Spark with Key/Value Pairs

Lab 4
November 2th, 2017

Jun Heo(j.heo@snu.ac.kr)
Computer Science and Engineering
Seoul National University

***Slide credits**: Jonghyun Bae, Jun Heo, Jae W. Lee, Zaharia M. (Learning Spark)*

Index

- **Working with Key/Value Pairs**
  - Creating Pair RDDs
  - Transformations on Pair RDDs
    - Aggregation/Grouping Data/Join/Sorting Data
  - Actions on Pair RDDs
    - countByKey/collectAsMap
- **Shuffle Operation**
- **RDD Persistence**
- **Exercises**

Before we start…

- **Exercise 1 answer**

```
>>> text_file = sc.textFile("hdfs://localhost:9000/input")
>>> filters = text_file.flatMap(lambda line: line.split(" ")) \
...             .map(lambda word: (word, 1)) \
...             .reduceByKey(lambda a, b: a + b) \
...             .filter(lambda a: a[0] == 'cracking' or a[0] == 'bucket')
>>> wordcounts.saveAsTextFile("hdfs://localhost:9000/exercise3_1")
```

Before we start…

- **Exercise 2 answer**

```
>>> text_file = sc.textFile(“hdfs://localhost:9000/input”)
>>> uniques = text_file.flatMap(lambda line: line.split(“ “)) \
...              .map(lambda word: (word, 1)) \
...              .reduceByKey(lambda a, b: a)
>>> uniques.saveAsTextFile(“hdfs://localhost:9000/exercise3_2”)
```

Before we start…

- Please connect your VM using SSH

```
# Please your public IP address in xxx.xxx.xxx.xxx
student@computer:~$ ssh -X -i bde3.pem ubuntu@xxx.xxx.xxx.xxx
Welcome to Ubuntu 14.04.5 LTS (GNU/Linux 3.13.0-125-generic x86_64)
[...snipp...]
ubuntu@ip-x-x-x:~$
```

Associative Array

- **Associative array**
  - Abstract data type composed of a collection of (key, value) pairs
  - Key: field name, identifier
  - Value: data

- **Associative arrays as primitive data types in many programming languages**
  - Java, C++ STL, Python, Ruby, Go, Lua, …
  - HashMap, map container, dictionaries, hash tables, …

- **Key/value store**
  - Data storage designed for storing, retrieving, and managing associative array
  - Redis, Memcached, Ignite, NoSQL, Cassandra, …

https://en.wikipedia.org/wiki/Key-value_database

Key/Value Pairs in Spark

- **Key/value RDDs are used to perform aggregations**
  - count up reviews for each product
  - group together data with the same key
  - group together two different RDDs

- **Pair RDDs**
  - Spark provides special operations on RDDs containing key/value pairs
  - Operations that act on each key in parallel or regroup data across the network
  - In Python, operations work on RDDs containing built-in Python tuples

Creating Pair RDDs

- **Create RDD from text**
  - run a flatMap() function that returns key/value pairs

```
>>> lines = sc.textFile("data.txt")
>>> rdd = lines.flatMap(lambda s: s.split(" "))
>>> pairs = rdd.map(lambda s: (s, 1))
>>> pairs.collect()
(u'Sed', 1), (u'tempor', 1), (u'tincidunt', 1), ..., (u'lorem.', 1)]
```

Transformations on Pair RDDs (1)

- **We need to pass transformations that operate on tuples**
- **Transformations on one pair RDD**
  - reduceByKey, groupByKey, combineByKey, mapValues, flatMapValues, sortByKey, …
- **Transformations on two pair RDDs**
  - subtractByKey, join, cogroup, …
- **Families of pair RDD functions**
  - Aggregations / Grouping / Joins / Sorting

* https://spark.apache.org/docs/2.1.0/api/python/pyspark.html#pyspark.RDD

Aggregating Data: reduceByKey() (1)

- **Aggregate statistics across all elements with the same key**
- **`reduceByKey(func)`**
  - Merge the values for each key using an associative and commutative reduce function

key	value
apple	(0, 1)
banana	(3, 1)
cherry	(3, 1)
apple	(1, 1)
banana	(4, 1)

`reduceByKey()`

key	value
apple	(1, 2)
banana	(7, 2)
cherry	(3, 1)

* https://spark.apache.org/docs/2.1.0/api/python/pyspark.html#pyspark.RDD

Aggregating Data: reduceByKey() (2)

- **Code & output**

```
>>> data = [("a", 0), ("b", 3), ("c", 3), ("a", 1), ("b", 4)]
>>> rdd = sc.parallelize(data)
>>> pairs = rdd.mapValues(lambda x: (x, 1)) \
>>>     .reduceByKey(lambda x, y: (x[0] + y[0], x[1] + y[1]))
>>> pairs.collect()
[('a', (1, 2)), ('c', (3, 1)), ('b', (7, 2))]
```

* https://spark.apache.org/docs/2.1.0/api/python/pyspark.html#pyspark.RDD

Aggregating Data: combineByKey() (1)

- `combineByKey(func1, func2, func3)`
  - combine the elements for each key using a custom set of aggregation functions
  - createCombiner, mergeValue, mergeCombiners

Partition 0

key	value
apple	1
banana	1
apple	2
banana	3

Partition 1

key	value
apple	2
banana	3
cherry	1
cherry	4

Partition 0

key	value
apple	(1, 1)
banana	(1, 1)
apple	(2, 1)
banana	(3, 1)

Partition 1

key	value
apple	(2, 1)
banana	(3, 1)
cherry	(1, 1)
cherry	(4, 1)

Partition 0

key	value
apple	(3, 2)
banana	(4, 2)

Partition 1

key	value
apple	(2, 1)
banana	(3, 1)
cherry	(5, 2)

key	value
apple	(5, 3)
banana	(7, 3)
cherry	(5, 2)

* https://spark.apache.org/docs/2.1.0/api/python/pyspark.html#pyspark.RDD

Aggregating Data: combineByKey() (3)

- **Code & output**

```
>>> data = [("a", 1),("b", 1),("a", 2),("b", 3),("a", 2),("b", 3), \
        ("c", 1),("c", 4)]
>>> rdd = sc.parallelize(data, 2)
>>> pairs = rdd.combineByKey((lambda x: (x,1)),  \
>>>     (lambda x, y: (x[0] + y, x[1] + 1)),  \
>>>     (lambda x, y: (x[0] + y[0], x[1] + y[1])))
>>> pairs.collect()
[('a', (5, 3)), ('c', (5, 2)), ('b', (7, 3))]
```

Grouping Data: groupByKey() (1)

- **Grouping data by key**
- **`groupByKey()`**
  - Group the values for each key in the RDD into a single sequence.
  - Hash-partitions the resulting RDD with numPartitions partitions

key	value
apple	0
banana	3
cherry	2
apple	1
banana	4
cherry	3

key	value
apple	(0, 1)
banana	(3, 4)
cherry	(2, 3)

* https://spark.apache.org/docs/2.1.0/api/python/pyspark.html#pyspark.RDD

Grouping Data: groupByKey() (2)

- **Code & output**

```
>>> data = [("a", 0),("b", 3),("a", 2),("a", 1),("b", 4),("c", 3)]
>>> rdd = sc.parallelize(data)
>>> pairs = rdd.groupByKey().mapValues(list)
>>> pairs.collect()
[('a', [0, 2, 1]), ('c', [3]), ('b', [3, 4])]
```

* https://spark.apache.org/docs/2.1.0/api/python/pyspark.html#pyspark.RDD

Join (1)

- **`Join(other)`**
  - Return an RDD containing all pairs of elements with matching keys in self and other.

key	value
apple	0
banana	1

key	value
apple	1
apple	2
banana	4
banana	6

`join()`

key	value
apple	(0, 1)
apple	(0, 2)
banana	(1, 4)
banana	(1, 6)

* https://spark.apache.org/docs/2.1.0/api/python/pyspark.html#pyspark.RDD

Join (2)

- **Code & output**

```
>>> x = [("a", 0),("b", 1)]
>>> y = [("a", 1),("a", 2),("b", 4),("b", 6),("c", 5)]
>>> x1 = sc.parallelize(x)
>>> y1 = sc.parallelize(y)
>>> pairs = x1.join(y1)
>>> pairs.collect()
[('a', (0, 1)), ('a', (0, 2)), ('b', (1, 4)), ('b', (1, 6))]
```

* https://spark.apache.org/docs/2.1.0/api/python/pyspark.html#pyspark.RDD

Sorting Data: sortByKey() (1)

- `sortByKey(ascending)`
  - Sorts this RDD, which is assumed to consist of (key, value) pairs

key	value
peach	4
cherry	5
apple	10
banana	1
apple	7
melon	0

`sortByKey()`

key	value
apple	10
apple	7
banana	1
cherry	5
melon	0
peach	4

* https://spark.apache.org/docs/2.1.0/api/python/pyspark.html#pyspark.RDD

Sorting Data: sortByKey() (1)

- **Code & output**

```
>>> data = [("p", 4),("c", 5),("a", 10),("b", 1),("a", 7),("m", 0)]
>>> rdd = sc.parallelize(data)
>>> pairs1 = rdd.sortByKey(1)
>>> pairs2 = rdd.sortByKey(0)
>>> pairs1.collect()
[('a', 10), ('a', 7), ('b', 1), ('c', 5), ('m', 0), ('p', 4)]
>>> pairs2.collect()
[('p', 4), ('m', 0), ('c', 5), ('b', 1), ('a', 10), ('a', 7)]
```


* https://spark.apache.org/docs/2.1.0/api/python/pyspark.html#pyspark.RDD

Actions on Pair RDDs

- **All of the traditional actions available on the base RDD are also available on pair RDDs**
- **Some additional actions are available on pair RDDs to take advantage of the key/value nature of the data**
  - countByKey, collectAsMap, Lookup

* https://spark.apache.org/docs/2.1.0/api/python/pyspark.html#pyspark.RDD

countByKey()

- **countByKey()**
  - Count the number of elements for each key, and return the result to the master as a dictionary
- **Code & output**

```
>>> list = [("p", 4),("c", 5),("a", 10),("b", 1),("a", 7),("m", 0)]
>>> temp = sc.parallelize(list)
>>> temp.countByKey()
defaultdict(<type 'int'>, {'a': 2, 'p': 1, 'c': 1, 'b': 1, 'm': 1})
```

* https://spark.apache.org/docs/2.1.0/api/python/pyspark.html#pyspark.RDD

collectAsMap()

- **collectAsMap()**
  - Return the key-value pairs in this RDD to the master as a dictionary
- **Code & output**

```
>>> data = [("p", 4),("c", 5),("a", 10),("b", 1),("a", 7),("m", 0)]
>>> rdd = sc.parallelize(data)
>>> rdd.collectAsMap()
{'a': 7, 'p': 4, 'c': 5, 'b': 1, 'm': 0}
```

* https://spark.apache.org/docs/2.1.0/api/python/pyspark.html#pyspark.RDD

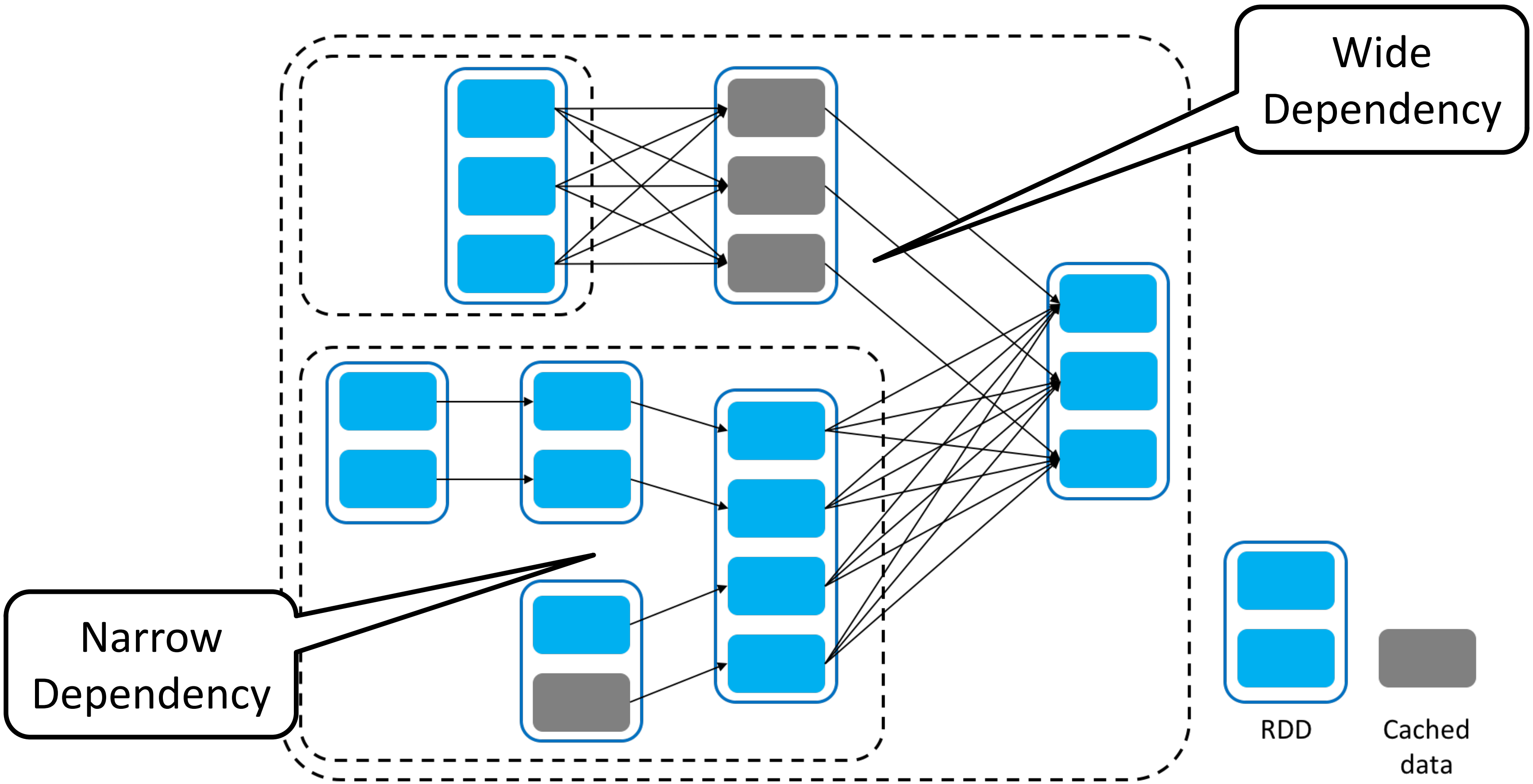
RDD Dependencies (1)

- **The shuffle is mechanism for re-distributing data so that it's grouped differently across partitions**

- **Narrow vs. wide dependencies**
  - Narrow: each partition of the parent RDD is used by at most one partition of child RDD
    - map, mapValues, flatMap, filter, mapPartitions, mapPartitionsWithIndex, …
  - Wide: each partition of the parent RDD is used by multiple child RDDs
    - join, groupByKey, reduceByKey, combineByKey, distinct, cogroup, …

RDD Dependencies (2)

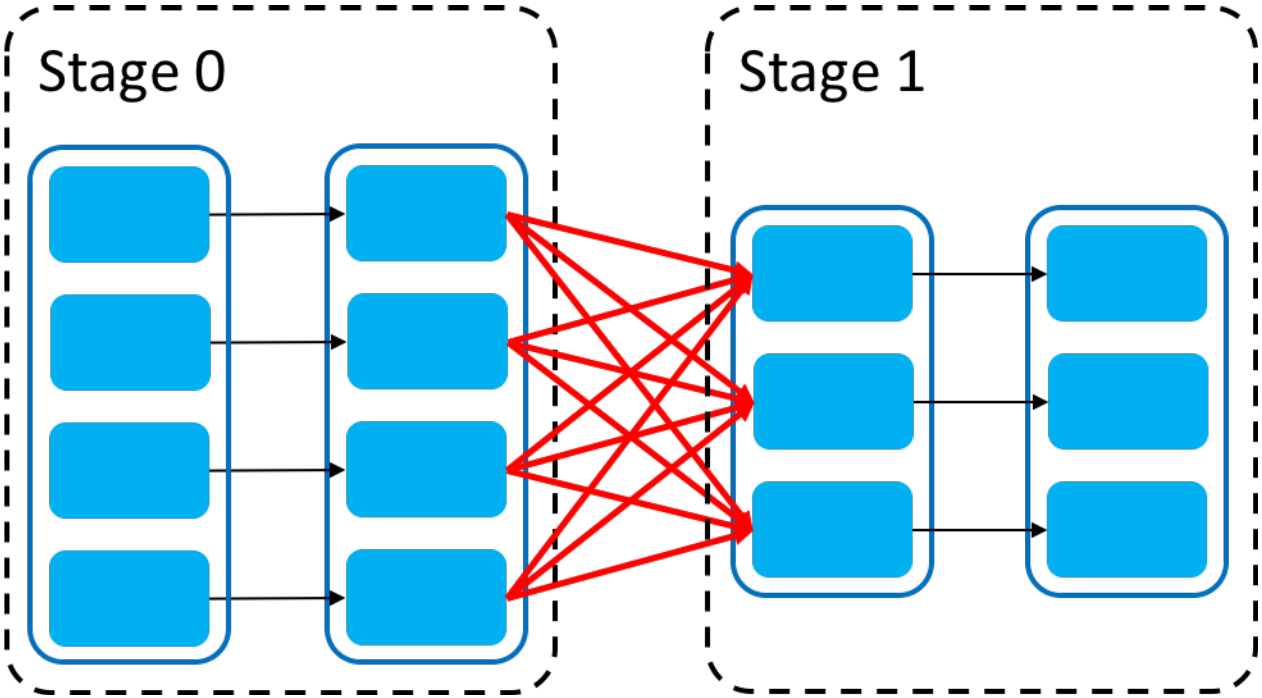
- Narrow vs. wide dependencies



* https://www.slideshare.net/aknahs/spark-16667619

Shuffle Operations (1)

- **What is the shuffle?**
  - Spark's mechanism for re-distributing data so that it's grouped differently across partitions
  - Copy data across executors and machines



https://www.slideshare.net/colorant/spark-shuffle-introduction

Shuffle Operations (2)

- **Example: `reduceByKey()`**
  - To organize all the data for a single reduceByKey reduce task to execute, Spark needs to perform an all-to-all operation
  - read from all partitions to find all the values for all keys
  - bring together values across partitions to compute the final result for each key

- **Operations which can cause a shuffle**
  - repartition operations like repartition and coalesce
  - *ByKey operations like groupByKey and reduceByKey
  - join operations like cogroup and join

RDD persistence

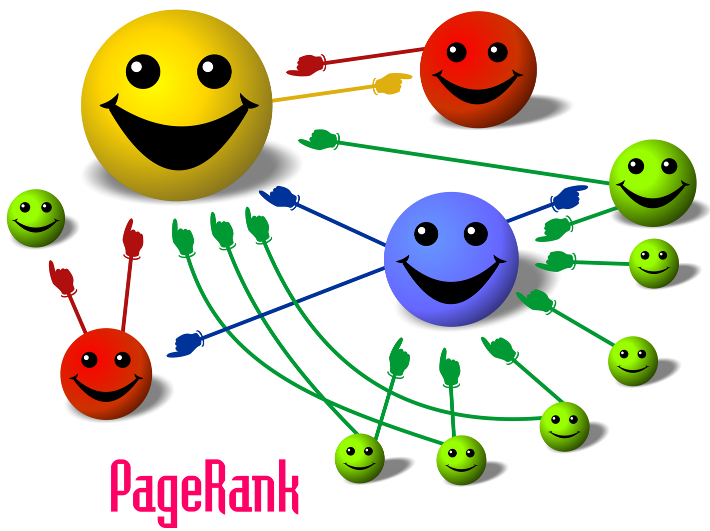
- **`persist()` or `cache()`**
  - Each node stores any partitions of it that it computes in memory and reuses them in other actions on that dataset
  - Caching is a key tool for iterative algorithms and fast interactive use
  - Spark recommends users call persist on the resulting RDD if they plan to reuse it

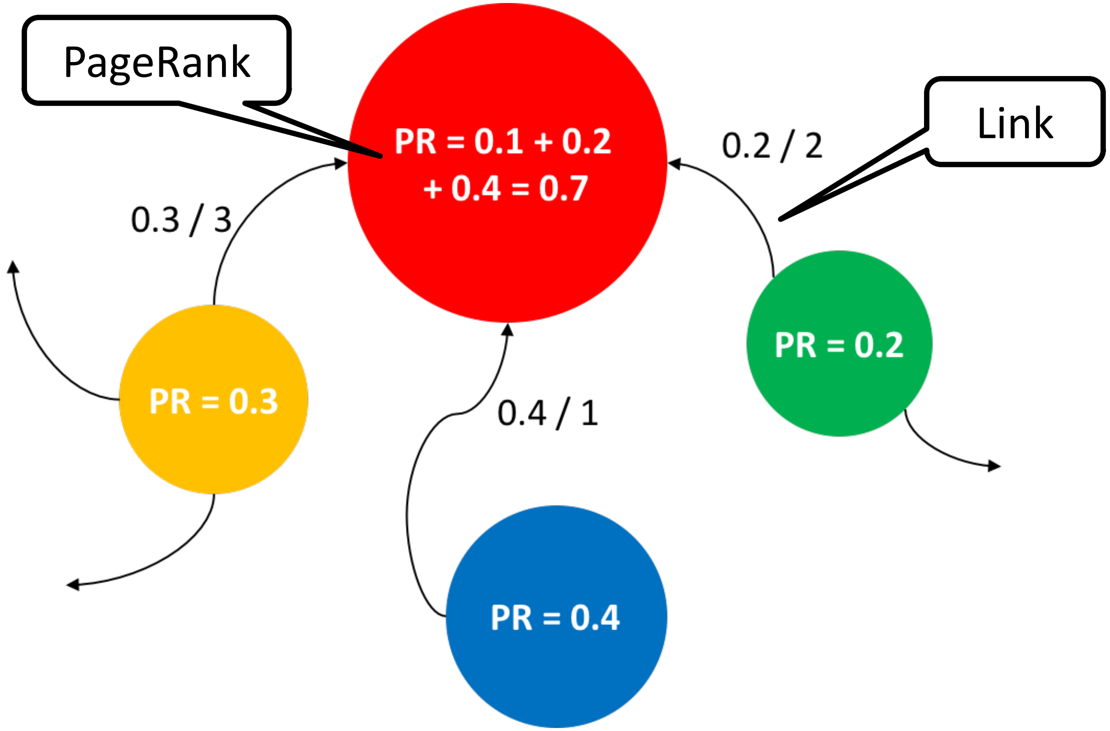
- **Storage Level**
  - MEMORY_ONLY, MEMORY_AND_DISK, DISK_ONLY
  - persist it in memory as deserialized/serialized Java objects
  - persist the dataset on disk
  - In Python, stored objects will always be serialized with the Pickle library

https://spark.apache.org/docs/2.2.0/rdd-programming-guide.html#rdd-persistence

Example: PageRank (1)

- **What is PageRank?**
  - Algorithm used by **Google Search** to rank websites in search engine
  - PageRank works by counting the number and quality of links to a page
    - Determine a rough estimate of how important the website is





* https://en.wikipedia.org/wiki/PageRank

Example: PageRank (2)

- **spark-2.1.0/examples/src/main/python/pagerank.py**

```
lines = spark.read.text(sys.argv[1]).rdd.map(lambda r: r[0])
links = lines.map(lambda urls: parseNeighbors(urls)).distinct() \
  .groupByKey().cache()
ranks = links.map(lambda url_neighbors: (url_neighbors[0], 1.0))
for iteration in range(int(sys.argv[2])):
  contribs = links.join(ranks).flatMap(
    lambda url_urls_rank: computeContribs(url_urls_rank[1][0],url_urls_rank[1][1]))
  ranks = contribs.reduceByKey(add).mapValues(lambda rank: rank * 0.85 + 0.15)
```

Example: PageRank (3)

- **Run pagerank.py**

```
ubuntu@ip-x-x-x:~/spark-2.1.0$ bin/spark-submit/examples/src/main/python/
pagerank.py data/mllib/pagerank_data.txt 100
WARN ...
[Stage 1: ================>                    (5 + 1) / 4]
```

Example: PageRank (4)

- **Check the effect of RDD cache()**

172.31.1.25:8080

Search

Spark 2.1.0 Spark Master at spark://172.31.1.25:7077

URL: spark://172.31.1.25:7077
REST URL: spark://172.31.1.25:6066 *(cluster mode)*
Alive Workers: 1
Cores in use: 1 Total, 1 Used
Memory in use: 10.0 GB Total, 10.0 GB Used
Applications: 1 Running, 1 Completed
Drivers: 0 Running, 0 Completed
Status: ALIVE

Workers

Worker Id	Address	State	Cores	Memory
worker-20171030101445-172.31.1.25-57806	172.31.1.25:57806	ALIVE	1 (1 Used)	10.0 GB (10.0 GB Used)

Running Applications

Application ID	Name	Cores	Memory per Node	Submitted Time	User	State	Duration
app-20171030101702-0001 (kill)	pagerank.py	1	10.0 GB	2017/10/30 10:17:02	ubuntu	RUNNING	3.4 min

Completed Applications

Example: PageRank (5)

- **Check the effect of RDD cache( )**

172.31.1.25:4040/jobs/

Spark 2.1.0 | Jobs | Stages | Storage | Environment | Executors | SQL | **pagerank.py** application UI

Spark Jobs (?)

User: ubuntu
Total Uptime: 3.7 min
Scheduling Mode: FIFO
Active Jobs: 1

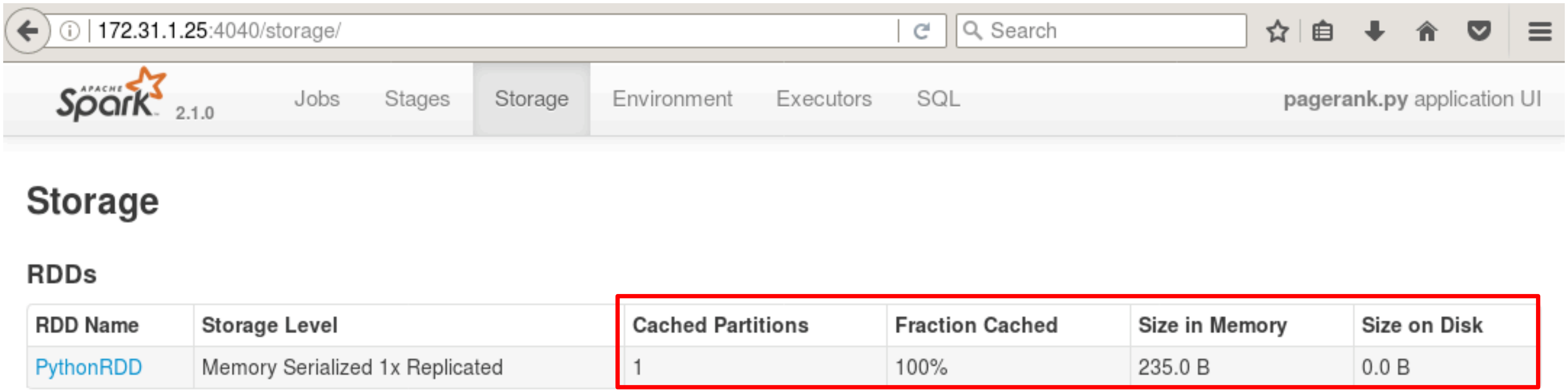
▸ Event Timeline

Active Jobs (1)

Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
0	collect at /home/ubuntu/spark-2.1.0/examples/src/main/python/pagera... (kill)	2017/10/30 10:17:09	3.6 min	114/203	3315/10403

Example: PageRank (6)

- Check the effect of RDD cache( )



Storage

RDDs

RDD Name	Storage Level	Cached Partitions	Fraction Cached	Size in Memory	Size on Disk
PythonRDD	Memory Serialized 1x Replicated	1	100%	235.0 B	0.0 B

Exercise (1)

- **Find words with 7 words and store them both in hdfs and txt (6 point)**
  - Save alphabetical order from z to a
- **Hint**
  - https://spark.apache.org/docs/2.1.0/api/python/pyspark.html#pyspark.RDD

Exercise (2)

- **In the following code, find where to apply cache( ) (4 point)**

```
lines = spark.read.text(sys.argv[1]).rdd.map(lambda r: r[0])    // 1
data = lines.map(parseVector)    // 2
kPoints = data.takeSample(False, K, 1)    // 3
while tempDist > convergeDist:
    closest = data.map(lambda p: (closestPoint(p, kPoints), (p, 1)))    // 4
    pointStats = closest.reduceByKey(
        lambda p1_c1, p2_c2: (p1_c1[0] + p2_c2[0], p1_c1[1] + p2_c2[1]))    // 5
    newPoints = pointStats.map(lambda st: (st[0], st[1][0] / st[1][1])).collect()    // 6
    tempDist = sum(np.sum((kPoints[iK] - p) ** 2) for (iK, p) in newPoints)
    for (iK, p) in newPoints:
        kPoints[iK] = p
```

- **Appendix**