CSEP 524 – Parallel Computation
University of Washington

Lecture 8: Data Analytics II/Spark

Michael Ringenburg
Spring 2015
Logistics

• First few presentations at end of class today
• Rest of you will be during the next two classes
  – Please send me your slides the night before
• Today is the last lecture … Continueing our data analytics exploration with a deep dive on Spark
Spark Execution Model

- "Master-slave" parallelism model
- Driver (master)
  - Executes main
  - Distributes data and work to executors
- Resilient Distributed Dataset (RDD)
  - Spark's primary data abstraction
  - Partitioned amongst executors
  - Fault-tolerant via lineage
- Executors (slaves)
  - Lazily execute tasks (operations on partitions of the RDD)
  - Global all-to-all shuffle (with barrier) for data exchange
RDD In Depth

• Primary data abstraction of Spark
  – (Prior to new DataFrames in 1.3)
• Five parts (two of which are optional)
  – Set of partitions
  – List of dependencies ("parent RDDs")
  – Function to compute my partition from my parents
  – Method of compute partitioning of data (optional)
  – Preferred location for each partition (e.g., HDFS block location)
• Notice that the RDDs contain a description of the data and computation, but not the actual data ... We will see why soon ...
val arr1M = Array.range(1,1000001)
val rdd1M = sc.parallelize(arr1M, 8)
val evens = rdd1M.filter(
a => (a%2) == 0
)
evens.take(5)

>>> Array[Int] = Array(2, 4, 6, 8, 10)
Spark Programming: Simple Example

Create array of \{1, 2, ..., 1,000,000\}

```
val arr1M = Array.range(1,1000001)
val rdd1M = sc.parallelize(arr1M, 8)
val evens = rdd1M.filter(
  a => (a%2) == 0
)
evens.take(5)

>>> Array[Int] = Array(2, 4, 6, 8, 10)
```
Spark Programming: Simple Example

Create array of 
{1, 2, ..., 1,000,000}

val arr1M = Array.range(1,1000001)
val rdd1M = sc.parallelize(arr1M, 8)
val evens = rdd1M.filter(
  a => (a%2) == 0
)
evens.take(5)

>>> Array[Int] = Array(2, 4, 6, 8, 10)

Partition array into a 8-partition RDD distributed across executor nodes. (Can also create from file.)
Spark Programming: Simple Example

- Create array of \{1, 2, \ldots, 1,000,000\}
- Partition array into a 8-partition RDD distributed across executor nodes. (Can also create from file.)
- Filter: example of a Spark transformation (create new RDD from old RDD). Filter keeps data for which the argument evaluates to true.

```scala
val arr1M = Array.range(1,1000001)
val rdd1M = sc.parallelize(arr1M, 8)
val evens = rdd1M.filter(
  a => (a % 2) == 0
)
evens.take(5)
```

```python
>>> Array[Int] = Array(2, 4, 6, 8, 10)
```
Spark Programming: Simple Example

Create array of \{1, 2, \ldots, 1,000,000\}

Partition array into a 8-partition RDD distributed across executor nodes. (Can also create from file.)

Filter: example of a Spark transformation (create new RDD from old RDD). Filter keeps data for which the argument evaluates to true.

Spark action (return result to driver)

val arr1M = Array.range(1,1000001)
val rdd1M = sc.parallelize(arr1M, 8)
val evens = rdd1M.filter(
    a => (a%2) == 0
)
evens.take(5)

>>> Array[Int] = Array(2, 4, 6, 8, 10)
Spark Programming: Simple Example

Create array of \{1, 2, \ldots, 1,000,000\}

Partition array into a 8-partition RDD distributed across executor nodes. (Can also create from file.)

Filter: example of a Spark transformation (create new RDD from old RDD). Filter keeps data for which the argument evaluates to true.

Spark action (return result to driver)

```scala
val arr1M = Array.range(1,1000001)
val rdd1M = sc.parallelize(arr1M, 8)
val evens = rdd1M.filter(
  a => (a%2) == 0
)
evens.take(5)
```

```scala
evens.take(5)
```

```scala
>>> Array[Int] = Array(2, 4, 6, 8, 10)
```

Lazy Evaluation: No computation until result requested
Example: Line-by-line

Conceptually ...

Driver:
{1, ..., 1,000,000}

Executor 0:
Executor 1:
Executor 2:
Executor 3:

val arr1M = Array.range(1,1000001)
Example: Line-by-line

```scala
val rdd1M = sc.parallelize(arr1M, 8)
```
Example: Line-by-line

Conceptually ...

Driver:
\{1, \ldots, 1,000,000\}

Executor 0:
\{2,4, \ldots 125000\}
\{500002,500004 \ldots\}

Executor 1:
\{125002, 125004 \ldots\}
\{625002, 625004 \ldots\}

Executor 2:
\{250000,250002 \ldots\}
\{750002,750004 \ldots\}

Executor 3:
\{375002,375004 \ldots\}
\{875002,875004 \ldots\}

val evens = rdd1M.filter(a => a%2==0)
Example: Line-by-line

Driver:
{1, ..., 1,000,000}
{2, 4, 6, 8, 10}

Executor 0:
{2, 4, ..., 125000}
{500002, 500004, ...}

Executor 1:
{125002, 125004, ...}
{625002, 625004, ...}

Executor 2:
{250000, 250002, ...}
{750002, 750004, ...}

Executor 3:
{375002, 375004, ...}
{875002, 875004, ...}

Conceptually ...

evens.take(5)
Example: Line-by-line

Now let's try it out ...
What's going on here?

Lazy Evaluation:

Driver:
\{1, \ldots, 1,000,000\}

Executor 0:

Executor 1:

Executor 2:

Executor 3:

\texttt{val arr1M = Array.range(1,1000001)}
What's going on here?

**Lazy Evaluation:**

- Driver: \{1, \ldots, 1,000,000\}
- Executor 0:
- Executor 1:
- Executor 2:
- Executor 3:
- RDD Partition 0
- RDD Partition 1
- RDD Partition 2
- RDD Partition 3
- RDD Partition 4
- RDD Partition 5
- RDD Partition 6
- RDD Partition 7

**Input:** Arr1M

**DAG (Directed Acyclic Graph) schedule**

```
val rdd1M = sc.parallelize(arr1M, 8)
```
What's going on here?

Lazy Evaluation:

Driver:
{1, ..., 1,000,000}

Executor 0:
Executor 1:
Executor 2:
Executor 3:

Input: Arr1M

RDD Partition 0
: RDD Partition 7
FilteredRDD 0
: FilteredRDD 7

val evens = rdd1M.filter(a => a%2==0)

DAG (Directed Acyclic Graph) schedule
What's going on here?

Lazy Evaluation:

Driver:
{1, ..., 1,000,000}

Executor 0:

Executor 1:

Executor 2:

Executor 3:

Input: Arr1M

RDD Partition 0

RDD Partition 7

FilteredRDD 0

FilteredRDD 7

Driver:
{1, ..., 1,000,000}

evens.take(5)

DAG (Directed Acyclic Graph) schedule

Spring 2015

UW CSEP 524 (PMP Parallel Computation): Ringenburg
What's going on here?

Lazy Evaluation:

Start computing!

Driver: 
{1, ..., 1,000,000}

Executor 0: 
{1 ... 125000}

Executor 1:

Executor 2:

Executor 3:

Input: Arr1M

RDD Partition 0

RDD Partition 7

RDD Partition 7

FilteredRDD 0

FilteredRDD 7

Take Result: 
RETURNS DATA

evens.take(5)
What's going on here?

Lazy Evaluation:

Driver:
{1, ..., 1,000,000}

Executor 0:
{2,4, ... 125000}

Executor 1:

Executor 2:

Executor 3:

Input: Arr1M

RDD Partition 0

RDD Partition 7

FilteredRDD 0

FilteredRDD 7

Take Result: RETURNS DATA

evens.take(5)

DAG (Directed Acyclic Graph) schedule
What's going on here?

Lazy Evaluation:

Driver:
\{1, \ldots, 1,000,000\}
\{2, 4, 6, 8, 10\}

Executor 0:
\{2,4, \ldots, 125000\}

Executor 1:

Executor 2:

Executor 3:

Input: Arr1M

RDD Partition 0

\vdots

RDD Partition 7

FilteredRDD 0

\vdots

FilteredRDD 7

Driver:
\{1, \ldots, 1,000,000\}
\{2, 4, 6, 8, 10\}

Take Result:
RETURNS DATA

evens.take(5)

DAG (Directed Acyclic Graph) schedule
Modified example

val arr1M = Array.range(1,1000001)
val rdd1M = sc.parallelize(arr1M, 8)
val evens = rdd1M.filter(a => (a%2) == 0)
val firstFiveEvens = evens.take(5)
  // How many evens?
val totalEvens = evens.count()
  // Sum of evens
val evenSum = evens.reduce((a,b) => a+b)

• Imagine we want to perform a number of operations on our filtered RDD of even integers.
• For each action, Spark will compute the DAG steps...
Modified example

• Imagine we want to perform a number of operations on our filtered RDD of even integers.
• For each action, Spark will compute the DAG steps...

```scala
val arr1M = Array.range(1,1000001)
val rdd1M = sc.parallelize(arr1M, 8)
val evens = rdd1M.filter(a => (a%2) == 0)
val firstFiveEvens = evens.take(5)
// How many evens?
val totalEvens = evens.count()
// Sum of evens
val evenSum = evens.reduce((a,b) => a+b)
```

Count returns the total size (# elems) of an RDD.
• Imagine we want to perform a number of operations on our filtered RDD of even integers.
• For each action, Spark will compute the DAG steps...

```scala
val arr1M = Array.range(1,1000001)
val rdd1M = sc.parallelize(arr1M, 8)
val evens = rdd1M.filter(a => (a%2) == 0)
val firstFiveEvens = evens.take(5)

// How many evens?
val totalEvens = evens.count()

// Sum of evens
val evenSum = evens.reduce((a,b) => a+b)
```

Count returns the total size (# elems) of an RDD.
Reduce performs a reduction over the dataset, combining elements with the argument function.
Modified example

```
val arr1M = Array.range(1,1000001)
val rdd1M = sc.parallelize(arr1M, 8)
val evens = rdd1M.filter(a => (a%2) == 0)
val firstFiveEvens = evens.take(5)
// How many evens?
val totalEvens = evens.count()
// Sum of evens
val evenSum = evens.reduce((a,b) => a+b)
```

- Problem: This means recomputing the filtered "evens" RDD three times – inefficient.
Modified example

Persist tells Spark to keep the data in memory even after it is done with the action. Allows future actions to reuse without recomputing. Cache is synonym for default storage level (memory). Can also persist on disk, etc.

val arr1M = Array.range(1,1000001)
val rdd1M = sc.parallelize(arr1M, 8)
val evens = rdd1M.filter(a => (a%2) == 0)
evens.persist() // or cache()
val firstFiveEvens = evens.take(5)
// How many evens?
val totalEvens = evens.count()
// Sum of evens
val evenSum = evens.reduce((a,b) => a+b)

• Problem: This means recomputing the filtered "evens" RDD three times – inefficient.
• Solution: Persist the RDD!
Modified example

Demo...
Communication Example

Let's like at a global communication example: computing the number of times each word occurs

- Load a text file
- Split it into words
- Group same words together (all-to-all communication)
- Count each word

```scala
val lines = sc.textFile("mytext")
val words = lines.flatMap(
    line => line.split(" ")
)
val wordKV = words.map(s => (s, 1))
val groupedWords = wordKV.groupByKey()
val wordCounts = groupedWords.map(
    t => (t._1, t._2.sum)
)
val counts = wordCounts.collect()
```
Communication Example

- Let's like at a global communication example: computing the number of times each word occurs
  - Load a text file
  - Split it into words
  - Group same words together (all-to-all communication)
  - Count each word

```scala
val lines = sc.textFile("mytext")
val words = lines.flatMap (
    line => line.split(" ")
)
val wordKV = words.map(s => (s, 1))
val groupedWords = wordKV.groupByKey()
val wordCounts = groupedWords.map(
    t => (t._1, t._2.sum)
)
val counts = wordCounts.collect()
```
Communication Example

val lines = sc.textFile("mytext")
val words = lines.flatMap (
  line => line.split(" ")
)
val wordKV = words.map(s => (s, 1))
val groupedWords = wordKV.groupByKey()
val wordCounts = groupedWords.map(
  t => (t._1, t._2.sum)
)
val counts = wordCounts.collect()

- Let's like at a global communication example: computing the number of times each word occurs
  - Load a text file
  - Split it into words
  - Group same words together (all-to-all communication)
  - Count each word
**Communication Example**

```scala
val lines = sc.textFile("mytext")
val words = lines.flatMap {
  line => line.split(" ")
}
val wordKV = words.map(s => (s, 1))
val groupedWords = wordKV.groupByKey()
val wordCounts = groupedWords.map {
  t => (t._1, t._2.sum)
}
val counts = wordCounts.collect()
```

- Let's like at a global communication example: computing the number of times each word occurs
  - Load a text file
  - Split it into words
  - Group same words together (all-to-all communication)
  - Count each word
Communication Example

- Load file, default number of partitions is # of HDFS blocks
- flatMap maps one value to (possibly) many, instead of one-to-one like map
- groupByKey combines all key-value pairs with the same key \((k, v_1), ..., (k, v_n)\) into a single key-value pair \((k, (v_1, ..., v_n))\).
- Collect returns all elements to the driver

```scala
val lines = sc.textFile("mytext")
val words = lines.flatMap (
  line => line.split(" ")
)
val wordKV = words.map(s => (s, 1))
groupedByWords = wordKV.groupByKey()
val wordCounts = groupedWords.map(  
t => (t._1, t._2.sum)  
)
val counts = wordCounts.collect()
```

- Let's like at a global communication example: computing the number of times each word occurs
  - Load a text file
  - Split it into words
  - Group same words together (all-to-all communication)
  - Count each word
The DAG

val lines = sc.textFile("mytext")
val words = lines.flatMap {
  line => line.split(" ")
}
val wordKV = words.map(s => (s, 1))
val groupedWords = wordKV.groupByKey()
val wordCounts = groupedWords.map {
  t => (t._1, t._2.sum)
}
val counts = wordCounts.collect()

Omitting collect due to space constraints
Execution

HDFS Block 1 → Partition 1 → Split 1 → Pair 1 → Group 1 → Count 1
HDFS Block 2 → Partition 2 → Split 2 → Pair 2 → Group 2 → Count 2
HDFS Block N → Partition N → Split N → Pair N → Group N → Count N
Execution

HDFS Block 1 → Partition 1 → Split 1 → Pair 1 → Group 1 → Count 1
HDFS Block 2 → Partition 2 → Split 2 → Pair 2 → Group 2 → Count 2
HDFS Block N → Partition N → Split N → Pair N → GroupN → Count N

Spring 2015

UW CSEP 524 (PMP Parallel Computation): Ringenburg
Execution

HDFS Block 1 → Partition 1 → Split 1 → Pair 1 → Group 1 → Count 1
HDFS Block 2 → Partition 2 → Split 2 → Pair 2 → Group 2 → Count 2
... → ... → ... → ... → ... → ...
HDFS Block N → Partition N → Split N → Pair N → Group N → Count N

No cross-node dependencies: operations pipelined into single task.

Spring 2015
Execution

HDFS Block 1 → Partition 1 → Split 1 → Pair 1 → Group 1 → Count 1
HDFS Block 2 → Partition 2 → Split 2 → Pair 2 → Group 2 → Count 2
... → ... → ... → ... → ... → ...
HDFS Block N → Partition N → Split N → Pair N → Group N → Count N

Spring 2015  UW CSEP 524 (PMP Parallel Computation): Ringenburg
Execution

```
(q, (1))
(b, (1, 1))

"the quick brown"

(f, (1))
(j, (1))
(o, (1))

"fox jumps over"

(t, (1, 1))
(d, (1))

"the brown dog"
```

```
HDFS Block 1 -> Partition 1 -> Split 1 -> Pair 1 -> Group 1 -> Count 1
HDFS Block 2 -> Partition 2 -> Split 2 -> Pair 2 -> Group 2 -> Count 2
HDFS Block N -> Partition N -> Split N -> Pair N -> GroupN -> Count N

"the quick brown"
"fox jumps over"
"the brown dog"
```
Execution

HDFS Block 1 → Partition 1 → Split 1 → Pair 1 → Group 1 → Count 1
HDFS Block 2 → Partition 2 → Split 2 → Pair 2 → Group 2 → Count 2
HDFS Block N → Partition N → Split N → Pair N → GroupN → Count N

(quick, 1) (brown, 2)
"the quick brown"

(fox, 1) (jumps, 1) (over, 1)
"fox jumps over"

(the, 2) (dog, 1)
"the brown dog"
Execution

HDFS Block 1 → Partition 1 → Split 1 → Pair 1 → Group 1 → Count 1
HDFS Block 2 → Partition 2 → Split 2 → Pair 2 → Group 2 → Count 2
... → ... → ... → ... → ... → ...
HDFS Block N → Partition N → Split N → Pair N → Group N → Count N

These are also pipelined into a single task per node

(quick, 1) (brown, 2)
"the quick brown"

(fox, 1) (jumps, 1) (over, 1)
"fox jumps over"

(the, 2) (dog, 1)
"the brown dog"
Execution

HDFS Block 1 → Partition 1 → Split 1 → Pair 1 → Group 1 → Count 1
HDFS Block 2 → Partition 2 → Split 2 → Pair 2 → Group 2 → Count 2
... → ... → ... → ... → ... → ...
HDFS Block N → Partition N → Split N → Pair N → Group N → Count N

"the quick brown"
"fox jumps over"
"the brown dog"

(quick, 1)
(brown, 2)

(fox, 1)
(jumps, 1)
(over, 1)

(the, 2)
(dog, 1)

Only barrier is here

These are also pipelined into a single task per node
Tasks and pipelining

• If an RDD partitions's dependencies are on a single other RDD partition (or on co-partitioned data), the operations can be pipelined into a single task
  – Co-partitioned: all of the parent RDD partitions are co-located with child RDD partitions that need them
  – Pipelined: Operations can occur as soon as the local parent data is ready – no synchronization
  – Task: A pipelined set of operations

• Every task ends with a shuffle, and output or returning data back to the driver.
Shuffle implementation

• All data exchanges between executors implemented via shuffle
  – Senders ("mappers") send data to block managers; block managers write to disks, tell scheduler how much destined for each reducer
  – Barrier until all mappers complete shuffle writes
  – Receivers ("reducers") request data from block managers that have data for them; block managers read and send
Shuffle Walkthrough

- (the, 1)
- (quick, 1)
- (brown, 1)

- (fox, 1)
- (jumps, 1)
- (over, 1)

- (the, 1)
- (brown, 1)
- (dog, 1)
Sort-based shuffle groups by destination "reducer" (receiver).
"Reducers" request blocks via block managers (only showing remote requests here to avoid clutter).
Shuffle Walkthrough

Data sent to reducers via block managers (BMs not shown).
Communication Example, Revisited

```scala
val lines = sc.textFile("mytext")
val words = lines.flatMap {
  line => line.split(" ")
}
val wordKV = words.map(s => (s, 1))
val groupedWords = wordKV.reduceByKey{(a,b) => a + b}
val counts = wordCounts.collect()
```

- Can do this more efficiently with "reduceByKey" – aggregates results locally before shuffling
  - Like Schwartz's algorithm
  - Reduces amount of data sent over the network
### Sample of Spark RDD Transformations

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>myRDD.map(function)</code></td>
<td>Generate new RDD by applying function to each element of myRDD</td>
</tr>
<tr>
<td><code>myRDD.filter(function)</code></td>
<td>Generate new RDD which contains elements of myRDD for which function returns true</td>
</tr>
<tr>
<td><code>myRDD.union(otherRDD)</code></td>
<td>Generate an RDD that contains all the elements of both myRDD and otherRDD</td>
</tr>
<tr>
<td><code>myRDD.intersection(otherRDD)</code></td>
<td>Generate an RDD that contains only elements which appear in both myRDD and otherRDD</td>
</tr>
<tr>
<td><code>myRDD.distinct()</code></td>
<td>Generate a new RDD that contains only the distinct elements of myRDD (duplicates removed)</td>
</tr>
<tr>
<td><code>myRDD.groupByKey()</code></td>
<td>If myRDD consists of key-value pairs, return a new RDD consisting of each key and a list of all of its values: e.g., (k1, x), (k2, y), (k1, z) → (k1, (x,z)), (k2, (y))</td>
</tr>
</tbody>
</table>

### Sample of Spark RDD Transformations

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>myRDD.reduceByKey(function)</code></td>
<td>Like <code>groupByKey</code>, but aggregate all values for each key using the parameter function</td>
</tr>
<tr>
<td><code>myRDD.join(otherRDD)</code></td>
<td>If <code>myRDD</code> and <code>otherRDD</code> are datasets of key-value pairs, where <code>myRDD</code> contains (K,V) pairs and <code>otherRDD</code> contains (K,W) pairs, generate a new RDD of (K, (V,W)) pairs for every V and W with the same key K.</td>
</tr>
<tr>
<td><code>myRDD.sortByKey()</code></td>
<td>Return a new RDD that contains the same key-value pairs as <code>myRDD</code>, but in sorted order (can optionally specify ascending or descending order).</td>
</tr>
<tr>
<td><code>myRDD.flatMap(function)</code></td>
<td>Generate a new RDD by applying function to each element of <code>myRDD</code>. Unlike <code>map</code>, function should return a sequence of 0 or more elements rather than a single element (so one original RDD element may generate 0 or many new RDD elements).</td>
</tr>
</tbody>
</table>

See [http://spark.apache.org/docs/latest/programming-guide.html](http://spark.apache.org/docs/latest/programming-guide.html) for more
## Sample of Spark RDD Actions

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>myRDD.reduce(function)</td>
<td>Perform a reduction using function across all elements of an RDD, returning result to driver (e.g., sum all elements of the RDD).</td>
</tr>
<tr>
<td>myRDD.count()</td>
<td>Return the total number of elements in the RDD (closest thing to a no-op action, since this data is always computed – can be used to force evaluation).</td>
</tr>
<tr>
<td>myRDD.collect()</td>
<td>Return all elements of myRDD. Make sure they will fit in the driver's memory!</td>
</tr>
<tr>
<td>myRDD.takeOrdered(n)</td>
<td>Return the first n sorted elements of myRDD (may optionally provide a comparator).</td>
</tr>
<tr>
<td>myRDD.countByKey()</td>
<td>Return a hashmap of (K, integer) pairs, where the integer is the number of elements with key K.</td>
</tr>
</tbody>
</table>

See [http://spark.apache.org/docs/latest/programming-guide.html](http://spark.apache.org/docs/latest/programming-guide.html) for more
Open Discussion

• Last formal lecture of the course ... what do you want to discuss/ask about?
• Hard stop at 8:30 for first few presentations.