

# Apache Spark

DSCI 551

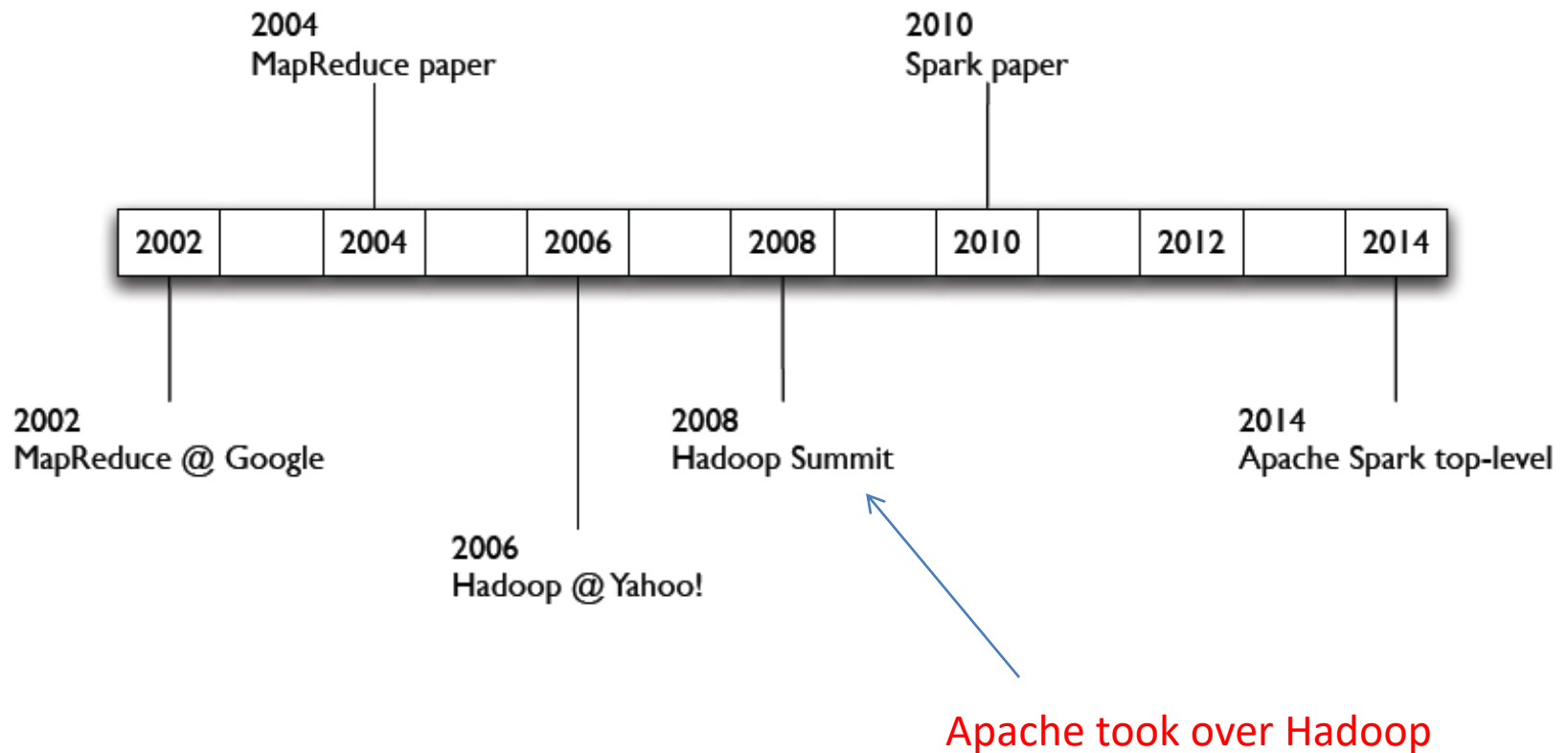
Wensheng Wu

# Roadmap

- Spark
  - History, features, RDD, and installation
- RDD operations
  - Creating initial RDDs
  - Actions
  - Transformations
- Examples
- Shuffling in Spark
- Persistence in Spark



# History



# Characteristics of Hadoop

- Acyclic data flow model
  - Data loaded from stable storage (e.g., HDFS)
  - Processed through a sequence of steps
  - Results written to disk
- Batch processing
  - No interactions permitted during processing

# Problems

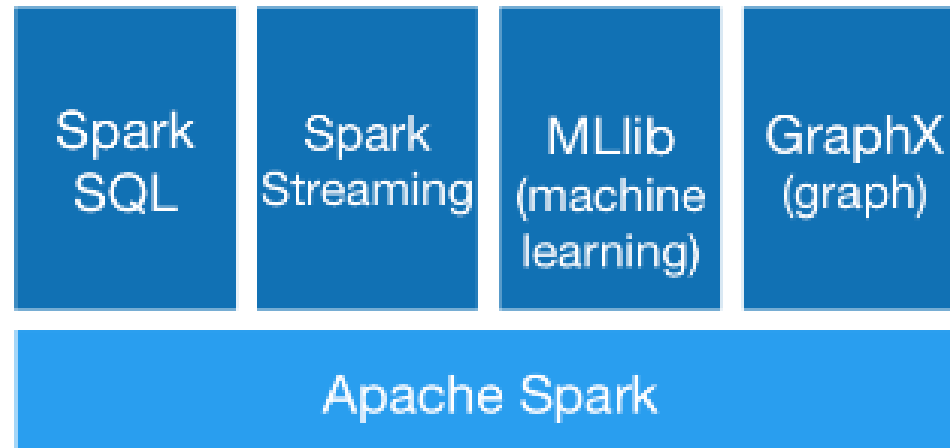
- Ill-suited for iterative algorithms that requires repeated reuse of data
  - E.g., machine learning and data mining algorithms such as k-means (clustering), PageRank, logistic regression
- Ill-suited for interactive exploration of data
  - E.g., OLAP on big data

# Spark

- Support working sets (of data) through RDD
  - Enabling reuse & fault-tolerance
- 10x faster than Hadoop in iterative jobs
- Interactively explore 39GB (Wikipedia dump) with sub-second response time
  - Data were distributed over 15 EC2 instances

# Spark

- Provides libraries to support
  - embedded use of SQL
  - stream data processing
  - machine learning algorithms
  - processing of graph data



# Spark

- Support diverse data sources including HDFS, Cassandra, HBase, and Amazon S3





# RDD: Resilient Distributed Dataset

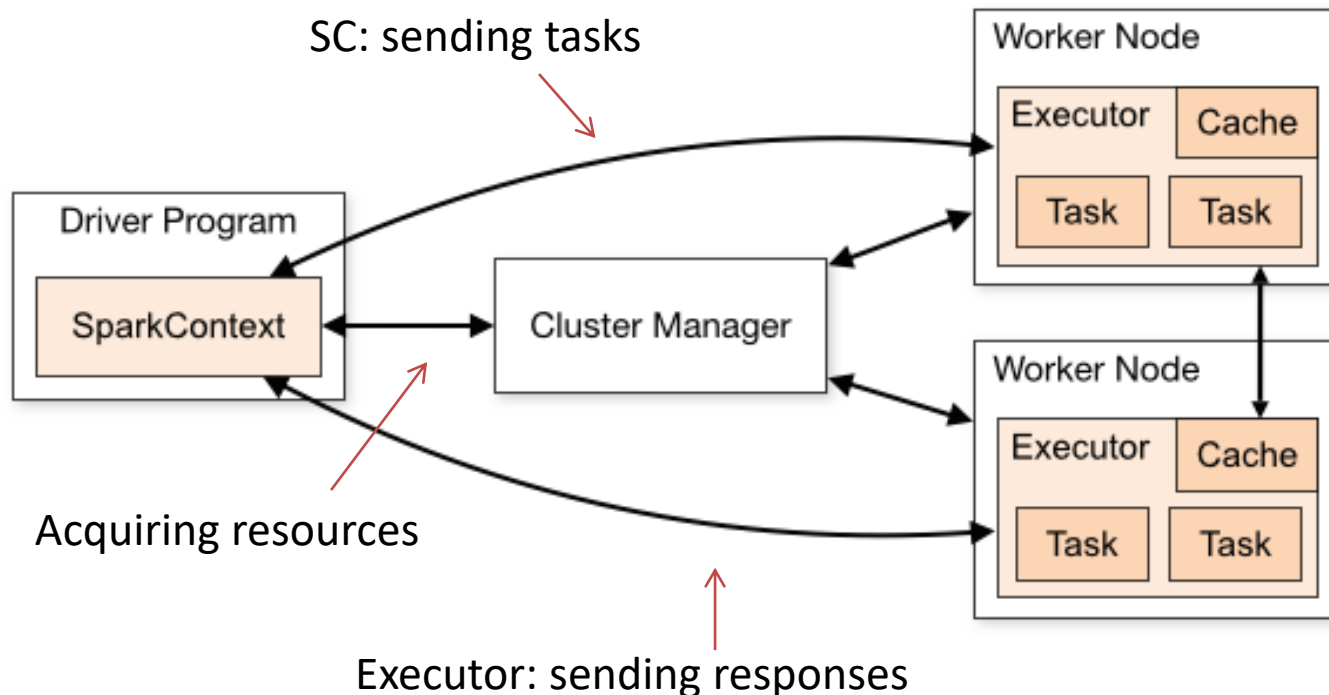
- RDD
  - Read-only, partitioned collection of records
  - Operations performed on partitions in parallel
  - Maintain lineage for efficient fault-tolerance
- Methods of creating an RDD
  - from an existing collection (e.g., Python list/tuple)
  - from an external file

# RDD: Resilient Distributed Dataset

- Distributed
  - Data are divided into a number of partitions
  - & distributed across nodes of a cluster to be processed in parallel
- Resilient
  - Spark keeps track of transformations to dataset
  - Enable **efficient** recovery on failure (no need to replicate large amount of data across network)

# Architecture

- SparkContext (SC) object coordinates the execution of application in multiple nodes
  - Similar to Job Tracker in Hadoop MapReduce



# Components

- Cluster manager
  - Allocate resources across applications
  - Can run Spark's own cluster manager or
  - Apache YARN (Yet Another Resource Negotiator)
- Executors
  - Run tasks & store data

# Accessing Spark from Python

- Interactive shell:
  - pyspark
  - A SparkContext object sc will be automatically created
- `pyspark --master local[4]`
  - This starts Spark on local host with 4 threads
  - "--master" specifies the location of Spark master node

# Accessing Spark from Python

- Standalone program
  - Executed using spark-submit script
  - E.g., spark-submit wc.py
- You may find many Python Spark examples under
  - `examples/src/main/python`


# wc.py

```
from pyspark import SparkContext
from operator import add
```

```
sc = SparkContext(appName="dsci351")
```

```
lines = sc.textFile('hello.txt')
```

Make sure you have this file  
under the same directory  
where wc.py is located



```
counts = lines.flatMap(lambda x: x.split(' ')) \
    .map(lambda x: (x, 1)) \
    .reduceByKey(add)
```

```
output = counts.collect()
```

```
for v in output:
    print(v[0], v[1])
```

# hello.txt

hello world


hello this world



# Suppress verbose log messages

- `cd conf`
- `cp log4j.properties.template log4j.properties`
- `edit log4j.properties`
  - change first line to:
    - `log4j.rootCategory=ERROR, console`
  - Or to:
    - `log4j.rootCategory=WARN, console`

# Roadmap

- Spark
  - History, features, RDD, and installation
- **RDD operations** 
  - Creating initial RDDs
  - Actions
  - Transformations
- Examples
- Shuffling in Spark
- Persistence in Spark

# Creating an initial RDD

- From an external file
  - `textFile(<path-to-file>, [# of partitions])`
  - `lines = sc.textFile("hello.txt", 2)`
- From an existing Python collection (e.g., list, tuple, and dictionary)
  - `data = sc.parallelize([1, 2, 3, 4, 5], 2)`
  - create two partitions from given list

# Creating RDD from an external file

- `lines = sc.textFile("hello.txt")` # lines is an RDD
  - Return a collection of lines
  - Spark does not check if file exists right away
  - Nor does it read from the file now

# Action

- Perform a computation on an RDD
  - Return a final value (not an RDD) to client
- Usually the last operation on an RDD
- E.g., `reduce(func)`
  - aggregates all elements in the RDD using `func`
  - returns aggregated value to client


# Actions

- `getNumPartitions()`
- `foreachPartition(func)`
- `collect()`
- `take(n)`
- `count()`, `sum()`, `max()`, `min()`, `mean()`
- `reduce(func)`
- `aggregate(zeroVal, seqOp, combOp)`
- `takeSample(withReplacement, num, [seed])`
- `countByKey()`

# getNumPartitions()

- How many partitions does an RDD have?
- E.g., lines.`getNumPartitions()`  
=> 1
- E.g., data.`getNumPartitions()`  
=> 2

# foreachPartition(func)

- What are in each partition?
  - `def printf(iterator):`  
    `par = list(iterator)`  
    `print 'partition:', par`  
 Iterator for the list of elements  
in the partition
  - `sc.parallelize([1, 2, 3, 4, 5], 2).foreachPartition(printf)`
- =>      `partition: [3, 4, 5]`  
         `partition: [1, 2]`



# collect()

- Show the entire content of an RDD
- `sc.parallelize([1, 2, 3, 4, 5], 2).collect()`
- `collect()`
  - Fetch the entire RDD as a Python list
  - RDD may be partitioned among multiple nodes
  - `collect()` brings all partitions to the client's node
- Problem:
  - may run out of memory when the data set is large

# take(n)

- take(n): collect first n elements from an RDD
- l = [1,2,3,4,5]
- rdd = sc.parallelize(l, 2)
- rdd.take(3)

=>

[1,2,3]

# count()

- Return the number of elements in the dataset
  - It first counts in each partition
  - Then sum them up in the client
- `l = [1,2,3,4,5]`
- `rdd = sc.parallelize(l, 2)`
- `rdd.count()`

=> 5

# sum()

- Add up the elements in the dataset
- `l = [1,2,3,4,5]`
- `rdd = sc.parallelize(l)`
- `rdd.sum()`

=> 15

# reduce(func)

- Use func to aggregate the elements in RDD
- func(a,b):
  - Takes two input arguments, e.g., a and b
  - Outputs a value, e.g., a + b
- func should be commutative and associative
  - Applied to each partition (like a combiner)

# reduce(func)

- func is continually applied to elements in RDD
  - [1, 2, 3]
  - First, compute  $\text{func}(1, 2) \Rightarrow x$
  - Then, compute  $\text{func}(x, 3)$
- If RDD has only one element  $x$ , it outputs  $x$
- Similar to `reduce()` in Python

# Recall Python example

- `def add(a, b): return a + b`
- `reduce(add, [1, 2, 3])`  
 $\Rightarrow 6$

Or simply `reduce(lambda a, b: a + b, [1, 2, 3])`

# Spark example

- `def add(a, b): return a + b`
- `data = sc.parallelize([1, 2, 3], 2)`
- `data.reduce(add)`  
     $\Rightarrow 6$

Or simply: `data.reduce(lambda a, b: a + b)`



# Implementation of reduce(func)

- Suppose [1, 2, 3, 4, 5] => two partitions:
  - [1, 2] and [3, 4, 5]
- `rdd = sc.parallelize([1, 2, 3, 4, 5], 2)`
- Consider `reduce(add)`

# Local reduction

- Apply add to reduce each partition locally
  - Using `mapPartition(func)` (see transformations)
- Func: apply 'add' function to reduce a partition
  - E.g., using Python reduce function
  - `reduce(add, [1, 2]) => 3`
  - `reduce(add, [3, 4, 5]) => 12`

# Global reduction

- Collect all local results
  - using `collect()`
  - => `res = [3, 12]`
- Use Python `reduce` to obtain final result
  - `reduce(add, res)` => `reduce(add, [3, 12]) = 15`

# Example: finding largest integers

- `data = [5, 4, 4, 1, 2, 3, 3, 1, 2, 5, 4, 5]`
- `pdata = sc.parallelize(data)`
- `pdata.reduce(lambda x, y: max(x, y))`  
 $\Rightarrow 5$
- Or simply: `pdata.reduce(max)`

# aggregate(zeroValue, seqOp, combOp)

But note reduce here is different from that in Python:  
zeroValue can have different type than values in p

- For each partition p (values in the partition),
  - "reduce"(seqOp, p, zeroValue)
  - Note if p is empty, it will return zeroValue
- For a list of values, vals, from all partitions, execute:
  - reduce(combOp, vals, zeroValue)

# seqOp and combOp

- seqOp(U, v):
  - how to aggregate values v's in the partition into U
  - U: accumulator, initially U = zeroValue
  - Note: U and v may be of different data type
- combOp(U, p):
  - how to combine results from multiple partitions
  - U: accumulator, initially U = zeroValue
  - p: result from a partition

# Python reduce() w/o initial value

- `reduce(func, list)`
- If list is empty => ERROR
- Else if list contains a single element `v`, return `v`
- Otherwise, set accumulator `x = list[0]`
  - for each of remaining element `list[i]`
    - `x = func(x, list[i])`
  - Return final value of `x`

# Python reduce() with initial value

- `reduce(func, list, initialValue)`
- Same as:
  - `reduce(func, [initialValue] + list)`
- Note: list can be empty now
  - `reduce()` will return `initialValue` when list is empty



# reduce(f) vs aggregate(z, f1, f2)

- func in reduce(func) needs to be commutative and associative
  - While f1 and f2 in aggregate(z, f1, f2) do not need to be
  - f1: similar to the combiner function in Hadoop
- Need to specify initial value for aggregate()
  - & it can be of different type than values in RDD

# Example

- `data = sc.parallelize([1], 2)`
- `data.foreachPartition(printf)`
  - P1: []
  - P2: [1]
- `data.aggregate(1, add, add)`
  - P1 => [1] => after reduction => 1
  - P2 => [1] + [1] = [1, 1] => 2
  - final: [1] + [1, 2] => [1, 1, 2] => 4

# Example

- `data.aggregate(2, add, lambda U, v: U * v)`
  - `P1 => 2`
  - `P2 => 3`
  - Final: `[2] + [2, 3] => 2 * 2 * 3 = 12`  
(where `[2]` is `zeroValue`, `[2,3]` is the list of values from partitions)

# Implementing count() using aggregate()

- `data = sc.parallelize([1, 2, 3, 4, 5])`
- ...

# Implementing mean() using aggregate()

- `data = sc.parallelize([1, 2, 3, 4, 5])`
- ...

# takeSample(withReplacement, num, [seed])

- Take a random sample of elements in rdd
- withReplacement: True if with replacement
- num: sample size
- optional seed: for random number generator
- Useful in many applications, e.g., k-means clustering

# Example

- `data = sc.parallelize(xrange(10))`
- `data.takeSample(False, 2, 1)`
  - `[8, 0]`

# countByKey()

- Only available on RDDs of type (K, V)
  - i.e., RDD that contains a list of key-value **pairs**, e.g., ('hello', 3)
- Return a hashmap (dictionary in Python) of (K, Int) pairs with count for each unique key in RDD
  - Count for key k = # of tuples whose key is k



# Example

- `d = [('hello', 1), ('world', 1), ('hello', 2), ('this', 1), ('world', 0)]`
  - `data = sc.parallelize(d)`
  - `data.countByKey()`
- `=> {'this': 1, 'world': 2, 'hello': 2}`

# Roadmap

- Spark
  - History, features, RDD, and installation
- RDD operations
  - Creating initial RDDs
  - Actions
  - Transformations
- Examples
- Shuffling in Spark
- Persistence in Spark



# Transformation

- Create a new RDD from an existing one
- E.g., `map(func)`
  - Applies `func` to each element of an RDD
  - Returns a new RDD representing mapped result

# Lazy transformations

- Spark does not apply them to RDD right away
  - Just remember what needs to be done
  - Perform transformations until an action is applied
- Advantage
  - Results of transformations pipelined to the action
  - No need to return intermediate results to clients

=> more efficient

# Avoid re-computation

- However, this means that the same RDD may be recomputed multiple times if it is used in multiple actions
  - => All transformations need to be redone
  - => Consequence: costly
- Solution: allow caching of RDDs in memory
  - May also persist them on disk

# Transformations


- `map(func)`
- `filter(func)`
- `flatMap(func)`
- `reduceByKey(func)`
- `groupByKey()`
- `sortByKey(True/False)`
  - `sortBy(keyfunc, ascending=True/False)`
- `distinct()`
- `mapPartitions(func)`

# Transformations

- `join(rdd, [numTasks])`
  - `leftOuterJoin`
  - `rightOuterJoin`
  - `fullOuterJoin`
- `aggregateByKey(zeroValue, seqOp, combOp, [numTasks])`
- `mapValues(func)`
- `flatMapValues(func)`
- `union/intersection/subtract`
- `subtractByKey`

# Transformations

*f, numPartitions=None, partitionFunc=<function  
portable\_hash>)*



- `groupBy(f)`
  - `f` is a function that produces the group key



# map(func)

- map(func): Apply a function func to each element in input RDD
  - func returns a value (could be a list)
- Output the new RDD containing the transformed values produced by func

# Example

- `lines = sc.textFile("hello.txt")`
- `lineSplit = lines.map(lambda s: s.split())`  
`=> [['hello', 'world'], ['hello', 'this', 'world']]`
- `lineLengths = lines.map(lambda s: len(s))`  
`=> [11, 16]`

# filter(func)

- filter(func): return a new RDD with elements of existing RDD for which func returns true
- func should be a **boolean** function
- `lines1 = lines.filter(lambda line: "this" in line)`  
⇒ `['hello this world']`
- What about: `lines.filter(lambda s: len(s) > 11)`?

# Notes

- `data = sc.parallelize([1, 2, 3, 4, 5, 1, 3, 5], 2)`
- `data.map(lambda x: x if x % 2 == 0 else None).collect()`

→ Result

`[None, 2, None, 4, None, None, None, None]`

- `def f(x):`  
    `if x % 2 == 0:`  
        `return x`  
    `else:`

`pass` → Same as "return None"

- `data.map(f).collect()`

→ Produce the same result as above

# Python filter

- `l = [1, 2, 3, 4, 5, 1, 3, 5]`
- `filter(lambda x: x % 2 == 0, l)`
  - `[2, 4]`

# Spark implementation of filter

- `def even(x): return x % 2 == 0`
- `data.filter(even)`

Implemented as follows:

- `def processPartition(iterator):`  
    `return filter(even, iterator)`
- `data.mapPartitions(processPartition)`

# mapPartitions(func)

- Apply transformation to a **partition**
  - input to func is an iterator (over the elements in the partition)
  - func must return an iterable (a list or use yield to return a generator)
- Different from map(func)
  - func in map(func) applies to an **element**

# Implementing aggregate()

- `rdd.aggregate((0,0), combFunc, reduFunc)`
- `def combFunc(U, x): return (U[0] + x, U[1] + 1)`
- `def reduFunc(U, V): return (U[0] + V[0], U[1] + V[1])`
- `def sumf(iterator):`  
    `return [reduce(combFunc, iterator, (0, 0))]`
- `rdd.mapPartitions(sumf).reduce(reduFunc)`



# Exercise

- Implement `count()` using `mapPartitions()` and `reduce()` only
  - `rdd = sc.parallelize([1, 1, 2, 3, 3, 3], 2)`
  - `rdd.count() => 6`

# flatMap(func)

- flatMap(func):
  - similar to map
  - But func here **must** return a list (or generator) of elements
  - & flatMap merges these lists into a single list
- lines.flatMap(lambda x: x.split())  
=>rdd: ['hello', 'world', 'hello', 'this', 'world']

# reduceByKey()

- `reduceByKey(func)`
  - Input: a collection of  $(k, v)$  pairs
  - Output: a collection of  $(k, v')$  pairs
- $v'$ : aggregated value of  $v$ 's in all  $(k, v)$  pairs with the same key  $k$  by applying `func`
- `func` is the aggregation function
  - Similar to `func` in the `reduce(func, list)` in Python

# reduceByKey(func)

- It first performs partition-site reduction & then global reduction
  - By executing the same reduce function
- In other words, func needs to be commutative and associative
- More details:
  - <http://spark.apache.org/docs/latest/api/python/pyspark.html>

# Example

- `rddp = sc.parallelize([(1,2), (1,3), (2,2), (1,4), (3,5), (2, 4), (1, 5), (2, 6)], 2)`
- `def printf(part):`  
    `print list(part)`
- `rddp.foreachPartition(printf)`
  - Partition 1: `[(1, 2), (1, 3), (2, 2), (1, 4)]`
  - Partition 2: `[(3, 5), (2, 4), (1, 5), (2, 6)]`

# Example

- `from operator import add`
- `rddp.reduceByKey(add)`
- It will first execute local reduce:
  - Partition 1:  $[(1, 2), (1, 3), (2, 2), (1, 4)] \Rightarrow (1, 9), (2, 2)$
  - Partition 2:  $[(3, 5), (2, 4), (1, 5), (2, 6)] \Rightarrow (3, 5), (1, 5), (2, 10)$

# Example

- Final reduce at reducer side
  - $(1, 9), (1, 5) \Rightarrow (1, 14)$
  - $(2, 2), (2, 10) \Rightarrow (2, 12)$
  - $(3, 5) \Rightarrow (3, 5)$
- Note that if there are two reducers, then:
  - Some keys, e.g., 1, may be reduced by one reducer
  - Others, e.g., 2 and 3, by the other

# reduceByKey() vs. reduce()

- `reduceByKey()` returns an RDD
  - Reduce values per key
- `reduce()` returns a non-RDD value
  - Reduce all values!



# Exercise

- Implement countByKey using reduceByKey
  - `rddp = sc.parallelize([(1,2), (1,3), (2,2), (1,4), (3,5), (2, 4), (1, 5), (2, 6)], 2)`
  - `rddp.countByKey() => {1: 4, 2: 3, 3: 1}`

# aggregateByKey

- `aggregateByKey(zeroValue, combOp, reduOp)`
  - Input RDD: a list of (k, v) pairs
  - Aggregate values for each key
- Return a value U for each key
  - Note that U may be a tuple
  - `zeroValue`: initial value for U
  - `combOp(U, v)`: (function for) local reduction
  - `reduOp(U1, U2)`: global reduction

# Computing group averages

- `rdd1 = rddp.aggregateByKey((0,0), lambda U,v: (U[0] + v, U[1] + 1), lambda U1,U2: (U1[0] + U2[0], U1[1] + U2[1]))`  
– `[(2, (12, 3)), (1, (14, 4)), (3, (5, 1))]`
- `rdd1.map(lambda (x, (y, z)): (x, float(y)/z))`  
– `[(2, 4.0), (1, 3.5), (3, 5.0)]`

# Example: aggregateByKey

- `data = sc.parallelize([(1, 1), (1,2), (1,3)], 2)`
- `data.foreachPartition(printf)`
  - `[(1, 1)]`
  - `[(1, 2), (1, 3)]`
- `data.aggregateByKey(1, add, add).collect()`
  - `[(1, 8)]`

# Compared with aggregate()

- `data = sc.parallelize([1, 2, 3], 2)`
- `data.foreachPartition(printf)`
  - `[1]`
  - `[2, 3]`
- `data.aggregate(1, add, add)`
  - `9`

# aggregateByKey vs. aggregate

- zeroValue in aggregateByKey
  - Used only combOp (i.e., reduction within a partition)
- zeroValue in aggregate
  - Used in both combOp and reduOp
  - E.g., `data.aggregate(1, add, add) => 9`

# aggregateByKey vs. reduceByKey

- aggregateByKey more general than reduceKey
  - Can specify different functions for combiner and reducer
  - can specify initial value for U, the accumulator
  - aggregated value may have different type than that of value v of input RDD
- E.g., in previous example:
  - v is an integer, while U is a tuple (sum, count)

# Exercise

- Implement `reduceByKey(add)` using `aggregateByKey()`
- `rddp = sc.parallelize([(1,2), (1,3), (2,2), (1,4), (3,5), (2, 4), (1, 5), (2, 6)], 2)`
  - `rddp.reduceByKey(add) => [(2, 12), (1, 14), (3, 5)]`



# groupByKey()

- groupByKey()
  - Similar to reduceByKey(func)
  - But without func & returning (k, Iterable(v)) instead
- rddp.groupByKey()  
⇒[(2, <iterable>), (1, ...), (3, ...)]

# Example

- `rddp.groupByKey().mapValues(list).collect()`
  - `mapValues` converts iterable value into a list

=> [(2, [2, 4, 6]), (1, [2, 3, 4, 5]), (3, [5])]

# groupBy()

- `rddp.groupBy(lambda t:  
t[0]).mapValues(list).collect()`

=>

```
[(2, [(2, 2), (2, 4), (2, 6)]), (1, [(1, 2), (1, 3), (1, 4),  
(1, 5)]), (3, [(3, 5)])]
```

# sortByKey(True/False)

- `sortByKey([asc])`
  - Sort input RDD with (k, v) pairs by key
  - Ascending if asc (a boolean value) is True
- `rddp.sortByKey(False).collect()`  
=> [(3, 5), (2, 2), (2, 4), (2, 6), (1, 2), (1, 3), (1, 4), (1, 5)]

# sortBy(keyfunc, ascending=True/False)

- `>>> rddp.sortBy(lambda p: p[1]).collect()`
  - `[(1, 2), (2, 2), (1, 3), (1, 4), (2, 4), (3, 5), (1, 5), (2, 6)]`
- `>>> rddp.sortBy(lambda p: -p[1]).collect()`
  - `[(2, 6), (3, 5), (1, 5), (1, 4), (2, 4), (1, 3), (1, 2), (2, 2)]`
- `>>> rddp.sortBy(lambda p: p[1], ascending=False).collect()`
  - `[(2, 6), (3, 5), (1, 5), (1, 4), (2, 4), (1, 3), (1, 2), (2, 2)]`

# distinct()

- Return an RDD with distinct elements of source RDD
  - `data = [5, 4, 4, 1, 2, 3, 3, 1, 2, 5, 4, 5]`
  - `pdata = sc.parallelize(data, 2)`
  - `pdata.distinct().collect()`
- => `[2, 4, 1, 3, 5]`

# Exercise

- Implement `distinct()` using `reduceByKey()/groupByKey()`
- `rdd = sc.parallelize([3, 1, 2, 3, 1, 3, 3, 2])`
- `rdd.distinct()`  
=> `[1, 2, 3]`

# join(rdd)

- `rdd1.join(rdd2)`
  - Joining tuples of two RDDs on the key
  - `rdd1`: an RDD containing a list of  $(k, v)$ 's
  - `rdd2`: another RDD containing a list of  $(k, w)$ 's
- Output an RDD containing  $(k, (v, w))$ 's
  - That is,  $(k, v)$  joins with  $(k, w) \Rightarrow (k, (v, w))$



# Example

- `ds1 = sc.parallelize([(1,2), (2,3)])`
- `ds2 = sc.parallelize([(2,4), (3,5)])`
- `ds1.join(ds2)`
  - `[(2, (3, 4))]`

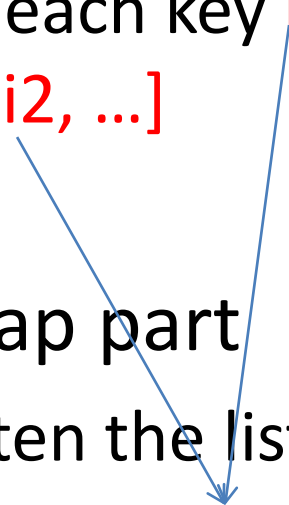
# Outer joins

- Also retain dangling tuples
- `ds1.leftOuterJoin(ds2)`
  - `[(1, (2, None)), (2, (3, 4))]`
- `ds1.rightOuterJoin(ds2)`
  - `[(2, (3, 4)), (3, (None, 5))]`
- `ds1.fullOuterJoin(ds2)`
  - `[(1, (2, None)), (2, (3, 4)), (3, (None, 5))]`

# mapValues

- `mapValues(func)`
  - For each key, apply `func` to each value of the key
- `x = sc.parallelize([("a", ["apple", "banana", "lemon"]), ("b", ["grapes"])]])`
- `x.mapValues(lambda l: len(l)).collect()`
  - `[('a', 3), ('b', 1)]`

# flatMapValues(func)

- mapValues part
    - For each key **k**, apply func to its value, return a **list** **[i1, i2, ...]**
  - flatMap part
    - flatten the lists into a single list but retain the key
    - => **[(k, i1), (k, i2), ..., (k', i1'), (k', i2'), ...]**
- 
- The diagram consists of two blue arrows. The first arrow originates from the word 'list' in the first bullet point and points down to the 'flatten' step in the second bullet point. The second arrow originates from the list '[i1, i2, ...]' in the first bullet point and also points down to the 'flatten' step in the second bullet point.




# Example

- `rdd = sc.parallelize([(1, "hello world"), (2, "hello this world")])`
  - For example, 1 and 2 may be document id's
- `rdd2 = rdd.flatMapValues(lambda s: s.split())`
  - `[(1, 'hello'), (1, 'world'), (2, 'hello'), (2, 'this'), (2, 'world')]`

# Exercise

- Use `mapValues()` and `flatMap()` implement `flatMapValues()` in the previous slide

# union(rdd)

- `rdd1.union(rdd2)`
  - Returns all elements in `rdd1` and `rdd2`
  - Does not remove duplicates (so bag union)
- `rdd1 = sc.parallelize([1, 1, 2, 3, 3, 3], 2)`  2 partitions
- `rdd2 = sc.parallelize([1, 2, 2, 5], 2)`  2 partitions
- `rdd1.union(rdd2)`  4 partitions
  - `[1, 1, 2, 3, 3, 3, 1, 2, 2, 5]`

# intersection(rdd)

- `rdd1.intersection(rdd2)`
  - Returns elements in both `rdd1` and `rdd2`
  - Duplicates will be removed! (so set-semantics)
- `rdd1 = sc.parallelize([1, 1, 2, 3, 3, 3])`
- `rdd2 = sc.parallelize([1, 2, 2, 5])`
- `rdd1.intersection(rdd2)`
  - `[2, 1]`




# subtract(rdd)

- `rdd1.subtract(rdd2)`
  - Return values in `rdd1` that do not appear in `rdd2`
  - Note: neither set nor bag semantics!
- `rdd1 = sc.parallelize([1, 1, 2, 3, 3, 3])`
- `rdd2 = sc.parallelize([1, 2, 2, 5])`
- `rdd1.subtract(rdd2)`
  - `[3, 3, 3]`
  - Note: **1 not included in result** (unlike bag difference)

# subtractByKey(rdd)

- `rdd1.subtractByKey(rdd2)`
  - Return each (key, value) pair in `rdd1` that has no pair with matching key in `rdd2`
- `rdd1 = sc.parallelize([1, 1, 2, 3, 3, 3]).map(lambda x: (x, 1))`
- `rdd2 = sc.parallelize([1, 2, 2, 5]).map(lambda x: (x, 1))`
- `rdd1.subtractByKey(rdd2)`
  - `[(3, 1), (3, 1), (3, 1)]`

# Roadmap

- Spark
  - History, features, RDD, and installation
- RDD operations
  - Creating initial RDDs
  - Actions
  - Transformations
- Examples 
- Shuffling in Spark

# WordCount

- from operator import add
  - lines = sc.textFile("hello.txt")
  - counts = lines.flatMap(lambda x: x.split(' ')) \
- .map(lambda x: (x, 1)) \
- .reduceByKey(add)
- counts.collect()

=> [(u'this', 1), (u'world', 2), (u'hello', 2)]

# Word length histogram

- long: if  $> 4$  letters
- short: otherwise
- ```
def myFunc(x):  
    if len(x) > 4:  
        return ('long', 1)  
    else:  
        return ('short', 1)
```

# Word length histogram

- `sc.textFile("hello.txt") \`  
    `.flatMap(lambda x: x.split(" ")) \`  
    `.map(myFunc) \`  
    `.reduceByKey(add) \`  
    `.collect()`

`=> [('short', 1), ('long', 4)]`

# Adding ratings for each person

Ratings.txt

(patrick, 4)

(matei, 3)

(patrick, 1)

(aaron, 2)

(aaron, 2)

(reynold, 1)

(aaron, 5)



(aaron, 9)

(patrick, 5)

...

# Adding ratings for each person

- `sc.textFile("ratings.txt") \`  
    `.map(lambda s: s[1:-1].split(",")) \`  
    `.collect()`

Strip off ()



=>

```
[[u'patrick', u'4'], [u'matei', u'3'], [u'patrick', u'1'],  
[u'aaron', u'2)'], [u'aaron', u'2'], [u'reynold', u'1'],  
[u'aaron', u'5']]
```



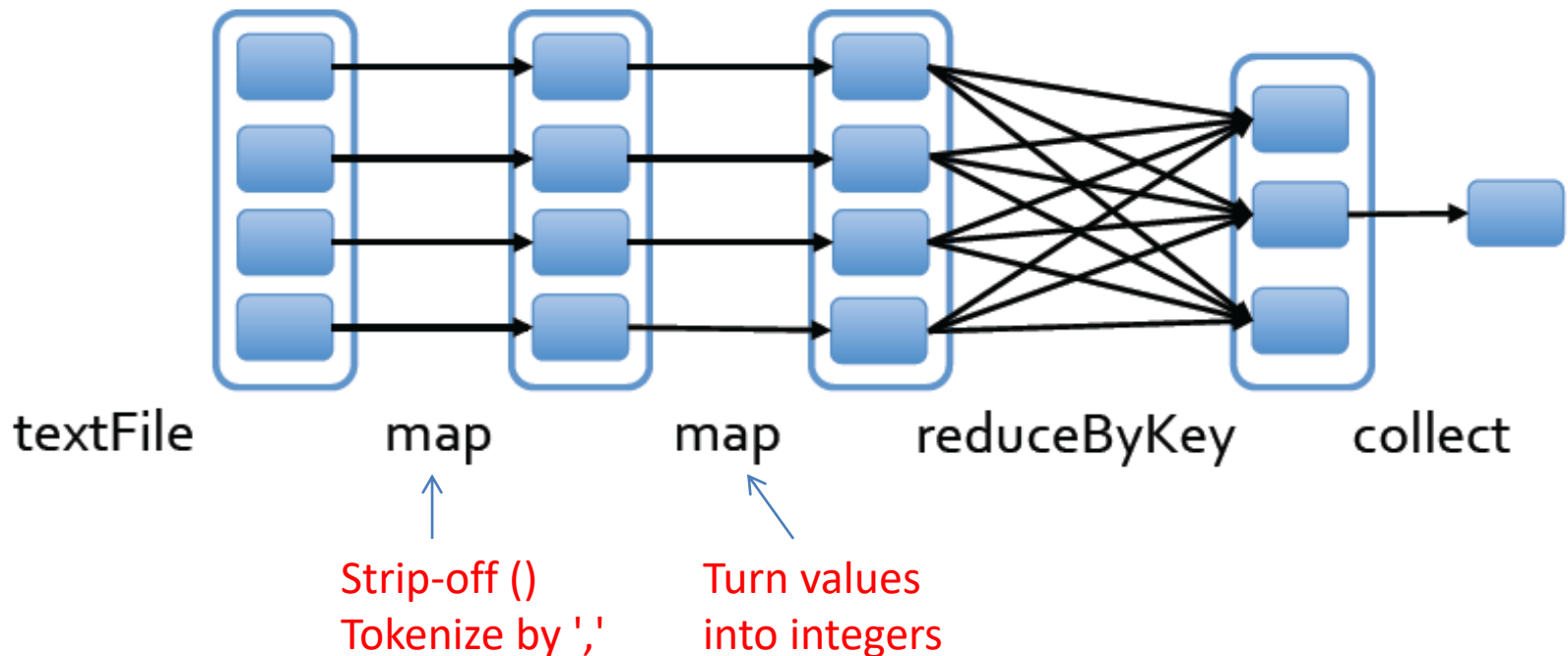
# Adding ratings for each person

- `sc.textFile("ratings.txt") \`  
    `.map(lambda s: s[1:-1].split(",")) \`  
    `.map(lambda p: (p[0], int(p[1]))) \`  
    `.reduceByKey(lambda a, b: a + b) \`  
    `.collect()`

=> [(u'patrick', 5), (u'aaron', 9), (u'reynold', 1),  
(u'matei', 3)]

# Execution steps

- Note that reduceByKey requires shuffling



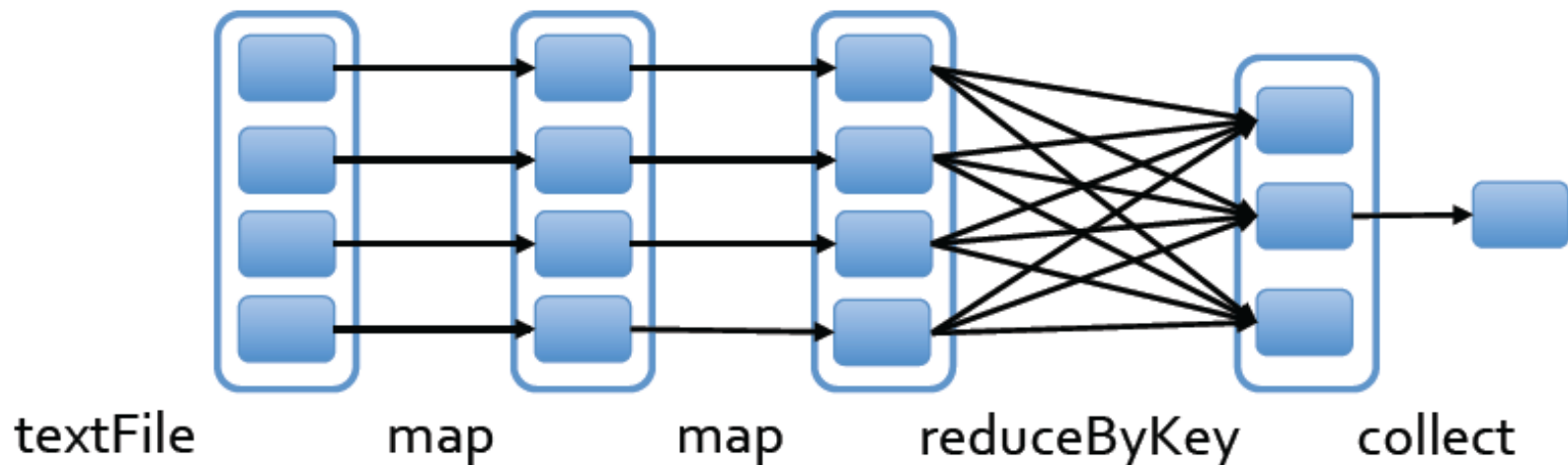
# Roadmap

- Spark
  - History, features, RDD, and installation
- RDD operations
  - Creating initial RDDs
  - Actions
  - Transformations
- Examples
- **Shuffling in Spark**
- Persistence in Spark




# Shuffling

- Data are essentially repartitioned
  - E.g., `reduceByKey` repartitions the data by key
- A costly operation: a lot of local & network I/O's



# Another example: sortByKey

- Sampling stage:
    - Sample data to create a range-partitioner
    - Ensure even partitioning
  - "Map" stage:
    - Write (sorted) data to destined partition for reduce stage
  - "Reduce" stage:
    - get map output for specific partition
    - Merge the sorted data
- Data are shuffled between Map and Reduce stage
- 
- The diagram consists of two blue arrows. The first arrow starts at the end of the text 'destined partition for reduce stage' in the 'Map' stage and points diagonally down and to the right. The second arrow starts at the end of the red text 'Data are shuffled between Map and Reduce stage' and points diagonally down and to the left, meeting the first arrow.

# Transformations that require shuffling

- `reduceByKey(func)`
- `groupByKey()`
- `sortByKey([asc])`
- `distinct()`

# Transformations that require shuffling

- `join(rdd):`
  - `leftOuterJoin`
  - `rightOuterJoin`
  - `fullOuterJoin`
- `aggregateByKey(zeroValue, seqOp, combOp)`
- `intersection/subtract`
- `subtractByKey`

# Transformations that do not need shuffling

- `map(func)`
- `filter(func)`
- `flatMap(func)`
- `mapValues(func)`
- `union`
- `mapPartitions(func)`



# Roadmap

- Spark
  - History, features, RDD, and installation
- RDD operations
  - Creating initial RDDs
  - Actions
  - Transformations
- Examples
- Shuffling in Spark
- Persistence in Spark



# RDD persistence

- `rdd.persist(<storageLevel>)`
- Store the content of RDD for later reuse
  - `storageLevel` specifies where content is stored
  - E.g., in memory (default) or on disk
- `rdd.persist()` or `rdd.cache()`
  - Content stored in main memory

# RDD persistence

- Executed at nodes having partitions of RDD
- Avoid re-computation of RDD in reuse

# Example

- `ratings = sc.textFile("ratings.txt") \`  
    `.map(lambda s: s[1:-1].split(",")) \`  
    `.map(lambda p: (p[0], int(p[1]))) \`  
    `.cache()`
- `ratings.reduceByKey(lambda a, b: a +`  
    `b).collect()`
  - ratings RDD will be computed for the first time & result cached

# Example

- `ratings.countByKey()`
  - It will use cached content of "ratings" rdd

# Automatic persistence

- Spark automatically persists intermediate data in shuffling operations (e.g., reduceByKey)
- This avoids re-computation when node fails

# K-means clustering

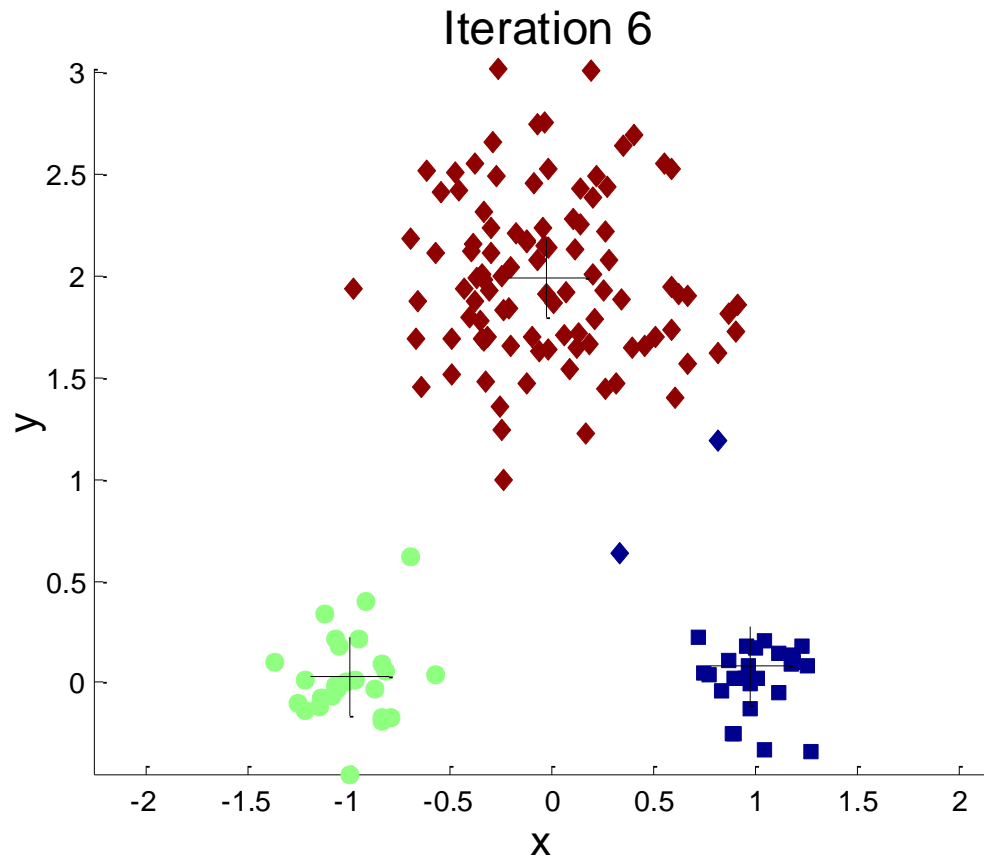
- Find  $k$  clusters in a data set
  - $k$  is pre-determined
- Iterative process
  - Start with initial guess of centers of clusters
  - Repeatedly refine the guess until stable (e.g., centers do not change much)
- Need to use data set at each iteration

# K-means clustering

- Assign point  $p$  to the closest center  $c$ 
  - Distance = Euclidean distance between  $p$  and  $c$
- Re-compute the centers based on assignments
- Coordinates of center of a cluster =
  - Average coordinate of all points in the cluster
  - E.g.,  $(1, 1, 1)$   $(3, 3, 3) \Rightarrow$  center:  $(2, 2, 2)$



# K-means clustering



```

sc = SparkContext(appName="PythonKMeans")
lines = sc.textFile(sys.argv[1])
data = lines.map(parseVector).cache()
K = int(sys.argv[2])
convergeDist = float(sys.argv[3])

kPoints = data.takeSample(False, K, 1)
tempDist = 1.0

while tempDist > convergeDist:
    closest = data.map(
        lambda p: (closestPoint(p, kPoints), (p, 1)))
    pointStats = closest.reduceByKey(
        lambda p1_c1, p2_c2: (p1_c1[0] + p2_c2[0], p1_c1[1] + p2_c2[1]))
    newPoints = pointStats.map(
        lambda st: (st[0], st[1][0] / st[1][1])).collect()
    tempDist = sum(np.sum((kPoints[iK] - p) ** 2) for (iK, p) in newPoints)
    for (iK, p) in newPoints:
        kPoints[iK] = p

print("Final centers: " + str(kPoints))
sc.stop()

```

Persist data points in memory

Initial centers

New centers

Sum of distances between new and old centers

# Parse input & find closest center

```
def parseVector(line):  
    return np.array([float(x) for x in line.split(' ')])  
  
def closestPoint(p, centers):  
    bestIndex = 0  
    closest = float("+inf")  
    for i in range(len(centers)):  
        tempDist = np.sum((p - centers[i]) ** 2)  
        if tempDist < closest:  
            closest = tempDist  
            bestIndex = i  
    return bestIndex
```

# kmeans-data.txt

- A text file contains the following lines

– 0.0 0.0 0.0

– 0.1 0.1 0.1

– 0.2 0.2 0.2

– 9.0 9.0 9.0

– 9.1 9.1 9.1

– 9.2 9.2 9.2

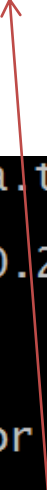
```
kmeans.py q.py yarn
[ec2-user@ip-172-31-52-194 spark-2.0.1-bin-hadoop2.7]$ cat kmeans-data.txt
0.0 0.0 0.0
0.1 0.1 0.1
0.2 0.2 0.2
9.0 9.0 9.0
9.1 9.1 9.1
9.2 9.2 9.2
```

- Each line is a 3-dimensional data point

# Parse & cache the input dataset

- "data" RDD is now cached in main memory

```
>>> lines = sc.textFile("kmeans-data.txt")
>>> lines.collect()
[u'0.0 0.0 0.0', u'0.1 0.1 0.1', u'0.2 0.2 0.2', u'9.0 9.0 9.0', u'9.1 9.1 9.1', u'9.2 9.2 9.2']
>>>
>>> def parseVector(line):
...     return np.array([float(x) for x in line.split(' ')])
...
>>> data = lines.map(parseVector).cache()
>>> data.collect()
[array([ 0.,  0.,  0.]), array([ 0.1,  0.1,  0.1]), array([ 0.2,  0.2,  0.2]), array([ 9.,  9.,  9.]), array([ 9.1,  9.1,  9.1]), array([ 9.2,  9.2,  9.2])]
```



# Generating initial centers

- Recall takeSample() action
  - False: sample without replacement
  - $K = 2$

```
>>> kPoints = data.takeSample(False, K, 1)
>>> kPoints
[array([ 0.1,  0.1,  0.1]), array([ 0.2,  0.2,  0.2])]
```

# Assign point to its closest center

- Center 0 has points: (0, 0, 0) and (.1, .1, .1)
- Center 1 has the rest: (.2, .2, .2), (.9, .9, .9), ...

```
>>> def closestPoint(p, centers):
...     bestIndex = 0
...     closest = float("+inf")
...     for i in range(len(centers)):
...         tempDist = np.sum((p - centers[i]) ** 2)
...         if tempDist < closest:
...             closest = tempDist
...             bestIndex = i
...     return bestIndex
...
>>> closest = data.map(lambda p: (closestPoint(p, kPoints), (p, 1)))
>>> closest.collect()
[(0, (array([ 0.,  0.,  0.]), 1)), (0, (array([ 0.1,  0.1,  0.1]), 1)),
(1, (array([ 0.2,  0.2,  0.2]), 1)), (1, (array([ 9.,  9.,  9.]), 1)), (
1, (array([ 9.1,  9.1,  9.1]), 1)), (1, (array([ 9.2,  9.2,  9.2]), 1))]
```

# Getting statistics for each center

- pointStats has a key-value pair for each center
- Key is center # (0 or 1 for this example)
- Value is a tuple (sum, count)
  - sum = the sum of coordinates over all points in the cluster
  - Count = # of points in the cluster


```
>>> pointStats = closest.reduceByKey(lambda p1_c1, p2_c2: (p1_c1[0] + p2_c2[0], p1_c1[1] + p2_c2[1]))
>>> pointStats.collect()
[(0, (array([ 0.1,  0.1,  0.1]), 2)), (1, (array([ 27.5,  27.5,  27.5]), 4))]
```



# Computing coordinates of new centers

- Coordinate = sum of point coordinates/count
  - E.g., center 0:  $[.1, .1, .1] / 2 = [.05, .05, .05]$

```
>>> newPoints = pointStats.map(lambda st: (st[0], st[1][0] / st[1][1])).  
collect()  
>>> newPoints  
[(0, array([ 0.05,  0.05,  0.05])), (1, array([ 6.875,  6.875,  6.875]))  
]
```



Can use mapValues here too:

```
newPoints1 = pointStats.mapValues(lambda stv: stv[0]/stv[1]).collect()
```

# Distance btw new & old centers

- Old center: [.1, .1, .1] and [.2, .2, .2]
- New center: [.05, .05, .05] and [6.875, 6.875, 6.875]
- Distance =  $(.1-.05)^2*3 + (6.875-.2)^2*3 \sim 133.67$ 
  - To be more exact, it is  $\text{sqrt}(133.67) = 11.56$

```
>>> tempDist = sum(np.sum((kPoints[iK] - p) ** 2) for (iK, p) in newPoints)
>>> tempDist
133.67437499999994
```

# RDD operations

- A complete list:
  - <http://spark.apache.org/docs/latest/api/python/yspark.html>

# Resources

- Spark programming guide:
  - <https://spark.apache.org/docs/latest/>
- [Lambda, filter, reduce and map:](http://www.python-course.eu/lambda.php)
  - <http://www.python-course.eu/lambda.php>
- Improving Sort Performance in Apache Spark: It's a Double
  - <http://blog.cloudera.com/blog/2015/01/improving-sort-performance-in-apache-spark-its-a-double/>

# Readings

- [Spark: Cluster Computing with Working Sets](#), 2010.
- [Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing](#), 2012.

# References

- Functional programming in Python
  - <https://docs.python.org/2/howto/functional.html>
- Learning Spark by Matei Zaharia, et. Al. O'Reilly, 2015
  - <https://www.safaribooksonline.com/library/view/learning-spark/9781449359034/>

# References

- Sort-based shuffle implementation
  - <https://issues.apache.org/jira/browse/SPARK-2045>
- Sort-Based Shuffle in Spark
  - <https://issues.apache.org/jira/secure/attachment/12655884/Sort-basedshuffledesign.pdf>