Slide 5:
Here we have a picture of the Spark unified stack. As you can see, the Spark core is at the center of it all. The Spark core is a general-purpose system providing scheduling, distributing, and monitoring of the applications across a cluster. Then you have the components on top of the core that are designed to interoperate closely, letting the users combine them, just like they would any libraries in a software project. The benefit of such a stack is that all the higher layer components will inherit the improvements made at the lower layers. Example: Optimization to the Spark Core will speed up the SQL, the streaming, the machine learning and the graph processing libraries as well. The Spark core is designed to scale up from one to thousands of nodes. It can run over a variety of cluster managers including Hadoop YARN and Apache Mesos. Or simply, it can even run as a standalone with its own built-in scheduler.

Spark SQL is designed to work with the Spark via SQL and HiveQL (a Hive variant of SQL). Spark SQL allows developers to intermix SQL with Spark's programming language supported by Python, Scala, and Java.

Spark Streaming provides processing of live streams of data. The Spark Streaming API closely matches that of the Sparks Core's API, making it easy for developers to move between applications that processes data stored in memory vs arriving in real-time. It also provides the same degree of fault tolerance, throughput, and scalability that the Spark Core provides.

Machine learning, MLlib is the machine learning library that provides multiple types of machine learning algorithms. All of these algorithms are designed to scale out across the cluster as well.

GraphX is a graph processing library with APIs to manipulate graphs and performing graph-parallel computations.

Slide 6:
Here's a brief history of Spark. I'm not going to spend too much time on this as you can easily find more information for yourself if you are interested. Basically, you can see that MapReduce started out over a decade ago. MapReduce was designed as a fault tolerant framework that ran on commodity systems. Spark comes out about a decade later with the similar framework to run data processing on commodity systems also using a fault tolerant framework. MapReduce started off as a general batch processing system, but there are two major limitations. 1) Difficulty in programming directly in MR and 2) Batch jobs do not fit many use cases. So this spawned specialized systems to handle other use cases. When you try to combine these third party systems in your applications, there are a lot of overhead.

Taking a looking at the code size of some applications on the graph on this slide, you can see that Spark requires a considerable amount less. Even with Spark's libraries, it only adds a small amount of code due to how tightly everything is integrated with very little overhead. There is great value to be able to express a wide variety of use cases with few lines of code.

Slide 7:
Now let's get into the core of Spark. Spark's primary core abstraction is called Resilient Distributed Dataset or RDD. Essentially it is just a distributed collection of elements that is parallelized across the cluster. You can have two types of RDD operations. Transformations and Actions. Transformations are those that do not return a value. In fact, nothing is evaluated during the definition of these transformation statements. Spark just creates these Direct Acyclic Graphs or DAG, which will only be evaluated at runtime. We call this lazy evaluation.

The fault tolerance aspect of RDDs allows Spark to reconstruct the transformations used to build the lineage to get back the lost data.

Actions are when the transformations get evaluted along with the action that is called for that RDD. Actions return values. For example, you can do a count on a RDD, to get the number of elements within and that value is returned.

So you have an image of a base RDD shown here on the slide. The first step is loading the dataset from Hadoop. Then uou apply successive transformations on it such as filter, map, or reduce. Nothing actually happens until an action is

called. The DAG is just updated each time until an action is called. This provides fault tolerance. For example, let's say a node goes offline. All it needs to do when it comes back online is to re-evaluate the graph to where it left off.

Caching is provided with Spark to enable the processing to happen in memory. If it does not fit in memory, it will spill to disk.