Frequently asked questions from the previous class survey

- Why use Hadoop if Spark is so much faster?

Topics covered in this lecture

- Spark Resilient Distributed Datasets
  - Transformations
  - Dependencies
  - Actions
  - Examples

A simple Scala word count example

```scala
def simpleWordCount(rdd: RDD[String]): RDD[(String, Int)] = {
  val words = rdd.flatMap(_.split(" "))
  val wordPairs = words.map((_, 1))
  val wordCounts = wordPairs.reduceByKey(_ + _)
  wordCounts
}
```

A CLOSER LOOK AT RDD OPERATIONS

RDDs support two types of operations

- Transformations
  - Operations that return a new RDD. E.g., `filter()`
- Actions
  - Operations that return a result to the driver program or write to storage
  - Kicks off a computation. E.g., `count()`

Distinguishing aspects

- Transformations return RDDs
- Actions return some other data type
Many transformations are element-wise
- Work on only one element at a time

Some transformations are not element-wise
- E.g.: We have a logfile, log.txt, with several messages, but we only want to select error messages

\[
\text{inputRDD} = \text{sc.textFile}(\text{"log.txt"})
\]
\[
\text{errorsRDD} = \text{inputRDD.filter}(\lambda x: \text{"error" in x})
\]

In our previous example...
- filter does not mutate inputRDD
  - Returns a pointer to an entirely new RDD
  - inputRDD can still be reused later in the program

- We could use inputRDD to search for lines with the word "warning"
  - While we are at it, we will use another transformation, union(), to print number of lines that contained either
    \[
    \text{errorsRDD} = \text{inputRDD.filter}(\lambda x: \text{"error" in x})
    \]
    \[
    \text{warningsRDD} = \text{inputRDD.filter}(\lambda x: \text{"warning" in x})
    \]
    \[
    \text{badlinesRDD} = \text{errorsRDD.union(warningsRDD)}
    \]

RDD lineage graphs
- As new RDDs are derived from each other using transformations, Spark tracks dependencies
- Lineage graph
  - Uses lineage graph to
    - Compute each RDD on demand
    - Recover lost data if part of persistent RDD is lost

In our previous example...
- Note how union() is different from filter()
  - Operates on 2 RDDs instead of one
  - Transformations can actually operate on any number of RDDs

Actions
- We can create RDDs from each other using transformations
- At some point, we need to actually do something with the dataset
  - Actions
  - Forces evaluations of the transformations required for the RDD they were called on
Each Spark program must contain an action

- Actions either:
  - Bring information back to the driver or
  - Write the data to stable storage
- Actions are what force evaluation of a Spark program
- Persist calls also force evaluation, but usually do not mark the end of a Spark job
- Actions that bring data back to the driver include `collect`, `count`, `collectAsMap`, `sample`, `reduce`, and `take`.

Let's try to print information about `badlinesRDD`

```scala
println(s"Input had \${badLinesRDD.count()} concerning lines")
println("here are 10 examples:")
for(line <- badLinesRDD.take(10))
  println(line)
```

RDDs also have a collect to retrieve the entire RDD

- Useful if program filters RDD to a very small size and you want to deal locally
  - Your entire dataset must fit in memory on a single machine to use `collect()` on it
  - Should NOT be used on large datasets
- In most cases, RDDs cannot be `collect()`ed to the driver
- Common to write data out to a distributed storage system … HDFS or S3

A caveat about actions and scaling

- Some of these actions do not scale well, since they can cause memory errors in the driver
- In general, it is best to use actions like `take`, `count`, and `reduce`, which bring back a fixed amount of data to the driver, rather than `collect` or `sample`.

Lazy Evaluation

- Transformations on RDDs are lazily evaluated
  - Spark will not begin to execute until it sees an action
- Uses this to reduce the number of passes it has to take over data by grouping operations together
- What does this mean?
  - When you call a transformation on an RDD (for e.g. `map`) the operation is not immediately performed
  - Spark internally records metadata that operation is requested

How you should think of RDDs

- Rather than thinking of it as containing specific data
  - Best to think of it as containing instructions on how to compute the data that we build through transformations
- Loading data into a RDD is lazily evaluated just as transformations are
### Wide and Narrow Transformations

#### October 3, 2017

**WIDE AND NARROW TRANSFORMATIONS**

#### October 3, 2017

**Transformations and Dependencies**

- Two categories of dependencies
  - Narrow
    - Each partition of the parent RDD is used by at most one partition of the child RDD
  - Wide
    - Multiple child RDD partitions may depend on a single parent RDD partition

The narrow versus wide distinction has significant implications for the way Spark evaluates a transformation and, consequently, for its performance.

---

**Narrow Transformations**

- Narrow transformations are those in which each partition in the child RDD has simple, finite dependencies on partitions in the parent RDD
- Can be determined at design time, irrespective of the values of the records in the parent partitions
- Partitions in narrow transformations can either depend on:
  - One parent (such as in the map operator), or
  - A unique subset of the parent partitions that is known at design time (coalesce)
- Narrow transformations can be executed on an arbitrary subset of the data without any information about the other partitions.

---

**Wide Transformations**

- Transformations with wide dependencies cannot be executed on arbitrary rows
- Require the data to be partitioned in a particular way, e.g., according the value of their key
- In sort, for example, records have to be partitioned so that keys in the same range are on the same partition
- Transformations with wide dependencies include sort, reduceByKey, groupByKey, join, and anything that calls the rePartition function

---

**Dependencies between partitions for narrow transformations**

**Dependencies between partitions for wide transformations**
COMMON TRANSFORMATIONS AND ACTIONS

Element-wise transformations: **filter()**
- Takes in a function and returns an RDD that only has elements that pass the filter() function

Element-wise transformations: **map()**
- Takes in 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

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()**

Difference between **map()** and **flatMap()**
- RDD1.map(tokenize)
- RDD1.flatMap(tokenize)

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
**Some simple set operations**

- **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**

- **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"))\}

---

**Common Actions**

- **reduce()**
  - Takes a function that operates on two elements in the RDD; returns an element of the same type
  - E.g. of such an operation? \( + \) sums the RDD
    \[
    \text{sum} = \text{rdd}.reduce(\lambda x, y: x+y)
    \]

- **fold()**
  - Takes a function with the same signature as \( \text{reduce()} \), 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 \( \text{fold()} \) and \( \text{reduce()} \) require return type to be of the same type as the RDD elements

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

---

**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. of such an operation? \( + \) sums the RDD
    \[
    \text{sum} = \text{rdd}.reduce(\lambda x, y: x+y)
    \]

- **fold()**
  - Takes a function with the same signature as \( \text{reduce()} \), 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

---

**Examples: Basic Actions on RDDs**

- Both \( \text{fold()} \) and \( \text{reduce()} \) require return type to be of the same type as the RDD elements
  - The \( \text{aggregate()} \) 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 [1/7]
- Our RDD contains {1, 2, 3, 3}
- **collect()**
  - Return all elements from the RDD
  - Invocation: `rdd.collect()`
  - Result: `{1, 2, 3, 3}`

Examples: Basic actions on RDDs [2/7]
- Our RDD contains {1, 2, 3, 3}
- **count()**
  - Number of elements in the RDD
  - Invocation: `rdd.count()`
  - Result: 4

Examples: Basic actions on RDDs [3/7]
- Our RDD contains {1, 2, 3, 3}
- **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 [4/7]
- Our RDD contains {1, 2, 3, 3}
- **take(num)**
  - Return num elements from the RDD
  - Invocation: `rdd.take(2)`
  - Result: `{1, 2}`

Examples: Basic actions on RDDs [5/7]
- Our RDD contains {1, 2, 3, 3}
- **reduce(func)**
  - Combine the elements of the RDD together in parallel
  - Invocation: `rdd.reduce( (x,y) => x + y )`
  - Result: 9

Examples: Basic actions on RDDs [6/7]
- Our RDD contains {1, 2, 3, 3}
- **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)`
Examples: Basic actions on RDDs

- Our RDD contains \(1, 2, 3, 3\)
- \texttt{foreach(func)}
  - Apply the provided function to each element of the RDD
  - Invocation: \texttt{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>
</tr>
</thead>
<tbody>
<tr>
<td>MEMORY_ONLY</td>
<td>High</td>
<td>Low</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>MEMORY_ONLY_SER</td>
<td>Low</td>
<td>High</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>MEMORY_AND_DISK</td>
<td>High</td>
<td>Medium</td>
<td>Some</td>
<td>Some</td>
</tr>
<tr>
<td>MEMORY_AND_DISK_SER</td>
<td>Low</td>
<td>High</td>
<td>Some</td>
<td>Some</td>
</tr>
</tbody>
</table>

Comments:
- Spills to disk if there is too much data to fit in memory.
- Some serialized representations in memory.

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

- Spark will \texttt{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, \texttt{unpersist()}
  - Manually remove data elements from the cache.
The contents of this slide-set are based on the following references:


- *Real-Time Analytics: Techniques to Analyze and Visualize Streaming Data*. Byron Ellis. Wiley. (Chapter 2)