EAE Business School

Hadoop: Spark

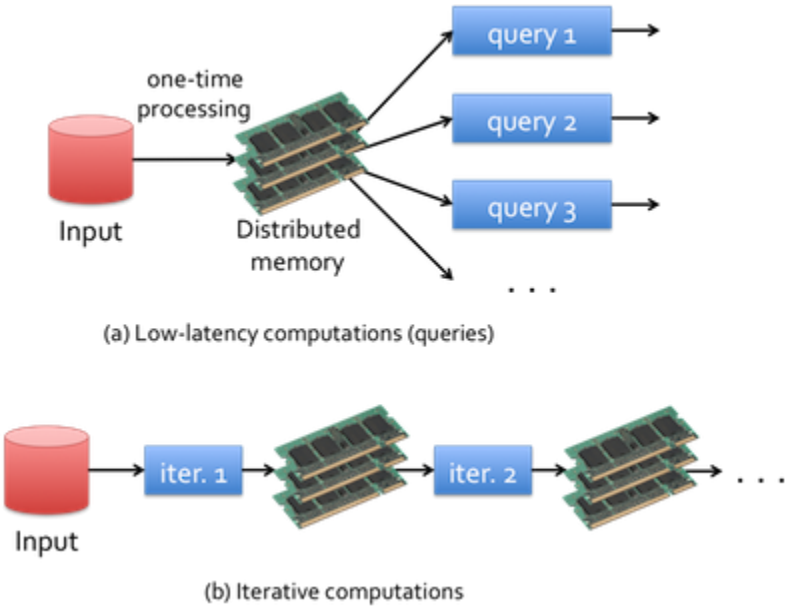
Máster en Business Intelligence e Innovación Tecnológica

Índice de contenidos

Introducción

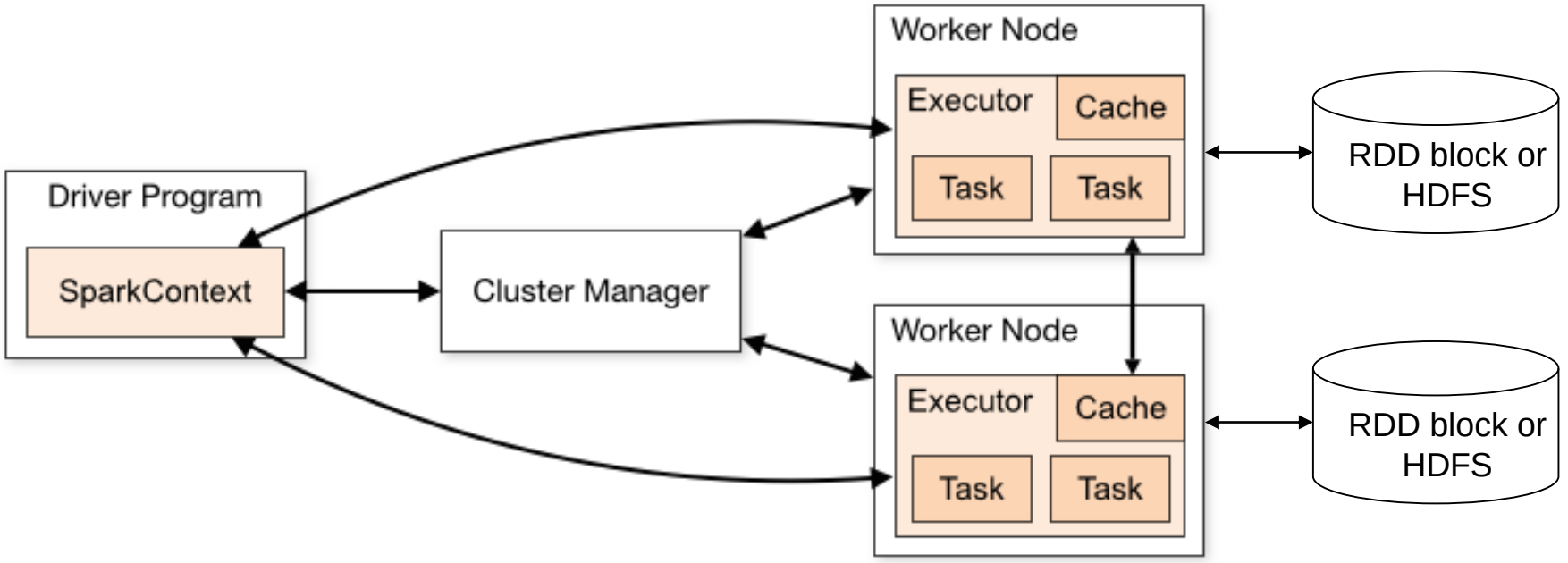
- Nace en AMPLab a partir de Mesos, un framework distribuido creado en UC Berkeley.
- En 2010 se licencia por parte de Apache.
- Hasta la fecha, las tendencias BigData venían dominadas por los paradigmas MapReduce, cuya gran dependencia de lectura en disco desembocaba en procesos de una gran latencia principalmente orientados a batch.
- Introduce una diferencia significativa: carga de datos en memoria y gran uso de “caché”, lo que le otorga una mayor velocidad (baja latencia) proporcionando una buena respuesta en peticiones real-time o streaming.
- Proporciona una API programable en varios lenguajes (Java, Python, Scala).
- Se basa en la explotación de varios sistemas distribuidos, permitiendo conectarse a distintos tipos de motores (Hadoop, NoSQL, SQL).
- Permite ejecutarse de forma distribuida o stand-alone.

Introducción



- Permite la explotación de varias consultas a partir de una memoria compartida.
- Permite la ejecución de algoritmos iterativos, más cercanos al concepto de Machine Learning y conceptualmente distintos al paradigma MapReduce.

Introducción



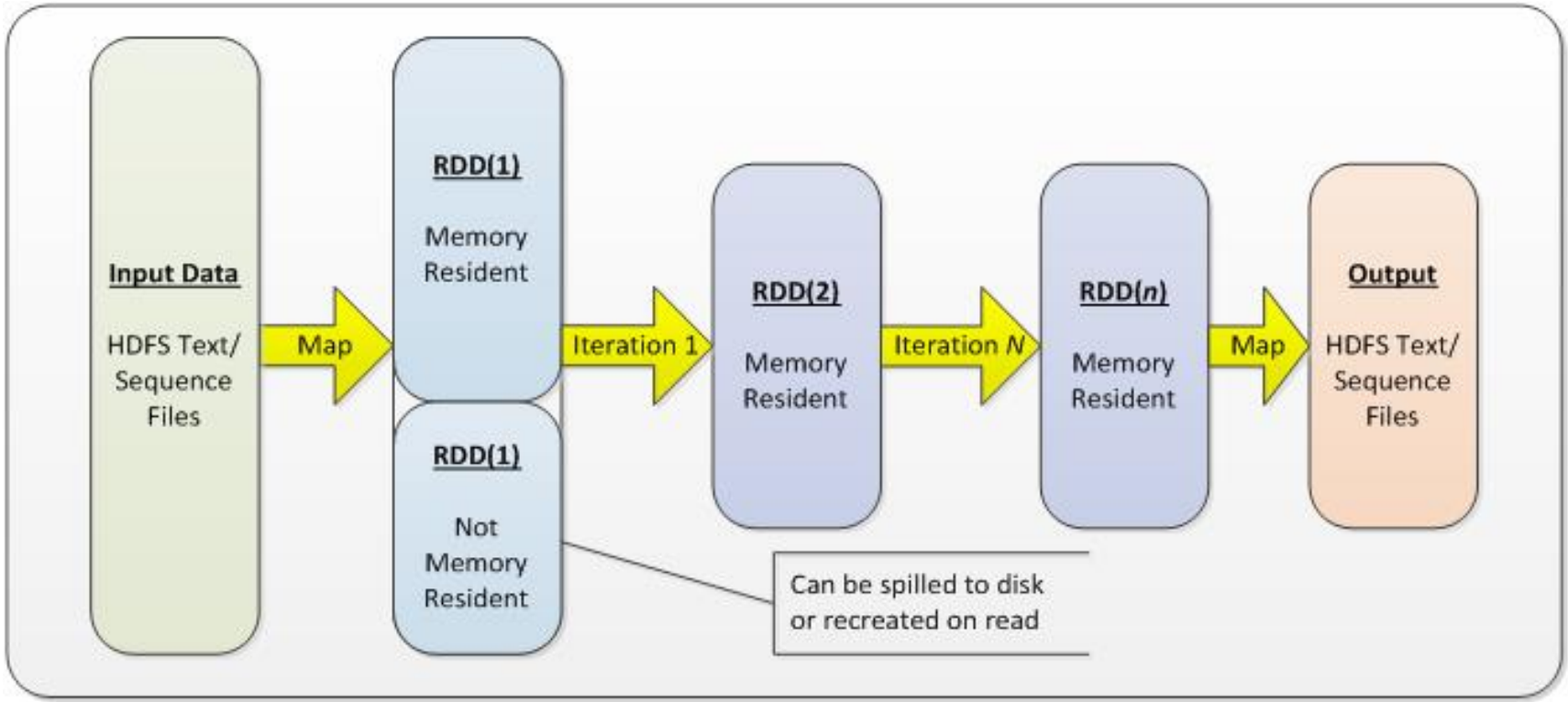
Conceptos

- RDD (Resilient Distributed Dataset)
  - Son colecciones lógicas, inmutables y particionadas de registros a lo largo del cluster.
  - Pueden ser reconstruidas si alguna partición se pierde (no necesitan ser materializadas pero si reconstruidas para mantener el almacenamiento estable a partir de su “lineage”).
  - Se crean mediante la transformación de datos utilizando para ello transformaciones (filtros, joins, Group by...).
  - Permiten cachear los datos mediante transformaciones como Reduce, Collect, count, save...
  - Los RDDs se pueden crear a partir de un archivo del sistema de ficheros de Hadoop (o de cualquier otro archivo soportado por el sistema de Hadoop pudiendo realizar transformaciones sobre los mismos.
  - Pueden persistirse en memoria, permitiendo reusar eficientemente las operaciones que se realizan en paralelo.

Conceptos

- Variables y contexto compartido
  - Se pueden utilizar variables compartidas para las operaciones que se realizan en paralelo.
  - Cuando Spark ejecuta una función en paralelo como un conjunto de tareas en diferentes nodos, este lleva una copia de cada variable utilizada en la función a cada tarea.
  - Una variable necesita ser compartida entre todas las tareas, o entre las tareas y el driver program, el programa principal que las coordina.
  - Spark soporta dos tipos de variables compartidas:
    - Broadcast variables: las cuales se usan para cachear un valor en memoria a todos los nodos.
    - Accumulators: las cuales son variables que solo admiten "añadir algo", como contadores y sumas.

Conceptos


Input Data
HDFS Text/ Sequence Files
Map
RDD(1)
Memory Resident
RDD(1)
Not Memory Resident
Iteration 1
RDD(2)
Memory Resident
Iteration N
RDD(n)
Memory Resident
Map
Output
HDFS Text/ Sequence Files
Can be spilled to disk or recreated on read

Instalación (Cloudera)

- Instalación:

```
sudo easy_install ipython==1.2.1
```

- Arranque del terminal:

```
PYSPARK_DRIVER_PYTHON=ipython pyspark
```

- Pantalla inicial:

- Chequeo de versión:

```
sc.version
```

Introducción

- PySpark es una API para el desarrollo de Spark en Python
- Permite:
  - inyectarse de forma nativa en los programas Python para ejecutarse de forma autónoma
  - ejecutarse en una interfaz (Shell) particular, dónde los RDD se van ejecutando de forma encadenada y mostrando los resultados en cada iteración.

Sintaxis

- Lectura de filesystem:

```
text_RDD = sc.textFile("file:///home/cloudera/testfile1"
```

- Lectura de HDFS:

```
text_RDD = sc.textFile("/user/cloudera/input/testfile1")
```

- Wordcount map:

```
def split_words(line): return line.split()
def create_pair(word): return (word, 1)
pairs_RDD=text_RDD.flatMap(split_words).map(create_pair)
```

- Wordcount reduce:

```
def sum_counts(a, b): return a + b
wordcounts_RDD = pairs_RDD.reduceByKey(sum_counts)
```

Sintaxis

- Resultados de un RDD:

```
wordcounts_RDD.collect()
```

- Sacar un registro de un RDD:

```
wordcounts_RDD.take(1)
```

Sintaxis

Transformaciones:
- Un RDD es inmutable, las transformaciones sobre un RDD resultan otro RDD.

- Map: aplica una función a cada elemento del RDD

```
def lower(line):
        return line.lower() lower_text_RDD =
text_RDD.map(lower)
```

- flatMap(func): ejecuta un map devolviendo un RDD de elementos unidimensionales
- filter(func): mantiene solo los elementos cuando func = true
- sample(withReplacement, fraction, seed): devuelve una fracción de datos aleatoria
- coalesce(numPartitions): une particiones para reducir su número
- groupByKey : agrupa pares (K, V) en función de la clave
- reduceByKey(func) : agrupa los pares aplicando la función *func*

Sintaxis

Acciones:
- Operaciones finales de un workflow, devuelven los resultados al driver o HDFS.

- Resultados de un RDD:

```
wordcounts_RDD.collect()
```

- Sacar un registro de un RDD:

```
wordcounts_RDD.take(1)
```

- reduce(func): realiza una agregación mediante la función func
- saveAsTextFile(filename): guarda en un directorio local o HDFS

Compartiendo información

Spark proporciona mecanismos para compartir información entre varias tareas como herramienta de aceleración

Concepto	Uso	Ejemplo
Paralelizar	Genera un dataset distribuido que puede ser operado en paralelo	data = Array(1, 2, 3, 4, 5) distData = sc.parallelize(data)

Cache	Almacena un RDD en memoria para ser consumido a posteriori.	def split_words(line):
		return line.split()
		def create_pair(word):
		return (word, 1)
		pairs_RDD=text_RDD.flatMap(split_words).map(create_pair) pairs_RDD.cache()
		def sum_counts(a, b): return a + b
		wordcounts_RDD = pairs_RDD.reduceByKey(sum_counts)
		First job: wordcounts_RDD.collect()
		Second job: pairs_RDD.take(1

Compartiendo información

Concepto	Uso	Ejemplo
Broadcast	Envía una variable a otros nodos.	config = sc.broadcast({"order":3, "filter":True}) config.value
Acumuladores	Realiza una operación de acumulación sobre una variable que puede ser consumida entre varios nodos.	accum = sc.accumulator(0) def test_accum(x): accum.add(x) sc.parallelize([1, 2, 3, 4]).foreach(test_accum) accum.value

El objeto sc representa el contexto (spark context). Con él podremos comunicarnos con otros nodos.

Ejemplo: join en spark

- Preparar datos:

```
[cloudera@quickstart join1]$ hdfs dfs -mkdir /user/cloudera/input
[cloudera@quickstart join1]$ hdfs dfs -put join*.txt /user/cloudera/input
[cloudera@quickstart join1]$ hdfs dfs -ls /user/cloudera/input
Found 3 items
-rw-r--r--   1 cloudera cloudera         37 2017-02-18 09:36 /user/cloudera/inpu
t/join1_FileA.txt
-rw-r--r--   1 cloudera cloudera        122 2017-02-18 09:36 /user/cloudera/inpu
t/join1_FileB.txt
-rw-r--r--   1 cloudera cloudera        157 2017-02-18 09:36 /user/cloudera/inpu
t/join1_output.txt
[cloudera@quickstart join1]$
```

- Arrancar Shell PySpark:

```
[cloudera@quickstart join1]$ PYSPARK_DRIVER_PYTHON=ipython pyspark
Python 2.6.6 (r266:84292, Feb 22 2013, 00:00:18)
Type "copyright", "credits" or "license" for more information.

IPython 1.2.1 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system
```

Ejemplo: join en spark

- Cargar fichero A y ver resultados

```
fileA = sc.textFile("input/join1_FileA.txt")
fileA.collect()
```

- Hacer lo propio con el B:

```
fileB = sc.textFile("input/join1_FileB.txt")
fileB.collect()
```

- Definir función de mapping para A:

```
def split_fileA(line):
    # separar la linea en una palabra y contador
    s = line.split(",")
    word = s[0]
    count = s[1]
    # convertir contador en int
    count = int(count)
    return (word, count)
```

Ejemplo: join en spark

- Testear función de mapeo A

```
In [6]: test_line = "able,991"

In [7]: split_fileA(test_line)
Out[7]: ('able', 991)
```

- Aplicar mapeo y comprobar:

```
In [8]: fileA_data = fileA.map(split_fileA)

In [9]: fileA_data.collect()

Out[9]: [(u'able', 991), (u'about', 11), (u'burger', 15), (u'actor', 22)]
```

- Crear función de mapeo para B:

```
In [10]: def split_fileB(line):
    ....:         # separa la linea de entrada en palabra, fecha y contador
    ....:         s = line.split(",")
    ....:         count_string = s[1]
    ....:         ss = s[0].split(" ")
    ....:         word = ss[1]
    ....:         date = ss[0]
    ....:         return (word, date + " " + count_string)
    ....:
```

Ejemplo: join en spark

- Testear función de mapeo B

```
In [11]: fileB_data = fileB.map(split_fileB)

In [12]: fileB_data.collect()
```

```
Out[12]:
[(u'able', u'Jan-01 5'),
 (u'about', u'Feb-02 3'),
 (u'about', u'Mar-03 8'),
 (u'able', u'Apr-04 13'),
 (u'actor', u'Feb-22 3'),
 (u'burger', u'Feb-23 5'),
 (u'burger', u'Mar-08 2'),
 (u'able', u'Dec-15 100')]
```

- Ejecutar join entre RDD:

```
In [13]: fileB_joined_fileA = fileB_data.join(fileA_data)
```

- Comprobar resultados:

```
In [12]: fileB data.collect()
Out[14]:
[(u'about', (u'Feb-02 3', 11)),
 (u'about', (u'Mar-03 8', 11)),
 (u'able', (u'Jan-01 5', 991)),
 (u'able', (u'Apr-04 13', 991)),
 (u'able', (u'Dec-15 100', 991)),
 (u'actor', (u'Feb-22 3', 22)),
 (u'burger', (u'Feb-23 5', 15)),
 (u'burger', (u'Mar-08 2', 15))]
```