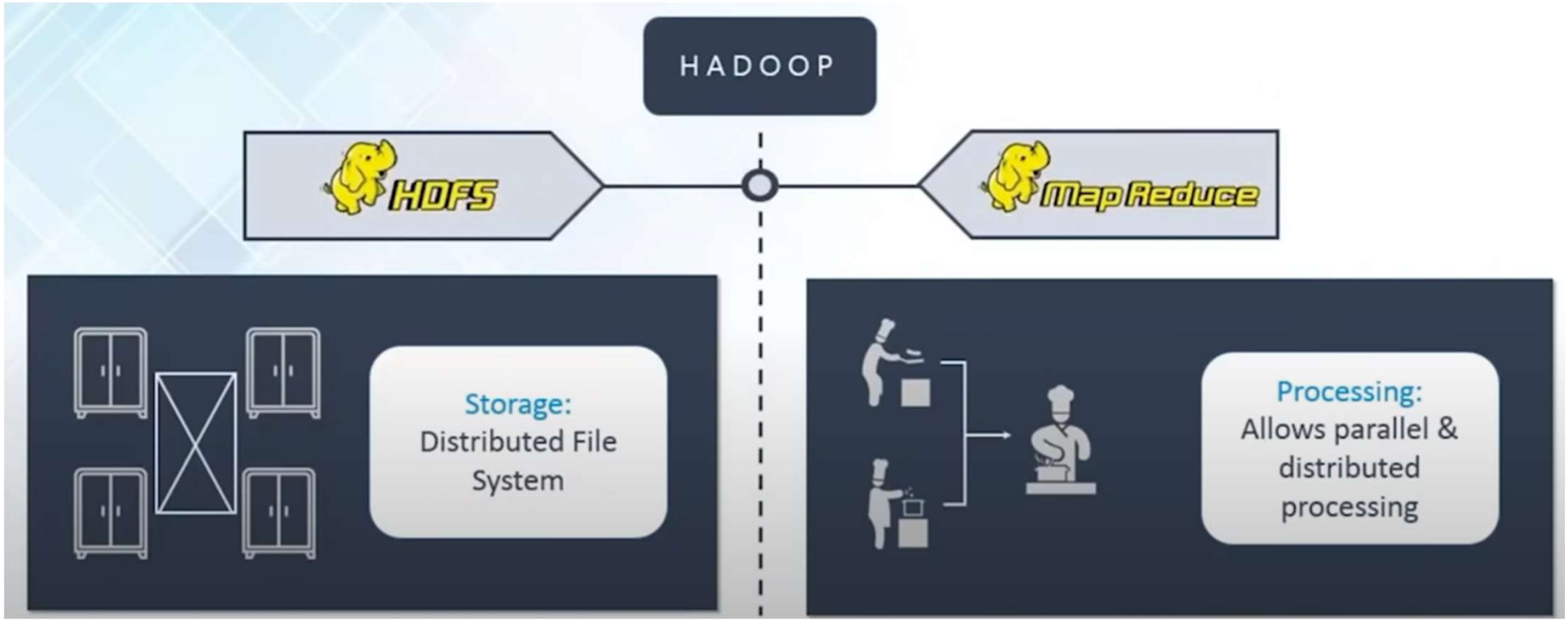

ETL

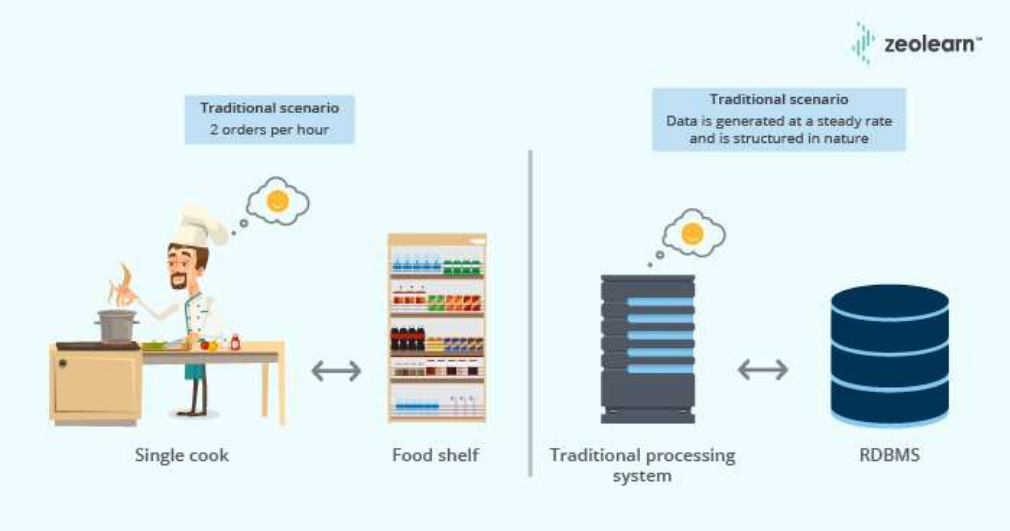
ETL

- ETL, which stands for extract, transform, and load, is the process data engineers use to extract data from different sources, transform the data into a usable and trusted resource, and load that data into the systems end-users can access and use downstream to solve business problems.

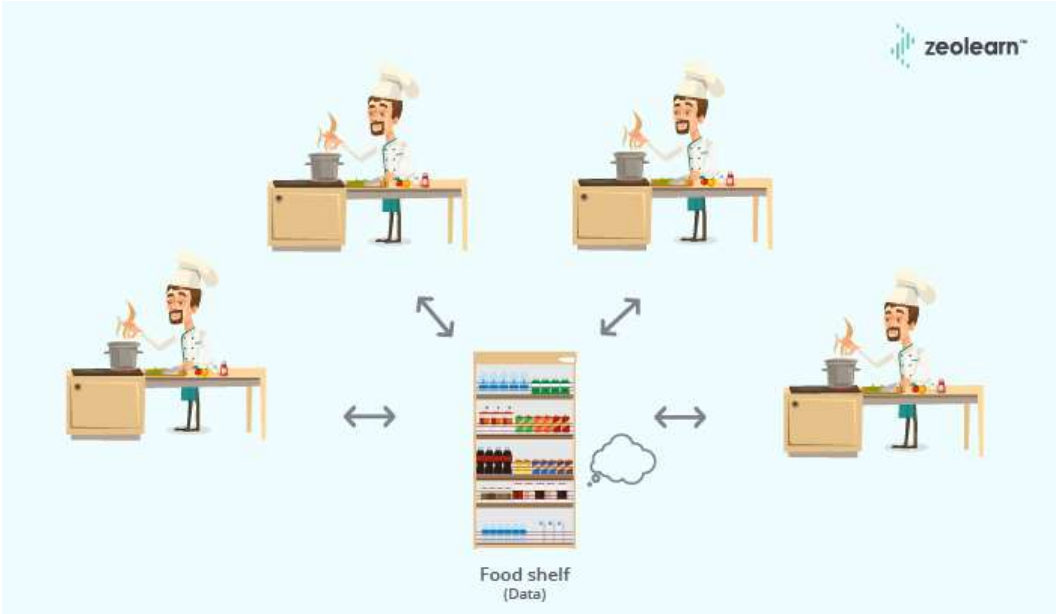
Framework

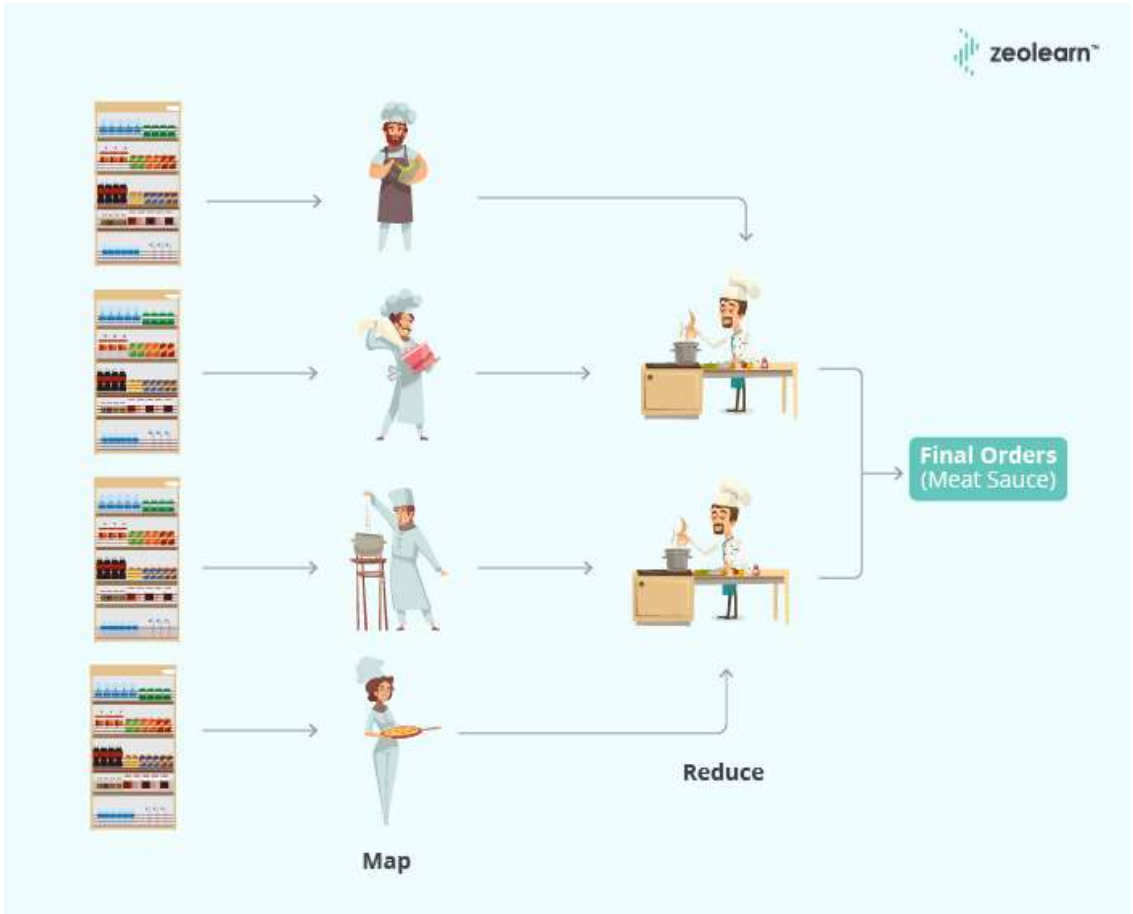

HADOOP
HDFS
MapReduce
Storage:
Distributed File
System
Processing:
Allows parallel &
distributed
processing

Analogy

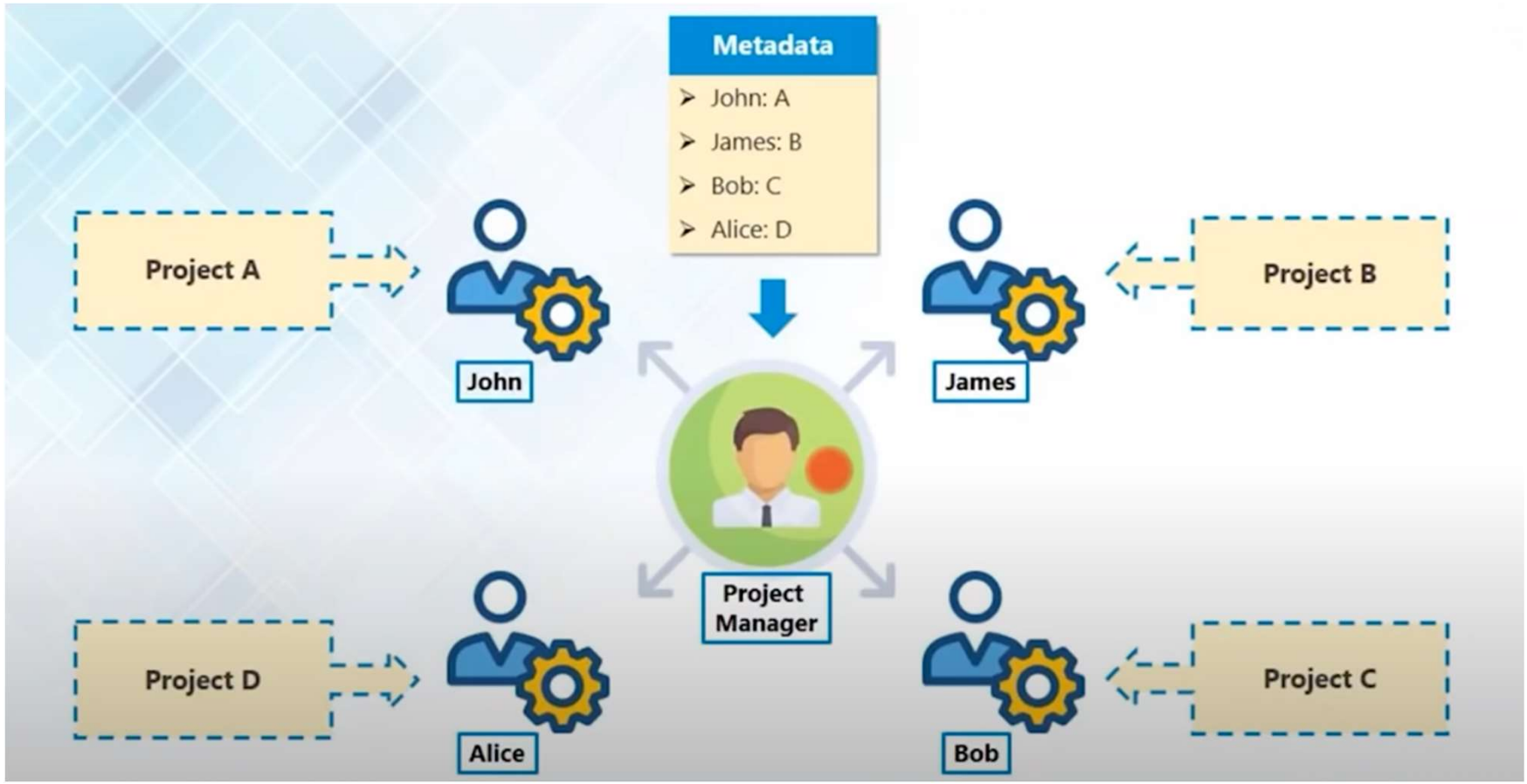

zeolearn
Traditional scenario
2 orders per hour
Traditional scenario
Data is generated at a steady rate and is structured in nature
Single cook
Food shelf
Traditional processing system
RDBMS

Analogy

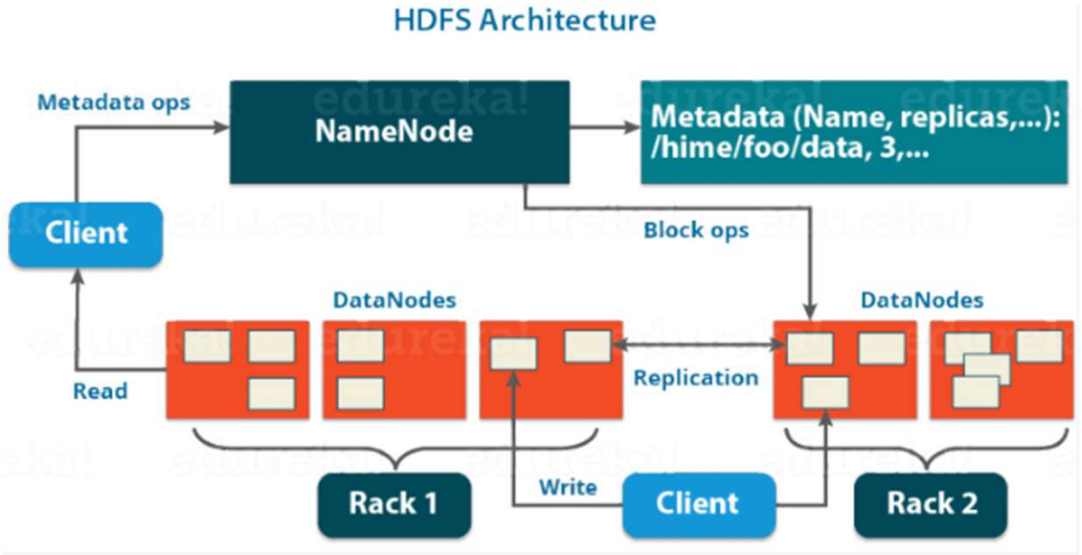



zeolearn
Final Orders
(Meat Sauce)
Reduce
Map

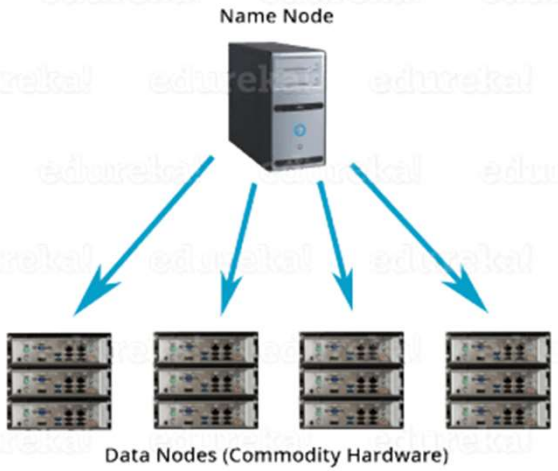
Master Slave Architecture



HDFS Architecture



Name Node



Name Node

- NameNode is the master node in the Apache Hadoop HDFS Architecture that maintains and manages the blocks present on the DataNodes (slave nodes).
- NameNode is a very highly available server that manages the File System Namespace and controls access to files by clients.
- The HDFS architecture is built in such a way that the user data never resides on the NameNode. The data resides on DataNodes only.

Name Node

- It is the master daemon that maintains and manages the DataNodes (slave nodes)
- It records the metadata of all the files stored in the cluster, e.g. The location of blocks stored, the size of the files, permissions, hierarchy, etc. There are two files associated with the metadata:
  - **FsImage:** It contains the complete state of the file system namespace since the start of the NameNode.
  - **EditLogs:** It contains all the recent modifications made to the file system with respect to the most recent FsImage.

Name Node

- It records each change that takes place to the file system metadata. For example, if a file is deleted in HDFS, the NameNode will immediately record this in the EditLog.
- It regularly receives a Heartbeat and a block report from all the DataNodes in the cluster to ensure that the DataNodes are live.
- It keeps a record of all the blocks in HDFS and in which these blocks are located.
- The NameNode is also responsible to take care of the **replication factor** of all the blocks which we will discuss in detail later in this HDFS tutorial blog.
- In **case of the DataNode failure**, the NameNode chooses new DataNodes for new replicas, balance disk usage and manages the communication traffic to the DataNodes.

Data Node

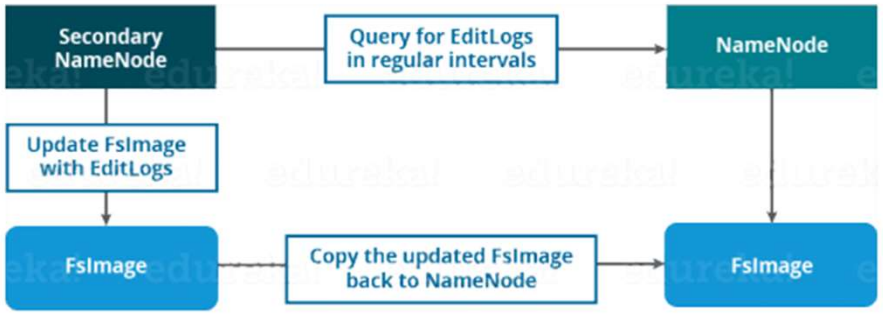
- DataNodes are the slave nodes in HDFS. Unlike NameNode, DataNode is a commodity hardware, that is, a non-expensive system which is not of high quality or high-availability. The DataNode is a block server that stores the data in the local file ext3 or ext4.

Data Node

- These are slave daemons or process which runs on each slave machine.
- The actual data is stored on DataNodes.
- The DataNodes perform the low-level read and write requests from the file system’s clients.
- They send heartbeats to the NameNode periodically to report the overall health of HDFS, by default, this frequency is set to 3 seconds.

Secondary Name Node

- Apart from these two daemons, there is a third daemon or a process called Secondary NameNode. The Secondary NameNode works concurrently with the primary NameNode as a **helper daemon.**



Secondary Name Node

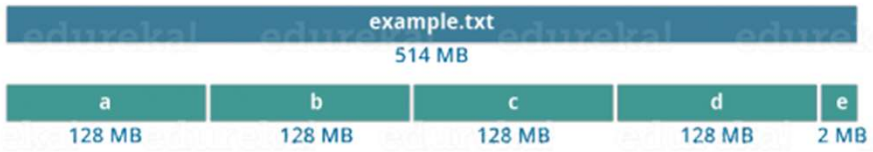
- The Secondary NameNode is one which constantly reads all the file systems and metadata from the RAM of the NameNode and writes it into the hard disk or the file system.
- It is responsible for combining the EditLogs with FsImage from the NameNode.
- It downloads the EditLogs from the NameNode at regular intervals and applies to FsImage. The new FsImage is copied back to the NameNode, which is used whenever the NameNode is started the next time.

Data Blocks

- Blocks are the nothing but the smallest continuous location on your hard drive where data is stored. In general, in any of the File System, you store the data as a collection of blocks.
- Similarly, HDFS stores each file as blocks which are scattered throughout the Apache Hadoop cluster.
- The default size of each block is 128 MB in Apache Hadoop 2.x (64 MB in Apache Hadoop 1.x) which you can configure as per your requirement.

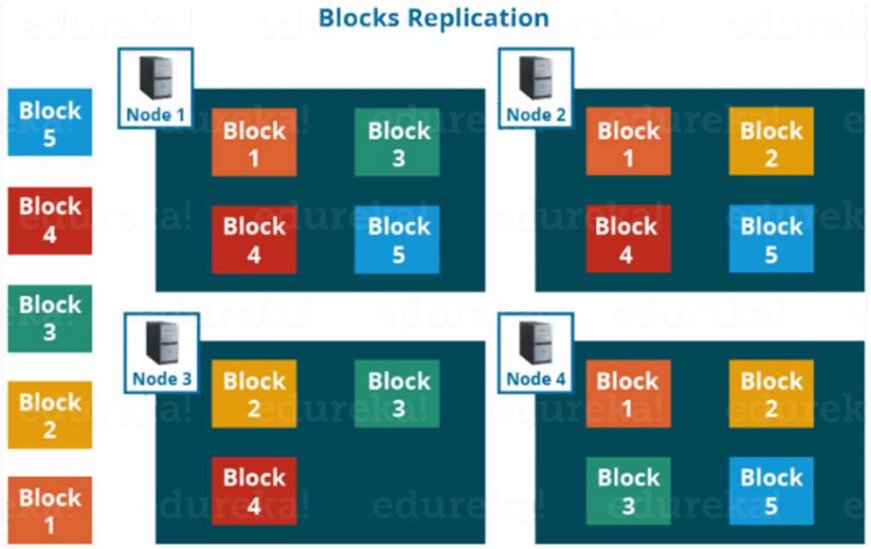
Data Blocks

- It is not necessary that in HDFS, each file is stored in exact multiple of the configured block size (128 MB, 256 MB etc.).
- Let's take an example where I have a file "example.txt" of size 514 MB as shown in above figure.
- Suppose that we are using the default configuration of block size, which is 128 MB. Then, how many blocks will be created?
- The first four blocks will be of 128 MB. But, the last block will be of 2 MB size only.



Replication Factor

- HDFS provides a reliable way to store huge data in a distributed environment as data blocks. The blocks are also replicated to provide fault tolerance. The default replication factor is 3 which is again configurable.
- So, as you can see in the figure where each block is replicated three times and stored on different DataNodes (considering the default replication factor):



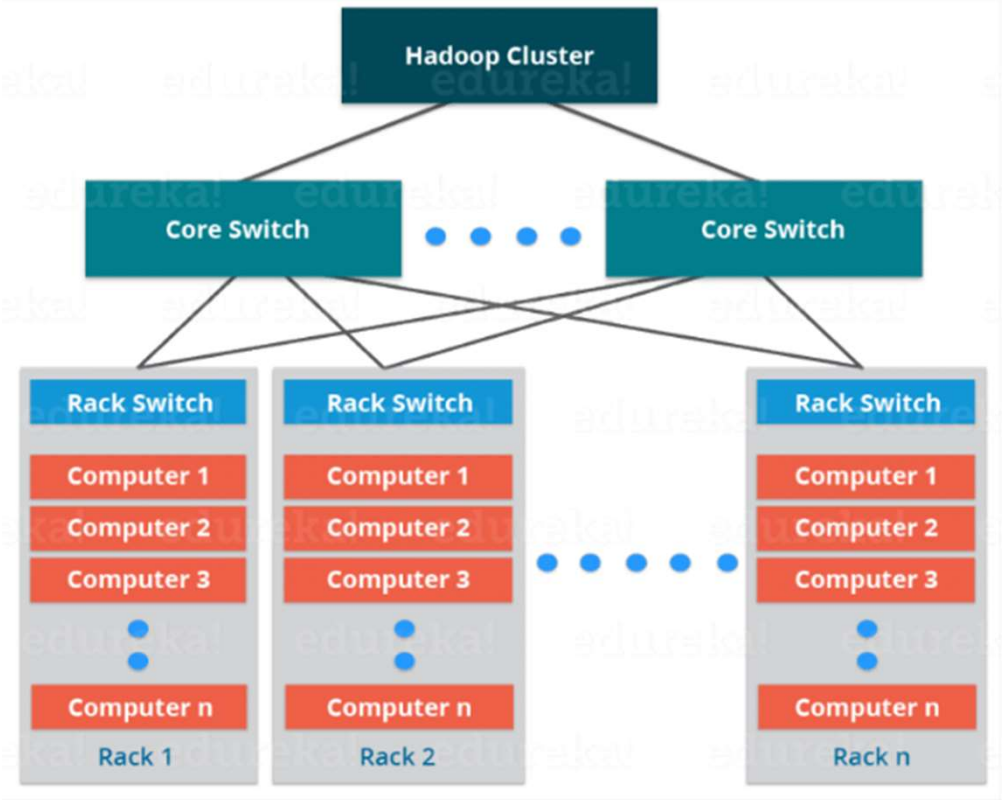
Replication Factor

- Therefore, if you are storing a file of 128 MB in HDFS using the default configuration, you will end up occupying a space of 384 MB (3*128 MB) as the blocks will be replicated three times and each replica will be residing on a different DataNo

Rack Awareness

- NameNode also ensures that all the replicas are not stored on the same rack or a single rack. It follows an in-built Rack Awareness Algorithm to reduce latency as well as provide fault tolerance.
- Considering the replication factor is 3, the Rack Awareness Algorithm says that the first replica of a block will be stored on a local rack and the next two replicas will be stored on a different (remote) rack but, on a different DataNode within that (remote) rack

Rack Awareness

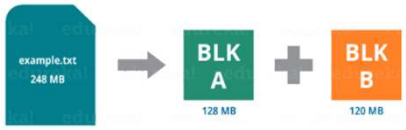


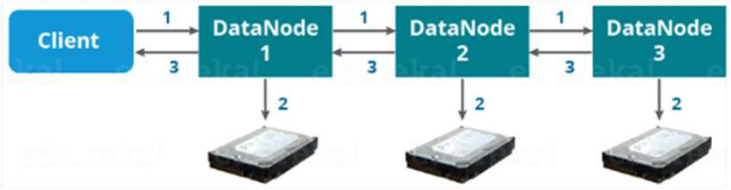
Rack Awareness

- **To improve the network performance:** The communication between nodes residing on different racks is directed via switch. In general, you will find *greater network bandwidth* between machines in the same rack than the machines residing in different rack. So, the Rack Awareness helps you to have reduce write traffic in between different racks and thus providing a better write performance. Also, you will be gaining increased read performance because you are using the bandwidth of multiple racks.
- **To prevent loss of data:** We don't have to worry about the data even if an entire rack fails because of the switch failure or power failure. And if you think about it, it will make sense, as it is said that *never put all your eggs in the same basket.*

HDFS Write Architecture

- At first, the HDFS client will reach out to the NameNode for a Write Request against the two blocks, say, Block A & Block B.
- The NameNode will then grant the client the write permission and will provide the IP addresses of the DataNodes where the file blocks will be copied eventually.
- The selection of IP addresses of DataNodes is purely randomized based on availability, replication factor and rack awareness that we have discussed earlier.
- Let's say the replication factor is set to default i.e. 3. Therefore, for each block the NameNode will be providing the client a list of (3) IP addresses of DataNodes. The list will be unique for each block.
- Suppose, the NameNode provided following lists of IP addresses to the client:
  - For Block A, list A = {IP of DataNode 1, IP of DataNode 4, IP of DataNode 6}
  - For Block B, set B = {IP of DataNode 3, IP of DataNode 7, IP of DataNode 9}
- Each block will be copied in three different DataNodes to maintain the replication factor consistent throughout the cluster.



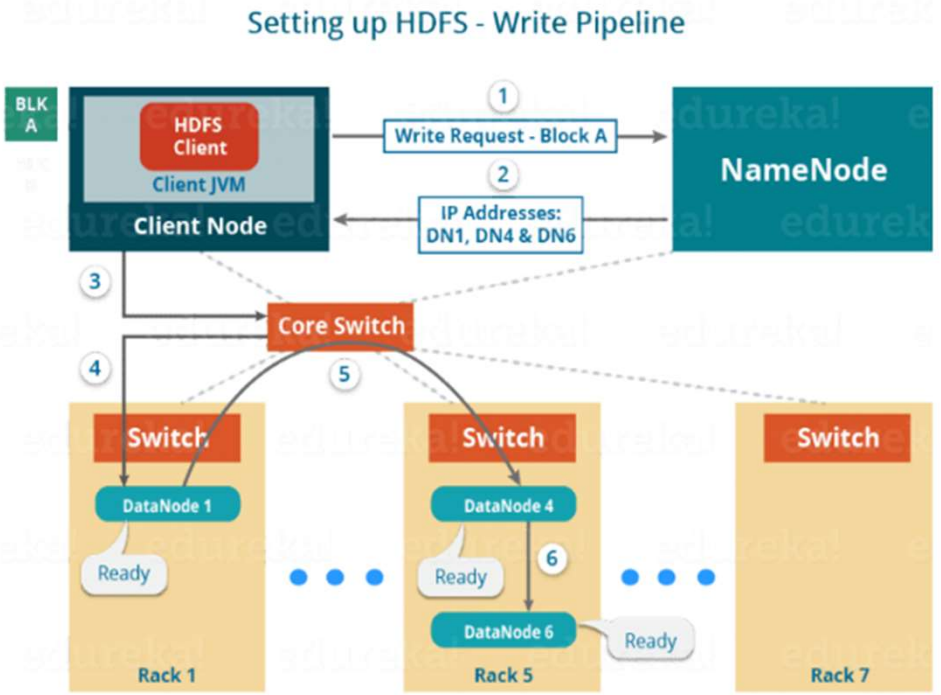


HDFS Write

- The whole process happens in 3 stages:
  1. Set up of Pipeline
  2. Data streaming and replication
  3. Shutdown of Pipeline (Acknowledgement stage)

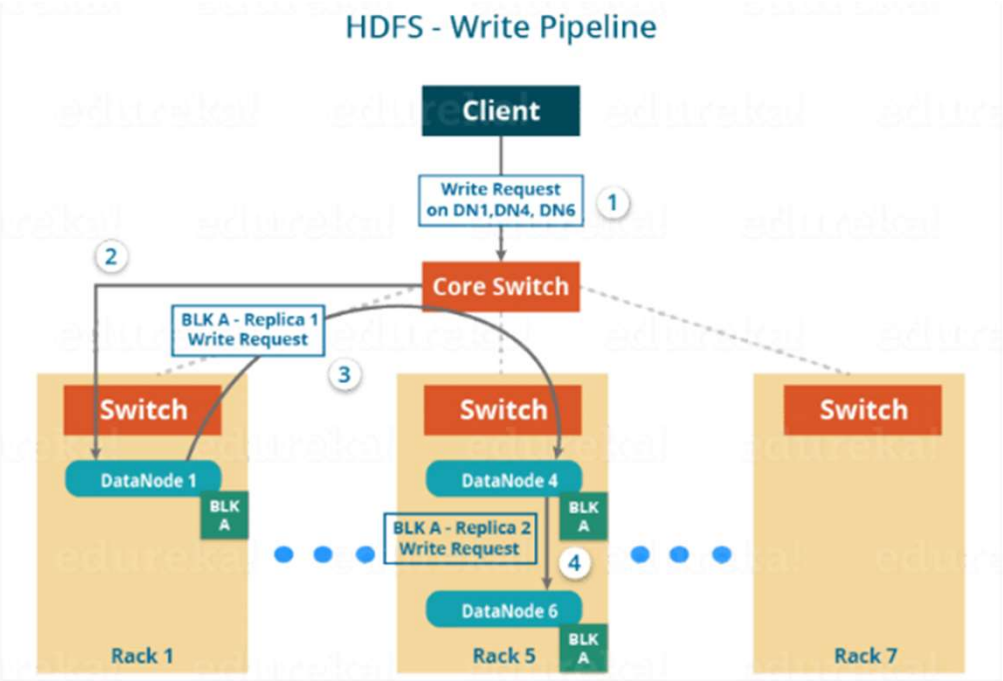
Set up of pipeline

- The client will choose the first DataNode in the list (DataNode IPs for Block A) which is DataNode 1 and will establish a TCP/IP connection.
- The client will inform DataNode 1 to be ready to receive the block. It will also provide the IPs of next two DataNodes (4 and 6) to the DataNode 1 where the block is supposed to be replicated.
- The DataNode 1 will connect to DataNode 4. The DataNode 1 will inform DataNode 4 to be ready to receive the block and will give it the IP of DataNode 6. Then, DataNode 4 will tell DataNode 6 to be ready for receiving the data.
- Next, the acknowledgement of readiness will follow the reverse sequence, i.e. From the DataNode 6 to 4 and then to 1.
- At last DataNode 1 will inform the client that all the DataNodes are ready and a pipeline will be formed between the client, DataNode 1, 4 and 6.
- Now pipeline set up is complete and the client will finally begin the data copy or streaming process.



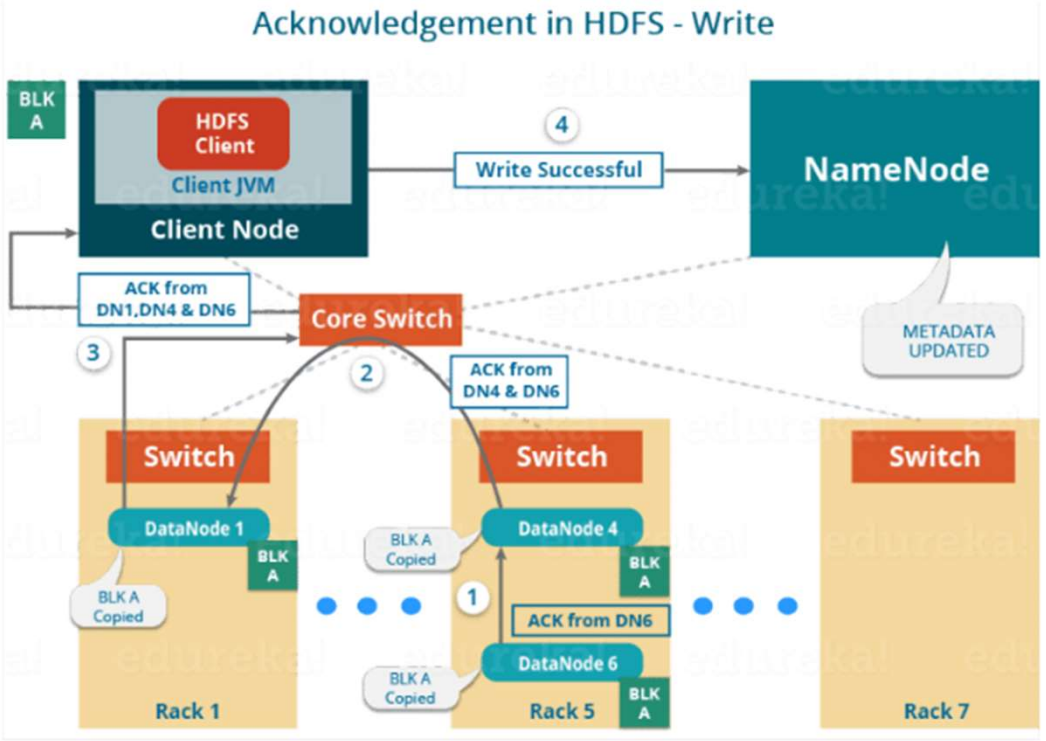
Data Streaming

- Once the block has been written to DataNode 1 by the client, DataNode 1 will connect to DataNode 4.
- Then, DataNode 1 will push the block in the pipeline and data will be copied to DataNode 4.
- Again, DataNode 4 will connect to DataNode 6 and will copy the last replica of the block.



Shutdown of Pipeline

- Once the block has been copied into all the three DataNodes, a series of acknowledgements will take place to ensure the client and NameNode that the data has been written successfully. Then, the client will finally close the pipeline to end the TCP session.
- As shown in the figure below, the acknowledgement happens in the reverse sequence i.e. from DataNode 6 to 4 and then to 1. Finally, the DataNode 1 will push three acknowledgements (including its own) into the pipeline and send it to the client. The client will inform NameNode that data has been written successfully. The NameNode will update its metadata and the client will shut down the pipeline.

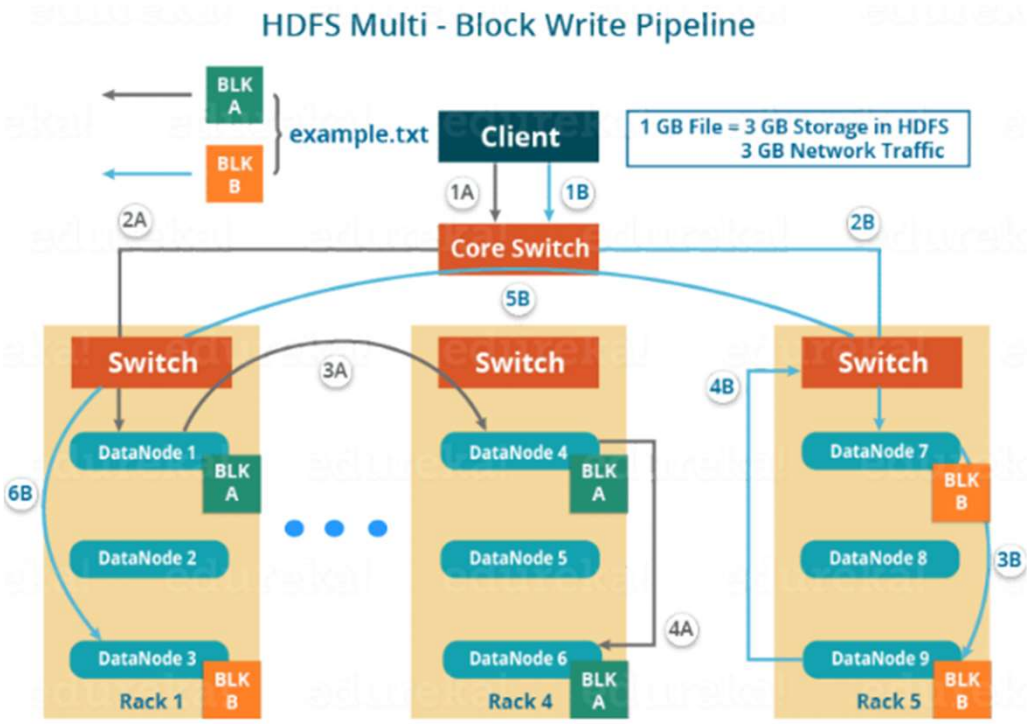


Multi Block Write

- As you can see in the above image, there are two pipelines formed for each block (A and B). Following is the flow of operations that is taking place for each block in their respective pipelines:
- For Block A:

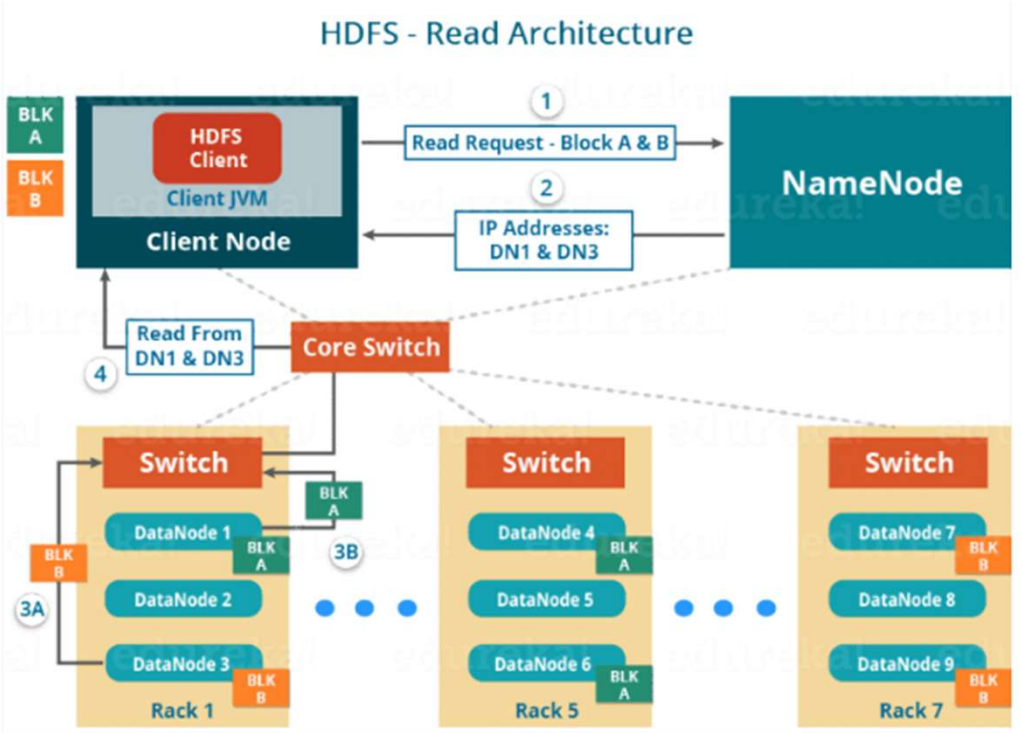
  1A -> 2A -> 3A -> 4A
- For Block B:

  1B -> 2B -> 3B -> 4B -> 5B -> 6B

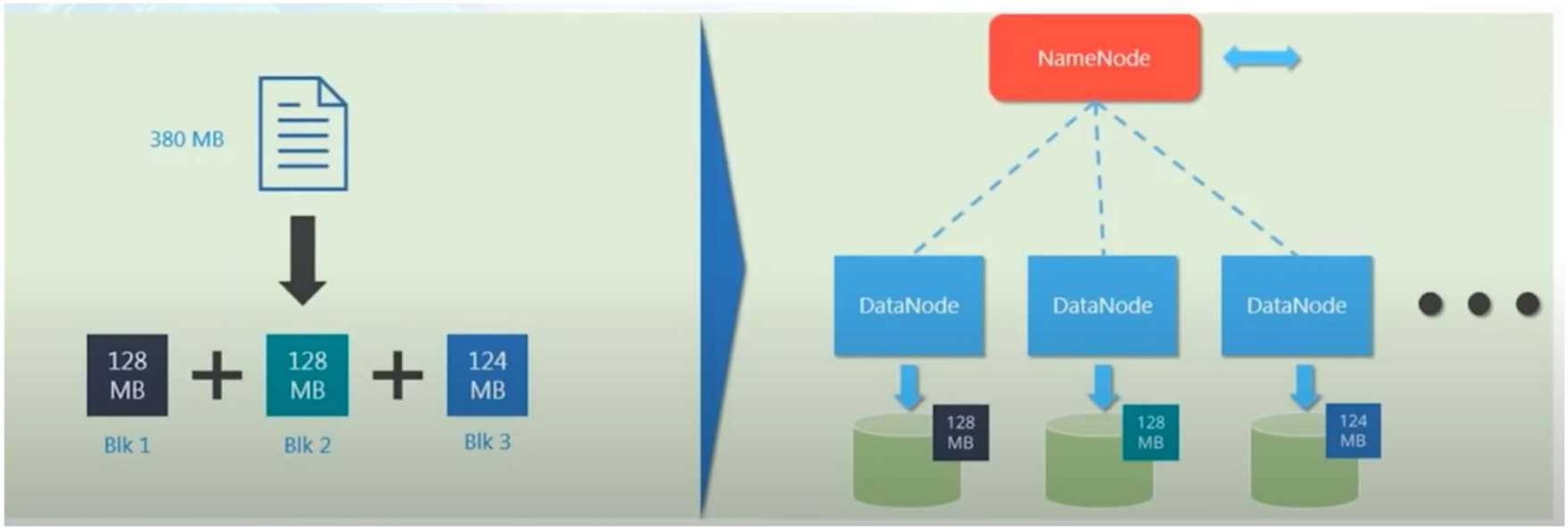


HDFS Read

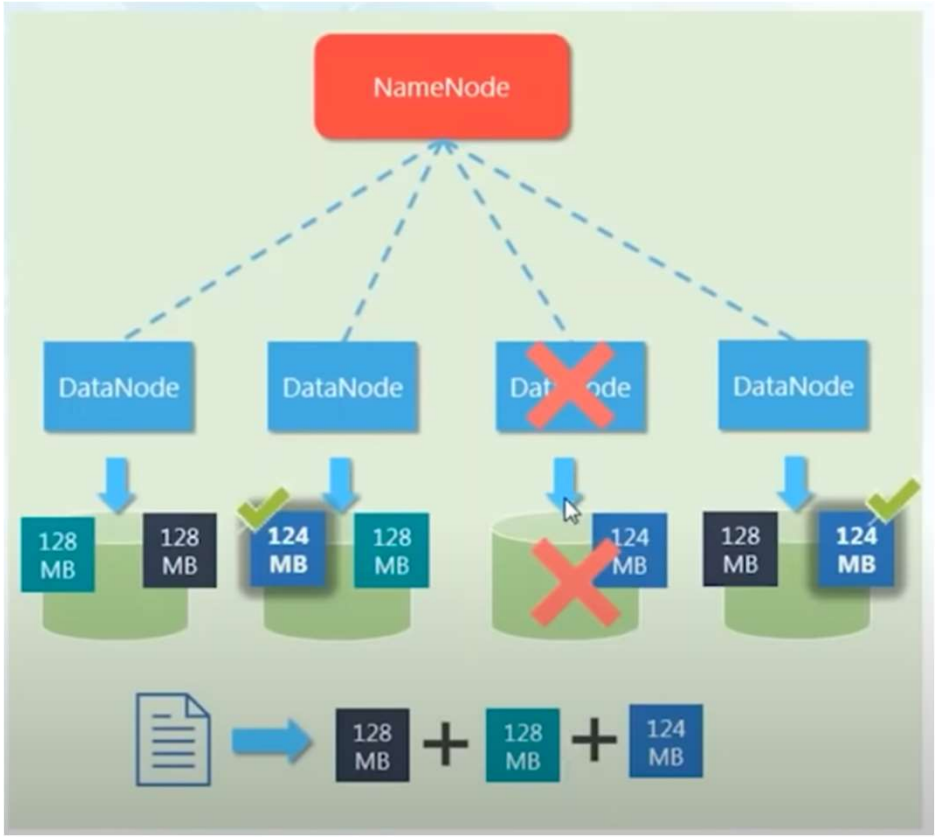
- The client will reach out to NameNode asking for the block metadata for the file “example.txt”.
- The NameNode will return the list of DataNodes where each block (Block A and B) are stored.
- After that client, will connect to the DataNodes where the blocks are stored.
- The client starts reading data parallel from the DataNodes (Block A from DataNode 1 and Block B from DataNode 3).
- Once the client gets all the required file blocks, it will combine these blocks to form a file.



HDFS Data Blocks



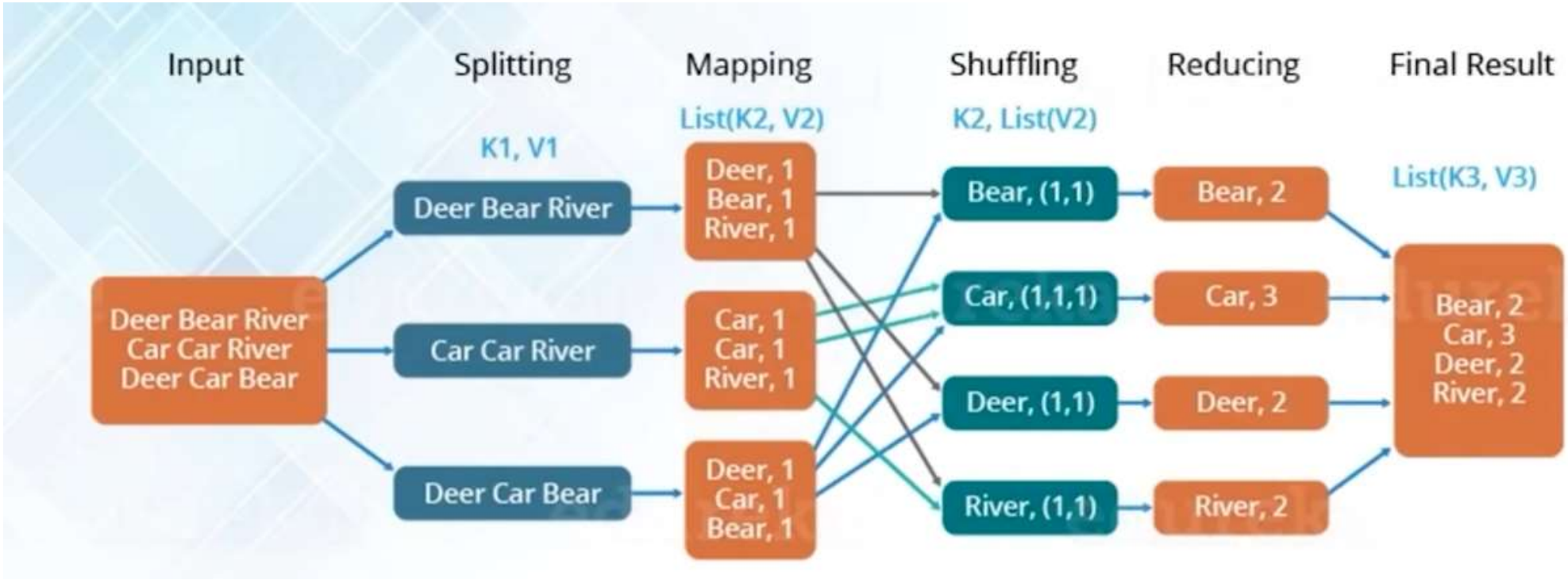
Fault Tolerance



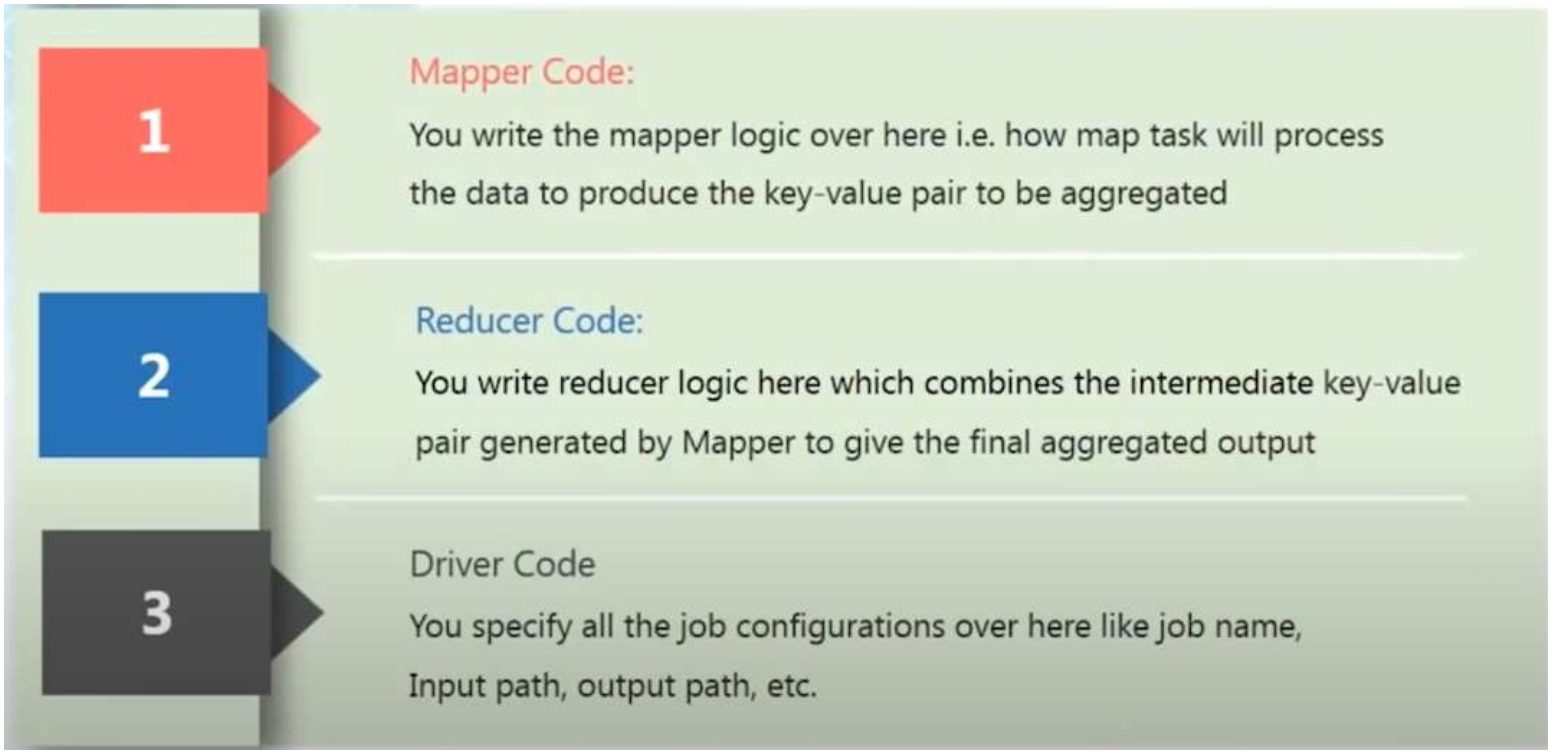
Solution:

Each data blocks are replicated (thrice by default) and are distributed across different DataNodes

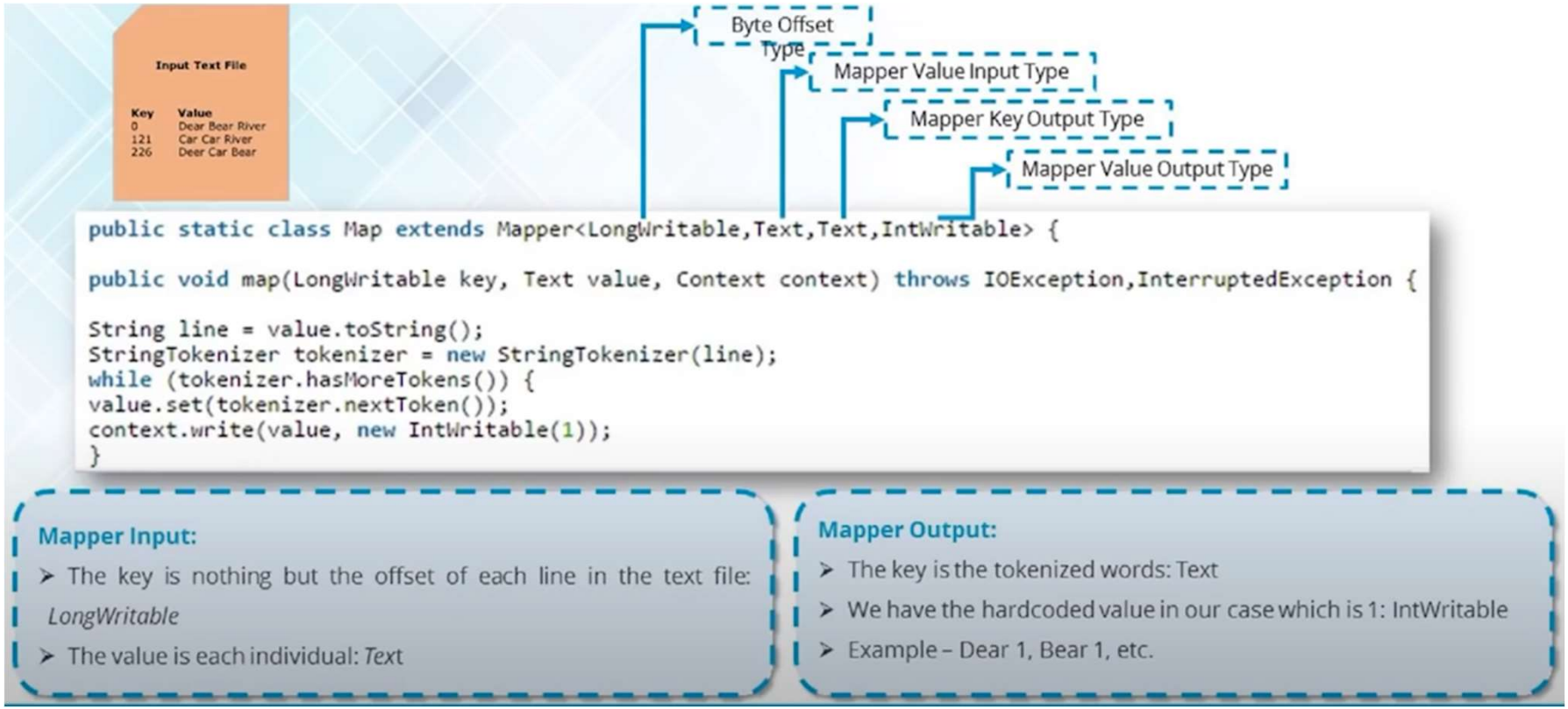
MapReduce Program



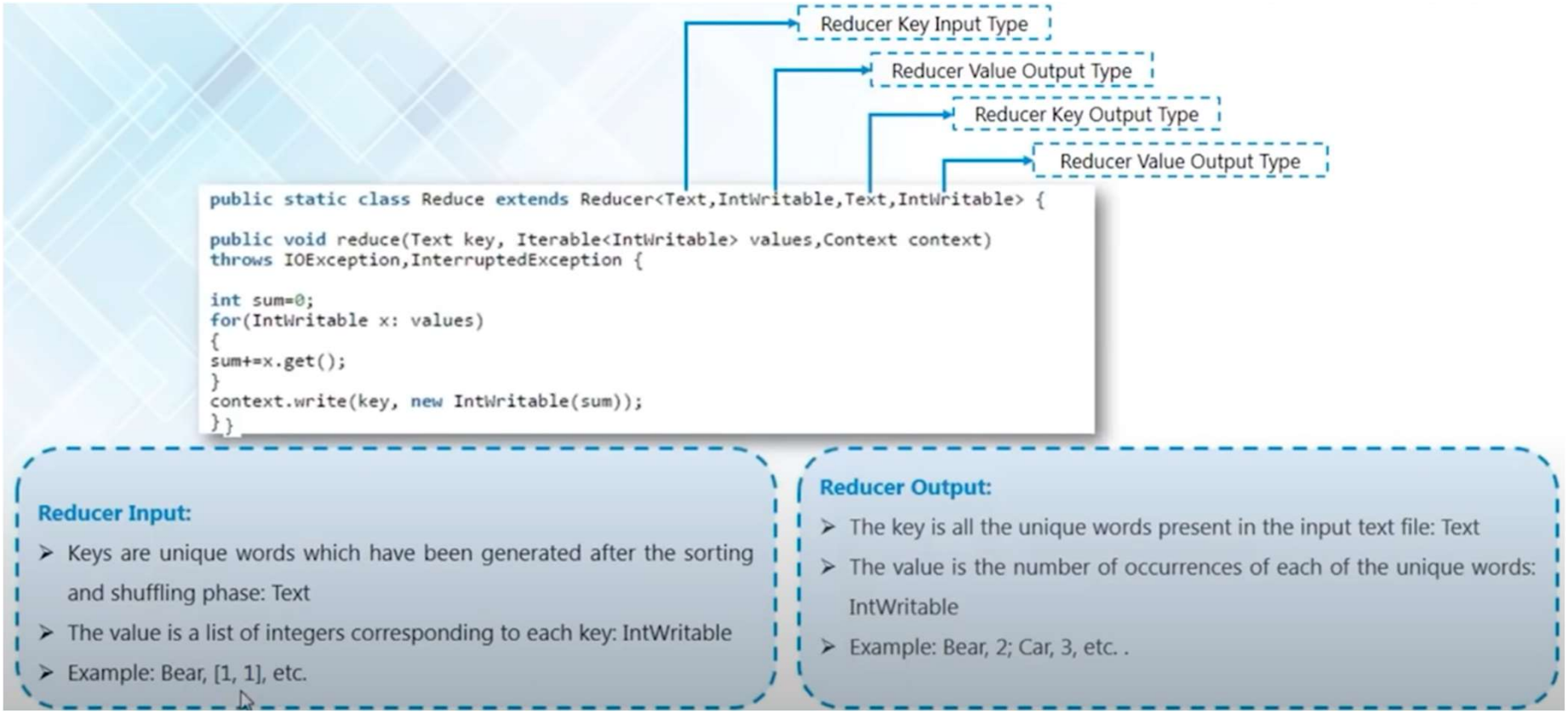
Major Parts of MapReduce Program



Mapper Code


Input Text File
Key Value
0 Dear Bear River
121 Car Car River
226 Deer Car Bear
Byte Offset Type
Mapper Value Input Type
Mapper Key Output Type
Mapper Value Output Type
public static class Map extends Mapper<LongWritable,Text,Text,IntWritable> {
public void map(LongWritable key, Text value, Context context) throws IOException,InterruptedException {
String line = value.toString();
StringTokenizer tokenizer = new StringTokenizer(line);
while (tokenizer.hasMoreTokens()) {
value.set(tokenizer.nextToken());
context.write(value, new IntWritable(1));
}
Mapper Input:
The key is nothing but the offset of each line in the text file: LongWritable
The value is each individual: Text
Mapper Output:
The key is the tokenized words: Text
We have the hardcoded value in our case which is 1: IntWritable
Example – Dear 1, Bear 1, etc.

Reducer Code



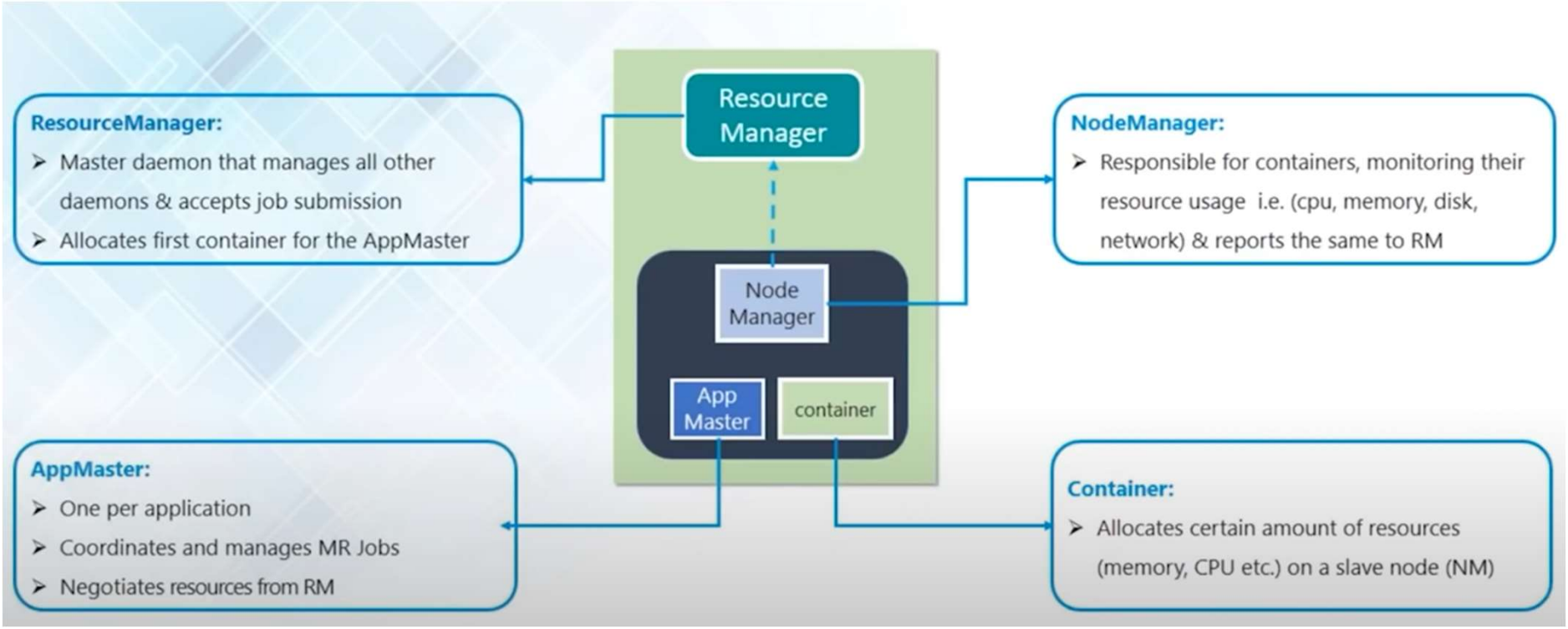
Driver Code

In the driver class, we set the configuration of our MapReduce job to run in Hadoop

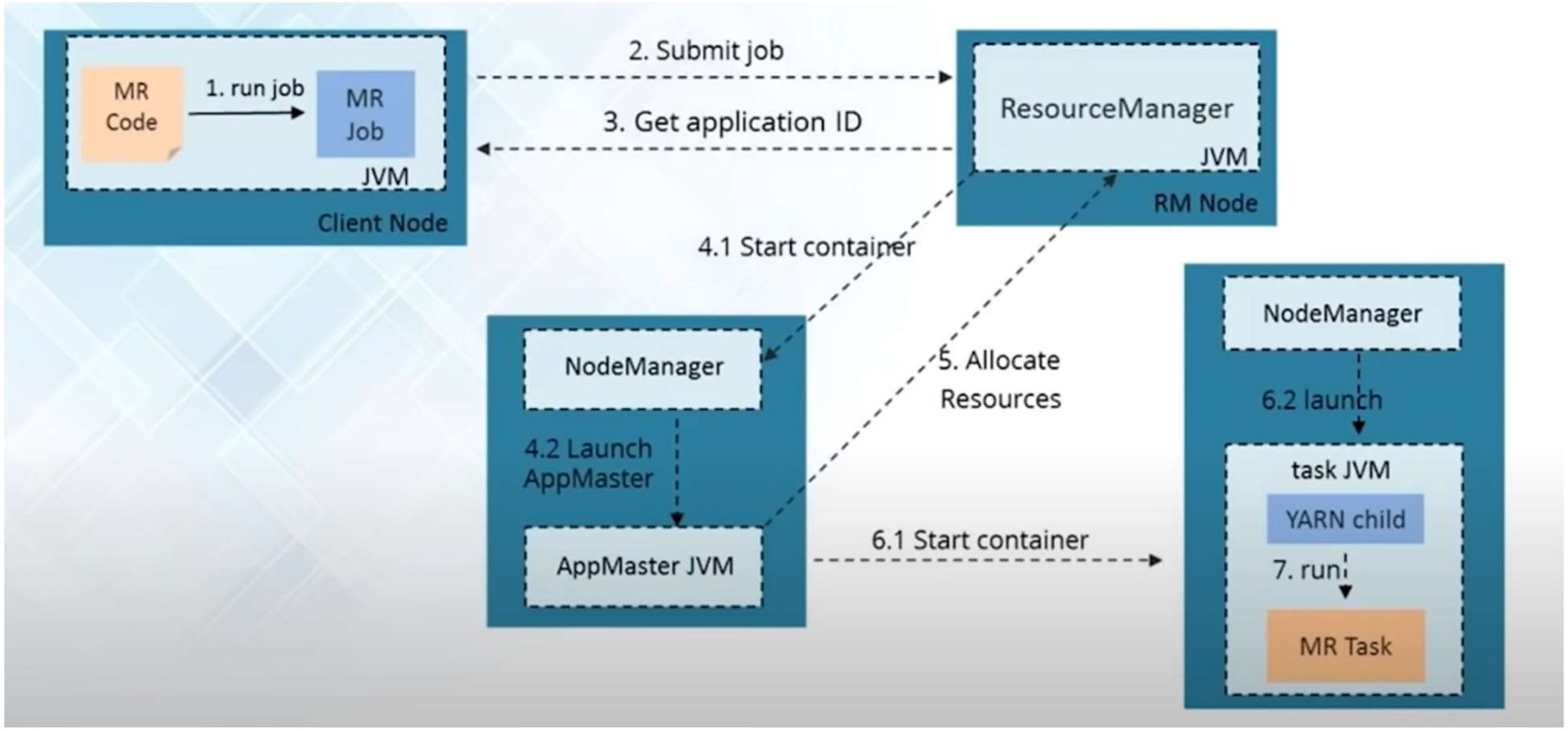
```
Configuration conf= new Configuration();
Job job = new Job(conf,"My Word Count Program");
job.setJarByClass(WordCount.class);
job.setMapperClass(Map.class);
job.setReducerClass(Reduce.class);
job.setOutputKeyClass(Text.class);

job.setOutputValueClass(IntWritable.class);
job.setInputFormatClass(TextInputFormat.class);
job.setOutputFormatClass(TextOutputFormat.class);
Path outputPath = new Path(args[1]);

//Configuring the input/output path from the filesystem into the job
FileInputFormat.addInputPath(job, new Path(args[0]));
FileOutputFormat.setOutputPath(job, new Path(args[1]));
```

- Specify the name of the job , the data type of input/output of the mapper and reducer
- Specify the names of the mapper and reducer classes.
- Path of the input and output folder
- The method setInputFormatClass () is used for specifying the unit of work for mapper
- Main() method is the entry point for the driver

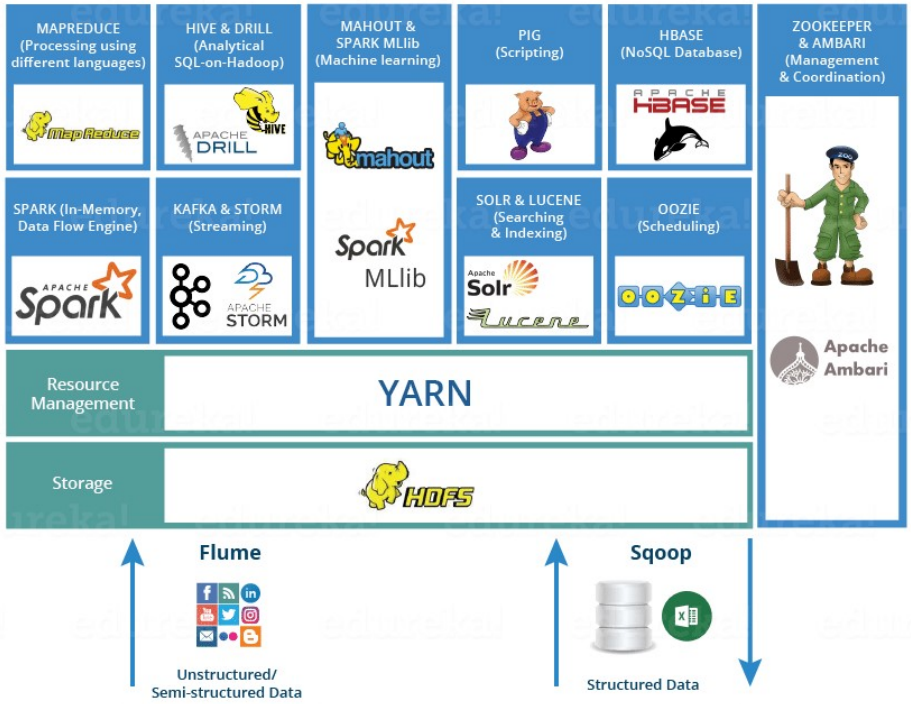
YARN Components



MapReduce Job Workflow



Hadoop Ecosystem



RDD

- RDD (Resilient Distributed Dataset) is a fundamental building block of PySpark which is fault-tolerant, immutable distributed collections of objects. Immutable meaning once you create an RDD you cannot change it.
- Each record in RDD is divided into logical partitions, which can be computed on different nodes of the cluster.
- RDD is computed on several processes scattered across multiple physical servers also called nodes in a cluster (while a Python collection lives and process in just one process)

In-Memory Processing

- PySpark loads the data from disk and process in memory and keeps the data in memory, this is the main difference between PySpark and Mapreduce (I/O intensive).
- In between the transformations, we can also cache/persists the RDD in memory to reuse the previous computations.

Immutability

- PySpark RDD’s are immutable in nature meaning, once RDDs are created you cannot modify. When we apply transformations on RDD, PySpark creates a new RDD and maintains the RDD Lineage.

Fault Tolerance

- PySpark operates on fault-tolerant data stores on HDFS, S3 etc hence if any RDD operation fails, it automatically reloads the data from other partitions.
- Also, When PySpark applications running on a cluster, PySpark task failures are automatically recovered for a certain number of times (as per the configuration) and finish the application seamlessly.

Lazy Evaluation

- PySpark does not evaluate the RDD transformations as they appear/encountered by Driver instead it keeps the all transformations as it encounters(DAG) and evaluates the all transformation when it sees the first RDD action.

Partitioning

- When you create RDD from a data, It by default partitions the elements in a RDD. By default it partitions to the number of cores available.

Creating RDD

- RDD's are created primarily in two different ways,
  - parallelizing an existing collection and
  - referencing a dataset in an external storage system

```
#Create RDD from parallelize
data = [1,2,3,4,5,6,7,8,9,10,11,12]
rdd=spark.sparkContext.parallelize(data)
```

```
#Create RDD from external Data source
rdd2 = spark.sparkContext.textFile("/path/textFile.txt")
```

Creating RDD

```
#Reads entire file into a RDD as single record.
rdd3 = spark.sparkContext.wholeTextFiles("/path/textFile.txt")
```

wholeTextFiles() function returns a PairRDD with the key being the file path and value being file content.

```
# Creates empty RDD with no partition
rdd = spark.sparkContext.emptyRDD
# rddString = spark.sparkContext.emptyRDD[String]

#Create empty RDD with partition
rdd2 = spark.sparkContext.parallelize([],10) #This creates 10 partitions

print("initial partition count:"+str(rdd.getNumPartitions()))
#Outputs: initial partition count:2
```

RDD Transformations

- flatMap – flatMap() transformation flattens the RDD after applying the function and returns a new RDD.

```
rdd2 = rdd.flatMap(lambda x: x.split(" "))
```

- map – map() transformation is used the apply any complex operations like adding a column, updating a column etc, the output of map transformations would always have the same number of records as input.

```
rdd3 = rdd2.map(lambda x: (x,1))
```

RDD Transformations

- reduceByKey – reduceByKey() merges the values for each key with the function specified.

```
rdd5 = rdd4.reduceByKey(lambda a,b: a+b)
```

- sortByKey – sortByKey() transformation is used to sort RDD elements on key.

```
rdd6 = rdd5.map(lambda x: (x[1],x[0])).sortByKey()
#Print rdd6 result to console
print(rdd6.collect())
```

- filter – filter() transformation is used to filter the records in an RDD.

```
rdd4 = rdd3.filter(lambda x : 'an' in x[1])
print(rdd4.collect())
```

RDD Actions

- **count**() – Returns the number of records in an RDD
- **first**() – Returns the first record.
- **max**() – Returns max record.
- **reduce**() – Reduces the records to single, we can use this to count or sum.
- **take**() – Returns the record specified as an argument.
- **collect**() – Returns all data from RDD as an array.
  - Be careful when you use this action when you are working with huge RDD with millions and billions of data as you may run out of memory on the driver.
- **saveAsTextFile**() – Using saveAsTestFile action, we can write the RDD to a text file.

Shuffle Operations

- Shuffling is a mechanism PySpark uses to redistribute the data across different executors and even across machines. PySpark shuffling triggers when we perform certain transformation operations like groupByKey(), reduceByKey(), join() on RDDS
- PySpark Shuffle is an expensive operation since it involves the following
  - Disk I/O
  - Involves data serialization and deserialization
  - Network I/O

Shuffle Operations

- For example, when we perform reduceByKey() operation, PySpark does the following
  - PySpark first runs map tasks on all partitions which groups all values for a single key.
  - The results of the map tasks are kept in memory.
  - When results do not fit in memory, PySpark stores the data into a disk.
  - PySpark shuffles the mapped data across partitions, some times it also stores the shuffled data into a disk for reuse when it needs to recalculate.
  - Run the garbage collection
  - Finally runs reduce tasks on each partition based on key.

RDD Persistance

- Using cache() and persist() methods, PySpark provides an optimization mechanism to store the intermediate computation of an RDD so they can be reused in subsequent actions.
- **Cost efficient** – PySpark computations are very expensive hence reusing the computations are used to save cost.
- **Time efficient** – Reusing the repeated computations saves lots of time.
- **Execution time** – Saves execution time of the job which allows us to perform more jobs on the same cluster.

RDD Persistance

- PySpark RDD cache() method by default saves RDD computation to storage level `MEMORY_ONLY` meaning it will store the data in the JVM heap as unserialized objects.
- PySpark persist() method is used to store the RDD to one of the storage levels MEMORY_ONLY,MEMORY_AND_DISK, MEMORY_ONLY_SER, MEMORY_AND_DISK_SER, DISK_ONLY, MEMORY_ONLY_2,MEMORY_AND_DISK_2 and more.
- You can also manually remove using unpersist() method.

RDD Shared Variables

- Broadcast variables are read-only shared variables that are cached and available on all nodes in a cluster in-order to access or use by the tasks.

```
broadcastVar = sc.broadcast([0, 1, 2, 3])
broadcastVar.value
```

- PySpark Accumulators are another type shared variable that are only “added” through an associative and commutative operation and are used to perform counters (Similar to Map-reduce counters) or sum operations.

```
accum = sc.longAccumulator("SumAccumulator")
sc.parallelize([1, 2, 3]).foreach(lambda x: accum.add(x))
```

Directed Acyclic Graph

- DAG a finite direct graph with no directed cycles. There are finitely many vertices and edges, where each edge directed from one vertex to another. It contains a sequence of vertices such that every edge is directed from earlier to later in the sequence.
- Apache Spark DAG allows the user to dive into the stage and expand on detail on any stage.

Need for DAG

- The limitations of Hadoop MapReduce became a key point to introduce DAG in Spark. The computation through MapReduce in three steps:
  - The data is read from HDFS.
  - Then apply Map and Reduce operations.
  - The computed result is written back to HDFS.
- Each MapReduce operation is independent of each other and HADOOP has no idea of which Map reduce would come next. Sometimes for some iteration, it is irrelevant to read and write back the immediate result between two map-reduce jobs. In such case, the memory in stable storage (HDFS) or disk memory gets wasted.

Need for DAG

- In multiple-step, till the completion of the previous job all the jobs block from the beginning. As a result, complex computation can require a long time with small data volume.

- While in Spark, a DAG (Directed Acyclic Graph) of consecutive computation stages is formed. In this way, we optimize the execution plan, e.g. to minimize shuffling data around. In contrast, it is done manually in MapReduce by tuning each MapReduce step.

DAG Working in Pyspark

- The interpreter is the first layer, using a Scala interpreter, Spark interprets the code with some modifications.
- Spark creates an operator graph when you enter your code in Spark console.
- When we call an Action on Spark RDD at a high level, Spark submits the operator graph to the DAG Scheduler.
- Divide the operators into stages of the task in the DAG Scheduler. A stage contains task based on the partition of the input data. The DAG scheduler pipelines operators together. For example, map operators schedule in a single stage.
- The stages pass on to the Task Scheduler. It launches task through cluster manager. The dependencies of stages are unknown to the task scheduler.
- The Workers execute the task on the slave.

DAG Visualization