Northeastern

EECE5698
Parallel Processing for Data Analytics

Lecture 4: Key-Value Pairs & Partitioning

Outline

- Key-Value Pairs
- Joins
- Parallelism & Partitioners

Northeastern

Outline

- ❑Key-Value Pairs
- ❑Joins
- ❑Parallelism & Partitioners

Working with Key-Value Pairs

- RDDs of **key-value pairs** play a special role in Spark
- Python: a pair is a **tuple** of **two elements**:

```
pair = (a, b)
pair[0] # => a
pair[1] # => b
```

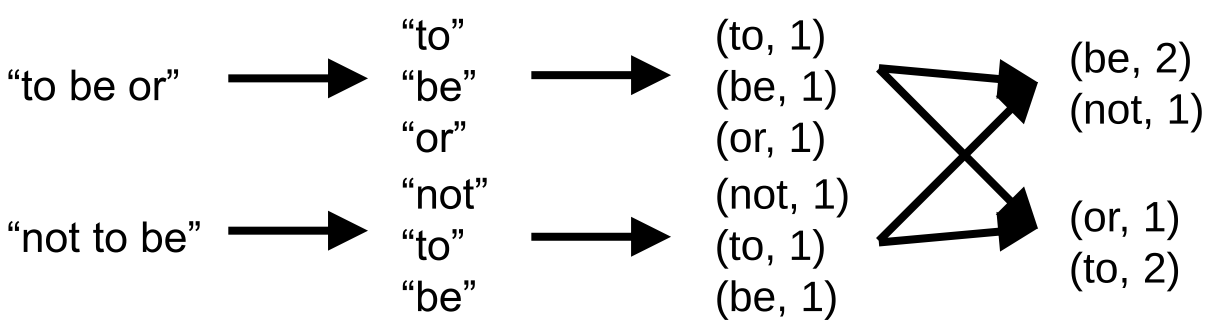
reduceByKey

```
pets = sc.parallelize([('cat', 1), ('dog', 1), ('cat', 2)])

pets.reduceByKey(lambda x, y: x + y)
# => {(cat, 3), (dog, 1)}
```

- It's a **transform, not an action** (produces a new rdd)
- Presumes that data is in (key,value) pair form
  - Error will be generated if they are not.
  - True for all …`ByKey()` operations

Example: Word Count

```
lines = sc.textFile("hamlet.txt")
counts = lines.flatMap(lambda line: line.split()) \
              .map(lambda word: (word, 1)) \
              .reduceByKey(lambda x, y: x + y)
```



How Does `reduceByKey` Work? (simplified)

Each key is mapped to a machine **(actually, partition)**

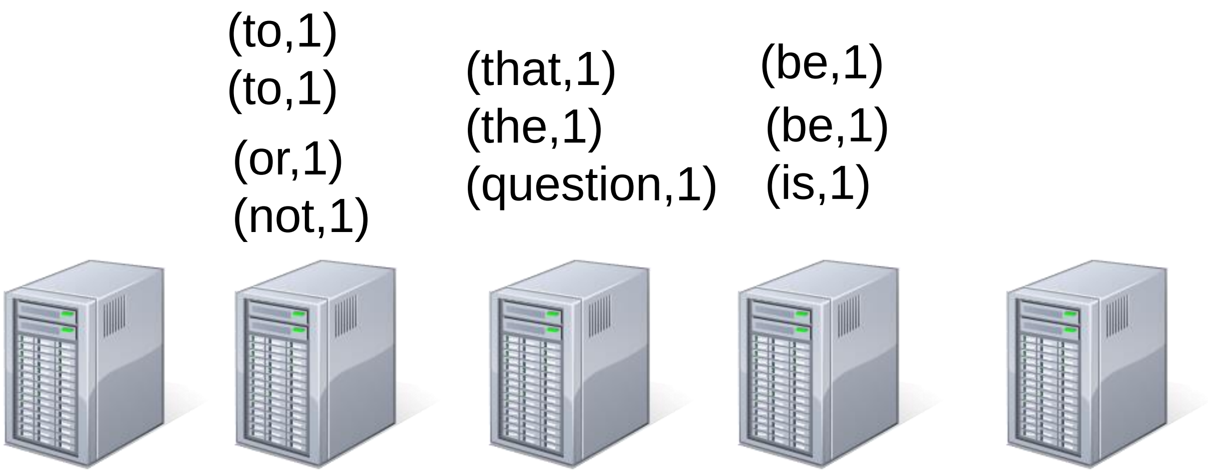
```
target_machine = hash(Key) % num_machines
```

using python's builtin `hash()` function

(to,1) (be,1) | (or,1) (not,1) | (to,1) (be,1) | (that,1) (is,1) | (the,1) (question,1)

How Does `reduceByKey` Work? (simplified)

A **shuffle** takes place: key-value pairs are moved appropriate machines, collocating pairs with identical keys



How Does `reduceByKey` Work? (simplified)

Reduce then applied locally at each machine

(to,2)
(or,1)
(not,1)

(that,1)
(the,1)
(question,1)

(be,2)
(is,1)

Optimizations

- Values are first combined locally **before the shuffle** takes place

- **Shuffles avoided** when not needed (partition-awareness, described soon)

- Big improvement in later versions of Spark: **sort-based** shuffle (we will not cover this)

Additional ByKey Transforms

```
pets = sc.parallelize([('cat', 1), ('dog', 1), ('cat', 2)])

pets.reduceByKey(lambda x, y: x + y)
# => {(cat, 3), (dog, 1)}

pets.groupByKey()
# => {(cat, Seq(1, 2)), (dog, Seq(1)}

pets.sortByKey()
# => {(cat, 1), (cat, 2), (dog, 1)}
```

Transforms Applied on Values only

```
pets = sc.parallelize([('cat', 1), ('dog', 1), ('cat', 2)])
```

```
pets.mapValues(lambda x: x + 1 )
# => [('cat', 2), ('dog', 2), ('cat', 3)]

pets.flatMapValues(lambda x: range(x+1))
# => [('cat', 0), ('cat', 1), ('dog', 0), ('dog', 1), ('cat',
0), ('cat', 1), ('cat', 2)]
```

Useful PairRDD Transforms/Actions/IO

```
pets = sc.parallelize([('cat', 1), ('dog', 1), ('parrot', 2)])

pets.values()                      # rdd containing values only
# => [1, 1, 2]

pets.keys()                        # rdd containing keys only
# => ['cat', 'dog', 'parrot']

pets.collectAsMap()                 # returns dictionary to driver
# => {'cat':1,  'dog':1, 'parrot':2}

allFiles = sc.wholeTextFiles('dir') # loads all files in
directory 'dir' in a PairRDD, with file names as keys end file
contents as values
```

combineByKey() -- similar to aggregate()

combineByKey(*createCombiner*, *mergeValue*, *mergeCombiners*)

turns a V into a C

merges a V into a C

combines two C's into a single one

Turns an RDD[(K, V)] into a result of type RDD[(K, C)], for a "combined type" C

combineByKey() -- similar to aggregate()

combineByKey(*createCombiner*, *mergeValue*, *mergeCombiners*)

turns a V into a C

merges a V into a C

combines two C's into a single one

Turns an RDD[(K, V)] into a result of type RDD[(K, C)], for a "combined type" C

Example: Computing a Per Key Average:

```
keyAvg = words.combineByKey(
            (lambda x: (x,1)),                               #converts val to (val,1)
            (lambda x, y: (x[0] + y, x[1] + 1)),             #adds val to running tally
            (lambda x, y: (x[0] + y[0], x[1] + y[1]))        #merges two tallies
            ).mapValues(lambda (val,count):1.*val/count)
```

Outline

- ❑Key-Value Pairs
- ❑Joins
- ❑Parallelism & Partitioners

Joins

```
grades = sc.parallelize([('Alice', 2), ('Bob', 4), ('Bob', 6)])
register = sc.parallelize([('Bob','UGrad'),(('Maria','Grad'))])

grades.join(register)        # Perform an inner join between two RDDs
# => [('Bob', (4, 'UGrad')), (Bob, (6, 'UGrad'))]
```

grades

key	value
Alice	2
Bob	4
Bob	6

⋈

register

key	value
Bob	UGrad
Maria	Grad

→

grades.join(register)

key	value
Bob	(4,UGrad)
Bob	(6,UGrad)

If `(key,val1)` in rdd and `(key,val2)` in other, join contains `(key,(val1,val2))`

How Does `join` Work? (simplified)

Joins require **shuffling**

rdd:grades
rdd:register

(Maria, Grad)
(Alice,2)
(Bob,4)
(Bob,6)
(Bob,Ugrad)

How Does `join` Work? (simplified)

Joins require **shuffling**

rdd:grades
rdd:register

(Bob,Ugrad)

(Maria, Grad)

(Bob,4)

(Alice,2)

(Bob,6)

How Does `join` Work? (simplified)

Joins require **shuffling**

rdd:grades

rdd:register

Shuffles avoided when not necessary through **partition-awareness**

(Bob,(4,Ugrad))

(Bob,(6,Ugrad))

Other Transformations On Pairs of RDDs

```
rdd = sc.parallelize([(1, 2), (3, 4), (3, 6)])
other = sc.parallelize([(3,9),(5,8)])

rdd.join(other)                          # Perform an inner join
# => [(3, (4, 9)), (3, (6, 9))]

rdd.leftOuterJoin(other)                 # Perform a left outer join
# => [(1, (2, None)), (3, (4, 9)), (3, (6, 9))]

rdd.rightOuterJoin(other)                # Perform a right outer join
# => [(3, (4, 9)), (3, (6, 9)), (5, (None, 8))]

rdd.subtractByKey(other)         # Remove elements with key in other
# => [(1,2)]

rdd.cogroup(other)          # Group data sharing the same key together
# => [(1,([2],[])), (3,([4, 6],[9])), (5,([],[8]))]
```


Northeastern

Outline

- ❑Key-Value Pairs
- ❑Joins
- ❑Parallelism & Partitioners

Controlling the Level of Parallelism

All the pair RDD operations, and some of the non-pair operations, take an optional second parameter for **number of partitions**

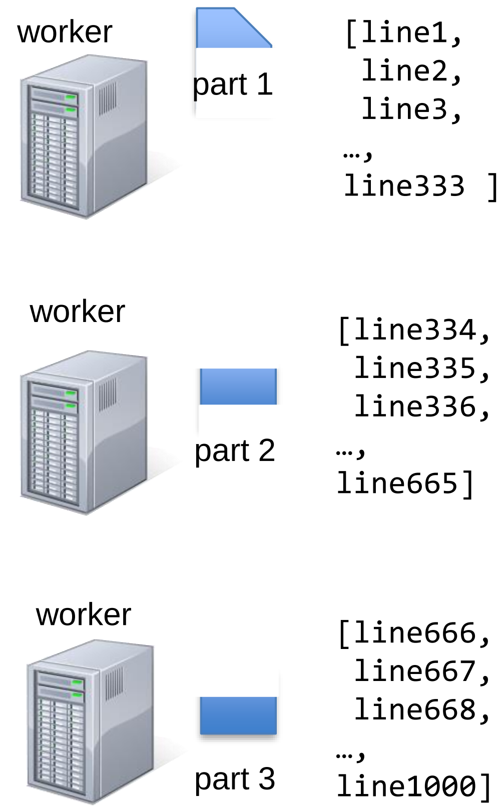
```
words.reduceByKey(lambda x, y: x + y, 5)

words.groupByKey(5)

visits.join(pageViews, 5)
```

This can be used to control the **level of parallelism**

Northeastern

What Are Partitions?

- RDDs are internally split into **partitions**




```
myrdd=sc.textFile("WarAndPeace.txt",3)
```

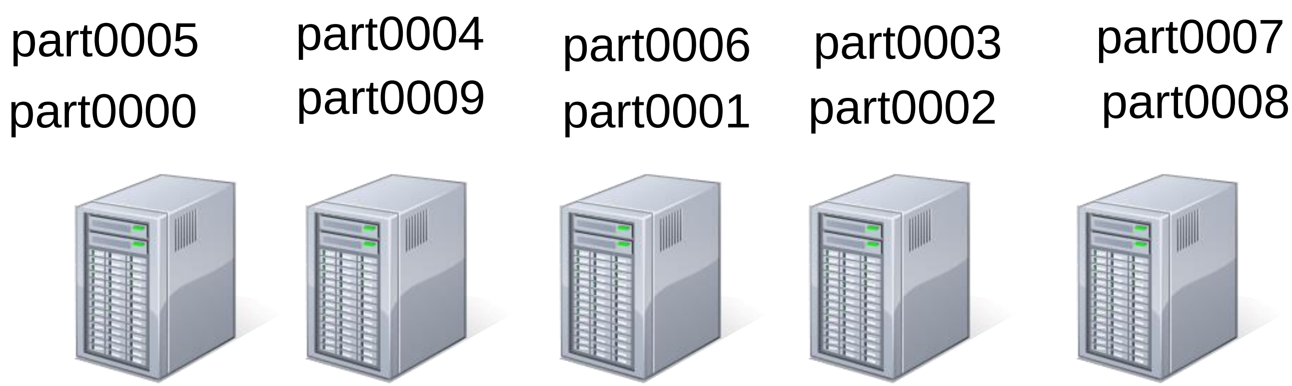

Northeastern

What Are Partitions?

 # partitions < #machines

What Are Partitions?

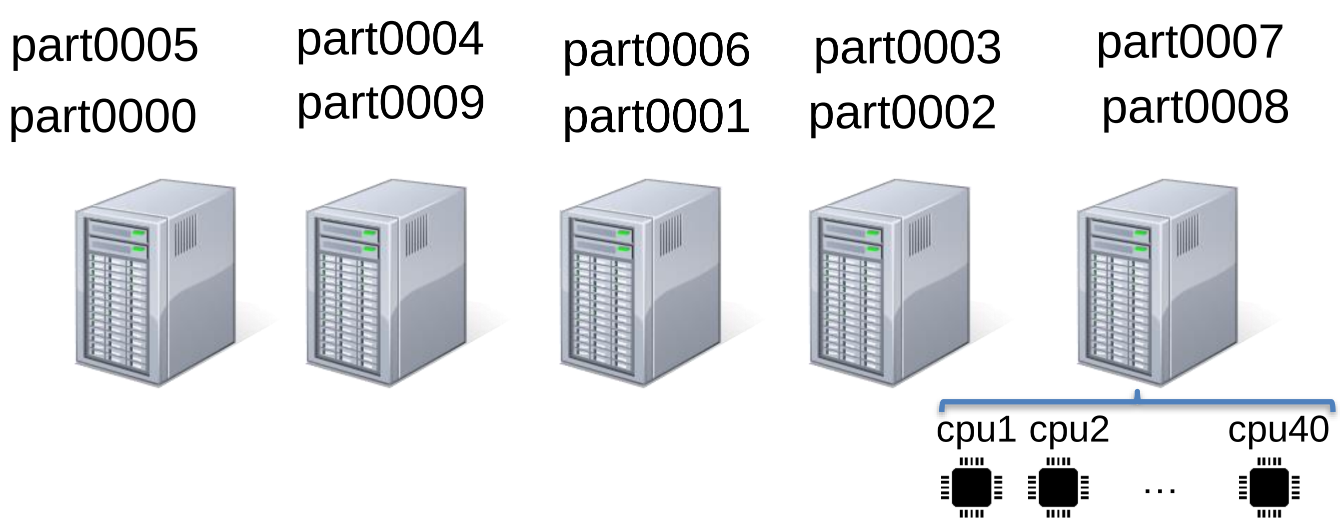
 # partitions > #machines



Number of Partitions Controls Parallelism

```
rdd.map(lambda x: x+1)
```

- ❑ Map executed **serially** in each partition
- ❑ If machine stores **k** partitions, and has **n>k** processors, partition evaluations **executed in parallel**



How Many Partitions Should One Use?

- *m* workers with *k* processors each

  "ideal" #partitions = $m\,k$

- **Fewer partitions**: not exploiting full parallelism in this operation
- **More partitions**: No advantage in speedup; in practice, there may be advantage in memory usage (each cpu dealing with smaller partition, spark less likely to crash/hang)

Northeastern

Mapping Data to Partitions

Each key is mapped to a ~~machine~~ **(actually, partition)**

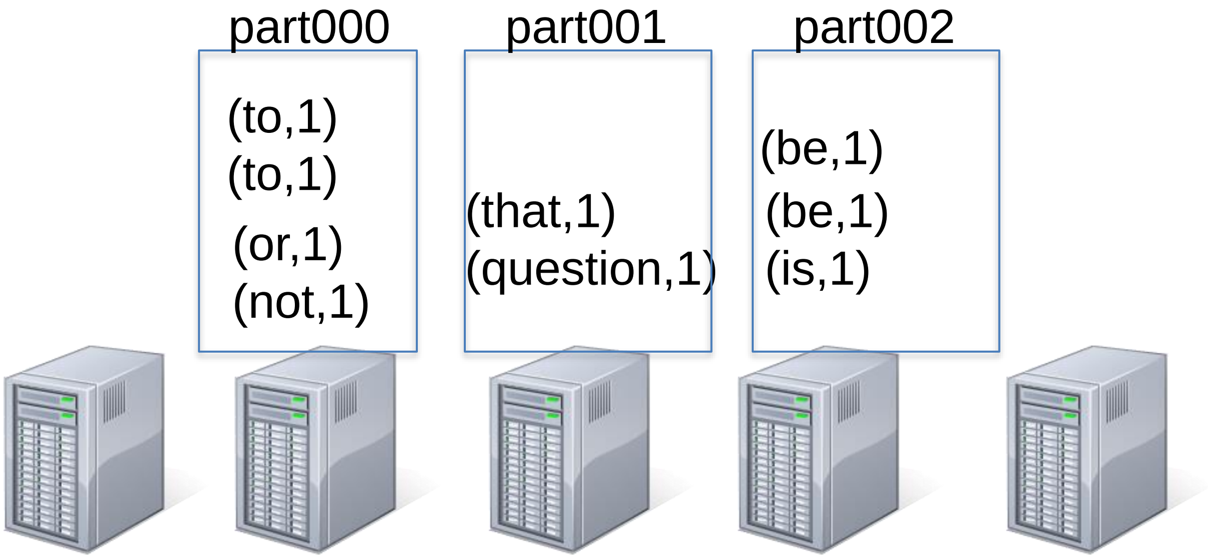
```
target_machine = hash(Key) % num_partitions
```

using python’s builtin `hash()` function

(to,1)	(or,1)	(to,1)	(that,1)	(the,1)
(be,1)	(not,1)	(be,1)	(is,1)	(question,1)

How Does `reduceByKey` Work? (simplified)

A **shuffle** takes place: key-value pairs are moved appropriate machines, collocating pairs with identical keys



Re-Partitioning a key-value pair RDD

```
words.partitionBy(numPartitions,partitionFunc=hash)



words.partitionBy(5) # create 5 partitions with default hash

words.partitionBy(5,partitionFunc=lambda x:hash(x)+10)
                # create 5 partitions with user-defined hash


somerdd.repartition(5)

                 # partition a non key-value pair rdd
```


Northeastern

Partition-Awareness

```
lines = sc.textFile("hamlet.txt")
counts = lines.flatMap(lambda line: line.split()) \
              .map(lambda word: (word, 1)) \
              .reduceByKey(lambda x, y: x + y,100)
```

rdd **not partitioned**, reduceByKey requires a shuffle

```
lines = sc.textFile("hamlet.txt")

wordsPartitioned= lines.flatMap(lambda line: line.split()) \
              .map(lambda word: (word, 1)) \
              .partitionBy(100)


counts = wordsPartitioned.reduceByKey(lambda x, y: x + y)
```

rdd **is partitioned**, reduceByKey does not shuffle!

How Does This Work?

- key-to partition map fully defined by
  - `NumPartitions`
  - `PartitionFunct` (default:hash)

- When rdd is shuffled by `partitionBy`, these are stored in private variables

- When `reduceByKey()` is called next, it checks to see if these fields are set; if so, it skips shuffling

Partition-Awareness & Joins

- key-to partition map fully defined by
  - `NumPartitions`
  - `PartitionFunct` (default:hash)

- Join of rdds with **the same partitioning information** do not require a shuffle!!!!
  - keys are already on the same machines

- Useful when doing repeated joins on large rdd

Partition-Awareness & Joins

```
grades = sc.parallelize([('Alice', 2), ('Bob', 4), …, ('Bob', 6)])\
             .partitionBy(100).cache() #Perform a shuffle, sets partition info
…
register = sc.parallelize([('Bob','UGrad'),(('Maria','Grad'))])\
             .partitionBy(100) #Perform a tiny shuffle, sets partition info

grades.join(register)        # NO SHUFFLING REQUIRED
```

`grades`

key	value
Alice	2
Bob	4
…	..
Bob	6

`register`

key	value
Bob	UGrad
Maria	Grad

`grades.join(register)`

key	value
Bob	(4,UGrad)
Bob	(6,UGrad)

Operations That Create/Preserve Partitioner

Create & Preserve

- ❑ `cogroup()`, `groupWith()`
- ❑ `join()`,`leftOuterJoin()`, `rightOuterJoin()`,
- ❑ `groupByKey()`, `reduceByKey()`, `combineByKey()`
- ❑ `partitionBy()`, `sortByKey()`

Preserve (if parent has partitioner)

- ❑ `mapValues()`, `flatMapValues()`
- ❑ `filter()`

Beware of non-key value operations!

- ❑ A `map` will **remove partitioning info**, even if it does not alter keys.