Topics covered in this lecture

- Transformations and Actions
  - RDDs
  - DataFrames

Element-wise transformations: `filter()`

- Takes a function and returns an RDD that only has elements that pass the `filter()` function

Example:

```
inputRDD = {1, 2, 3, 4}
map x => x*x
```

Filtered RDD = {2, 3}

```
inputRDD = {1, 2, 3, 4}
filter x => x != 1
```

Filtered RDD = {2, 3, 4}

Element-wise transformations: `map()`

- Takes a function and applies it to each element in the RDD
- Result of the function is the new value of each element in the resulting RDD

Example:

```
map x => x*x
```

Mapped RDD = {1, 4, 9, 16}

Filtered RDD = {2, 3, 4}
Things that can be done with `map()`
- Fetch website associated with each URL in collection to just squaring numbers
- `map()`'s return type does not have to be the same as its input type
- Multiple output elements for each input element?
  - Use `flatMap()`

```python
lines=sc.parallelize(['Hello world', 'Hi'])
words=lines.flatMap(lambda line: line.split(' '))
words.first() # returns hello
```

Difference between `map` and `flatMap`

```python
RDD1.map(lambda x: x.upper())
```

Psuedo set operations
- RDDs support many of the operations of mathematical sets such as union, intersection, etc.
- Even when the RDDs themselves are not properly sets

```python
RDD1 = {coffee, coffee, panda, monkey, tea}
RDD2 = {coffee, monkey, kitty}
RDD1.distinct() = {coffee, monkey, panda, tea}
RDD1.union(RDD2) = {coffee, coffee, coffee, panda, monkey, monkey, tea, kitty}
RDD1.intersection(RDD2) = {coffee, monkey}
RDD1.subtract(RDD2) = {panda, tea}
```

Cartesian product between two RDDs

```python
RDD1: [User1, User2, User3]
RDD2: [Venue("Betabrand"), Venue("Asha Tree House"), Venue("Ritual")]
RDD1.cartesian(RDD2) = [(User1, Venue("Betabrand")), (User1, Venue("Asha Tree House")), (User1, Venue("Ritual")), (User2, Venue("Betabrand")), (User2, Venue("Asha Tree House")), (User2, Venue("Ritual")), (User3, Venue("Betabrand")), (User3, Venue("Asha Tree House")), (User3, Venue("Ritual"))]
```

Some simple set operations

Common Actions
Actions on Basic RDDs

- **reduce()**
  - Takes a function that operates on two elements in the RDD, returns an element of the same type.
  - E.g., sum of all elements in the RDD: `sum = rdd.reduce(lambda x, y: x + y)`

- **fold()**
  - Takes a function with the same signature as `reduce()`, but also takes a "zero value" for initial call.
  - "Zero value" is the identity element for initial call.
  - E.g., `0` for `+`, `1` for `*`, empty list for concatenation.

Both `fold()` and `reduce()` require return type to be of the same type as the RDD elements

- The `aggregate()` removes that constraint.
  - For e.g., when computing a running average, maintain both the count so far and the number of elements.

Examples: Basic actions on RDDs

- **collect()**
  - Return all elements from the RDD.
  - Invocation: `rdd.collect()`
  - Result: `{1, 2, 3, 3}`

- **count()**
  - Number of elements in the RDD.
  - Invocation: `rdd.count()`
  - Result: `4`

- **countByValue()**
  - Number of times each element occurs in the RDD.
  - Invocation: `rdd.countByValue()`
  - Result: `{(1,1), (2,1), (3,2)}`
Examples: Basic actions on RDDs

- **Our RDD contains (1, 2, 3, 3)**
  - **take(num)**
    - Return `num` elements from the RDD
    - Invocation: `rdd.take(2)`
    - Result: `{1, 2}`

- **reduce(func)**
  - Combine the elements of the RDD together in parallel
  - Invocation: `rdd.reduce((x, y) => x + y)`
  - Result: 9

- **aggregate(zeroValue)(seqOp, combOp)**
  - Similar to `reduce()` but used to return a different type
  - Invocation:
    - `rdd.aggregate((0,0)) ((x,y) => (x._1 + y, x._2 +1), (x,y) => (x._1 + y._1, x._2 + y._2))`
  - Result: (9, 4)

- **foreach(func)**
  - Apply the provided function to each element of the RDD
  - Invocation: `rdd.foreach(func)`
  - Result: Nothing

**Why persistence?**

- Spark RDDs are lazily evaluated, and we may sometimes wish to use the same RDD multiple times
  - Naively, Spark will **recompute RDD and all of its dependencies** each time we call an action on the RDD
  - Super expensive for iterative algorithms
- To avoid recomputing RDD multiple times?
  - Ask Spark to **persist** the data
  - The nodes that compute the RDD store the partitions
Coping with failures

- If a node that has data persisted on it fails?
  - Spark recomputes lost partitions of data when needed
- Also, replicate data on multiple nodes
  - To handle node failures without slowdowns

Persistence Levels for Spark

<table>
<thead>
<tr>
<th>Level</th>
<th>Space Used</th>
<th>CPU time</th>
<th>In Memory</th>
<th>On disk</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>MEMORY_ONLY</td>
<td>High</td>
<td>Low</td>
<td>Y</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>MEMORY_ONLY_SER</td>
<td>Low</td>
<td>High</td>
<td>Y</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>MEMORY_AND_DISK</td>
<td>High</td>
<td>Medium</td>
<td>Some</td>
<td>Some</td>
<td>Spills to disk if there is too much data to fit in memory</td>
</tr>
<tr>
<td>MEMORY_AND_DISK_SER</td>
<td>Low</td>
<td>High</td>
<td>Some</td>
<td>Some</td>
<td>Spills to disk if there is too much data to fit in memory.  Stores serialized representation in memory</td>
</tr>
<tr>
<td>DISK_ONLY</td>
<td>Low</td>
<td>High</td>
<td>N</td>
<td>Y</td>
<td></td>
</tr>
</tbody>
</table>

What if you attempt to cache too much data to fit in memory?

- Spark will evict old partitions using a Least Recently Used Cache policy
  - For memory only storage partitions, it will be recomputed the next time they are accessed
  - For memory_and_disk ones? Write them out to disk
- RDDs also come with a method, unpersist()
  - Manually remove data elements from the cache

RDDs of key/value pairs

- Key/value RDDs are commonly used to perform aggregations
  - Might have to do ETL (Extract, Transform, and Load) to get data into key/value formats
- Advanced feature to control layout of pair RDDs across nodes
  - Partitioning

RDDs containing key/value pairs

- Are called pair RDDs
- Useful building block in many programs
  - Expose operations that allow actions on each key in parallel or regroup data across network
  - reduceByKey() to aggregate data separately for each key
  - join() to merge two RDDs together by grouping elements of the same key
Pair RDDs

- RDDs that contain key/value pairs
- Expose partitions that allow you to act on each key in parallel or regroup data across the network

Creating Pair RDDs

- `pair=lines.map(lambda x: (x.split(" "))[0], x)`
  - Creates a pair RDD using the first word as the key
- Java does not have a built-in tuple type
  - `scala.Tuple2`
  - `new Tuple2(elem1, elem2)`

Transformations on Pair RDDs

1/5

- Pair RDD = {(1,2), (3,4), (3,6) }
- `reduceByKey(func)`
  - Combine values with the same key
  - Invocation: `rdd.reduceByKey((x, y) => x + y)`
  - Result: { (1, 2), (3,10) }

2/5

- Pair RDD = {(1,2), (3,4), (3,6) }
- `groupByKey(func)`
  - Group values with the same key
  - Invocation: `rdd.groupByKey()`
  - Result: { (1,[2]), (3,[4, 6]) }

3/5

- Pair RDD = {(1,2), (3,4), (3,6) }
- `mapValues(func)`
  - Apply function to each value of a pair RDD without changing the key
  - Invocation: `rdd.mapValues(x=> x+1)`
  - Result: { (1, 3), (3, 5), (3, 7) }
Transformations on Pair RDDs [4/5]

- Pair RDD = {{1,2}, {3,4}, {3,6}}
- values()
  - Return an RDD of just the values
  - Invocation: rdd.values()
  - Result: {2, 4, 6}

Transformations on Pair RDDs [5/5]

- Pair RDD = {{1,2}, {3,4}, {3,6}}
- sortByKey()
  - Return an RDD sorted by the key
  - Invocation: rdd.sortByKey()
  - Result: {{1,2}, {3,4}, {3,6}}

Transformations on two Pair RDDs [1/5]

- rdd = {{1,2}, {3,4}, {3,6}}
- other = {{3,9}}
- subtractByKey()
  - Remove elements with a key present in the other RDD
  - Invocation: rdd.subtractByKey(other)
  - Result: {{1,2}}

Transformations on two Pair RDDs [2/5]

- rdd = {{1,2}, {3,4}, {3,6}}
- other = {{3,9}}
- join()
  - Perform an inner join between two RDDs. Only keys that are present in both pair RDDs are output
  - Invocation: rdd.join(other)
  - Result: {{3, {4,9}}, {3, {6,9}}}

Transformations on two Pair RDDs [3/5]

- rdd = {{1,2}, {3,4}, {3,6}}
- other = {{3,9}}
- leftOuterJoin()
  - Perform a join between two RDDs where the key must be present in the first RDD.
  - Value associated with each key is a tuple of the value from the source and an Option for the value from the other pair RDD
  - In Python if a value is not present, None is used.
  - Invocation: rdd.leftOuterJoin(other)
  - Result: {{1, {2,None}}, {3, {4, 9}}, {3, {6, 9}}}
Transformations on two Pair RDDs [4/5]

- \( \text{rdd} = \{(1,2), (3,4), (3,6)\} \quad \text{other} = \{(3,9)\} \)

- \text{rightOuterJoin()}
  - Perform a join between two RDDs where the key must be present in the other RDD.
  - Tuple has an option for the source rather than other RDD
  - Invocation: \( \text{rdd.rightOuterJoin(other)} \)
  - Result: \( \{(3, (4,9)), (3, (6,9))\} \)

Transformations on two Pair RDDs [5/5]

- \( \text{rdd} = \{(1,2), (3,4), (3,6)\} \quad \text{other} = \{(3,9)\} \)

- \text{cogroup()}
  - Group data from both RDDs using the same key
  - Invocation: \( \text{rdd.cogroup(other)} \)
  - Result: \( \{(1, ([2],[])), (3, ([4, 6], [9]))\} \)

Example of chaining operations

```
key  value
panda 0
pink  3
pirate 3
panda 1
pink  4
```

```
key  value
panda (0, 1)
pink  (3, 1)
pirate (3, 1)
panda (1, 1)
pink  (4, 1)
```

```
rdd = sc.textfile("s3://…")
words = rdd.flatMap(lambda x: x.split(" "))
result = words.map(lambda x: (x,1)).reduceByKey(lambda x, y: x+y)
```

A word count example

- We are using \text{flatMap()} to produce a pair RDD of words and the number 1.

```
rdd = sc.textfile("s3://…")
words = rdd.flatMap(lambda x: x.split(" "))
result = words.map(lambda x: (x,1)).reduceByKey(lambda x, y: x+y)
```

Dataframe Review

- Maintains features of RDD’s
  - In-memory, resilient, distributed computing
  - Supports some transformations and actions
  - API’s in a variety of languages

- Differs from RDD’s by
  - Maintenance of data schema
  - Additional optimizations to query plan (Catalyst rules)
Dataframe sources

- Dataframes may be initialized from a variety of sources
  - Distributed File Systems (CSV, JSON, XML, etc)
  - Databases (MySQL, Cassandra, Hive, Redis, etc)
  - RDD’s
- Able to “inferSchema” of structured data

Example: Dataframe Initialization

```scala
val df1 = sqlContext.read.format("csv")
  .option("header", "true")
  .option("inferSchema", "true")

val df2 = sqlContext.read
  .format("org.apache.spark.sql.cassandra")
  .option("table", "students")
  .option("keyspace", "csu").load()
```

Column Manipulations

1/4

- `withColumn(columnName, func)`
  - Return an Dataframe with the additional column
  - Invocation: `df.withColumn("dogYears", df.age / 7)`

- `dropColumn(columnName)`
  - Return an Dataframe without the column
  - Invocation: `df.dropColumn("age")`

2/4

- `select(columnNames)`
  - Return an Dataframe with the specified columns
  - Invocation: `df.select("firstName", "age")`

- `describe(columnName)`
  - Compute summary statistics over Dataframe columns
  - Invocation: `df.describe("age"), df.describe()`

3/4

```scala
val df = Seq(
  ("Peterson", "Marcus", 54),
  ("Batey", "Edward", 36),
  ("Bruce", "Karen", 35)
).toDF("lastName", "firstName", "age")

df.withColumn("dogYears", df.age / 7.0)
  df.describe("age", "dogYears")
```
Column Manipulations [4/4]

Join column comparison
- Supports a variety of criteria
  - Sequence of column names (ex. Seq("id", "age"))
  - Elaborate comparison definitions (ex. df1("age") >= df2("age"))

Join Type
- Dataframes may perform multiple styles of join
  - Inner: typical dataset join with key to key match
  - Outer, left-outer, right-outer: result contains all rows, filling in columns with 'null' values where data doesn’t exist
  - Left-semi, right-semi: similar to outer join, but result only contains rows in specified source dataset

Example: Spark SQL
```scala
val df = Seq(
  ("Peterson", "Marcus", 54),
  ("Batey", "Edward", 36),
  ("Bruce", "Karen", 35)
).toDF("lastName", "firstName", "age")

df.createOrReplaceTempView("people")
spark.sql("SELECT firstName, age, age / 7.0 as dogYears AS FROM people where age < 50")
```

Tuning the level of parallelism
Tuning the level of parallelism

- Every RDD has a fixed number of partitions
  - Determine the degree of parallelism when executing operations
  - During aggregations or grouping operations, you can ask Spark to use a specific number of partitions
  - This will override defaults that Spark uses

Example: Tuning the level of parallelism

```python
data = [('a', 3), ('b', 4), ('a', 1)]
sc.parallelize(data).reduceByKey(lambda x, y: x+y)  # default
sc.parallelize(data).reduceByKey(lambda x, y: x+y, 10)  # Custom
```

What if you want to tune parallelism outside of grouping and aggregation operations?

- There is the repartition() operation
  - Shuffles data across the network to create a new set of partitions
  - Very expensive operation!

- There is the coalesce() operation
  - Allow avoiding data movement
  - But only if you are decreasing the number of partitions
  - Check `rdd.getNumPartitions()` and make sure you are coalescing to fewer partitions

The contents of this slide-set are based on the following references