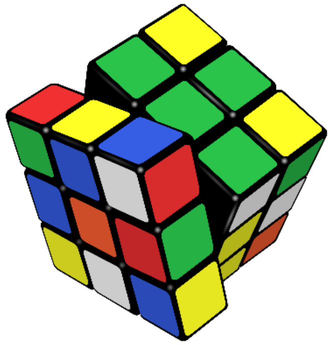
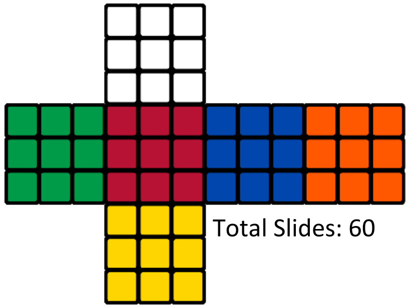


Spark - Resilient Distributed Datasets

Suria

(suria@nus.edu.sg)

NUS-ISS

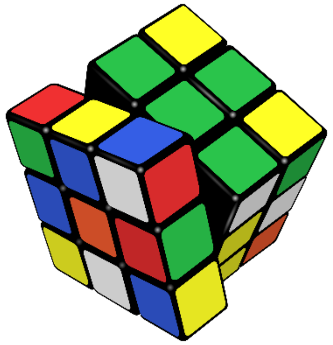


Learning Objectives

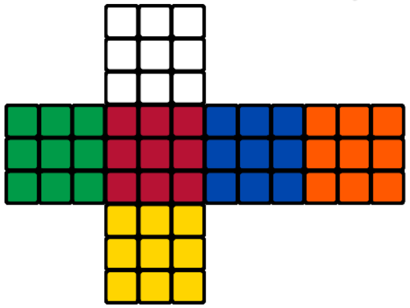
- Learn the key Concepts of Resilient Distributed Datasets (RDDs) and How to create RDDs
- Understand and write effective programs using RDD and the Spark functional programming

Agenda

- Spark RDD Persistence
- Spark RDD Actions
- Spark RDD Transformations
- Multi RDD Operations
- Summary



Big Data Engineering For Analytics



Spark RDD

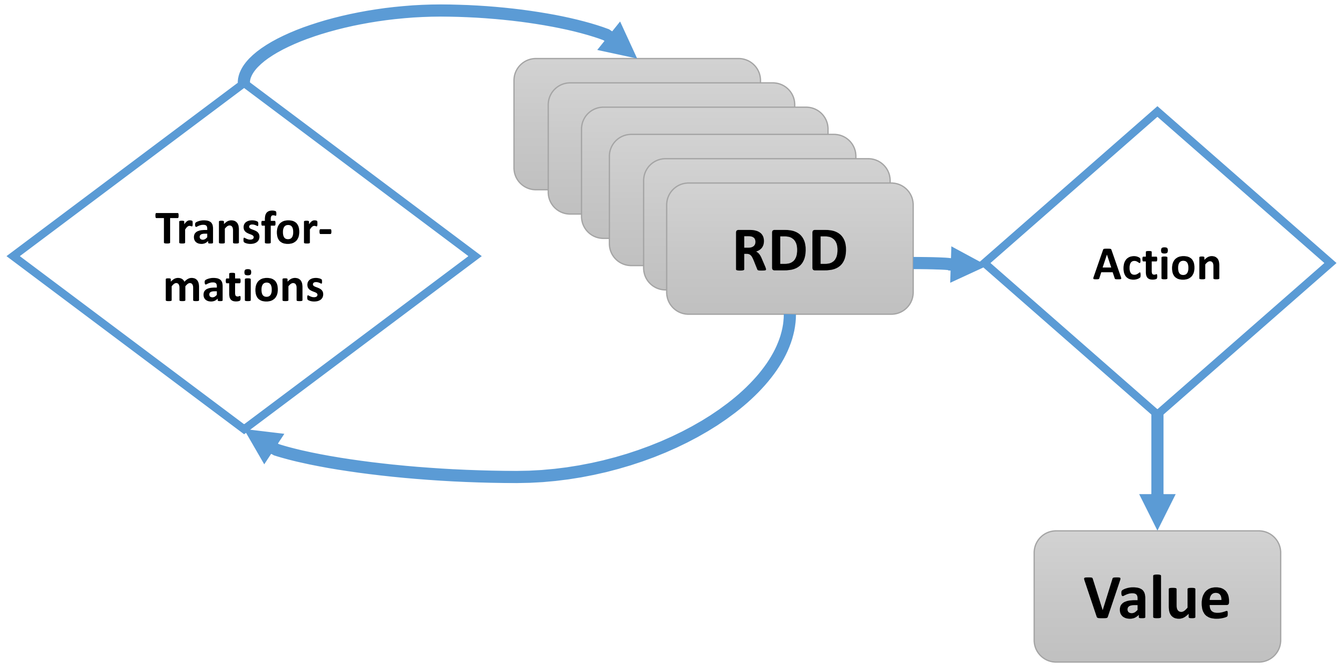
- *If you torture the data long enough, it will confess.*
  - *~Ronald Coase*

Resilient Distributed Datasets

- Spark offers a computing framework to handle iterative algorithms and interactive data mining tools.
  - ➢Keeps data in memory to improve performance
- One Key Abstraction: **Resilient Distributed Datasets (RDDs)**
  - ➢Distributed memory abstraction that lets programmers perform in-memory computations on large clusters in a fault-tolerant manner
  - ➢provide a restricted form of shared memory, based on coarse-grained transformations rather than fine-grained updates to shared state.
- RDD Two Operations:
  - ➢**Transformations**: Lazy Evaluated
  - ➢**Actions**: Are final and trigger computation
- RDD's are ***Lineage***

RDD Operations

- Lazy Evaluation



Spark Operations

Transformations (define a new RDD)	map filter sample groupByKey reduceByKey sortByKey	flatMap union join cogroup cross mapValues
Actions (return a result to driver program)	collect reduce count save lookupKey	

What is a Lineage?

- RDDs can only be created through deterministic operations on either:
  - ➢(1) data in stable storage or
  - ➢(2) other RDDs.
- Essential Property: an RDD has enough information about how it was derived from other datasets (its lineage) to compute its partitions from data in stable storage.
- Directed Acyclic Graph (DAG) of transformations
  - ➢Each time a transformation is applied to an **immutable** RDD
  - ➢A new RDD gets created with new lineage information
- Anytime you know, what transformations to apply on a single “partition” of data

Resilient Distributed Dataset Characteristics

- RDDs are read-only, **partitioned collection of records**.
  - Resilient – if data in memory is lost, it can be recreated
  - Distributed – processed across the cluster
  - Dataset – collection of rows and columns - data can come from a file or from any other source
  - RDDs are the fundamental unit of data in Spark
  - Most Spark programming consists of performing operations on RDDs
- RDDs allow the developers to use a **particular dataset from memory** and also makes a pipeline process possible.
- Data Scientists can run through an RDD multiple times through their querying process and **take advantage of their in-memory representation** and not wait for disk IO.

Create RDD

- Ways to create an RDD
  - ➢Parallelizing existing collection
  - ➢From a data in an external storage system (Local location, HDFS, Hbase)
  - ➢From another RDD
  - ➢From a DataFrame or DataSet

Parallelized collection

- In the initial stage when we learn Spark, RDDs are generally created by parallelized collection i.e. by taking an existing collection in the program and passing it to SparkContext's parallelize() method.

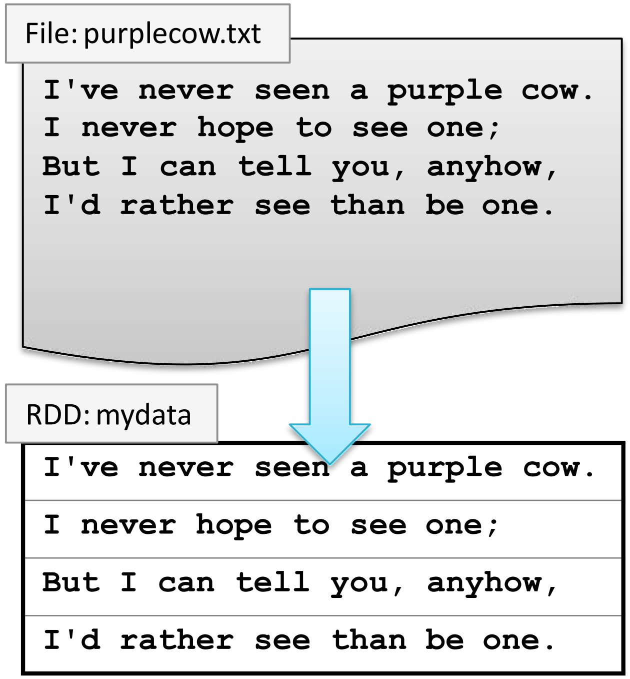
```
>  val mydata = sc.parallelize(Array(1,2,3,4,5))
>  mydata.collect()
> val rdd1 = sc.parallelize(Array(“mon”,”tue”,”wed”,”thu”,”fri”,”sat”,”sun”))
> rdd1.collect()
```

Example: A File-based RDD

```
> val mydata = sc.textFile("purplecow.txt")
…
15/01/29 06:20:37 INFO storage.MemoryStore:
   Block broadcast_0 stored as values to
   memory (estimated size 151.4 KB, free 296.8
   MB)

> mydata.count()

…
15/01/29 06:27:37 INFO spark.SparkContext: Job
   finished: take at <stdin>:1, took
   0.160482078 s
4
```


File: purplecow.txt

```
I've never seen a purple cow.
I never hope to see one;
But I can tell you, anyhow,
I'd rather see than be one.
```

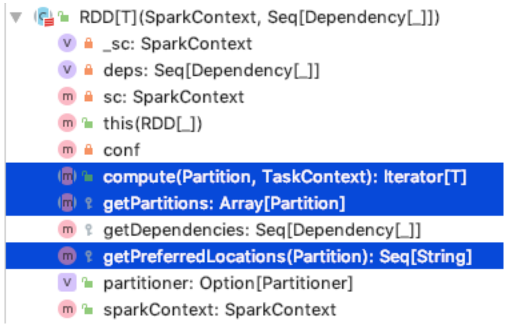
RDD: mydata

I've never seen a purple cow.
I never hope to see one;
But I can tell you, anyhow,
I'd rather see than be one.

Rich Set of Operations

An RDD is represented as an abstraction and is defined by the following **FIVE pieces of information**:

- A set of partitions, which are the chunks that make up the entire dataset
- A set of dependencies on parent RDDs
- A function for computing all the rows in the data set
- Metadata about the partitioning scheme (optional)
- Where the data lives on the cluster (optional); if the data lives on HDFS, then it would be where the block locations are located



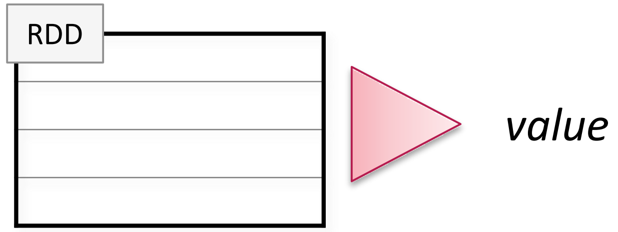
```
RDD[T](SparkContext, Seq[Dependency[_]])
    _sc: SparkContext
    deps: Seq[Dependency[_]]
    sc: SparkContext
    this(RDD[_])
    conf
    compute(Partition, TaskContext): Iterator[T]
    getPartitions: Array[Partition]
    getDependencies: Seq[Dependency[_]]
    getPreferredLocations(Partition): Seq[String]
    partitioner: Option[Partitioner]
    sparkContext: SparkContext
```

Big Data Engineering For Analytics

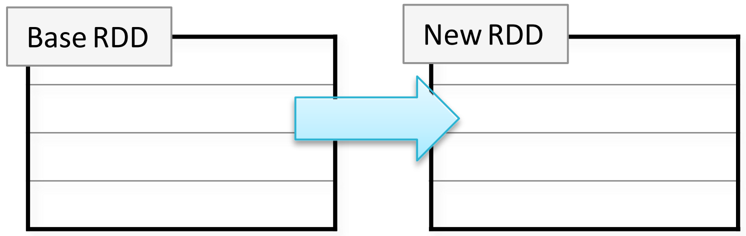
RDD Operations

Two types of RDD operations

Actions – return values



Transformations – define a new RDD based on the current one(s)



THINK: Which type of operation is **`count()`**?

RDD Persistence

- One of the most important capabilities in Spark is *persisting* (or *caching*) a dataset in memory across operations.
- When you persist an RDD, each node stores any partitions of it that it computes in memory and reuses them in other actions on that dataset (or datasets derived from it).
- This allows future actions to be much faster (often by more than 10x). Caching is a key tool for iterative algorithms and fast interactive use.
- If data is too big to be cached, it will split to disk and memory
- Persistence methods:
  - `persist()`
  - `cache()`
- Removing data. Spark automatically monitors cache usage on each node and drops out old data partitions in a least-recently-used (LRU) fashion. Use the `unpersist()` to manually remove data.

RDD Persistence (Storage Levels)

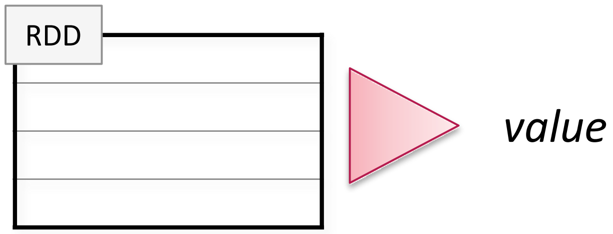
Storage Level	Meaning
MEMORY_ONLY	Store RDD as deserialized Java objects in the JVM. If the RDD does not fit in memory, some partitions will not be cached and will be recomputed on the fly each time they're needed. This is the default level.
MEMORY_AND_DISK	Store RDD as deserialized Java objects in the JVM. If the RDD does not fit in memory, store the partitions that don't fit on disk, and read them from there when they're needed.
MEMORY_ONLY_SER (Java and Scala)	Store RDD as *serialized* Java objects (one byte array per partition). This is generally more space-efficient than deserialized objects, especially when using a fast serializer, but more CPU-intensive to read.
MEMORY_AND_DISK_SER (Java and Scala)	Similar to MEMORY_ONLY_SER, but spill partitions that don't fit in memory to disk instead of recomputing them on the fly each time they're needed.
DISK_ONLY	Store the RDD partitions only on disk.
MEMORY_ONLY_2, MEMORY_AND_DISK_2, etc.	Same as the levels above, but replicate each partition on two cluster nodes.

https://spark.apache.org/docs/latest/rdd-programming-guide.html#rdd-persistence

RDD Operation: Actions

Some common actions

- **`count()`** – return the number of elements
- **`take(`*`n`*`)`** – return an array of the first *n* elements
- **`collect()`** – return an array of all elements
- **`saveAsTextFile(`*`file`*`)`** – save to text file(s)



```
> mydata =
  sc.textFile("purplecow.txt")

> mydata.count()
4

> for line in mydata.take(2):
    print line
I've never seen a purple cow.
I never hope to see one;
```

```
> val mydata =
  sc.textFile("purplecow.txt")

> mydata.count()
4

> for (line <- mydata.take(2))
    println(line)
I've never seen a purple cow.
I never hope to see one;
```

Basic actions on an RDD – (1)

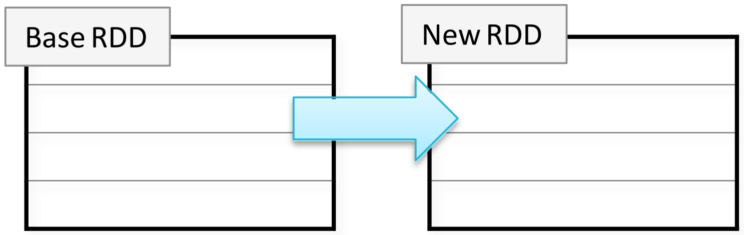
Function name	Purpose	Example	Result
collect()	Return all elements from the RDD.	`rdd.collect()`	`{1, 2, 3, 3}`
count()	Number of elements in the RDD.	`rdd.count()`	`4`
countByValue()	Number of times each element occurs in the RDD.	`rdd.countByValue()`	`{(1, 1), (2, 1), (3, 2)}`
take(num)	Return numelements from the RDD.	`rdd.take(2)`	`{1, 2}`
top(num)	Return the topnum elements the RDD.	`rdd.top(2)`	`{3, 3}`
takeOrdered(num)(ordering)	Return numelements based on provided ordering.	`rdd.takeOrdered(2)(myOrdering)`	`{3, 3}`

Basic actions on an RDD – (2).

Function name	Purpose	Example	Result
takeSample(withReplacement, num, [seed])	Return numelements at random.	`rdd.takeSample(false, 1)`	**Nondeterministic**
reduce(func)	Combine the elements of the RDD together in parallel (e.g., sum).	`rdd.reduce((x, y) => x + y)`	9
fold(zero)(func)	Same as reduce() but with the provided zero value.	`rdd.fold(0)((x, y) => x + y)`	9
aggregate(zeroValue)(seqOp, combOp)	Similar to reduce() but used to return a different type.	`rdd.aggregate((0, 0)) ((x, y) => (x._1 + y, x._2 + 1), (x, y) => (x._1 + y._1, x._2 + y._2))`	(9, 4)
foreach(func)	Apply the provided function to each element of the RDD.	`rdd.foreach(func)`	Nothing

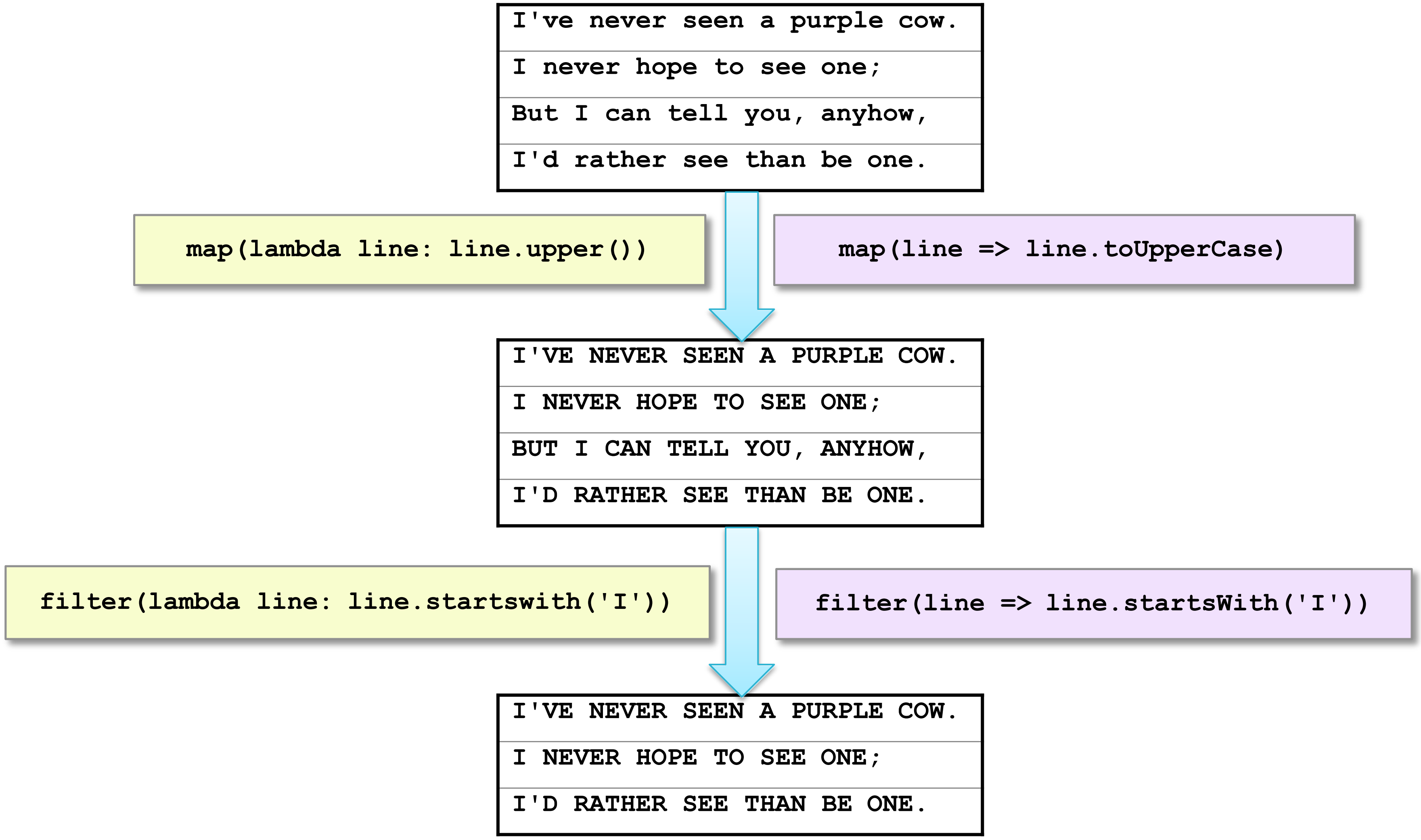
RDD Operation: Transformations

- Transformations create a new RDD from an existing one
  - ➢RDDs are immutable
  - ➢Data in an RDD is never changed



- Transform in sequence to modify the data as needed
- Some common transformations
  - ➢**`map(function)`** – creates a new RDD by performing a function on each record in the base RDD
  - ➢**`filter(function)`** – creates a new RDD by including or excluding each record in the base RDD according to a boolean function

Example map & `filter` Transformation



```
I've never seen a purple cow.
I never hope to see one;
But I can tell you, anyhow,
I'd rather see than be one.
```

`map(lambda line: line.upper())`

`map(line => line.toUpperCase)`

```
I'VE NEVER SEEN A PURPLE COW.
I NEVER HOPE TO SEE ONE;
BUT I CAN TELL YOU, ANYHOW,
I'D RATHER SEE THAN BE ONE.
```

`filter(lambda line: line.startswith('I'))`

`filter(line => line.startsWith('I'))`

```
I'VE NEVER SEEN A PURPLE COW.
I NEVER HOPE TO SEE ONE;
I'D RATHER SEE THAN BE ONE.
```

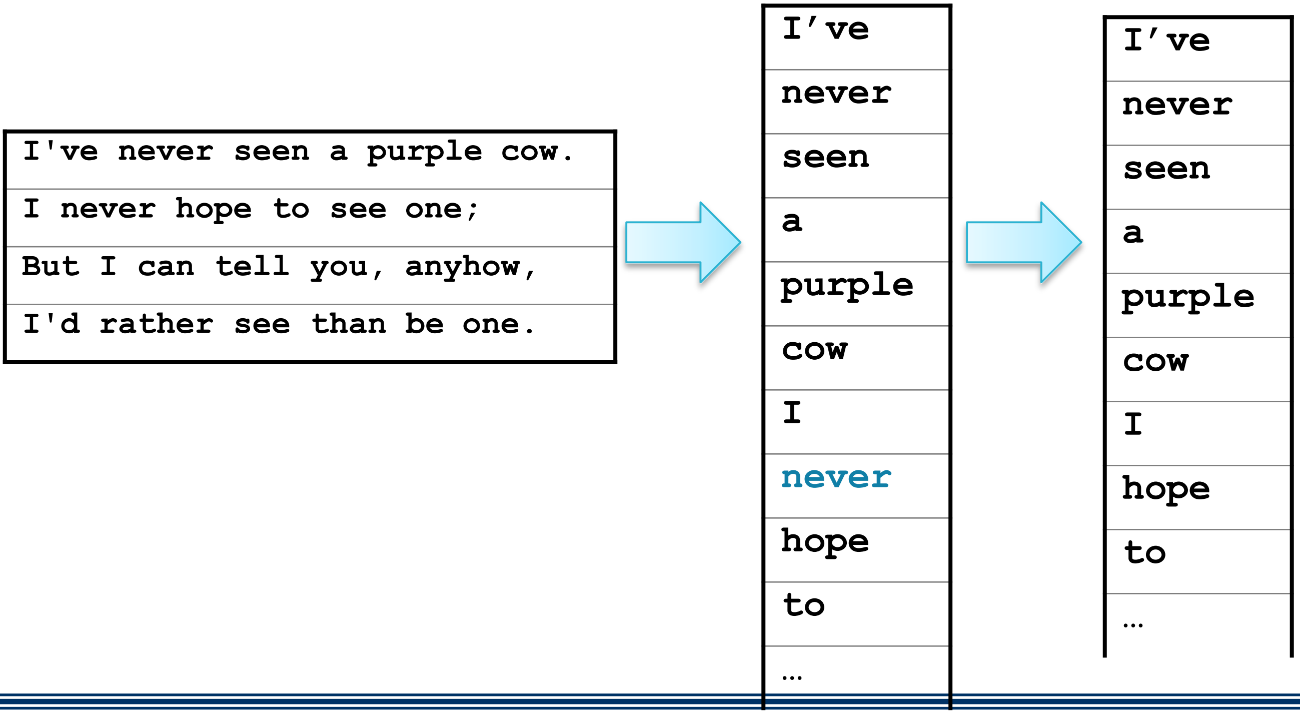
Example: `flatMap` and `distinct`

Python

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

Scala

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



Basic RDD transformations (1)

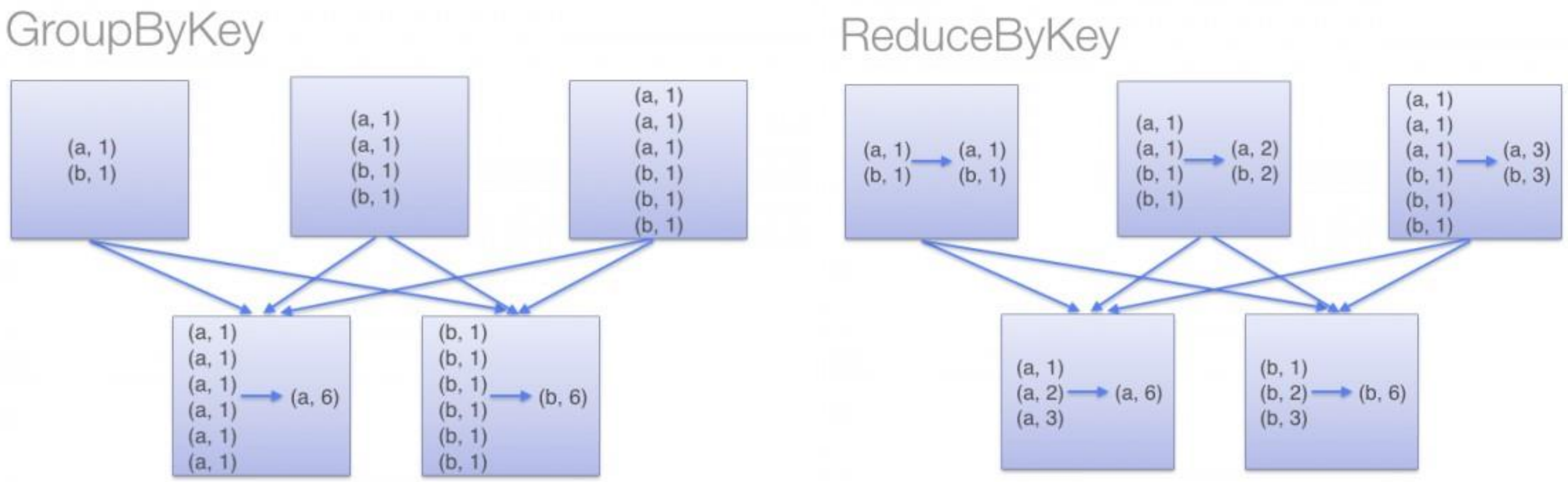
Big Data Engineering For Analytics

Function name	Purpose	Example	Result
map(func)	Apply a function to each element in the RDD and return an RDD of the result. Using map() transformation we take in any function, and that function is applied to every element of RDD	`RDD = (1,2,3,3)` `rdd.map(x => x + 1)`	`{2, 3, 4, 4}`
mapPartitions(fun)	The **mapPartition** converts each *partition* of the source RDD into many elements of the result. In mapPartition(), the map() function is applied on each partitions simultaneously.	`sc.parallelize(1 to 9, 3)` `.mapPartitions(x=>(Array("Hello")` `.iterator)).collect()`	`Array(hello,hello,hello` `)`
flatMap()	Apply a function to each element in the RDD and return an RDD of the contents of the iterators returned. Often used to extract words.	`rdd.flatMap(x => x.to(3))`	`{1, 2, 3, 2, 3, 3, 3}`
filter()	Return an RDD consisting of only elements that pass the condition passed to filter().	`rdd.filter(x => x != 1)`	`{2, 3, 3}`

Basic RDD transformations (2).

Function name	Purpose	Example	Result
distinct()	Remove duplicates.	`rdd.distinct()`	`{1, 2, 3}`
groupByKey()	When called on a dataset of (K, V) pairs, returns a dataset of (K, Iterable<V>) pairs.	`groupRDD = sc.parallelize(Array((‘k’,5),(‘s’,3),(‘s’,4),(‘p’,7))).groupByKey()`	`Array((p,CompactBuffer(7)) (s,CompactBuffer(3,4)) (k,CompactBuffer(5)))`
reduceByKey(func, [numTasks])	When called on a dataset of (K, V) pairs, returns a 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	`reduceRDD = sc.parallelize(Array((‘k’,5),(‘s’,3),(‘s’,4),(‘p’,7))).reduceByKey((x,y)=>(x+y))`	`Array((p,7),(s,7),(k,5))`
sortByKey()	When called on a dataset of (K, V) pairs where K implements Ordered, returns a dataset of (K, V) pairs sorted by keys in ascending or descending order, as specified in the boolean ascending argument.	`sortRDD = sc.parallelize(Array(("India",91),("USA",1),("Brazil",55))).sortByKey()`	`Array((Brazil,55),(Inida,9),(USA,1))`

GroupByKey vs ReduceByKey



https://www.edureka.co/community/11996/groupbykey-vs-reducebykey-in-apache-spark

Lazy Execution (1)

Data in RDDs is not processed until an *action* is performed

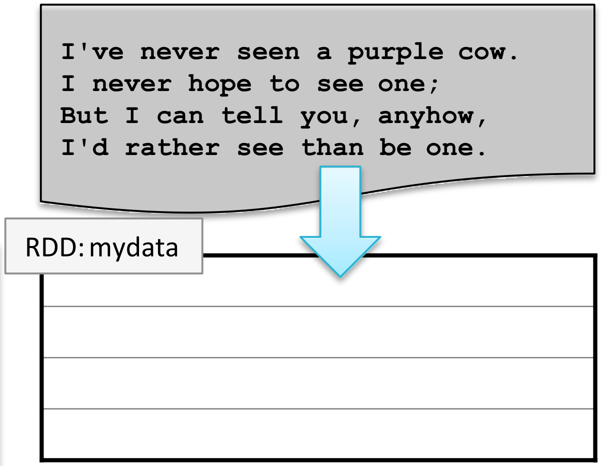
```
I've never seen a purple cow.
I never hope to see one;
But I can tell you, anyhow,
I'd rather see than be one.
```

```
>
```

Lazy Execution (2)

Data in RDDs is not processed until an *action* is performed

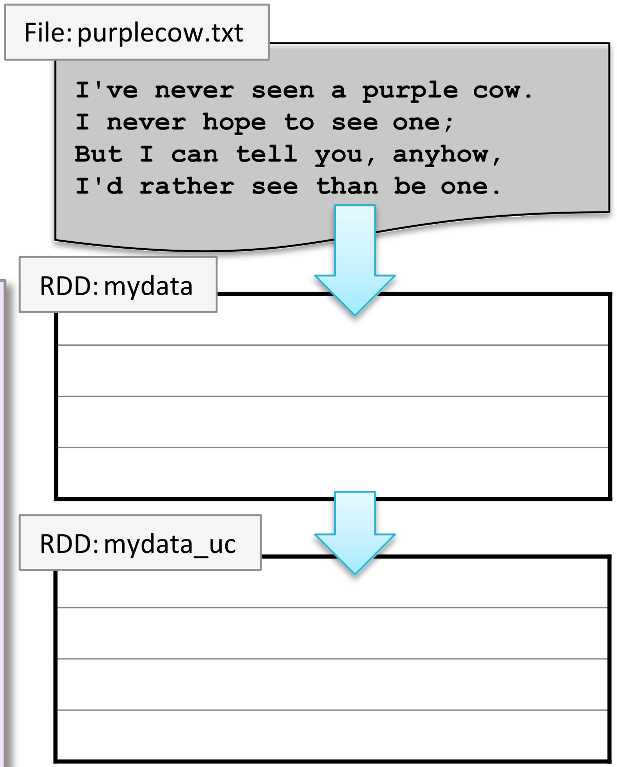
```
> val mydata = sc.textFile("purplecow.txt")
```


I've never seen a purple cow.
I never hope to see one;
But I can tell you, anyhow,
I'd rather see than be one.
RDD: mydata

Lazy Execution (3)

Data in RDDs is not processed until an *action* is performed

```
> val mydata = sc.textFile("purplecow.txt")
> val mydata_uc = mydata.map(line =>
  line.toUpperCase())
```

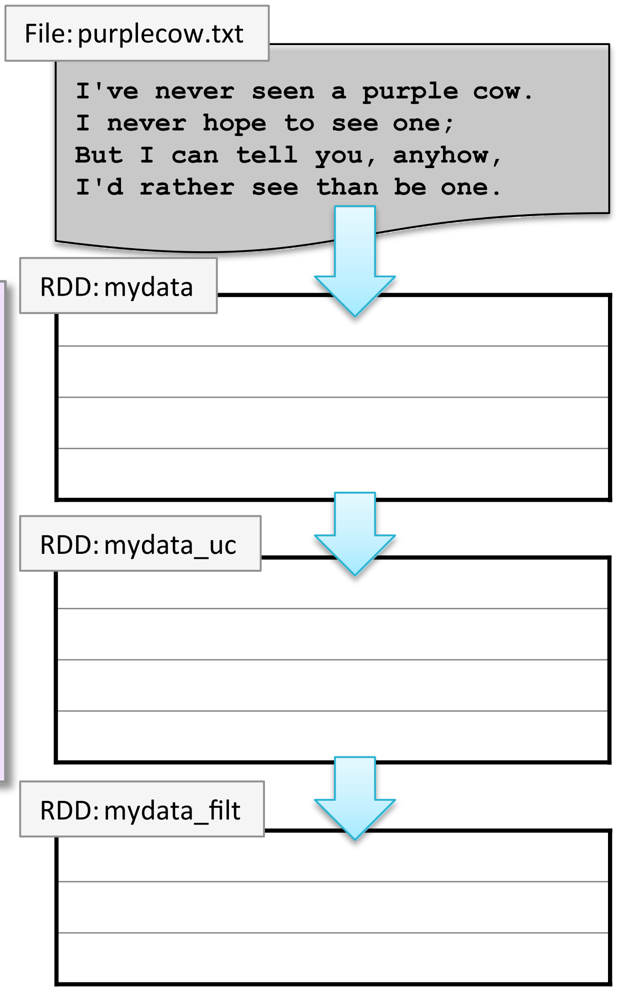


```
I've never seen a purple cow.
I never hope to see one;
But I can tell you, anyhow,
I'd rather see than be one.
```

Lazy Execution (4)

Data in RDDs is not processed until an *action* is performed

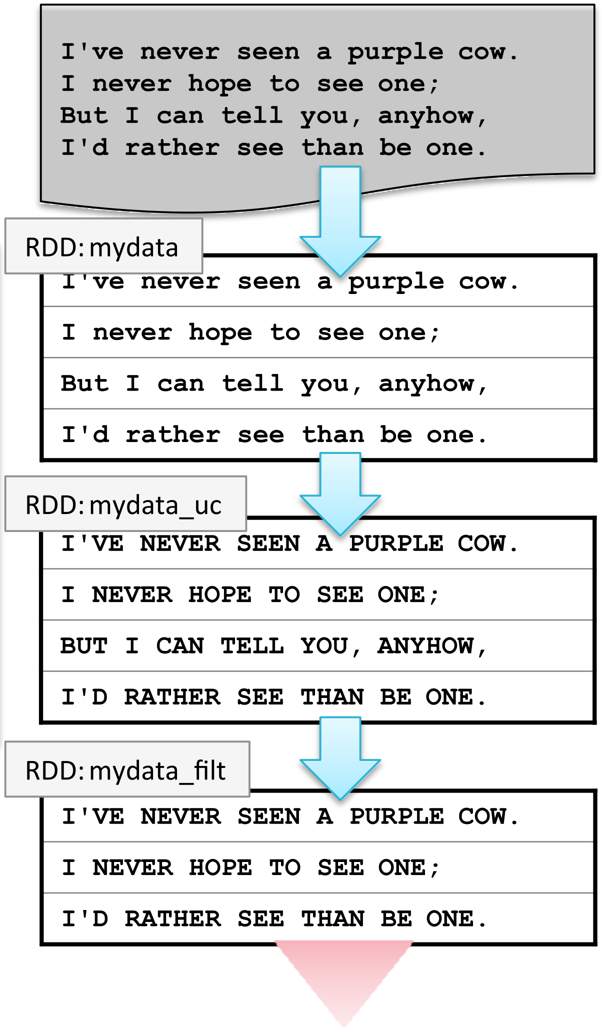
```
> val mydata = sc.textFile("purplecow.txt")
> val mydata_uc = mydata.map(line =>
  line.toUpperCase())
> val mydata_filt = mydata_uc.filter(line
  => line.startsWith("I"))
```



Lazy Execution (5).

Data in RDDs is not processed until an *action* is performed

```
> val mydata = sc.textFile("purplecow.txt")
> val mydata_uc = mydata.map(line =>
  line.toUpperCase())
> val mydata_filt = mydata_uc.filter(line
  => line.startsWith("I"))
> mydata_filt.count()
3
```



I've never seen a purple cow.
I never hope to see one;
But I can tell you, anyhow,
I'd rather see than be one.
RDD: mydata
I've never seen a purple cow.
I never hope to see one;
But I can tell you, anyhow,
I'd rather see than be one.
RDD: mydata_uc
I'VE NEVER SEEN A PURPLE COW.
I NEVER HOPE TO SEE ONE;
BUT I CAN TELL YOU, ANYHOW,
I'D RATHER SEE THAN BE ONE.
RDD: mydata_filt
I'VE NEVER SEEN A PURPLE COW.
I NEVER HOPE TO SEE ONE;
I'D RATHER SEE THAN BE ONE.

Chaining Transformations (Scala)

Transformations may be chained together

```
> val mydata = sc.textFile("purplecow.txt")
> val mydata_uc = mydata.map(line => line.toUpperCase())
> val mydata_filt = mydata_uc.filter(line => line.startsWith("I"))
> mydata_filt.count()
3
```

is exactly equivalent to

```
> sc.textFile("purplecow.txt").map(line => line.toUpperCase()).
   filter(line => line.startsWith("I")).count()
3
```

Chaining Transformations (Python)

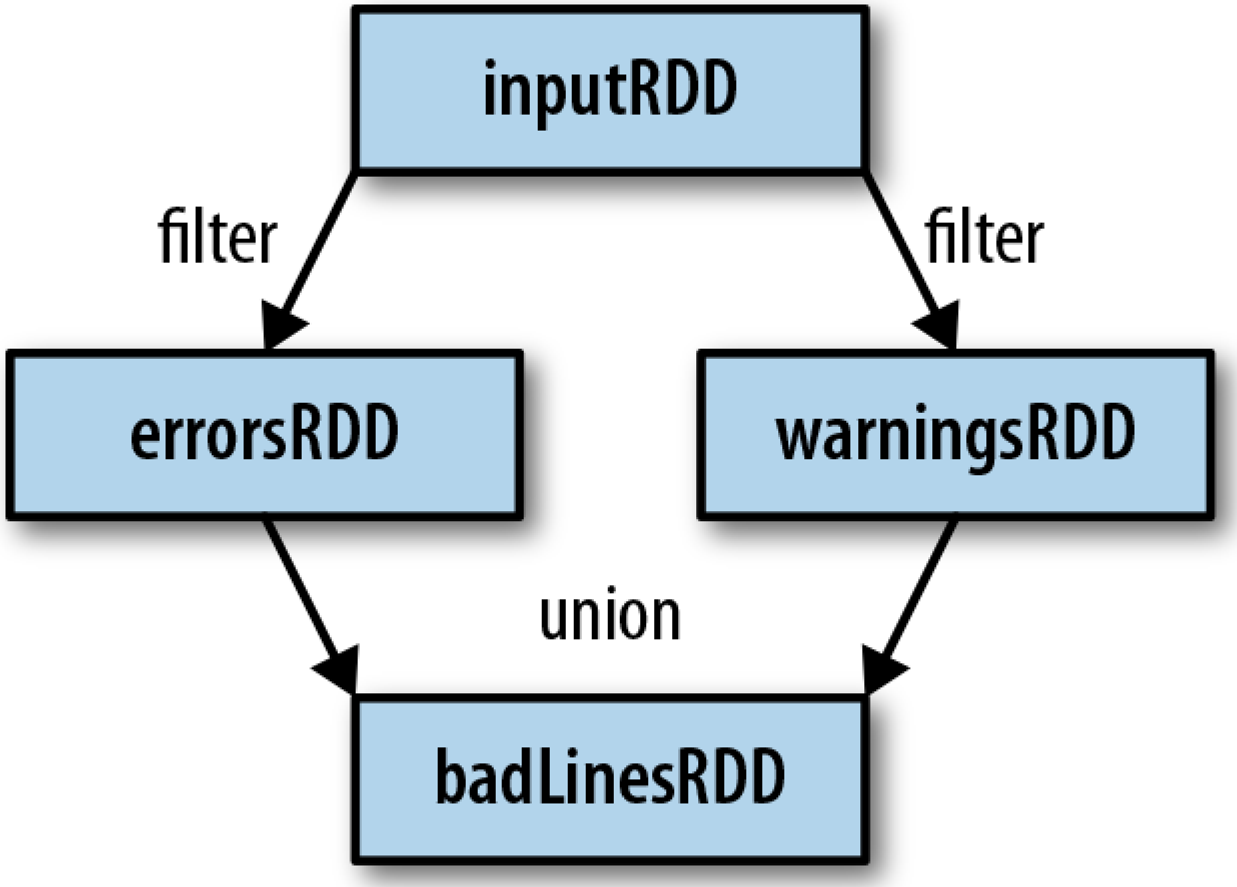
Same example in Python

```
> mydata = sc.textFile("purplecow.txt")
> mydata_uc = mydata.map(lambda s: s.upper())
> mydata_filt = mydata_uc.filter(lambda s: s.startswith('I'))
> mydata_filt.count()
3
```

is exactly equivalent to

```
> sc.textFile("purplecow.txt").map(lambda line: line.upper()) \
   .filter(lambda line: line.startswith('I')).count()
3
```

RDD lineage graph created during log analysis



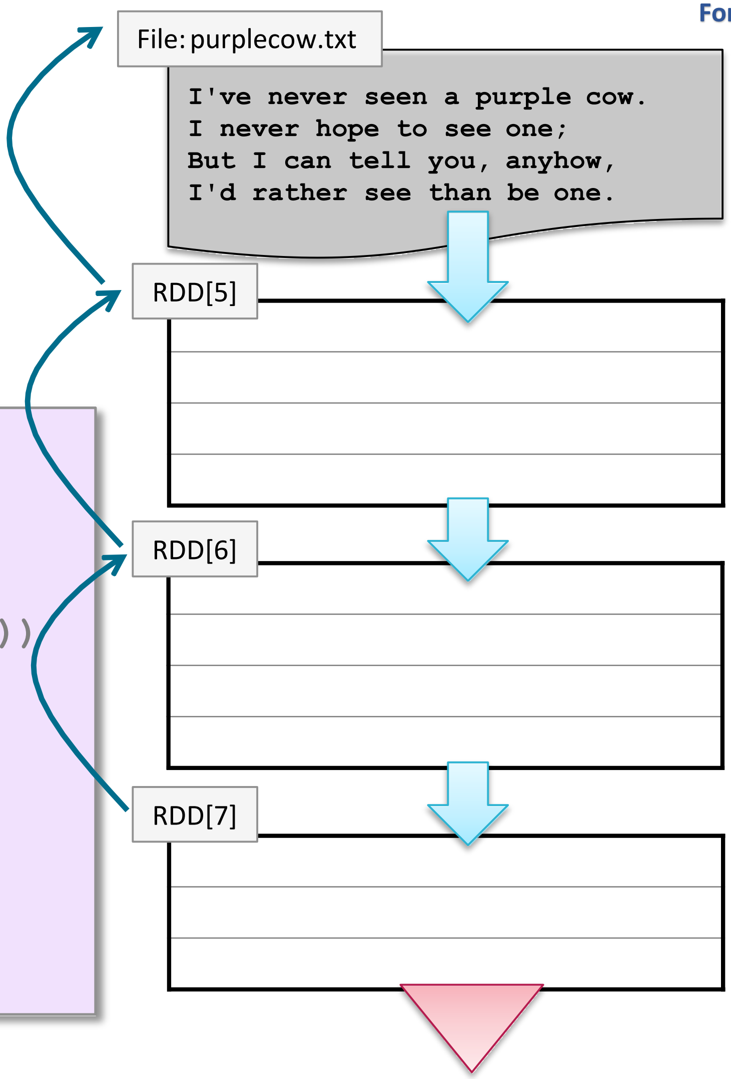
RDD Lineage and `toDebugString` (Scala)

Spark maintains each RDD's *lineage* – the previous RDDs on which it depends

Use `toDebugString` to view the lineage of an RDD

```
> val mydata_filt =
    sc.textFile("purplecow.txt").
    map(line => line.toUpperCase()).
    filter(line => line.startsWith("I"))
> mydata_filt.toDebugString

(2) FilteredRDD[7] at filter …
 |  MappedRDD[6] at map …
 |  purplecow.txt MappedRDD[5] …
 |  purplecow.txt HadoopRDD[4] …
```


File: purplecow.txt
I've never seen a purple cow.
I never hope to see one;
But I can tell you, anyhow,
I'd rather see than be one.
RDD[5]
RDD[6]
RDD[7]

RDD Lineage and `toDebugString` (Python)

- `toDebugString` output is not displayed as nicely in Python

```
> mydata_filt.toDebugString()
(1) PythonRDD[8] at RDD at …\n |  purplecow.txt MappedRDD[7] at textFile
at …[]\n |  purplecow.txt HadoopRDD[6] at textFile at …[]
```

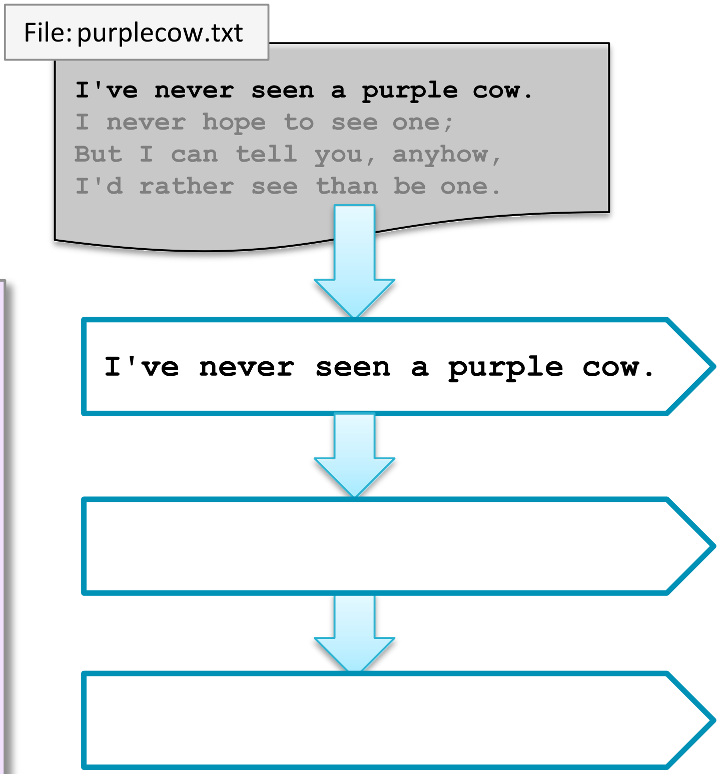
- Use `print` for output

```
> print mydata_filt.toDebugString()
(1) PythonRDD[8] at RDD at …
 |  purplecow.txt MappedRDD[7] at textFile at …
 |  purplecow.txt HadoopRDD[6] at textFile at …
```

Pipelining (1)

When possible, Spark will perform sequences of transformations by row so no data is stored

```
> val mydata = sc.textFile("purplecow.txt")
> val mydata_uc = mydata.map(line =>
  line.toUpperCase())
> val mydata_filt = mydata_uc.filter(line
  => line.startsWith("I"))
> mydata_filt.take(2)
```

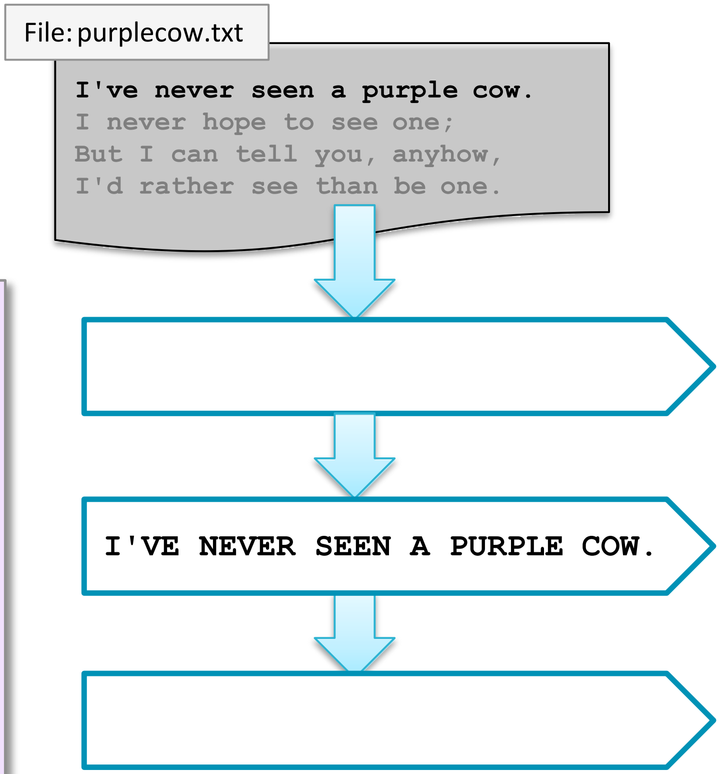

File: purplecow.txt
I've never seen a purple cow.
I never hope to see one;
But I can tell you, anyhow,
I'd rather see than be one.
I've never seen a purple cow.

Pipelining (2)

Big Data Engineering For Analytics

When possible, Spark will perform sequences of transformations by row so no data is stored

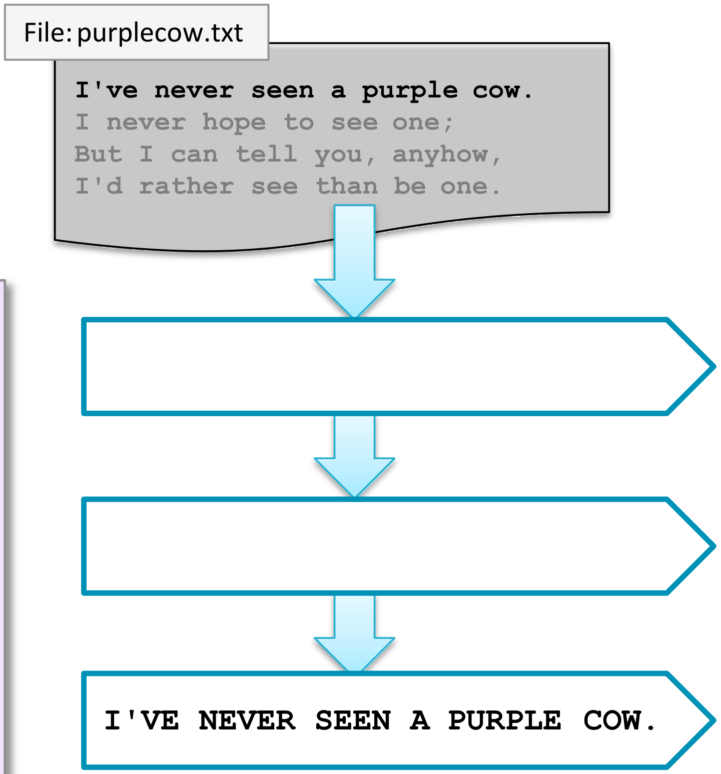
```
> val mydata = sc.textFile("purplecow.txt")
> val mydata_uc = mydata.map(line =>
  line.toUpperCase())
> val mydata_filt = mydata_uc.filter(line
  => line.startsWith("I"))
> mydata_filt.take(2)
```


File: purplecow.txt
I've never seen a purple cow.
I never hope to see one;
But I can tell you, anyhow,
I'd rather see than be one.
I'VE NEVER SEEN A PURPLE COW.

Pipelining (3)

When possible, Spark will perform sequences of transformations by row so no data is stored

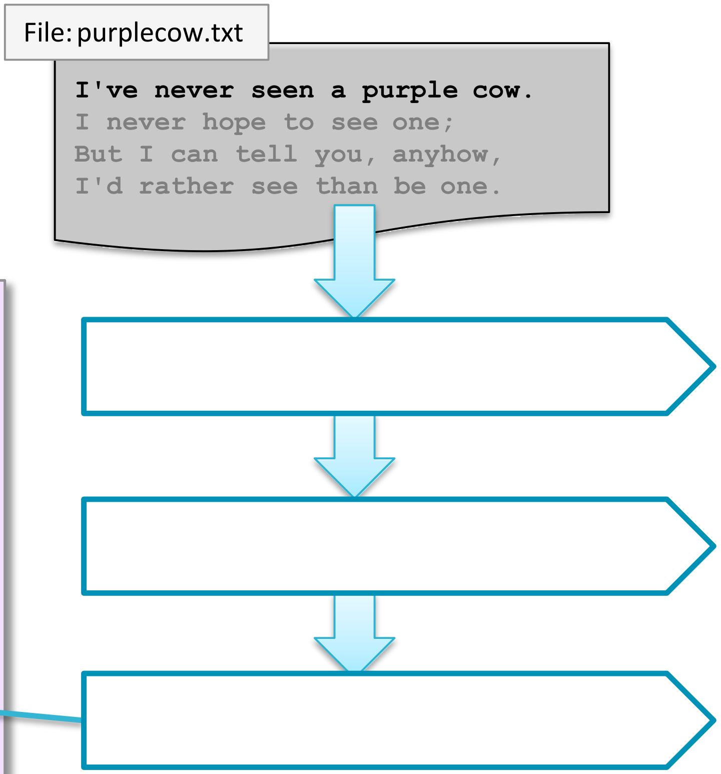
```
> val mydata = sc.textFile("purplecow.txt")
> val mydata_uc = mydata.map(line =>
  line.toUpperCase())
> val mydata_filt = mydata_uc.filter(line
  => line.startsWith("I"))
> mydata_filt.take(2)
```


File: purplecow.txt
I've never seen a purple cow.
I never hope to see one;
But I can tell you, anyhow,
I'd rather see than be one.
I'VE NEVER SEEN A PURPLE COW.

Pipelining (4)

When possible, Spark will perform sequences of transformations by row so no data is stored

```
> val mydata = sc.textFile("purplecow.txt")
> val mydata_uc = mydata.map(line =>
  line.toUpperCase())
> val mydata_filt = mydata_uc.filter(line
  => line.startsWith("I"))
> mydata_filt.take(2)
I'VE NEVER SEEN A PURPLE COW.
```

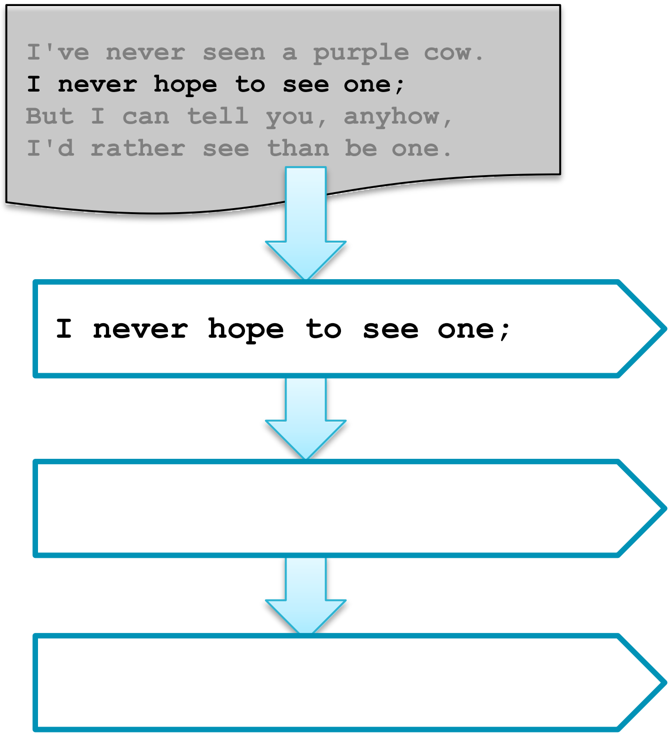


```
I've never seen a purple cow.
I never hope to see one;
But I can tell you, anyhow,
I'd rather see than be one.
```

Pipelining (5)

When possible, Spark will perform sequences of transformations by row so no data is stored

```
> val mydata = sc.textFile("purplecow.txt")
> val mydata_uc = mydata.map(line =>
  line.toUpperCase())
> val mydata_filt = mydata_uc.filter(line
  => line.startsWith("I"))
> mydata_filt.take(2)
I'VE NEVER SEEN A PURPLE COW.
```



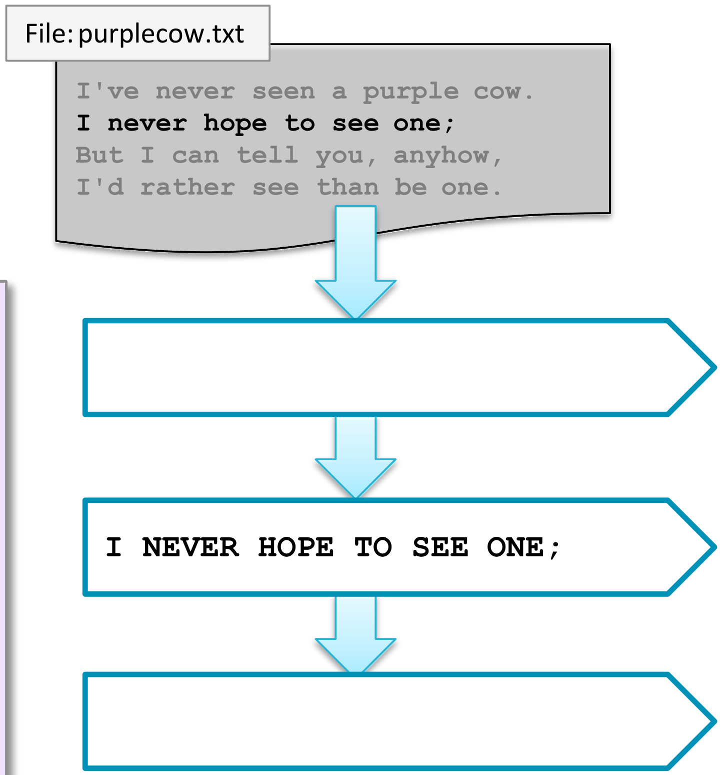
```
I've never seen a purple cow.
I never hope to see one;
But I can tell you, anyhow,
I'd rather see than be one.
```

```
I never hope to see one;
```

Pipelining (6)

When possible, Spark will perform sequences of transformations by row so no data is stored

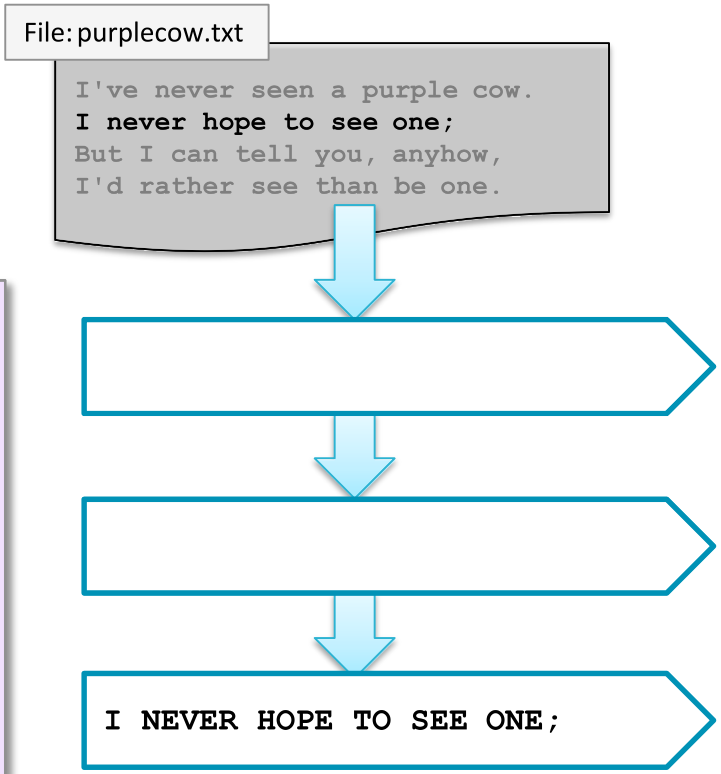
```
> val mydata = sc.textFile("purplecow.txt")
> val mydata_uc = mydata.map(line =>
  line.toUpperCase())
> val mydata_filt = mydata_uc.filter(line
  => line.startsWith("I"))
> mydata_filt.take(2)
I'VE NEVER SEEN A PURPLE COW.
```


File: purplecow.txt
I've never seen a purple cow.
I never hope to see one;
But I can tell you, anyhow,
I'd rather see than be one.
I NEVER HOPE TO SEE ONE;

Pipelining (7)

When possible, Spark will perform sequences of transformations by row so no data is stored

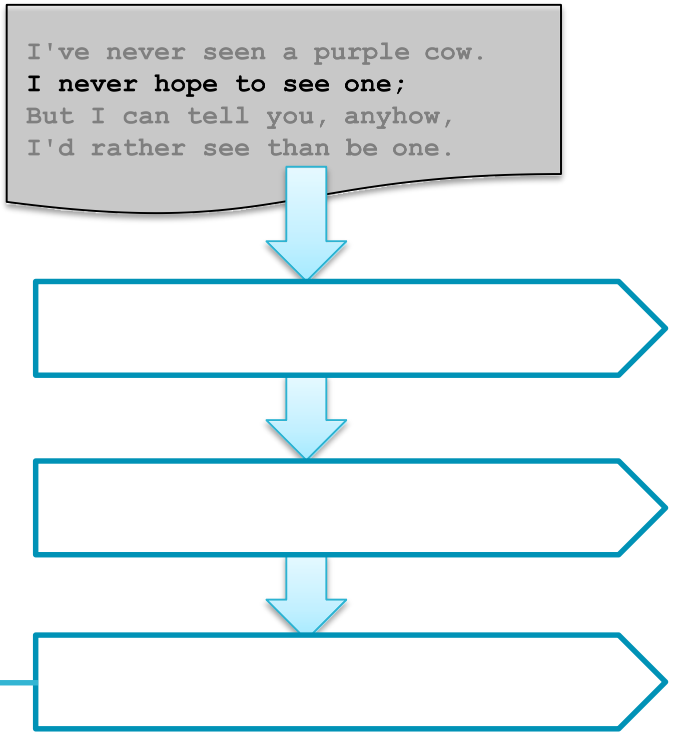
```
> val mydata = sc.textFile("purplecow.txt")
> val mydata_uc = mydata.map(line =>
  line.toUpperCase())
> val mydata_filt = mydata_uc.filter(line
  => line.startsWith("I"))
> mydata_filt.take(2)
I'VE NEVER SEEN A PURPLE COW.
```


File: purplecow.txt
I've never seen a purple cow.
I never hope to see one;
But I can tell you, anyhow,
I'd rather see than be one.
I NEVER HOPE TO SEE ONE;

Pipelining (8).

When possible, Spark will perform sequences of transformations by row so no data is stored

```
> val mydata = sc.textFile("purplecow.txt")
> val mydata_uc = mydata.map(line =>
  line.toUpperCase())
> val mydata_filt = mydata_uc.filter(line
  => line.startsWith("I"))
> mydata_filt.take(2)
I'VE NEVER SEEN A PURPLE COW.
I NEVER HOPE TO SEE ONE;
```


I've never seen a purple cow.
I never hope to see one;
But I can tell you, anyhow,
I'd rather see than be one.

Functional Programming in Spark

- Spark depends heavily on the concepts of functional programming
  - Functions are the fundamental unit of programming
  - Functions have input and output only
    - No state or side effects
- Key concepts
  - Passing functions as input to other functions
  - Anonymous functions
- Many RDD operations take functions as parameters
  - Pseudo code for the RDD map operation
  - Applies function fn to each record in the RDD

```
RDD {
    map(fn(x)) {
         foreach record
         in      rdd
         emit fn(record)
    }
}
```

Example: Passing Named Functions

Python

```
> def toUpper(s):
      return s.upper()
> mydata = sc.textFile("purplecow.txt")
> mydata.map(toUpper).take(2)
```

Scala

```
> def toUpper(s: String): String =
    { s.toUpperCase }
> val mydata = sc.textFile("purplecow.txt")
> mydata.map(toUpper).take(2)
```

Passing Anonymous Functions

- Functions defined in-line without an identifier
  - Best for short, one-off functions
- Supported in many programming languages
  - Python: `lambda x: . . .`
  - Scala: `x => . . .`

Python:

```
> mydata.map(lambda line: line.upper()).take(2)
```

Scala:

```
> mydata.map(line => line.toUpperCase()).take(2)
```

OR

```
> mydata.map(_.toUpperCase()).take(2)
```

Scala allows anonymous parameters using underscore (_)

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 Collection

- 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

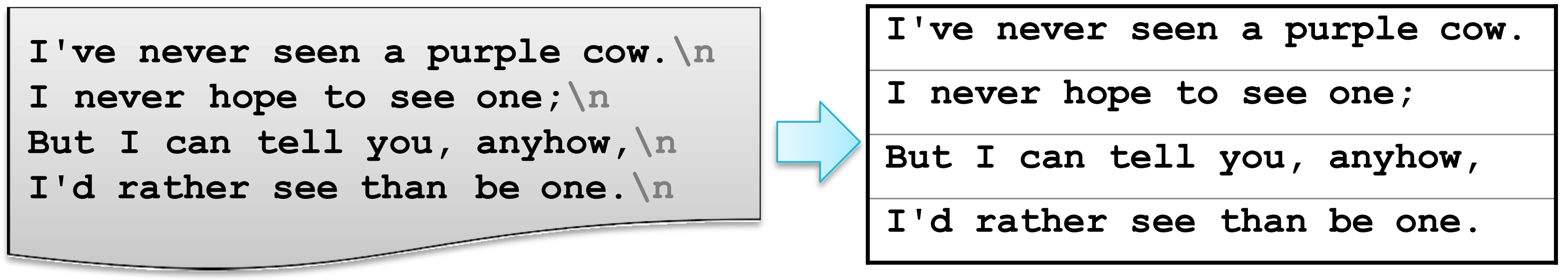
Creating RDDs from files (1)

- **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



```
I've never seen a purple cow.\n
I never hope to see one;\n
But I can tell you, anyhow,\n
I'd rather see than be one.\n
```

I've never seen a purple cow.
I never hope to see one;
But I can tell you, anyhow,
I'd rather see than be one.

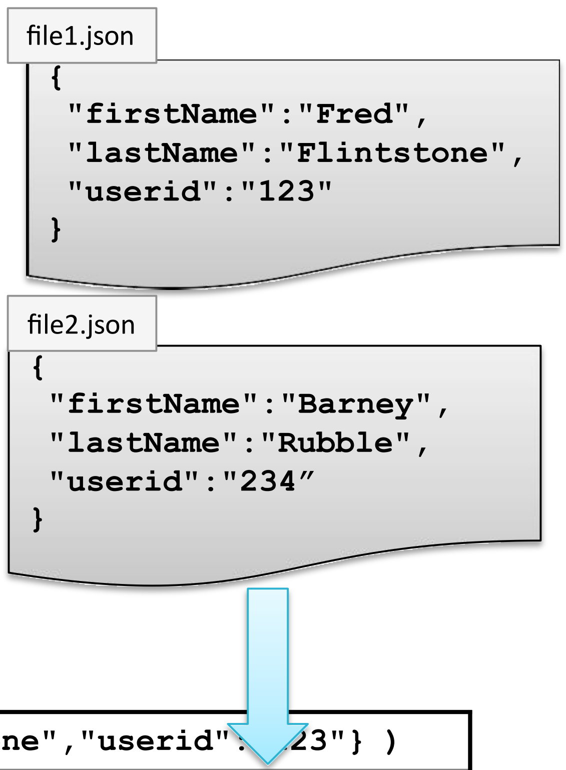
- **`textFile`** only works with line-delimited text files
- What about other formats?

Input and Output Formats

- **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
- **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)



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

```
{
 "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"]
```

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))
```

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
- **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`**

Multi-RDD Transformations

Function name	Purpose
union()	Get the elements of both the RDD. The key rule is that two RDDs should be of the same type
intersection()	Get only the common element of both the RDD in new RDD. The key rule is that two RDDs should be of the same type
subtract()	It returns an RDD that has only value present in the first RDD and not in second RDD.
zip()	The zip function is used to combine two RDDs into the RDD of the Key / Value form. The default number of RDD partitions and the number of elements are the same, otherwise an exception is thrown.
join()	It combines the fields from two table using common values. join() operation in Spark is defined on pair-wise RDD.

Example: Multi-RDD Transformations

rdd1	rdd2
Chicago	San Francisco
Boston	Boston
Paris	Amsterdam
San Francisco	Mumbai
Tokyo	McMurdo Station

`rdd1.subtract(rdd2)`

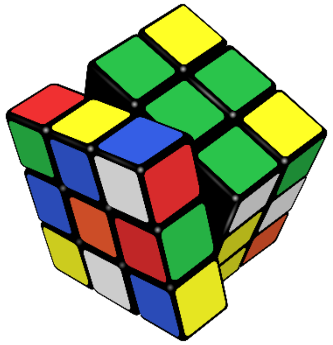
Tokyo
Paris
Chicago

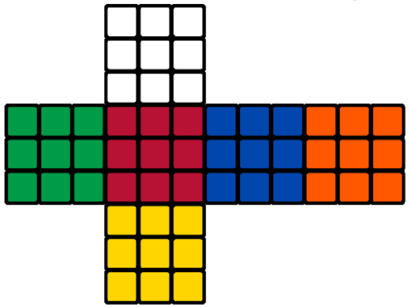
`rdd1.zip(rdd2)`

(Chicago,San Francisco)
(Boston,Boston)
(Paris,Amsterdam)
(San Francisco,Mumbai)
(Tokyo,McMurdo Station)

`rdd1.union(rdd2)`

Chicago
Boston
Paris
San Francisco
Tokyo
San Francisco
Boston
Amsterdam
Mumbai
McMurdo Station





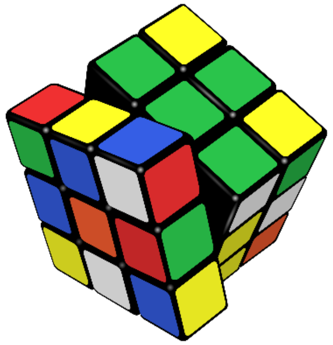
Summary

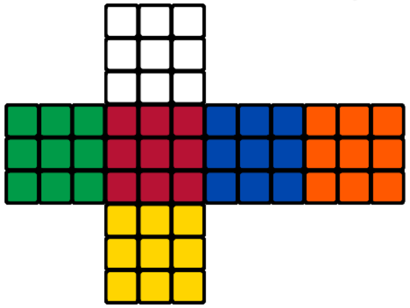
Simplicity is the ultimate sophistication.

~Leonardo da Vinci

Essential Points

- The main abstraction Spark provides is a **resilient distributed dataset (RDD),** which is a collection of elements partitioned across the nodes of the cluster that can be operated on in parallel.
  - ➢The ability to always re-compute an RDD is actually why RDDs are called "resilient." When a machine holding RDD data fails, Spark uses this ability to re-compute the missing partitions, transparent to the user.





References

Genius ain't anything more than elegant common sense.

~Josh Billings

References

- Official Spark Documentation and Wiki
- Programmer and User Guides
- Books:
  - Learning Spark, by Holden Karau, Andy Konwinski, Patrick Wendell and Matei Zaharia (O'Reilly Media)
  - Spark in Action, by Marko Bonaci and Petar Zecevic (Manning)
  - Fast Data Processing with Spark, by Krishna Sankar and Holden Karau (Packt Publishing)
  - Spark Cookbook, by Rishi Yadav (Packt Publishing)
  - Mastering Apache Spark, by Mike Frampton (Packt Publishing)