Week 2: Introduction to Spark and Resilient Distributed Datasets

Resilient Distributed Datasets (RDDs)

- the fundamental abstraction for distributed data computation in Spark
- immutable collection of objects that is (or can be) distributed across a cluster
- can be manipulated in parallel

- three basic operations: creation, transformation, and action

RDD Creation

- can be created by reading a file (or some external resource) or by parallelizing a collection of objects (e.g. a set or list)

```
lines = sc.textFile("README.md")
```

```
lines = sc.parallelize(["line1", "line 2"])
```

Data partitioning

- remember the general idea of parallel computing: things happen at the same time
- partitioning refers to breaking the data up into parts
- by default, data is broken up into partitions that are multiples of 64MB
- BUT we can tune that parameter to match the number of cores on our system

Data partitioning

- note that data can only be partitioned if it's available on all nodes of a cluster
- therefore, if just reading a file from a local directory the file will not be split
- more information in the readings on Hadoop Distributed File System or HDFS

RDD Actions

- actions take data from RDDs and create non-RDD data structures
- these actions move data from Spark to non-Spark
- commonly used to get the results of transformations

RDD Actions

Commonly used actions:

- count(): return number of elements
- collect(): return list of all elements
- take(n): retrieve first n elements
- first(): retrieve first element

- NOTE: be careful with collect()!!! [WHY?]

RDD Transformations

- RDD transformations result in new RDDs
- examples include filtering, mapping, and reducing

Filtering

- consider the following text:

The University of Michigan School of Information is pleased to offer a new, fully online, master's program in applied data science.
We teach comprehensive applied data science at the intersection of people and technology.
We provide critical insight into data collection, computation, and analytics, and help develop hands-on skills using a multidisciplinary approach embedded in information, computer science, and statistics.
Coursework and projects focus on applying data science to real-world problems.
For more information on our program and its admissions criteria, we invite you to join our interest list.

Filtering

- let's assume each line is one complete sentence:

The University of Michigan School of Information is pleased to offer a new, fully online, master’s program in applied data science.

We teach comprehensive applied data science at the intersection of people and technology.

We provide critical insight into data collection, computation, and analytics, and help develop hands-on skills using a multidisciplinary approach embedded in information, computer science, and statistics.

Coursework and projects focus on applying data science to real-world problems.

For more information on our program and its admissions criteria, we invite you to join our interest list.

Filtering

Let's count the lines that contain the word "data":

```
lines = sc.textfile('data/blurb.txt')
data_lines = lines.filter(lambda line: 'data' in line)
data_lines.count()

data_lines.collect()
```

Filtering

```
data_lines = lines.filter(lambda line: 'data' in line)
```

could be equivalently implemented as:

```
def data_filter(s):
    return 'data' in s
data_lines = lines.filter(data_filter)
```

Mapping

- apply a function to each element

Example:

```
nums = sc.parallelize([1, 2, 3, 4])
squared = nums.map(lambda x: x * x).collect()
for num in squared:
    print('%i ' % (num))
```

Mapping

```
lines = sc.parallelize(["hello world", "hi"])
words = lines.map(lambda line: line.split(" "))
print(words.collect())
```

```
[['hello', 'world'], ['hi']]
```

FlatMapping

```
lines = sc.parallelize(["hello world", "hi"])
words = lines.flatMap(lambda line: line.split(" "))
print(words.collect())

['hello', 'world', 'hi']
```

Set-like transformations

```
errorsRDD = inputRDD.filter(lambda x: "error" in x)
warningsRDD = inputRDD.filter(lambda x: "warning" in x)
badLinesRDD = errorsRDD.union(warningsRDD)
```

```
rdd1.union(rdd2)
rdd1.intersection(rdd2)
rdd1.subtract(rdd2)
rdd1.distinct()
```