Slide 17
Now I want to get a bit into RDD persistence. You have seen this used already. That is the cache function. The cache function is actually the default of the persist function with the MEMORY_ONLY storage.

One of the key capability of Spark is its speed through persisting or caching. Each node stores any partitions of the cache and computes it in memory. When a subsequent action is called on the same dataset, or a derived dataset, it uses it from memory instead of having to retrieve it again. Future actions in such cases are often 10 times faster. The first time a RDD is persisted, it is kept in memory on the node. Caching is fault tolerant because if it any of the partition is lost, it will automatically be recomputed using the transformations that originally created it.

There are two methods to invoke RDD persistence. persist() and cache(). The persist() method allows you to specify a different storage level of caching. For example, you can choose to persist the data set on disk, persist it in memory but as serialized objects to save space, etc. Again the cache() method is just the default way of using persistence by storing deserialized objects in memory.

The table here shows the storage levels and what it means. Basically, you can choose to store in memory or memory and disk. If a partition does not fit in the specified cache location, then it will be recomputed on the fly. You can also decide to serialized the objects before storing this. This is space efficient, but will require the RDD to deserialized before it can be read, so it takes up more CPU workload. There's also the option to replicate each partition on two cluster nodes. Finally, there is an experimental storage level storing the serialized object in Tachyon. This level reduces garbage collection overhead and allows the executors to be smaller and to share a pool of memory. You can read more about this on Spark's website.

Slide 18
A lot of text on this page, but don't worry. It can be used as a reference when you have to decide the type of storage level.

There are tradeoffs between the different storage levels. You should analyze your current situation to decide which level works best. You can find this information here on Spark's website.

Basically if your RDD fits within the default storage level, by all means, use that.

It is the fastest option to fully take advantage of Spark's design. If not, you can serialized the RDD and use the MEMORY_ONLY_SER level. Just be sure to choose a fast serialization library to make the objects more space efficient and still reasonably fast to access.

Don't spill to disk unless the functions that compute your datasets are expensive or it requires a large amount of space.

If you want fast recovery, use the replicated storage levels. All levels are fully fault tolerant, but would still require the recomputing of the data. If you have a replicated copy, you can continue to work while Spark is reconstruction a lost partition.

Finally, use Tachyon if your environment has high amounts of memory or multiple applications. It allows you to share the same pool of memory and significantly reduces garbage collection costs. Also, the cached data is not lost if the individual executors crash.

Slide 19
On these last two slides, I'll talk about Spark's shared variables and the type of operations you can do on key-value pairs.

Spark provides two limited types of shared variables for common usage patterns: broadcast variables and accumulators. Normally, when a function is passed from the driver to a worker, a separate copy of the variables are

used for each worker. Broadcast variables allow each machine to work with a read-only variable cached on each machine. Spark attempts to distribute broadcast variables using efficient algorithms. As an example, broadcast variables can be used to give every node a copy of a large dataset efficiently.

The other shared variables are accumulators. These are used for counters in sums that works well in parallel. These variables can only be added through an associated operation. Only the driver can read the accumulators value, not the tasks. The tasks can only add to it. Spark supports numeric types but programmers can add support for new types. As an example, you can use accumulator variables to implement counters or sums, as in MapReduce.

Last, but not least, key-value pairs are available in Scala, Python and Java. In Scala, you create a key-value pair RDD by typing val pair = ('a', 'b'). To access each element, invoke the ._ notation. This is not zero-index, so the ._1 will return the value in the first index and ._2 will return the value in the second index. Java is also very similar to Scala where it is not zero-index. You create the Tuple2 object in Java to create a key-value pair. In Python, it is a zero-index notation, so the value of the first index resides in index 0 and the second index is 1.

Slide 20
There are special operations available to RDDs of key-value pairs. In an application, you must remember to import the SparkContext package to use PairRDDFunctions such as reduceByKey.

The most common ones are those that perform grouping or aggregating by a key. RDDs containing the Tuple2 object represents the key-value pairs. Tuple2 objects are simply created by writing (a, b) as long as you import the library to enable Spark's implicit conversion.

If you have custom objects as the key inside your key-value pair, remember that you will need to provide your own equals() method to do the comparison as well as a matching hashCode() method.

So in the example, you have a textFile that is just a normal RDD. Then you perform some transformations on it and it creates a PairRDD which allows it to invoke the reduceByKey method that is part of the PairRDDFunctions API.

I want to spend a little bit of time here to explain some of the syntax that you see on the slide. Note that in the first reduceByKey example with the a,b => a + b. This simply means that for the values of the same key, add them up together. In the example on the bottom of the slide, reduceByKey(_+_) uses the shorthand for anonymous function taking two parameters (a and b in our case) and adding them together, or multiplying, or any other operations for that matter.

Another thing I want to point is that for the goal of brevity, all the functions are concatenated on one line. When you actually code it yourself, you may want split each of the functions up. For example, do the flatMap operation and return that to a RDD. Then from that RDD, do the map operation to create another RDD. Then finally, from that last RDD, invoke the reduceByKey method. That would yield multiple lines, but you would be able to test each of the transformation to see if it worked properly.

Slide 21
So having completed this lesson, you should now be able to describe pretty well, RDDs. You should also understand how to create RDDs using various methods including from existing datasets, external datasets such as a textFile or from HDFS, or even just from existing RDDs. You saw various RDD operations and saw how to work with shared variables and key-value pairs.

Slide 22
Next steps. Complete lab exercise #2 Working with RDD operations. Then proceed to the next lesson in this course.