ENSF 612
Lecture – PySpark Programming

Dr. Gias Uddin, Assistant Professor

Electrical and Software Engineering

University of Calgary

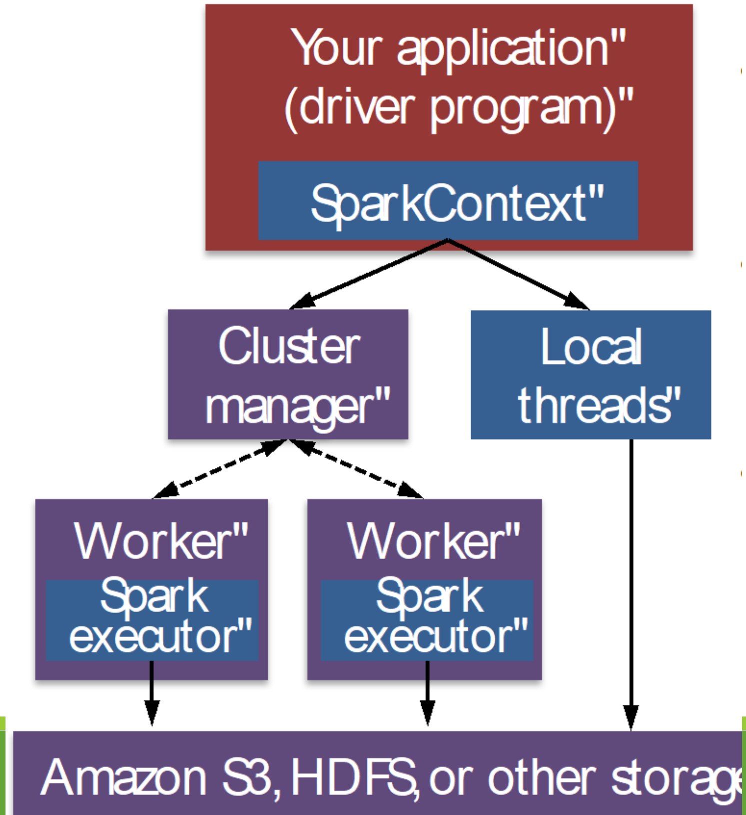
https://giasuddin.ca/

Spark programming

- Will use the Python interface to Spark pySpark
- PySpark code let us do the following:
  - “Here are some operations. Apply them to my data”
- RDDs are the key concept
  - RDD – Resilient Distributed Dataset
  - RDD is a collection of data elements partitioned across the nodes of a cluster that can be operated in parallel

Spark programming – cont'd

- Spark program is two programs
  - A **driver** program and a **workers** program
- Worker programs run on cluster nodes typically
- RDDs are distributed across workers



Spark programming – cont'd

- Initialization of Spark

```
import org.apache.spark.SparkContext
import org.apache.spark.SparkConf
```

```
conf = SparkConf().setAppName(appName).setMaster(master)
sc = SparkContext(conf=conf)
```

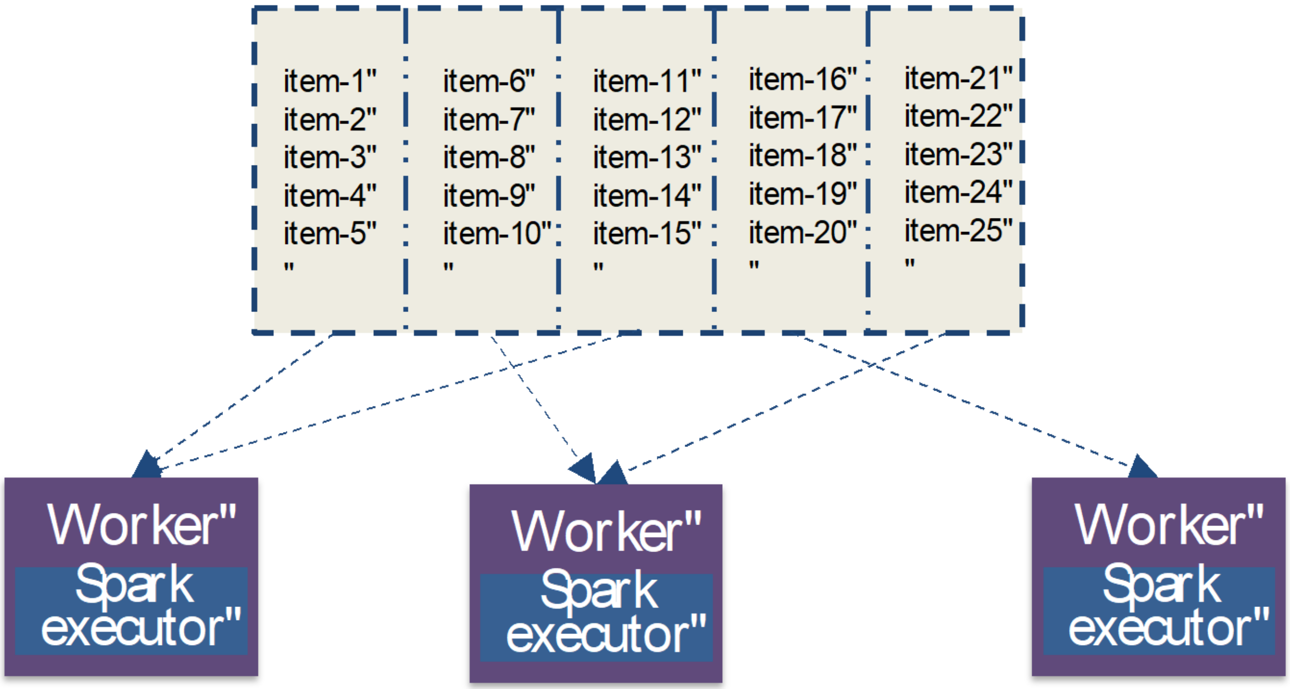
- A Spark program first creates a ***SparkContext*** object
  - Tells Spark how and where to access a cluster
  - pySpark shell automatically creates ***sc*** variable
  - iPython programs must use a constructor to create ***SparkContext***
- ***SparkContext*** object can be used to create RDDs

Spark programming – cont'd

- RDDs are immutable once they are constructed
  - Can transform them to other RDDs
  - Can perform actions on them, e.g., read elements
  - However, can't change them
- RDDs allow parallel operations on collections
- How do we construct RDDs?
  - From Python collections (lists)
  - By transforming other RDDs
  - From files stored in HDFS and other storage

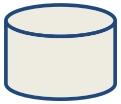
Spark programming – cont'd

- How is an RDD parallelized?
- RDD can be split into programmer specified # of partitions
- More partitions – more parallelism



Spark programming – cont'd

- Two types of operations – *transformations* and *actions*
- Transformations are lazily evaluated for efficiency
  - Transformations execute only when action performed on RDD
- RDDs can be persisted (cached) in memory or disk

Create an RDD from a data source:  <list>

Apply transformations to an RDD: map filter

Apply actions to an RDD: collect count

`collect` action causes **`parallelize`**, **`filter`**, and **`map`** transforms to be executed

Spark programming – cont'd

- Will now look at transformations and actions in Spark
- Need to review Python **lambda** functions first
- Small anonymous functions not bound to a name

lambda a,b: a+b

Returns sum of two argument a and b

- Restricted to single expressions

Spark programming – cont'd

- Frequently used Spark *transformations*

Transformation	Description
`map(`*func*`)`	return a new distributed dataset formed by passing each element of the source through a function *func*
`filter(`*func*`)`	return a new dataset formed by selecting those elements of the source on which *func* returns true
`distinct([`*numTasks*`]))`	return a new dataset that contains the distinct elements of the source dataset
`flatMap(`*func*`)`	similar to map, but each input item can be mapped to 0 or more output items (so *func* should return a Seq rather than a single item)

Spark programming – cont'd

- Example *transformations*

```
>>> rdd = sc.parallelize([1, 2, 3, 4])
>>> rdd.map(lambda x: x * 2)
RDD: [1, 2, 3, 4] → [2, 4, 6, 8]
```

Function literals (green) are closures automatically passed to workers

```
>>> rdd.filter(lambda x: x % 2 == 0)
RDD: [1, 2, 3, 4] → [2, 4]

>>> rdd2 = sc.parallelize([1, 4, 2, 2, 3])
>>> rdd2.distinct()
RDD: [1, 4, 2, 2, 3] → [1, 4, 2, 3]
```

```
>>> rdd = sc.parallelize([1, 2, 3])
>>> rdd.Map(lambda x: [x, x+5])
RDD: [1, 2, 3] → [[1, 6], [2, 7], [3, 8]]

>>> rdd.flatMap(lambda x: [x, x+5])
RDD: [1, 2, 3] → [1, 6, 2, 7, 3, 8]
```

Spark programming – cont'd

- Spark *action* causes transformations to be applied
- Mechanism for getting results out of Spark
- Frequently used *actions* in Spark

Action	Description
`reduce(`*func*`)`	aggregate dataset's elements using function *func*. *func* takes two arguments and returns one, and is commutative and associative so that it can be computed correctly in parallel
`take(`*n*`)`	return an array with the first *n* elements
`collect()`	return all the elements as an array WARNING: make sure will fit in driver program
`takeOrdered(`*n*`, `*key=func*`)`	return n elements ordered in ascending order or as specified by the optional key function

Spark programming – cont'd

- Examples of *actions*

```
>>> rdd = sc.parallelize([1, 2, 3])
>>> rdd.reduce(lambda a, b: a * b)
Value: 6

>>> rdd.take(2)
Value: [1,2] # as list

>>> rdd.collect()
Value: [1,2,3] # as list
```

```
>>> rdd = sc.parallelize([5,3,1,2])
>>> rdd.takeOrdered(3, lambda s: -1 * s)
Value: [5,3,2] # as list
```

Spark programming – cont'd

- Lets put transformations and actions together

#Create RDD that has 4 partitions
#Each element of RDD is a line of the text file
lines = sc.textFile("…",4)
#isNotComment function only outputs a line if it is not a #comment
noComments = lines.filter(isNotComment)
print lines.count()

`count()` causes Spark to:

- read data
- sum within partitions
- combine sums in driver

Spark programming – cont'd

- Spark may be forced to read data multiple times

```
#Create RDD that has 4 partitions
#Each element of RDD is a line of the text file
lines = sc.textFile("…",4)
#isNotComment function only outputs a line if it is not a #comment
noComments = lines.filter(isNotComment)
print lines.count(), noComments.count()
```

Spark recomputes lines:
- read data (again)
- sum within partitions
- combine sums in driver

Spark programming – cont'd

- **Spark can persist (cache) RDDs to be more efficient**

```
#Create RDD that has 4 partitions
#Each element of RDD is a line of the text file
lines = sc.textFile("…",4)
#save lines in memory – don't recompute!
lines.cache()
#isNotComment function only outputs a line if it is not a #comment
noComments = lines.filter(isNotComment)
print lines.count(), noComments.count()
```

Spark programming – cont'd

- Lifecycle of a Spark program
- Create RDDs from external data or Python collections
- Lazily transform them into new RDDs
- Cache some RDDs for reuse
- Perform actions to trigger parallel computations and results

Spark programming – cont'd

- Similar to MapReduce, Spark supports Key-Value pairs
- Spark allows **pair RDDs** to be created
- Each element is a tuple consisting of a key and value

```
>>> rdd = sc.parallelize([(1, 2), (3, 4)])
RDD: [(1, 2), (3, 4)]
```

Spark programming – cont'd

- Examples of Key-Value transformations

Key-Value Transformation	Description
`reduceByKey(`*`func`*`)`	return a new distributed dataset of (K,V) pairs where the values for each key are aggregated using the given reduce function *func*, which must be of type (V,V) ➔ V
`sortByKey()`	return a new dataset (K,V) pairs sorted by keys in ascending order
`groupByKey()`	return a new dataset of (K, Iterable<V>) pairs

Spark programming – cont'd

- Examples of Key-Value transformations

```
>>> rdd = sc.parallelize([(1,2), (3,4), (3,6)])
>>> rdd.reduceByKey(lambda a, b: a + b)
RDD: [(1,2), (3,4), (3,6)] → [(1,2), (3,10)]

>>> rdd2 = sc.parallelize([(1,'a'), (2,'c'), (1,'b')])
>>> rdd2.sortByKey()
RDD: [(1,'a'), (2,'c'), (1,'b')] →
             [(1,'a'), (1,'b'), (2,'c')]

>>> rdd2.groupByKey()
RDD: [(1,'a'), (1,'b'), (2,'c')] →
             [(1,['a','b']), (2,['c'])]
```

Be careful using groupByKey() – can cause a lot of data movement across the network!

Spark programming – cont'd

- Spark supports working with structured data
- Supports ***join*** operations on pair RDDs
- **X.join(Y)**
  - Return RDD of all pairs of elements with matching keys in X and Y
  - Each pair is (k,(v1,v2)) tuple, where (k, V1) is in X and (k,V2) is in Y

```
>>> x = sc.parallelize([("a", 1), ("b", 4)])
>>> y = sc.parallelize([("a", 2), ("a", 3)])
>>> sorted(x.join(y).collect())

Value: [('a', (1, 2)), ('a', (1, 3))]
```

Spark programming – cont'd

- **X.leftOuterJoin(Y)**
  - For each element (k,v) in X, resulting RDD will either contain
    - All pairs of (k,(v,w)) for w in Y
    - Or the pair (k,(v,None)) if no element in Y has key k

```
>>> x = sc.parallelize([("a", 1), ("b", 4)])
>>> y = sc.parallelize([("a", 2)])
>>> sorted(x.leftOuterJoin(y).collect())

Value: [('a', (1, 2)), ('b', (4, None))]
```

Spark programming – cont'd

- **Y.rightOuterJoin(X)**
  - For each element (k,w) in X, resulting RDD will either contain
    - All pairs of (k,(v,w)) for v in Y
    - Or the pair (k,(None,w)) if no element in Y has key k

```
>>> x = sc.parallelize([("a", 1), ("b", 4)])
>>> y = sc.parallelize([("a", 2)])
>>> sorted(y.rightOuterJoin(x).collect())

Value: [('a', (2, 1)), ('b', (None, 4))]
```

Spark programming – cont'd

- **Y.fullOuterJoin(X)**
  - For each element (k,v) in X, resulting RDD will either contain
    - All pairs (k,(v,w)) for w in Y, or (k,(v,None)) if no elements in Y have k
  - For each element (k,w) in Y, resulting RDD will either contain
    - All pairs (k,(v,w)) for v in X, or (k,(None,w)) if no elements in X have k

```
>>> x = sc.parallelize([("a", 1), ("b", 4)])
>>> y = sc.parallelize([("a", 2), ("c", 8)])
>>> sorted(x.fullOuterJoin(y).collect())

Value: [('a', (1, 2)), ('b', (4, None)) , ('c', (None, 8))]
```

Spark programming – cont'd

- Spark **shared variables**
- Spark automatically creates closures for:
  - Functions that run on RDDs at workers
  - Global variables used by those workers
- One closure per worker
  - Sent for every task
  - No communication between workers
  - Changes to global variables at workers not sent to drivers
- **This model may be inefficient for many use cases**

Spark programming – cont'd

- Consider following 2 use cases
- Iterative or single jobs with large global variables
  - Sending large lookup table to workers
  - Sending large feature vectors in a ML algorithm to workers
- Counting events that occur during job execution
  - E.g., How many input lines were blank?
  - E.g., How many input records were corrupt?
- Problems
  - Closures are (re)sent with every job
  - Inefficient to send large data to each worker
  - Closures are one way – driver→worker

Spark programming – cont'd

- **Broadcast** and **accumulator** variables address this
- Broadcast variables
  - Efficiently send large, **read-only** values to all workers
  - Saved at workers for use in one or more Spark operations
- Accumulators
  - Aggregate values from workers back to the driver
  - Only driver can access the value of accumulator
  - For tasks, accumulators are **write-only**

Spark programming – cont'd

- Broadcast variables
  - Ship to each worker only once instead of with each task
  - Efficiently give each worker a large dataset
  - Distributed using efficient broadcast algorithms

```
At the driver:
>>> broadcastVar = sc.broadcast([1, 2, 3])

At a worker (in code passed via a closure)
>>> broadcastVar.value4
[1, 2, 3]
```

Spark programming – cont'd

- Broadcast variables example

Country code lookup for HAM radio call signs"

```
#,Lookup,the,locations,of,the,call,signs,on,the,
#,RDD,contactCounts.,We,load,a,list,of,call,sign,,
#,prefixes,to,country,code,to,support,this,lookup,,
signPrefixes = loadCallSignTable()

def4processSignCount(sign_count, signPrefixes):
    country = lookupCountry(sign_count[0], signPrefixes)
    count = sign_count[1]
    return4(country, count)

countryContactCounts = (contactCounts
                        .map(processSignCount)
                        .reduceByKey((lambda4x, y: x+ y)))
```

Expensive to send large table"
(Re-)sent for every processed file "

From: http://shop.oreilly.com/product/0636920028512.do "

Spark programming – cont'd

- Broadcast variables example

```
#,Lookup,the,locations,of,the,call,signs,on,the,
#,RDD,contactCounts.,We,load,a,list,of,call,sign,,
#,prefixes,to,country,code,to,support,this,lookup,,
signPrefixes = sc.broadcast(loadCallSignTable())

def4processSignCount(sign_count, signPrefixes):
    country = lookupCountry(sign_count[0], signPrefixes.value)
    count = sign_count[1]
    return4(country, count)

countryContactCounts = (contactCounts
                        .map(processSignCount)
                        .reduceByKey((lambda4x, y: x+ y)))
```

Efficiently sent once to workers"

From: http://shop.oreilly.com/product/0636920028512.do "

Spark programming – cont'd

- Accumulators
  - Variables that can only be "added" to by associative operation
  - Used to efficiently implement parallel counters and sum
  - Only driver can read accumulator value – not tasks

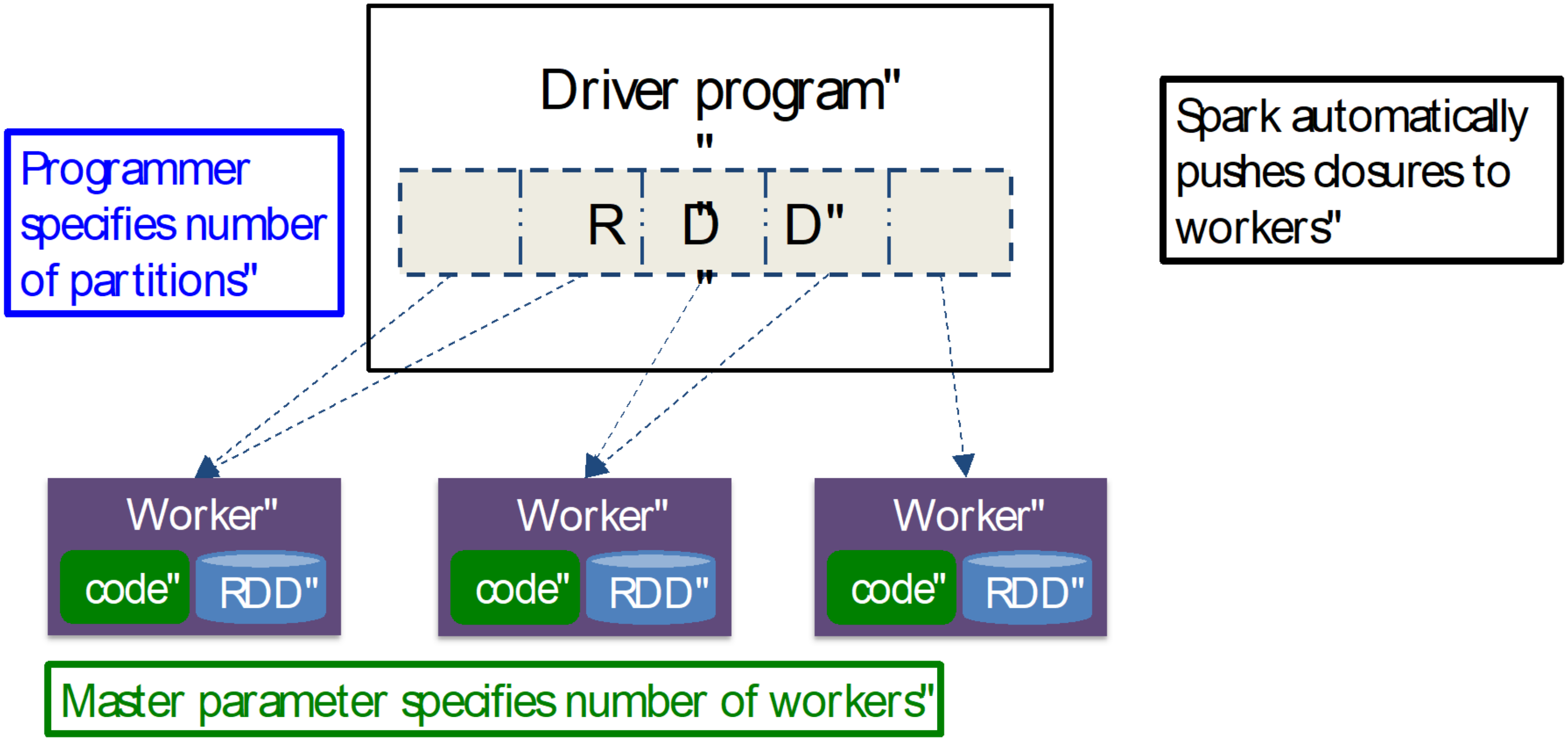
```
>>> accum = sc.accumulator(0)
>>> rdd = sc.parallelize([1, 2, 3, 4])
>>> def f(x):
>>>   global accum
>>>   accum += x

>>> rdd.foreach(f)
>>> accum.value
Value: 10
```

Spark programming – cont'd

- Accumulators
  - Tasks at workers can't access accumulator's values
  - Tasks see accumulators as write-only variables
  - Can be used in actions or transformations
    - Actions – each update to accumulator applied only once
    - Transformations – no such guarantees! (why?)

Spark programming – cont'd

- Summary of Spark programming model



Acknowledgements

Portions of these slides were adapted from external material available under creative commons license CC-BY-NC-SA 4.0. This license grants the ability to share and adapt the material for non-commercial purposes.

Name of the creator: Dr. Anthony Joseph, University of Berkeley and team

Link to material: https://courses.edx.org/courses/BerkeleyX/CS100.1x/1T2015/info