# Advanced MapReduce

Developing your MapReduce Job

# MapReduce

- **Job – execution of map and reduce**
  - functions to accomplish a task
    - Equal to Java's main

- **Task – single Mapper or Reducer**
  - Performs work on a fragment of data

# WordCount Job

1. **Configure the Job**
   - Specify Input, Output, Mapper, Reducer and Combiner

2. **Implement Mapper**
   - Input is text – a line from sample.txt
   - Tokenize the text and emit first character with a count of 1 - <token, 1>

3. **Implement Reducer**
   - Sum up counts for each letter
   - Write out the result to HDFS

4. **Run the job**

# 1. Configure Job

- **Job class**
  - Encapsulates information about a job
  - Controls execution of the job

```
Job job = new Job();
```

- **A job is packaged within a jar file**
  - Hadoop Framework distributes the jar on your behalf
  - Needs to know which jar file to distribute
  - The easiest way to specify the jar that your job resides in is by calling job.setJarByClass

```
job.setJarByClass(WordCount.class);
```

  - Hadoop will locate the jar file that contains the provided class

# 1. Configure Job – Specify Input

```
FileInputFormat.addInputPath(job, new
Path(otherArgs[0]));
job.setInputFormatClass(TextInputFormat.class);
```

- Can be a file, directory or a file pattern
  - Directory is converted to a list of files as an input
- Input is specified by implementation of InputFormat - in this case TextInputFormat
  - Responsible for creating splits and a record reader
  - Controls input types of key-value pairs, in this case LongWritable and Text
- File is broken into lines, mapper will receive 1 line at a time

# 1. Configure Job – Specify Output

```
FileOutputFormat.setOutputPath(job, new
Path(otherArgs[1]));
job.setOutputFormatClass(TextOutputFormat.class);
```

- OutputFormat defines specification for outputting data from Map/Reduce job

- **WodCount job utilizes an implementation of OutputFormat** - TextOutputFormat
  - Define output path where reducer should place its output
    - If path already exists then the job will fail
  - Each reducer task writes to its own file
    - By default a job is configured to run with a single reducer
  - Writes key-value pair as plain text

# 1. Configure Job – Specify Output

```
job.setOutputKeyClass(Text.class);
job.setOutputValueClass(IntWritable.class);
```

- Specify the output key and value types for both mapper and reducer functions
  - Many times the same type
  - If types differ then use
    - setMapOutputKeyClass()
    - setMapOutputValueClass()

# 1. Configure Job

- **Specify Mapper, Reducer and Combiner**
  - At a minimum will need to implement these classes
  - Mappers and Reducer usually have same output key

```
job.setMapperClass(TokenizerMapper.class);
job.setReducerClass(IntSumReducer.class);
job.setCombinerClass(IntSumReducer.class);
```

# 1. Configure Job

- **job.waitForCompletion(true)**
  - Submits and waits for completion
  - The boolean parameter flag specifies whether output should be written to console
  - If the job completes successfully ‘true’ is returned, otherwise ‘false’ is returned

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

# Our Count Job is configured to

- Chop up text files into lines
- Send records to mappers as key-value pairs
  - Line number and the actual value
- **Mapper class is TokenizeMapper**
  - Receives key-value of <IntWritable,Text>
  - Outputs key-value of <Text, IntWritable>
- **Reducer class is IntSumReducer**
  - Receives key-value of <Text, IntWritable>
  - Outputs key-values of <Text, IntWritable> as text
- Combiner class is IntSumReducer

# 1. Configure Count Job

```
public class WordCount {
public static void main(String[] args) throws Exception {

try{
   if (args.length != 2) {
      System.out.printf("Usage: wordcount <input dir> <output dir>\n");
      System.exit(-1);
     }

Job job = new Job();
job.setJarByClass(WordCount.class);

job.setOutputKeyClass(Text.class);
job.setOutputValueClass(IntWritable.class);

job.setMapperClass(TokenizeMapper.class);
job.setReducerClass(TokenizeReducer.class);

job.setOutputKeyClass(Text.class);
job.setOutputValueClass(IntWritable.class);

FileInputFormat.addInputPath(job, new Path(args[0]));
FileOutputFormat.setOutputPath(job, new Path(args[1]));

System.exit(job.waitForCompletion(true) ? 0 : 1);
}catch (Exception e){
e.printStackTrace();
}
}
```

# 2. Implement Mapper Class

- **Class has 4 Java Generics parameters**
  - (1) input key (2) input value (3) output key (4) output value
  - Input and output utilizes hadoop's IO framework
    - org.apache.hadoop.io

- **Your job is to implement map() method**
  - Input key and value
  - Output key and value
  - Logic is up to you

- **map() method injects Context object, use to:**
  - Write output
  - Create your own counters

# 2. Implement Maper

```
public class TokenizerMapper extends Mapper<Object, Text, Text,
   IntWritable> {

@Override
public void map(Object key, Text value, Context context)
throws IOException, InterruptedException {

StringTokenizer itr = new StringTokenizer(value.toString());

while (itr.hasMoreTokens()) {
context.write(new Text(itr.nextToken()), new IntWritable(1));
}
}
}
```

# 3. Implement Reducer

- **Analogous to Mapper – generic class with four types**
  - (1) input key (2) input value (3) output key (4) output value
  - The output types of map functions must match the input types of reduce function
    - In this case Text and IntWritable
  - **Map/Reduce framework groups key-value pairs produced by mapper by key**
    - For each key there is a set of one or more values
    - Input into a reducer is sorted by key
    - Known as Shuffle and Sort
  - Reduce function accepts key->setOfValues and outputs key-value pairs
    - Also utilizes Context object (similar to Mapper)

# 3. Implement Reducer

```
public class IntSumReducer
extends Reducer<Text, IntWritable, Text, IntWritable>{

@Override
public void reduce (Text key, Iterable<IntWritable> values, Context
   context)
throws IOException, InterruptedException {
int sum = 0;
for (IntWritable val : values)
{
sum += val.get();
}

context.write(key, new IntWritable(sum));
}
}
```

# 3. Reducer as a Combiner

- Combine data per Mapper task to reduce amount of data transferred to reduce phase
- **Reducer can very often serve as a combiner**
  - Only works if reducer's output key-value pair types are the same as mapper's output types
- Combiners are not guaranteed to run
  - Optimization only
  - Not for critical logic

# 4. Run Count Job

- **DEMO** -- Specify how to run

- Create a JAR class
- Run the jar

```
hadoop jar wordcount.jar <in> <out>
```

# Output From Your Job

- **Provides job id**
  - Used to identify, monitor and manage the job
- **Shows number of generated splits**
- **Reports the Progress**
- **Displays Counters** – statistics for the job
  - Sanity check that the numbers match what you expected

# Input and Output

# Hadoop IO Classes

- **Hadoop uses it's own serialization mechanism** for writing data in and out of network, database or files
  - Optimized for network serialization
  - A set of basic types is provided
  - Easy to implement your own

- **Extends Writable** Interface
  - Framework's serialization mechanism
  - Defines how to read and write fields

- **org.apache.hadoop.io** package
  - LongWritable for Long
  - IntWritable for Integer
  - Text for String
  - Etc...

# 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

# WritableComparable<T> Implementations

| Hadoop's Class | Explanation |
|---|---|
| BooleanWritable | Boolean implementation |
| BytesWritable | Bytes implementation |
| DoubleWritable | Double implementation |
| FloatWritable | Float implementation |
| IntWritable | Int implementation |
| LongWritable | Long implementation |
| NullWritable | Writable with no data |

# Hadoop's Input Format

- **Hadoop eco-system is packaged with many InputFormats**
  - TextInputFormat
  - NLineInputFormat
  - DBInputFormat
  - TableInputFormat (HBASE)
  - StreamInputFormat
  - SequenceFileInputFormat
  - Etc...

- **Configure on a Job object**
  - job.setInputFormatClass(XXXInputFormat.class);

# TextInput Format

- Plaint 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 |

# TableInput 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) |

# HashPartitioner

- **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

# OutputFormat

- **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 “Key /t value”
    - Can be configured via “mapreduce.output.textoutputformat.separator” property
  - Key and Value may of any type - call .toString()

# Hadoop's Output Format

- **Hadoop eco-system is packaged with many OutputFormats**
  - TextOutputFormat
  - DBOutputFormat
  - TableOutputFormat (HBASE)
  - MapFileOutputFormat
  - SequenceFileOutputFormat
  - NullOutputFormat
  - Etc...
- **Configure on Job object**
  - job.setOutputFormatClass(XXXOutputFormat.class);
  - job.setOutputKeyClass(XXXKey.class);
  - job.setOutputValueClass(XXXValue.class);

# MRv2 / YARN

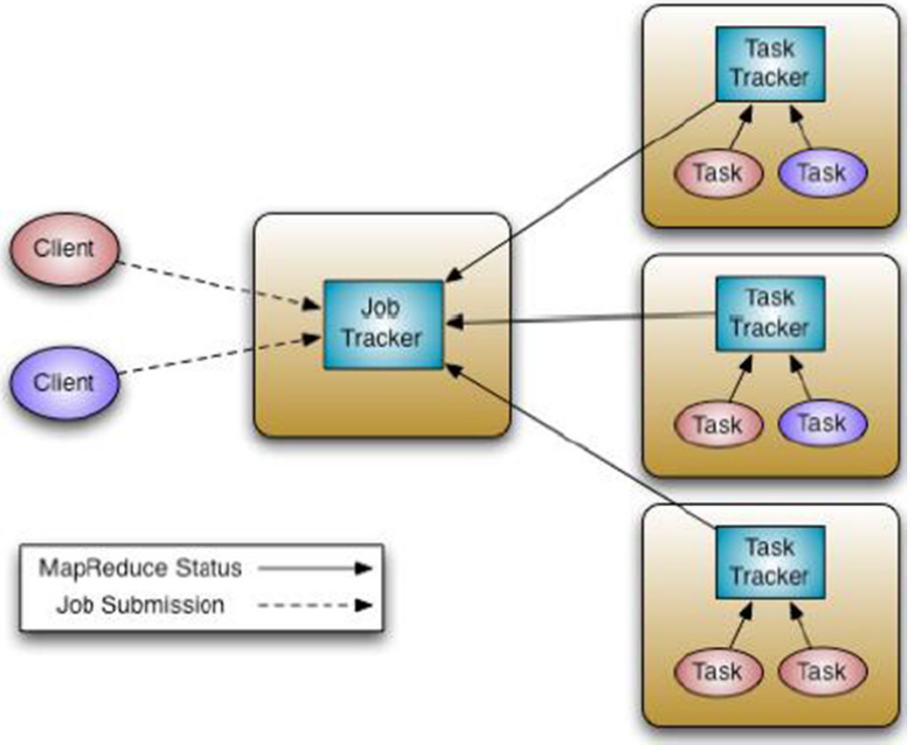
The future of next-gen computing

# YARN

## Yet Another Resource Negotiator

- YARN: a generic resource-management and distributed application framework
  - In Aug 2012, YARN was promoted to be a sub-project of Hadoop in Apache. Before this, YARN was part of the Hadoop MapReduce project.
  - MR is not sufficient for all use cases, like PageRank or many ML algorithms
  - MR become just one of the applications that can be run in YARN
  - Future YARN algorithms: MPI/Iterative processing, graph-processing, simple services, real time (stream processing, CEPFresil)
  - Since all the data in the enterprise is already available in Hadoop HDFS having multiple paths for processing is critical
  - All client-facing MapReduce interfaces are unchanged, which means that there is no need to make any source code changes to run on top of Hadoop 0.23

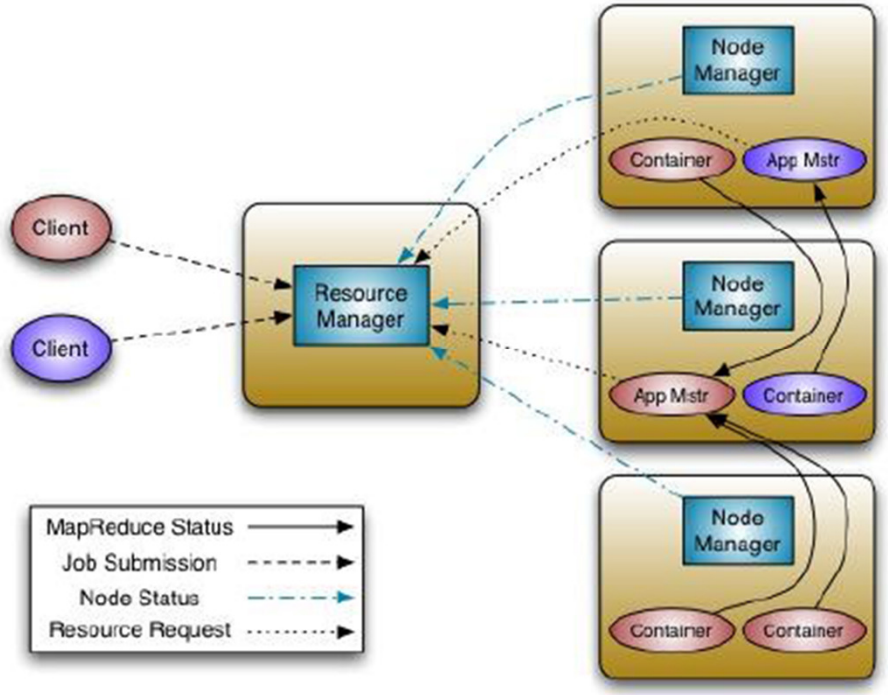
# MRv1 Quick Recap




**JT responsibilities:**

- Resource management (TTs),
- Tracking resource consumption/availability,
- Job life-cycle management

# MRv2 Overview




**Fundamental idea:**

Re-architect JT's Resource Management and Job scheduling & monitoring into two separate components: Resource Manager & AppMaster

# Building Blocks

- **ResourceManager:** manages the global assignment of compute resources to applications, has a pluggable scheduler for allocating resources to the running applications subject constraints of capacities, queues. It optimizes for cluster utilization (keep all resources in use all the time) against various constraints such as capacity guarantees, fairness, and SLAs, does NOT do fault tolerance for resources (AM does)

- **ApplicationMaster:** manages the application's scheduling and coordination, negotiates appropriate resource containers from the Scheduler and tracking their progress

- **NodeManager:** per machine slave, responsible for launching applications' containers, monitoring their resources (cpu, memory, disk) and reporting to the ResourceManager

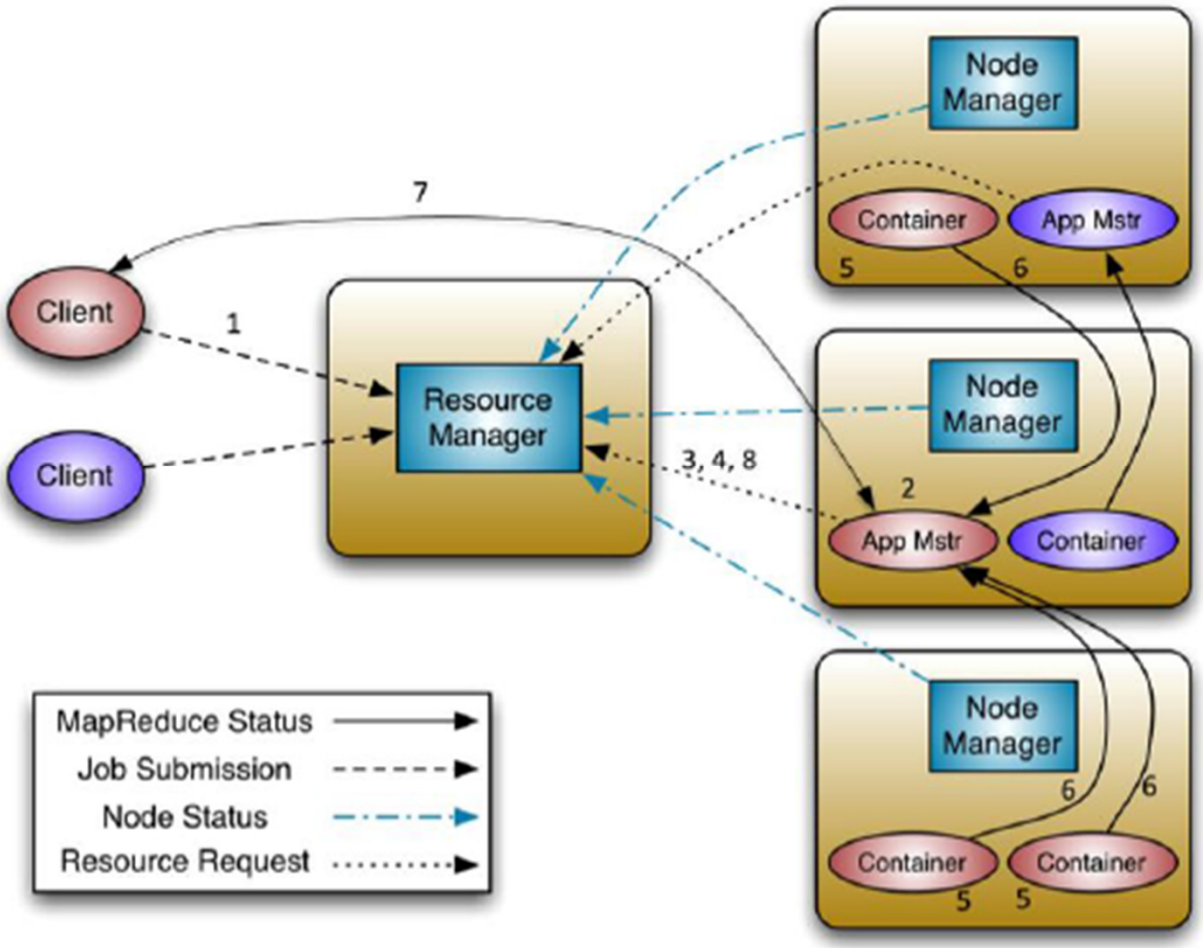
- **The AM can request very specific requirements from the RM for the containers, like:**
  - Resource name, hostname, rack name,
  - Memory (in MB)
  - CPU (in cores), added after March 2012
  - Future: disk, network, GPUs, etc

# Building Blocks

- A resource request from the AM to the scheduler in the RM has the following:
  - Resource name: hostname, rackname, (Future: VMs on a host, networks)
  - Priority: priority within the application, not across cluster
  - Resource requirement: memory, CPU (F: GPUs)
  - # of Containers: just a #

- A container is basically a resource allocation that grants rights to an application to use a specific amount of resources (memory, CPU) on a specific host

# Application Execution Sequence



# Application Execution Sequence

1) A client program submits the application, including the necessary specifications to launch the application-specific ApplicationMaster itself.

2) The ResourceManager assumes the responsibility to negotiate a specified container in which to start the ApplicationMaster and then launches the ApplicationMaster.

3) The ApplicationMaster, on boot-up, registers with the ResourceManager – the registration allows the client program to query the ResourceManager for details, which allow it to directly communicate with its own ApplicationMaster.

4) During normal operation the ApplicationMaster negotiates appropriate resource containers via the resource-request protocol.

5) On successful container allocations, the ApplicationMaster launches the container by providing the container launch specification to the NodeManager. The launch specification, typically, includes the necessary information to allow the container to communicate with the ApplicationMaster itself.

6) The application code executing within the container then provides necessary information (progress, status etc.) to its ApplicationMaster via an application-specific protocol.

7) During the application execution, the client that submitted the program communicates directly with the ApplicationMaster to get status, progress updates etc. via an application-specific protocol.

8) Once the application is complete, and all necessary work has been finished, the ApplicationMaster deregisters with the ResourceManager and shuts down, allowing its own container to be repurposed.

***Thank You***