Spark RDDs

Initializing Spark

- We have to initialize the Spark context to get started.

```
SparkConf conf = new SparkConf().setAppName("SimpleApp")
    .setMaster("local");
JavaSparkContext sc = new JavaSparkContext(conf);
```

- There can only be one context at a given point in our program. So we would have to stop one if it was already running and we wanted to create a new context.

Resilient Distributed Datasets

- A RDD (Resilient Distributed Dataset) is a fault-tolerant collection of elements that can be operated on in parallel.
- There are two ways to create RDDs:
  - parallelizing an existing collection in your driver program
  - referencing a dataset in an external storage system, such as a shared filesystem, HDFS, HBase, or any data source offering a Hadoop InputFormat.

RDDs from Existing Collection

- The `parallelize` method on the `JavaSparkContext` is used to create a RDD from a `Collection`. For example:

```
List<Integer> data = Arrays.asList(1, 2, 3, 4, 5);
JavaRDD<Integer> distData1 = sc.parallelize(data); //use
    default #partitions
JavaRDD<Integer> distData2 = sc.parallelize(data, 10); //use
     10 partitions
```

- Now we could use parallel operations on this RDD. For example:

```
distData1.reduce((a, b) -> a + b);
```

RDDs from External Datasets

- Spark can create distributed datasets from any storage source supported by Hadoop, including your local file system, HDFS, HBase, Amazon S3 and others.
- Spark supports text files, `SequenceFiles`, and any other Hadoop `InputFormat`.
- Text file RDDs can be created using SparkContext's `textFile` method. This method takes an URI for the file (either a local path on the machine, or a hdfs://, s3a://, etc URI) and reads it as a collection of lines. For example:

```
JavaRDD<String> distFile = sc.textFile("data.txt");
//now, we can use parallel operators
int totalLength = distFile.map(s -> s.length()).reduce((a, b) ->
     a + b)
```

- All of Spark's file-based input methods, including textFile, support running on directories, compressed files, and wildcards as well.
- `JavaSparkContext.wholeTextFiles` method lets you read a directory containing multiple small text files, and returns each of them as (filename, content) pairs.
- `JavaRDD.saveAsObjectFile` and `JavaSparkContext.objectFile` support saving an RDD in a simple format consisting of serialized Java objects.

RDD Operations

- RDDs support two types of operations:
  - transformation: creates a new dataset from an existing one
  - action: returns a value to the driver program after running a computation on the dataset
- All transformations in Spark are lazy, in that they do not compute their results right away. Spark remembers the order of the transformations. This allows it be more efficient and fault tolerant.
- By default, each transformed RDD may be recomputed each time we run an action on it. However, we may also persist an RDD in memory using the `persist` (or `cache`) method There is also support for persisting RDDs on disk, or replicated across multiple nodes.
- Example:

```
JavaRDD<String> lines = sc.textFile("data.txt");
JavaRDD<Integer> lineLengths = lines.map(s -> s.length());
// to persist
lineLengths.persist(StorageLevel.MEMORY_ONLY());
int totalLength = lineLengths.reduce((a, b) -> a + b);
```

Function Passing to Spark

- Spark's API relies heavily on passing functions in the driver program to run on the cluster. In Java, functions are represented by classes implementing the interfaces in the `org.apache.spark.api.java.function` package. This can be done in two ways:
  - We can create an anonymous or a named class that implements one of the defined function interfaces and pass an instance to Spark
  - Use lambda expressions to concisely define an implementation
  - Note that anonymous inner classes in Java can also access variables in the enclosing scope as long as they are marked final. Spark will ship copies of these variables to each worker node as it does for other languages.

Closures

- The closure is those variables and methods that must be visible for the executor to perform its computations on the RDD. Prior to execution, Spark computes the task's closure. This closure is serialized and sent to each executor.
- Example 1: What is the value of the counter? For local mode? For local[2] mode? For a cluster?

```
int counter = 0;
JavaRDD<Integer> rdd = sc.parallelize(data);

// Wrong: Don't do this!!
rdd.foreach(x -> counter += x);
println("Counter value: " + counter);
```

- We would use an `Accumulator` for the above scenario.
- Example 2:

```
rdd.foreach(println);
rdd.map(println);
// both of the above idioms won't work, why?

// use one of the below idioms (dependping on the size)
rdd.collect().foreach(println);
rdd.take(k).foreach(println); //print first k values
```

Working with Key-Value Pairs

- Key-value pairs are represented by the `Tuple2` class (from Scala). For example: `tuple = new Tuple2(a, b)`. We can access the fields by `tuple._1()`, `tuple._2()`.
- RDDs of key-value pairs are represented by `JavaPairRDD` class. These are built using special versions of `map` operations such as `mapToPair` and `flatMapToPair`.
- Example: Count how many times each line of text occurs in a file.

```
JavaRDD<String> lines = sc.textFile("data.txt");
JavaPairRDD<String, Integer> pairs = lines.mapToPair(
    s -> new Tuple2(s, 1));
JavaPairRDD<String, Integer> counts = pairs.
    reduceByKey((a, b) -> a + b);
// more examples: counts.sortByKey(), counts.collect
    ();
```

Transformations (1)

- `map`(func)
- `filter`(func)
- `flatMap`(func)
- `mapPartitions`(func)
- `mapPartitionsWithIndex`(func)
- `sample`(withreplacement, fraction, seed)
- `union`(otherDataset)
- `intersesction`(otherDataset)
- `distinct`([numPartitions])

Transformations (2)

- `groupByKey`([numPartitions])
- `reduceByKey`(func, [numPartitions])
- `aggregateByKey`(zeroValue, seqOp, combOp, [numPartitions])
- `sortByKey`([ascending], [numPartitions])
- `join`(otherDataset, [numPartitions])
  Also available: `leftOuterJoin`, `rightOuterJoin`, `fullOuterJoin`
- `cogroup`(otherDataset, [numPartitions])

Transformations (3)

- `cartesian`(otherDataset)
- `pipe`(command, [envVars])
- `coalesce`(numPartitions)
- `repartition`(numPartitions)
- `repartitionAndSortWithinPartitions`(partitioner)

Actions

- `reduce`(func)
- `collect`()
- `count`()
- `first`()
- `take`(n)
- `takeSample`(withReplacement, num, [seed])
- `takeOrdered`(n, [ordering])
- `saveAsTextFile`(path)
- `saveAsSequenceFile`(path)
- `saveAsObjectFile`(path)
- `countByKey`()
- `foreach`(func)

Shuffle Operations (1)

- Shuffle operation re-distributes the data so that it's grouped differently across partitions. Although the set of elements in each partition of newly shuffled data will be deterministic, and so is the ordering of partitions themselves, the ordering of these elements is not.
- Involves disk I/O, data serialization, and network I/O. Complex and costly.
- Shuffle can be triggered by operations like `repartition`, `coalesce`, *byKey* operations (except for counting) such as `groupByKey` and `reduceByKey`, and join operations like `cogroup` and `join`.

Performance of Shuffle

- The shuffle is implemented using map and reduce (simialr to Hadoop MapReduce)
- Internally, results from individual map tasks are kept in memory until they can't fit. Then, these are sorted based on the target partition and written to a single file. On the reduce side, tasks read the relevant sorted blocks.
- Certain shuffle operations can consume significant amounts of heap memory since they employ in-memory data structures to organize records before or after transferring them. When data does not fit in memory Spark will spill these tables to disk, incurring the additional overhead of disk I/O and increased garbage collection.
- Shuffle also generates a large number of intermediate files on disk. These files are preserved until the corresponding RDDs are no longer used and are garbage collected.

RDD Persistence

Persisting (caching) allows future actions to be much faster (often more than 10x). Caching is a key tool for interactive algorithms and fast interactive use.

```
rdd.persist(StorageLevel.MEMORY_ONLY());
```

MEMORY_ONLY	Store in memory. Recompute as needed
MEMORY_AND_DISK	
MEMORY_ONLY_SER	
MEMORY_AND_DISK_SER	
DISK_ONLY	
MEMORY_ONLY_2	Replicate each partition on two nodes
MEMORY_AND_DISK_2	
OFF_HEAP	