, devuelvo una lista de palabras)

Acciones:

- reduce(): Opera sobre 2 elementos del RDD y devuelve un resultado (Ej: Suma, cuentas o agregaciones “val sum = rdd.reduce((x,y) => x + y)
- collect(): devuelve el dataset completo (Solo para pequeños datasets)
- count(): cuenta el numero de elementos (filas) de un RDD
- take(n): devuelve un array de n elementos
- saveAsTextFile(text): guarda el RDD a un fichero de texto

Otras transformaciones tipicas:

- distinct(): Borra los duplicados
- union(): Produce un RDD conteniendo elementos de ambos 2 RDDs
- intersection(): Produce un RDD conteniendo elementos que se encuentran en los 2 RDD
- substract(): Borra el contenido de un RDD
- cartesian(): Producto cartesiano con el otro RDD

PAIR RDDs: RDDs con formato Clave-Valor
Transformaciones y acciones sobre Pair RDDs:

- **groupByKey:** Aplica una función a un grupo de elementos y devuelve un resultado de tipo map.
  Ejemplo:
  val ages = List(2, 52, 44, 23, 64)
  val grouped = ages.groupBy { age =>
  if (age >= 18 && age < 65) “adult”
  else if (age < 18) “child”
  else “senior”
  }

reduceByKey: Toma una función y únicamente tiene en cuenta los V del PairRDD. Esto es así porque se da por hecho que los valores ya estan agrupados por K
Ejemplo:
val eventsRDD = sc.parallelize(...).map(event => (event.organizer, event.budget))
val budgetRDD = eventsRDD.reduceByKey(_+_)
budgetRDD.collect().foreach(println)

mapValues: Aplica una función sobre los valores de un pairRDD

Ejemplo contar palabras con reduceByKey
val input = sc.textFile("s3://…")
val words = input.flatMap(x => x.split(" "))
val results = words.map(x => (x, 1)).reduceByKey((x,y) => x + y)

countByKey: Acción. devuelve un Map(K,V) con K = la clave del MAP sobre el que se aplicó la función y V = el número de elementos para esa clave

keys: No tiene argumentos. Devuelve un RDD con las K de cada PairRDD

Ejemplo: contar el numero de visitantes distintos y unicos de una web
val visits:RDD[Visitor] = sc.textFile(...).map(v => (v.ip,v.duration))
val numUniqueVisits = visits.keys.distinct().count()

sortByKey(): recibe un parametro que indica si queremos que el oden sea ascendente o no.

Inner Join: devuelven un nuevo RDD que contiene la combinacion de PairRDD cuyas K estan en ambos PairRDD

Ejemplo:
val tracked = abos.join(locations)

Spark SQL

DataFrame: Abstracción que representa el equivalente a una tabla necesitamos SparkSession (import org.apache.spark.sql.SparkSession)

Crear SQLContext:

• var ssc = new org.apache.spark.sql.SQLContext(sc)

Importa implicits que permiten convertir RDDs en DataFrames

• import sqlContext.implicits._

Cargar dataset

• var zips = ssc.load("file:/home/BIT/data/zips.json", "json")

Visualiza los datos

• zips.show()

Obtén las filas cuyos códigos postales cuya población es superior a 10000 usando el api de DataFrames

• zips.filter(zips("pop") > 10000).collect()

O

• ssc.sql("select * from zips where pop > 10000").collect()

Guarda tabla en fichero temporal

• zips.registerTempTable("zips")

CONSULTAS SQL

ssc.sql("select SUM(pop) as POPULATION from zips where state='WI'").show()

ssc.sql("select * from zips where pop > 10000").collect()

ssc.sql("select state, SUM(pop) as POPULATION from zips group by state order by SUM(pop) DESC ").show()

Spark Streaming

Ejercicio: Spark Streaming I (modulo 6.1)

- con el comando “**nc -lkv 4444**”, todo lo que escribas se envíe al puerto 4444
- arrancar el shell de spark con 2 threads: **“spark-shell --master local[2]”**
- **Importar las clases:**

  import org.apache.spark.streaming.StreamingContext

  import org.apache.spark.streaming.StreamingContext._

  import org.apache.spark.streaming.Seconds

- Crear un **SparkContext** con una duración de 5 segundos

  var ssc = new StreamingContext(sc,Seconds(5))

- Crear un **DStream** para leer texto del puerto que pusiste en el comando “nc”

  var mystream = ssc.socketTextStream("localhost",4444)

- Crea un **MapReduce**, como vimos en los apuntes, para contar el número de palabras que aparecen en cada Stream

  var words = mystream.flatMap(line => line.split("\\W"))

  var wordCounts = words.map(x => (x, 1)).reduceByKey((x,y) => x+y)

- **Imprime** por pantalla los resultados de cada batch

  wordCounts.print()

- Arranca el **Streaming Context** y llama a **awaitTermination** para esperar a que la tarea termine

  ssc.start()

  ssc.awaitTermination()

- spark-shell --master local[2] -i prueba.scala