Introduction to Big Data with Apache Spark
This Lecture

Programming Spark

Resilient Distributed Datasets (RDDs)

Creating an RDD

Spark Transformations and Actions

Spark Programming Model
Python Spark (pySpark)

• We are using the Python programming interface to Spark (pySpark)

• pySpark provides an easy-to-use programming abstraction and parallel runtime:
  » “Here’s an operation, run it on all of the data”

• RDDs are the key concept
Spark Driver and Workers

- A Spark program is two programs:
  - A **driver program** and a **workers program**
- Worker programs run on cluster nodes or in local threads
- RDDs are distributed across workers
Spark Context

• A Spark program first creates a **SparkContext** object
  » Tells Spark how and where to access a cluster
  » pySpark shell and Databricks Cloud automatically create the **sc** variable
  » iPython and programs must use a constructor to create a new **SparkContext**

• **Use SparkContext** to create RDDs

*In the labs, we create the SparkContext for you*
Spark Essentials: Master

- The **master** parameter for a **SparkContext** determines which type and size of cluster to use.

<table>
<thead>
<tr>
<th>Master Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>local</code></td>
<td>run Spark locally with one worker thread (no parallelism)</td>
</tr>
<tr>
<td><code>local[K]</code></td>
<td>run Spark locally with K worker threads (ideally set to number of cores)</td>
</tr>
<tr>
<td><code>spark://HOST:PORT</code></td>
<td>connect to a Spark standalone cluster; PORT depends on config (7077 by default)</td>
</tr>
<tr>
<td><code>mesos://HOST:PORT</code></td>
<td>connect to a Mesos cluster; PORT depends on config (5050 by default)</td>
</tr>
</tbody>
</table>

In the labs, we set the master parameter for you
Resilient Distributed Datasets

• The primary abstraction in Spark
  » Immutable once constructed
  » Track lineage information to efficiently recompute lost data
  » Enable operations on collection of elements in parallel

• You construct RDDs
  » by parallelizing existing Python collections (lists)
  » by transforming an existing RDDs
  » from files in HDFS or any other storage system
RDDs

- Programmer specifies number of partitions for an RDD

(Default value used if unspecified)

more partitions = more parallelism
RDDs

- Two types of operations: *transformations* and *actions*
- Transformations are lazy (*not computed immediately*)
- Transformed RDD is executed when action runs on it
- Persist (cache) RDDs in memory or disk
Working with RDDs

- Create an RDD from a data source:
- Apply transformations to an RDD: map, filter
- Apply actions to an RDD: collect, count

```
<list>
parallelize

RDD → filtered RDD → mapped RDD

map
filter

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

• http://spark.apache.org/docs/latest/programming-guide.html

• http://spark.apache.org/docs/latest/api/python/index.html
Creating an RDD

• Create RDDs from Python collections (lists)

```python
>>> data = [1, 2, 3, 4, 5]
>>> data
[1, 2, 3, 4, 5]

>>> rDD = sc.parallelize(data, 4)
>>> rDD
ParallelCollectionRDD[0] at parallelize at PythonRDD.scala:229
```

No computation occurs with `sc.parallelize()`
• Spark only records how to create the RDD with four partitions
Creating RDDs

- From HDFS, text files, Hypertable, Amazon S3, Apache Hbase, SequenceFiles, any other Hadoop InputFormat, and directory or glob wildcard: /data/201404*

```python
>>> distFile = sc.textFile("README.md", 4)
>>> distFile

MappedRDD[2] at textFile at
    NativeMethodAccessorImpl.java:-2
```
Creating an RDD from a File

distFile = sc.textFile("...", 4)

- RDD distributed in 4 partitions
- Elements are lines of input
- *Lazy evaluation* means no execution happens now
Spark Transformations

• Create new datasets from an existing one

• Use *lazy evaluation*: results not computed right away – instead Spark remembers set of transformations applied to base dataset
  » Spark optimizes the required calculations
  » Spark recovers from failures and slow workers

• Think of this as a recipe for creating result
Some Transformations

<table>
<thead>
<tr>
<th>Transformation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>map(func)</code></td>
<td>return a new distributed dataset formed by passing each element of the source through a function <code>func</code></td>
</tr>
<tr>
<td><code>filter(func)</code></td>
<td>return a new dataset formed by selecting those elements of the source on which <code>func</code> returns true</td>
</tr>
<tr>
<td><code>distinct([numTasks])</code></td>
<td>return a new dataset that contains the distinct elements of the source dataset</td>
</tr>
<tr>
<td><code>flatMap(func)</code></td>
<td>similar to map, but each input item can be mapped to 0 or more output items (so <code>func</code> should return a <code>Seq</code> rather than a single item)</td>
</tr>
</tbody>
</table>
Review: Python `lambda` Functions

- Small anonymous functions (not bound to a name)
  
  `lambda a, b: a + b`
  
  returns the sum of its two arguments

- Can use lambda functions wherever function objects are required

- Restricted to a single expression
Transformations

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

>>> 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]
```

Function literals (green) are closures automatically passed to workers.
Transformations

```python
>>> 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]
```

Function literals (green) are closures automatically passed to workers
Transforming an RDD

```python
textFile
```

```python
lines = sc.textFile("...", 4)
```

```python
comments = lines.filter(isComment)
```

Lazy evaluation means nothing executes – Spark saves recipe for transforming source
Spark Actions

- Cause Spark to execute recipe to transform source
- Mechanism for getting results out of Spark
## Some Actions

<table>
<thead>
<tr>
<th>Action</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>reduce(func)</code></td>
<td>aggregate dataset’s elements using function <code>func</code>. <code>func</code> takes two arguments and returns one, and is commutative and associative so that it can be computed correctly in parallel</td>
</tr>
<tr>
<td><code>take(n)</code></td>
<td>return an array with the first <code>n</code> elements</td>
</tr>
<tr>
<td><code>collect()</code></td>
<td>return all the elements as an array</td>
</tr>
<tr>
<td><code>takeOrdered(n, key=func)</code></td>
<td>return <code>n</code> elements ordered in ascending order or as specified by the optional key function</td>
</tr>
</tbody>
</table>

**WARNING:** make sure will fit in driver program.
Getting Data Out of RDDs

```python
>>> 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
```
Getting Data Out of RDDs

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

lines = `sc.textFile("...", 4)`

print lines.count()

`count()` causes Spark to:
- read data
- sum within partitions
- combine sums in driver
Spark Programming Model

```python
lines = sc.textFile("...", 4)
comments = lines.filter(isComment)
print lines.count(), comments.count()
```

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

```python
lines = sc.textFile("...", 4)
lines.cache()  # save, don't recompute!
comments = lines.filter(isComment)
print lines.count(), comments.count()
```

Spark Program Lifecycle

1. Create RDDs from external data or parallelize a collection in your driver program
2. Lazily transform them into new RDDs
3. cache() some RDDs for reuse
4. Perform actions to execute parallel computation and produce results
Spark Key-Value RDDs

- Similar to Map Reduce, Spark supports Key-Value pairs
- Each element of a Pair RDD is a pair tuple

```python
>>> rdd = sc.parallelize([(1, 2), (3, 4)])
RDD: [(1, 2), (3, 4)]
```
Some Key-Value Transformations

<table>
<thead>
<tr>
<th>Key-Value Transformation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>reduceByKey(func)</code></td>
<td>return a new distributed dataset of <code>(K,V)</code> pairs where the values for each key are aggregated using the given reduce function <code>func</code>, which must be of type <code>(V,V) \rightarrow V</code></td>
</tr>
<tr>
<td><code>sortByKey()</code></td>
<td>return a new dataset <code>(K,V)</code> pairs sorted by keys in ascending order</td>
</tr>
<tr>
<td><code>groupByKey()</code></td>
<td>return a new dataset of <code>(K,Iterable&lt;V&gt;)</code> pairs</td>
</tr>
</tbody>
</table>
Key-Value Transformations

```python
>>> 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')]
```
Key-Value Transformations

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

Be careful using `groupByKey()` as it can cause a lot of data movement across the network and create large Iterables at workers.
pySpark Closures

- Spark automatically creates closures for:
  - Functions that run on RDDs at workers
  - Any global variables used by those workers

- One closure per worker
  - Sent for *every* task
  - No communication between workers
  - Changes to global variables at workers are not sent to driver
Consider These Use Cases

• Iterative or single jobs with large global variables
  » Sending large read-only lookup table to workers
  » Sending large feature vector in a ML algorithm to workers

• Counting events that occur during job execution
  » How many input lines were blank?
  » How many input records were corrupt?
Consider These Use Cases

• Iterative or single jobs with large global variables
  » Sending large read-only lookup table to workers
  » Sending large feature vector in a ML algorithm to workers

• Counting events that occur during job execution
  » How many input lines were blank?
  » 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
pySpark Shared Variables

Broadcast Variables
- Efficiently send large, read-only value to all workers
- Saved at workers for use in one or more Spark operations
- Like sending a large, read-only lookup table to all the nodes

Accumulators
- Aggregate values from workers back to driver
- Only driver can access value of accumulator
- For tasks, accumulators are write-only
- Use to count errors seen in RDD across workers
Broadcast Variables

- Keep *read-only* variable cached on workers
  - Ship to each worker only once instead of with each task
- Example: efficiently give every worker a large dataset
- Usually distributed using efficient broadcast algorithms

At the driver:

```python
>>> broadcastVar = sc.broadcast([1, 2, 3])
```

At a worker (in code passed via a closure)

```python
>>> broadcastVar.value
[1, 2, 3]
```
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()

def processSignCount(sign_count, signPrefixes):
    country = lookupCountry(sign_count[0], signPrefixes)
    count = sign_count[1]
    return (country, count)

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

From: http://shop.oreilly.com/product/0636920028512.do
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 = sc.broadcast(loadCallSignTable())

def processSignCount(sign_count, signPrefixes):
    country = lookupCountry(sign_count[0], signPrefixes.value)
    count = sign_count[1]
    return (country, count)

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

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

- Variables that can only be “added” to by associative op
- Used to efficiently implement parallel counters and sums
- Only driver can read an accumulator’s value, not tasks

```python
>>> 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
```
Accumulators Example

- Counting empty lines

```python
def extractCallSigns(line):
    global blankLines # Make the global variable accessible
    if (line == ''):
        blankLines += 1
    return line.split(" ")

callSigns = file.flatMap(extractCallSigns)
print "Blank lines: %d" % blankLines.value
```

```plaintext
file = sc.textFile(inputFile)
# Create Accumulator[Int] initialized to 0
blankLines = sc.accumulator(0)
```
Accumulators

- Tasks at workers cannot access accumulator’s values
- Tasks see accumulators as write-only variables
- Accumulators can be used in actions or transformations:
  - Actions: each task’s update to accumulator is **applied only once**
  - Transformations: **no guarantees** (use only for debugging)
- Types: integers, double, long, float
  - See lab for example of custom type
Summary

- Programmer specifies number of partitions
- Driver program
  - Spark automatically pushes closures to workers
- Worker code
  - Master parameter specifies number of workers
  - RDD