

Map Reduce Fundamentals

WELCOME

“Secret Of the **Business** is to know Something that nobody else know”



Presented by
Siva Kumar Bhuchipalli

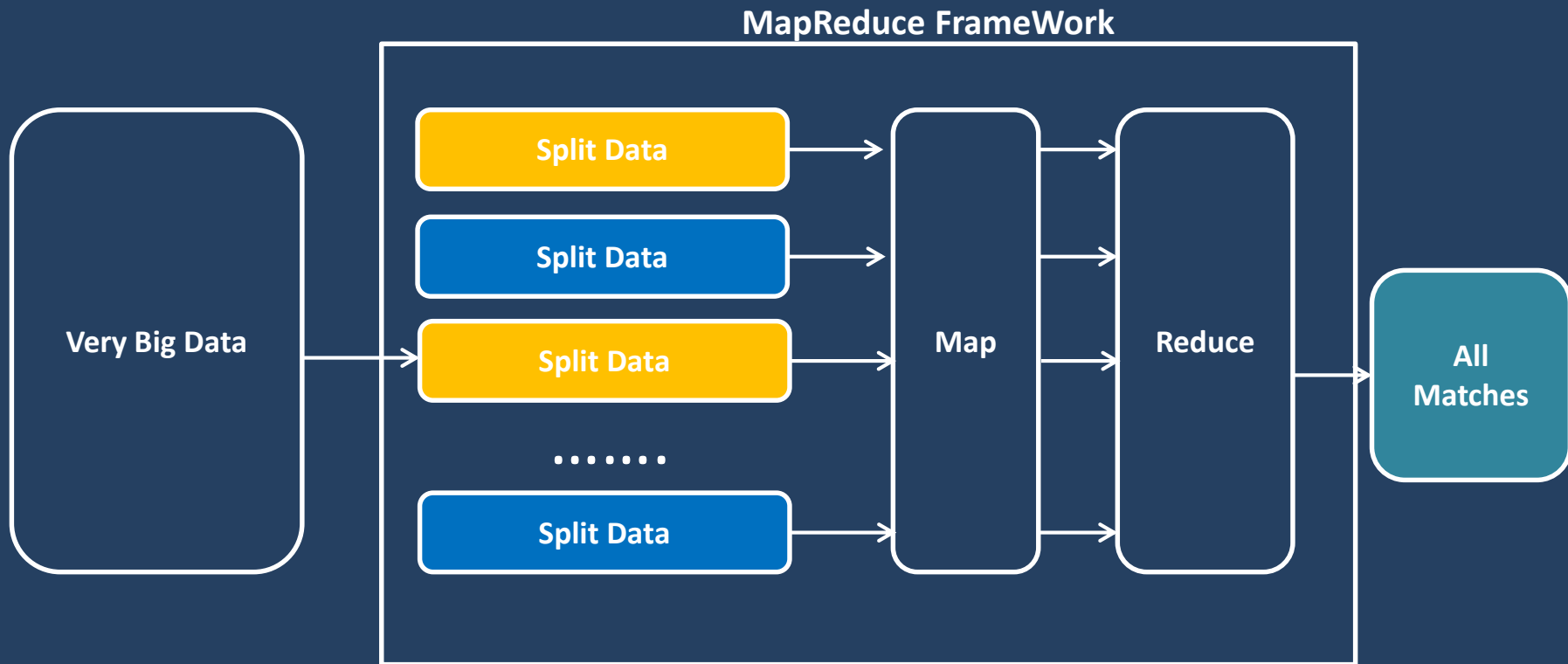


Contents

- Introduction
- History
- (Key, Value) Model
- MR Paradigm for WordCount



Map Reduce Solution

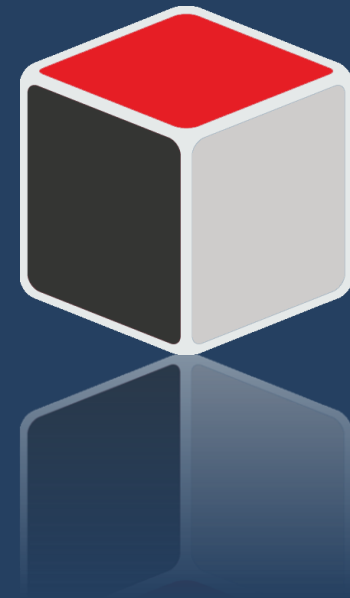


No data transfer and achieving data locality.

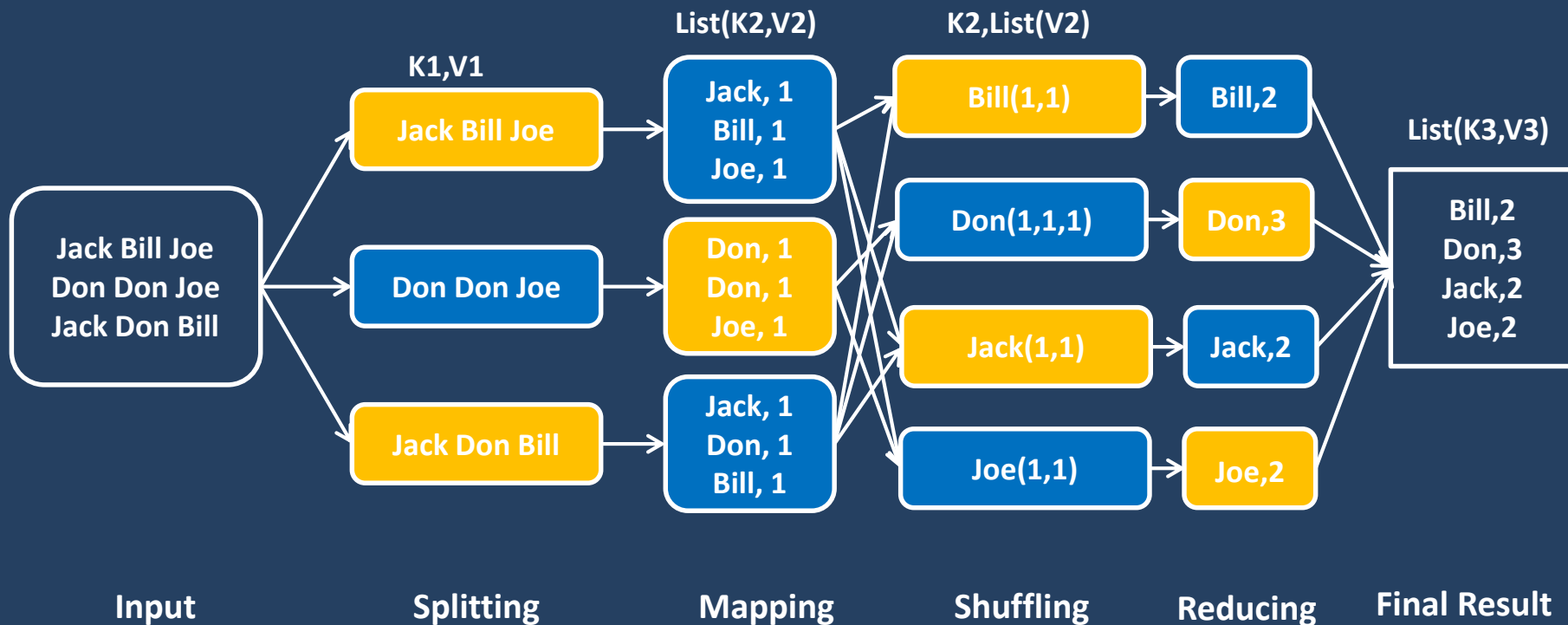
Map Reduce Model

- ❖ Imposes Key-Value Input/Output
- ❖ Define Map and Reduce Functions
 - map: $(K1, V1) \rightarrow \text{list}(K2, V2)$
 - reduce: $(K2, \text{List}(V2)) \rightarrow \text{list}(K3, V3)$

- ✓ Map function is applied to every input key-value pair.
- ✓ Map function generates intermediate key-value pair.
- ✓ Intermediate key-values are sorted and grouped by key.
- ✓ Reduce is applied to sorted and grouped intermediate key-values.
- ✓ Reduce emits results key-values.



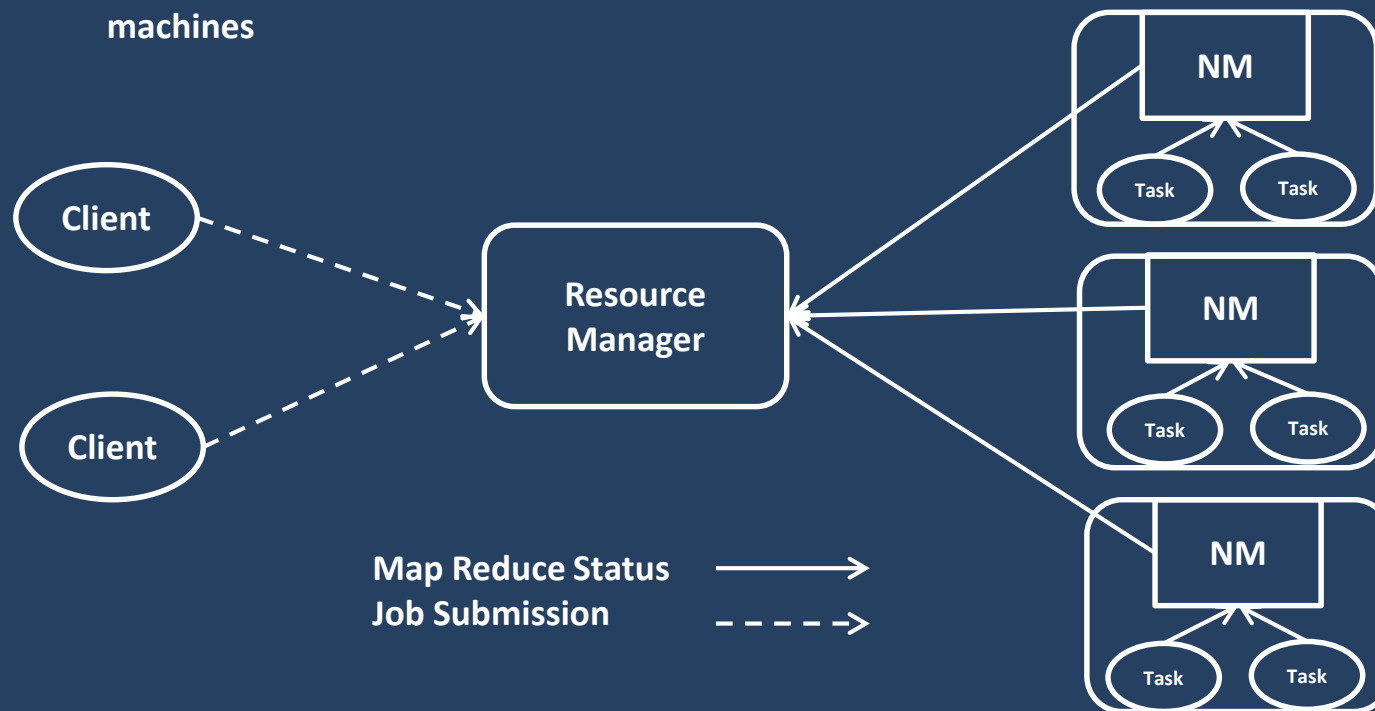
Map Reduce Paradigm(Word Count)

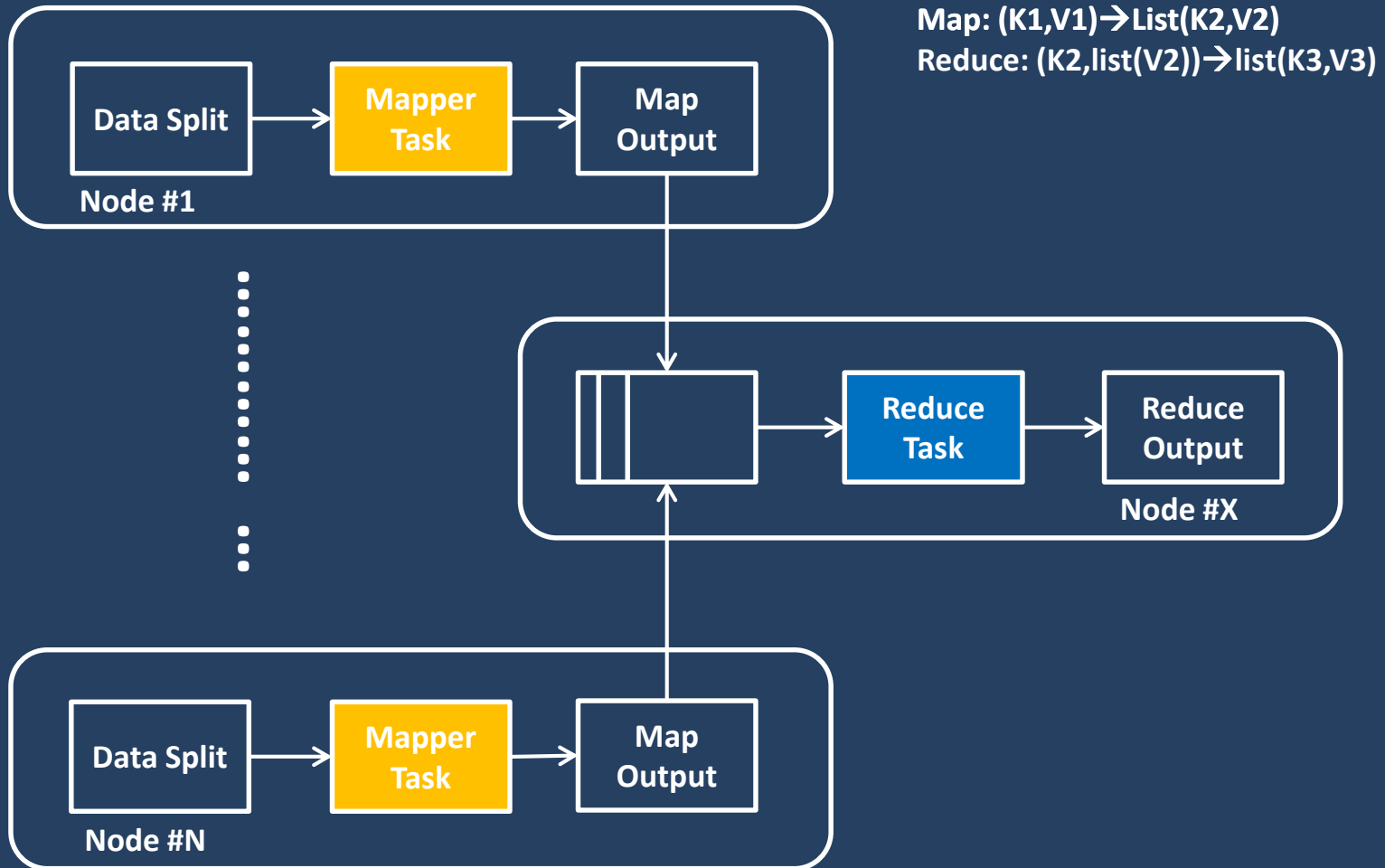


Map Reduce Working Flow

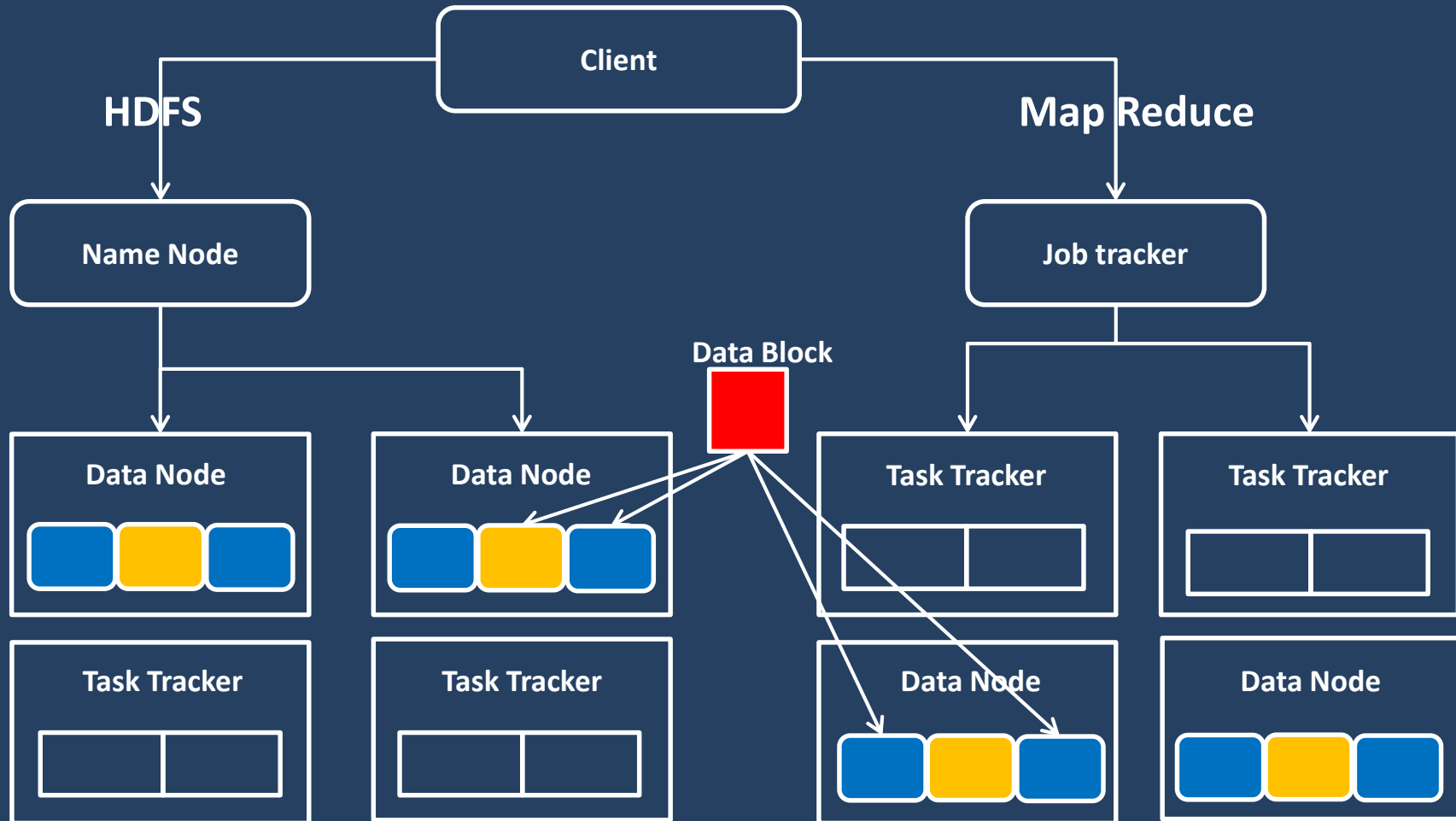


- ❖ Input data is present in data nodes
- ❖ Map tasks = Input Splits
- ❖ Mappers produce intermediate data
- ❖ Data is exchanged among nodes in “shuffling”
- ❖ All data of same key goes to same reducer
- ❖ Reducer output is stored at output location
- ❖ Jobs are broken down into smaller chunks called tasks
- ❖ These tasks are scheduled
- ❖ Code is moved to where the data is. Shuffle and Sort barrier re-arranges and moves data between machines





Map Reduce v1.0



Map Reduce Limitation

- ❖ Job Tracker Overburdened - Spends significant amount of time and effort managing the life-cycle of applications
- ❖ MRv1(Only Map and Reduce Tasks).
- ❖ Inflexible Resource Management(MapReduce1 had slot based model)
- ❖ Cluster Resource sharing and allocation flexibility
 - ✓ Slot based approach (ex. 10 slots per machine no matter how small or big those tasks are)



Map Reduce 2.0 On YARN

- ❖ YARN - Yet Another Resource Negotiator
- ❖ YARN was designed to address scalability and performance issues with MRv1
- ❖ Job Tracker is split into 2 daemons
 - ✓ Resource Manager - administers resources on the cluster
 - ✓ Application Master - manages applications such as MapReduce
- ❖ Fine-Grained memory management model
 - ✓ Application Master requests resources by asking for “containers” with a certain memory limit (ex 2G)
 - ✓ YARN administers these containers and enforces memory usage
 - ✓ Each Application/Job has control of how much memory to request

MRV1

Process & Speed

Resource Management

API

API

mapper , reducer ==> processor

||||

Job Tracker, Task Tracker

mapred.Mapper, mapred.Reducer

YARN

Mrv2 (mapreduce.Mapper, mapreduce.Reducer) ||||

Resource Manager, AppMaster, Node Manager

Tez

Spark

02-11-2016

www.hadooptutorial.info/

15



Daemons In MRv2

❖ YARN Daemons

❑ Resource Manager

- ✓ Manages Resources for a Cluster
- ✓ Instructs Node Manager to allocate resources
- ✓ Application Master negotiates for resources with Resource Manager
- ✓ There is only one instance of Resource Manager

❑ Node Manger

- ✓ Manages resources of a single node
- ✓ There is one instance per node in the cluster

❑ Application Master

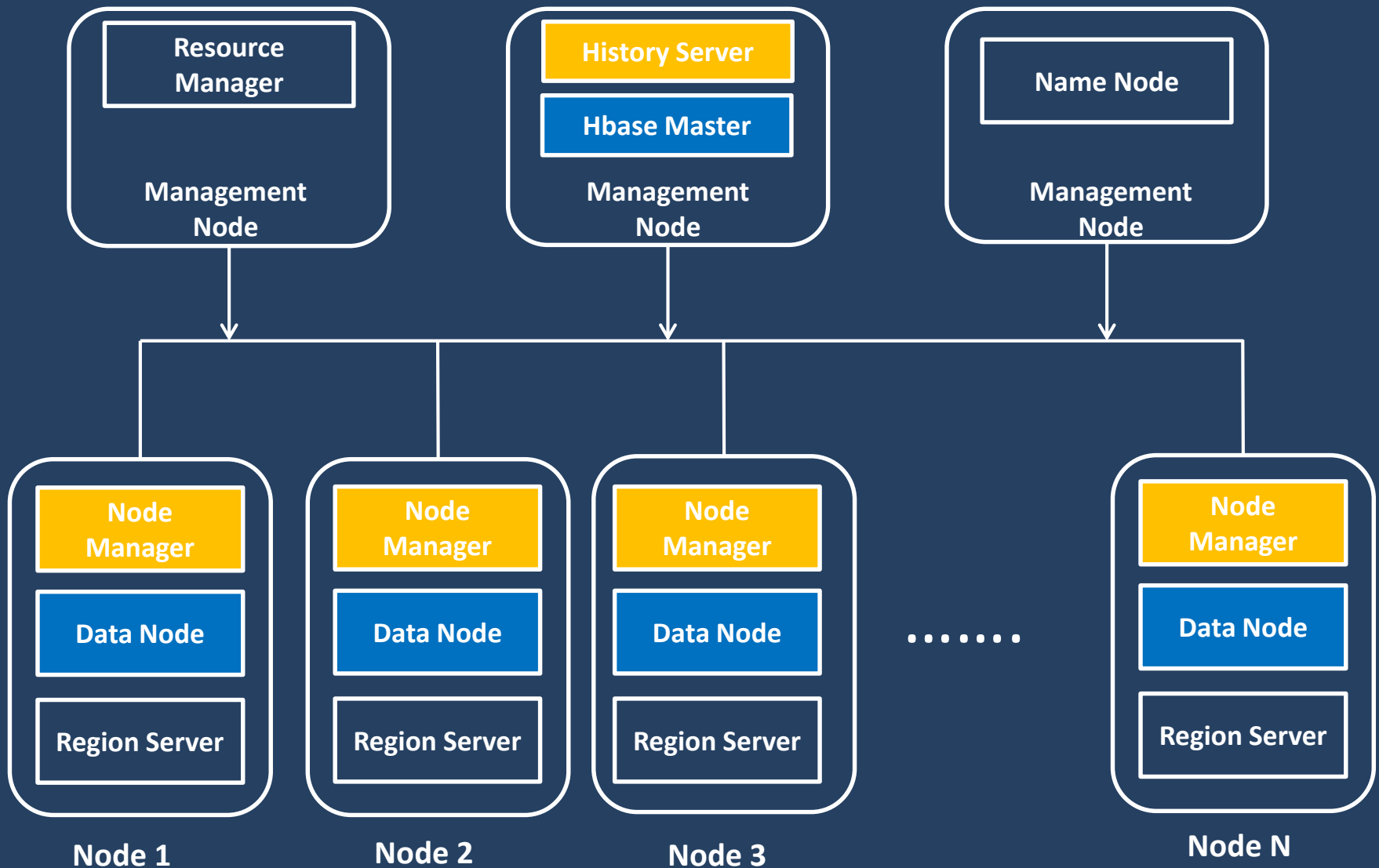
- ✓ Active only during job execution
- ✓ Runs on any one of the available Node Managers
- ✓ Monitors the job tasks and their progression

❖ Map Reduce Specific Daemon

❑ Map Reduce History Server

- ✓ Archives Jobs' metrics and meta-data

Daemons In MRv2



MR Word Count Job

❖ Implement Mapper

- ✓ Input is text – a line from test.txt
- ✓ Tokenize the text and emit word with a count of 1 - <token, 1>
- ❖ Our class should extend Mapper class
 - ✓ Mapper<KeyIn, ValueIn, KeyOut, ValueOut>
- ❖ Life cycle of Mapper
 - ✓ The framework first calls setup(Context)
 - ✓ For each key/value pair in the split:
 - map(Key, Value, Context)
 - ✓ Finally cleanup(Context) is called

```
class Mapper{  
    public void setup(Context context)  
    {  
    }  
    map(key, value, context)  
    {  
    }  
    cleanup(Context context)  
    {  
    }  
    run()  
    {  
        setup();  
        for (all key-value pairs)  
        {  
            map(k1,v1,c);  
        }  
        cleanup();  
    }  
}
```



MR Word Count Job

❖ Implement Reducer

- ✓ Sum up counts for each letter
- ✓ Write out the result to HDFS
- ❖ Our class should extend Reducer class
 - ✓ `Reducer<KeyIn, ValueIn, KeyOut, ValueOut>`
 - ✓ `KeyIn` and `ValueIn` types must match output types of mapper
- ❖ Receives input from mappers' output
 - ✓ Sorted on key
 - ✓ Grouped on key of key-values produced by mappers
 - ✓ Input is directed by Partitioner implementation
- ❖ Simple life-cycle – similar to Mapper
 - ✓ The framework first calls `setup(Context)`
 - ✓ For each key → `list(value)` calls
 - `reduce(Key, Values, Context)`
 - ✓ Finally `cleanup(Context)` is called

❖ Configure the Job

- ✓ Specify Input, Output, Mapper, Reducer and Combiner

❖ Run the job - **With single Input file and multiple input files**



WordCountMapper.java

```
package test.mr.project;

import java.io.IOException;
import java.util.StringTokenizer;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;

public class WordCountMapper extends Mapper<LongWritable, Text, Text,
IntWritable> {

    @Override
    protected void map(LongWritable key, Text value, Context context)
        throws IOException, InterruptedException {
        String[] words = value.toString().split(' ');
        for(String word: words) {
            context.write(new Text(word), new IntWritable(1));
        }
    }
}
```

WordCountReducer.java

```
package test.mr.project;

import java.io.IOException;

import org.apache.hadoop.io.*;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.log4j.Logger;

public class StartsWithCountReducer extends
    Reducer<Text, IntWritable, Text, IntWritable> {
    Logger log = Logger.getLogger(StartsWithCountMapper.class);
    @Override
    protected void reduce(Text token, Iterable<IntWritable> counts,
        Context context) throws IOException, InterruptedException {
        int sum = 0;
        for (IntWritable count : counts) {
            sum += count.get();
        }
        context.write(token, new IntWritable(sum));
    }
}
```


WordCountJob.java

```
package test.mr.project;

public class WordCountJob {
    public static void main(String[] args) throws Exception {
        Job job = Job.getInstance(getConf(), "WordCount");
        job.setJarByClass(getClass());

        TextInputFormat.addInputPath(job, new Path(args[0]));
        job.setInputFormatClass(TextInputFormat.class);

        // configure mapper and reducer
        job.setMapperClass(WordCountMapper.class);
        job.setReducerClass(WordCountReducer.class);

        TextOutputFormat.setOutputPath(job, new Path(args[1]));
        job.setOutputFormatClass(TextOutputFormat.class);
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(IntWritable.class);

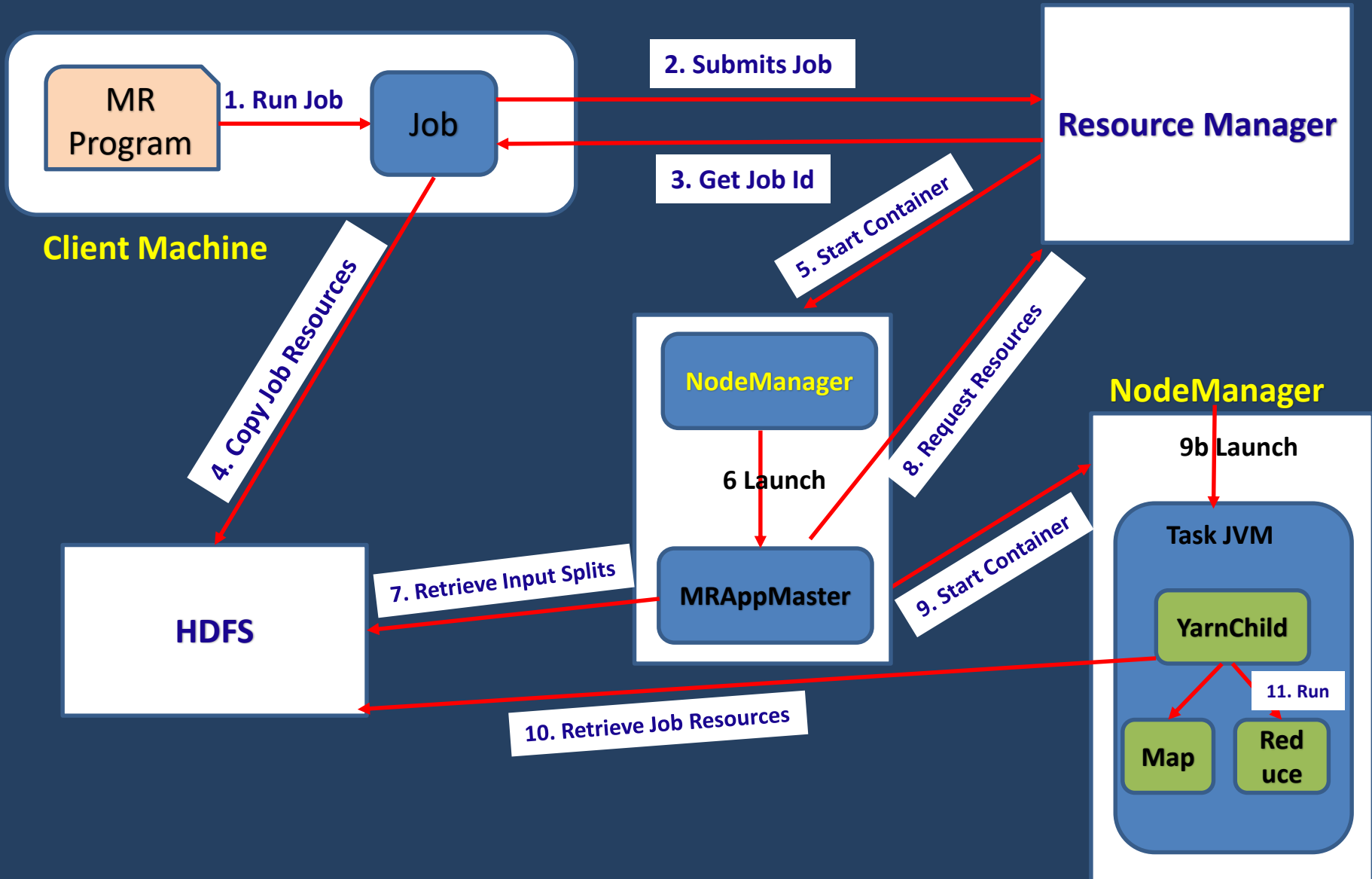
        System.exit(job.waitForCompletion(true) ? 0 : 1);
    }
}
```

YARN Job Anatomy

- ❖ **Client** – submits Map Reduce Job
- ❖ **Resource Manager** – controls the use of resources across the Hadoop cluster
- ❖ **Node Manager** – runs on each node in the cluster; creates execution container, monitors container's usage
- ❖ **Map Reduce Application Master** – Coordinates and manages Map Reduce Jobs; negotiates with Resource Manager to schedule tasks; the tasks are started by Node Manager(s)
- ❖ **HDFS** – shares resources and jobs' artifacts between YARN components



Job Anatomy



MRv2 On YARN Job Execution

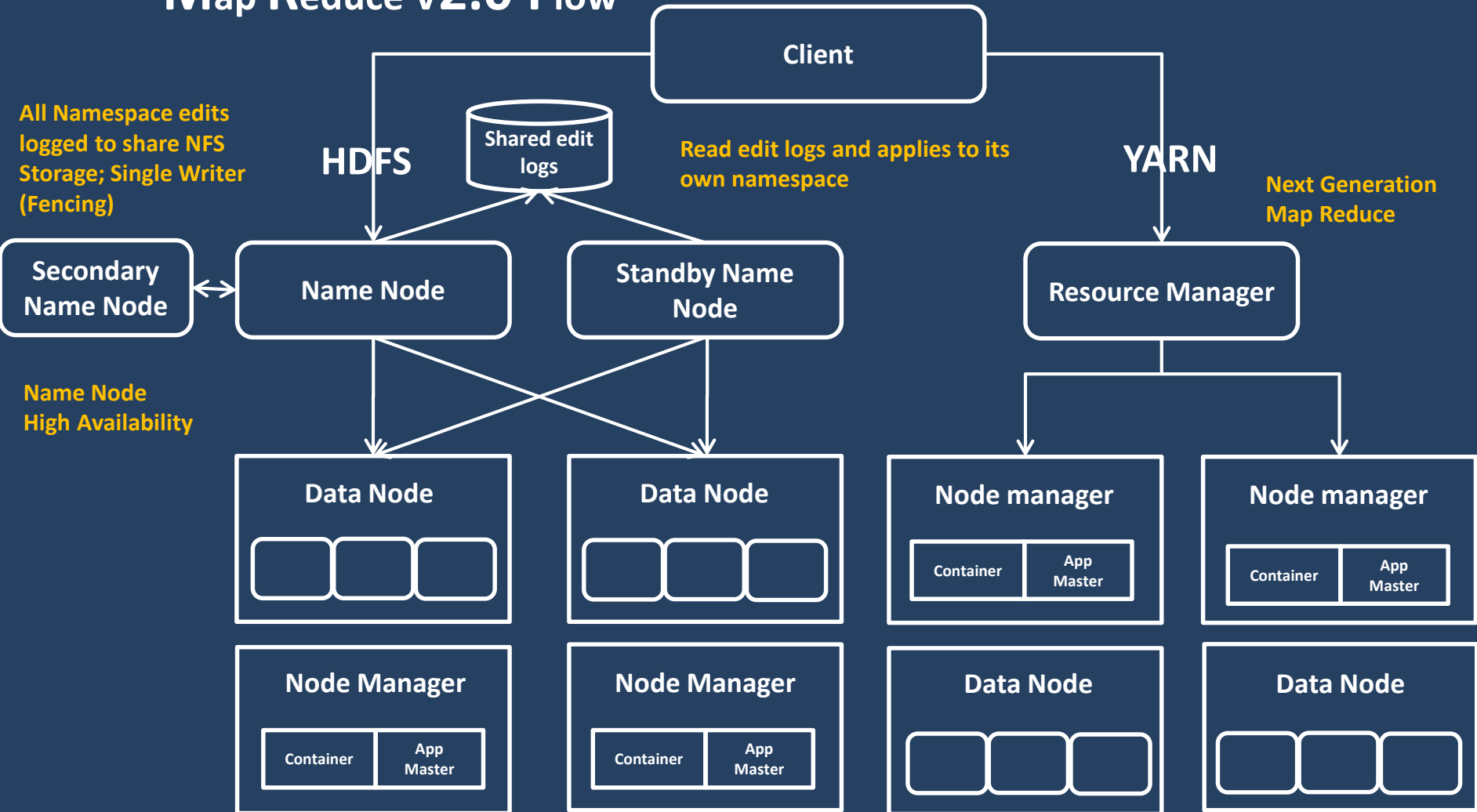
- ❖ Client submits MapReduce job by interacting with Job objects; Client runs in it's own JVM
- ❖ Use `org.apache.hadoop.mapreduce.Job` class to configure the job
- ❖ Submit the job to the cluster and wait for it to finish.
`job.waitForCompletion(true)`
- ❖ The YARN protocol is activated when `mapreduce.framework.name` property in `mapred-site.xml` is set to `yarn`
- ❖ Job's code interacts with Resource Manager to acquire application meta-data, such as application id
- ❖ Job's code moves all the job related resources to HDFS to make them available for the rest of the job
- ❖ Resource Manager chooses a Node Manager with available resources and requests a container for MRAppMaster
- ❖ Node Manager allocates container for MRAppMaster; MRAppMaster will execute and coordinate MapReduce job



- ❖ MRAppMaster grabs required resource from HDFS, such as Input Splits; these resources were copied there in step 4
- ❖ MRAppMaster negotiates with Resource Manager for available resources; Resource Manager will select Node Manager that has the most resources
- ❖ MRAppMaster tells selected NodeManager to start Map and Reduce tasks
- ❖ NodeManager creates YarnChild containers that will coordinate and run tasks
- ❖ YarnChild acquires job resources from HDFS that will be required to execute Map and Reduce tasks
- ❖ YarnChild executes Map and Reduce tasks



Map Reduce v2.0 Flow



Failure In MRv2

- ❖ Failures can occur in
 - ✓ Tasks
 - ✓ Application Master – MRAppMaster
 - ✓ Node Manager
 - ✓ Resource Manager

Task Failures

- ❖ Most likely offender and easiest to handle
- ❖ Task's exceptions and JVM crashes are propagated to MRAppMaster
 - ✓ Attempt (not a task) is marked as 'failed'
- ❖ Hanging tasks will be noticed, killed
 - ✓ Attempt is marked as failed
 - ✓ Control via **mapreduce.task.timeout** property
- ❖ Task is considered to be failed after 4 attempts
 - ✓ Set for map tasks via **mapreduce.map.maxattempts**
 - ✓ Set for reduce tasks via **mapreduce.reduce.maxattempts** 32



Application Master Failures MRAppMaster

- ❖ MRAppMaster Application can be re-tried
 - ✓ By default will not re-try and will fail after a single application failure
- ❖ Enable re-try by increasing **yarn.resourcemanager.am.max-retries** property
- ❖ Resource Manager receives heartbeats from MRAppMaster and can restart in case of failure(s)
- ❖ Restarted MRAppMaster can recover latest state of the tasks
 - ✓ Completed tasks will not need to be re-run
 - ✓ To enable set **yarn.app.mapreduce.am.job.recovery.enable** property to true



Node Manager Failures

- ❖ Failed Node Manager will not send heartbeat messages to Resource Manager
- ❖ Resource Manager will black list a Node Manager that hasn't reported within 10 minutes
 - ✓ Configure via property:
 - `yarn.resourcemanager.nm.liveness-monitor.expiryinterval-ms`
 - ✓ Usually there is no need to change this setting
- ❖ Tasks on a failed Node Manager are recovered and placed on healthy Node Managers



Node Manager Blacklisted By MRAppMaster

- ❖ MRAppMaster may blacklist Node Managers if the number of failures is high on that node
- ❖ MRAppMaster will attempt to reschedule tasks on a blacklisted Node Manager onto Healthy Nodes
- ❖ Blacklisting is per Application/Job therefore doesn't affect other Jobs
- ❖ By default blacklisting happens after 3 failures on the same node
 - ✓ Adjust default via `mapreduce.job.maxtaskfailures.per.tracker`



Resource Manager Failure

- ❖ The most serious failure = downtime
 - ✓ Jobs or Tasks can not be launched
- ❖ Resource Manager was designed to automatically recover
 - ✓ Incomplete implementation at this point
 - ✓ Saves state into persistent store by configuring `yarn.resourcemanager.store.class` property
 - `org.apache.hadoop.yarn.server.resourcemanager.recover`
`y.MemStore`
 - ✓ The only stable option for now is in-memory store
 - You can track progress via

<https://issues.apache.org/jira/browse/MAPREDUCE-4345>



Speculative Execution

- ❖ Will spawn a speculative task when
 - ✓ All the tasks have been started
 - ✓ Task has been running for an extended period of time over a minute
 - ✓ Did not make significant progress as compared to the rest of the running tasks
- ❖ After task's completion duplicates are killed
- ❖ Just an optimization
- ❖ Can be turned off by setting these properties to false
 - ✓ **mapred.map.tasks.speculative.execution**
 - Turn on/off speculative execution for map phase
 - ✓ **mapred.reduce.tasks.speculative.execution**
 - Turn on/off speculative execution for reduce phase
- ❖ When should I disable Speculative Execution?
 - ✓ Task is outputting directly to a shared resource; then starting a duplicate task may cause unexpected results
 - ✓ Minimize cluster and bandwidth usage; duplicate tasks use up resources



Job Scheduling

- ❖ By default FIFO scheduler is used
 - ✓ First In First Out
 - ✓ Supports basic priority model: VERY_LOW, LOW, NORMAL, HIGH, and VERY_HIGH
 - ✓ Two ways to specify priority
 - `mapreduce.job.priority` property
 - `job.setPriority(JobPriority.HIGH)`
- ❖ Specify scheduler via `yarn.resourcemanager.scheduler.class` property
 - ✓ CapacityScheduler
 - ✓ FairScheduler



Map Reduce Old VS New JAVA API

- ❖ There are two flavors of MapReduce API which became known as Old and New
- ❖ Old API classes reside under
 - ✓ `org.apache.hadoop.mapred`
- ❖ New API classes can be found under
 - ✓ `org.apache.hadoop.mapreduce`
 - ✓ `org.apache.hadoop.mapreduce.lib`
- ❖ We will use new API exclusively
- ❖ New API was re-designed for easier evolution
- ❖ Early Hadoop versions deprecated old API but
- ❖ Recently deprecation was removed
- ❖ Do not mix new and old API



Advanced Concepts Of MR

- ❖ Combiners
- ❖ Partitioners
- ❖ Counters
- ❖ Different input and output formats
- ❖ Custom Data Types
- ❖ Joins
 - ✓ Map-side Join
 - ✓ Reduce-side Join
- ❖ Distributed Cache

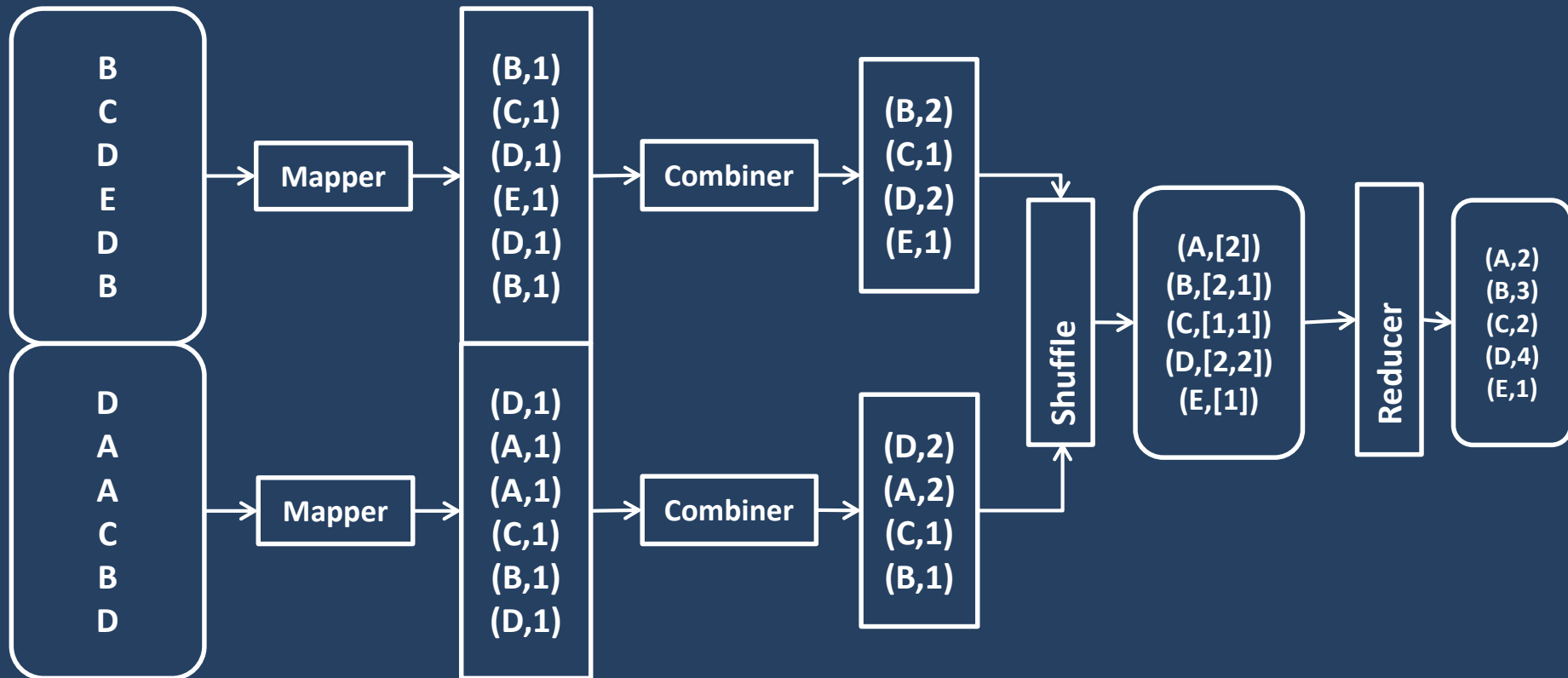


Combiner In MR

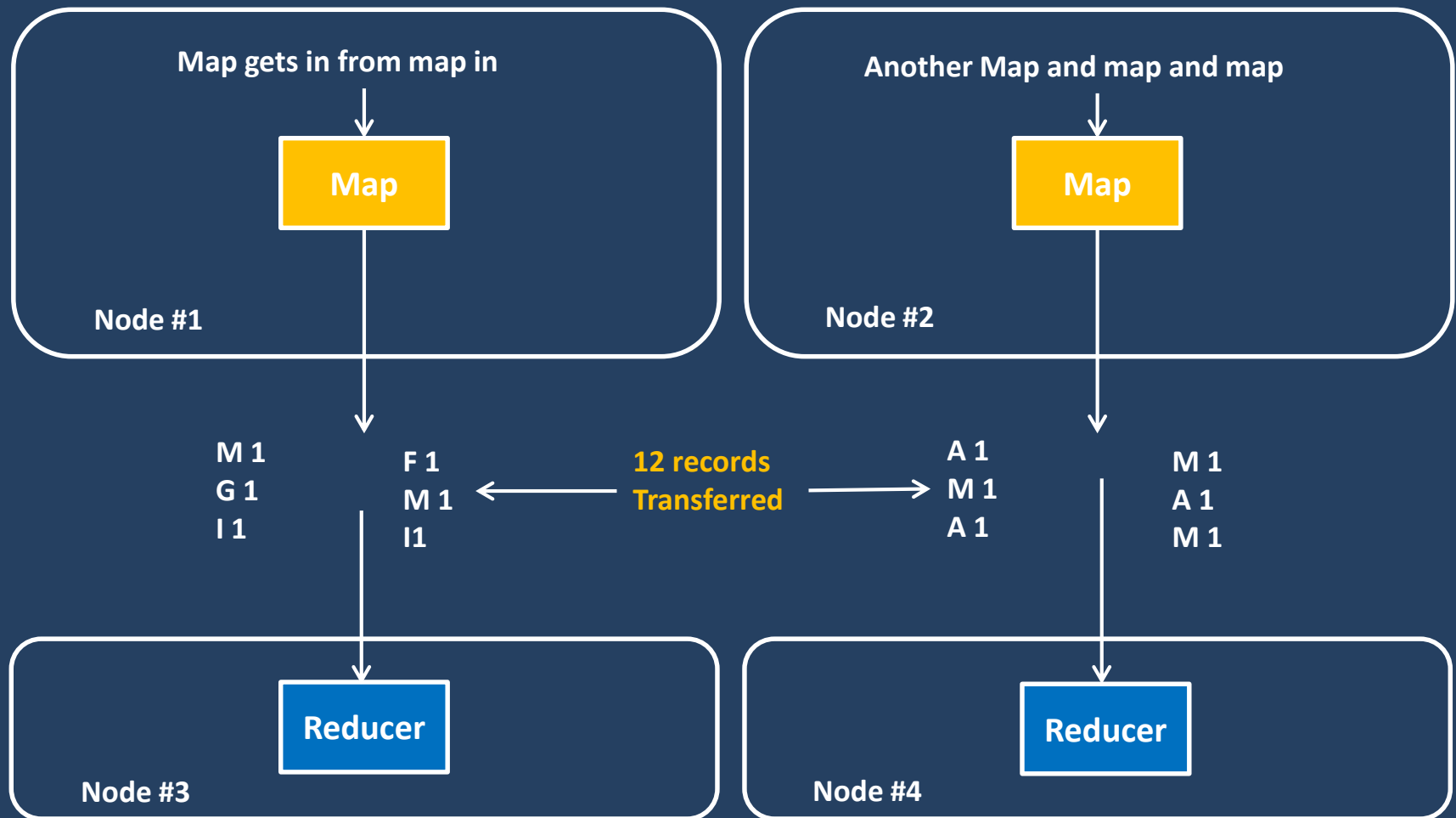
- ❖ Combiners can be treated as "local-reducers" in the Mapper phase
- ❖ Combiners are used to improve the performance of MR Job
- ❖ Combiners can primarily use for decreasing the amount of data needed to be processed by Reducers. In some cases, because of the nature of the algorithm you implement, this function can be the same as the Reducer. But in some other cases this function can of course be different.
- ❖ One constraint that a Combiner will have, unlike a Reducer, is that the input/output key and value types must match the output types of your Mapper.
- ❖ Combiners can only be used on the functions that are **commutative**($a.b = b.a$) and **associative** $\{a.(b.c) = (a.b).c\}$. This also means that combiners may operate only on a subset of your keys and values or may not execute at all, still you want the output of the program to remain same.
- ❖ Reducers can get data from multiple Mappers as part of the partitioning process. Combiners can only get its input from one Mapper.
- ❖ As it is just an optimization technique Hadoop may run Combiner 0 or many times. It can't guarantee the no of runs
- ❖ Combiner is good for getting Aggregations like **Sum, Count, Max Min but not for Average, Mean, Std deviations, variance etc.**



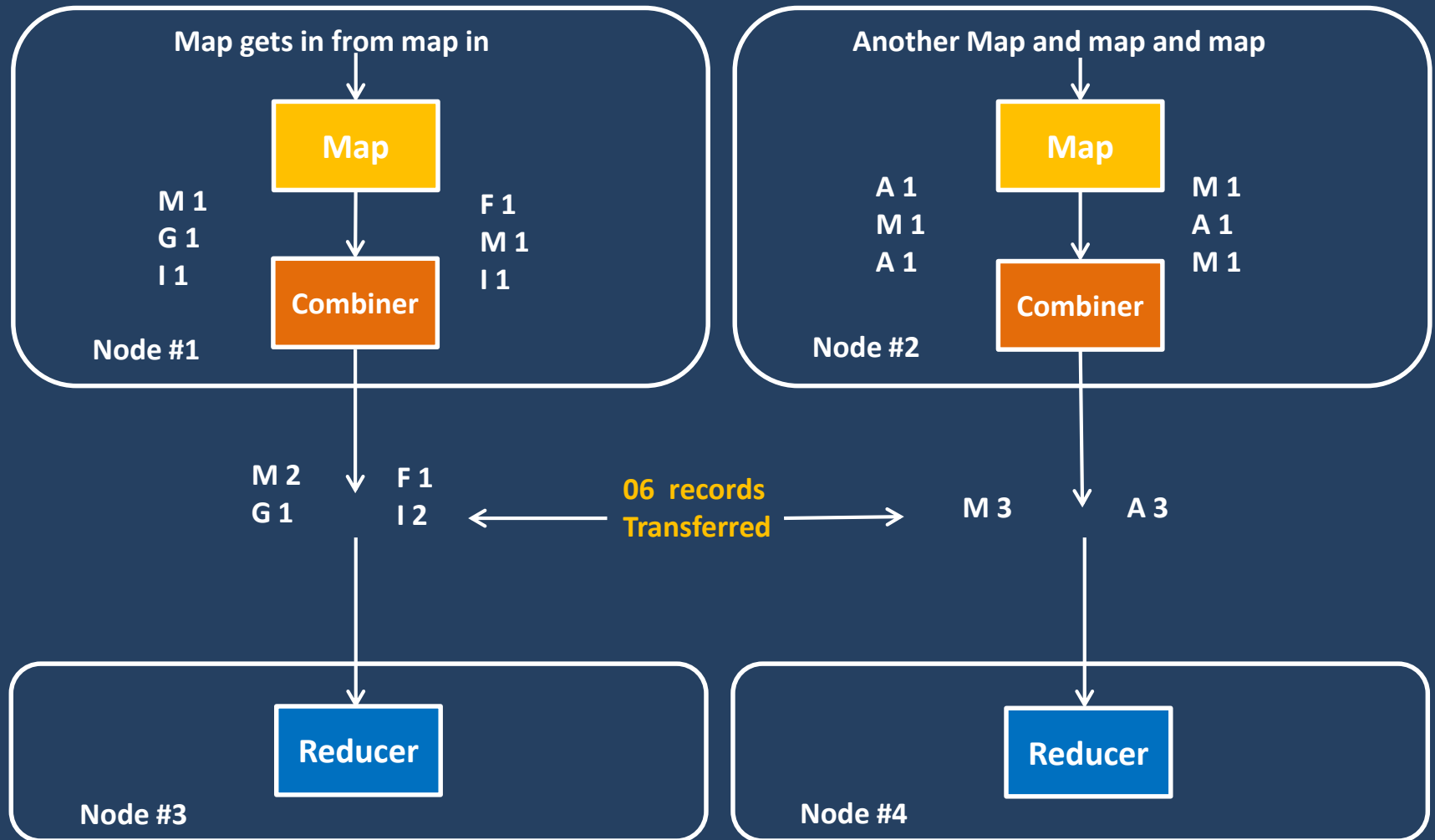
Combiner Flow



Running A Job Without Combiner



Running A Job With Combiner



Data transfer will be less when we use combiner.

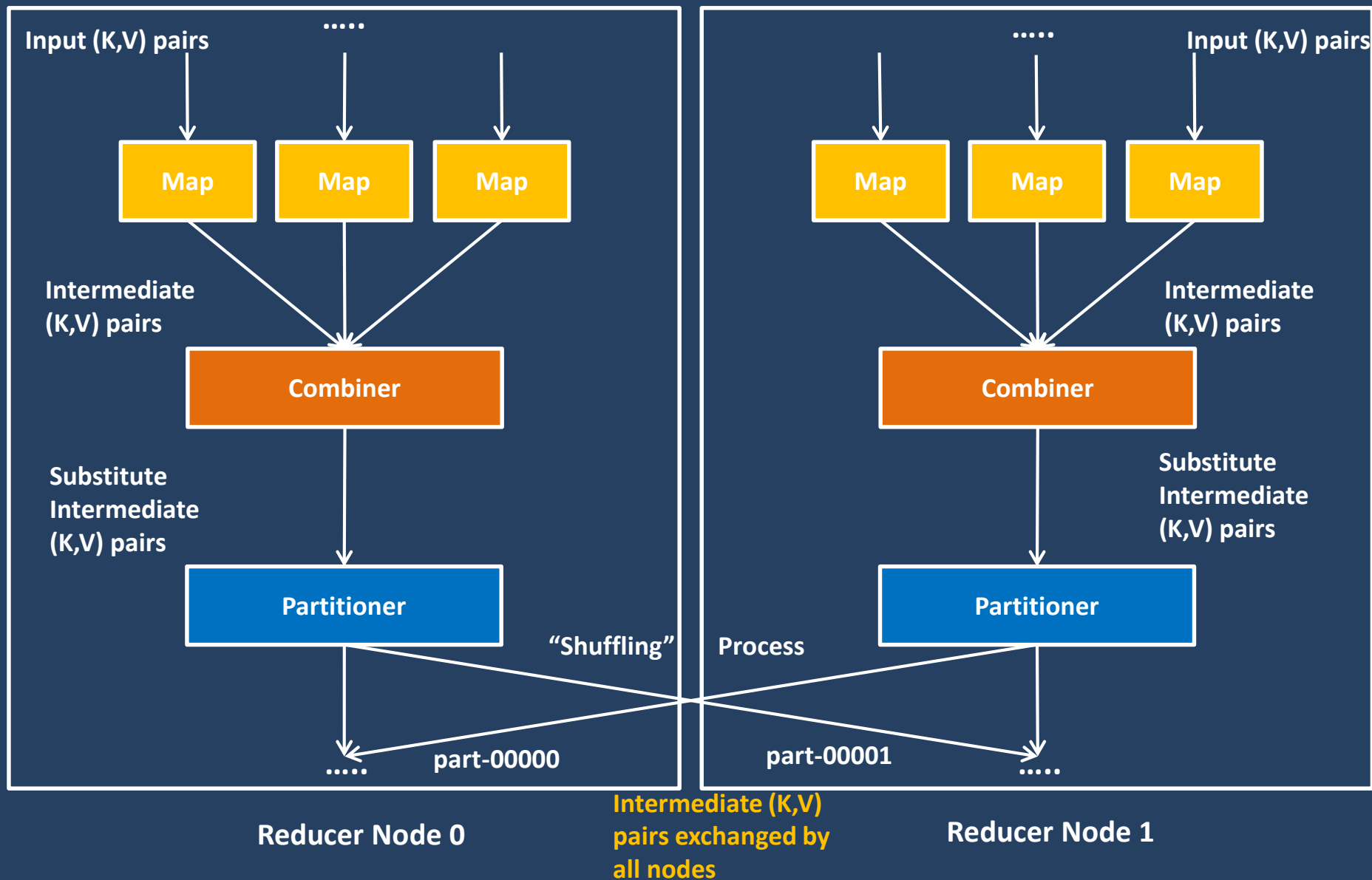
Partitioners

Partitioning phase takes place after the map phase and before the reduce phase. The number of partitions is equal to the number of reducers. The data gets partitioned across the reducers according to the partitioning function .

The difference between a partitioner and a combiner is that the partitioner divides the data according to the number of reducers so that all the data in a single partition gets executed by a single reducer. However, the combiner functions similar to the reducer and processes the data in each partition.

The combiner is an optimization to the reducer. The default partitioning function is the hash partitioning function where the hashing is done on the key. However it might be useful to partition the data according to some other function of the key or the value.

Partitioner Flow



Hash Partitioner

```
Public class HashPartitioner<K, V> extends Partitioner<K, V>
{
    Public Int getPartition(K key, V values, int numreduceTasks)
    {
        Return (key.hashCode() & Integer.MAX_VALUE) %
        numReduceTasks;
    }
}
```

❖ Calculate Index Of Partition:

- ✓ Convert key's hash into non-negative number.
 - Logical AND with maximum integer value.
- ✓ Modulo by number of Reduce tasks.

❖ In case of more than 1 reducer


- ✓ Records distributed evenly across available reduce tasks
 - Assuming a good hashCode() function
- ✓ Records with same key will make it into the same reduce task
- ✓ Code is independent from the # of Partitions/reducers specified

Custom Partitioner

```
Public class CustomPartitioner extends Partitioner<Text, BlogWritable>
{
  @override
  Public Int getPartition(Text key, BlogWritable values, int numreduceTasks)
  {
    Int positiveHash = blog.getAuthor().hashCode() & Integer.MAX_VALUE;

    Return positiveHash % numReduceTasks;
  }
}
```

Use Authors hash only, AND with
max integer to get a positive value



Assign a reduce task by index



All blogs with the same Author will end up in the same Reducer task

Counters In MR

- ❖ Counters are lightweight objects in MapReduce that allow you to keep track of system progress in both the map and reduce stages of processing
- ❖ Counters are used to gather information about the data we are analyzing, like how many types of records were processed, how many invalid records were found while running the job, and so on

```
14/07/24 05:56:17 INFO mapreduce.Job: Running job: job_1406091546185_0024
14/07/24 05:56:25 INFO mapreduce.Job: Job job_1406091546185_0024 running in uber mode : false
14/07/24 05:56:25 INFO mapreduce.Job: map 0% reduce 0%
14/07/24 05:56:34 INFO mapreduce.Job: map 100% reduce 0%
14/07/24 05:56:47 INFO mapreduce.Job: map 100% reduce 100%
14/07/24 05:56:47 INFO mapreduce.Job: Job job_1406091546185_0024 completed successfully
14/07/24 05:56:47 INFO mapreduce.Job: Counters: 43
    File System Counters
        FILE: Number of bytes read=23754
        FILE: Number of bytes written=206017
        FILE: Number of read operations=0
        FILE: Number of large read operations=0
        FILE: Number of write operations=0
        HDFS: Number of bytes read=9458
        HDFS: Number of bytes written=1154
        HDFS: Number of read operations=6
        HDFS: Number of large read operations=0
        HDFS: Number of write operations=2
    Job Counters
        Launched map tasks=1
        Launched reduce tasks=1
        Data-local map tasks=1
        Total time spent by all maps in occupied slots (ms)=6555
        Total time spent by all reduces in occupied slots (ms)=9184
```


- ❖ Framework provides a set of built-in metrics
 - ✓ For example bytes processed for input and output
- ❖ User can create new counters
 - ✓ Number of records consumed
 - ✓ Number of errors or warnings
- ❖ Counters are divided into groups
- ❖ Tracks total, Mapper and Reducer counts



Built In Counters In MR

- ❖ Maintains and sums up counts
- ❖ Several groups for built-in counters
 - ✓ Job Counters – documents number of map and reduce tasks launched, number of failed tasks
 - ✓ File System Counters – number of bytes read and written
 - ✓ Map-Reduce Framework – mapper, reducer, combiner input and output records counts, time and memory statistics
- ❖ Web UI exposes counters for each Job



Logged in as webuser

Counters for job_1339260384899_0001

Counter Group	Name	Map	Reduce	Total
File System Counters	FILE: Number of bytes read	298	0	298
	FILE: Number of bytes written	104542	0	104542
	FILE: Number of large read operations	0	0	0
	FILE: Number of large write operations	0	0	0
	FILE: Number of read operations	0	0	0
	FILE: Number of write operations	0	0	0
	HDFS: Number of bytes read	51853	0	51853
	HDFS: Number of bytes written	62586	0	62586
	HDFS: Number of large read operations	0	0	0
	HDFS: Number of large write operations	0	0	0
Job Counters	Launched map tasks	0	0	1
	Launched reduce tasks	0	0	1
	Total time spent by all maps in occupied slots (ms)	0	0	4408
Map-Reduce Framework	CPU time spent (ms)	1920	0	1920
	Failed Shuffles	0	0	0
	GC time elapsed (ms)	21	0	21
	Input split bytes	169	0	169
	Map input records	1	0	1
	Map output records	0	0	0
	Merged Map outputs	0	0	0
	Physical memory (bytes) snapshot	85540864	0	85540864
	Spilled Records	0	0	0
	Total committed heap usage (bytes)	37355520	0	37355520

Implemented Custom Counters In MR

- ❖ Counters are lightweight objects in MapReduce that allow you to keep track of system progress in both the map and reduce stages of processing
- ❖ Counters are used to gather information about the data we are analyzing, like how many types of records were processed, how many invalid records were found while running the job, and so on
- ❖ We can retrieve Counter from Context object
 - ✓ Framework injects Context object into map and reduce Methods
- ❖ Increment Counter's value
 - ✓ Can increment by 1 or more
- ❖ We can get counter object with the help of Context object which is available in map and reduce methods
 - ✓ `void map(Key key, Value value, Context context)`
 - ✓ `void reduce(Key key, Iterable<Value> values, Context context)`
- ❖ We can Increment or even set the value of counter
 - ✓ `void setValue(long value);`
 - ✓ `void increment(long incr);`



Example Usage Of Custom Counters In Map

- ❖ Update Mapper to document counts for
 - Total tokens processed
 - Number of tokens that start with uppercase
 - Number of tokens that start with lowercase
- ❖ First create an enum to reference these counters:
`public enum Tokens { Total, FirstCharUpper, FirstCharLower }`

Protected void map (LongWritable key, Text Value, Context context)

Throws IOException, InterruptedException

```
{  
    StringTokenizer tokenizer = new StringTokenizer(value.toString());  
    while(tokenizer.hasMoreTokens())  
    {  
        String token = tokenizer.nextToken();  
        reusableText.set(token.substring(0,1));  
        context.write(reusableText, countOne);  
  
        context.getCounter(Tokens.Total).increment(1);  
        Char firstChar = token.charAt(0);  
        If(Character.isUpperCase(firstChar))  
        {  
            context.getCounter(Tokens.FirstCharUpper).increment(1);  
        }  
        else  
        {  
            context.getCounter(Tokens.FirstCharLower).increment(1);  
        }  
    }  
}
```

Keep Count of total
Tokens processed

Stats on tokens that start
with Upper case vs Lower

Example Usage Of Custom Counters In Map

```
$ yarn jar $PLAY_AREA/HadoopSamples.jar  
mr.wordcount.StartsWithCountJob_UserCounters  
/training/data/hamlet.txt /training/playArea/wordCount/
```

...

...

...

Map output records=34189

Map output bytes=205134

Combine input records=34189

Combine output records=69

Reduce input records=69

Reduce output records=69

mr.wordcount.StartsWithCountMapper_UserCounters\$Tokens

FirstCharLower=26080

FirstCharUpper=8109

Total=34189

File Input Format Counters

Bytes Read=211294


File Output Format Counters

Bytes Written=385

Custom Counters
Section



Stats on tokens that start with Upper
case vs Lower



Example Usage Of Custom Counters In Map

- ❖ We Can customize counter and group names when using enums
- ❖ Create a properties file <classname>.properties defining counter name properties
 - Inner classes are substituted by underscore
 - For example: org.com.MyMapper\$CustomEnum would be MyMapper_CustomEnum.properties
- ❖ Place properties file in the same package as the class that defines Enum
- ❖ In our case the enum was defined in
 - test.mr.project.WordCountMapper\$Tokens
- ❖ Therefore the file is to be named
 - WordCountMapper_Tokens.properties
- ❖ Define Group and Counter names:
 - ✓ CounterGroupName = Token Processed
 - ✓ Total.name=Total Tokens Processed
 - ✓ FirstCharUpper.name=Tokens start with Uppercase
 - ✓ FirstCharLower.name=Tokens start with Lowercase

Different Input And Output Formats

❖ What is input format

Specification for reading data

- Creates Input Splits
 - ✓ Breaks up work into chunks
- Specifies how to read each split
 - ✓ Divides splits into records
 - ✓ Provides an implementation of RecordReader

```
public abstract class InputFormat<K, V>
{
    public abstract List<InputSplit> getSplits(JobContext context)
    throws IOException, InterruptedException;
    public abstract RecordReader<K,V> createRecordReader(InputSplit split,
    TaskAttemptContext context )
    throws IOException, InterruptedException;
}
```

Input Split

- ❖ Splits are a set of logically arranged records
 - ✓ A set of lines in a file
 - ✓ A set of rows in a database table
- ❖ Each instance of mapper will process a single split
 - ✓ Map instance processes one record at a time
 - `map(k,v)` is called for each record
- ❖ Splits are implemented by extending `InputSplit` class
- ❖ Framework provides many options for `InputSplit` implementations
 - ✓ Hadoop's `FileSplit`
 - ✓ HBase's `TableSplit`
- ❖ Don't usually need to deal with splits directly
 - ✓ `InputFormat`'s responsibility



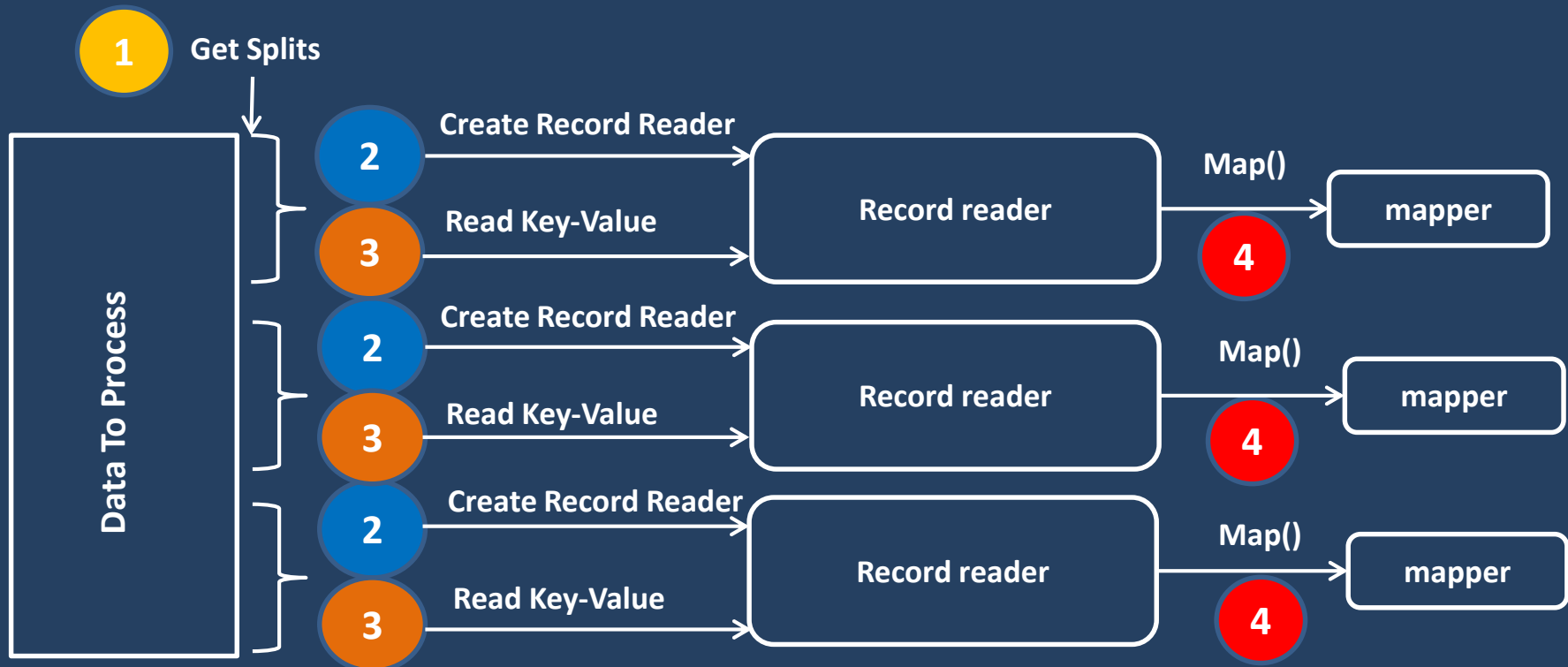
Use Of Input Formats In MR

1 Generate Splits

2 Each Split gets its own RecordReader

3 RecordReader reads key-value pairs

4 For each pair map(key, value) is called



Different Types Of Input Formats

❖ Hadoop eco-system is packaged with many Input Formats

- ✓ TextInputFormat
- ✓ KeyValueTextInputFormat
- ✓ NLineInputFormat
- ✓ DBInputFormat
- ✓ TableInputFormat (HBASE)
- ✓ HCatInputFormat
- ✓ SequenceFileInputFormat etc...

❖ Configure on a job object

- ✓ `job.setInputFormatClass(XXXInputFormat.class)`

Text Input Format

- ❖ Plain Text Input
- ❖ Default Format

Split:	Single HDFS Block(Can be Configured)
Record:	Single line of text; linefeed or carriage-return used to locate end of line.
Key:	LongWritable-Position in the file.
Value:	Text – Line of text.

N – Line Input Format

- ❖ Same as TextInputFormat but splits equal to configured N lines

Split:	N Lines; Configured via <code>mapred.line.input.format</code> or <code>NLineInputFormat.SetNumLinesPerSplit(job, 100)</code>
Record:	Single line of text;
Key:	LongWritable-Position in the file.
Value:	Text – Line of text.

Table Input Format

- ❖ Converts Data in HTable to format consumable to MapReduce.
- ❖ Mapper Must accept proper key/values

Split:	Rows in one HBase Region (Provided Scan may Narrow down the result)
Record:	Row, returned columns are controlled by a provided scan
Key:	ImmutableBytesWritable
Value:	Result(HBase class)

Output Format

- ❖ Specification for writing data
 - ✓ The other side of InputFormat
- ❖ Implementation of OutputFormat<K,V>
- ❖ TextOutputFormat is the default implementation
 - ✓ output records as lines of text
 - ✓ Key and Values are tab separated by “Key \t Values”
 - can be configured via
“**mapreduce.output.textoutputformat.Separator**” Property
 - ✓ Key and Values may of any type – call .toString()



Different Types Of Output Format

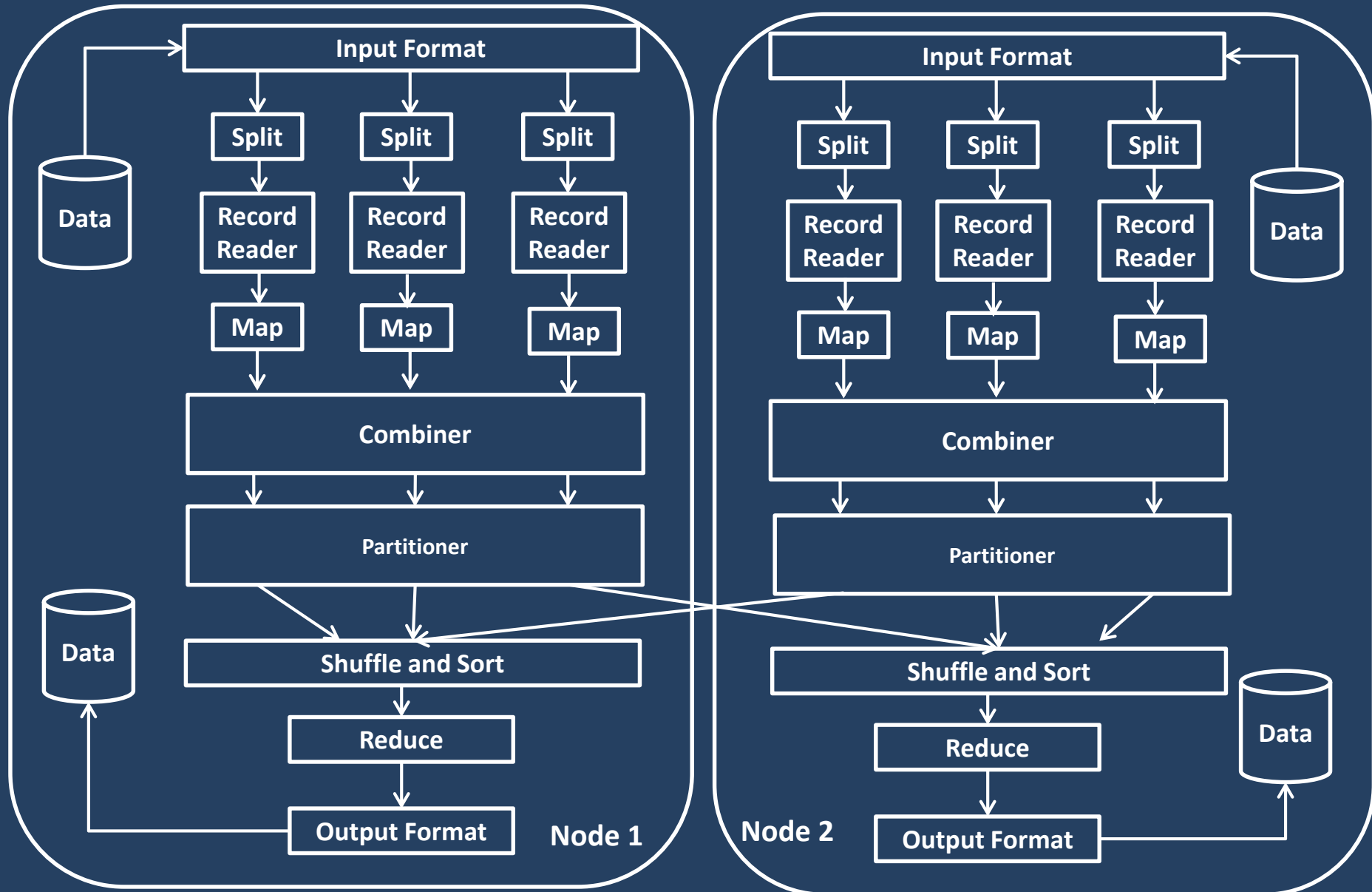
- ❖ Hadoop eco-system is packaged with many Input Formats
 - ✓ TextOutputFormat
 - ✓ DBOutputFormat
 - ✓ TableOutputFormat (HBASE)
 - ✓ NullOutputFormat
 - ✓ SequenceFileOutputFormat etc...
- ❖ Configure on a job object
 - ✓ `job.setOutputFormatClass(XXXOutputFormat.class)`
 - ✓ `job.setOutputKeyClass(XXXKey.class)`
 - ✓ `job.setOutputValueClass(XXXValue.class)`

Text Output Format

- ❖ Output plain text
- ❖ Saves key-value pairs separated by tab
 - ✓ Configured Via
`mapreduce.output.textoutputformat.seperator` property
- ❖ Set Output Path
 - ✓ `TextOutputFormat.setOutputPath(job, new Path(myPath));`
- ❖ TableOutputFormat Saves data into HTable
- ❖ Reducer output key is ignored
- ❖ Reducer output value must be HBase's put or delete objects



Component Overview



Key And Value Types

- ❖ Keys must implement WritableComparable interface
 - ✓ Extends Writable and java.lang.Comparable<T>
 - ✓ Required because keys are sorted prior reduce phase
- ❖ Hadoop is shipped with many default implementations of WritableComparable<T>
 - ✓ Wrappers for primitives (String, Integer, etc...)
 - ✓ Or you can implement your own



Custom Data Types

- ❖ MapReduce API provides feasibility to write our own data type.
- ❖ To write our custom data type below rules are to be followed
- ❖ A custom Hadoop writable data type which needs to be used as field in MapReduce programs must implement Writable interface **org.apache.hadoop.io.Writable**
- ❖ MapReduce key types should have the ability to compare against each other for sorting purposes. A custom hadoop writable data type that can be used as key field in MapReduce programs must implement WritableComparable interface which intern extends Writable (org.apache.hadoop.io.Writable) and Comparable (java.lang.Comparable) interfaces. So, i.e. a data type created by implementing **WritableComparable** Interface can be used as either key or value field data type
- ❖ <http://hadooptutorial.info/hadoop-data-types/>
- ❖ <http://hadooptutorial.info/creating-custom-hadoop-writable-data-type/>

Implement Custom WritableComparable<T>

- ❖ We need to Implement 3 methods for custom datatype
 - ✓ write(DataOutput)
 - Serialize your attributes
 - ✓ readFields(DataInput)
 - De-Serialize your attributes
 - ✓ compareTo(T)
 - Identify how to order your objects
 - If your custom object is used as the key it will be sorted prior to reduce phase



BlogWritable — Implementation Of WritableComparable<T>

```
Public class BlogWritable implements WritableComaparable<BlogWritable>
{
    Private string author;
    Private string content;
    Public BlogWritable()
    {
    }
    Public BlogWritable(string author, string content)
    {
        this.author = author;
        This.content = content;
    }
    Public String getAuthor()
    {
        return author;
    }
    Public String getContent()
    {
        return content;
    }
}
```

.....

.....

.....

.....

@override

Public void readFields(DataInput input) throws IOException

{

author = input.readUTF();

content = input.readUTF();

}

How the data is read

@override

Public void Write(DataOutput output) throws IOException

{

output.writeUTF(author);

output.writeUTF(content);

}

How to write data

@override

Public int compareTo(BlogWritable other)

{

Return author.compareTo(other.author)

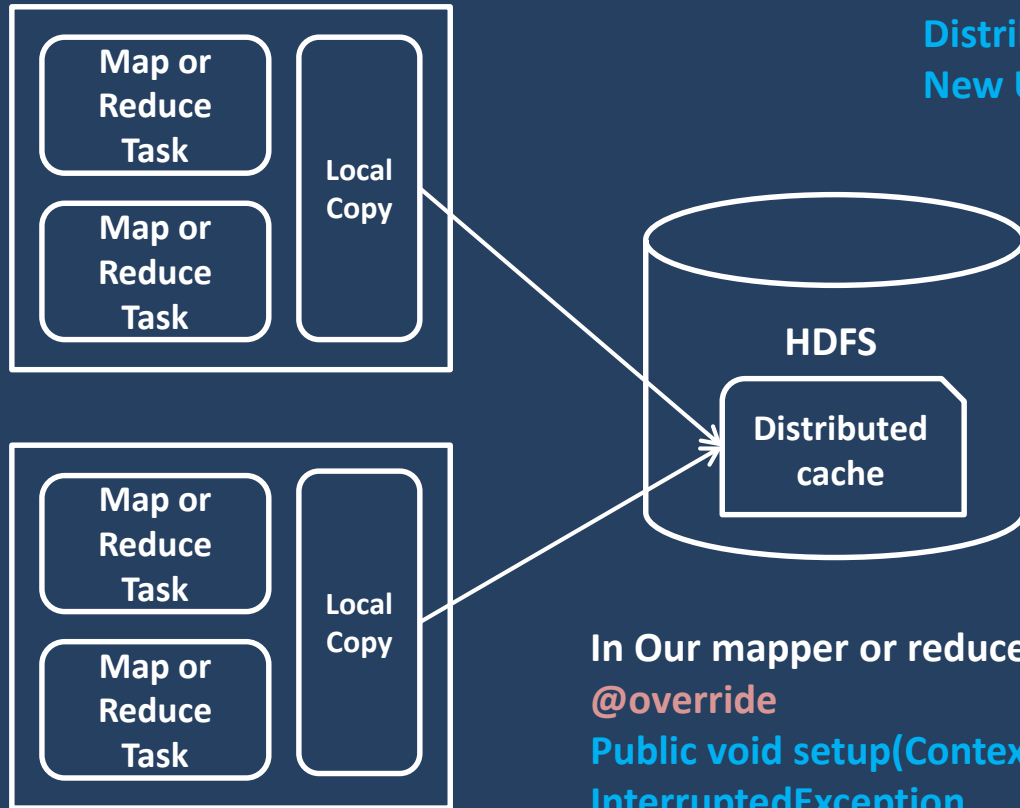
}

How to order
BlogWritables

}

Distributed Cache

We need some data and libraries on all the Nodes



In Our Driver:

```
DistributedCache.addCacheFile(  
    New URI("/Some/path/to/ourfile.txt), conf);
```

In Our mapper or reducer:

@override

```
Public void setup(Context Context) throws IOException,  
InterruptedException
```

```
{  
    Configuration conf = context.getConfiguration();  
    localFiles = DistributedCache.getLocalCacheFiles(Conf);  
}
```

Distributed Cache Working

- ❖ Accepts two types: files and archives
 - ✓ Archives are unarchived on the local node
- ❖ Items specified to the \$yarn command via `-files`, `-libjars` and `-archives` are copied to HDFS
- ❖ Supports
 - Simple text files
 - Jars
 - Archives: zip, tar, tgz/tar.gz
- ❖ Prior to task execution these files are copied locally from HDFS
 - ✓ Files now reside on a local disk – local cache
- ❖ Files provided to the `-libjars` are appended to task's CLASSPATH
- ❖ Locally cached files become qualified to be deleted after all tasks utilizing cache complete
- ❖ Files in the local cache are deleted after a 10GB threshold is reached
 - ✓ Allow space for new files
 - ✓ Configured via `yarn.nodemanager.localizer.cache.targetsize - mb` property
- ❖ Files to cached must be exist in HDFS
- ❖ Local Cache is stored under `${yarn.nodemanager.local-dirs}/usercache/$user/filecache`

JAVA API Distributed Cache

Adding cache file via methods on job

Public int run(String args[]) throws Exception

{

```
    Job job = job.getInstance(getConf(),getclass().getSimpleName());  
    job.SetJarByClass(getClass());
```

....

....

....

```
    path tocache = new path ("/training/data/StartWithExcludeFile.txt");  
    job.addCacheFile(toCache.toUri());  
    job.createSymlink();
```

Add File to Distributed Cache

Create Symbolic links for all files in DistributedCache; without the links you would have to use fully qualified path, in this case
"/training/data/startWithExcludeFile.txt"

```
    return job.waitForCompletion(true) ? 0 : 1;
```

}


JAVA API Distributed Cache

Retrieved File from Cache

```
public final static String EXCLUDE_FILE = "startWithExcludeFile.txt";
@Override
Protected void setup(Context context) throws IOException;
InterruptedException
{
    FileReader reader = new FileReader(new File(EXCLUDE_FILE));
    Try
    {
        BufferedReader bufferReader = new BufferedReader(reader);
        String line;
        While((line=bufferReader.readLine())!=null)
        {
            Excludeset.add(line);
            Log.info(ignoring Words that start with ["+line+"] );
        }
    }
    finally
    {
        Reader.close();
    }
}
```

JAVA API Distributed Cache

```
$ yarn jar distcacheExample.jar MainClassName \  
  -files LFS/Path/startWithExcludeFile.txt \  
  /input/path/in.txt \  
  /output/path/wordcount
```

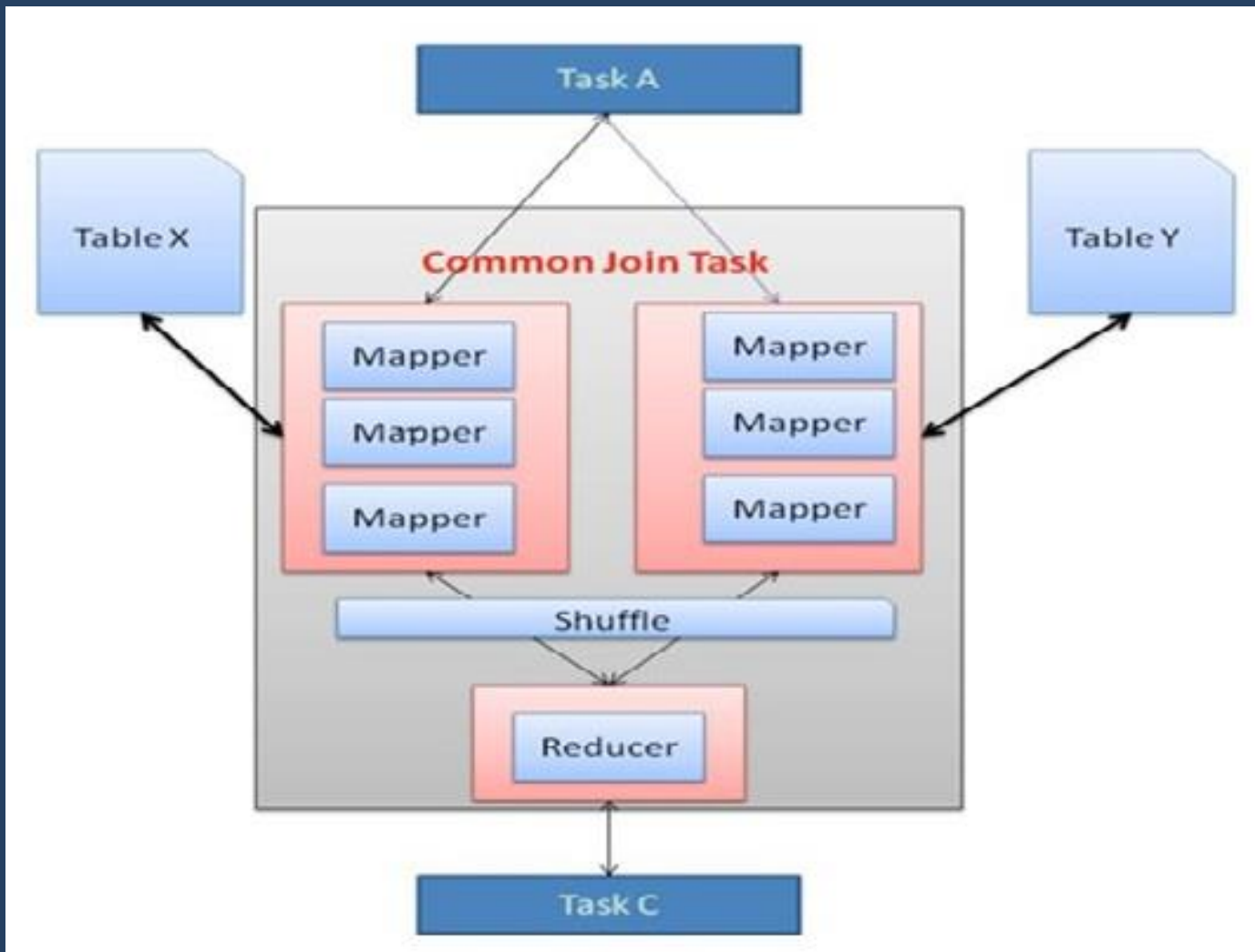


Using -files option yarn command will place the file onto DistributedCache.

- If we set the distributed cache through Java API in driver program via `job.addCacheFile("hdfs://namenode/actual/path#symbolic_link")`, the file needs to be present in HDFS
- But if we use yarn jar command's `-files` option then our files can be reside on LFS also



JOINS



Joins

❖ Two Types

- ❑ Map Side Joins
- ❑ Reduce Side Joins

❖ *Advantages of using Map-side join:*

- ❑ Reducing the expenses that we use for sorting and merging in the shuffle and reduce stages.
- ❑ Optimizes the performance of the task i.e. reducing the time to complete the task.

❖ *Disadvantages of Map-side join:*

- ❑ Map side joins will be suitable when one of the tables on which you perform map-side join operation should be small enough to fit into memory.

Miscellaneous Concept In MR

- ❖ **Record Reader** : The InputSplit has defined a slice of work, but does not describe how to access it. The RecordReader class actually loads the data from its source and converts it into (key, value) pairs suitable for reading by the Mapper. The RecordReader instance is defined by the Input Format.
- ❖ **Shuffle**: After the first map tasks have completed, the nodes may still be performing several more map tasks each. But they also begin exchanging the intermediate outputs from the map tasks to where they are required by the reducers. This process of moving map outputs to the reducers is known as shuffling.
- ❖ **Sort**: Each reduce task is responsible for reducing the values associated with several intermediate keys. The set of intermediate keys on a single node is automatically sorted by Hadoop before they are presented to the Reducer.
- ❖ **Hadoop Streaming**: Streaming is a generic API that allows programs written in virtually any language to be used as Hadoop Mapper and Reducer implementations. Languages supported in Hadoop streaming are Perl, Ruby, Python, C, C#.

USEFULL REFERENCE

- ❖ <http://hadoop.apache.org/docs/r2.4.1/>
- ❖ <http://hadooptutorial.info/>



Questions Or Doubts



“Your Career is your **Business**
Its time for you to manage it as
a **CEO**”

Siva Kumar Bhuchipalli
A/O

www.hadooptutorial.info/
Hyderabad

THANK YOU

