빅데이터 분석 및 응용

L04: Graph Mining on MapReduce

Summer 2020

Kookmin University

Outline

- **Spark Overview**
- RDDs: Resilient Distributed Datasets
- Transformations
- Actions
- Spark Key-Value RDDs

What is Spark?

Fast and expressive cluster computing system compatible with Apache Hadoop

- **Efficient**
  - General execution graphs
  - In-memory storage
- **Usable**
  - Rich APIs in Java, Scala, Python
  - Interactive shell

Hadoop programming...

When you meet Spark

Key Concepts

Write programs in terms of **transformations** on distributed datasets

- **Resilient Distributed Datasets (RDDs)**
  - Collections of objects spread across a cluster, stored in RAM or on Disk
  - Built through parallel transformations
  - Automatically rebuilt on failure
- **Operations**
  - Transformations (e.g. map, filter, groupBy)
  - Actions (e.g. count, collect, save)

Language Support

Python

```
lines = sc.textFile(...)
lines.filter(lambda s: “ERROR” in s).count()
```

Scala

```
val lines = sc.textFile(...)
lines.filter(x => x.contains(“ERROR”)).count()
```

Java

```
JavaRDD<String> lines = sc.textFile(...);
lines.filter(new Function<String, Boolean>() {
  Boolean call(String s) {
    return s.contains(“error”);
  }
}).count();
```

Standalone Programs

- Python, Scala, & Java

Interactive Shells

- Python & Scala

Performance

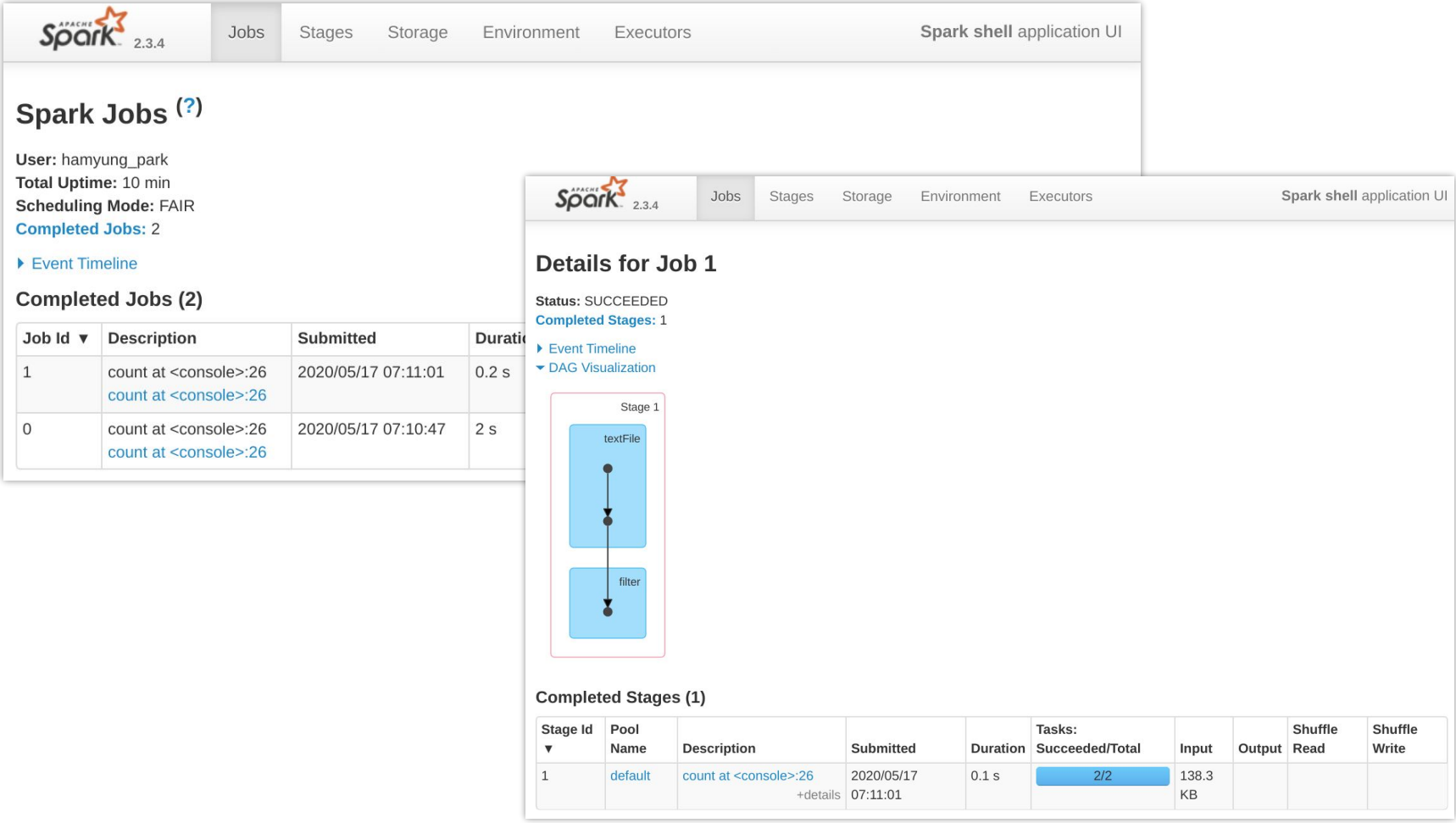
- Java & Scala are faster due to static typing
- …but Python is often fine

Interactive Shell

- The fastest way to learn Spark
- Available in Python (`pyspark`) and Scala (`spark-shell`)
- Runs as an application on an existing Spark cluster…
- OR can run locally

```
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 2.3.4
      /_/

Using Scala version 2.11.8 (OpenJDK 64-Bit Server VM, Java 1.8.0_242)
Type in expressions to have them evaluated.
Type :help for more information.

scala> val file = sc.textFile("hdfs:///user/hamyung_park/the_little_prince.txt")
file: org.apache.spark.rdd.RDD[String] = hdfs:///user/hamyung_park/the_little_prince.txt MapParti
tionsRDD[1] at textFile at <console>:24

scala> file.count()
res0: Long = 1892

scala> file.filter{line => line.contains("prince")}.count()
res1: Long = 184

scala>
```

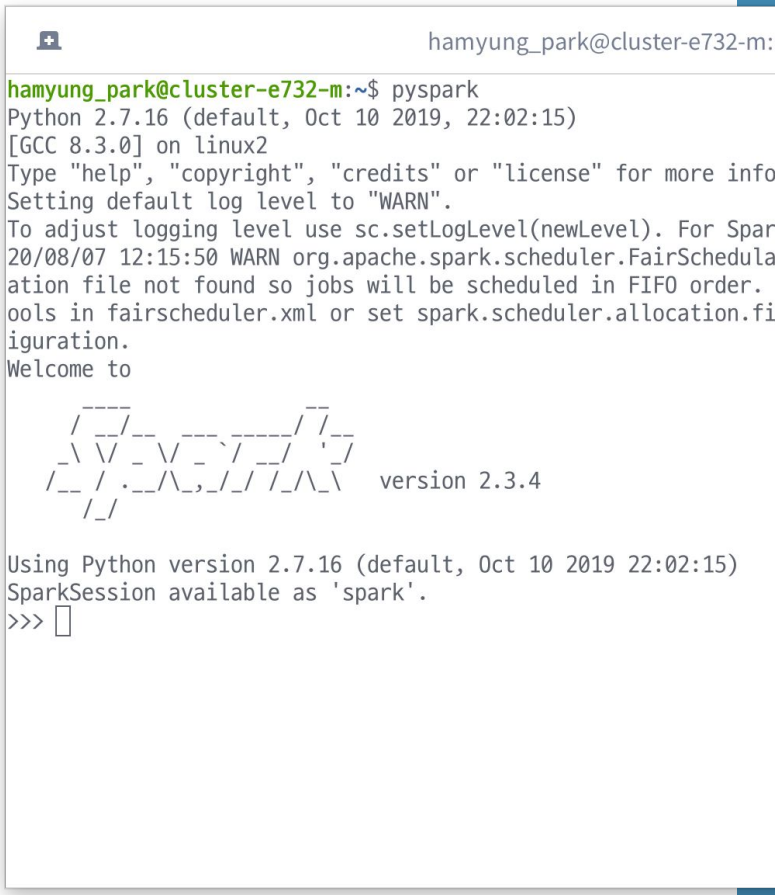
Administrative GUIs



Using The Shell

- Launching: `spark-shell`
- Modes
  - `--master <master URL>`: Specifies which master SparkContext connects to
  - Examples

```
pyspark --master local
# local, 1 worker thread
pyspark --master local[2]
# local, 2 worker threads
pyspark --master spark://host:port
# Spark standalone cluster
# (port = 7077 by default)
```



Spark Driver and Workers

- A Spark program consists of two programs:
  - A **driver program** and a **workers program**
- Worker programs run on cluster nodes or in local threads
- RDDs are distributed across workers

Spark Context

- A Spark program first creates a SparkContext object
  - Main entry point to Spark functionality
  - Tells Spark how and where to access a cluster
  - **pyspark** automatically creates the **sc** variable
- Standalone programs must use a constructor to create a new SparkContext
- Use SparkContext to create RDDs

```
conf = SparkConf().setAppName("My First Spark App")
sc = SparkContext(conf=conf)
```

Spark Essentials: Master

- The master parameter for a SparkContext determines which type and size of cluster to use.

Master Parameter	Description
`local`	run Spark locally with one worker thread (no parallelism)
`local[K]`	run Spark locally with K worker threads (ideally set to number of cores)
`spark://HOST:PORT`	connect to a Spark standalone cluster; PORT depends on config (7077 by default)
`mesos://HOST:PORT`	connect to a Mesos cluster ; PORT depends on config (5050 by default)
`yarn`	connect to a YARN cluster; cluster location to be found based on HADOOP_CONF_DIR or YARN_CONF_DIR variable.

Outline

- Spark Overview
- **RDDs: Resilient Distributed Datasets**
- Transformations
- Actions
- Spark Key-Value RDDs

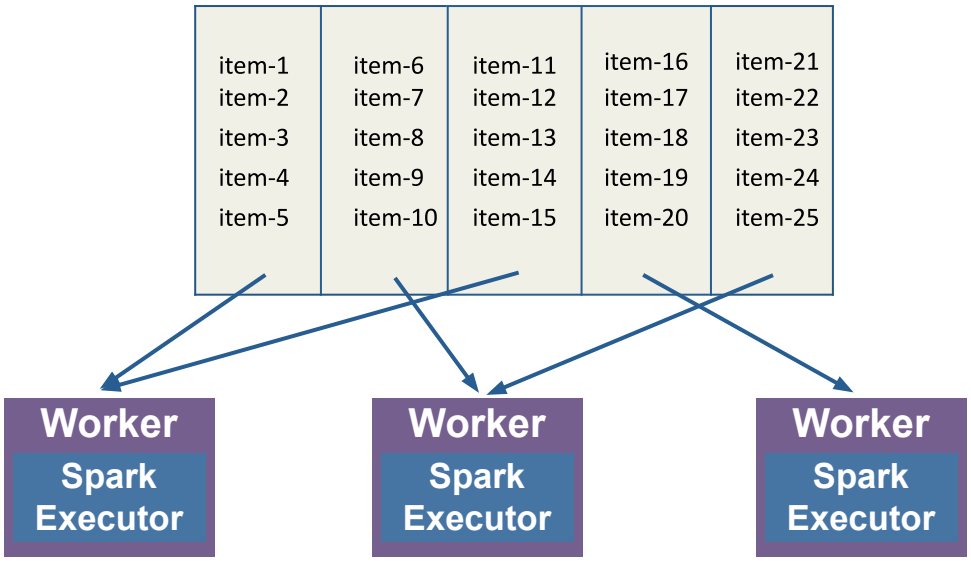
RDDs : Resilient Distributed Datasets

- The primary abstraction in Spark
  - Immutable once constructed
  - Track lineage information to efficiently recompute lost data
  - Enable operations on collection of elements in parallel
- You construct RDDs
  - by parallelizing existing Python collections (lists)
  - by transforming an existing RDDs
  - from files in HDFS or any other storage system

RDDs : Resilient Distributed Datasets

- Programmer specifies number of partitions for an RDD

RDD split into 5 partitions



(Default value used if unspecified)

more partitions → more parallelism

```
x = sc.parallelize([1,2,3,4,5], 80)
```

RDDs : Resilient Distributed Datasets

- Two types of operations:

Transformations and **Actions**

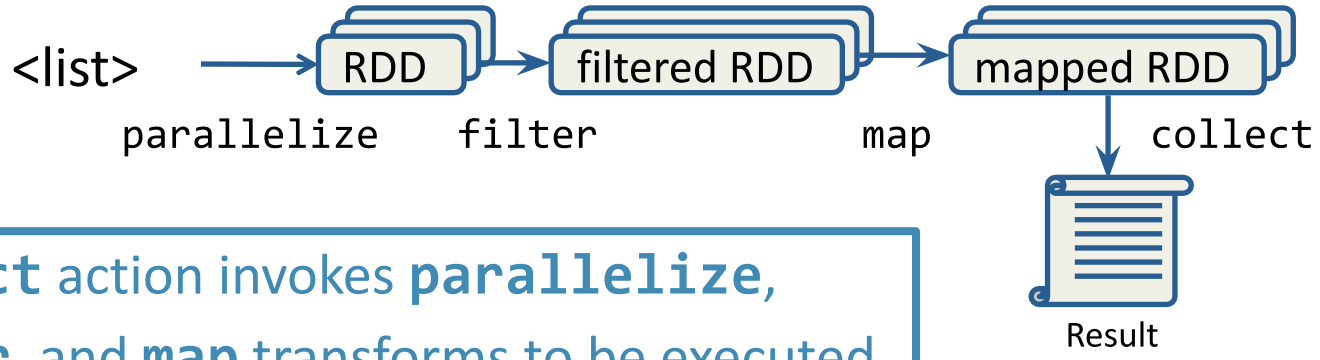
- Transformations are **lazy** (not computed immediately)
- Transformed RDD is **computed when action runs** on it
- Persist (cache) RDDs in memory or disk

Working with RDDs

- Create an RDD from a data source:

`textFile`, `parallelize`, ...

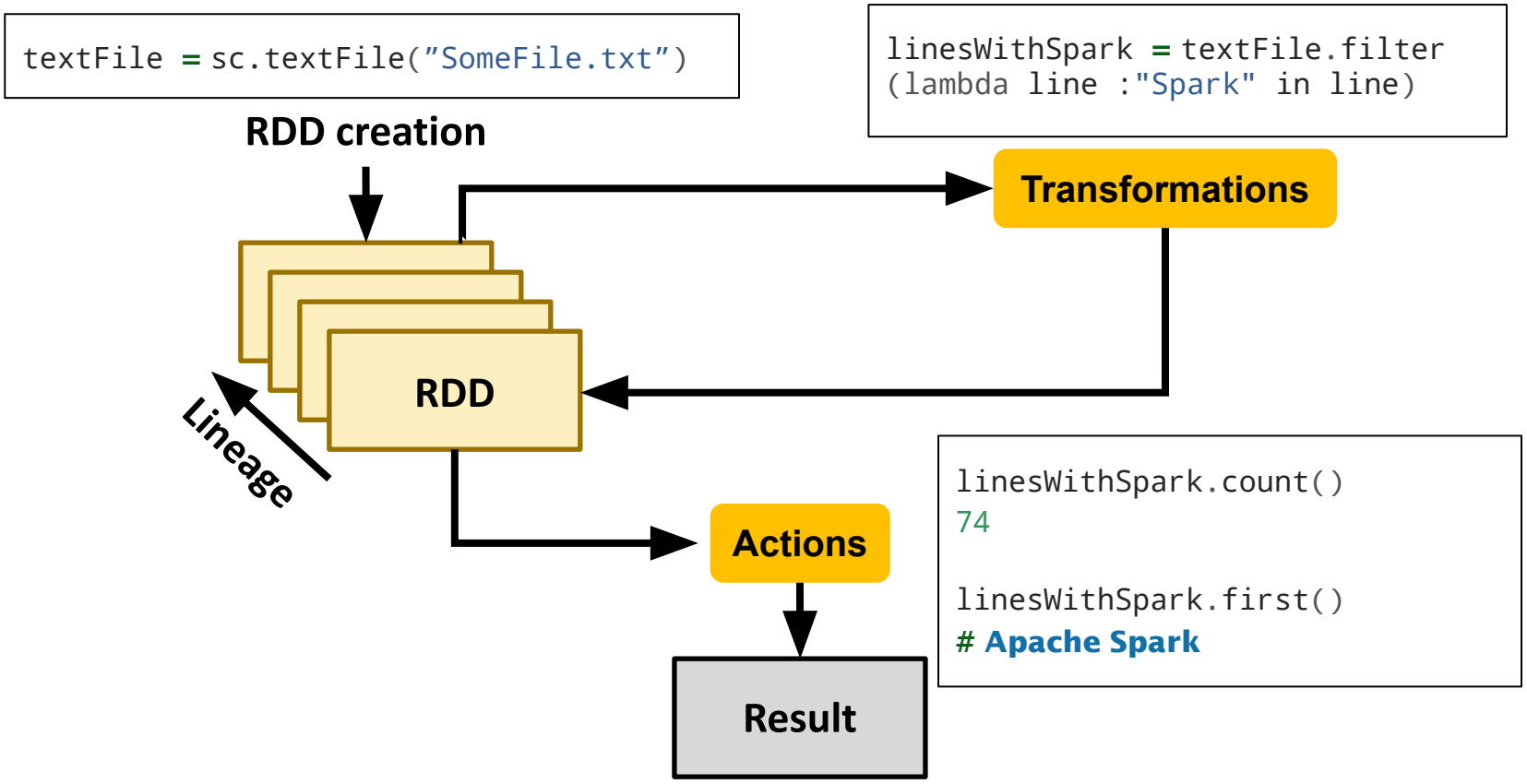
- Apply transformations to an RDD: `map`, `filter`, ⋯
- Apply actions to an RDD: `collect`, `count`, ...



`collect` action invokes **`parallelize`**, **`filter`**, and **`map`** transforms to be executed

Working with RDDs

- Another view…



Creating an RDD

```
# Turn a Scala collection into an RDD
>  sc.parallelize([1, 2, 3])

# Load text file from local FS, HDFS, or S3
>  sc.textFile("file.txt")
>  sc.textFile("directory/*.txt")
>  sc.textFile("hdfs://namenode:9000/path/file")

# Use existing Hadoop InputFormat (Java/Scala only)
>  sc.hadoopFile(keyClass, valClass, inputFmt, conf)
```

Outline

- Spark Overview
- RDDs: Resilient Distributed Datasets
- **Transformations**
- Actions
- Spark Key-Value RDDs

Spark Transformations

- Create new datasets from an existing one
- Use **lazy evaluation**: results not computed right away – instead Spark remembers set of transformations applied to base dataset
  - Spark optimizes the required calculations
  - Spark recovers from failures and slow workers
- Think of this as a recipe for creating result

Spark Transformations

- Some examples

Transformation	Description
`map(`*`func`*`)`	return a new distributed dataset formed by passing each element of the source through a function *func*
`filter(`*`func`*`)`	return a new dataset formed by selecting those elements of the source on which *func* returns true
`distinct([`*`numTasks`*`]))`	return a new dataset that contains the distinct elements of the source dataset
`flatMap(`*`func`*`)`	similar to map, but each input item can be mapped to 0 or more output items (so *func* should return a Seq rather than a single item)

Spark Transformations

- Using Lambda expressions

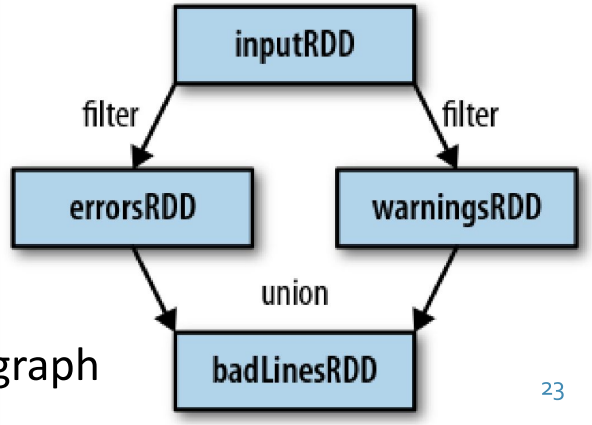
```
>>> val rdd = sc.parallelize([1, 2, 3])
>>> rdd.map(lambda x: [x, x+5])
RDD: RDD(1, 2, 3) → RDD([1, 6], [2, 7], [3, 8])

>>> rdd.flatMap(lambda x: [x, x+5])
RDD: RDD(1, 2, 3) → RDD(1, 6, 2, 7, 3, 8)
```

Function literals (green) are automatically passed to workers

Spark Transformations

- Handling a log file

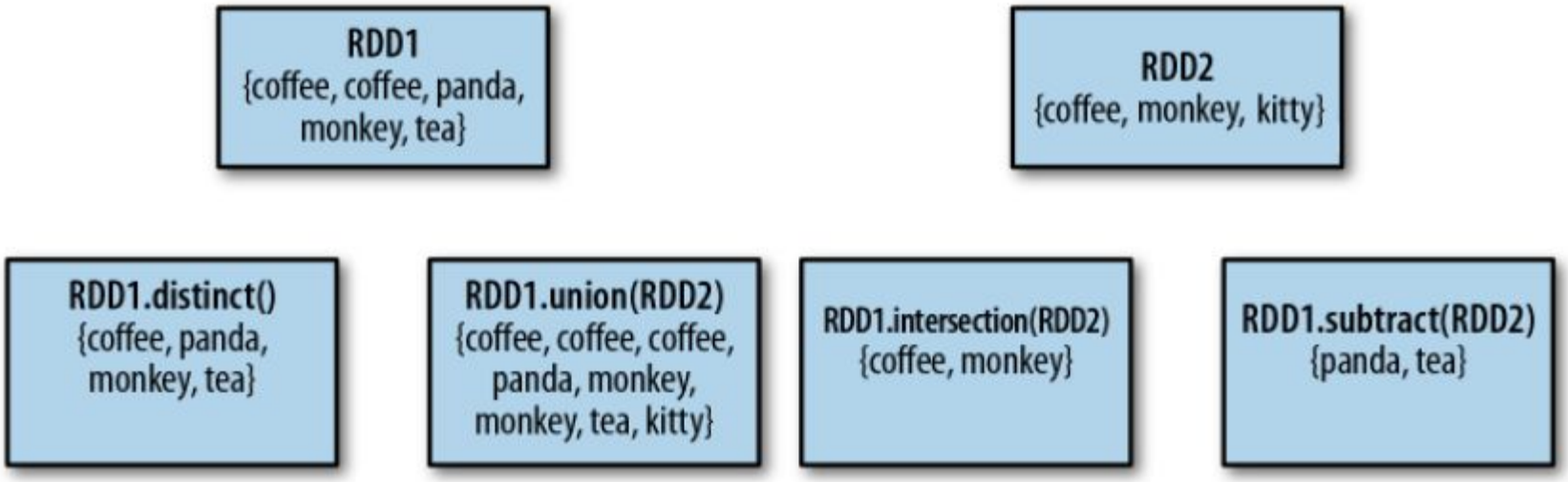
```
> inputRDD = sc.textFile("log.txt")

# RDD for "error" lines
> errorsRDD = inputRDD.filter(lambda x: "error" in x)

# RDD for "warning" lines
> warningsRDD = inputRDD.filter(lambda x: "warning" in x)

# Union them into badlinesRDD
> badlinesRDD = errorsRDD.union(warningsRDD)
```



RDD lineage graph

Spark Transformations

- Four set operations on RDD1 and RDD2
  - `distinct, union, intersection, subtract`



Passing Functions to Spark

- Spark's API relies heavily on passing functions in the driver program to run on the cluster.
- There are three recommended ways to do this.
  - Lambda expression
  - Top-level functions in a module
  - Local `def`'s inside the function calling into Spark

```
word = rdd.filter{lambda s: "error" in s}

def containsError(s):
   s.contains("error")

val word = rdd.filter(containsError)
```

```
counter = 0
rdd = sc.parallelize(data)
// Wrong: Don't do this!!
rdd.foreach{x => counter += x}
```

Source: http://spark.apache.org/docs/latest/rdd-programming-guide.html#passing-functions-to-spark

Outline

- Spark Overview
- RDDs: Resilient Distributed Datasets
- Transformations
- **Actions**
- Spark Key-Value RDDs

Spark Actions

- Cause Spark to execute recipe to transform source
- Mechanism for getting results out of Spark

Spark Actions

- Some example actions

Action	Description
`reduce(`*`func`*`)`	aggregate dataset's elements using function *func*. *func* takes two arguments and returns one, and is commutative and associative so that it can be computed correctly in parallel
`take(`*`n`*`)`	return an array with the first *n* elements
`collect()`	return all the elements as an array WARNING: make sure will fit in driver program
`takeOrdered(`*`n, key=func`*`)`	return *n* elements ordered in ascending order or as specified by the optional key function
`saveAsTextFile(`*`path`*`)`	save this RDD as a text file to *path*, using string representations of elements

Spark Actions

- Getting data out of RDDs

```
>>> rdd = sc.parallelize([5, 3, 1, 2])
>>> rdd.reduce(lambda a, b: a * b)
30
>>> rdd.take(2)
[5, 3]
>>> rdd.collect()
[5, 3, 1, 2]
>>> rdd.takeOrdered(3, lambda x: -x)
[5, 3, 2]
```

두개의 결과를 하나로 만들어 준다

Spark Actions

- More examples

```
> nums = sc.parallelize([1, 2, 3])

# Retrieve RDD contents as a local collection
> nums.collect() # => [1, 2, 3]

# Return first k elements
> nums.take(2)   # => [1, 2]

# Count number of elements
> nums.count()   # => 3

# Merge elements with an associative function
> nums.reduce(lambda x, y: x + y)   # => 6

# Write elements to a text file
> nums.saveAsTextFile("hdfs://file.txt")
```

Outline

- Spark Overview
- RDDs: Resilient Distributed Datasets
- Transformations
- Actions
- **Spark Key-Value RDDs**

Spark Key-Value RDDs

- Similar to MapReduce, Spark supports Key-Value pairs
- Spark provides special operations on RDDs containing key/value pairs.
  - These RDDs are called pair RDDs
- Each element of a pair RDD is a pair tuple.

```
>>> rdd = sc.parallelize([(1, 2), (3, 4)])
RDD: [(1, 2), (3, 4)]
```

Working with Key-Value Pairs

- Spark's "distributed reduce" transformations operate on RDDs of key-value pairs

Python:

```
pair = (a, b)
       pair[0] # => a
       pair[1] # => b
```

Scala:

```
val pair = (a, b)
       pair._1 // => a
       pair._2 // => b
```

Java:

```
Tuple2 pair = new Tuple2(a, b);
       pair._1 // => a
       pair._2 // => b
```

Key–Value Transformations

- Some Key–Value transformations

Key-Value Transformation	Description
`reduceByKey(`*`func`*`)`	return a new distributed dataset of (K,V) pairs where the values for each key are aggregated using the given reduce function *func*, which must be of type (V,V) → V
`sortByKey()`	return a new dataset (K,V) pairs sorted by keys in ascending order
`groupByKey()`	return a new dataset of (K, Iterable<V>) pairs

Key-Value Transformations

- Examples

```
>>> rdd =  sc.parallelize([(1,2), (3,4), (3,6)])
>>> rdd.reduceByKey(lambda a, b: a + b)
RDD: [(1,2), (3,4), (3,6)] →[(1,2), (3,10)]

>>> rdd2 = sc.parallelize([(1,'a'), (2,'c'), (1,'b')])
>>> rdd2.sortByKey()
RDD: [(1,'a'), (2,'c'), (1,'b')] → [(1,'a'), (1,'b'), (2,'c')]
```

Key-Value Transformations

- groupByKey example

```
>>> rdd2 = sc.parallelize([(1,'a'), (2,'c'), (1,'b')])
>>> rdd2.groupByKey()
RDD: [(1,'a'), (1,'b'), (2,'c')] → [(1,['a','b']), (2,['c'])]
```

Be careful using groupByKey() as it can cause a lot of data movement across the network and create large Iterables at workers

Key-Value Transformations

- More key-value operations

```
> pets = sc.parallelize([("cat", 1), (“dog", 1), ("cat", 2)])

# Reduction example
> pets.reduceByKey(lambda x, y: x + y) # => {(cat, 3), (dog, 1)}

# groupByKey
> pets.groupByKey()    # => {(cat, [1, 2]), (dog, [1])}

# Count number of elements
> pets.sortByKey()     # => {(cat, 1), (cat, 2), (dog, 1)}
```

  - `reduceByKey` also automatically implements combiners on the map side

Key–Value Transformations

- Setting the Level of Parallelism
  - All the pair RDD operations take an optional second parameter for the number of tasks

```
> words.reduceByKey(lambda x, y: x + y, 5)
> words.groupByKey(5)
> visits.join(pageViews, 5)
```

Questions?