Unit 3: Distributed Storage and Processing of Big Data

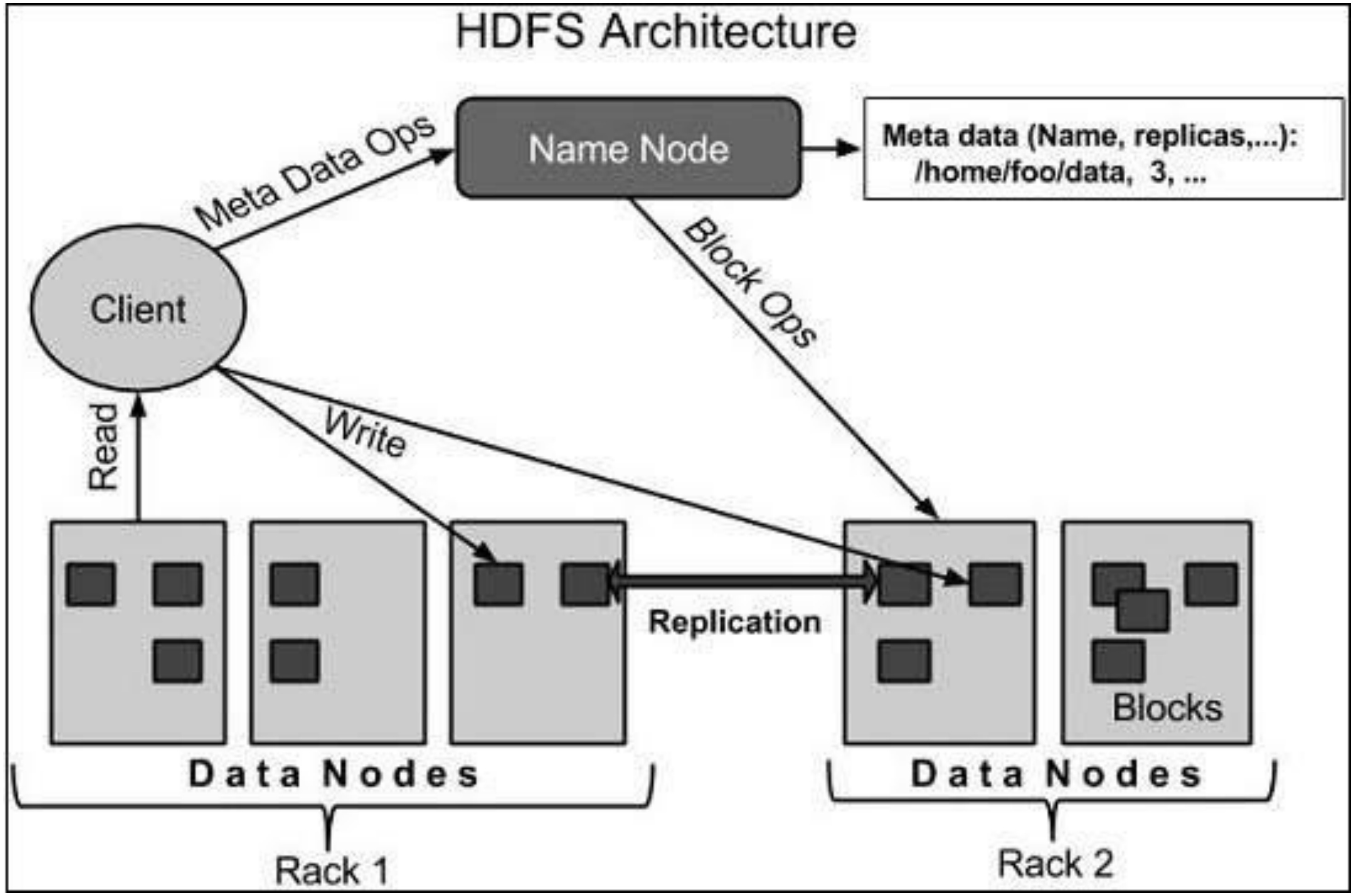
Design of HDFS

- Hadoop File System was developed using distributed file system design. It is run on commodity hardware. Unlike other distributed systems, HDFS is highly faulttolerant and designed using low-cost hardware.
- HDFS holds very large amount of data and provides easier access. To store such huge data, the files are stored across multiple machines. These files are stored in redundant fashion to rescue the system from possible data losses in case of failure. HDFS also makes applications available to parallel processing.

Features of HDFS

- It is suitable for the distributed storage and processing.
- Hadoop provides a command interface to interact with HDFS.
- The built-in servers of namenode and datanode help users to easily check the status of cluster.
- Streaming access to file system data.
- HDFS provides file permissions and authentication.

HDFS Architecture



- HDFS follows the master-slave architecture and it has the following elements.

Namenode

- The namenode is the commodity hardware that contains the GNU/Linux operating system and the namenode software. It is a software that can be run on commodity hardware. The system having the namenode acts as the master server and it does the following tasks −
- Manages the file system namespace.
- Regulates client’s access to files.
- It also executes file system operations such as renaming, closing, and opening files and directories.

Datanode

- The datanode is a commodity hardware having the GNU/Linux operating system and datanode software. For every node (Commodity hardware/System) in a cluster, there will be a datanode. These nodes manage the data storage of their system.
- Datanodes perform read-write operations on the file systems, as per client request.
- They also perform operations such as block creation, deletion, and replication according to the instructions of the namenode.

Block

- Generally the user data is stored in the files of HDFS. The file in a file system will be divided into one or more segments and/or stored in individual data nodes. These file segments are called as blocks. In other words, the minimum amount of data that HDFS can read or write is called a Block. The default block size is 64MB (version 1) and 128MB in version 2, but it can be increased as per the need to change in HDFS configuration.

Why HDFS?

- **Fault detection and recovery** − Since HDFS includes a large number of commodity hardware, failure of components is frequent. Therefore HDFS should have mechanisms for quick and automatic fault detection and recovery.
- **Huge datasets** − HDFS should have hundreds of nodes per cluster to manage the applications having huge datasets.
- **Hardware at data** − A requested task can be done efficiently, when the computation takes place near the data. Especially where huge datasets are involved, it reduces the network traffic and increases the throughput.

Hadoop file system interfaces,

- Hadoop is an open-source software framework written in Java along with some shell scripting and C code for performing computation over very large data. Hadoop is utilized for batch/offline processing over the network of so many machines forming a physical cluster. The framework works in such a manner that it is capable enough to provide distributed storage and processing over the same cluster. It is designed to work on cheaper systems commonly known as commodity hardware where each system offers its local storage and computation power.
- Hadoop is capable of running various file systems and HDFS is just one single implementation that out of all those file systems. The Hadoop has a variety of file systems that can be implemented concretely. The Java abstract class ***org.apache.hadoop.fs.FileSystem*** represents a file system in Hadoop.

Hadoop file system interfaces

Filesystem	**URI scheme**	**Java implementation (all under org.apache.hadoop)**	**Description**
Local	file	fs.LocalFileSystem	The Hadoop Local filesystem is used for a locally connected disk with client-side checksumming. The local filesystem uses RawLocalFileSystem with no checksums.
HDFS	hdfs	hdfs.DistributedFileSystem	HDFS stands for Hadoop Distributed File System and it is drafted for working with MapReduce efficiently.
HFTP	hftp	hdfs.HftpFileSystem	The HFTP filesystem provides read-only access to HDFS over HTTP. There is no connection of HFTP with FTP. This filesystem is commonly used with ***distcp*** to share data between HDFS clusters possessing different versions.
HSFTP	hsftp	hdfs.HsftpFileSystem	The HSFTP filesystem provides read-only access to HDFS over HTTPS. This file system also does not have any connection with FTP.
HAR	har	fs.HarFileSystem	The HAR file system is mainly used to reduce the memory usage of NameNode by registering files in Hadoop HDFS. This file system is layered on some other file system for archiving purposes.

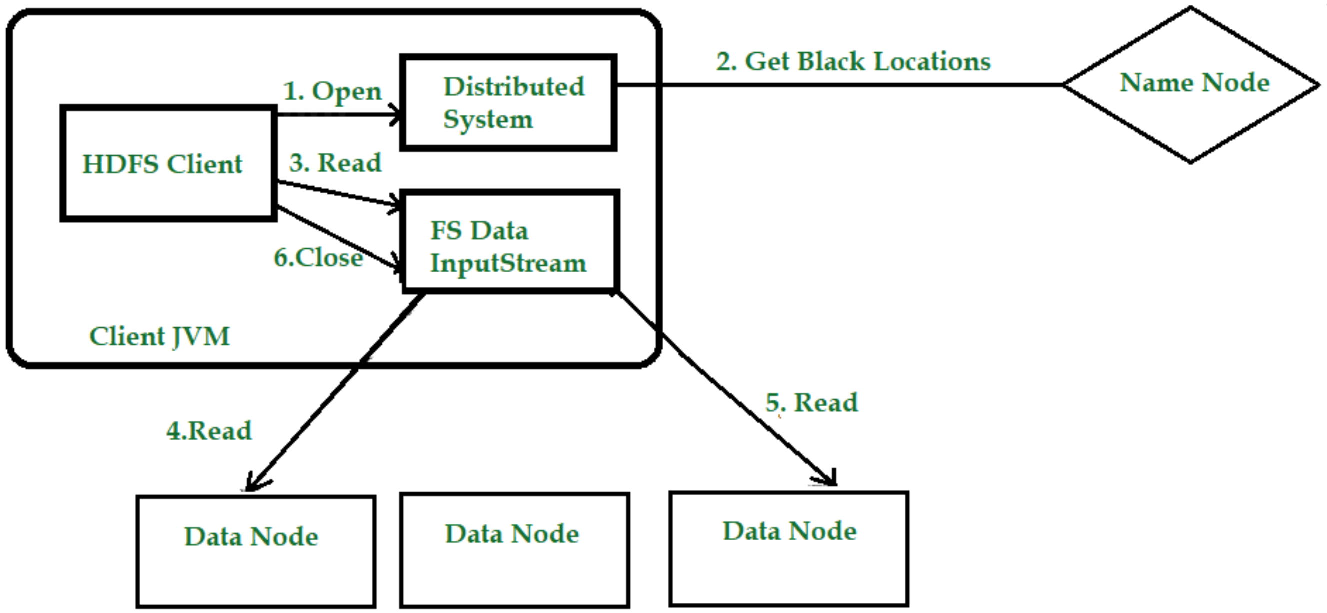
KFS (Cloud-Store)	kfs	fs.kfs.KosmosFileSystem	cloud store or KFS(KosmosFileSystem) is a file system that is written in c++. It is very much similar to a distributed file system like HDFS and GFS(Google File System).
FTP	ftp	fs.ftp.FTPFileSystem	The FTP filesystem is supported by the FTP server.
S3 (native)	s3n	fs.s3native.NativeS3FileSystem	This file system is backed by AmazonS3.
S3 (block-based)	s3	fs.s3.S3FileSystem	S3 (block-based) file system which is supported by Amazon s3 stores files in blocks(similar to HDFS) just to overcome S3's file system 5 GB file size limit.

Hadoop file system interfaces,

- Hadoop gives numerous interfaces to its various filesystems, and it for the most part utilizes the URI plan to pick the right filesystem example to speak with. You can use any of this filesystem for working with MapReduce while processing very large datasets but distributed file systems with data locality features are preferable like HDFS and KFS(KosmosFileSystem).
- Reference: https://www.geeksforgeeks.org/various-filesystems-in-hadoop/

Data flow of HDFS or question can be like: Anatomy of read/write in HDFS (VVVI)

Anatomy of File Read in HDFS



Anatomy of File Read in HDFS

- **Step 1:** The client opens the file it wishes to read by calling open() on the File System Object(which for HDFS is an instance of Distributed File System).
- **Step 2:** Distributed File System(DFS) calls the name node, using remote procedure calls (RPCs), to determine the locations of the first few blocks in the file. For each block, the name node returns the addresses of the data nodes that have a copy of that block. The DFS returns an FSDataInputStream to the client for it to read data from. FSDataInputStream in turn wraps a DFSInputStream, which manages the data node and name node I/O.
- **Step 3:** The client then calls read() on the stream. DFSInputStream, which has stored the info node addresses for the primary few blocks within the file, then connects to the primary (closest) data node for the primary block in the file.

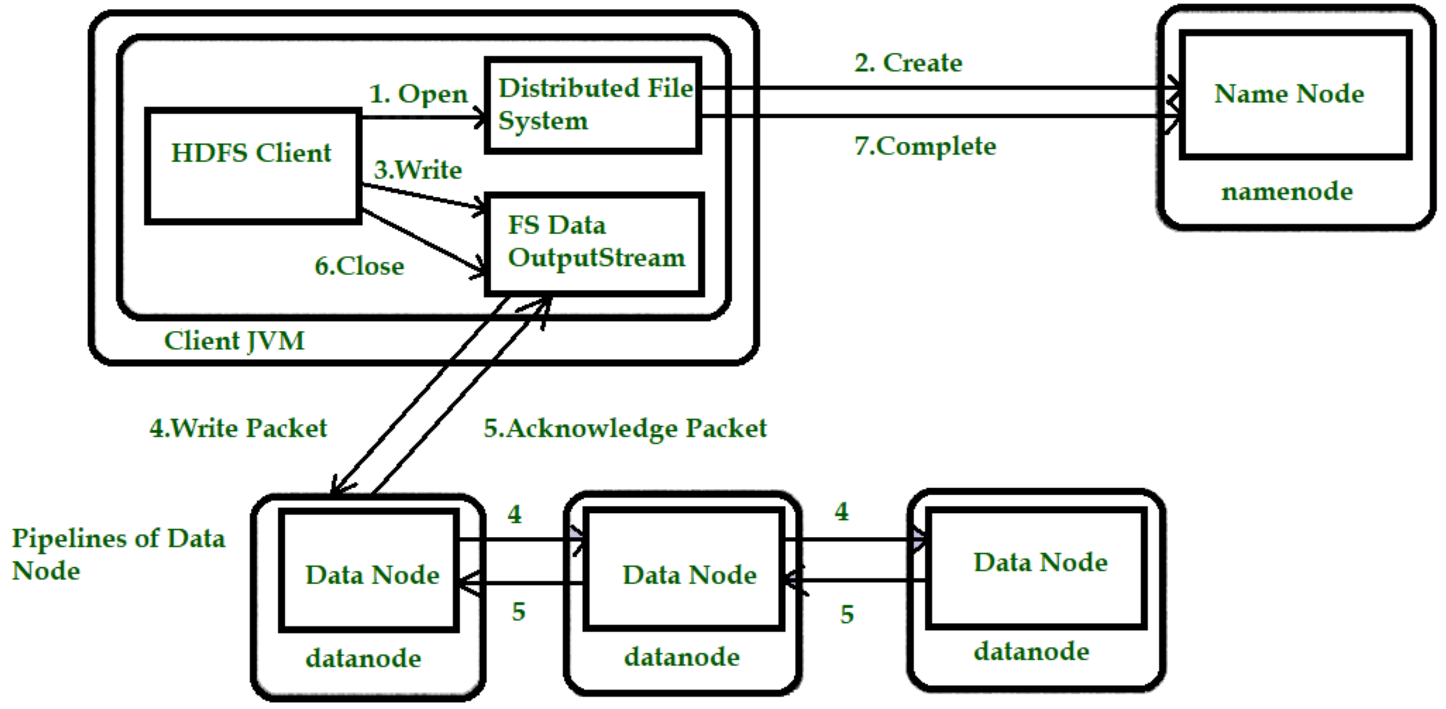
Anatomy of File Read in HDFS

- **Step 4:** Data is streamed from the data node back to the client, which calls read() repeatedly on the stream.
- **Step 5:** When the end of the block is reached, DFSInputStream will close the connection to the data node, then finds the best data node for the next block. This happens transparently to the client, which from its point of view is simply reading an endless stream. Blocks are read as, with the DFSInputStream opening new connections to data nodes because the client reads through the stream. It will also call the name node to retrieve the data node locations for the next batch of blocks as needed.
- **Step 6:** When the client has finished reading the file, a function is called, close() on the FSDataInputStream.

Anatomy of File Write in HDFS

- **Note:** HDFS follows the Write once Read many times model. In HDFS we cannot edit the files which are already stored in HDFS, but we can append data by reopening the files.

Anatomy of File Write in HDFS



Anatomy of File Write in HDFS

- **Step 1:** The client creates the file by calling create() on DistributedFileSystem(DFS).
- **Step 2:** DFS makes an RPC call to the name node to create a new file in the file system's namespace, with no blocks associated with it. The name node performs various checks to make sure the file doesn't already exist and that the client has the right permissions to create the file. If these checks pass, the name node prepares a record of the new file; otherwise, the file can't be created and therefore the client is thrown an error i.e. IOException. The DFS returns an FSDataOutputStream for the client to start out writing data to.

Anatomy of File Write in HDFS

- **Step 3:** Because the client writes data, the DFSOutputStream splits it into packets, which it writes to an indoor queue called the info queue. The data queue is consumed by the DataStreamer, which is liable for asking the name node to allocate new blocks by picking an inventory of suitable data nodes to store the replicas. The list of data nodes forms a pipeline, and here we'll assume the replication level is three, so there are three nodes in the pipeline. The DataStreamer streams the packets to the primary data node within the pipeline, which stores each packet and forwards it to the second data node within the pipeline.

Anatomy of File Write in HDFS

- **Step 4:** Similarly, the second data node stores the packet and forwards it to the third (and last) data node in the pipeline.
- **Step 5:** The DFSOutputStream sustains an internal queue of packets that are waiting to be acknowledged by data nodes, called an “ack queue”.

Anatomy of File Write in HDFS

- **Step 6:** This action sends up all the remaining packets to the data node pipeline and waits for acknowledgments before connecting to the name node to signal whether the file is complete or not.

Data flow of HDFS or question can be like: Anatomy of read/write in HDFS (VVVI)

- HDFS follows Write Once Read Many models. So, we can't edit files that are already stored in HDFS, but we can include them by again reopening the file. This design allows HDFS to scale to a large number of concurrent clients because the data traffic is spread across all the data nodes in the cluster. Thus, it increases the availability, scalability, and throughput of the system.

Basic HDFS commands,

- **ls:** This command is used to list all the files. Use *lsr* for recursive approach. It is useful when we want a hierarchy of a folder.
- **Syntax:**

```
bin/hdfs dfs -ls <path>
```

Example:

It will print all the directories present in HDFS. bin directory contains executables so, *bin/hdfs* means we want the executables of hdfs particularly *dfs*(Distributed File System) commands.

Basic HDFS commands,

- Write about 10 hdfs commands with syntax example and description for exam.
- Goto this link for those commands:
    - https://www.geeksforgeeks.org/hdfs-commands/

MapReduce: Functional programing, MapReduce fundamentals,

- Check chapter 3.pdf

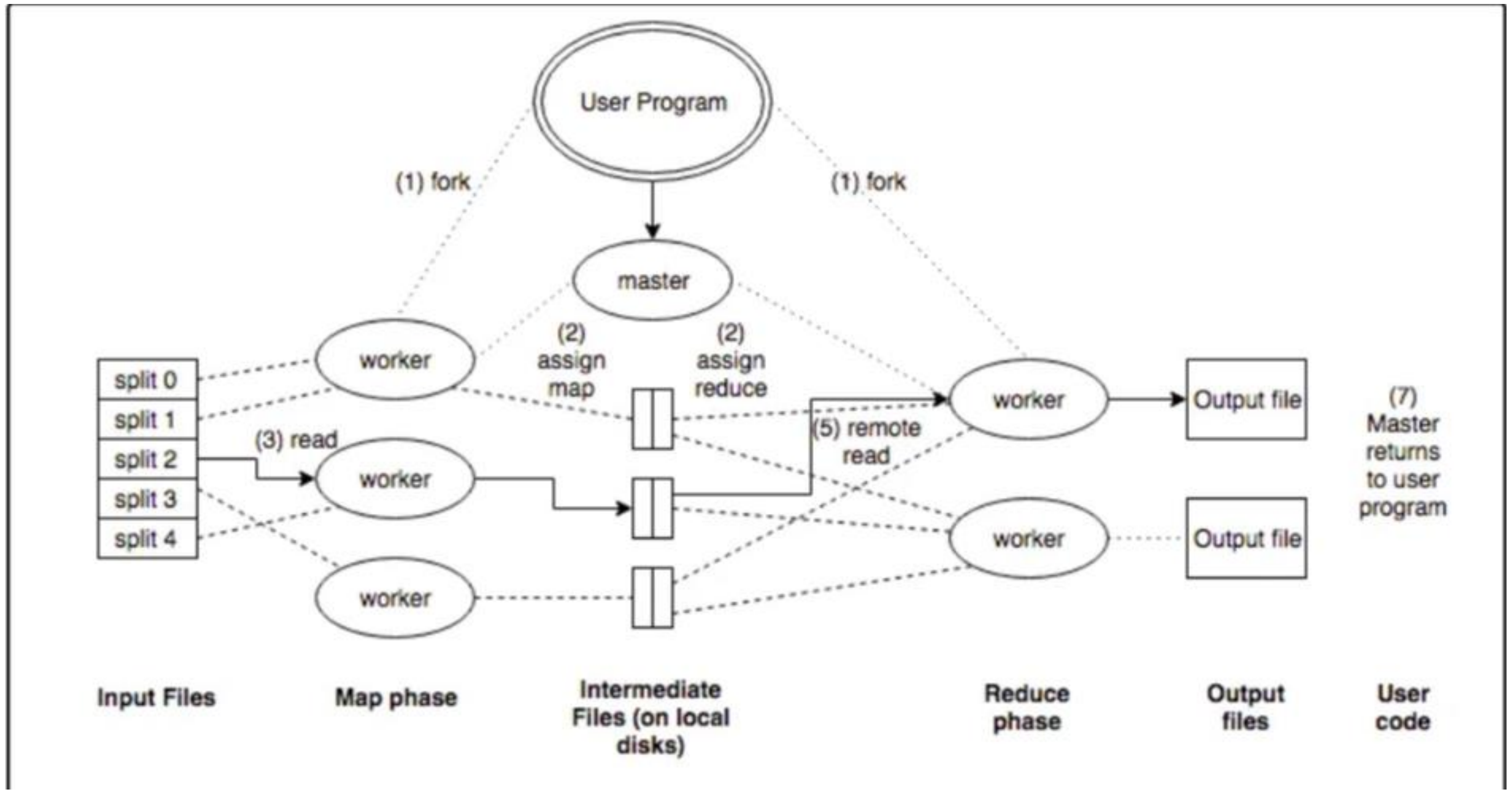
Execution overview of MapReduce (VVVI)

- The Map invocations are distributed across multiple machines by automatically partitioning the input data into a set of **M** splits or *shards,* which are what will be processed across the machines.

Execution overview of MapReduce (VVVI)

- Reduce invocations are distributed by partitioning the intermediate key space into **R** pieces using a partitioning function specified by the user.
- The following image depicts the overall flow sequence of MapReduce operations:

Execution overview of MapReduce (VVVI)



Execution overview of MapReduce (VVVI)

- The MapReduce library in the user program first shards the input files into M pieces of typically 16MB-64MB/piece. It then starts up many copies of the program on a cluster of machines.
- One of the the copies of the program is special: the master. The rest are workers that are assigned work by the master. There are M map tasks and R reduce tasks to assign. The master picks idle workers and assigns each one either an M or R task.

Execution overview of MapReduce (VVVI)

- A worker who is assigned a map task reads the content of the corresponding input shard. It parses key/value pairs out of the input data and passes each pair to the users-defined Map function. The intermediate K/V pairs produced are buffered in memory.
- Periodically, the buffered pairs are written to local disk, partitioned into R regions by the partitioning function. The locations of these buffered pairs on the local disk are passed back to master, who is responsible for forwarding these locations to the reduce workers.

Execution overview of MapReduce (VVVI)

- When a reduce worker gets the location from master, it uses remote calls to read the buffered data from the disks. When a reduce worker has read all intermediate data, it sorts it by the intermediate keys so all of the same occurrences are grouped together. (**note: if the amount of intermediate data is too large to fit in memory, an external sort is used)**

Execution overview of MapReduce (VVVI)

- The reduce worker iterates over the sorted intermediate data and for each unique intermediate key encountered, it passes the key and corresponding set of intermediate values to the user’s Reduce function. The output of the Reduce function is appended to a final output file for this reduce partition.
- When all map tasks and reduce tasks have been completed, the master wakes up the user program. At this point, the MapReduce call in the user program returns back to the user code.

Execution overview of MapReduce (VVVI)

- After successful completion, the output of the MapReduce execution is available in the R output files.
- To detect failure, the master pings every worker periodically. If no worker response after a certain point, the worker is marked a “failed” and all previous task work by that worker is reset, to become eligible for rescheduling on other workers.
- Completed map tasks are re-executed when failure occurs because their output is stored on the local disk(s) of the failed machine and therefore inaccessible. Completed reduce tasks do not need to be re-executed since their output is stored in a global file system.

MapReduce Examples

- Some examples of Map Reduce tasks:

- **Distributed Grep —** Map Function emits a line if a pattern is matched. The reduce function is an identity function that just copies the supplied intermediate data to the output.
- **Count of URL Access Frequency —** The map function processes logs of web page requests and outputs **<URL, 1>**. The reduce function adds together all values for the same URL and emits a **<URL, total count>** pair.

MapReduce Examples

- **Reverse Web-Link Graph —** The map function outputs **<target, source>** pairs for each link to a target URL found in a page named "source". The reduce function concatenates the list of all source URLs associated with a given target URL and emits the pair: **<target, list(source)>**.
- **Inverted Index —** The map function parses each document, and emits a sequence of **<word, document ID>** pairs. The reduce function accepts all pairs for a given word, sorts the corresponding document IDs and emits a **<word, list(document ID>** pair. The set of all output pairs forms a simple inverted index. It's also easy to augment this computation to keep track of the word *positions* as well.

You also need to give example of map reduce using word count problem.

Check chapter3.pdf for it.

Basic MapReduce API Concepts,

- we learn about the classes and methods used in MapReduce programming.

Basic MapReduce API Concepts,

- MapReduce Mapper Class
- In MapReduce, the role of the Mapper class is to map the input key-value pairs to a set of intermediate key-value pairs. It transforms the input records into intermediate records.
- These intermediate records associated with a given output key and passed to Reducer for the final output.

Methods of Mapper Class

void cleanup(Context context)	This method called only once at the end of the task.
void map(KEYIN key, VALUEIN value, Context context)	This method can be called only once for each key-value in the input split.
void run(Context context)	This method can be override to control the execution of the Mapper.
void setup(Context context)	This method called only once at the beginning of the task.

MapReduce Reducer Class

- In MapReduce, the role of the Reducer class is to reduce the set of intermediate values. Its implementations can access the Configuration for the job via the JobContext.getConfiguration() method.

Methods of Reducer Class

void cleanup(Context context)	This method called only once at the end of the task.
void map(KEYIN key, Iterable<VALUEIN> values, Context context)	This method called only once for each key.
void run(Context context)	This method can be used to control the tasks of the Reducer.
void setup(Context context)	This method called only once at the beginning of the task.

Anatomy of a MapReduce job run, Failures, Shuffle and sort, Task execution

- Give example of map reduce word count which is in chapter3.pdf

Anatomy of a MapReduce job run, Failures, Shuffle and sort, Task execution

- **Anatomy of a mapreduce job** run can be used to work with a solitary method call: submit() on a Job object (you can likewise call waitForCompletion(), which presents the activity on the off chance that it hasn’t been submitted effectively, at that point sits tight for it to finish).

Anatomy of a MapReduce job run, Failures, Shuffle and sort, Task execution

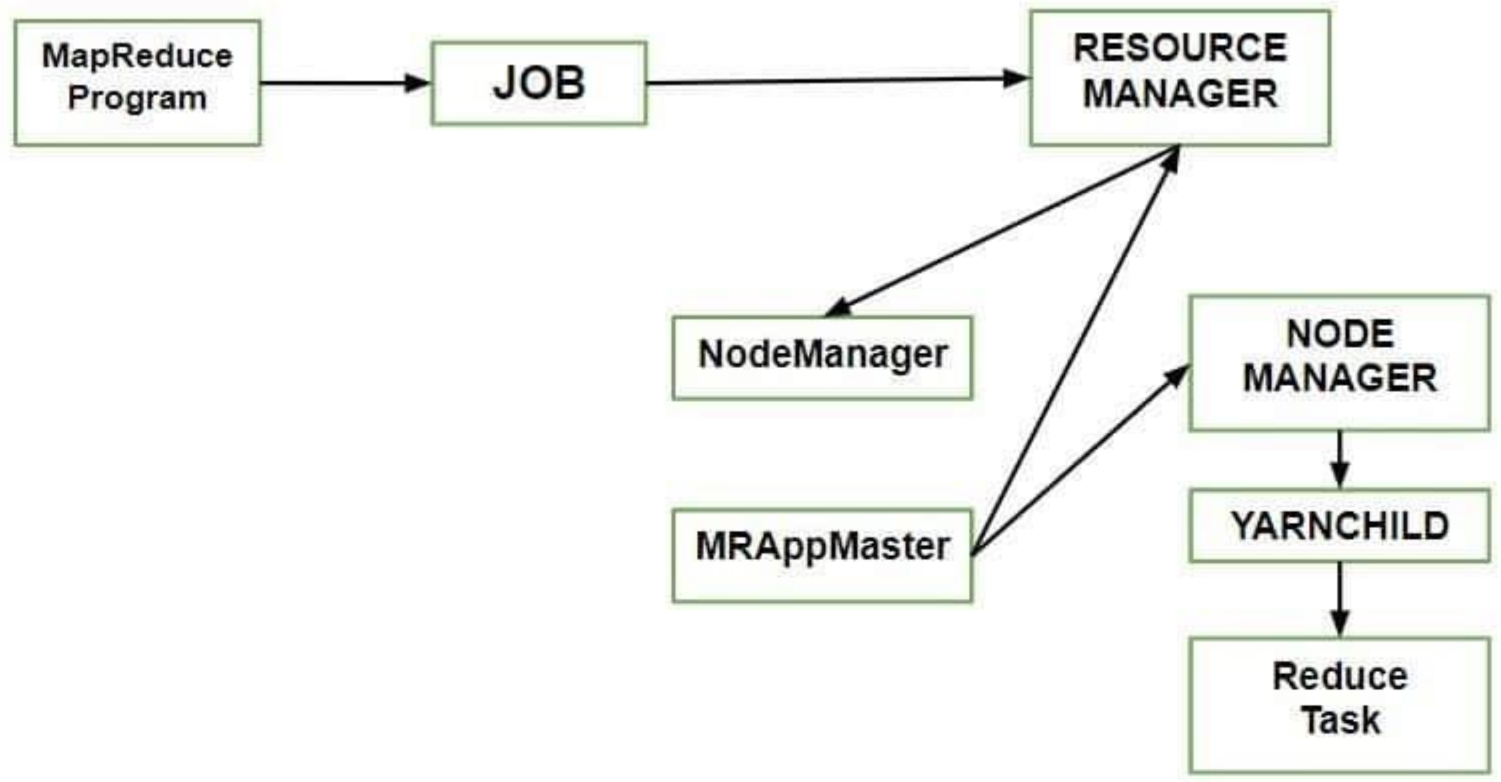
Let’s understand the components –

- Client: Submitting the MapReduce job.
- Yarn node manager: In a cluster, it monitors and launches the compute containers on machines.
- Yarn resource manager: Handles the allocation of computing resources coordination on the cluster.
- MapReduce application master Facilitates the tasks running the MapReduce work.
- Distributed Filesystem: Shares job files with other entities.

Anatomy of a MapReduce job run, Failures, Shuffle and sort, Task execution

To create an internal JobSubmitter instance, use the submit() which further calls submitJobInternal() on it. Having submitted the job, waitForCompletion() polls the job's progress after submitting the job once per second. If the reports have changed since the last report, it further reports the progress to the console. The job counters are displayed when the job completes successfully. Else the error (that caused the job to fail) is logged to the console.

Anatomy of a MapReduce job run, Failures, Shuffle and sort, Task execution



Anatomy of a MapReduce job run, Failures, Shuffle and sort, Task execution

Processes implemented by JobSubmitter for submitting the Job :

- The resource manager asks for a new application ID that is used for MapReduce Job ID.
- Output specification of the job is checked. For e.g. an error is thrown to the MapReduce program or the job is not submitted or the output directory already exists or it has not been specified.
- If the splits cannot be computed, it computes the input splits for the job. This can be due to the job is not submitted and an error is thrown to the MapReduce anatomy program.
- Resources needed to run the job are copied – it includes the job JAR file, and the computed input splits, to the shared filesystem in a directory named after the job ID and the configuration file.
- It copies job JAR with a high replication factor, which is controlled by mapreduce.client.submit.file.replication property. AS there are a number of copies across the cluster for the node managers to access.
- By calling submitApplication(), submits the job to the resource manager.

Description of Map Reduce

A Hadoop MapReduce cluster employs a masterslave architecture where one master node (known as JobTracker) manages a number of worker nodes (known as the TaskTrackers). Hadoop launches a Anatomy of MapReduce job by first splitting (logically) the input dataset into multiple data splits. Each map task is then scheduled to one TaskTracker node where the data split resides. A Task Scheduler is responsible for scheduling the execution of the tasks as far as possible in a data-local manner. A few different types of schedulers have been already developed for the MapReduce environment.

Description of Map Reduce

From a bird's eye view, a anatomy of MapReduce job is not all that complex because Hadoop hides most of the complexity of writing parallel programs for a cluster of computers. In a Hadoop cluster, every node normally starts multiple map tasks (many times depending on the number of cores a machine has) and each task will read a portion of the input data in a sequence, process every row of the data and output a <key, value> pair. The reducer tasks in turn collect the keys and values outputted by the mapper tasks and merge the identical keys into one key and the different map values into a collection of values. An individual reducer then will work on these merged keys and the reducer task outputs data of its choosing by inspecting its input of keys and associated collection of values. The programmer needs to supply only the logic and code for the map() and the reduce() functions. This simple paradigm can solve surprisingly large types of computational problems and is a keystone of the anatomy of mapreduce Job in big data processing revolution.

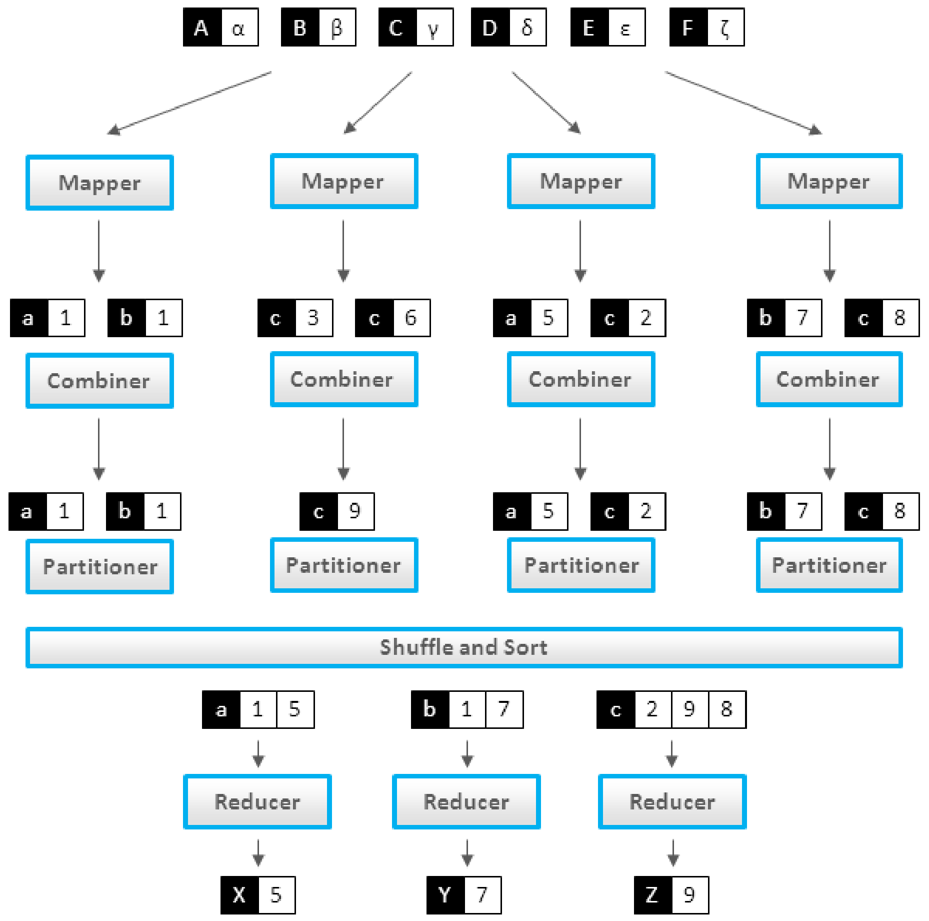
Description of Map Reduce

- In a typical Anatomy of MapReduce job, input files are read from the Hadoop Distributed File System (HDFS). Data is usually compressed to reduce file sizes. After decompression, serialized bytes are transformed into Java objects before being passed to a user-defined map() function. Conversely, output records are serialized, compressed, and eventually pushed back to HDFS. However, behind this apparent simplicity, the processing is broken down into many steps and has hundreds of different tunable parameters to fine-tune the job's running characteristics. We detail these steps starting from when a job is started all the way up to when all the map and reduce tasks are complete and the JobTracker (JT) cleans up the job.
- Hadoop MapReduce jobs are divided into a set of map tasks and reduce tasks that run in a distributed fashion on a cluster of computers. Each task work on a small subset of the data it has been assigned so that the load is spread across the cluster.
- The input to a MapReduce job is a set of files in the data store that are spread out over the HDFS. In Hadoop, these files are split with an input format, which defines how to separate a files into input split. You can assume that input split is a byte-oriented view of a chunk of the files to be loaded by a map task.

Description of Map Reduce

- The map task generally performs loading, parsing, transformation and filtering operations, whereas reduce task is responsible for grouping and aggregating the data produced by map tasks to generate final output. This is the way a wide range of problems can be solved with such a straightforward paradigm, from simple numerical aggregation to complex join operations and cartesian products.
- Each map task in Hadoop is broken into following phases: record reader, mapper, combiner, Hadoop partitioner. The output of map phase, called intermediate key and values are sent to the reducers. The reduce tasks are broken into following phases: shuffle, sort, reducer and output format. The map tasks are assigned by Hadoop framework to those DataNodes where the actual data to be processed resides. This ensures that the data typically doesn’t have to move over the network to save the network bandwidth and data is computed on the local machine itself so called map task is data local.

Description of Map Reduce



Description of Map Reduce

- **Mapper**
  - **Record Reader:**

    The record reader translates an input split generated by input format into records. The purpose of record reader is to parse the data into record but doesn't parse the record itself. It passes the data to the mapper in form of key/value pair. Usually the key in this context is positional information and the value is a chunk of data that composes a record. In our future articles we will discuss more about NLineInputFormat and custom record readers.
  - **Map:**

    Map function is the heart of mapper task, which is executed on each key/value pair from the record reader to produce zero or more key/value pair, called intermediate pairs. The decision of what is key/value pair depends on what the MapReduce job is accomplishing. The data is grouped on key and the value is the information pertinent to the analysis in the reducer.

Description of Map Reduce

Mapper:

Combiner:

Its an optional component but highly useful and provides extreme performance gain of MapReduce job without any downside. Combiner is not applicable to all the MapReduce algorithms but where ever it can be applied it is always recommended to use. It takes the intermediate keys from the mapper and applies a user-provided method to aggregate values in a small scope of that one mapper. e.g sending (hadoop, 3) requires fewer bytes than sending (hadoop, 1) three times over the network. We will cover combiner in much more depth in our future articles.

Partitioner:

The Hadoop partitioner takes the intermediate key/value pairs from mapper and split them into shards, one shard per reducer. This randomly distributes the keyspace evenly over the reducer, but still ensures that keys with the same value in different mappers end up at the same reducer. The partitioned data is written to the local filesystem for each map task and waits to be pulled by its respective reducer.

Description of Map Reduce

- **Reducer:**
- **Shuffle and Sort:**

The reduce task start with the shuffle and sort step. This step takes the output files written by all of the hadoop partitioners and downloads them to the local machine in which the reducer is running. These individual data pipes are then sorted by keys into one larger data list. The purpose of this sort is to group equivalent keys together so that their values can be iterated over easily in the reduce task.

- **Reduce:**

The reducer takes the grouped data as input and runs a reduce function once per key grouping. The function is passed the key and an iterator over all the values associated with that key. A wide range of processing can happen in this function, the data can be aggregated, filtered, and combined in a number of ways. Once it is done, it sends zero or more key/value pair to the final step, the output format.

Map reduce types and formats.

- **FileInputFormat**

It serves as the foundation for all file-based InputFormats. FileInputFormat also provides the input directory, which contains the location of the data files. When we start a MapReduce task, FileInputFormat returns a path with files to read. This InputFormat will read all files. Then it divides these files into one or more InputSplits.

Map reduce types and formats.

- **TextInputFormat**
- It is the standard InputFormat. Each line of each input file is treated as a separate record by this InputFormat. It does not parse anything. TextInputFormat is suitable for raw data or line-based records, such as log files. Hence:
- Key: It is the byte offset of the first line within the file (not the entire file split). As a result, when paired with the file name, it will be unique.
- Value: It is the line's substance. It does not include line terminators.

Map reduce types and formats.

- **KeyValueTextInputFormat**
- It is comparable to TextInputFormat. Each line of input is also treated as a separate record by this InputFormat. While TextInputFormat treats the entire line as the value, KeyValueTextInputFormat divides the line into key and value by a tab character ('/t'). Hence:
- Key: Everything up to and including the tab character.
- Value: It is the remaining part of the line after the tab character.

Map reduce types and formats.

- **SequenceFileInputFormat**
- It's an input format for reading sequence files. Binary files are sequence files. These files also store binary key-value pair sequences. These are block-compressed and support direct serialization and deserialization of a variety of data types. Hence Key & Value are both user-defined.
- **SequenceFileAsTextInputFormat**
- It is a subtype of SequenceFileInputFormat. The sequence file key values are converted to Text objects using this format. As a result, it converts the keys and values by running 'tostring()' on them. As a result, SequenceFileAsTextInputFormat converts sequence files into text-based input for streaming.

Map reduce types and formats.

- **NlineInputFormat**
- It is a variant of TextInputFormat in which the keys are the line's byte offset. And values are the line's contents. As a result, each mapper receives a configurable number of lines of TextInputFormat and KeyValueTextInputFormat input. The number is determined by the magnitude of the split. It is also dependent on the length of the lines. So, if we want our mapper to accept a specific amount of lines of input, we use NLineInputFormat.
- N- It is the number of lines of input received by each mapper.
- Each mapper receives exactly one line of input by default (N=1).
- Assuming N=2, each split has two lines. As a result, the first two Key-Value pairs are distributed to one mapper. The second two key-value pairs are given to another mapper.

Map reduce types and formats.

- **DBInputFormat**
- Using JDBC, this InputFormat reads data from a relational Database. It also loads small datasets, which might be used to connect with huge datasets from HDFS using multiple inputs. Hence:
- Key: LongWritables
- Value: DBWritables.

Map reduce types and formats.

- **Output Format in MapReduce:**
- The output format classes work in the opposite direction as their corresponding input format classes. The TextOutputFormat, for example, is the default output format that outputs records as plain text files, although key values can be of any type and are converted to strings by using the toString() method. The tab character separates the key-value character, but this can be changed by modifying the separator attribute of the text output format.
- SequenceFileOutputFormat is used to write a sequence of binary output to a file for binary output. Binary outputs are especially valuable if they are used as input to another MapReduce process.
- DBOutputFormat handles the output formats for relational databases and HBase. It saves the compressed output to a SQL table.