**Slide 2: Agenda**

"We'll be covering three key aspects today: first, we'll get to understand the Hadoop Distributed File System, also known as HDFS. We'll then move on to MapReduce, its fundamental principles, and its pivotal role in processing Big Data. Lastly, we'll engage in a hands-on exercise where we'll write and execute our own MapReduce programs."

---

**Slide 3: HDFS Introduction**

"HDFS, short for Hadoop Distributed File System, is designed to run on commodity hardware and to be highly fault-tolerant. It forms the backbone of many big data processing tasks, given its ability to manage data across many servers in a cluster.

As you know, in big data processing, it's common to handle petabytes of data. Storing such vast amounts of data on a single disk is impractical, and even if we did, reading the data off a single disk would be too time-consuming for most real-time applications. So HDFS follows a distributed approach to data management.

It breaks the data into blocks, usually of size 128 MB or 256 MB, and stores each block of data on a separate machine in the cluster. This practice ensures that the data is distributed across the cluster, allowing it to be processed faster since the data can be read off the disks on multiple machines in parallel.

Moreover, HDFS also replicates each block of data on multiple machines to ensure high availability and fault tolerance. So, if one machine fails, the data is available on another machine. These replication techniques also make HDFS resilient to network failures or data corruption."

---

**Slide 4: HDFS Architecture**

"HDFS operates on a Master-Worker architecture. The Master, also known as the NameNode, manages the filesystem metadata. It keeps track of all the files in the system, and tracks the file data's location across the cluster. The wrokers in this architecture, known as the DataNodes, are responsible for storing and retrieving data blocks when they are told to do so by clients or the NameNode."

---

**Slide 5: How Data is Written to HDFS**

"When data is written to HDFS, a series of intricate steps unfold. Initially, the client communicates with the NameNode and requests to create a new file. The NameNode performs various checks like ensuring the file doesn't already exist and that the client has the right permissions to write to the directory. If all checks pass, the NameNode inserts the file name into the file system hierarchy and allocates a new block for the file.

The new block is identified by a block ID, a globally unique ID across the HDFS cluster. NameNode returns to the client with the ID of the block and a list of DataNode addresses where it should write the data. This list represents the DataNodes that will hold copies of this block. It's worth noting that the NameNode does not directly send data to DataNodes; instead, it instructs the client to write the block to specific DataNodes.

The client then writes the data to the DataNodes in a pipeline fashion, meaning the data is written to the first DataNode, which relays the same data to the second, and so on. This approach enables high data throughput. Upon successfully writing data to the DataNode pipeline, the client communicates to the NameNode that the block was written. The NameNode then finalizes the file creation process by recording the size of the block. If more blocks are needed because the client has more data to write, the process is repeated."

---

**Slide 6:**

"HDFS offers a command-line interface known as FS shell which allows users to interact with data stored in Hadoop. This shell commands are modeled after the UNIX file system commands, so they're easy to learn if you're familiar with UNIX or Linux command line operations. But even if you aren't, they're straightforward and intuitive.

Let's go through some common operations that you can perform on HDFS using this shell.

First, to list files and directories in the root of your HDFS, you would use the command hadoop fs -ls /. This command is very similar to the UNIX command for listing files. Here -ls stands for listing, and the slash / is the directory whose files you want to list.

Next, to create a directory within HDFS, you can use the command hadoop fs -mkdir /test. Here -mkdir stands for 'make directory', and /test is the name of the directory being created. This command again closely mirrors its UNIX counterpart.

Often, we'll want to move data from our local file system into HDFS. To do this, we use the -put command. For example, if we have a file on our local system named 'localfile.txt' that we want to put into our newly created directory /test in HDFS, we would use the command hadoop fs -put localfile.txt /test. This command copies the specified file into the designated directory within HDFS.

Lastly, if you want to read the contents of a file stored in HDFS from the command line, you can use the -cat command. For example, to read the contents of 'localfile.txt' that we just stored in HDFS, we would use the command hadoop fs -cat /test/localfile.txt.

As you can see, Hadoop provides a robust and user-friendly interface to interact with data stored in HDFS, making it easy to manipulate and manage your data in a distributed environment."

--

**Slide 7: MapReduce Introduction**

"MapReduce is a fundamental concept in big data processing. Named after the two primary operations, the 'Map' operation, and the 'Reduce' operation, it's a programming model that allows us to process large datasets across distributed clusters.

In the 'Map' step, the input data, which can be in the order of terabytes or even petabytes, is divided into independent chunks. These chunks are processed in parallel by the map tasks in a completely distributed manner. The map function takes input pairs and produces a set of intermediate key-value pairs. For instance, if you're counting the frequency of words in a text corpus, the map function emits each word and an associated count of 1.

After the 'Map' phase, we have the 'Shuffle and Sort' phase, where the output of the map tasks is shuffled across the reduce tasks, and the input to every reduce task is sorted.

The 'Reduce' step then takes these intermediate key-value pairs and combines the values corresponding to the same intermediate key. In our word frequency example, the reduce function sums up the counts for each word and emits a single output of the word and its total count.

This MapReduce model, with its ability to scale up to thousands of servers, forms the core of many big data tasks. It's used in various domains, from processing web page text to analyzing log files, and from indexing web pages to machine learning and statistical modeling."

---

**Slide 8: MapReduce Workflow**

"The MapReduce workflow, designed to process large data sets in a distributed and parallel manner, is composed of three stages. The 'Map' stage is where the input data is divided into smaller sub-problems, and a map function is applied to each sub-problem. This function takes in a set of key-value pairs as input and produces an intermediate set of key-value pairs.

The 'Shuffle and Sort' stage is an implicit phase between the Map and Reduce stages. The output from the Map function, i.e., the key-value pairs, are shuffled, sorted, and sent to the reducers. This stage involves several vital tasks. The framework assigns key-value pairs to reducers based on the keys' hashcodes. The pairs are then sorted by key, ensuring all pairs with the same key end up at the same reducer.

In the final 'Reduce' stage, the reduce function is applied to each set of values having the same key, generating a final output. The reduce function takes the intermediate key-value pairs from the 'Shuffle and Sort' stage and merges those values together to form a smaller set of values. In essence, it summarizes the large set of intermediate data values associated with a unique key."

---

**Slide 9: MapReduce Example: Word Count**

"Let's consider the classic Word Count problem to understand MapReduce better. In this problem, we aim to count the frequency of each word in a given text. The Map function scans the text line by line and outputs a key-value pair for each word encountered. The key here is the word itself, and the value is the count, which is initially set to one."

---

**Slide 10: Map Function Code in Python**

"The Map function's purpose is to process input data and produce key-value pairs. In our Python implementation, this function reads input data from standard input. The input data is a line from our text file. We use Python's 'sys.stdin' to read from standard input.

The function then uses the 'split()' method to break the line into words. It is vital to note that the 'split()' function uses space as a default delimiter. However, for real-world data, one might need to consider other delimiters or even regular expressions to correctly tokenize the data.

The function finally uses a 'for' loop to iterate over each word and writes a key-value pair to standard output, where the key is the word and the value is 1. We use Python's 'sys.stdout.write' to write to standard output. This simple function gives us an intermediate output of word-count pairs for each word in the input data."

---

**Slide 11: Reduce Function**

"The Reduce function takes over once all the Map tasks are finished. It accepts these key-value pairs where the key is a word and the value is a list of counts. The Reduce function sums up the counts for each word and emits a key-value pair with the word and the total count."

---

**Slide 12: Reduce Function Code in Python**

"The Reduce function's role is to take the intermediate output from the Map function, sort it, and then reduce it to produce the final output. Our Python implementation starts by creating an empty Python dictionary 'word2count' to hold word-count pairs.

The function then reads from standard input using 'sys.stdin'. It reads one line at a time, where each line represents a key-value pair from the Map function. For each line, it splits the line into word and count using the 'split()' method.

Next, it checks if the word is already present in the 'word2count' dictionary. If it is, it adds the new count to the existing count. If it's not, it adds a new entry to the dictionary with the word as the key and the count as the value. It's important to note that in our simple example, the count is always 1, but in more complex scenarios, the Map function could emit counts greater than 1.

Finally, the function writes the contents of the 'word2count' dictionary to standard output using 'sys.stdout.write'. Each output line contains a word and its total count separated by a tab. This

---

**Slide 13: Executing MapReduce Programs**

"We can run MapReduce jobs using the Hadoop Streaming API. This API allows us to use any executable or script as the mapper and reducer. In this case, we are using the Python scripts we wrote for the Map and Reduce functions. The command here first specifies the locations of the map and reduce scripts, followed by the input and output paths on HDFS."

---

**Slide 14: What Happens When a Python MapReduce Job is Submitted**

"When we submit a MapReduce job, the client application first submits the job to the JobClient, which then communicates with the JobTracker (in YARN, this would be the ResourceManager). The JobTracker creates a new job ID and records the job details in memory and in a system directory. The input data is divided into splits, and a Map task is created for each split. The JobTracker communicates with the TaskTrackers (in YARN, these are NodeManagers) to allocate Map tasks. The data locality principle is followed where possible, i.e., Map tasks are allocated to TaskTrackers running on the same node where the data resides. Each TaskTracker runs the Map task as a separate process and periodically reports the progress back to the JobTracker. Once all Map tasks are complete, the JobTracker schedules Reduce tasks in a similar manner. Reduce tasks fetch the intermediate data from the Map tasks, sort it, and then apply the Reduce function. The final output of the Reduce tasks is written back to HDFS. The JobTracker then notifies the client about the job completion."

---

**Slide 15: Conclusion**

"In conclusion, HDFS and MapReduce are fundamental concepts in Hadoop, providing a robust framework for storing and processing large data sets across clusters of computers. Today's lecture has aimed to provide a comprehensive understanding of these principles, and through our hands-on exercise, we've managed to write and execute a MapReduce program."