Spark Working with RDDs

Prof. Hyuk-Yoon Kwon

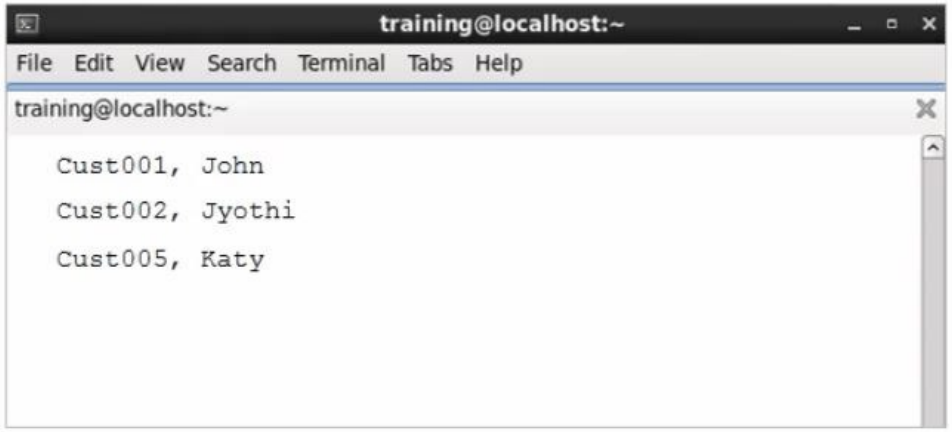
Working with RDDs

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

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



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']
```

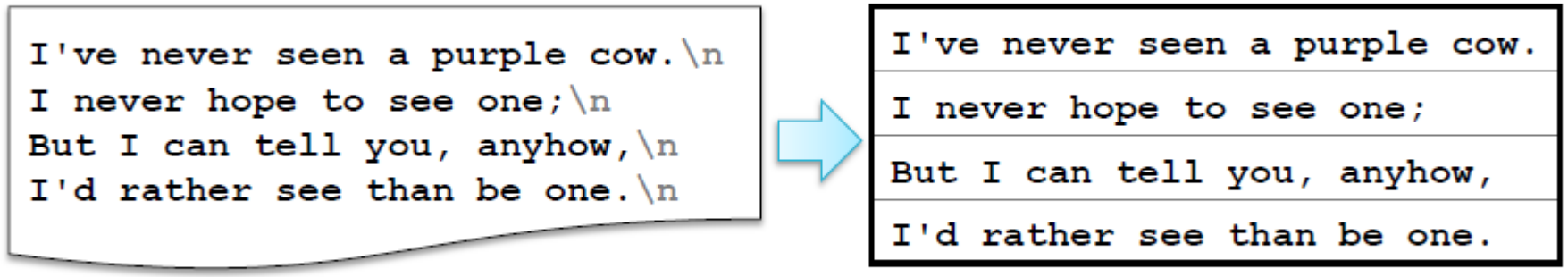
- Useful when
  - Testing
  - Generating data programmatically
  - Integrating

Creating RDDs from Files (1)

- **For file-based RDDs, use `SparkContext.textFile`**
  - Accepts a single file, a wildcard list of files, or a comma-separated list of files
  - Examples
    - `sc.textFile("myfile.txt")`
    - `sc.textFile("mydata/*.log")`
    - `sc.textFile("myfile1.txt,myfile2.txt")`
  - Each line in the file(s) is a separate record in the RDD
- **Files are referenced by absolute or relative URI**
  - Absolute URI:
    - `file:/home/training/myfile.txt`
    - `hdfs://localhost/loudacre/myfile.txt`
  - Relative URI (uses default file system): `myfile.txt`

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

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

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`**

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)

file1.json

```
{
 "firstName":"Fred",
 "lastName":"Flintstone",
 "userid":"123"
}
```

file2.json

```
{
 "firstName":"Barney",
 "lastName":"Rubble",
 "userid":"234"
}
```

(file1.json,{"firstName":"Fred","lastName":"Flintstone","userid":"123"})
(file2.json,{"firstName":"Barney","lastName":"Rubble","userid":"234"})
(file3.xml,…)
(file4.xml,…)

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"]
```

```
> 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))
```

Output:

```
Fred
Barney
```

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

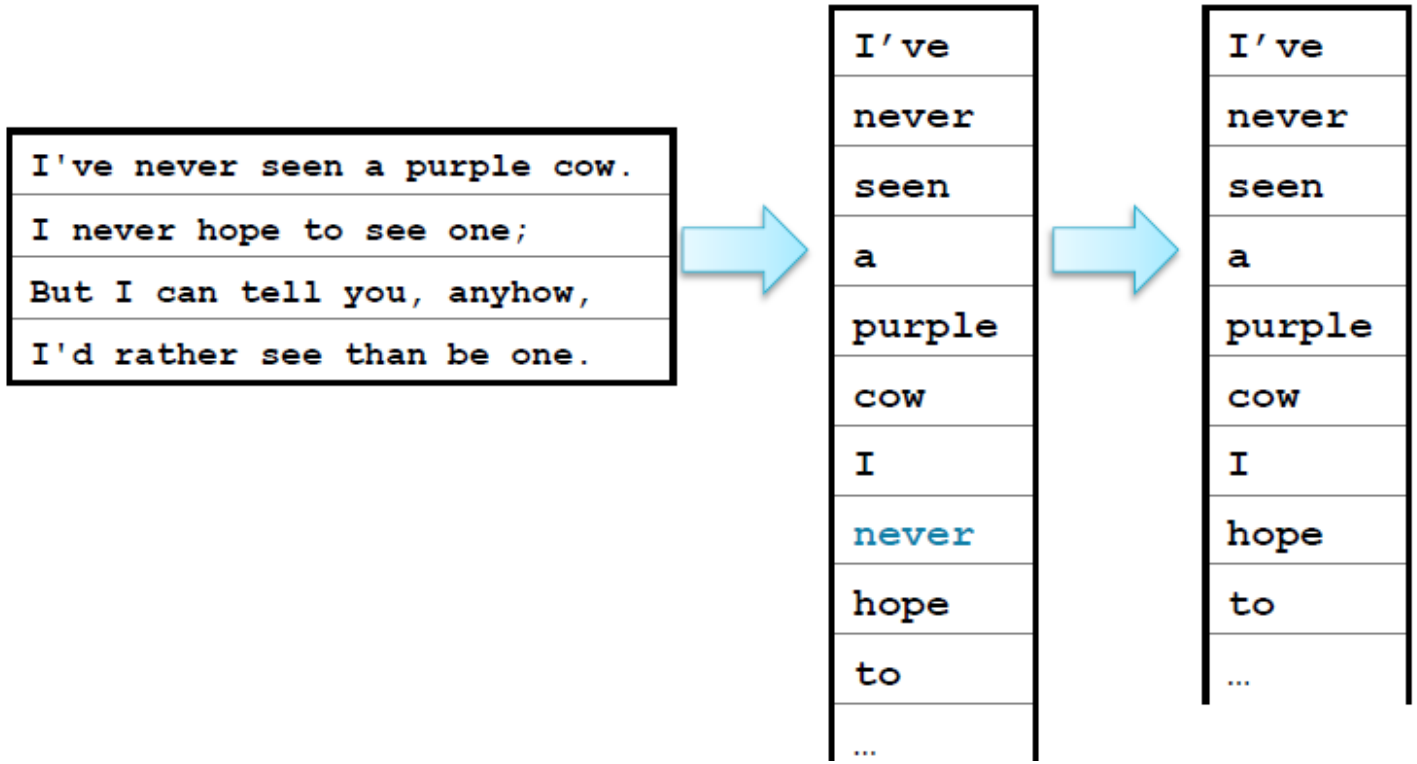
Example: flatMap and distinct

Python

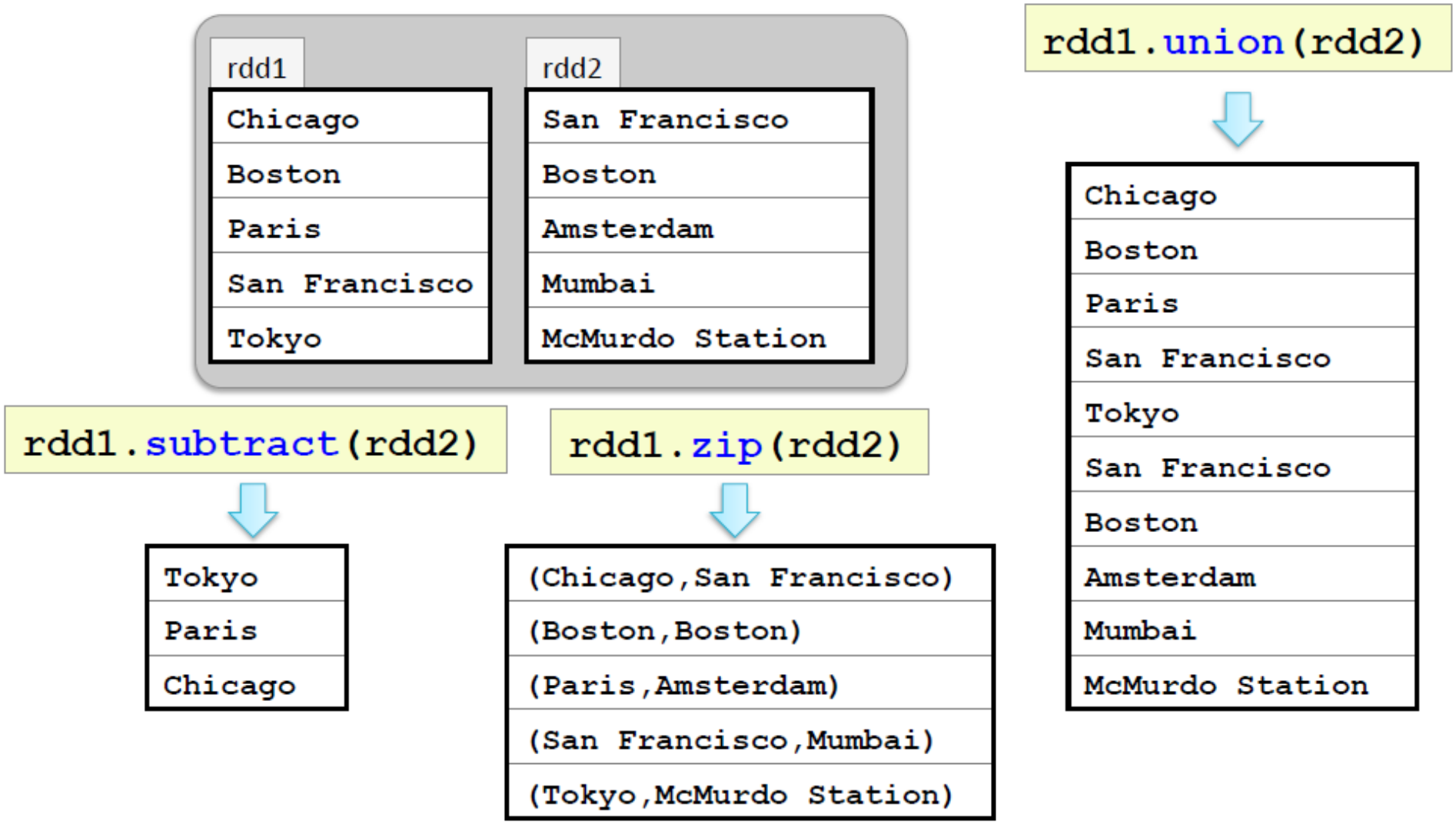
```
> sc.textFile(file) \
  .flatMap(lambda line: line.split()) \
  .distinct()
```

Scala

```
> sc.textFile(file).
   flatMap(line => line.split(' ')).
   distinct()
```



Examples: Multi-RDD Transformations



Practice: Process Data Files with Spark (1)

Files and locations

- Exercise directory: $DEV1/exercises/spark-etl
- Data files
  - $DEV1DATA/activations/*

Goal

- Parse a set of activation records in XML format to extract the account numbers and model names

Data

- Review the data in **$DEV1DATA/activations**. Each XML file contains data for all the devices activated by customers during a specific month
- Copy this data to **/loudacre** in HDFS

Task

- Your code should process a set of activation XML files and extract the account number and device model for each activation, and save the list to a file formatted as **account_number:model**

```
1234:iFruit 1
987:Sorrento F00L
4566:iFruit 1
…
```

1. Start with the **ActivationModels.pyspark** script uploaded in the e-class. Note that for convenience you have been provided with functions to parse the XML, as that is not the focus of this practice.
2. Use **wholeTextFiles** to create and RDD from the activations dataset. The resulting RDD will consist of tuples, in which the first value is the name of the file, and the second value is the contents of the file (XML) as a sting
3. Each XML file can contain many activation records; use **flatMap** to map the contents of each file to a collection of XML records by calling the provided **getactivations** function. **getactivations** takes an XML string, parses it, and returns a collection of XML records; **flatMap** maps each record to a separate RDD element.
4. Map each activation record to a string in the format **account-number:model**. Use the provided **getaccount** and **getmodel** functions to find the values from the activation record
5. Save the formatted strings to a text file in the directory **/loudacre/account-models.**

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`**

Practice: Process Data Files with Spark (2)

Files and locations

- Exercise directory: $DEV1/exercises/spark-etl
- Data files
  - $DEV1DATA/devicestatus.txt

Goal

- Process data in order to get it into a standardized format for later processing

Data

- Review the contents of the file $DEV1DATA/devicestatus.txt. This file contains data collected from mobile devices on Loudacre's network, including device ID, current status, location, and so on. Because Loudacre previously acquired other mobile provider's networks, the data from different subnetworks has a different format. Note that the records in this file have different field delimiters: some use commas, some use pipes (|) and so on.

The task

1. Load the dataset
2. Determine which delimiter to use (hint: the character at position 20 is the first use of the delimiter)
3. Filter out any records which do not parse correctly (hint: each record should have exactly 14 values)
4. Extract the date (first field), model (second field), device ID(third field), and latitude and longitude (13^{th} and 14^{th} fields respectively)
5. The second field contains the device manufacturer and model name (e.g., Ronin S2.) Split this field by spaces to separate the manufacturer from the model (e.g., manufacturer Ronin, model S2).
6. Save the extracted data to comma delimited text files in the **/loudacre/devicestat_etl** directory on HDFS
7. Confirm that the data in the file(s) was saved correctly

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