Software para el análisis de datos

Paralelismo y Sistemas Distribuidos

Tipos de software

- Modelos de programación y *runtimes*
- Almacenamiento de datos
- Procesado de datos

Requerimientos

- Se prioriza la escalabilidad horizontal y la tolerancia a fallos sobre el alto rendimiento
- Tipos de arquitectura para sistemas distribuidos
  - Cliente-Servidor (master-slave)
    - modelo *Thin-client*: el cliente sólo hace de interfaz con el usuario
    - modelo *Fat client:* el servidor sólo se encarga de la gestión de datos y el cliente se encarga de implementar la lógica de la aplicación
  - Peer-to-Peer: arquitectura decentralizada en la que cualquier nodo puede recibir peticiones e implementar el servicio
  - (…)

1.3

Modelos de programación

- Pensados para ejecutar las mismas funciones sobre una gran cantidad de datos → Computación guiada por los datos
  - Crear partición de datos
  - Ejecutar de manera concurrente la misma función sobre cada partición
- Ejemplos
  - Apache Hadoop
  - Spark

1.4

Modelos de programación: MapReduce

- Propuesto por Google e inspirado en los lenguajes de programación funcionales
- Pensado para aplicaciones masivamente paralelas: mismos cálculos sobre grupos independientes de datos
- El procesado de datos se divide en dos partes
  - Map:
    - Su entrada es una porción de datos identificados por una clave
    - Su resultado es un resultado (parcial) identificado por una clave que puede ser distinta a la de entrada
  - Reduce:
    - Es opcional y ejecuta la agregación de los resultados parciales (generados por la función de *map*)
    - Su entrada es una lista de valores generados por el map que tienen la misma clave

1.5

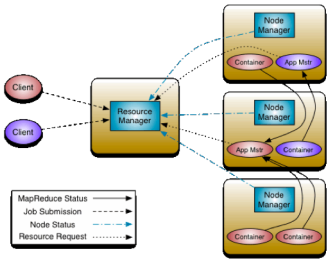
Frameworks para MapReduce

- El framework que implenta el modelo debe ofrecer
  - Interfaz para el particionado de datos
  - Gestión de ejecución y de paralelismo
  - Integración con almacenaje de datos
  - Están pensados para ejecutarse en datacenter/clouds de grandes dimensiones
  - Ofrecen escalabilidad y tolerancia a fallos
- Ejemplo: Apache Hadoop

1.6

Apache Hadoop: Arquitectura

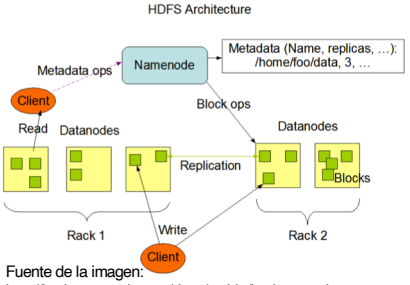
- Implementa un entorno de ejecución MapReduce junto con un sistema de ficheros distribuido (HDFS → Hadoop Distributed File System)
- Arquitectura del entorno MapReduce
  - Basada en Apache YARM (Yet Another Resource Manager)
  - Componentes
    - 1 ResourceManager (master): planificación de tareas y reinicia el container del master de la aplicación si falla
    - 1 NodeManager per node (slave): responsable de crear los *containers* necesarios para ejecutar la aplicación
    - 1 MRAppMaster por aplicacion: se ejecuta en un *container* y comprueba el estado de las tareas, reiniciando las que hayan fallado



Fuente de la imagen: https://hadoop.apache.org/docs/r2.7.2/hadoop-yarn/hadoop-yarn-site/YARN.html

Apache Hadoop: HDFS

- Objetivos
  - Cada nodo *Comodity* hardware
    - Pensado para clusters con gran cantidad de nodos
  - Tolerancia a fallos
  - Grandes conjuntos de datos
  - Tipo de uso: se escribe una sola vez y se leen muchas
    - Simplifica la gestión de coherencia y se centra en aumenter el throughput
- Arquitectura
  - Namenode
    - Master
    - Implementa el espacio de nombre y la asignación de bloques a Datanodes
  - Datanodes
    - Slave
    - Implementa los accesos a bloques



Fuente de la imagen: http://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html#Introduction

Apache Hadoop: HDFS

- Fichero divididos en bloques del mismo tamaño (excepto el último)
- Los bloques están distribuidos entre los Data nodes
- Replicación para dar toleracia a fallos
  - Política de asignación de réplicas es muy relevante
- Se intenta favorecer la localidad en el acceso a datos
  - Normalemente los DataNodes corren en los nodos de procesado
- Los ficheros sólo se pueden escribir una vez, y sólo se permite un escritor al mismo tiempo
- Tamaño de bloque y número de réplicas configurable a nivel de fichero
- Tolerancia a fallos del server
  - Multiples Namenodes (uno activo y el otro en *stand by)*
  - NameNode activo puede mantener varias copias de los metadatos
  - Sistema de ficheros compartido entre los NameNodes para acelerar el proceso de recuperación

1.9

Apache Hadoop: MapReduce

- Pasos de una aplicación MapReduce
  - Dividir los datos de entrada en porciones independientes
    - El entorno proporciona clases que implementan el particionado (InputFormat) pero cada usuario puede implementar la suya propia
  - Procesar cada una en paralelo (*map task*)
  - El entorno ordena los resultados parciales y se los envía a las tareas de *reduce* como datos de entrada
- Los datos de entrada y de salida están organizados en parejas <key,value>
  - conjunto de pares <key,value>
  - las clases de key y de value deben implementar el interfaz *Writable* (para ser serializables) y la clase de key también debe implementar *WritableComparable* para la fase de ordenación
- Flujo de datos:

  <k1,v1>→ **map**→<k2,v2>→**combine**→ <k2,v2>→**reduce**→<k3,v3>
- Mínima implementación: función de map y/o de reduce

1.10

Apache Hadoop: API

- map
  - Recibe como entrada un conjunto de pares <key,value> y produce como salida otro conjunto de pares <key,value>, que pueden ser de distintio tipo y/o valores
  - Se lanza un mapper para cada partición de los datos de entrada (*InputSplit*)
- combiner
  - Función opcional para agrupar valores de salida con la misma key de los mappers locales (para minimizar mensajes a los reducers). El input es la salida de los mappers, y el output los pares <key, lista_valores>
- reducer
  - La salida del map (o de los combiner) son ordenados y particionados para mandárselos a los reducers
  - El número de reducers no depende de los datos de entrada y es un parámetro configurable (se puede hacer que sean 0 si no se necesita fase de reduce)
  - El programador puede decidir cómo hacer las particiones para cada reducer si implementa su propio *partitioner*

1.11

Ejemplo de código: WordCount (I)

```
public class WordCount {

  /* here definition of TokenizerMapper.class and IntSumReducer

      public static void main(String[] args) throws Exception {
          Configuration conf = new Configuration();
          Job job = Job.getInstance(conf, "word count");
          job.setJarByClass(WordCount.class);
          job.setMapperClass(TokenizerMapper.class);
          job.setCombinerClass(IntSumReducer.class);
          job.setReducerClass(IntSumReducer.class);
          job.setOutputKeyClass(Text.class);
          job.setOutputValueClass(IntWritable.class);
          FileInputFormat.addInputPath(job, new Path(args[0]));
          FileOutputFormat.setOutputPath(job, new Path(args[1]));
          System.exit(job.waitForCompletion(true) ? 0 : 1);
      }

}
```

1.12

Ejemplo de código: WordCount (II)

```
public static class TokenizerMapper extends Mapper<Object, Text, Text, IntWritable>{
private final static IntWritable one = new IntWritable(1);
private Text word = new Text();
public void map(Object key, Text value, Context context ) throws IOException,
InterruptedException {
    StringTokenizer itr = new StringTokenizer(value.toString());
    while (itr.hasMoreTokens()) {
        word.set(itr.nextToken());
        context.write(word, one);
    }
}
}
```

1.13

Ejemplo de código: WordCount (III)

```
public static class IntSumReducer extends Reducer<Text,IntWritable,Text,IntWritable> {
private IntWritable result = new IntWritable();
public void reduce(Text key, Iterable<IntWritable> values, Context context ) throws
IOException, InterruptedException {
    int sum = 0;
    for (IntWritable val : values) {
        sum += val.get();
    }
    result.set(sum);
    context.write(key, result);
}
}
```

1.14

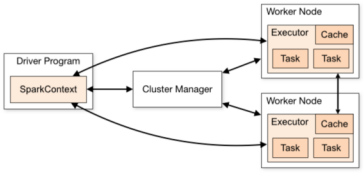
Apache Spark

- Implementa el modelo MapReduce
- Mejora el rendimiento de Hadoop ahorrando la escritura de resultados intermedios en disco (entre otras cosas)
- Mismos objetivos que Hadoop: tolerancia a fallos y escalabilidad
- Gestión transparente del paralalelismo
- Ofrece API en muchos lenguajes (Python, Java, Scala)

1.15

Apache Spark: arquitectura

- Arquitectura master-slave
  - *driver* se ejecuta en el master
    - analiza el código y pide los recursos necesarios
    - Asigna *tasks* a *executors* y monitoriza la ejecución para detectar y corregir erores
  - *cluster manager*
    - asigna los recursos y crea los procesos *executors* en los nodos exclavos
    - Spark soporta 3 tipos de gestión de clusters: standalone, basada en hadoop YARN y basada en Apache Mesos



Fuente de la imagen: https://spark.apache.org/docs/latest/cluster-overview.html

1.16

Apache Spark: resource scheduling

- Cuando varias aplicaciones intentan usar el cluster al mismo tiempo hay que decidir cómo se reparten los recursos del cluster
- Particionado estático: se le asigna a cada aplicación una partición de la máquina
  - Si se usa Mesos hay más flexibilidad en función de las necesidades de las aplicaciones
- Para standalone clusters
  - Por defecto una aplicación consume todos los cores disponibles y hasta que ella no acaba no se pasa a ejecutar la siguiente
  - Es posible indicar un número máximo de cores y/o de memoria y entonces el resto se puede aprovechar para otra aplicación

1.17

Apache Spark: componentes

- Spark Core: parte central, encargada del procesado
  - Gestión de memoria, recuperación de errores, planificación, asignación de trabajos, monitorización de la ejecución e interacción con los sistemas de almacenamiento
- Spark Streaming: procesado de datos en tiempo real
- Spark SQL: proporciona interfaz SQL
- GraphX: computación en grafos
- Mllib: machine learning
- SparkR: soporte para R

1.18

Apache Spark: API para el acceso a datos(I)

- Resilient Distributed Dataset (RDD)
  - Sólo de lectura
  - Conjunto de records
  - Los datos están distribuidos por todo el cluster, y tienen réplicas
  - Se particionan para que se puedan procesar en paralelo
    - El programador puede implementar su propio particionado si los datos no están balanceados
  - Se pueden usar para transformar en otro RDD (*transformation*) o para generar el resultado final (*Action*)
  - Lazy evaluation: se retrasa el momento del cálculo hasta que hace falta (cuando se aplica una *Action*)
  - Algunas *transformations*: map, filter, union, intersection,…
  - Algunas *actions*: count, countByValue,reduce,…
- Otros interfaces: Dataframes y Datasets
  - Pensados para datos estructurados, organizados por columnas

1.19

Apache Spark: API para el acceso a datos(II)

- Creación de RDD
  - A partir de colecciones en memoria

    data = [1, 2, 3, 4, 5]

    distData = sc.parallelize(data) → número de particiones decidio en función de la configuración del cluster (entre 2 y 4 por CPU)

    distData=sc.parallelize(data, 5) → se divide en 5 particiones
  - A partir de datos almacenados en disco (HDFS, Sistema de ficheros compartido, Base de datos….)

    distData=sc.TextFile(”data.txt”)
  - A partir de otros RDD, mediante una transformación
- Las particiones determinan el número de tasks (una por partición)

1.20

Apache Spark: pasos para crear un programa(I)

- Creacion de SparkContext
  - Recibe varios parámetros pero los más usados son la url del master a la que hay que conectarse y el nombre de la aplicación
  - La url puede ser el string “local”

```
from pyspark import SparkContext
sc = SparkContext("local", ”Nombre")
```

  - También se puede usar la clase SparkConf para configurar los parámetros del contexto

```
from pyspark import SparkConf, SparkContext
conf = SparkConf().setAppName(”Nombre").setMaster(”local")
sc = SparkContext(conf=conf)
```

- Creación del RDD que hará de input
  - A partir de un conjunto de datos en memoria

```
milista=[“esto”,”es”,”una”,”lista”,”de”,”palabras”]
palabras = sc.parallelize (milista)
```

  - A partir de datos externos (el fichero debe estar en el mismo path tanto para el master como para los workers)

```
rddFicheroLocal = sc.textFile(‘path’)
rddFicheroHDFS =sc.textFile(‘hdfs:///path’)
```

1.21

Apache Spark: pasos para crear un programa(II)

- Operaciones sobre los RDD
  - Ejemplos de acciones:
    1. palabras.count()
    2. contenido=palabras.collect()
    3. def f(x): print(x)
       contenido_filtrado=palabras.foreach(f)
    4. from operator import add
       nums = sc.parallelize([1, 2, 3, 4, 5])
       suma = nums.reduce(add)

    (…)
  - Ejemplo de transformaciones
    1. palabras_filtradas=palabras.filter(lambda x: ‘esto’ in x)
    2. palabras_map = palabras.map(lambda x: (x, 1))
  - Si queremos guardar el resultado en un fichero
    palabras_map.saveAsText(‘nombre_fichero’)

1.22

Ejemplo de código: WordCount

```
from pyspark import SparkContext
sc = SparkContext("local", "Nombre")
text_file = sc.textFile("hdfs://...")
counts = text_file.flatMap(lambda line: line.split(" ")).map(lambda word: (word, 1)).reduceByKey(lambda a, b: a + b)
counts.saveAsTextFile("hdfs://...")
```

1.23

Transformations y Actions

- *Transformations*: cálculo sobre un RDD que genera otro RDD
  - Dos tipos de transformaciones
    - Narrow: todos los datos necesarios están en la misma partición, se pueden resolver de manera local
    - Wide transformation: hacen falta datos de diferentes particiones, hace falta intercambio de datos
- *Actions*: genera un resultado que se guarda en almacenamiento o que se muestra en pantalla
- Las *actions* son las que dan por finalizada una cadena de transformaciones y las que desencadenan que se inicie el cálculo
- Por defecto, cada vez que se ejecuta una action sobre un RDD se vuelve a calcular el RDD. Se puede evitar usando el método persist.

```
lines = sc.textFile("data.txt")
lineLengths = lines.map(lambda s: len(s))
totalLength = lineLengths.reduce(lambda a, b: a + b)
maxLength = lineLengths.reduce(lambda a,b: max(a,b))
```

lineLengths.persist()

1.24

Algunas transformaciones

- map
- flatMap

```
def minusculas(lines):
  lines = lines.lower()
  lines = lines.split()
rdd1 = rdd.map(minusculas)
rdd2=rdd.flatmap(minusculas)
rdd1.take(5)
rdd2.take(5)
```

- filter

```
stopwords = ['is','am','are','the','for','a']
rdd3 = rdd2.filter(lambda x: x not in stopwords) rdd3.take(10)
```

- groupBy

```
rdd4 = rdd3.groupBy(lambda w: w[0:3])
```

- reduceByKey
- aggregateByKey
- sortByKey
- join

1.25

Algunas acciones

- collect
- count, max,min,sum…
- take
- reduce

1.26

Directed Acyclic Graph: tasks y etapas

- Driver crea un grafo que representa el procesado (*logical execution plan*)
  - Expresa dependencias entre RDD
    - Aristas trasformaciones
    - Vertices RDD
  - Cuando el driver se encuentra una *action* envía el grafo *al DAG scheduler*
- El DAGscheduler convierte el *logical execution plan* en *physical execution plan* basado en etapas:
  - Etapa: conjunto de transformaciones que se pueden resolver de manera local (sólo usa datos de una partición): narrow transformations
  - Las transformaciones locales se pueden realizar intentando usar sólo memoria
- El DAGscheduler manda las stages al task scheduler: el número de tasks depende del número de particiones y del número de transformaciones

1.27

Spark runtime engine (II)

- Por defecto: dejar que el sistema decida el número de particiones y el criterio para crearlas
- Opciones
  - Decido cuántas particiones→ influye en el número de tasks y en el grado de paralelismo

```
rdd=sc.parallelize(input, 15)
```

  - Elegir cómo agrupo elementos de la partición

```
rdd=sc.parallelize(input).map(lambda x:
x['country'],x).partitionBy(15,country_hash)
```

1.28

Spark Streaming (I)

- Extensión de Spark para soportar entrada de datos en tiempo real
- Ofrece la integración con software de generación de datos (Kafka, S3,…)
- La capa de streaming recibe los datos y genera grupos de datos (RDD's)
- Se basa en la abstracción Dstream (Discretized Stream)

Fuente de la imagen: https://spark.apache.org/docs/latest/streaming-programming-guide.html

1.29

Spark Streaming (II)

- DStream (Discretized Stream)
  - Representan un flujo de datos que se inicializa con los datos que se reciben o a partir de transformaciones hechas sobre otros Dstreams
  - Un DStream tiene por debajo una secuencia de RDD's
    - Cada RDD's contiene los datos recibidos en un cierto intervalo de tiempo (intervalo de batch)
    - El intervalo de tiempo se define al crear el DStream
  - Las operaciones hechas sobre los DStream se aplican a los RDD's que tienen debajo
- Cada DStream está asociado con un Receiver
  - Proceso de Spark que guarda lee los datos y los deja en memoria

1.30

Spark Streaming (III)

- Pasos para implementar una aplicación
  - Instanciar un *StreamingContext,* a partir de un *SparkContext* y el intervalo de *batch*
    - Importante: indicar al menos que se quieren crear tantos threads como receivers más uno para el procesado
  - Crear un DStream, que definirá de dónde vienen los datos
  - Definir la secuencia de transformaciones que se aplicarán a cada DStream (map, filter,…)
    - Las operaciones de salida son las que provocan que empiece la ejecución de las transformaciones (como las actions sobre los RDD's)
  - Iniciar la recepción y el procesado de datos
  - Esperar el final de datos

Spark Streaming (III)

- Ejemplo: WordCount

```
from pyspark import SparkContext
from pyspark.streaming import StreamingContext
sc = SparkContext("local[2]", "NetworkWordCount") #2 cores
ssc = StreamingContext(sc, 1) #intervalo de batch es 1 segundo
lines = ssc.socketTextStream("localhost", 9999) # crea el Dstream que recibe datos a través de un socket
#transformaciones
counts = lines.flatMap(lambda line: line.split(" ")).map(lambda word: (word, 1)).reduceByKey(lambda a, b: a + b)
counts.pprint()
ssc.start() # Empieza el procesado
ssc.awaitTermination()
```

Spark Streaming (IV)

- Para proporcionar los datos, en otra ventana hay que enviar datos al puerto 9999. Se puede hacer ejecutando el comando nc

```
$nc -lk 9999
hello world
```


Almacenamiento de datos

- Sistemas de almacenamiento tradicionales están basados en bases de datos relacionales
- Desventajas de estos sistemas para el almacenamiento de BigData
  - No gestionan bien grandes volúmenes de datos heterogéneos, sin esquema
  - No gestionan imágenes
  - Transacciones (ACID) –> alto coste cuando la cantidad de datos crece
  - Para ser eficientes necesitan hardware costoso y no escalan horizontalmente
- Alternativas
  - Sistemas de ficheros distribuidos (como HDFS o GFS)
  - Bases de datos NoSQL (Cassandra, BigTable, DinamoDB, MongoDB, CouchDB, …)

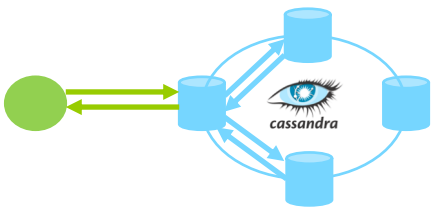
Bases de datos NoSQL

- Ventajas
  - Tolerancia a fallos
  - Soporte de datos no estructurados
  - Se puede ejecutar sobre comodity hardware
  - Escalables horizontalmente
  - Queries simples pero rápidas
- Desventajas
  - No hay un lenguaje estándar
  - Eficiciencia de la query es altamente dependiente de la organización de los datos
  - Falta trabajar más aspectos relacionados con la privacidad de los datos
- Características que suelen tener
  - Datos distribuidos y replicados
  - Consistencia eventual

1.35

Cassandra

- Basada en DynamoDB (la base de datos de Amazon)
- Clave-valor, orientada a columnas
- No hace falta definir un esquema de datos
- No hace falta que todos los records tengan la misma estructura
- Arquitectura peer-to-peer
  - Cualquier nodo puede recibir una petición de datos que reenvía al nodo que los tenga



1.36

Cassandra

- Particionado de datos: pretende distribuir los datos de manera uniforme
  - Función de hash (h) aplicada a la clave de particionado (pk)

- Replicación de datos para facilitar la tolerancia a fallos y la disponibilidad de los datos
  - A nivel de keyspace
  - Se puede seleccionar la estrategia para repartir las replicas
    - SimpleStrategy: La copia la tiene el siguiente nodo en el ring
    - NetworkTopologyStrategy: La copia la tiene el siguiente nodo en el ring que pertenezca a un rack distinto
- Eventualmente consistente

1.37

Teorema del CAP

- Un sistema distribuido no puede mantener al mismo tiempo las siguientes 3 propiedades:
  - **C**onsistency: Todos los nodos ven los mismos datos al mismo tiempo
  - **A**vailability: Se garantiza que toda petición recibe una respuesta
  - **P**artition Tolerance: El sistema es capaz de continuar aunque falle una parte
- Cassandra prioriza **AP**, aunque es posible configurarla para que sea consistente

1.38

Niveles de consistencia: escritura

- Quorum = (suma_factores_replicación/2)+1
- Nivel de consistencia comunes para escritura (cuándo se da por finalizada la operación)
  - All: cuando todos los nodos que guardan una réplica han completado la escritura
  - Each quorum: cuando se usan varios datacenters, si en cada datacenter hay quorum de escrituras acabadas
  - Quorum: si hay quorum considerando todos los datacenters
  - Local quorum: quorum considerando los nodos del mismo datacenter del coordinador (el que ha recibido la petición)
  - Any: al menos una réplica en el coordinador (si ningún encargado de la replica responde)
  - One, two, three, local one
- Por defecto, nivel de consistencia ONE

1.39

Niveles de consistencia: lectura

- All, Quorum,Local quorum, One, Two, Three, Local one, …
- Cuando el coordinador recibe todas las réplicas, si hay discrepancia en los datos lanza la operación *read_repair*

1.40

Cassandra

- Organización de datos en dos niveles: keyspace (parecido a un directorio) y tablas
- Modelo de datos:
  - primary key – valores
    - La primary key puede estar compuesta de una partition key y n clustering keys
  - La únidad para cada petición es la partición
    - Partición: todas las rows que comparten partition key
- Rendimiento muy influido por el modelo de datos
  - La partition key define la repartición de datos entre nodos
  - Normalmente se definen las queries que se quieren implementar y a partir de ahí se organizan los datos

1.41

Modelo de datos en Cassandra

- Objetivos al definir el modelo de datos
  - Distribuir los datos uniformemente entre los nodos (maximiza el potencial paralelismo)
  - Reducir el número de particiones leídas por query
    - Particiones distintas pueden involucrar diferentes nodos
    - Aumenta el trabajo de coordinación de la query
  - La duplicación de los datos “no importa”
  - Las condiciones de filtrado son más eficientes cuando involucran a la partition key

1.42

Modelo de datos en Cassandra

- Ejemplo: agenda de clientes
  - Datos que guardamos
    - DNI
    - Nombre
    - Dirección postal
    - Ciudad
    - País
    - Número de télefono
  - Para la query: datos de un usuario determinado
    - Partition key: DNI
    - Todos los datos estarán repartidos uniformemente y para esta query sólo accedemos a una partición

1.43

Modelo de datos en Cassandra

- Para la query dame los datos de todos los clientes que viven en una ciudad determinada
  - Para que la query sea eficiente, los campos de búsqueda deben formar parte de la primary key
  - Opciones
    - Primary key: (DNI, ciudad), partition key DNI
      - La query involucra tantas particiones como clientes vivan en la ciudad
    - Primary key: (ciudad, DNI), partition key ciudad
      - La query involucra una sola partitición, pero si las ciudades tienen muchos clientes o si están muy desbalanceadas los datos no estarán distribuidos uniformemente
      - Dividir la partición, por ejemplo rangos de DNI's

1.44

Acceso a Cassandra

- Path de escritura
  - Escritura en el commit log (fichero local) y en la Memtable (tabla en memoria)
  - Cada cierto número de escrituras la Memtable se vuelca en la SSTable (tabla en disco)
  - Hay una Memtable por cada tabla
  - Las SSTable son inmutables (datos de la misma tabla en diferentes SSTables) → compaction
  - Si se modifica un dato, se guarda en otra posición. Durante la compaction se queda con las últimas versiones del dato
  - Para borrar un dato, se marca pendiente de borrar durante un cierto tiempo, para tratar con nodos que tienen una réplica y no responden
- Path de lectura
  - Se leen particiones enteras, si sólo interesan unas columnas se filtra en memoria
  - Primero se comprueba si los datos están en la Memtable
  - También es posible usar una cache de rows, para las rows que se reutilizan
  - Estructuras de datos en memoria para intentar acelerar la localización del dato

1.45

Interfaz de Cassandra

- Cassandra se puede usar con una herramienta interactiva (cqlsh) o programáticamente
  - Interfaz para python, java, C++, Scala, perl,…
  - Se puede integrar con Spark
- Algunos comandos
  - Crear/Borrar keyspace
  - Crear/Borrar/Truncar tablas
  - Consultar características de keyspaces o tablas
  - Insertar datos en las tablas (o cargarlas desde un fichero .csv o .json)
  - Consultar datos de las tablas
  - Modificar características de las tablas o de los keyspaces
- Documentación: https://docs.datastax.com/en/ddac/doc/

1.46

Interfaz interactivo de Cassandra: cqlsh

- Operaciones sobre KeySpace

```
describe keyspaces;
create keyspace psd WITH replication = {'class': 'SimpleStrategy', 'replication_factor': 2};
describe keyspace psd;
drop keyspace psd;
```

- Operaciones sobre tablas

```
create table PSD.tweets (time timestamp , user text, tweet text, primary key (user,time));
describe table psd.tweets;
insert into psd.tweets (time, user, tweet) values ( 1,'usuario1','empezando el dia');
copy psd.tweets from 'pathtocsvfilev'  with delimiter = ',';
select * from psd.tweets where user='usuario1';
truncate table psd.tweets;
drop table psd.tweets;
```

1.47

Interfaz de Cassandra: driver de Python (I)

- Pasos para usar Cassandra desde un código en python
  - Inicializar el cluster, configurando comportamiento del driver de Cassandra (por ejemplo, política para elegir el nodo de contacto de Cassandra), pasando la IP de algún nodo de Cassandra (si no es localhost) y el puerto (si no es el de por defecto)

```
from cassandra.cluster import Cluster
from cassandra.policies import DCAwareRoundRobinPolicy
cluster = Cluster( [IP1,IP2],
                   load_balancing_policy=DCAwareRoundRobinPolicy(),
                   PUERTO=NUM_PUERTO)
```

  - Crear la conexión

```
session = cluster.connect()
```

  - Opcionalmente, se puede especificar el keyspace con el que se quiere trabajar (también como parrametro en el connect)

```
session.set_keyspace(NOMBRE_KEYSPACE)
```

  - Ejecutar queries:

```
rows = session.execute('SELECT user FROM 'psd.tweets')
for row in rows:
   print row.name, row.age, row.email
session.execute( "INSERT INTO tweets (user, time, tweet) VALUES (%s, %d, %s) ",
(1,'usuario1','empezando el dia") )
```

1.48

Interfaz de Cassandra: driver de Python (II)

- Se puede pasar el nivel de consistencia que se quiere

```
session.execute( """ INSERT INTO users (name, age, email) VALUES (%s, %d, %s),
consistency_level=ConsistencyLevel.QUORUM """, ("John O'Reilly", 42, "dummy@gmail.com") )
```

- Se pueden ejecutar queries asíncronas. Dos maneras de gestionarlas:
  1. Cuando se van a usar los datos se comprueba si han llegado y si no es así se bloquea esperando

```
query = "SELECT * FROM psd.tweets WHERE user=%s"
id=session.execute_async(query, ['usuario1'])
…
rows=id.result()
```

  2. Se especifica la rutina que queremos ejecutar cuando lleguen los datos y el driver la invoca (*callback*)

```
def tratar_datos(rows):
  hago_algo(rows[0])
def tratar_error(excepcion):
  tratar_error(excepcion)
query = "SELECT * FROM psd.tweets WHERE user_id=%s"
id= session.execute_async(query,['usuario1'])
id.add_callbacks(tratar_datos, tratar_error)
```

1.49

Interfaz de Cassandra: driver de Python (III)

- O para queries frecuentes usar *prepared statements*

```
preparada= session.prepare("SELECT * FROM users WHERE user_id=?")
preparada.consistency_level=ConsistencyLevel.QUORUM  #opcional
users=[]
for user_id in user_ids_to_query:
  user = session.execute(preparada, [user_id])
  users.append(user)
```

1.50

Integración de Cassandra con Spark

- Se puede hacer a través del API del core Spark (basado en RDD's) o a través Spark SQL
- Spark SQL
  - Pensado para datos estructurados (ficheros json, tablas de bases de datos,…)
  - Porporciona interfaz SQL y también a través de la estructura de datos Dataset o Dataframes (la versión de Python sólo soporta Dataframes)
  - Dataframe caso particular de Dataset
    - Dataset organizado por columnas
    - Equivale a una tabla de base de datos
    - Se puede construir a partir ficheros estructurados (tipo json), tablas de bases de datos
- Es posible configurar el RDD para que use una función de particionado de datos de Cassandra (sólo si el RDD se transforma usando keyby y la clave forma parte de la clave de particionado de cassandra)

1.51

Integración de Cassandra con Spark: SparkSQL

- Carga de datos en un Dataframe:

  df = spark.read.load("Tweets.json", format="json")

  df = spark.read.json("Tweets.json")

  df = spark.read.load("Tweets.csv", format="csv", sep=":", inferSchema="true", header="true")
- Consulta de datos de un Dataframe:

  df.select("text").show()

  df.createOrReplaceTempView("mitabla")

  df=spark.sql("select text from mitabla").show()
- Conversión de un dataframe de sparkSQL a un dataframe de pandas y al revés

  pff=df.toPandas()

  df=spark.createDataframe(pf)

1.52

Integración de Cassandra con Spark: connector

- Necesitamos el código capaz de interactuar con la base de datos: datastax tiene una implementación disponible
- Necesitamos un cluster de Cassandra en ejecución
- Al crear la configuración hay que decir dónde está el paquete del conector y los datos del cluster de Cassandra (ip y puerto) y al cargar los datos indicar que el formato es el de Cassandra

```
miconf=SparkConf().set("spark.cassandra.connection.host",”XXX.XX.XX.X").set("spark.cassandra.connection.port",”X")
sc=SparkContext("local[*]", "pyspark-cass",conf=miconf)
spark = SparkSession(sc)
df=spark.read.format("org.apache.spark.sql.cassandra").options(table=”nombre tabla",keyspace=”nombre keyspace").load()
```

- A partir de ahí se pueden usar las sentencias de SparkSQL para acceder a la base de datos
- Al ejecutar la aplicación hay que decir donde está el paquete con el conector

```
spark-submit –jars path_to_jar path_to_appl.py
```

1.53

Spark MLlib

- Algoritmos de machine learning implementados sobre Spark (https://spark.apache.org/docs/latest/ml-guide.html)
  - Utilidades de estadística y álgebra lineal (min, max, media, varianza, chi-squared, correlación,…)
  - Algoritmos para el aprendizaje automático supervisado
    - Clasificación (decission tree classifier, random forest, naive bayes,…)
    - Regresiones (linear regression, ….)
  - Algoritmos de clustering, basados en distancia o en densidad (K-means, …)
  - Extracción y selección de features
    - HashingTF y IDF (TF–IDF), para obtener la relevancia de palabras en un conjunto de documentos
    - Word2vec, para obtener por ejemplo la similitud entre dos documentos)
    - ….
  - Sistemas de recomendación

1.54