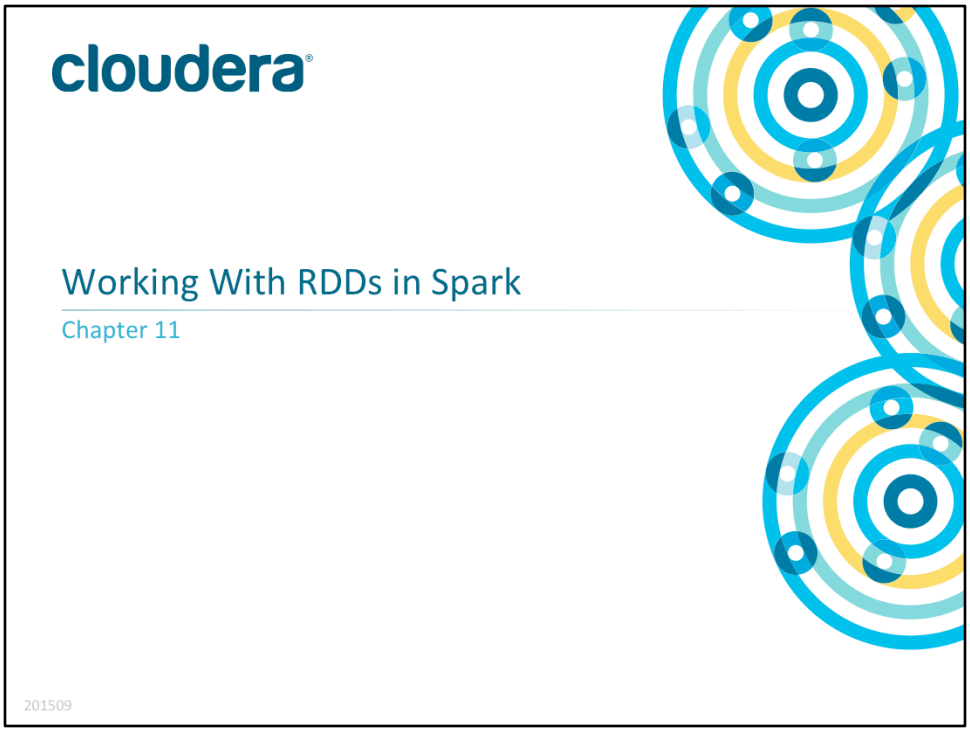

cloudera
Working With RDDs in Spark
Chapter 11
201509

Course Chapters

1	Introduction	Course Introduction
2 3	Introduction to Hadoop and the Hadoop Ecosystem Hadoop Architecture and HDFS	Introduction to Hadoop
4 5 6 7 8	Importing Relational Data with Apache Sqoop Introduction to Impala and Hive Modeling and Managing Data with Impala and Hive Data Formats Data File Partitioning	Importing and Modeling Structured Data
9	Capturing Data with Apache Flume	Ingesting Streaming Data
10 **11** 12 13 14 15 16 17	Spark Basics **Working with RDDs in Spark** Aggregating Data with Pair RDDs Writing and Deploying Spark Applications Parallel Processing in Spark Spark RDD Persistence Common Patterns in Spark Data Processing Spark SQL and DataFrames	**Distributed Data Processing with Spark**
18	Conclusion	Course Conclusion

Working With RDDs

In this chapter you will learn

- **How RDDs are created from files or data in memory**
- **How to handle file formats with multi-line records**
- **How to use some additional operations on RDDs**

Chapter Topics

Working With RDDs in Spark | **Distributed Data Processing with Spark**

- **Creating RDDs**
- Other General RDD Operations
- Conclusion
- Homework: Process Data Files with Spark

RDDs

- **RDDs can hold any type of element**
  - Primitive types: integers, characters, booleans, etc.
  - Sequence types: strings, lists, arrays, tuples, dicts, etc. (including nested data types)
  - Scala/Java Objects (if serializable)
  - Mixed types
- **Some types of RDDs have additional functionality**
  - Pair RDDs
    - RDDs consisting of Key-Value pairs
  - Double RDDs
    - RDDs consisting of numeric data

Full list of special Pair RDD functions:
http://spark.apache.org/docs/latest/api/core/index.html#org.apache.spark.rdd.PairRDDFunctions

A note on keys: although keys can be any serializable type, some of the functions (For instance, sortByKey) won't work on keys whose type doesn't have an implicit ordering.

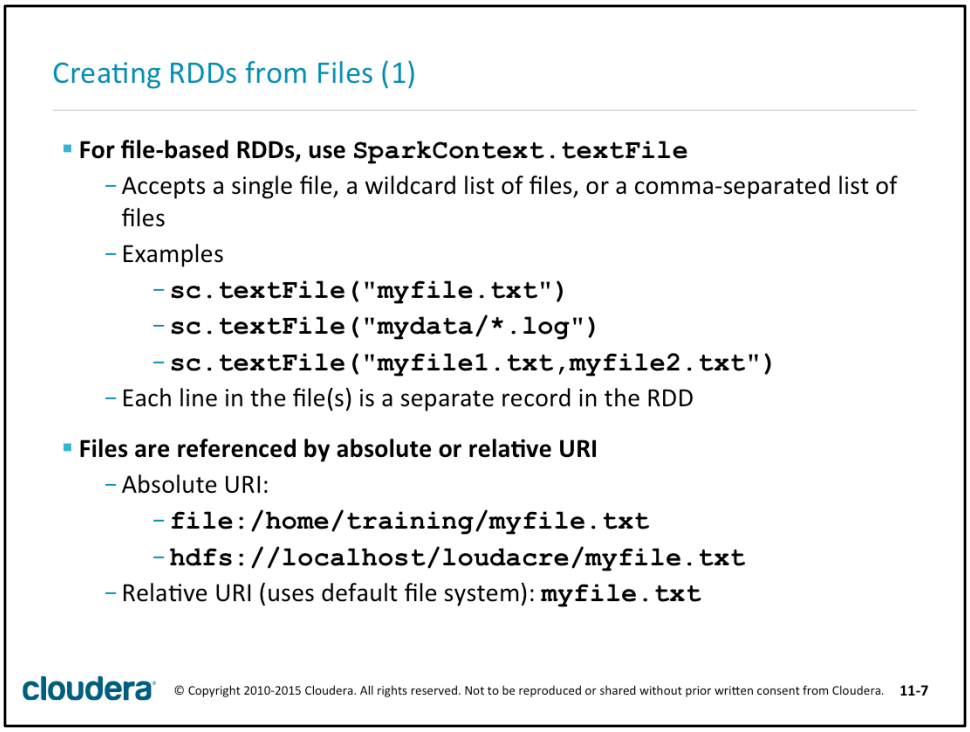
Creating RDDs From Collections

- **You can create RDDs from collections instead of files**
  - **`sc.parallelize(`*`collection`*`)`**

```
> myData = ["Alice","Carlos","Frank","Barbara"]
> myRdd = sc.parallelize(myData)
> myRdd.take(2)
['Alice', 'Carlos']
```

- **Useful when**
  - Testing
  - Generating data programmatically
  - Integrating

Another use case might be performing CPU intensive calculations on a small dataset produced by some other program. (If it were a very large dataset, it wouldn't fit in memory in the driver to begin with.)



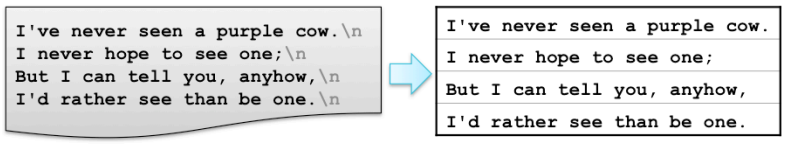
A note on file URIs: if you don't specify a file system (e.g. "file:/") Spark will assume the URI is relative to the default file system. In vanilla Spark, that is the local file system on the machine the code is running on. In CDH Spark, it defaults to HDFS. We are using the CDH configuration in class, so you must specify file: explicitly (and specify the full path) to reference any files on the local file system.

This may initially seem to be unreasonable, because we haven't yet talked about running on a cluster. That will be explained in the next chapter but you may wish to point out to students that eventually their code will be running on a worker node on a cluster, and relative file URIs will resolve to the local machine it is running on, NOT the driver. So we've set the default to be more work, but ultimately less confusing when running on a cluster, which is the presumed setting for Spark in the real world.

A note on file input formats:
Students with experience in MapReduce may ask about this, but it is beyond the scope of the class:
Spark by default uses Hadoop's file access libraries. The textFile method uses Hadoop's TextInputFormat (which in turn uses LineRecordReader), which reads a text file and makes each line of the file a separate record. This is most common, but Spark in theory does support the use of other Hadoop InputFormats, in which case you would use hadoopFile (instead of textFile) to read the file, and saveAsHadoopFile

Creating RDDs from Files (2)

- **`textFile` maps each line in a file to a separate RDD element**



- **`textFile` only works with line-delimited text files**
- **What about other formats?**

There are two approaches to loading multiple files:
For large files, or files of varying or unknown size: sc.textFile("mydir/*") (or "mydir/file1,mydir/file2") which is what we've been using in the labs

When loading multiple files, there will be at least one partition for each file (more if the files are larger than the default partition size.)

This is important for two reasons:
1 – sometimes you need to be able to operation on all the data from a single file. We will see this later when we have XML files that need to be parsed. We will use getPartition to process a single partition as a whole rather than one row at a time.
2 – if you are loading many small files, you will end up with many small partitions, which can be inefficient. We will learn more in the performance chapter about repartitioning.

For small files – that is, files where the content is reasonable to fit into a single record:
sc.wholeTextFiles("mydir"): this creates a paired RDD, where the key is the name of the file, and the contents are the value.

Input and Output Formats (1)

- **Spark uses Hadoop `InputFormat` and `OutputFormat` Java classes**
  - Some examples from core Hadoop
    - **`TextInputFormat`** / **`TextOutputFormat`** – newline delimited text files
    - **`SequenceInputFormat`** / **`SequenceOutputFormat`**
    - **`FixedLengthInputFormat`**
  - Many implementations available in additional libraries
    - e.g. **`AvroInputFormat`** / **`AvroOutputFormat`** in the Avro library

https://hadoop.apache.org/docs/current/api/org/apache/hadoop/mapreduce/lib/input/FixedLengthInputFormat.html

FixedLengthInputFormat is an input format used to read input files which contain fixed length records. The content of a record need not be text. It can be arbitrary binary data. Users must configure the record length property by calling: FixedLengthInputFormat.setRecordLength(conf, recordLength); or conf.setInt(FixedLengthInputFormat.FIXED_RECORD_LENGTH, recordLength);

Input and Output Formats (2)

- **Specify any input format using `sc.hadoopFile`**
  - or **`newAPIhadoopFile`** for New API classes
- **Specify any output format using `rdd.saveAsHadoopFile`**
  - or **`saveAsNewAPIhadoopFile`** for New API classes
- **`textFile` and `saveAsTextFile` are convenience functions**
  - **`textFile`** just calls **`hadoopFile`** specifying **`TextInputFormat`**
  - **`saveAsTextFile`** calls **`saveAsHadoopFile`** specifying **`TextOutputFormat`**

There's also saveAsHadoopDataset, which is beyond the scope of this course. That saves to a non-file-based output.

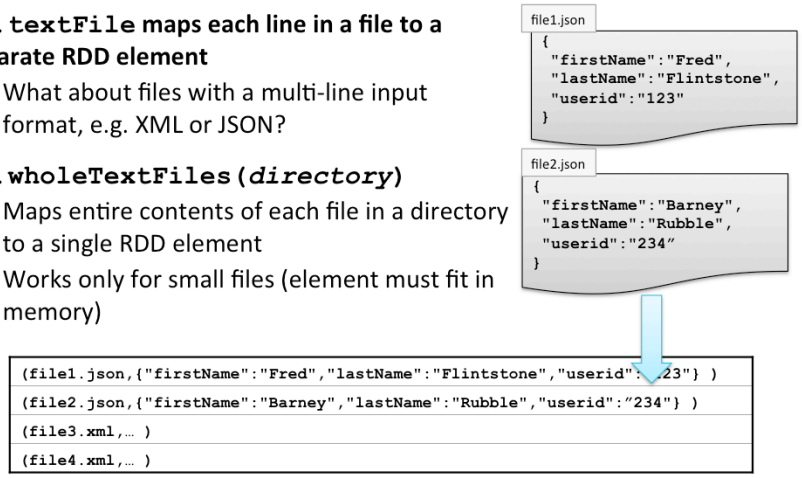
From the Scala API Docs:
def newAPIHadoopFile[K, V, F <: InputFormat[K, V]](path: String, fClass: Class[F], kClass: Class[K], vClass: Class[V], conf: Configuration = hadoopConfiguration): RDD[(K, V)]

Get an RDD for a given Hadoop file with an arbitrary new API InputFormat and extra configuration options to pass to the input format.

Note: Because Hadoop's RecordReader class re-uses the same Writable object for each record, directly caching the returned RDD or directly passing it to an aggregation or shuffle operation will create many references to the same object. If you plan to directly cache, sort, or aggregate Hadoop writable objects, you should first copy them using a map function.

Whole File-Based RDDs (1)

- **`sc.textFile` maps each line in a file to a separate RDD element**
  - What about files with a multi-line input format, e.g. XML or JSON?
- **`sc.wholeTextFiles(`*`directory`*`)`**
  - Maps entire contents of each file in a directory to a single RDD element
  - Works only for small files (element must fit in memory)



Whole File-Based RDDs (2)

```
> import json
> myrdd1 = sc.wholeTextFiles(mydir)
> myrdd2 = myrdd1
  .map(lambda (fname,s): json.loads(s))
> for record in myrdd2.take(2):
>     print record["firstName"]
```

Output:

```
Fred
Barney
```

```
> import scala.util.parsing.json.JSON
> val myrdd1 = sc.wholeTextFiles(mydir)
> val myrdd2 = myrdd1
  .map(pair => JSON.parseFull(pair._2).get.
          asInstanceOf[Map[String,String]])
> for (record <- myrdd2.take(2))
    println(record.getOrElse("firstName",null))
```


This example does the following:
1. uses wholeTextFiles to load the dataset on the previous slide (rdd1)
2. uses the language's JSON library to map the contents of each file to a dictionary (in python) or map (in Scala) of values from the file
3. prints out the firstName value from each Json file (for the first two files/elements)

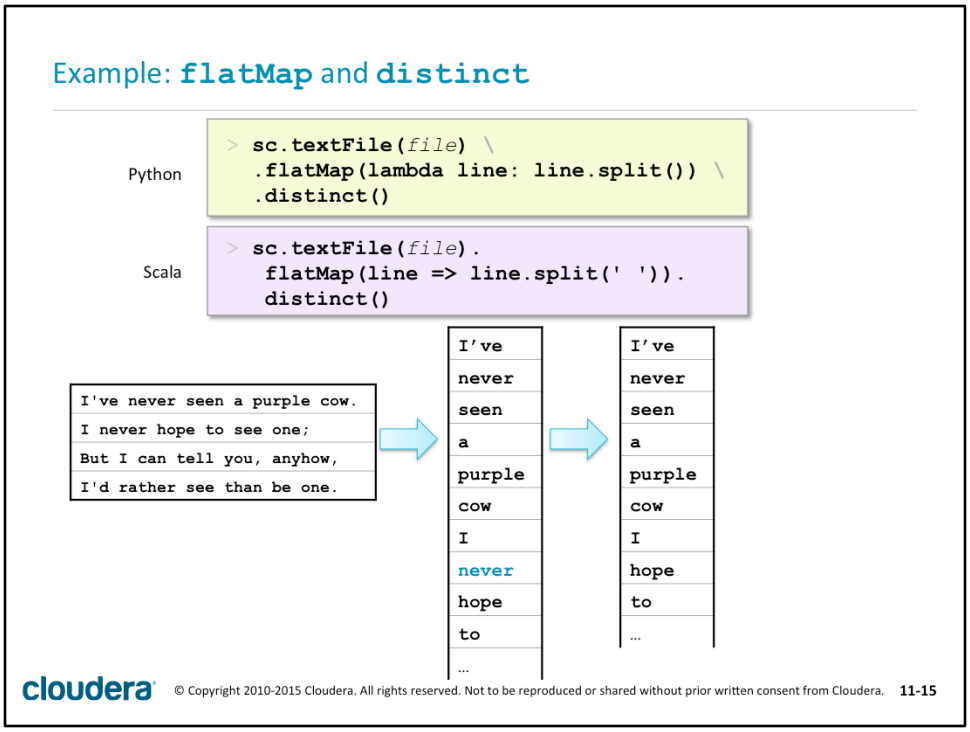
Chapter Topics

Working With RDDs in Spark | **Distributed Data Processing with Spark**

- Creating RDDs
- **Other General RDD Operations**
- Conclusion
- Homework: Process Data Files with Spark

Some Other General RDD Operations

- **Single-RDD Transformations**
  - **`flatMap`** – maps one element in the base RDD to multiple elements
  - **`distinct`** – filter out duplicates
  - **`sortBy`** – use provided function to sort
- **Multi-RDD Transformations**
  - **`intersection`** – create a new RDD with all elements in both original RDDs
  - **`union`** – add all elements of two RDDs into a single new RDD
  - **`zip`** – pair each element of the first RDD with the corresponding element of the second



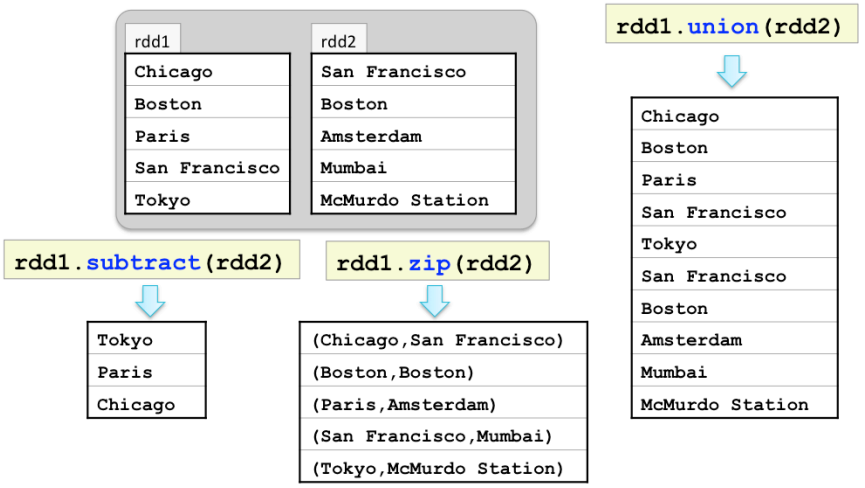
This code example takes a text file and results in a list of all unique words in the file.

Step 1 uses flatMap. Spend some time here, this is important and confusing to students, and will be used later. The important point here is that the function passed to flatMap must return a collection (e.g an array, iterator, etc.) Each item in the returned collection is mapped to a single element in the result RDD.

Step 2 filters out duplicates.

Note: the output here is for the Python example. It's slightly different for Scala because of the way the split function works in the two languages. The default split() function splits into words, and considers an apostrophe to be part of the word, so "I've" parses as one word. In Scala, we are specifying the regex for splitting on any non-word (non-alphanumeric) character, so "I've" parses into "I" and "ve". No need to call this out unless someone asks, as it is not relevant to the example.

Examples: Multi-RDD Transformations



Zip requires that the RDDs have the same number of elements, otherwise an error will result when the transformation is executed. The other operations work regardless.

Note that subtract (and likewise intersection, not shown) do not preserve order. Later in the course you can mention that this is because data is shuffled.

Some Other General RDD Operations

- **Other RDD operations**
  - **`first`** – return the first element of the RDD
  - **`foreach`** – apply a function to each element in an RDD
  - **`top(n)`** – return the largest *n* elements using natural ordering
- **Sampling operations**
  - **`sample`** – create a new RDD with a sampling of elements
  - **`takeSample`** – return an array of sampled elements
- **Double RDD operations**
  - Statistical functions, e.g., **`mean`**, **`sum`**, **`variance`**, **`stdev`**

first is like take(1) but it returns a single element instead of an array containing a single element.

"top" uses the implicit ordering of whatever the element type is: numerical comparison for numbers, alphanumeric ordering for strings, and so on. Pairs are ordered by first value, then second value.

first and top are actions.
foreach is different than other operations, because it neither returns a new RDD (like a transformation) nor returns a value (like an action). This is usually used for side effects…display, output, or accumulators.

Chapter Topics

Working With RDDs in Spark | **Distributed Data Processing with Spark**

- Creating RDDs
- Other General RDD Operations
- **Conclusion**
- Homework: Process Data Files with Spark

Essential Points

- RDDs can be created from files, parallelized data in memory, or other RDDs
- `sc.textFile` reads newline delimited text, one line per RDD record
- `sc.wholeTextFile` reads entire files into single RDD records
- Generic RDDs can consist of any type of data
- Generic RDDs provide a wide range of transformation operations

Chapter Topics

Working With RDDs in Spark | **Distributed Data Processing with Spark**

- Creating RDDs
- Other General RDD Operations
- Conclusion
- **Homework: Process Data Files with Spark**

Homework: Process Data Files with Spark

- **In this homework assignment you will**
  - Process a set of XML files using **`wholeTextFiles`**
  - Reformat a dataset to standardize format (bonus)
- **Please refer to the Homework description**

