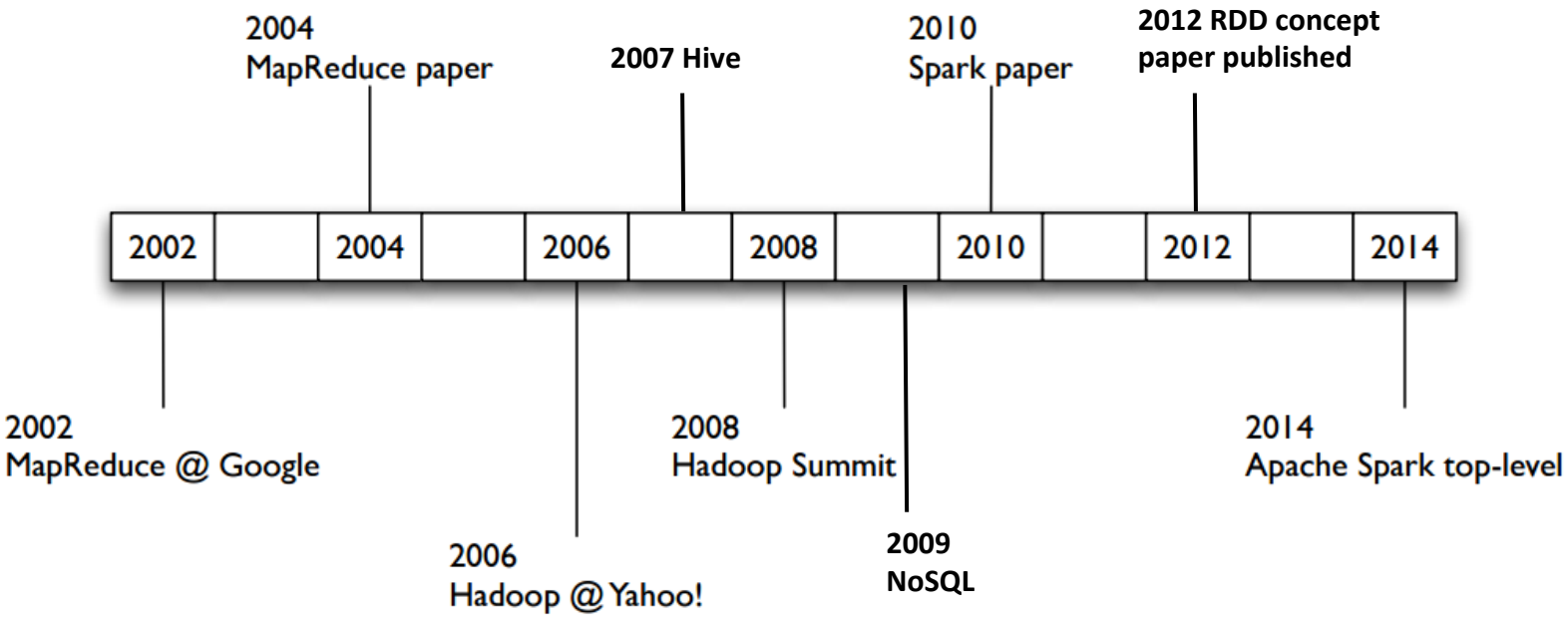
Lightning-fast cluster computing

Spark introduction

RDD

Building and running Spark applications



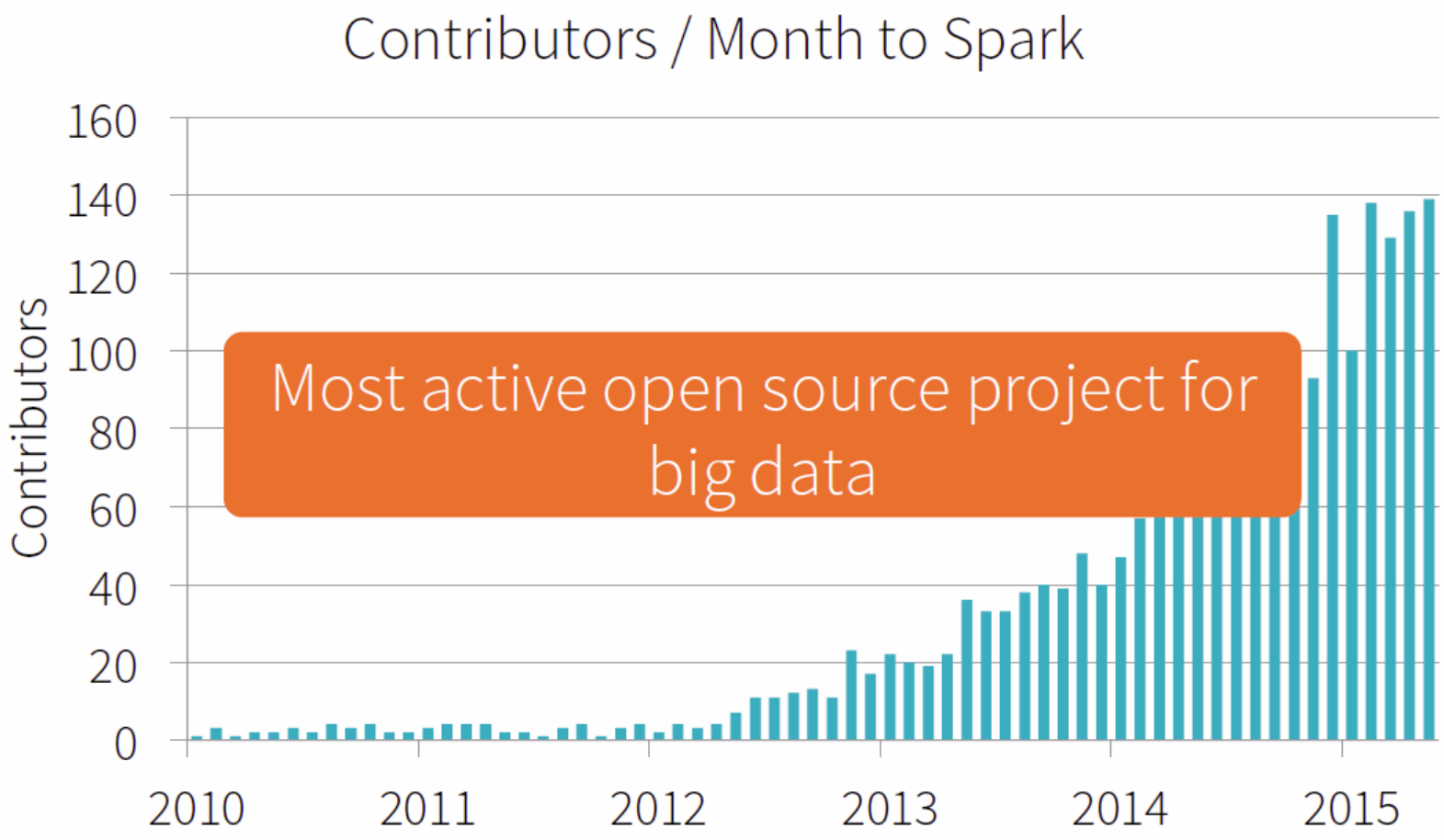
Along with Pig

The beginning of Spark

- Originator: Matei Zaharia
- Start in 2009 as a class project in UC Berkeley's AMPlab

  - Need to do machine learning faster on HDFS
- Doctoral dissertation (2013)
- http://www.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-12.pdf

- Hear Matei talking
- https://www.youtube.com/watch?v=BFtQrfQ2rn0


Contributors / Month to Spark
Contributors
160
140
120
100
80
60
40
20
0
2010
2011
2012
2013
2014
2015
Most active open source project for big data

databricks

Over 1000 deployments, clusters up to 8000 nodes

databricks

Many talks online at spark-summit.org

IBM


News room > News releases >

IBM Announces Major Commitment to Advance Apache®Spark™, Calling it Potentially the Most Significant Open Source Project of the Next Decade

IBM Joins Spark Community, Plans to Educate More Than 1 Million Data Scientists

- 2015.6
- https://www-03.ibm.com/press/us/en/pressrelease/47107.wss

cloudera

Apache Spark

An integrated part of CDH and supported with Cloudera Enterprise, Spark is the open standard for flexible in-memory data processing that enables batch, real-time, and advanced analytics on the Apache Hadoop platform. Via the One Platform initiative, Cloudera is committed to helping the ecosystem adopt Spark as a replacement for MapReduce in the Hadoop ecosystem as the default data execution engine for analytic workloads.

2015.9

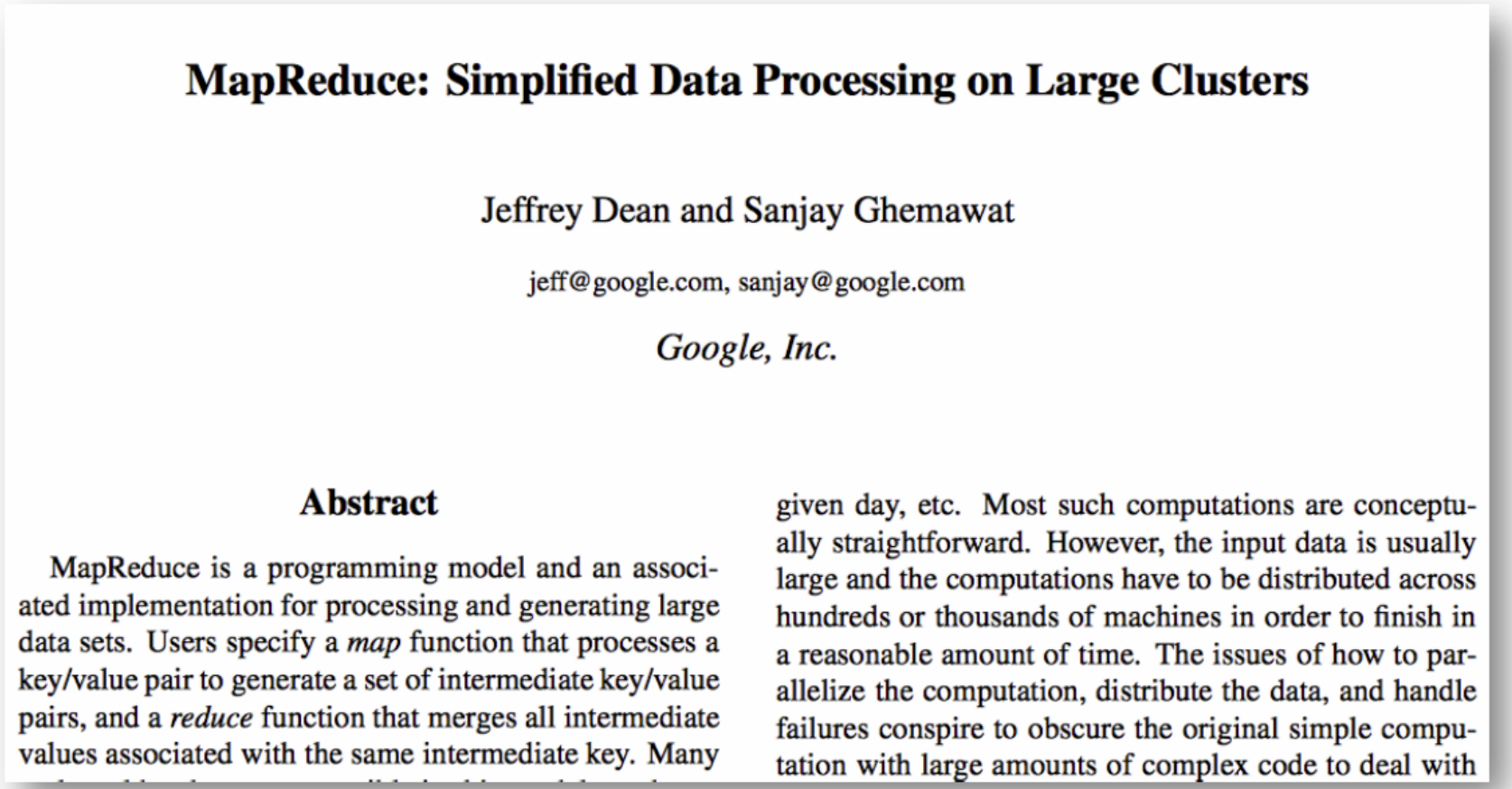
What is Spark?

- A general execution engine to improve/replace MapReduce

- Spark's operators are a strict superset of MapReduce

What's wrong with the original MapReduce?

2004



MapReduce: Simplified Data Processing on Large Clusters

Jeffrey Dean and Sanjay Ghemawat

jeff@google.com, sanjay@google.com

Google, Inc.

Abstract

MapReduce is a programming model and an associated implementation for processing and generating large data sets. Users specify a *map* function that processes a key/value pair to generate a set of intermediate key/value pairs, and a *reduce* function that merges all intermediate values associated with the same intermediate key. Many

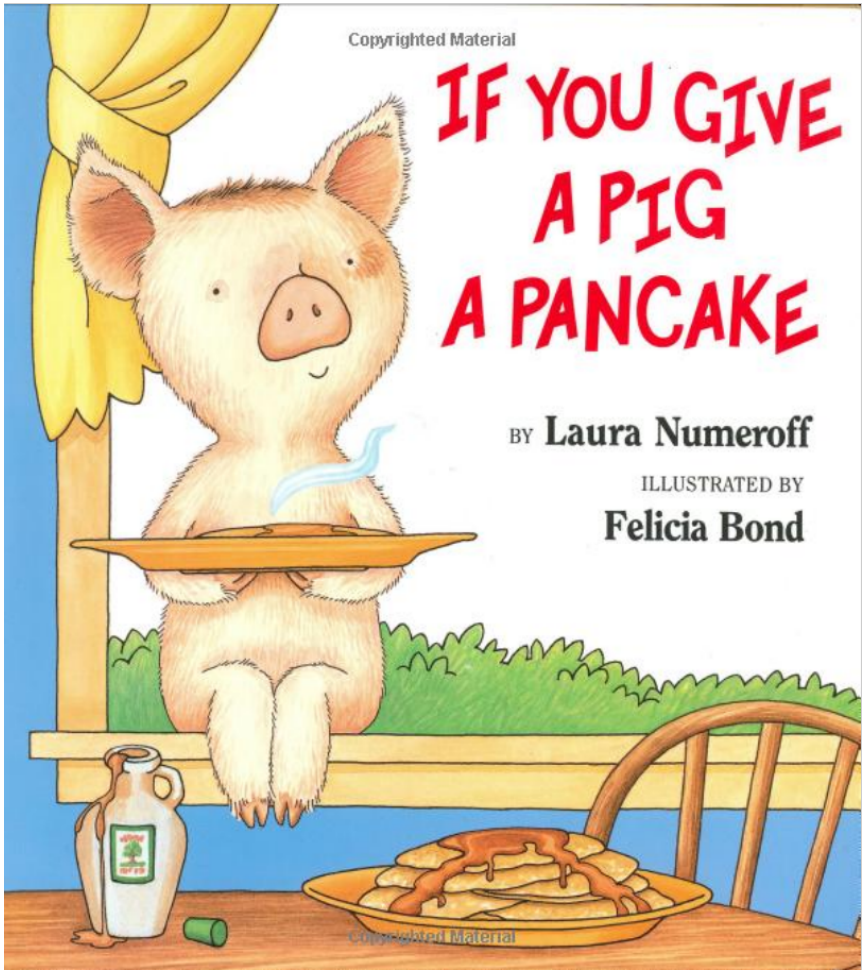
given day, etc. Most such computations are conceptually straightforward. However, the input data is usually large and the computations have to be distributed across hundreds or thousands of machines in order to finish in a reasonable amount of time. The issues of how to parallelize the computation, distribute the data, and handle failures conspire to obscure the original simple computation with large amounts of complex code to deal with

We wrote the first version of the MapReduce library in February of 2003, and made significant enhancements to it in August of 2003, including the locality optimization, dynamic load balancing of task execution across worker machines, etc. Since that time, we have been pleasantly surprised at how broadly applicable the MapReduce library has been for the kinds of problems we work on. It has been used across a wide range of domains within Google, including:

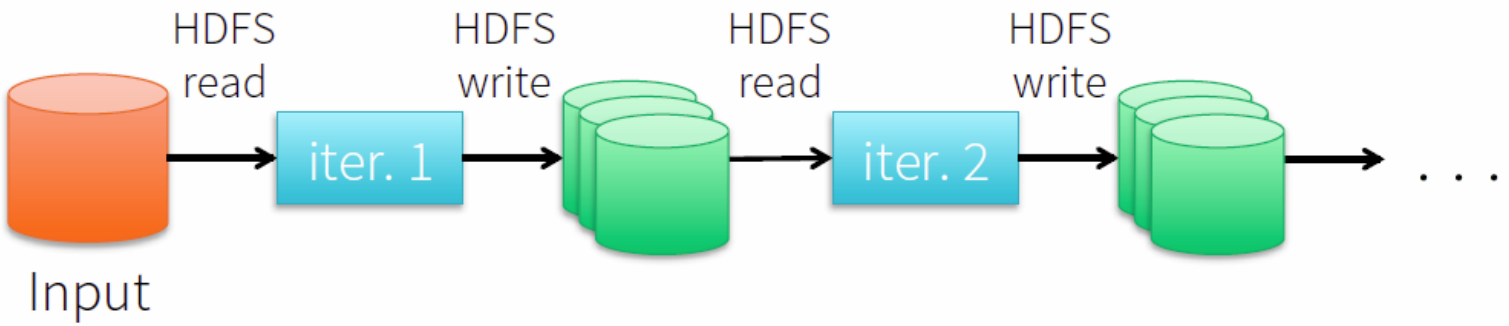
What's wrong with the original MapReduce?

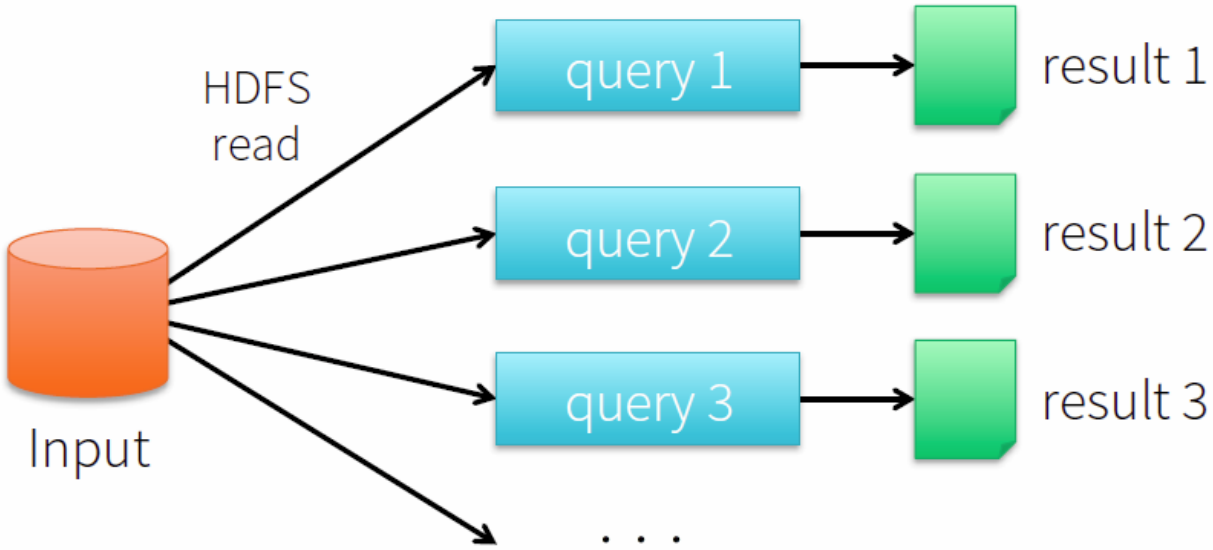
- Limitations of MapReduce.
  - Originated around year 2000. Old technology.
  - Designed for batch-processing large amount of webpages in Google
  - And it does that job very well!
- Not fit for
  - Complex, multi-passing algorithms
  - Interactive ad-hoc queries
  - Real-time stream processing

We are asking too much from MapReduce



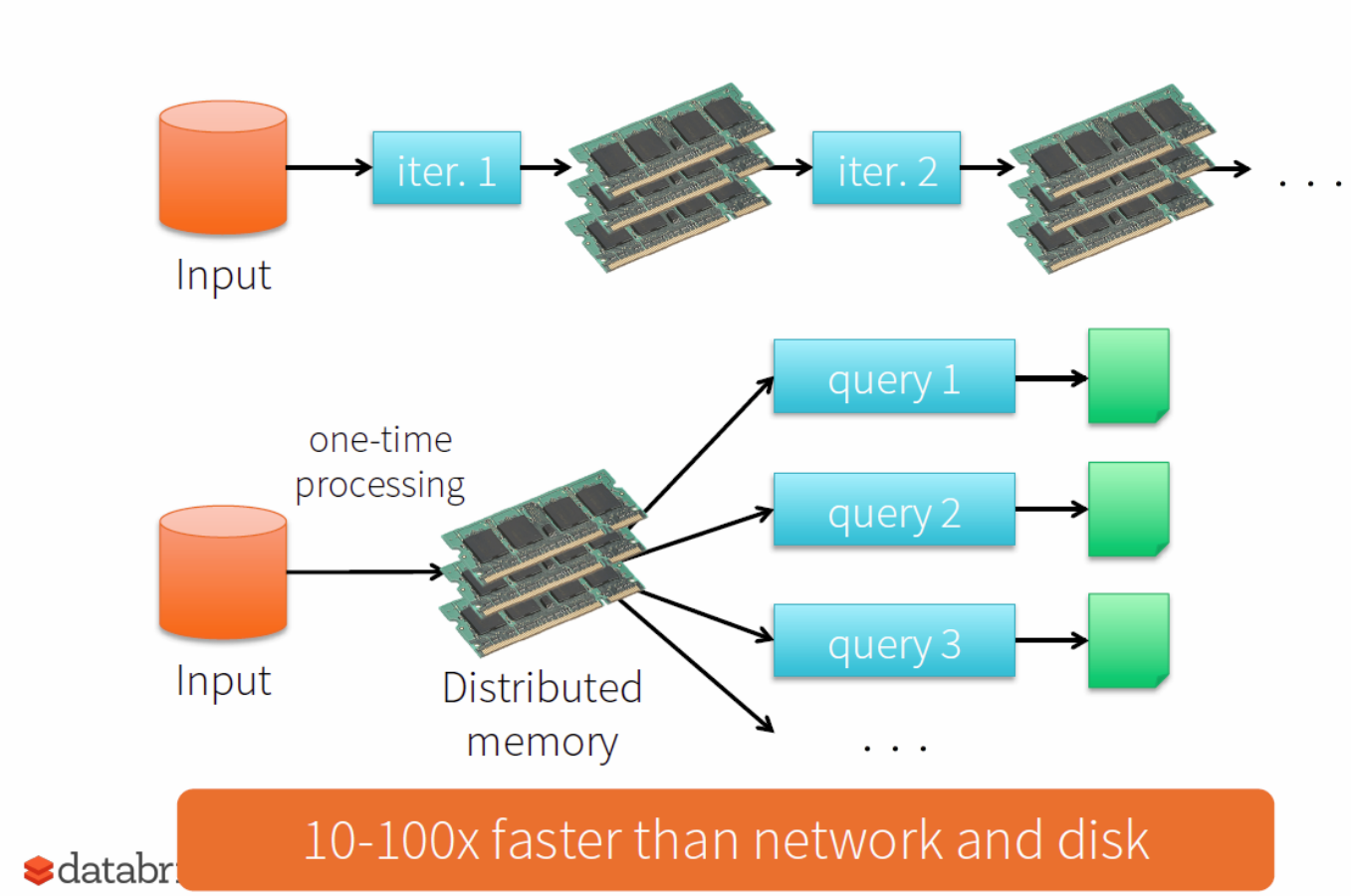
Data Sharing in MapReduce

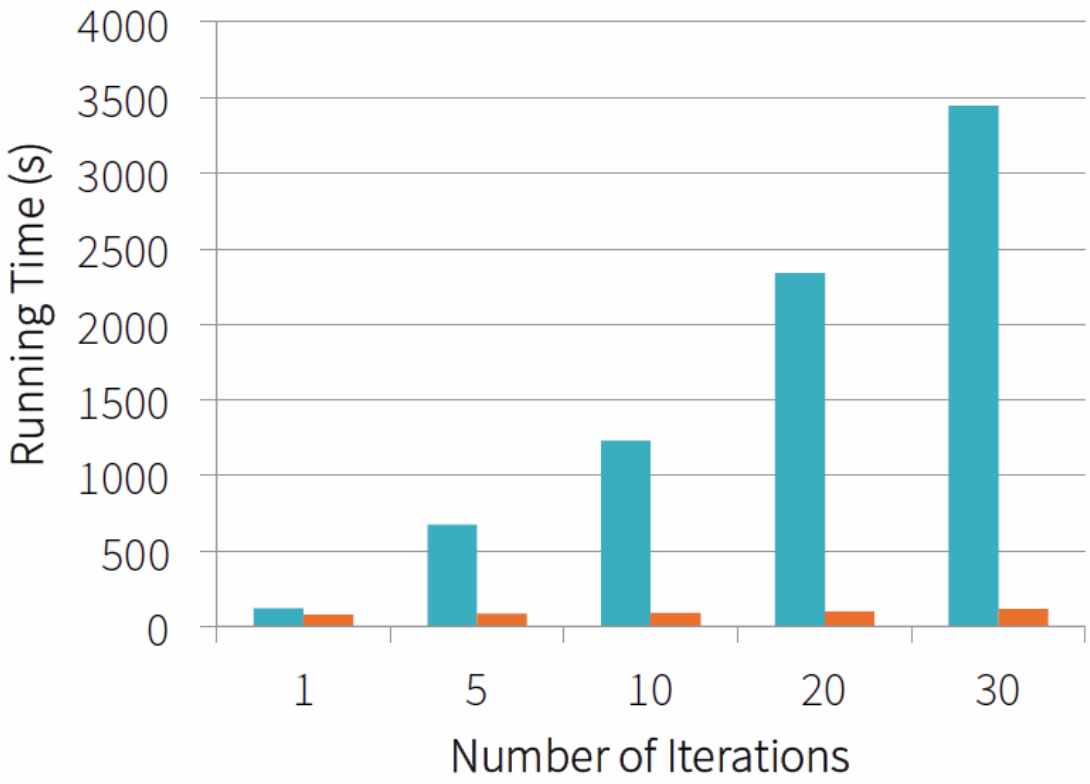




Slow due to replication and disk I/O

The Spark way!





110 s / iteration

Hadoop

Spark

first iteration 80 s
further iterations 1 s

databricks

Combining Processing Types

```
// Load data using SQL
points = ctx.sql("select latitude, longitude from tweets")

// Train a machine learning model
model = KMeans.train(points, 10)

// Apply it to a stream
sc.twitterStream(...)
  .map(lambda t: (model.predict(t.location), 1))
  .reduceByWindow("5s", lambda a, b: a + b)
```

databricks

Easier to develop on Spark

- Think of Assembly language

8086 Assembly [edit]

```
DOSSEG
.MODEL TINY
.DATA
TXT DB "Hello world!$"
.CODE
START:
        MOV ax, @DATA
        MOV ds, ax

        MOV ah, 09h             ; prepare output function
        MOV dx, OFFSET TXT      ; set offset
        INT 21h                 ; output string TXT

        MOV AX, 4C00h           ; go back to DOS
        INT 21h
END START
```

Original MapReduce

- Python

  print “Hello world!”

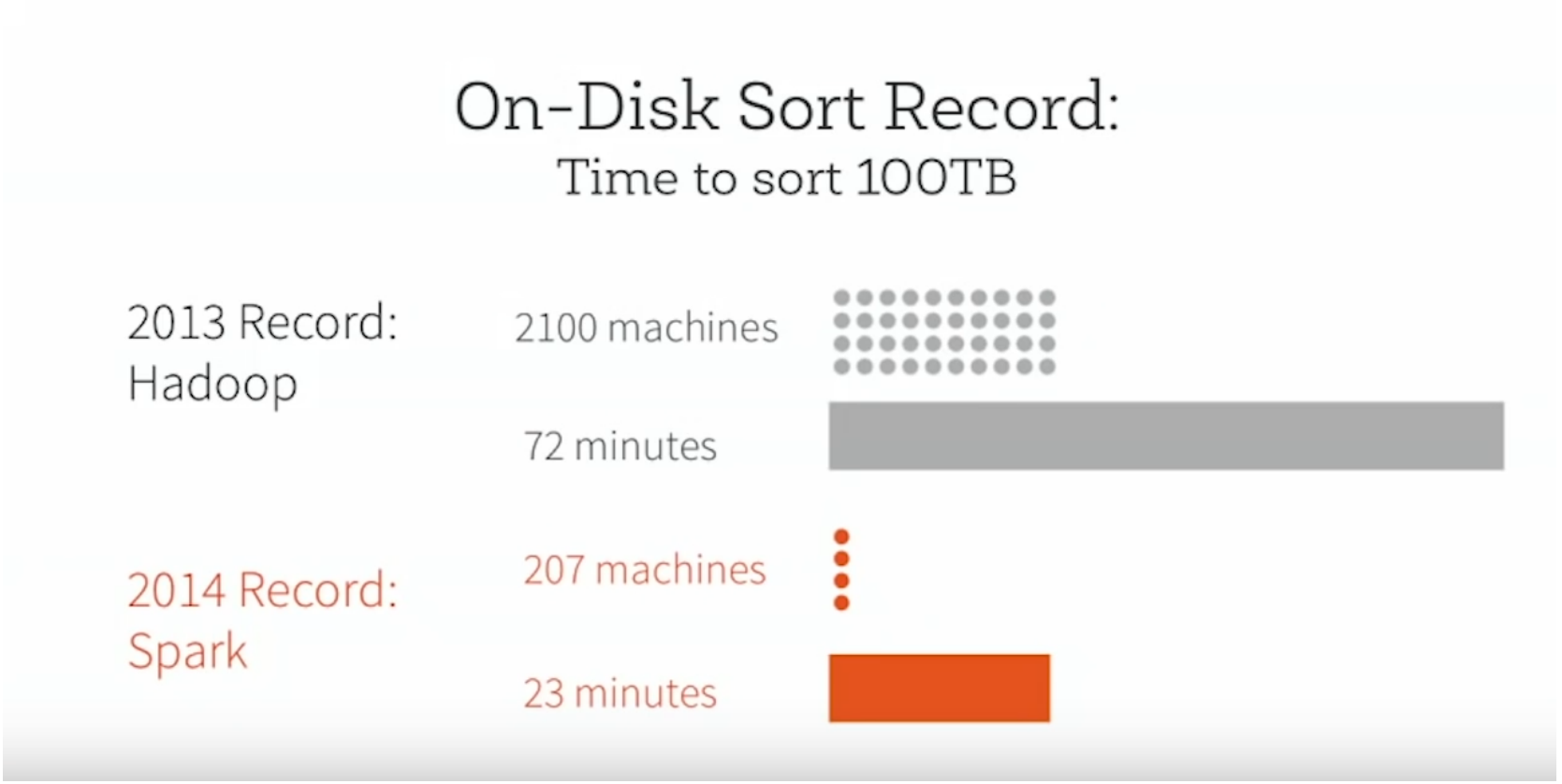
Spark

Word count

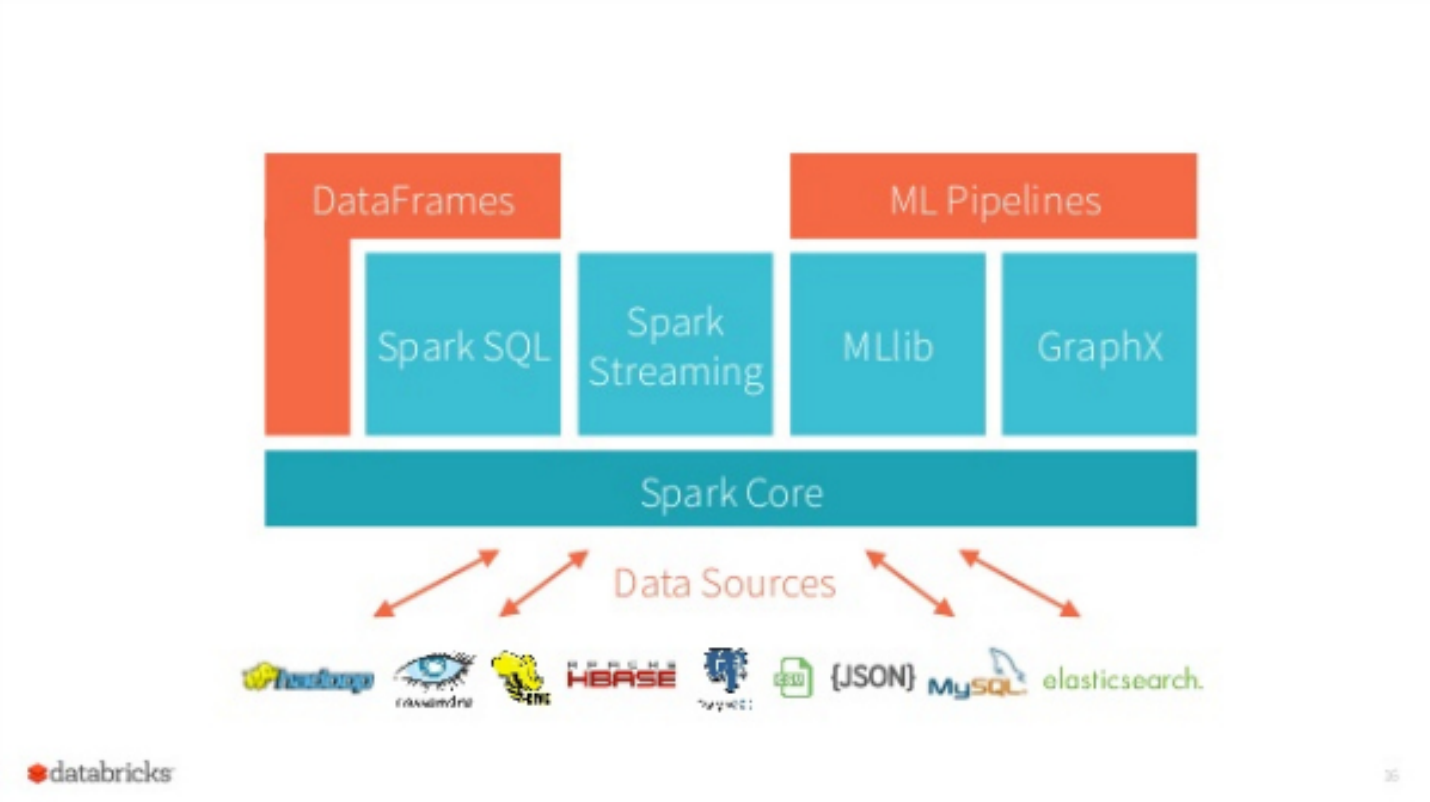
- Mapreduce:
  - https://hadoop.apache.org/docs/r1.2.1/mapred_tutorial.html#Example%3A+WordCount+v2.0

- Spark
  - https://spark.apache.org/examples.html

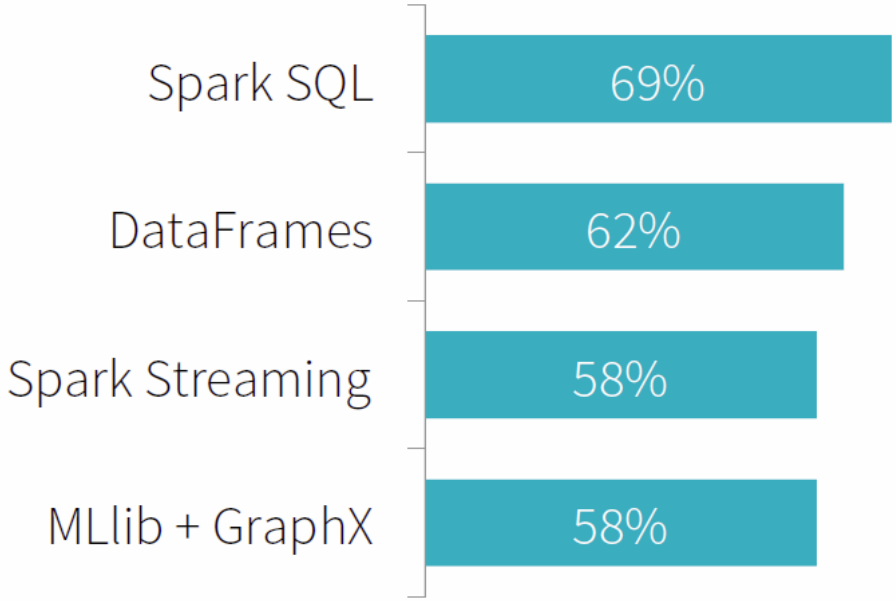
Spark is not just in-memory processing -- it is faster on disk too!



A unified engine



Spark Components Used

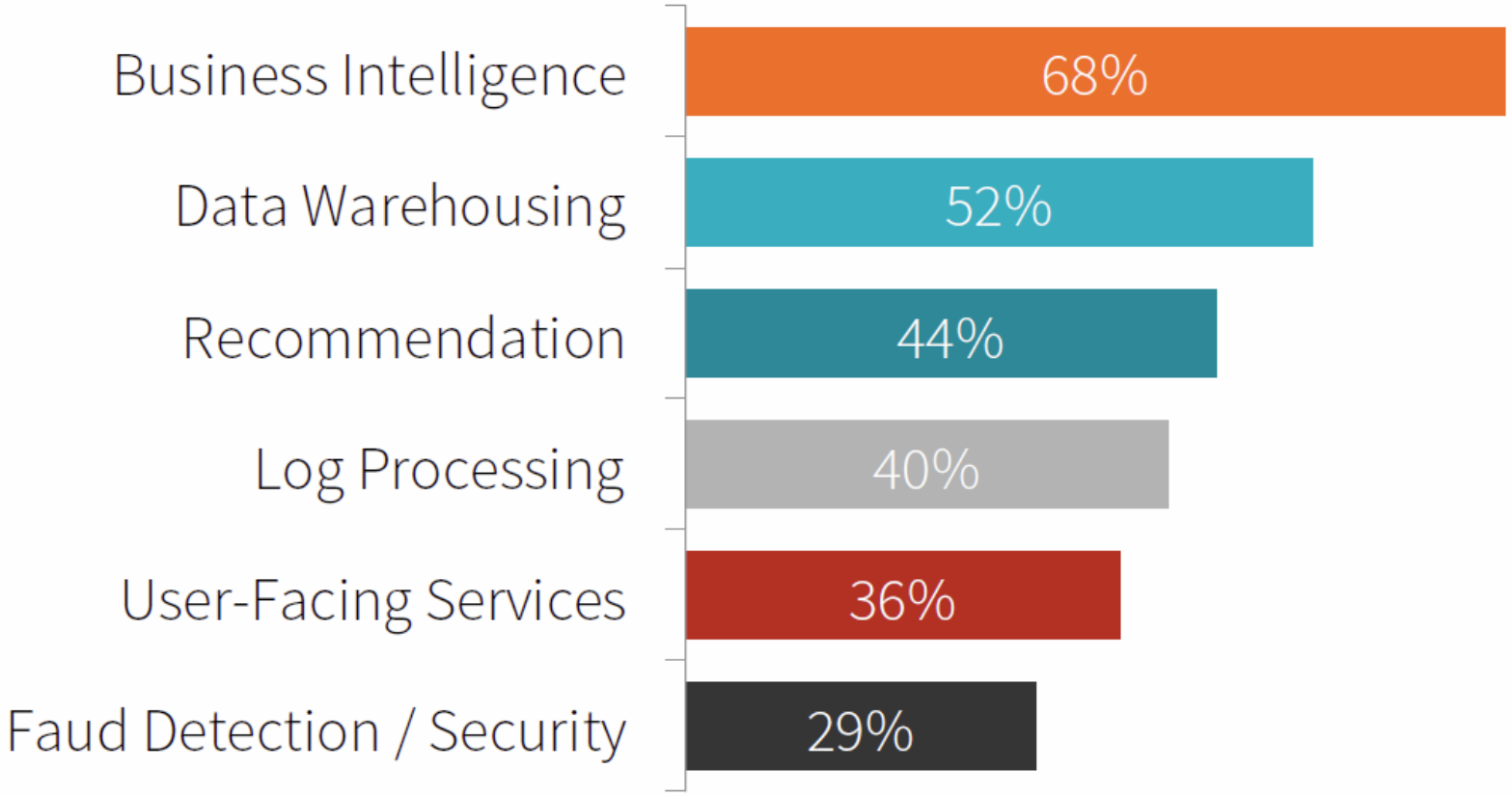


75%

of users use more than one component

databricks

Top Applications



databricks

Core Spark data abstraction

- Resilient Distributed Dataset (RDD)

  - **RDD (Resilient Distributed Dataset)**
    - Resilient – if data in memory is lost, it can be recreated
    - Distributed – processed across the cluster
    - Dataset – initial data can come from a file or be created programmatically
  - **RDDs are the fundamental unit of data in Spark**
  - **Most Spark programming consists of performing operations on RDDs**

RDDs

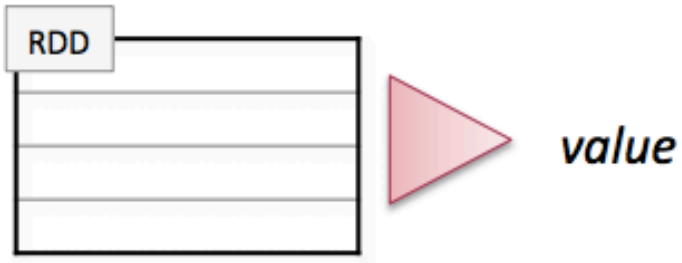
- **RDDs can hold any type of element**
  - Primitive types: integers, characters, booleans, etc.
  - Sequence types: strings, lists, arrays, tuples, dicts, etc. (including nested data types)
  - Scala/Java Objects (if serializable)
  - Mixed types
- **Some types of RDDs have additional functionality**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

Creating RDD

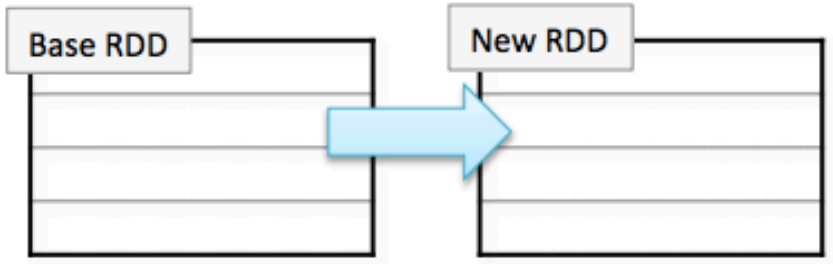
- **Three ways to create an RDD**
  - From a file or set of files
  - From data in memory
  - From another RDD

RDD operations

- **Two types of RDD operations**
  - Actions – return values

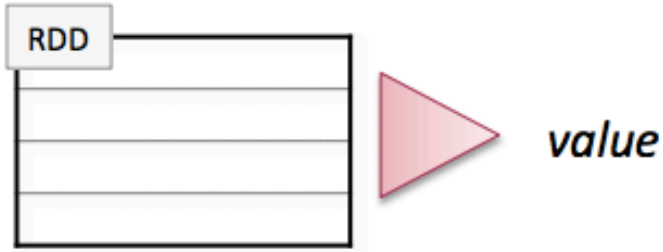


  - Transformations – define a new RDD based on the current one(s)



RDD operations: 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)



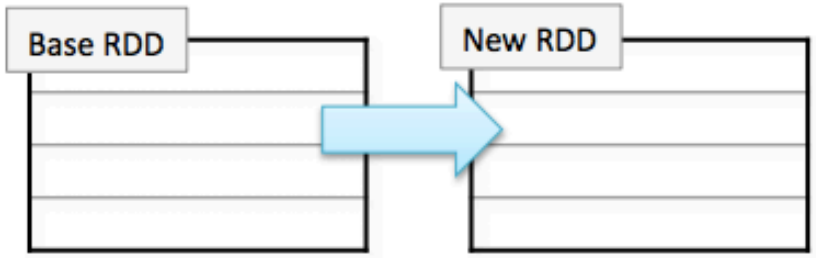

```
> mydata =
  sc.textFile("purplecow.txt")

> 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;
```

RDD operations: Transformation

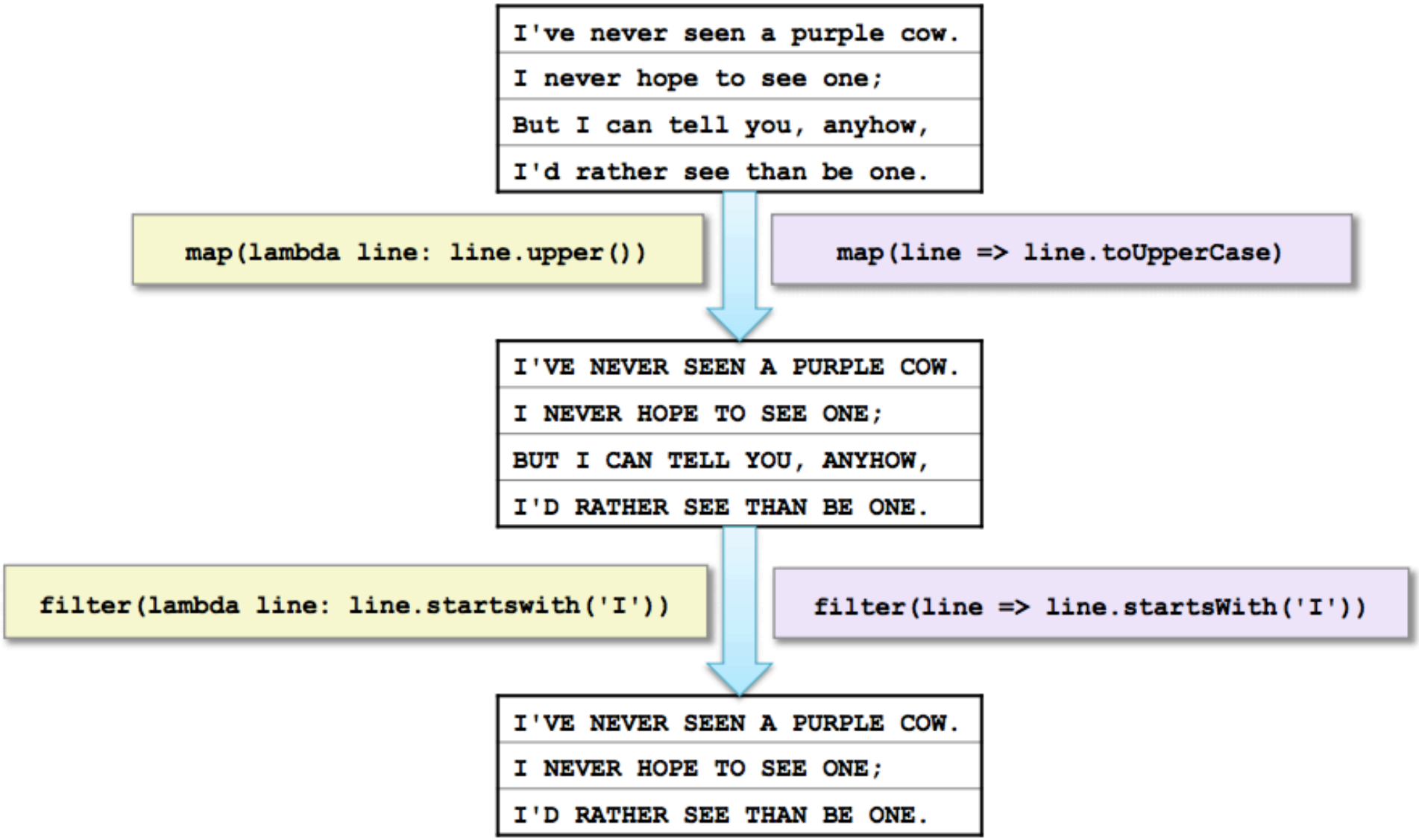
RDD Operations: Transformations

- **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 and filter



Lazy execution

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

Chaining transformations

- **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 and toDebugString

- Spark maintains each RDD's *lineage* – the previous RDDs on which it depends
- Use `toDebugString` to view the lineage of an RDD
- `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 prettier 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 …
```

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

Passing functions as parameters

- **Many RDD operations take functions as parameters**
- **Pseudocode 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)
    }
}
```

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)
```

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 => ...`
  - Java 8: `x -> ...`

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

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']
```

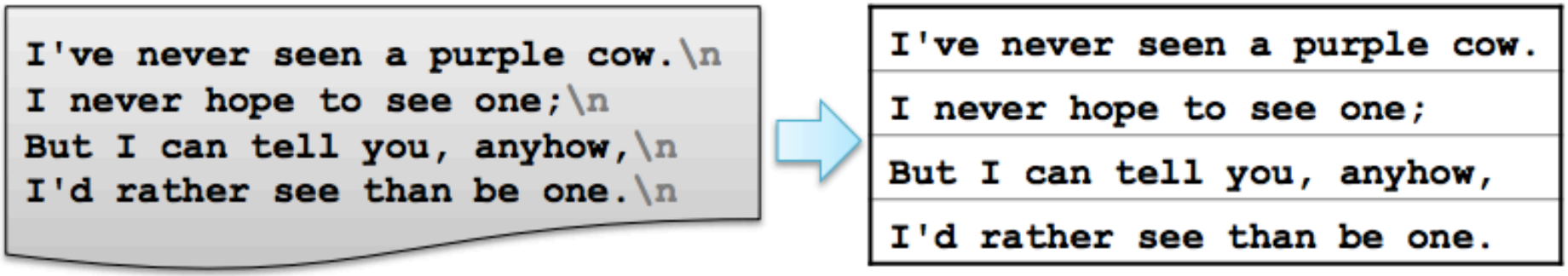
- **Useful when**
  - Testing
  - Generating data programmatically
  - Integrating

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**
- **What about other formats?**

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))
```

Some other 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

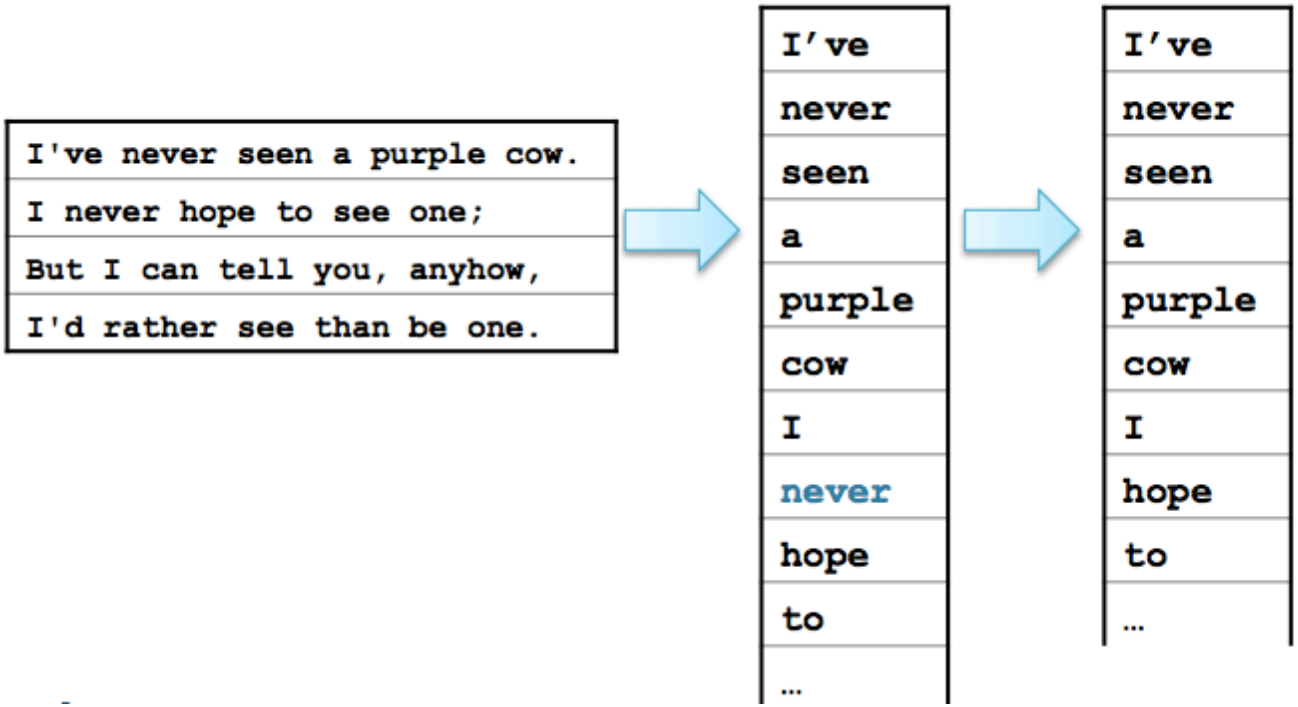
Example: flatMap and distinct

Python

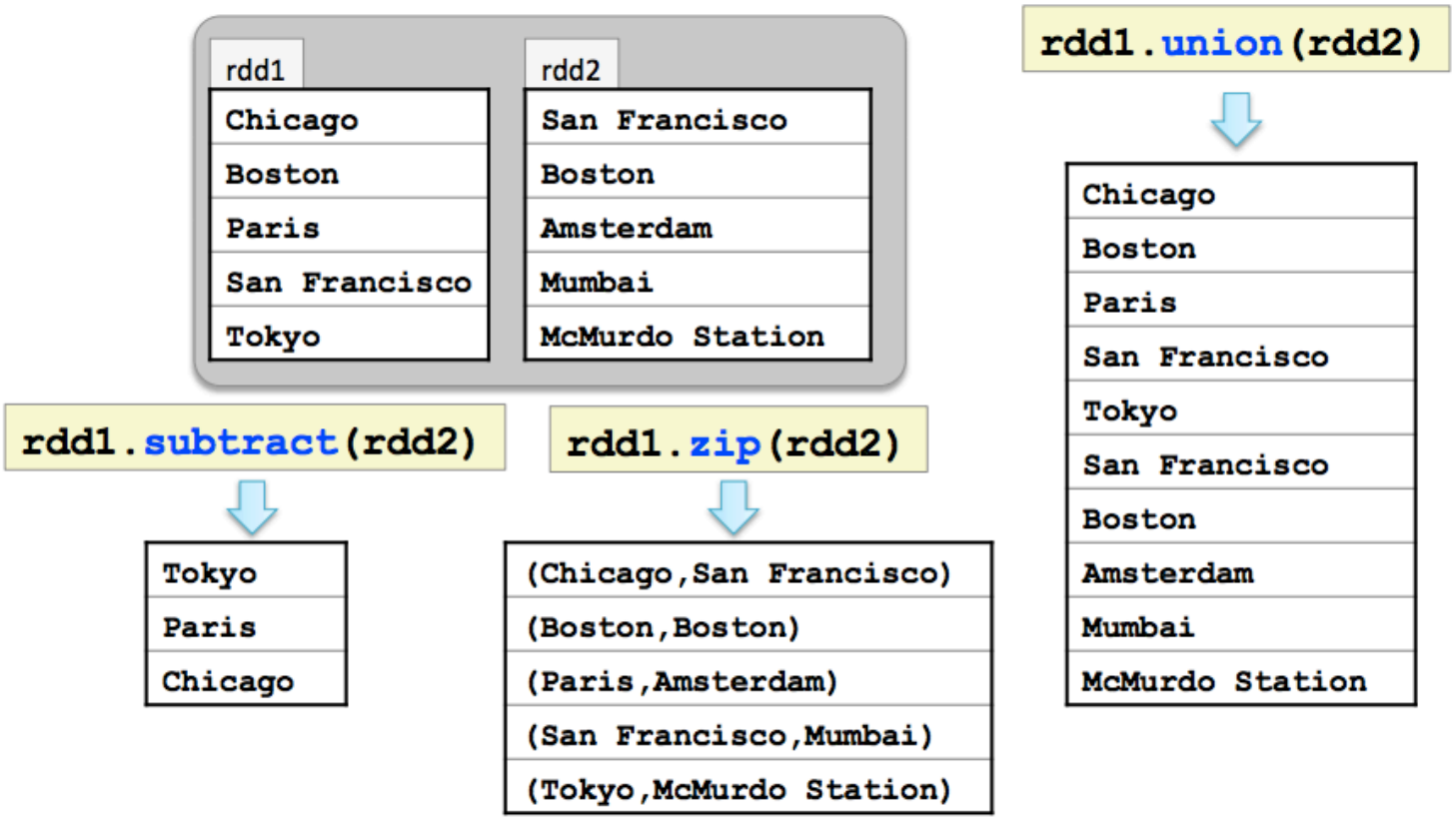
```
> sc.textFile(file) \
  .flatMap(lambda line: line.split()) \
  .distinct()
```

Scala

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



Example: multi-RDD transformations



Some other 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`**

Conclusion

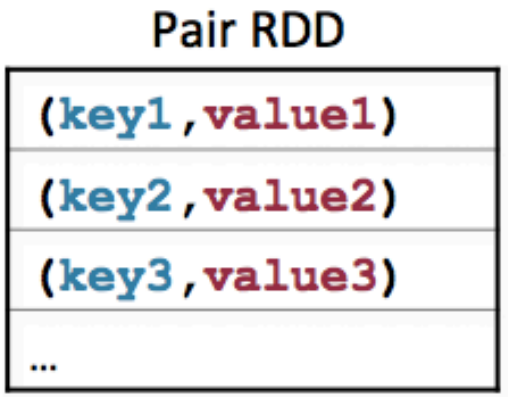
- **RDDs can be created from files, parallelized data in memory, or other RDDs**
- **`sc.textFile` reads newline delimited text, one line per RDD record**
- **`sc.wholeTextFile` reads entire files into single RDD records**
- **Generic RDDs can consist of any type of data**
- **Generic RDDs provide a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

Aggregating data with pair RDDs

- **How to create Pair RDDs of key-value pairs from generic RDDs**
- **Special operations available on Pair RDDs**
- **How map-reduce algorithms are implemented in Spark**

Pair RDDs

- **Pair RDDs are a special form of RDD**
  - Each element must be a key-value pair (a two-element tuple)
  - Keys and values can be any type
- **Why?**
  - Use with map-reduce algorithms
  - Many additional functions are available for common data processing needs
    - e.g., sorting, joining, grouping, counting, etc.



Creating pair RDDs

- **The first step in most workflows is to get the data into key/value form**
  - What should the RDD should be keyed on?
  - What is the value?
- **Commonly used functions to create Pair RDDs**
  - **`map`**
  - **`flatMap` / `flatMapValues`**
  - **`keyBy`**

Example: a simple pair RDDs

- **Example: Create a Pair RDD from a tab-separated file**

Python

```
> users = sc.textFile(file) \
    .map(lambda line: line.split('\t')) \
    .map(lambda fields: (fields[0],fields[1]))
```

Scala

```
> val users = sc.textFile(file) \
    .map(line => line.split('\t')) \
    .map(fields => (fields(0),fields(1)))
```

```
user001\tFred Flintstone
user090\tBugs Bunny
user111\tHarry Potter
…
```

```
(user001,Fred Flintstone)
(user090,Bugs Bunny)
(user111,Harry Potter)
…
```

Example: keying by user ID

Python

```
> sc.textFile(logfile) \
    .keyBy(lambda line: line.split(' ')[2])
```

Scala

```
> sc.textFile(logfile) \
    .keyBy(line => line.split(' ')(2))
```

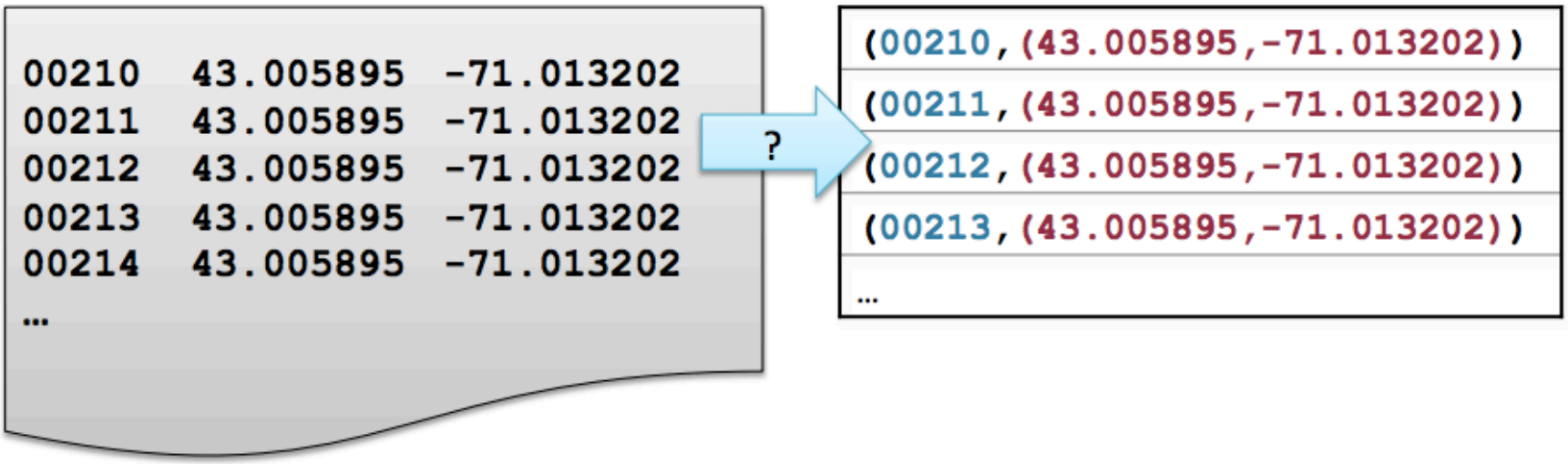
User ID

```
56.38.234.188 - 99788 "GET /KBDOC-00157.html HTTP/1.0" …
56.38.234.188 - 99788 "GET /theme.css HTTP/1.0" …
203.146.17.59 - 25254 "GET /KBDOC-00230.html HTTP/1.0" …
…
```

```
(99788,56.38.234.188 - 99788 "GET /KBDOC-00157.html…)
(99788,56.38.234.188 - 99788 "GET /theme.css…)
(25254,203.146.17.59 - 25254 "GET /KBDOC-00230.html…)
…
```

Question1: pairs with complex values

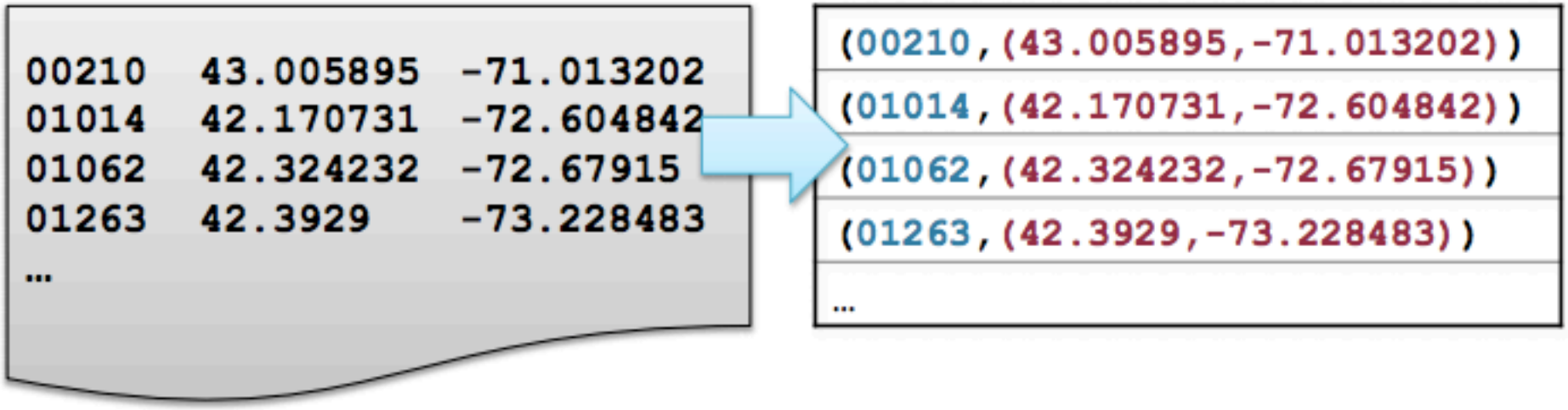
- **How would you do this?**
  - Input: a list of postal codes with latitude and longitude
  - Output: postal code (key) and lat/long pair (value)



Answer1: pairs with complex values

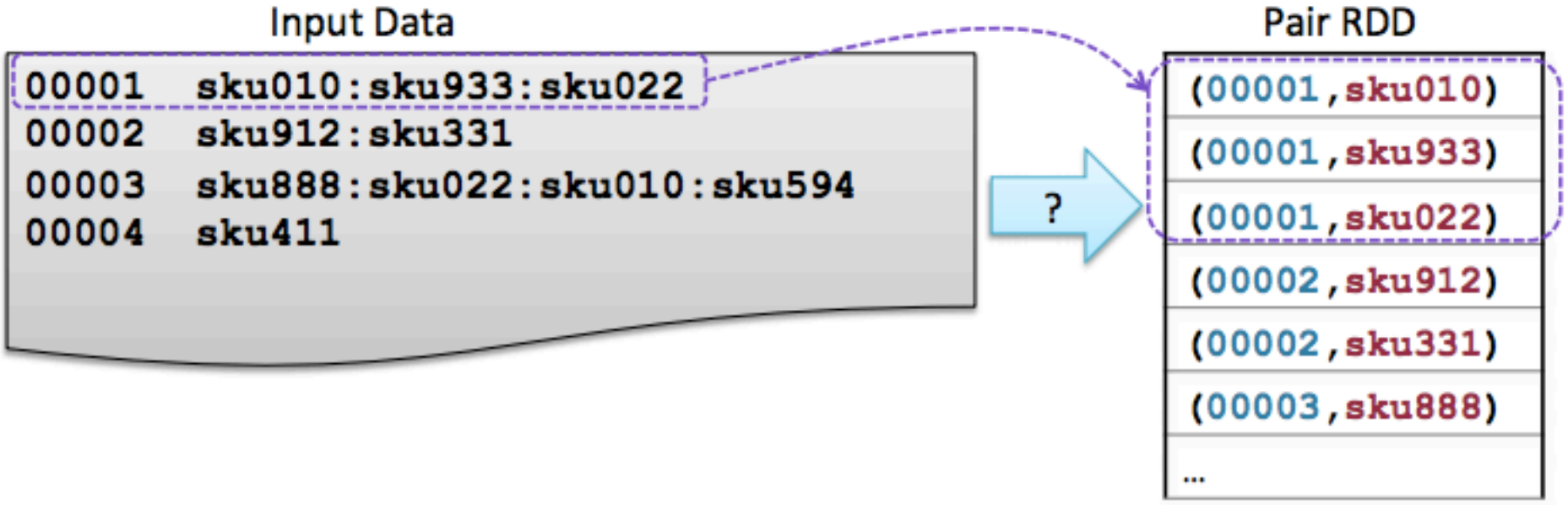
```
> sc.textFile(file) \
  .map(lambda line: line.split()) \
  .map(lambda fields: (fields[0],(fields[1],fields[2])))
```

```
> sc.textFile(file).
   map(line => line.split('\t')).
   map(fields => (fields(0),(fields(1),fields(2))))
```



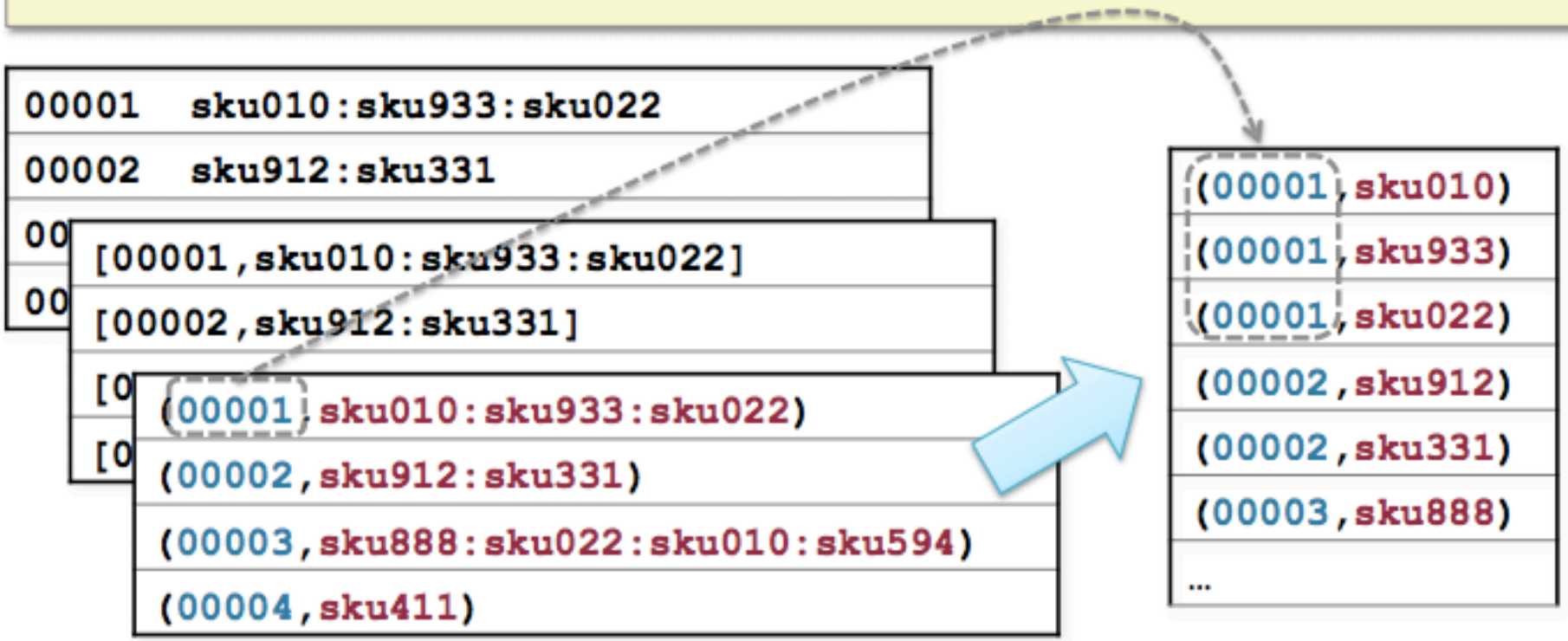
Question2: mapping single rows to multiple pairs

- **How would you do this?**
  - Input: order numbers with a list of SKUs in the order
  - Output: order (key) and sku (value)



Answer2: mapping single rows to multiple pairs

```
> sc.textFile(file) \
  .map(lambda line: line.split('\t')) \
  .map(lambda fields: (fields[0],fields[1]))
  .flatMapValues(lambda skus: skus.split(':'))
```



Map-reduce

- **Map-reduce is a common programming model**
  - Easily applicable to distributed processing of large data sets
- **Hadoop MapReduce is the major implementation**
  - Somewhat limited
    - Each job has one Map phase, one Reduce phase
    - Job output is saved to files
- **Spark implements map-reduce with much greater flexibility**
  - Map and reduce functions can be interspersed
  - Results can be stored in memory
    - Operations can easily be chained

Map-reduce in spark

- **Map-reduce in Spark works on Pair RDDs**
- **Map phase**
  - Operates on one record at a time
  - "Maps" each record to one or more new records
  - e.g. **`map`**, **`flatMap`**, **`filter`**, **`keyBy`**
- **Reduce phase**
  - Works on map output
  - Consolidates multiple records
  - e.g. **`reduceByKey`**, **`sortByKey`**, **`mean`**

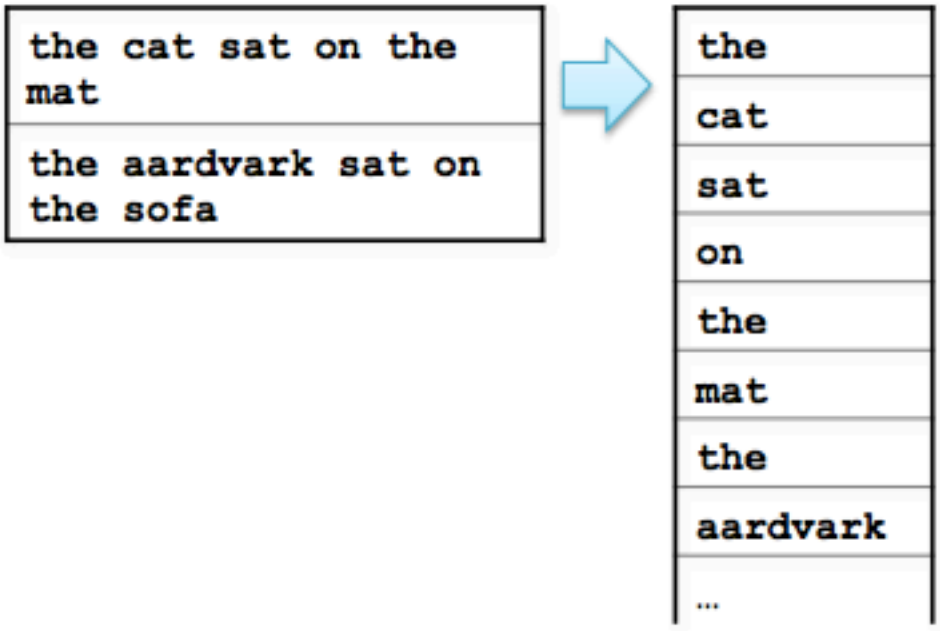
Example: word-count (1)

```
> counts = sc.textFile(file)
```

the cat sat on the mat
the aardvark sat on the sofa

Example: word-count (2)

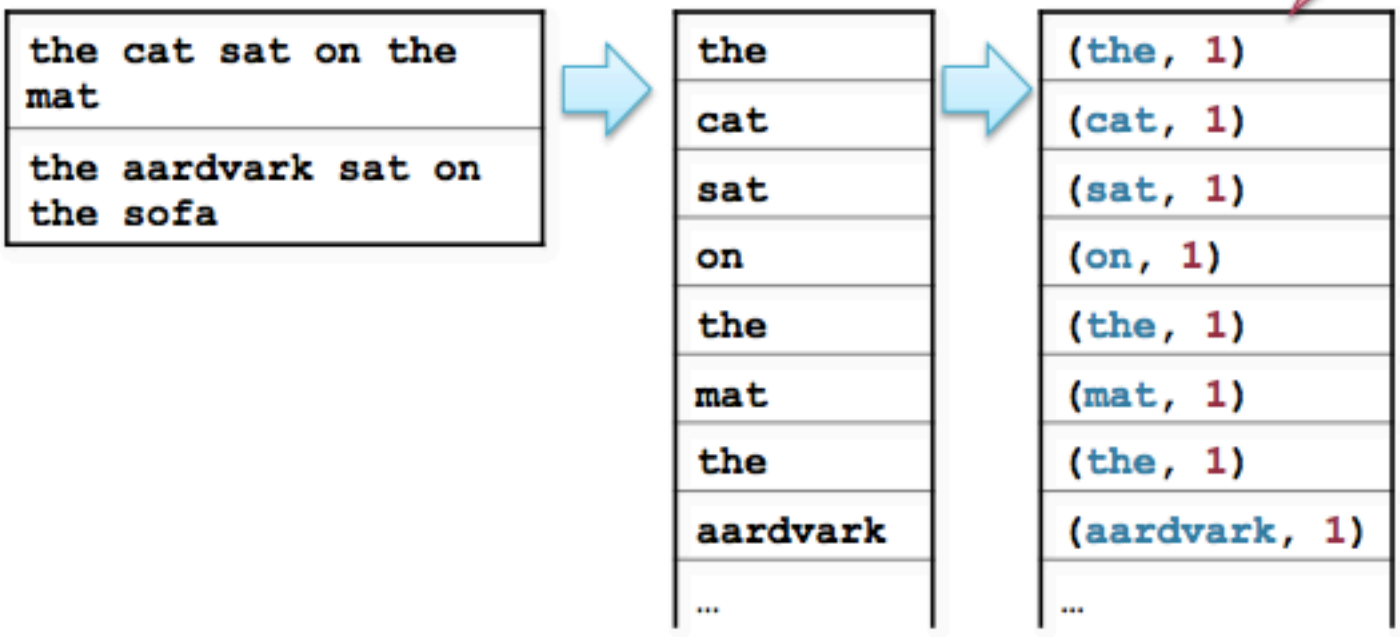
```
> counts = sc.textFile(file) \
    .flatMap(lambda line: line.split())
```



Example: word-count (3)

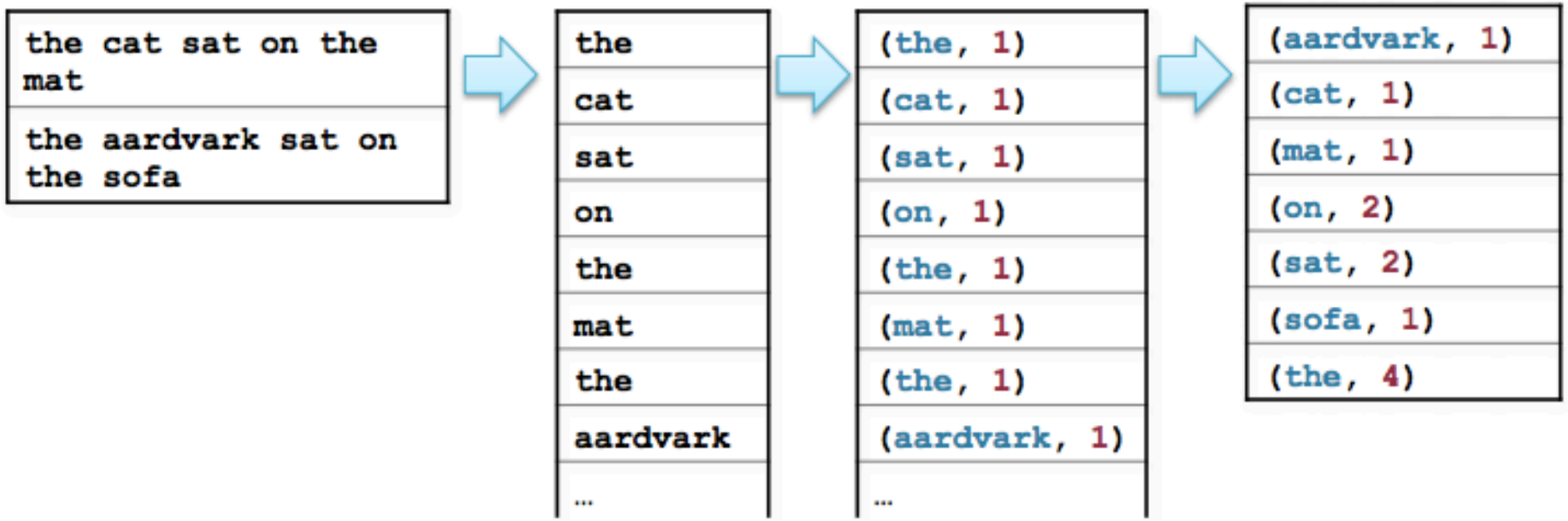
```
> counts = sc.textFile(file) \
    .flatMap(lambda line: line.split()) \
    .map(lambda word: (word,1))
```

Key-Value Pairs



Example: word-count (4)

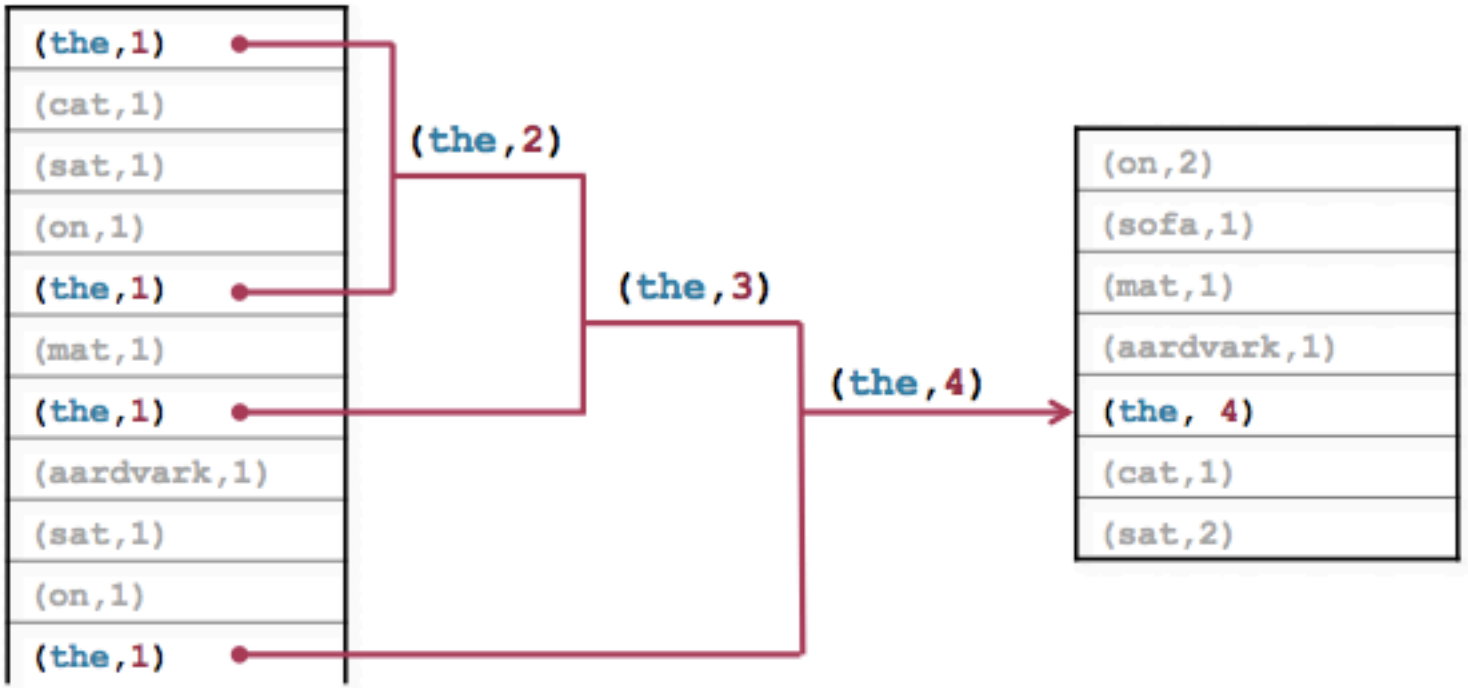
```
> counts = sc.textFile(file) \
    .flatMap(lambda line: line.split()) \
    .map(lambda word: (word,1)) \
    .reduceByKey(lambda v1,v2: v1+v2)
```



ReduceByKey (1)

- **The function passed to `reduceByKey` combines values from two keys**
  - Function must be binary

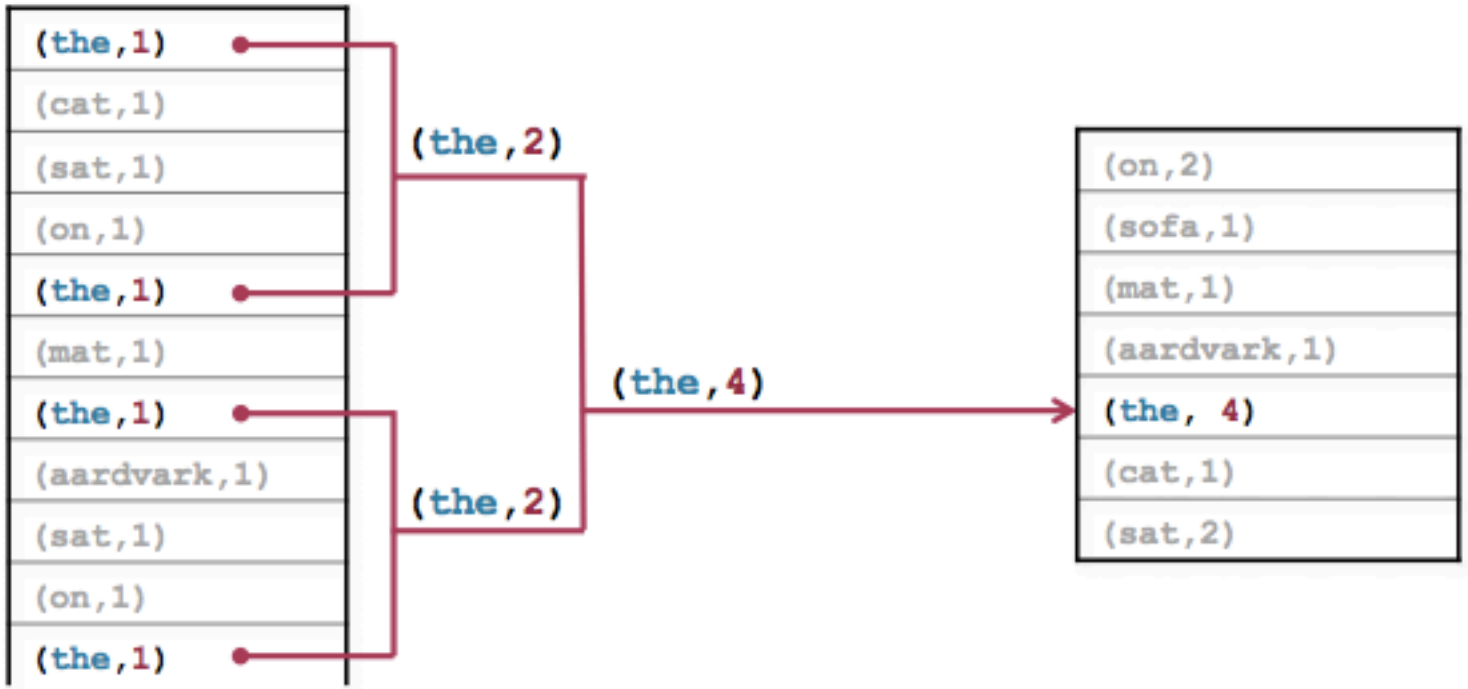
```
> counts = sc.textFile(file) \
  .flatMap(lambda line: line.split()) \
  .map(lambda word: (word,1)) \
  .reduceByKey(lambda v1,v2: v1+v2)
```



ReduceByKey (2)

- **The function might be called in any order, therefore must be**
  - Commutative – x+y = y+x
  - Associative – (x+y)+z = x+(y+z)

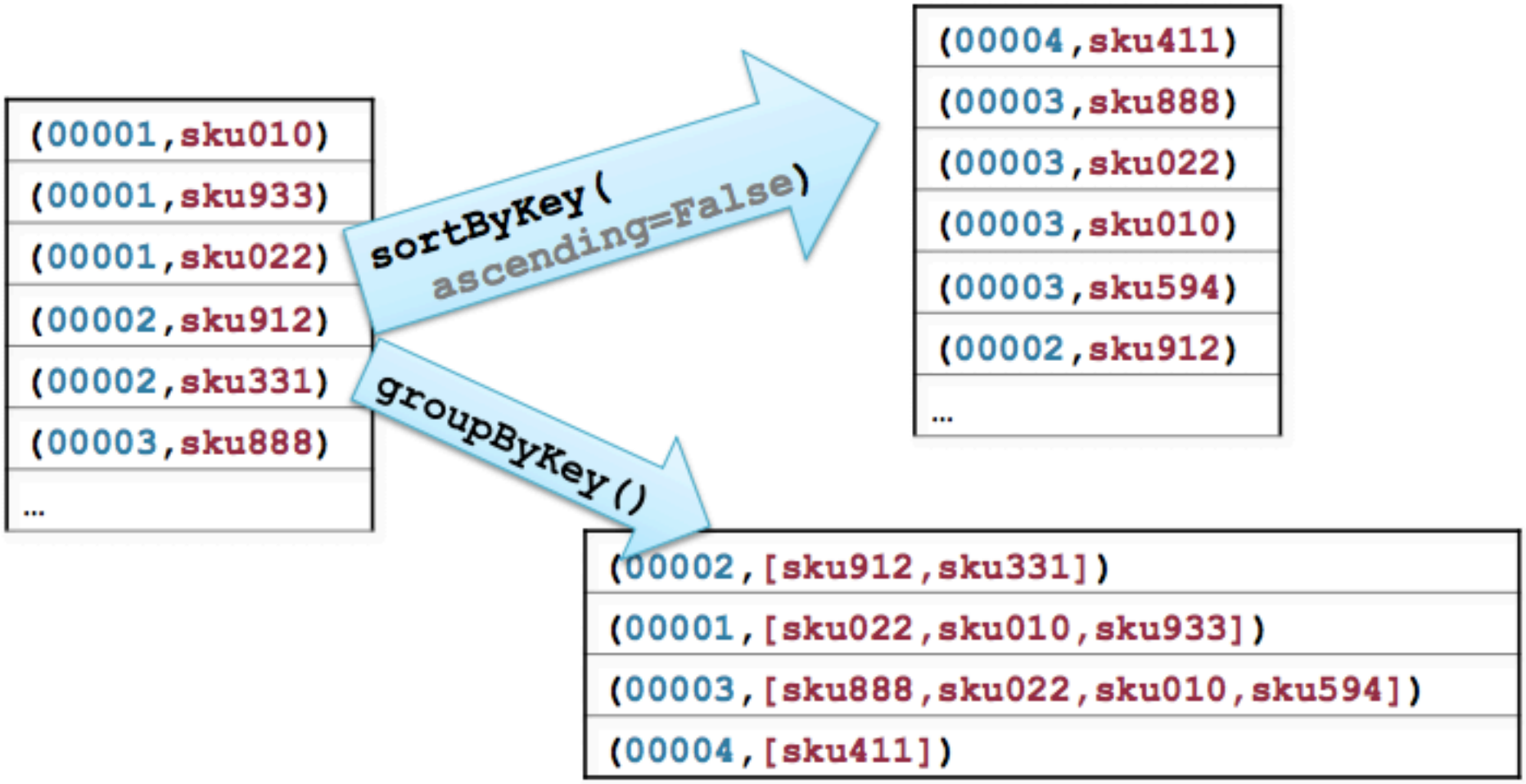
```
> counts = sc.textFile(file) \
  .flatMap(lambda line: line.split()) \
  .map(lambda word: (word,1)) \
  .reduceByKey(lambda v1,v2: v1+v2)
```



Other pair RDD operations

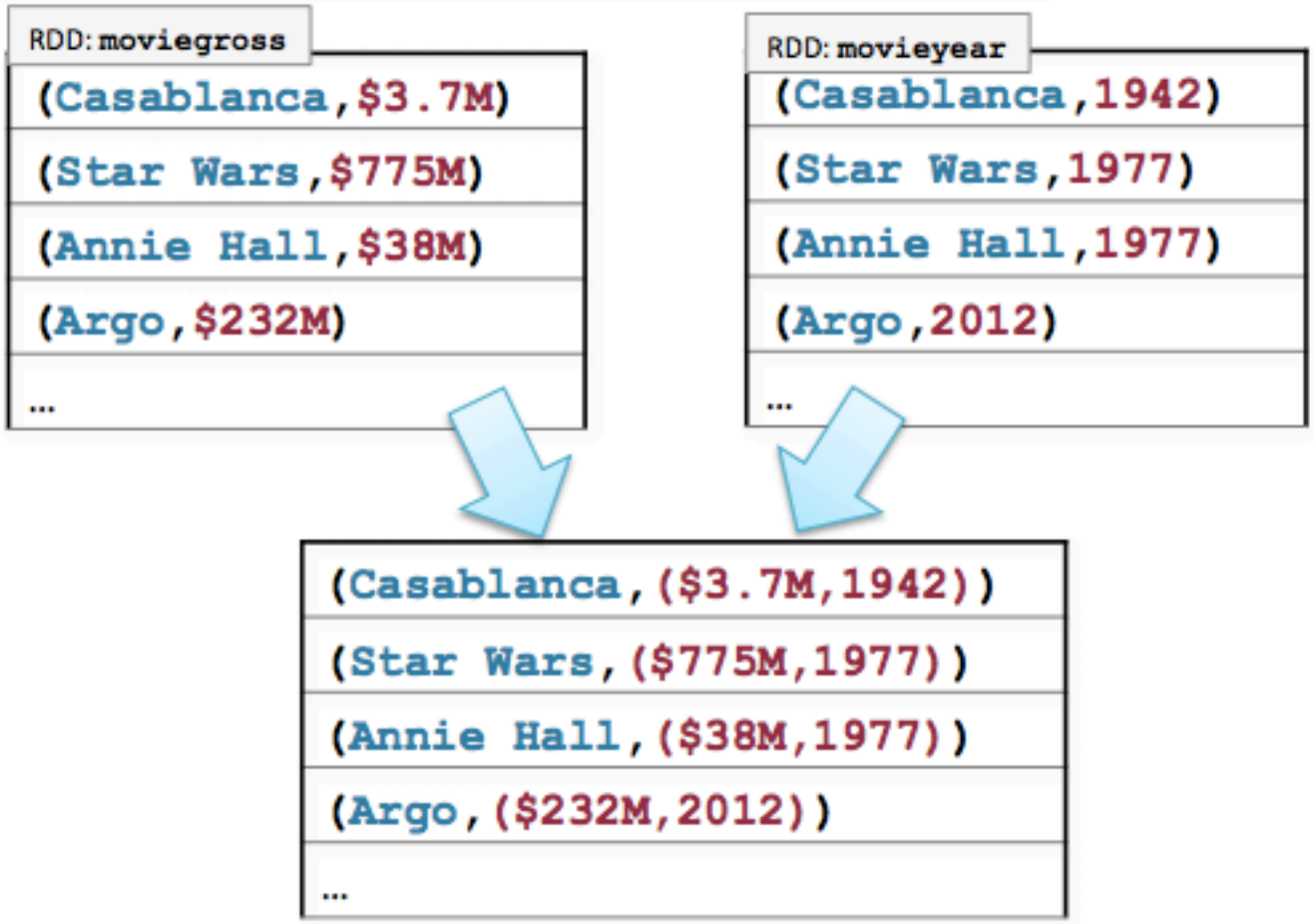
- **In addition to `map` and `reduce` functions, Spark has several operations specific to Pair RDDs**
- **Examples**
  - **`countByKey`** – return a map with the count of occurrences of each key
  - **`groupByKey`** – group all the values for each key in an RDD
  - **`sortByKey`** – sort in ascending or descending order
  - **`join`** – return an RDD containing all pairs with matching keys from two RDDs

Example: pair RDD operations



Example: joining by key

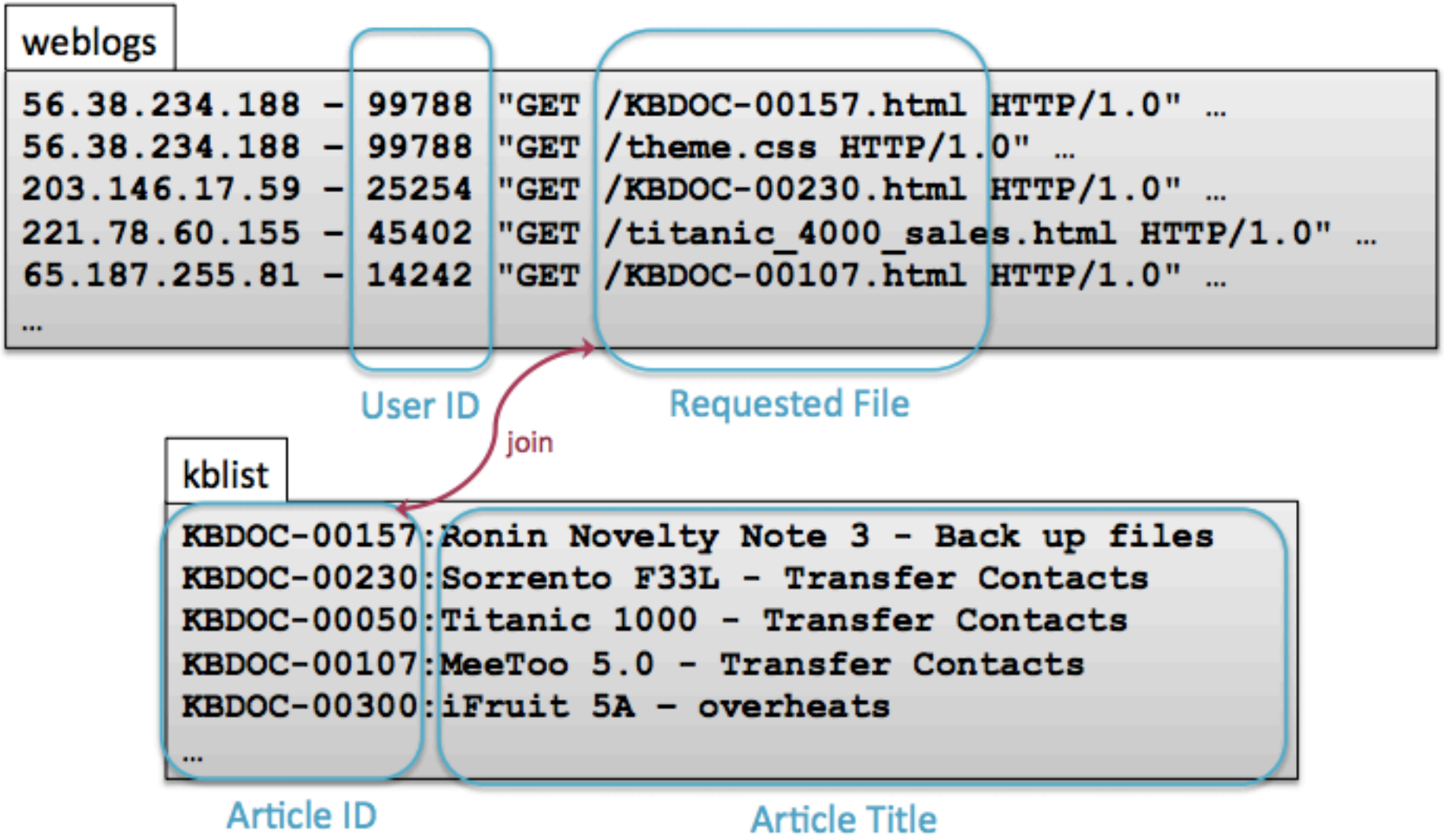
```
> movies = moviegross.join(movieyear)
```



Using join

- **A common programming pattern**
  1. Map separate datasets into key-value Pair RDDs
  2. Join by key
  3. Map joined data into the desired format
  4. Save, display, or continue processing…

Example: join web log with knowledge base article



Example: join web log with knowledge base article

- **Steps**
  1. Map separate datasets into key-value Pair RDDs
     a. Map web log requests to `(docid,userid)`
     b. Map KB Doc index to `(docid,title)`
  2. Join by key: `docid`
  3. Map joined data into the desired format: `(userid,title)`
  4. Further processing: group titles by User ID

Step 1a: Map Web Log Requests to (docid,userid)

```
> import re
> def getRequestDoc(s):
    return re.search(r'KBDOC-[0-9]*',s).group()

> kbreqs = sc.textFile(logfile) \
    .filter(lambda line: 'KBDOC-' in line) \
    .map(lambda line: (getRequestDoc(line),line.split(' ')[2])) \
    .distinct()
```

```
56.38.234.188 - 99788 "GET /KBDOC-00157.html HTTP/1.0" …
56.38.234.188 - 99788 "GET /theme.css HTTP/1.0" …
203.146.17.59 - 25254 "GET /KBDOC-00230.html HTTP/1.0"
221.78.60.155 - 45402 "GET /titanic_4000_sales.html
65.187.255.81 - 14242 "GET /KBDOC-00107.html HTTP/1.
…
```

kbreqs …

(KBDOC-00157,99788)
(KBDOC-00203,25254)
(KBDOC-00107,14242)
…

Step 1b: Map KB Index to (docid, title)

```
> kblist = sc.textFile(kblistfile) \
   .map(lambda line: line.split(':')) \
   .map(lambda fields: (fields[0],fields[1]))
```

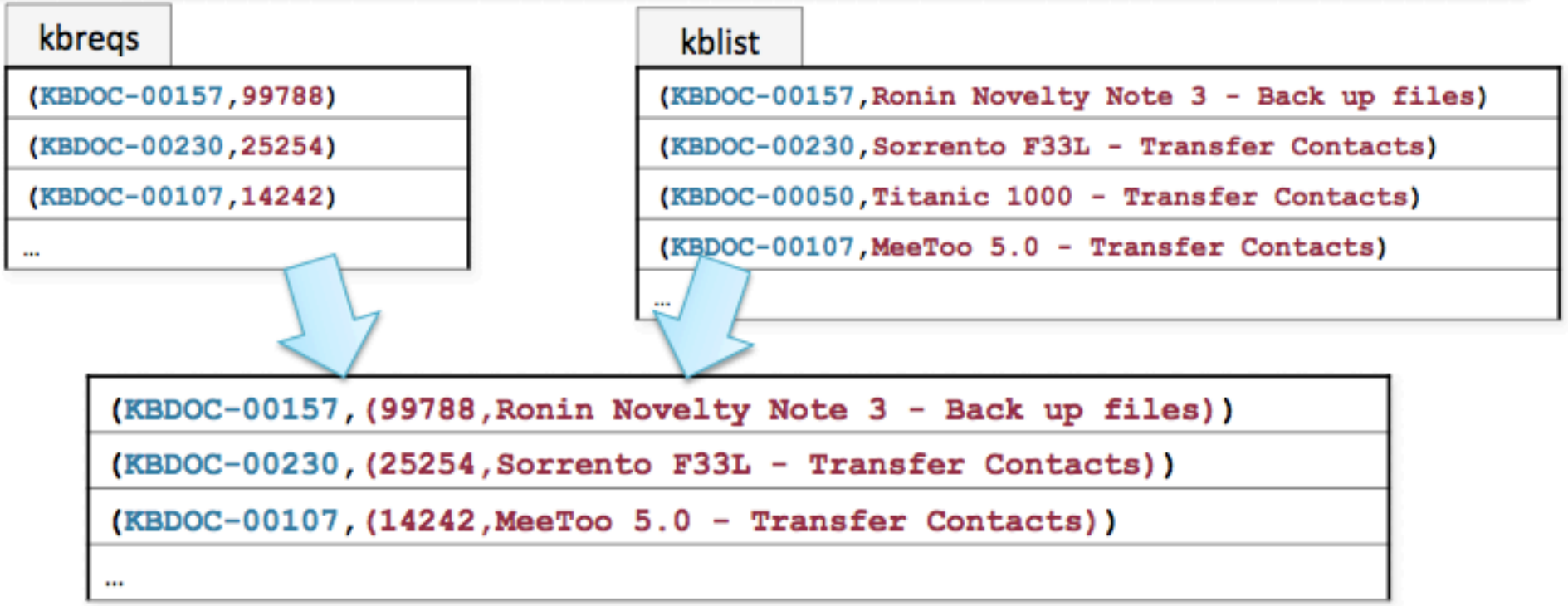
```
KBDOC-00157:Ronin Novelty Note 3 - Back up files
KBDOC-00230:Sorrento F33L - Transfer Contacts
KBDOC-00050:Titanic 1000 - Transfer Contacts
KBDOC-00107:MeeToo 5.0 - Transfer Contacts
KBDOC-00206:iFruit 5A - overheats
…
```

kblist

```
(KBDOC-00157,Ronin Novelty Note 3 - Back up files)
(KBDOC-00230,Sorrento F33L - Transfer Contacts)
(KBDOC-00050,Titanic 1000 - Transfer Contacts)
(KBDOC-00107,MeeToo 5.0 - Transfer Contacts)
…
```

Step 2: Join By Key **docid**

```
> titlereqs = kbreqs.join(kblist)
```



Step 3: Map Result to Desired Format (userid,title)

```
> titlereqs = kbreqs.join(kblist) \
    .map(lambda (docid,(userid,title)): (userid,title))
```

(KBDOC-00157,(99788,Ronin Novelty Note 3 - Back up files))
(KBDOC-00230,(25254,Sorrento F33L - Transfer Contacts))
(KBDOC-00107,(14242,MeeToo 5.0 - Transfer Contacts))
…

(99788,Ronin Novelty Note 3 - Back up files)
(25254,Sorrento F33L - Transfer Contacts)
(14242,MeeToo 5.0 - Transfer Contacts)
…

Step 4: Continue Processing – Group Titles by User ID

```
> titlereqs = kbreqs.join(kblist) \
    .map(lambda (docid,(userid,title)): (userid,title)) \
    .groupByKey()
```

(99788,Ronin Novelty Note 3 - Back up files)
(25254,Sorrento F33L - Transfer Contacts)
(14242,MeeToo 5.0 - Transfer Contacts)
…

(99788,[Ronin Novelty Note 3 - Back up files, Ronin S3 - overheating])
(25254,[Sorrento F33L - Transfer Contacts])
(14242,[MeeToo 5.0 - Transfer Contacts, MeeToo 5.1 - Back up files, iFruit 1 - Back up files, MeeToo 3.1 - Transfer Contacts])
…

Note: values are grouped into Iterables

Example output

```
> for (userid,titles) in titlereqs.take(10):
    print 'user id: ',userid
    for title in titles: print '\t',title

user id: 99788
  Ronin Novelty Note 3 - Back up files
  Ronin S3 - overheating
user id: 25254
  Sorrento F33L - Transfer Cont
user id: 14242
  MeeToo 5.0 - Transfer Contact
  MeeToo 5.1 - Back up files
  iFruit 1 - Back up files
  MeeToo 3.1 - Transfer Contacts
...
```

```
(99788,[Ronin Novelty Note 3 - Back up files,
        Ronin S3 - overheating])
(25254,[Sorrento F33L - Transfer Contacts])
(14242,[MeeToo 5.0 - Transfer Contacts,
        MeeToo 5.1 - Back up files,
        iFruit 1 - Back up files,
        MeeToo 3.1 - Transfer Contacts])
...
```

Other pair operations

- **Some other pair operations**
  - **`keys`** – return an RDD of just the keys, without the values
  - **`values`** – return an RDD of just the values, without keys
  - **`lookup(key)`** – return the value(s) for a key
  - **`leftOuterJoin`**, **`rightOuterJoin`**, **`fullOuterJoin`** – join, including keys defined in the left, right or either RDD respectively
  - **`mapValues`**, **`flatMapValues`** – execute a function on just the values, keeping the key the same
- **See the `PairRDDFunctions` class Scaladoc for a full list**

Writing and deploying spark applications

The SparkContext

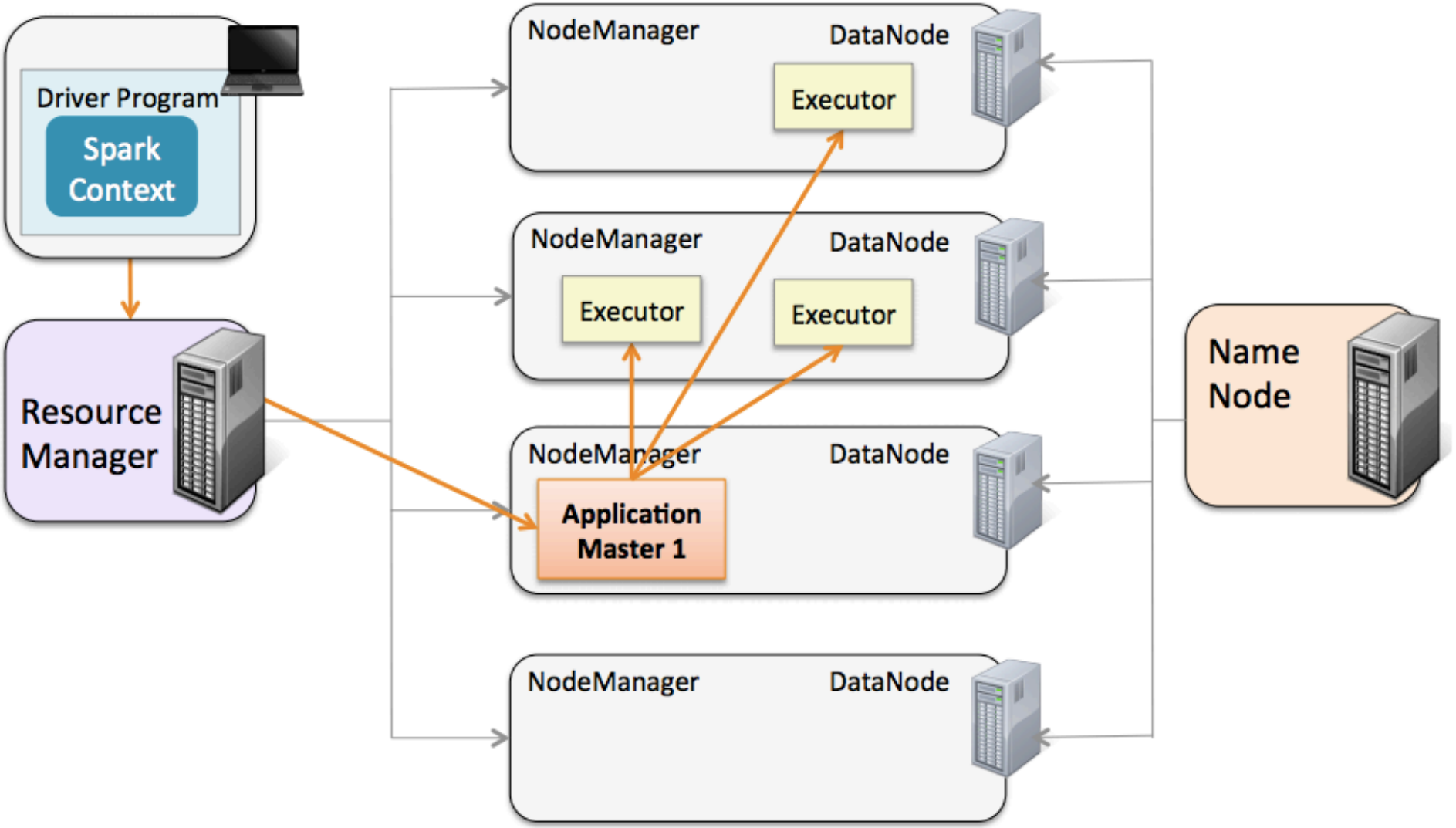
- **Every Spark program needs a SparkContext**
  - The interactive shell creates one for you
- **In your own Spark application you create your own SparkContext**
  - Named **`sc`** by convention
  - Call **`sc.stop`** when program terminates

Python example: word-count

```
import sys
from pyspark import SparkContext

if __name__ == "__main__":
  if len(sys.argv) < 2:
     print >> sys.stderr, "Usage: WordCount <file>"
     exit(-1)

  sc = SparkContext()

  counts = sc.textFile(sys.argv[1]) \
    .flatMap(lambda line: line.split()) \
    .map(lambda word: (word,1)) \
    .reduceByKey(lambda v1,v2: v1+v2)

  for pair in counts.take(5): print pair

  sc.stop()
```

Building a spark application

- **Scala or Java Spark applications must be compiled and assembled into JAR files**
  - JAR file will be passed to worker nodes
- **Apache Maven is a popular build tool**
  - For specific setting recommendations, see `http://spark.apache.org/docs/latest/building-with-maven.html`
- **Build details will differ depending on**
  - Version of Hadoop (HDFS)
  - Deployment platform (Spark Standalone, YARN, Mesos)
- **Consider using an IDE**
  - IntelliJ or Eclipse are two popular examples
  - Can run Spark locally in a debugger


Driver Program
Spark Context
Resource Manager
NodeManager
DataNode
Executor
NodeManager
DataNode
Executor
Executor
NodeManager
DataNode
Application Master 1
NodeManager
DataNode
Name Node

Running a spark application

- **The easiest way to run a Spark Application is using the `spark-submit` script**

Python

```
$ spark-submit WordCount.py fileURL
```

Scala

Java

```
$ spark-submit --class WordCount \
  MyJarFile.jar fileURL
```

Running spark applications locally

- **Use `spark-submit --master` to specify cluster option**
  - Local options
    - **`local[*]`** – run locally with as many threads as cores (default)
    - **`local[n]`** – run locally with n threads
    - **`local`** – run locally with a single thread

Python

```
$ spark-submit --master local[3] \
  WordCount.py fileURL
```

Scala
Java

```
$ spark-submit --master local[3] --class \
  WordCount MyJarFile.jar fileURL
```

Running spark applications on cluster

- **Use `spark-submit --master` to specify cluster option**
  - Cluster options
    - **`yarn-client`**
    - **`yarn-cluster`**
    - **`spark://masternode:port`** (Spark Standalone)
    - **`mesos://masternode:port`** (Mesos)

Python

```
$ spark-submit --master yarn-cluster \
  WordCount.py fileURL
```

Scala
Java

```
$ spark-submit --master yarn-cluster --class \
  WordCount MyJarFile.jar fileURL
```

Starting shell locally or on cluster

- **The Spark Shell can also be run on a cluster**
- **Pyspark and spark-shell both have a `--master` option**
  - **`yarn`** (client mode only)
  - Spark or Mesos cluster manager URL
  - **`local[*]`** – run with as many threads as cores (default)
  - **`local[n]`** – run locally with *n* worker threads
  - **`local`** – run locally without distributed processing

Python

```
$ pyspark --master yarn
```

Scala

```
$ spark-shell --master yarn
```

Options when Submitting a Spark Application to a Cluster

- **Some other `spark-submit` options for clusters**
  - **`--jars`** – additional JAR files (Scala and Java only)
  - **`--py-files`** – additional Python files (Python only)
  - **`--driver-java-options`** – parameters to pass to the driver JVM
  - **`--executor-memory`** – memory per executor (e.g. 1000M, 2G) (Default: 1G)
  - **`--packages`** -- Maven coordinates of an external library to include
- **Plus several YARN-specific options**
  - **`--num-executors`**
  - **`--queue`**
- **Show all available options**
  - **`--help`**