Let’s talk about some IBM distinctives that come with BigInsights. The bulk of BigInsights are the Hadoop open source components. But there are some components of BigInsights that are unique to IBM. The BigInsights Console is an IBM distinctive. And so is GPFS – FPO and Adaptive MapReduce. GPFS – FPO is General Parallel File System – File Placement Option. It is a replacement for HDFS. Adaptive MapReduce is designed to better performance and availability when compared to Apache MapReduce. Both of these components are options to be installed during the installation of BigInsights.

GPFS was originally developed for SAN file systems. It is able to handle thousands of node and petabytes of storage all with high performance. It is POSIX compliant and has the following enterprise features:

- Security
- Access Control Lists (ACLs)
- Snapshots
- Backup and restore
- Archiving
- Caching
- Replication

It has workload isolation, meaning that you can define storage pools to make sure that data for a particular application only resides on specific nodes. This also allows you to treat individual sets of data differently. In Hadoop, using HDFS, all data is treated the same. HDFS is designed to spread the data in each file to every node in the cluster and to replicate all data the same way. For that reason, a local file system is required to store temporary data. But the storage pool capabilities in GPFS, allow you to have just a single file system since not all of your data has to be treated the same.

GPFS, as originally developed for SAN environments, assumed that data was going to be stored on network devices and that access to all data would be across the network. That is not an ideal situation for Hadoop. So GPFS - FPO was developed. Based upon GPFS, it was adapted to a shared nothing cluster environment where locality of data was important and the failure of components would be fairly common.

GPFS - FPO allows different block sizes. Hadoop was originally conceived to process huge amounts of machine logging data. And then it was adopted to process other types of large amounts of data. Since the data was being processed sequentially, block sizes of 64MB or 128MB were used to limit the overhead of starting and stopping MapReduce tasks. Large block sizes also had the effect of lessening the metadata stored in the NameNode. Initially it made sense to define a single block size for the entire Hadoop file system. But as Hadoop is applied to different types of analysis, the files being process are not necessarily very large. To force these files into a single block sizes of 64MB or 128MB may not be prudent.

GPFS - FPO introduces the concept of a block group factor. Now you can define your block size appropriately to handle small files or index lookup files. But then, using the block group factor, you can *group* a number of blocks together for a larger *effective* block size. With this approach files can have their own block size and are not forced to use a system-wide block size, although it is possible to define a system-wide block group factor.

GPFS - FPO does not use a NameNode, like you see with HDFS. The NameNode in HDFS is a single point of failure. Rather, GPFS - FPO distributes the metadata across multiple nodes so that there is not a single point of failure. The metadata needs to be treated differently than the normal Hadoop data. As it turns out, metadata is better suited for the way that regular GPFS handles data, whereas the normal Hadoop data is better handled by the features of GPFS - FPO. So how do you get around this? Well GPFS - FPO has the concept of storage pools. Because of that, you can treat sets of data differently. Metadata should be assigned to a storage pool with a property of no write affinity. And the normal Hadoop data should be assigned to a storage pool with a property of write affinity.

Let’s talk about the GPFS cluster and file system concepts. In this visual, there are three nodes that have metadata and all nodes can have application data. There are also three nodes that participate in the quorum. (Quorum nodes are specified by you.) Depending on the size of the cluster, there can be as many as seven quorum nodes. If there are any detected problems, the quorum members do something about it. For example, assume that there was a problem with the File System Manager node. The quorum members would elect a new FSM. The same thing would happen with the Cluster Manger. If the Primary Configuration Server were to go offline, then the quorum members would tell the Secondary Configuration Server to take over as the primary.

We talked about there not being a NameNode to act as a single point of failure. In a GPFS system, each node is able to access data that is on another node. So the replicated metadata is accessible by all nodes. Even if a node that has metadata fails, that same metadata can be read from another node.

What happens if a quorum node fails? As long as there is a quorum (more than half of the number of quorum nodes), the system can continue to run. One of the purposes for the quorum nodes is to make sure that the system does not get into what is referred to as a *split brain* situation. What if the switch failed? Now you have quorum members that cannot communicate with other members. You surely do not want to have both racks doing updates independently of each other, each acting as though they are the only brain. So in this scenario, the quorum member on rack 1 would not be able to communicate with the members on rack 2. The quorum members on rack 2 can still communicated with each other so they have a quorum and continue to process. The quorum member on rack 1 determines that it is all alone and does not have a quorum. It then shuts down.

Let’s look at emerging workload patterns. Although Hadoop and MapReduce are batch oriented, there is a greater desire for near real time analytics. There are already a number of program languages, like Pig and Jaql available in the Hadoop arena that simplify the coding of MapReduce applications. What you code in a high level language gets distilled, under the covers, into MapReduce programs. Taking that farther, you have SQL access to Hadoop data in the form of Hive and Big SQL that also convert the high level SQL statements into MapReduce programs. When using these languages, programmers do not want to wait five to ten minutes for the results of a query. If you are querying multiple terabytes of data, then you can expect to wait on your results. But Hadoop is now being used for files that are much smaller in size. As queries are run against smaller datasets, the overhead of Apache MapReduce becomes more noticeable. BigInsights' Adaptive MapReduce is designed to address that type of problem.

One of the goals for Adaptive MapReduce is for low latency task scheduling. The default for task scheduling for Apache MapReduce is 30 seconds. This involves the TaskTacker contacting the JobTracker to see if there is a workload on which it can work. If there is not, then the TaskTracker will wait another 30 seconds to check to see if there is any work for it. With Adaptive MapReduce the model changes from a pull model, where the TaskTracker is pulling scheduling information from the JobTracker, to a push model, where the scheduling information is pushed from a central location to the workers. Where you have small job workloads, with Apache MapReduce, you would have a lot of starts and stops by the TaskTrackers. The Adaptive MapReduce approach allows a single TaskTracker, after once starting, to keep working on multiple tasks.

A second part of Adaptive MapReduce is the redesigning of the shuffling algorithm. The new algorithm will, when possible, skip data being spilled to disk before the data is passed to the reduce task.

There is also higher availability with Adaptive MapReduce. With Apache MapReduce, the workload scheduling and the resource scheduling are two different layers but they are tightly coupled. (The TaskTracker sends it current state back to the JobTracker and the JobTracker decides how to dispatch the workloads.) Adaptive MapReduce separates these two layers. The worker nodes report back to the central site that they are available for work. The workload scheduler then pushes out the workload to any idle worker nodes. This allows the central site to have the status of all worker nodes and therefore the central site, if need be, can be moved to another node in the cluster. In this way, it reduces the single point of failure for the JobTracker.

This all was accomplished with staying Apache MapReduce compatible.

Please continue with the next video in this lesson.