Architecture of Enterprise Applications 24
Hadoop

Haopeng Chen

***RE**liable, **IN**telligent and **S**calable Systems Group (**REINS**)*
Shanghai Jiao Tong University
Shanghai, China
http://reins.se.sjtu.edu.cn/~chenhp
e-mail: chen-hp@sjtu.edu.cn

Contents and Objectives

- Hadoop
  - Basic Concepts
  - MapReduce
  - YARN

- Objectives
  - 能够针对大数据批处理需求，设计并实现基于 MapReduce/YARN 的并行处理方案

Apache Hadoop

- The Apache™ Hadoop® project
  - develops open-source software for reliable, scalable, distributed computing.
- The Apache Hadoop software library
  - is a framework that allows for the distributed processing of large data sets across clusters of computers using simple programming models.
  - It is designed to scale up from single servers to thousands of machines, each offering local computation and storage.
  - Rather than rely on hardware to deliver high-availability, the library itself is designed to detect and handle failures at the application layer, so delivering a highly-available service on top of a cluster of computers, each of which may be prone to failures.

  - https://hadoop.apache.org

Apache Hadoop - Modules

- The project includes these modules:
  - **Hadoop Common**: The common utilities that support the other Hadoop modules.
  - **Hadoop Distributed File System (HDFS™)**: A distributed file system that provides high-throughput access to application data.
  - **Hadoop YARN**: A framework for job scheduling and cluster resource management.
  - **Hadoop MapReduce**: A YARN-based system for parallel processing of large data sets.
  - **Hadoop Ozone**: An object store for Hadoop.

Pseudo-Distributed Operation

- etc/hadoop/core-site.xml:

```
<configuration>
    <property>
        <name>fs.defaultFS</name>
        <value>hdfs://localhost:9000</value>
    </property>
    <property>
        <name>hadoop.tmp.dir</name>
        <value>/var/hadoop</value>
     </property>
</configuration>
```

- etc/hadoop/hdfs-site.xml:

```
<configuration>
    <property>
        <name>dfs.replication</name>
        <value>1</value>
    </property>
</configuration>
```

Setup passphraseless ssh

- Now check that you can ssh to the localhost without a passphrase:
- $ ssh localhost

- If you cannot ssh to localhost without a passphrase, execute the following commands:
- $ ssh-keygen -t rsa -P '' -f ~/.ssh/id_rsa
- $ cat ~/.ssh/id_rsa.pub >> ~/.ssh/authorized_keys
- $ chmod 0600 ~/.ssh/authorized_keys

- Setup dfs directories
- $ sudo mkdir /var/hadoop
- $ sudo mkdir /var/hadoop/dfs
- $ sudo mkdir /var/hadoop/dfs/name
- $ sudo chmod -R a+w /var/hadoop

Execution

REliable, INtelligent & Scalable Systems

- Format the filesystem:
- $ bin/hdfs namenode -format

- Start NameNode daemon and DataNode daemon:
- $ sbin/start-dfs.sh

- Browse the web interface for the NameNode; by default it is available at:
  - NameNode - http://localhost:9870/

Execution

Hadoop | Overview | Datanodes | Datanode Volume Failures | Snapshot | Startup Progress | Utilities

Overview 'localhost:9000' (active)

Started:	Sat Dec 04 14:00:39 +0800 2021
Version:	3.2.2, r7a3bc90b05f257c8ace2f76d74264906f0f7a932
Compiled:	Sun Jan 03 17:26:00 +0800 2021 by hexiaoqiao from branch-3.2.2
Cluster ID:	CID-3046ebbb-8b04-485e-aa0f-9061e6d6087f
Block Pool ID:	BP-1664417573-127.0.0.1-1638597623149

Summary

Security is off.

Safemode is off.

1 files and directories, 0 blocks (0 replicated blocks, 0 erasure coded block groups) = 1 total filesystem object(s).

Heap Memory used 105.33 MB of 161 MB Heap Memory. Max Heap Memory is 4 GB.

Non Heap Memory used 47.82 MB of 52 MB Commited Non Heap Memory. Max Non Heap Memory is <unbounded>.

Configured Capacity:	931.55 GB
Configured Remote Capacity:	0 B
DFS Used:	4 KB (0%)

Map Reduce

RElIable, INtelligent & Scalable Systems

- OSDI'04
  - MapReduce: Simplified Data Processing on Large Clusters

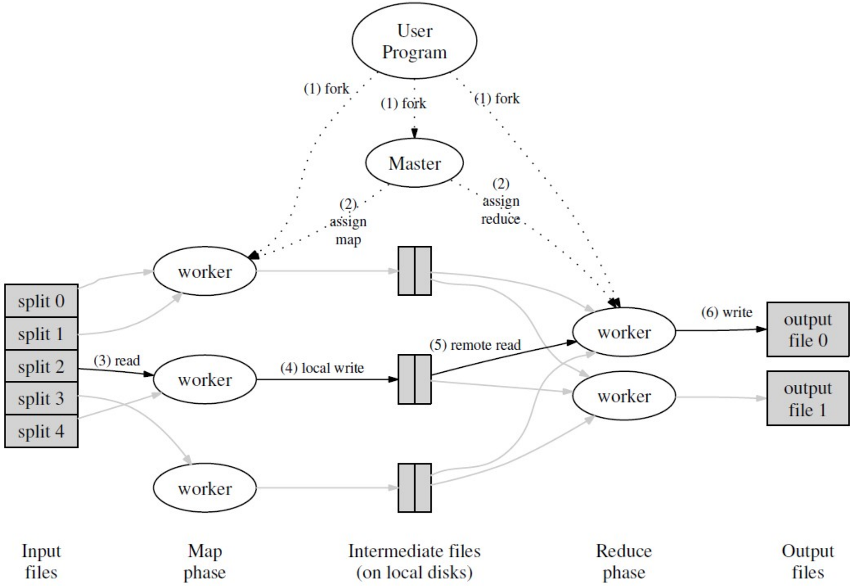


Figure 1: Execution overview

Map Reduce

- Pseudo-code
  - Word Count Example

```
map(String key, String value):
  // key: document name
  // value: document contents
  for each word w in value:
    EmitIntermediate(w, "1");

reduce(String key, Iterator values):
  // key: a word
  // values: a list of counts
  int result = 0;
  for each v in values:
    result += ParseInt(v);
  Emit(AsString(result));
```

Map Reduce

REliable, INtelligent & Scalable Systems

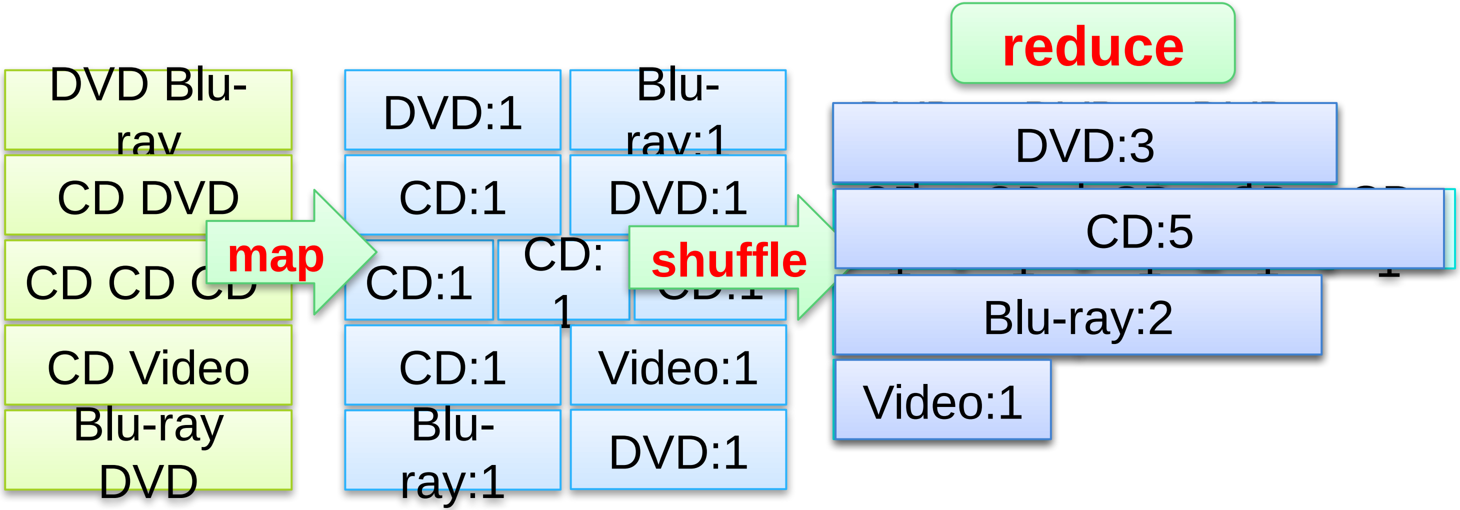
- The MapReduce framework
  - operates exclusively on <key, value> pairs, that is, the framework views the input to the job as a set of <key, value> pairs and produces a set of <key, value> pairs as the output of the job, conceivably of different types.

- The key and value classes
  - have to be serializable by the framework and hence need to implement the Writable interface. Additionally, the key classes have to implement the WritableComparable interface to facilitate sorting by the framework.

- Input and Output types of a MapReduce job:
  - (input) <k1, v1> -> **map** -> <k2, v2> -> **combine** -> <k2, v2> -> **reduce** -> <k3, v3> (output)

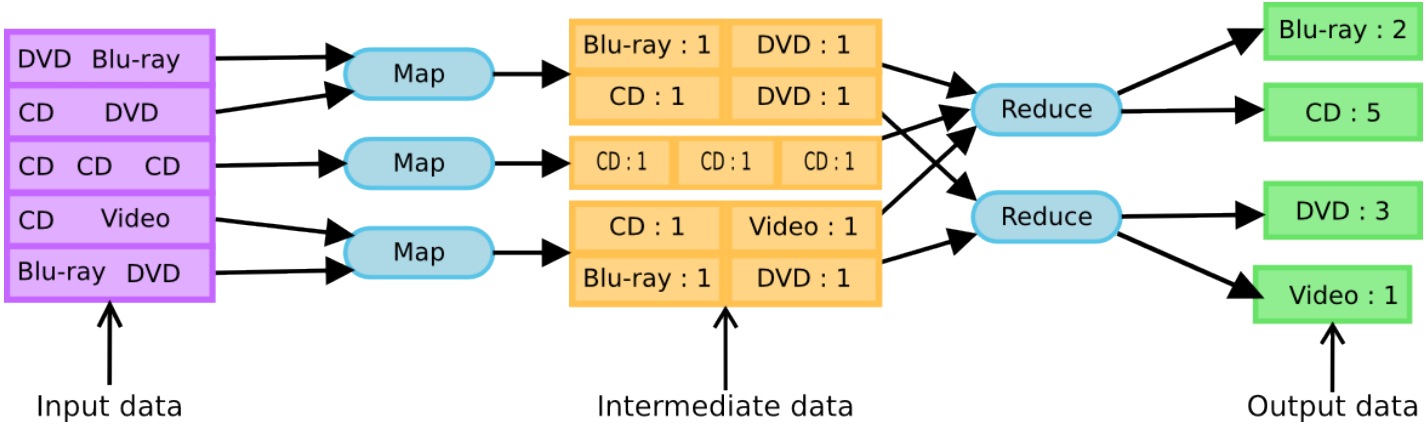
MapReduce Basics

- Paradigm
  - –
  - –
- Word Count



MapReduce Basics

- Execution View



Map Reduce - WordCount


```
public class WordCount {

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

        private final static IntWritable one = new IntWritable(1);
        private Text word = new Text();

        public void map(Object key, Text value, Context context) throws IOException,
InterruptedException {
            StringTokenizer itr = new StringTokenizer(value.toString());
            while (itr.hasMoreTokens()) {
                word.set(itr.nextToken());
                context.write(word, one);
            }
        }
    }
```

Map Reduce - WordCount

REliable, INtelligent & Scalable Systems

```
    public static class IntSumReducer extends Reducer<Text,IntWritable,Text,IntWritable> {
        private IntWritable result = new IntWritable();

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

Map Reduce - WordCount

REliable, INtelligent & Scalable Systems

```java
    public static void main(String[] args) throws Exception {
        Configuration conf = new Configuration();
        conf.set("dfs.defaultFS", "hdfs://hadoop:9000");
        Job job = Job.getInstance(conf, "word count");
        job.setJarByClass(WordCount.class);
        job.setMapperClass(TokenizerMapper.class);
        job.setCombinerClass(IntSumReducer.class);
        job.setReducerClass(IntSumReducer.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);
    }
}
```

Map Reduce - WordCount

REliable, INtelligent & Scalable Systems

```xml
    <dependencies>
        <dependency>
            <groupId>org.apache.hadoop</groupId>
            <artifactId>hadoop-common</artifactId>
            <version>3.2.1</version>
        </dependency>
        <dependency>
            <groupId>org.apache.hadoop</groupId>
            <artifactId>hadoop-mapreduce-client-core</artifactId>
            <version>3.2.1</version>
        </dependency>
        <dependency>
            <groupId>org.apache.hadoop</groupId>
            <artifactId>hadoop-mapreduce-client-common</artifactId>
            <version>3.2.1</version>
        </dependency>
    </dependencies>
</project>
```

Map Reduce - WordCount

- file01

  Hello World Bye World

- file01

  Hello Hadoop Bye Hadoop

- part-r-0000001

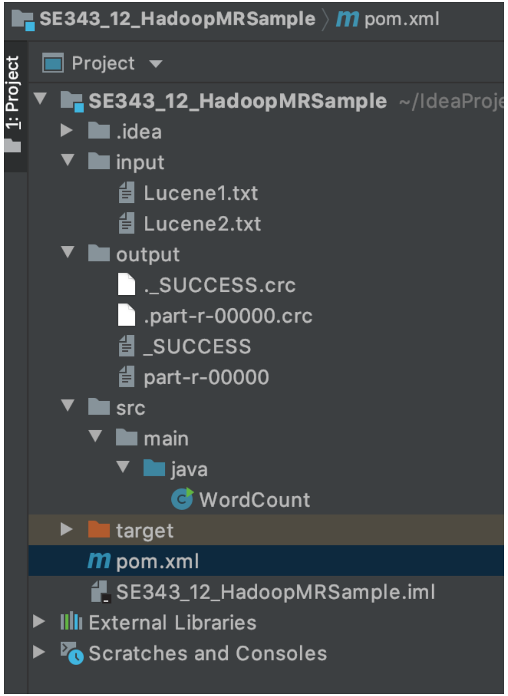
  Bye 1

  Goodbye 1

  Hadoop 2

  Hello 2

  World 2



Map Reduce - WordCount

REliable, INtelligent & Scalable Systems

```
public void map(Object key, Text value, Context context) throws IOException,
InterruptedException {
    StringTokenizer itr = new StringTokenizer(value.toString());
    while (itr.hasMoreTokens()) {
        word.set(itr.nextToken());
        context.write(word, one);
    }
}
```

- For the given sample input the first map emits:

  < Hello, 1>

  < World, 1>

  < Bye, 1>

  < World, 1>

- The second map emits:

  < Hello, 1>

  < Hadoop, 1>

  < Goodbye, 1>

  < Hadoop, 1>

Map Reduce - WordCount


```
job.setCombinerClass(IntSumReducer.class);
```

- The output of the first map:

  < Bye, 1>

  < Hello, 1>

  < World, 2>

- The output of the second map:

  < Goodbye, 1>

  < Hadoop, 2>

  < Hello, 1>

Map Reduce - WordCount

REliable, INtelligent & Scalable Systems

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

- Thus the output of the job is:

  < Bye, 1>

  < Goodbye, 1>

  < Hadoop, 2>

  < Hello, 2>

  < World, 2>

MapReduce

REliable, INtelligent & Scalable Systems

- A Weather Dataset
  - The data we will use is from the National Climatic Data Center (NCDC, http://www.ncdc.noaa.gov/). The data is stored using a line-oriented ASCII format, in which each line is a record.
  - Sample: The line has been split into multiple lines to show each field: in the real file, fields are packed into one line with no delimiters.

```
0057
332130      # USAF weather station identifier
99999       # WBAN weather station identifier
19500101  # observation date
0300        # observation time
4
+51317      # latitude (degrees x 1000)
+028783   # longitude (degrees x 1000)
FM-12
+0171       # elevation (meters)
……
```

MapReduce

- MapReduce works by breaking the processing into two phases: the map phase and the reduce phase.
  - Each phase has key-value pairs as input and output, the types of which may be chosen by the programmer.
  - The programmer also specifies two functions: the map function and the reduce function.

MapReduce

REliable, INtelligent & Scalable Systems

- The input to our map phase is the raw NCDC data.
  - We choose a text input format that gives us each line in the dataset as a text value.
  - The key is the offset of the beginning of the line from the beginning of the file.

- This map function is simple.
  - We pull out the year and the air temperature, since these are the only fields we are interested in.
  - In this case, the map function is just a data preparation phase, setting up the data in such a way that the reducer function can do its work on it: finding the maximum temperature for each year.
  - The map function is also a good place to drop bad records: here we filter out temperatures that are missing, suspect, or erroneous.

MapReduce

- To visualize the way the map works, consider the following sample lines of input data

  0067011990999991950051507004...9999999N9+00001+99999999999...
  0043011990999991950051512004...9999999N9+00221+99999999999...
  0043011990999991950051518004...9999999N9-00111+99999999999...
  0043012650999991949032412004...0500001N9+01111+99999999999...
  0043012650999991949032418004...0500001N9+00781+99999999999...

- These lines are presented to the map function as the key-value pairs:

  (0, 0067011990999991950051507004...9999999N9+00001+99999999999...)
  (106, 0043011990999991950051512004...9999999N9+00221+99999999999...)
  (212, 0043011990999991950051518004...9999999N9-00111+99999999999...)
  (318, 0043012650999991949032412004...0500001N9+01111+99999999999...)
  (424, 0043012650999991949032418004...0500001N9+00781+99999999999...)

MapReduce

- The keys are the line offsets within the file, which we ignore in our map function.
  - The map function merely extracts the year and the air temperature (indicated in bold text), and emits them as its output (the temperature values have been interpreted as integers):

(1950, 0)

(1950, 22)

(1950, −11)

(1949, 111)

(1949, 78)

- The output from the map function is processed by the MapReduce framework before being sent to the reduce function.

- This processing sorts and groups the key-value pairs by key.
  - So, continuing the example, our reduce function sees the following input:

(1949, [111, 78])

(1950, [0, 22, −11])

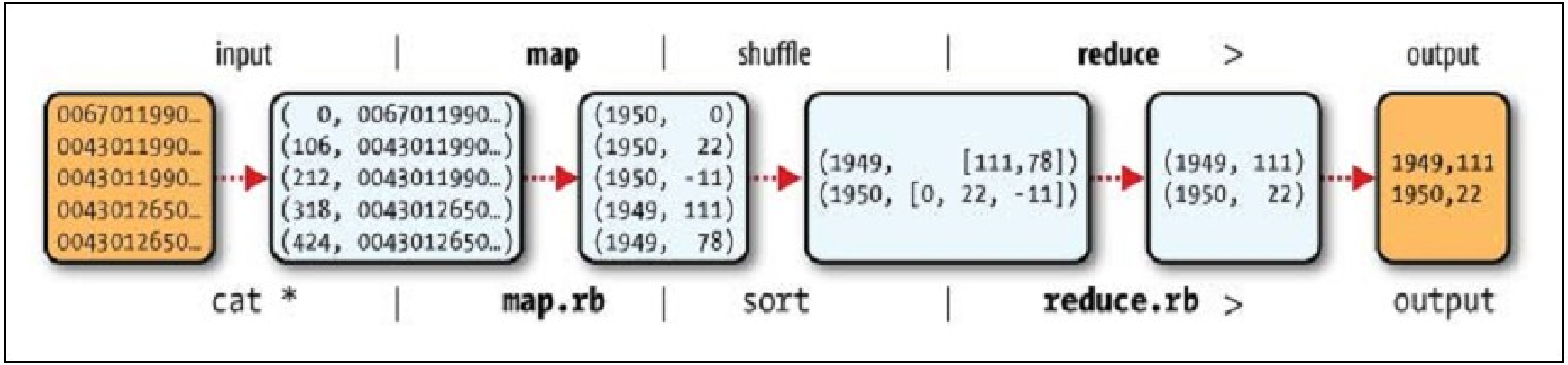
- Each year appears with a list of all its air temperature readings.

MapReduce

REliable, INtelligent & Scalable Systems

- All the reduce function has to do now is iterate through the list and pick up the maximum reading:
  (1949, 111)
  (1950, 22)

- This is the final output:
  – the maximum global temperature recorded in each year.



Java MapReduce

REliable, INtelligent & Scalable Systems

```
public class MaxTemperatureMapper extends Mapper<LongWritable, Text, Text, IntWritable> {
    private static final int MISSING = 9999;
    @Override
    public void map(LongWritable key, Text value, Context context) throws IOException,
InterruptedException {
        String line = value.toString();
        String year = line.substring(15, 19);

        int airTemperature;
        if (line.charAt(87) == '+') { // parseInt doesn't like leading plus signs
            airTemperature = Integer.parseInt(line.substring(88, 92));
        } else {
            airTemperature = Integer.parseInt(line.substring(87, 92));
        }

        String quality = line.substring(92, 93);
        if (airTemperature != MISSING && quality.matches("[01459]")) {
            context.write(new Text(year), new IntWritable(airTemperature));
        }
    }
}
```

Java MapReduce


```
public class MaxTemperatureReducer extends Reducer<Text, IntWritable, Text,
IntWritable> {
    @Override
    public void reduce(Text key, Iterable<IntWritable> values,
                       Context context)
            throws IOException, InterruptedException {
        int maxValue = Integer.MIN_VALUE;
        for (IntWritable value : values) {
            maxValue = Math.max(maxValue, value.get());
        }
        context.write(key, new IntWritable(maxValue));
    }
}
```

Java MapReduce


```java
public class MaxTemperature {
    public static void main(String[] args) throws Exception {
        if (args.length != 2) {
            System.err.println("Usage: MaxTemperature <input path> <output path>");
            System.exit(-1);
        }
        Configuration conf = new Configuration();
        conf.set("dfs.defaultFS", "hdfs://hadoop:9000");
        Job job = Job.getInstance(conf, "max temperature");
        job.setJarByClass(MaxTemperature.class);
        job.setJobName("Max temperature");
        FileInputFormat.addInputPath(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));

        job.setMapperClass(MaxTemperatureMapper.class);
        job.setReducerClass(MaxTemperatureReducer.class);

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

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

output2/part-r-00000

1901	317
1902	244

Scaling Out

- To scale out, we need to store the data in a distributed file system, typically HDFS
  - to allow Hadoop to move the MapReduce computation to each machine hosting a part of the data.

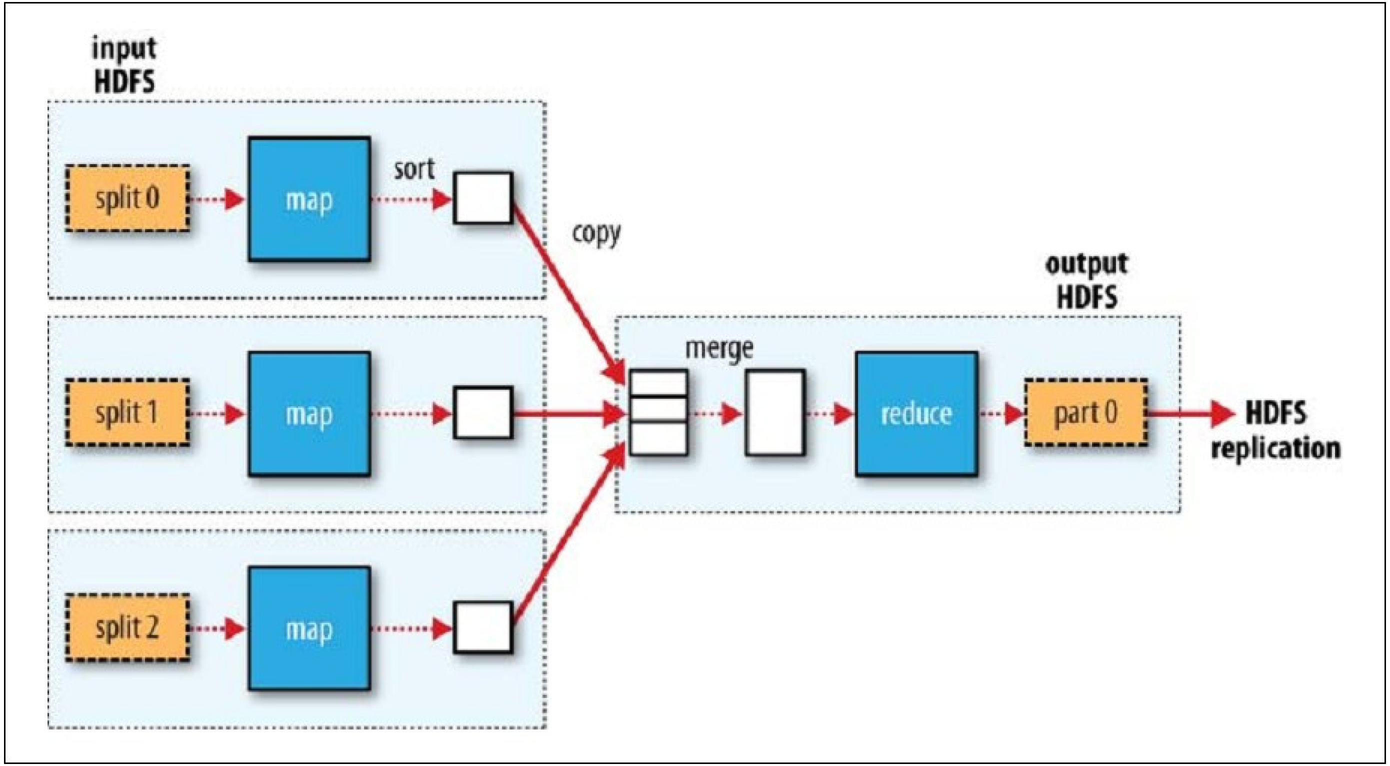
- A MapReduce job is a unit of work that the client wants to be performed:
  - it consists of the input data, the MapReduce program, and configuration information.
  - Hadoop runs the job by dividing it into tasks, of which there are two types: map tasks and reduce tasks.

Scaling Out

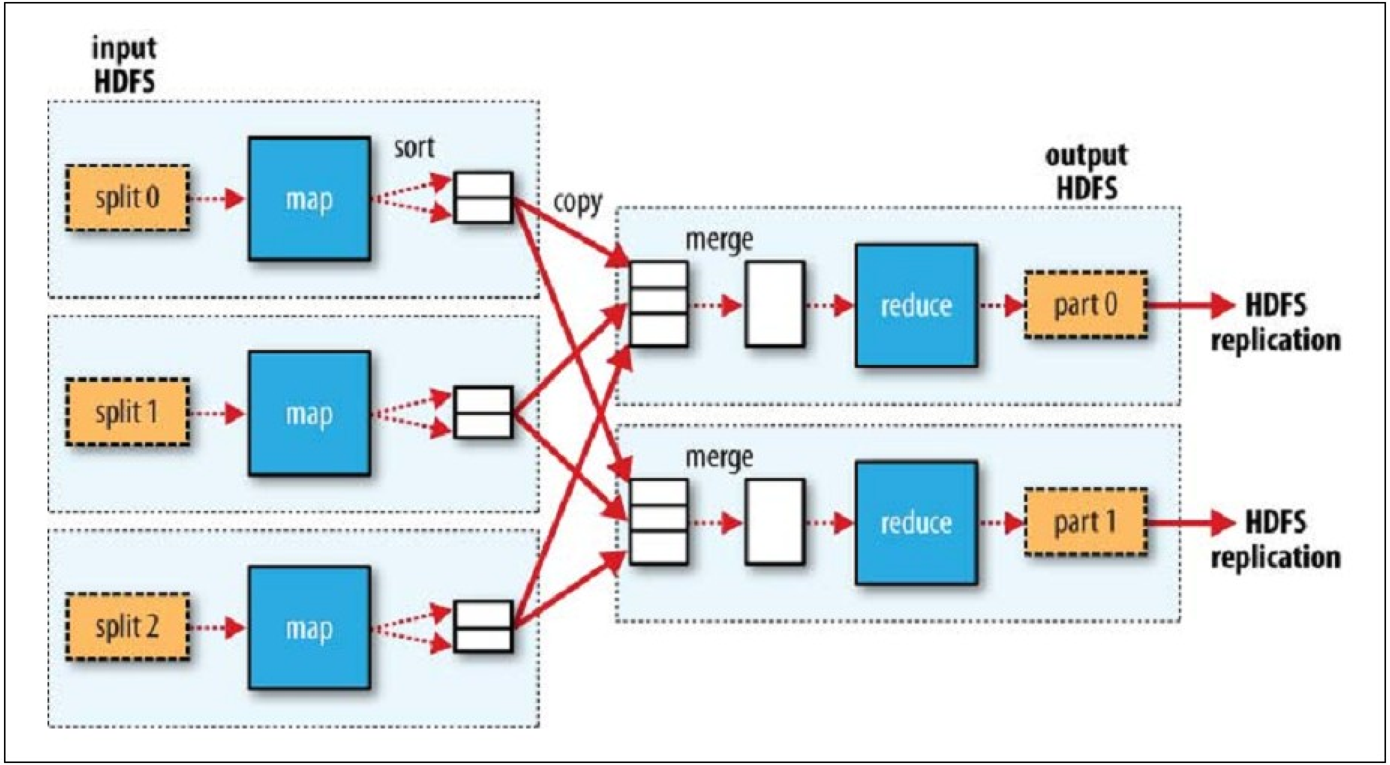
- Hadoop divides the input to a MapReduce job into fixed-size pieces called input splits, or just splits.
  - Hadoop creates one map task for each split, which runs the user defined map function for each record in the split.

- Having many splits means the time taken to process each split is small compared to the time to process the whole input.
  - On the other hand, if splits are too small, then the overhead of managing the splits and of map task creation begins to dominate the total job execution time.
  - For most jobs, a good split size tends to be the size of an HDFS block, 64 MB by default.

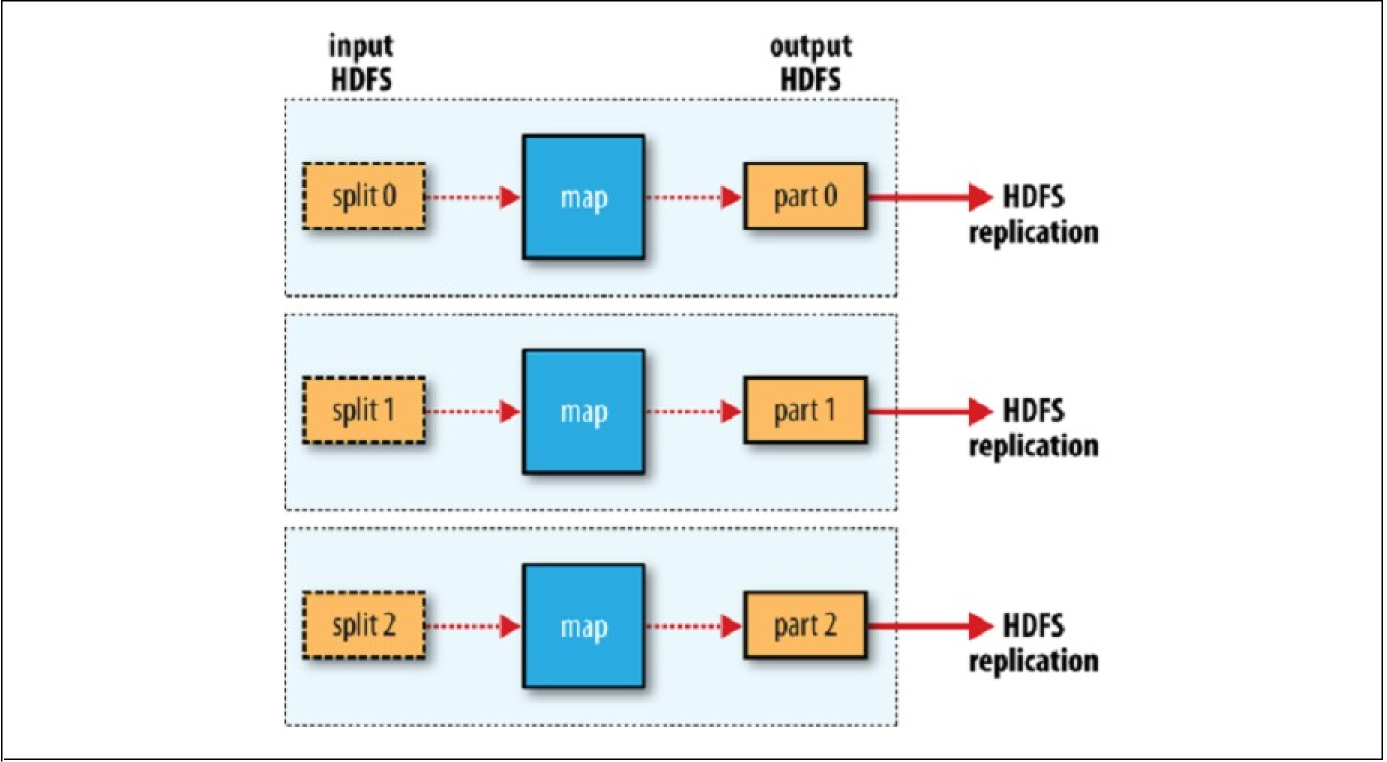
Scaling Out


input
HDFS
split 0
map
sort
split 1
map
split 2
map
copy
output
HDFS
merge
reduce
part 0
HDFS
replication

Scaling Out

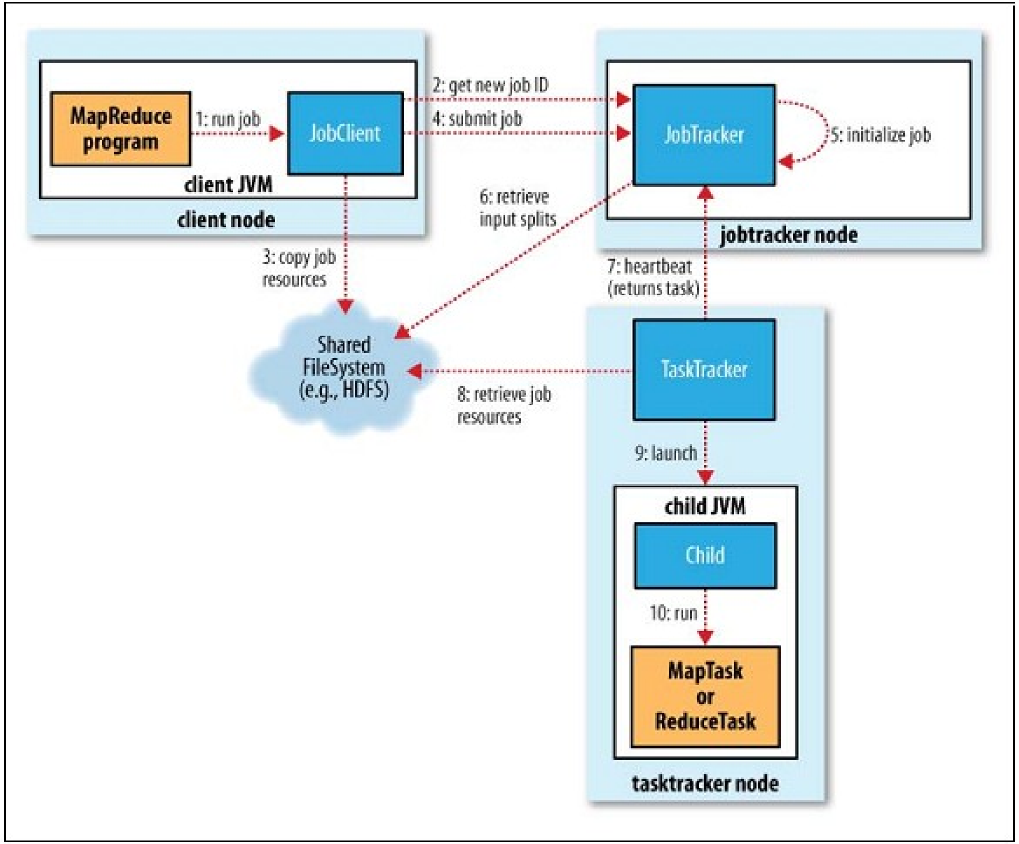

input
HDFS
split 0
split 1
split 2
map
map
map
sort
copy
output
HDFS
merge
merge
reduce
reduce
part 0
part 1
HDFS
replication
HDFS
replication

Scaling Out


input
HDFS
output
HDFS
split 0
map
part 0
HDFS
replication
split 1
map
part 1
HDFS
replication
split 2
map
part 2
HDFS
replication

MapReduce inside: JobTracker

REliable, INtelligent & Scalable Systems



Mapper

- Mapper maps input key/value pairs to a set of intermediate key/value pairs.
  - Maps are the individual tasks that transform input records into intermediate records.
  - The transformed intermediate records do not need to be of the same type as the input records.
  - A given input pair may map to zero or many output pairs.
- The Hadoop MapReduce framework spawns one map task for each InputSplit generated by the InputFormat for the job.
  - Overall, mapper implementations are passed to the job via Job.setMapperClass(Class) method.
  - The framework then calls map(WritableComparable, Writable, Context) for each key/value pair in the InputSplit for that task.
  - Applications can then override the cleanup(Context) method to perform any required cleanup.
- Output pairs do not need to be of the same types as input pairs.
  - A given input pair may map to zero or many output pairs.
  - Output pairs are collected with calls to context.write(WritableComparable, Writable).

Mapper

REliable, INtelligent & Scalable Systems

- All intermediate values associated with a given output key are subsequently grouped by the framework,
  - and passed to the Reducer(s) to determine the final output.
  - Users can control the grouping by specifying a Comparator via Job.setGroupingComparatorClass(Class).
- The Mapper outputs are sorted and then partitioned per Reducer.
  - The total number of partitions is the same as the number of reduce tasks for the job.
  - Users can control which keys (and hence records) go to which Reducer by implementing a custom Partitioner.
- Users can optionally specify a combiner, via Job.setCombinerClass(Class),
  - to perform local aggregation of the intermediate outputs, which helps to cut down the amount of data transferred from the Mapper to the Reducer.
  - The intermediate, sorted outputs are always stored in a simple (key-len, key, value-len, value) format.

How Many Mappers?

REliable, INtelligent & Scalable Systems

- The number of maps is usually driven by the total size of the inputs, that is,
  - the total number of blocks of the input files.
- The right level of parallelism for maps seems to be around 10-100 maps per-node,
  - although it has been set up to 300 maps for very cpu-light map tasks.
  - Task setup takes a while, so it is best if the maps take at least a minute to execute.
- Thus,
  - if you expect 10TB of input data and have a blocksize of 128MB, you'll end up with 82,000 maps,
  - unless Configuration.set(MRJobConfig.NUM_MAPS, int) (which only provides a hint to the framework) is used to set it even higher.

Reducer

- Reducer reduces a set of intermediate values which share a key to a smaller set of values.
  - The number of reduces for the job is set by the user via Job.setNumReduceTasks(int).
  - Overall, Reducer implementations are passed the Job for the job via the Job.setReducerClass(Class) method and can override it to initialize themselves.
  - The framework then calls reduce(WritableComparable, Iterable<Writable>, Context) method for each <key, (list of values)> pair in the grouped inputs.
  - Applications can then override the cleanup(Context) method to perform any required cleanup.
- Reducer has 3 primary phases: shuffle, sort and reduce.

Shuffle & Sort

- **Shuffle**
  - Input to the Reducer is the sorted output of the mappers.
  - In this phase the framework fetches the relevant partition of the output of all the mappers, via HTTP.

- **Sort**
  - The framework groups Reducer inputs by keys (since different mappers may have output the same key) in this stage.
  - The shuffle and sort phases occur simultaneously; while map-outputs are being fetched they are merged.

Secondary Sort & Reduce

REliable, INtelligent & Scalable Systems

- **Secondary Sort**
  - If equivalence rules for grouping the intermediate keys are required to be different from those for grouping keys before reduction,
  - then one may specify a Comparator via Job.setSortComparatorClass(Class).
  - Since Job.setGroupingComparatorClass(Class) can be used to control how intermediate keys are grouped, these can be used in conjunction to simulate *secondary sort on values*.
- **Reduce**
  - In this phase the reduce(WritableComparable, Iterable<Writable>, Context) method is called for each <key, (list of values)> pair in the grouped inputs.
  - The output of the reduce task is typically written to the FileSystem via Context.write(WritableComparable, Writable).
  - The output of the Reducer is *not sorted*.

How Many Reducers?

- The right number of reduces seems to
  - be 0.95 or 1.75 multiplied by (*<no. of nodes>* * *<no. of maximum containers per node>*).
  - With 0.95 all of the reduces can launch immediately and start transferring map outputs as the maps finish.
  - With 1.75 the faster nodes will finish their first round of reduces and launch a second wave of reduces doing a much better job of load balancing.
- Increasing the number of reduces
  - increases the framework overhead,
  - but increases load balancing and lowers the cost of failures.
- The scaling factors above are slightly less than
  - whole numbers to reserve a few reduce slots in the framework for speculative-tasks and failed tasks.

Reduce NONE & Partitioner

- **Reducer NONE**
  - It is legal to set the number of reduce-tasks to *zero* if no reduction is desired.
  - In this case the outputs of the map-tasks go directly to the FileSystem, into the output path set by FileOutputFormat.setOutputPath(Job, Path).
  - The framework does not sort the map-outputs before writing them out to the FileSystem.
- **Partitioner**
  - Partitioner partitions the key space.
  - Partitioner controls the partitioning of the keys of the intermediate map-outputs.
  - The key (or a subset of the key) is used to derive the partition, typically by a *hash function*.
  - The total number of partitions is the same as the number of reduce tasks for the job.
  - Hence this controls which of the m reduce tasks the intermediate key (and hence the record) is sent to for reduction.
  - HashPartitioner is the default Partitioner.

Job Configuration

- Job represents a MapReduce job configuration.
  - Job is the primary interface for a user to describe a MapReduce job to the Hadoop framework for execution.
- Job is typically used to specify the Mapper, combiner (if any), Partitioner, Reducer, InputFormat, OutputFormat implementations.
- Optionally, Job is used to specify other advanced facets of the job such as
  - the Comparator to be used,
  - files to be put in the DistributedCache,
  - whether intermediate and/or job outputs are to be compressed (and how),
  - whether job tasks can be executed in a *speculative* manner (setMapSpeculativeExecution(boolean))/ setReduceSpeculativeExecution(boolean)),
  - maximum number of attempts per task (setMaxMapAttempts(int)/ setMaxReduceAttempts(int)) etc.
- Of course, users can use Configuration.set(String, String)/ Configuration.get(String) to set/get arbitrary parameters needed by applications.

Task Execution & Environment

- The MRAppMaster executes the Mapper/Reducer *task* as a child process in a separate jvm.
  - The child-task inherits the environment of the parent MRAppMaster.
  - The user can specify additional options to the child-jvm via the mapreduce.{map|reduce}.java.opts and configuration parameter in the Job
    - such as non-standard paths for the run-time linker to search shared libraries via -Djava.library.path=<> etc.
  - If the mapreduce.{map|reduce}.java.opts parameters contains the symbol *@taskid@* it is interpolated with value of taskid of the MapReduce task.

Task Execution & Environment

- Here is an example
  - showing jvm GC logging,
  - and start of a passwordless JVM JMX agent so that it can connect with jconsole and watch child memory, threads and get thread dumps.
  - It also sets the maximum heap-size of the map and reduce child jvm to 512MB & 1024MB respectively.
  - It also adds an additional path to the java.library.path of the child-jvm.

```
<property>
  <name>mapreduce.map.java.opts</name>
  <value>
  -Xmx512M -Djava.library.path=/home/mycompany/lib -verbose:gc -Xloggc:/tmp/@taskid@.gc
  -Dcom.sun.management.jmxremote.authenticate=false -Dcom.sun.management.jmxremote.ssl=false
  </value>
</property>

<property>
  <name>mapreduce.reduce.java.opts</name>
  <value>
  -Xmx1024M -Djava.library.path=/home/mycompany/lib -verbose:gc -Xloggc:/tmp/@taskid@.gc
  -Dcom.sun.management.jmxremote.authenticate=false -Dcom.sun.management.jmxremote.ssl=false
  </value>
</property>
```

Map Parameters

- A record emitted from a map will be serialized into a buffer and metadata will be stored into accounting buffers.
  - When either the serialization buffer or the metadata exceed a threshold, the contents of the buffers will be sorted and written to disk in the background while the map continues to output records.
  - If either buffer fills completely while the spill is in progress, the map thread will block.
  - When the map is finished, any remaining records are written to disk and all on-disk segments are merged into a single file.
  - Minimizing the number of spills to disk can decrease map time, but a larger buffer also decreases the memory available to the mapper.

Name	Type	Description
mapreduce.task.io.sort.mb	int	The cumulative size of the serialization and accounting buffers storing records emitted from the map, in megabytes.
mapreduce.map.sort.spill.percent	float	The soft limit in the serialization buffer. Once reached, a thread will begin to spill the contents to disk in the background.

Shuffle/Reduce Parameters

- Each reduce
  - fetches the output assigned to it by the Partitioner via HTTP into memory and periodically merges these outputs to disk.
  - If intermediate compression of map outputs is turned on, each output is decompressed into memory.
  - The following options affect the frequency of these merges to disk prior to the reduce and the memory allocated to map output during the reduce.

Shuffle/Reduce Parameters

Name	Type	Description
mapreduce.task.io.soft.factor	int	Specifies the number of segments on disk to be merged at the same time. It limits the number of open files and compression codecs during merge. If the number of files exceeds this limit, the merge will proceed in several passes. Though this limit also applies to the map, most jobs should be configured so that hitting this limit is unlikely there.
mapreduce.reduce.merge.inmem.thresholds	int	The number of sorted map outputs fetched into memory before being merged to disk. Like the spill thresholds in the preceding note, this is not defining a unit of partition, but a trigger. In practice, this is usually set very high (1000) or disabled (0), since merging in-memory segments is often less expensive than merging from disk (see notes following this table). This threshold influences only the frequency of in-memory merges during the shuffle.
mapreduce.reduce.shuffle.merge.percent	float	The memory threshold for fetched map outputs before an in-memory merge is started, expressed as a percentage of memory allocated to storing map outputs in memory. Since map outputs that can't fit in memory can be stalled, setting this high may decrease parallelism between the fetch and merge. Conversely, values as high as 1.0 have been effective for reduces whose input can fit entirely in memory. This parameter influences only the frequency of in-memory merges during the shuffle.
mapreduce.reduce.shuffle.input.buffer.percent	float	The percentage of memory- relative to the maximum heapsize as typically specified in mapreduce.reduce.java.opts- that can be allocated to storing map outputs during the shuffle. Though some memory should be set aside for the framework, in general it is advantageous to set this high enough to store large and numerous map outputs.
mapreduce.reduce.input.buffer.percent	float	The percentage of memory relative to the maximum heapsize in which map outputs may be retained during the reduce. When the reduce begins, map outputs will be merged to disk until those that remain are under the resource limit this defines. By default, all map outputs are merged to disk before the reduce begins to maximize the memory available to the reduce. For less memory-intensive reduces, this should be increased to avoid trips to disk.

Configured Parameters

- The following properties are localized in the job configuration for each task's execution:

Name	Type	Description
mapreduce.job.id	String	The job id
mapreduce.job.jar	String	job.jar location in job directory
mapreduce.job.local.dir	String	The job specific shared scratch space
mapreduce.task.id	String	The task id
mapreduce.task.attempt.id	String	The task attempt id
mapreduce.task.is.map	boolean	Is this a map task
mapreduce.task.partition	int	The id of the task within the job
mapreduce.map.input.file	String	The filename that the map is reading from
mapreduce.map.input.start	long	The offset of the start of the map input split
mapreduce.map.input.length	long	The number of bytes in the map input split
mapreduce.task.output.dir	String	The task's temporary output directory

Job Submission & Monitoring

REliable, INtelligent & Scalable Systems

- The job submission process involves:
  - Checking the input and output specifications of the job.
  - Computing the InputSplit values for the job.
  - Setting up the requisite accounting information for the DistributedCache of the job, if necessary.
  - Copying the job’s jar and configuration to the MapReduce system directory on the FileSystem.
  - Submitting the job to the ResourceManager and optionally monitoring it’s status.
- Job history files are also logged to
  - user specified directory mapreduce.jobhistory.intermediate-done-dir and mapreduce.jobhistory.done-dir, which defaults to job output directory.
- User can view the history logs summary in specified directory using the following command $ mapred job -history output.jhist
  - This command will print job details, failed and killed tip details.
  - More details about the job such as successful tasks and task attempts made for each task can be viewed using the following command $ mapred job -history all output.jhist

Job Control

- Users may need to chain MapReduce jobs to accomplish complex tasks which cannot be done via a single MapReduce job.
  - This is fairly easy since the output of the job typically goes to distributed file-system, and the output, in turn, can be used as the input for the next job.
- However, this also means that the onus on ensuring jobs are complete (success/failure) lies squarely on the clients.
- In such cases, the various job-control options are:
  - Job.submit() : Submit the job to the cluster and return immediately.
  - Job.waitForCompletion(boolean) : Submit the job to the cluster and wait for it to finish.

Job Input

- InputFormat describes the input-specification for a MapReduce job.
- The MapReduce framework relies on the InputFormat of the job to:
  - Validate the input-specification of the job.
  - Split-up the input file(s) into logical InputSplit instances, each of which is then assigned to an individual Mapper.
  - Provide the RecordReader implementation used to glean input records from the logical InputSplit for processing by the Mapper.
- The default behavior of file-based InputFormat implementations,
  - is to split the input into *logical* InputSplit instances based on the total size, in bytes, of the input files.
  - However, the FileSystem blocksize of the input files is treated as an upper bound for input splits.
  - A lower bound on the split size can be set via mapreduce.input.fileinputformat.split.minsize.
- TextInputFormat is the default InputFormat.

InputSplit

- InputSplit represents the data to be processed by an individual Mapper.
  - Typically InputSplit presents a byte-oriented view of the input, and it is the responsibility of RecordReader to process and present a record-oriented view.
- FileSplit is the default InputSplit.
  - It sets mapreduce.map.input.file to the path of the input file for the logical split.

RecordReader

REliable, INtelligent & Scalable Systems

- RecordReader reads <key, value> pairs from an InputSplit.
  - Typically the RecordReader converts the byte-oriented view of the input, provided by the InputSplit, and presents a record-oriented to the Mapper implementations for processing.
  - RecordReader thus assumes the responsibility of processing record boundaries and presents the tasks with keys and values.

Job Output

- OutputFormat describes the output-specification for a MapReduce job.
- The MapReduce framework relies on the OutputFormat of the job to:
  - Validate the output-specification of the job; for example, check that the output directory doesn't already exist.
  - Provide the RecordWriter implementation used to write the output files of the job. Output files are stored in a FileSystem.
- TextOutputFormat is the default OutputFormat.

OutputCommitter

- OutputCommitter describes the commit of task output for a MapReduce job.
- The MapReduce framework relies on the OutputCommitter of the job to:
  - Setup the job during initialization.
  - Cleanup the job after the job completion.
  - Setup the task temporary output.
    - Task setup is done as part of the same task, during task initialization.
  - Check whether a task needs a commit.
    - This is to avoid the commit procedure if a task does not need commit.
  - Commit of the task output.
    - Once task is done, the task will commit it’s output if required.
  - Discard the task commit.
    - If the task has been failed/killed, the output will be cleaned-up.
    - If task could not cleanup (in exception block), a separate task will be launched with same attempt-id to do the cleanup.
- FileOutputCommitter is the default OutputCommitter.

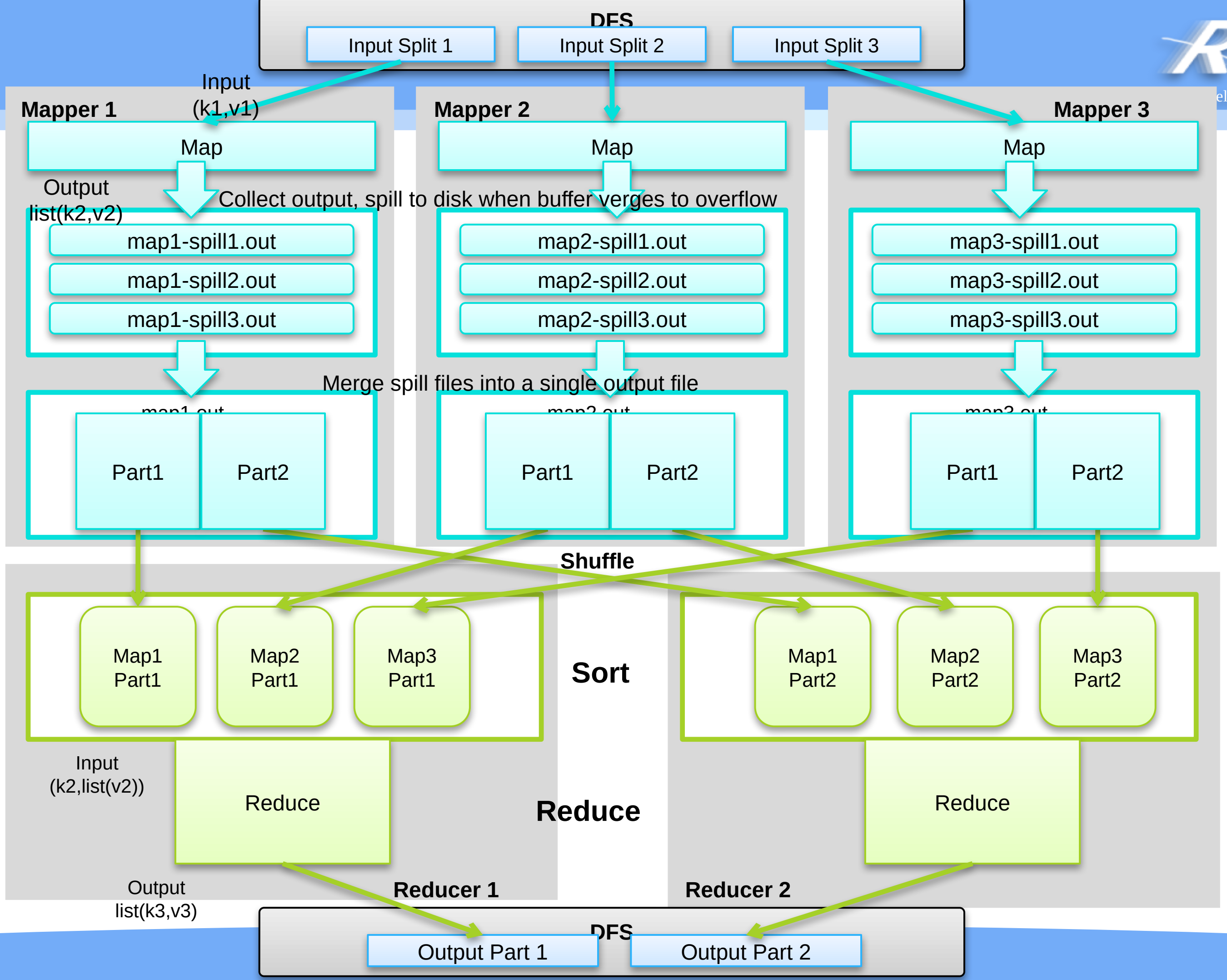
RecordWriter

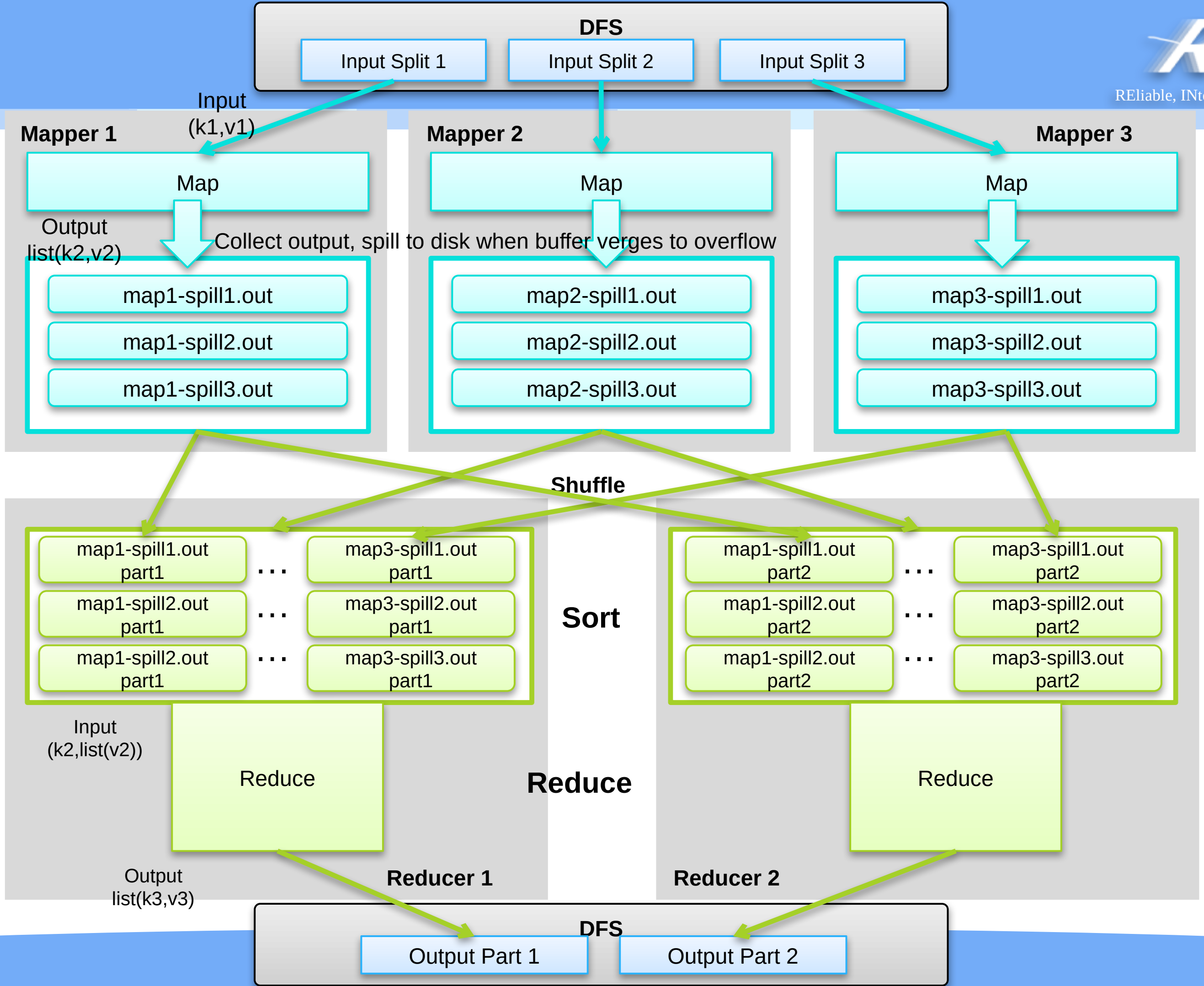
REliable, INtelligent & Scalable Systems

- RecordWriter writes the output <key, value> pairs to an output file.
- RecordWriter implementations write the job outputs to the FileSystem.

Other Useful Features

- **Submitting Jobs to Queues**
- **Counters**
- **DistributedCache**
- **Profiling**
- **Debugging**
- **Data Compression**
- **Skipping Bad Records**

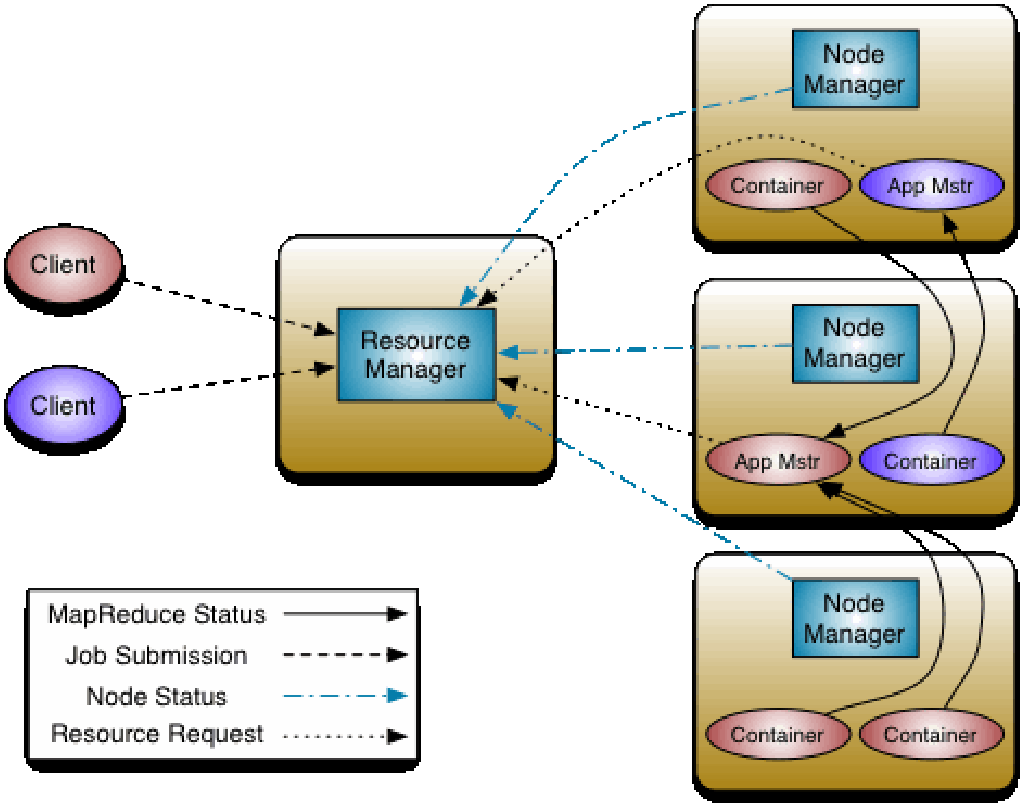

DFS
Input Split 1
Input Split 2
Input Split 3
Input
(k1,v1)
Mapper 1
Mapper 2
Mapper 3
Map
Map
Map
Output
list(k2,v2)
Collect output, spill to disk when buffer verges to overflow
map1-spill1.out
map1-spill2.out
map1-spill3.out
map2-spill1.out
map2-spill2.out
map2-spill3.out
map3-spill1.out
map3-spill2.out
map3-spill3.out
Merge spill files into a single output file
Part1
Part2
Part1
Part2
Part1
Part2
Shuffle
Map1
Part1
Map2
Part1
Map3
Part1
Sort
Map1
Part2
Map2
Part2
Map3
Part2
Input
(k2,list(v2))
Reduce
Reduce
Reduce
Output
list(k3,v3)
Reducer 1
Reducer 2
DFS
Output Part 1
Output Part 2


DFS
Input Split 1
Input Split 2
Input Split 3
Input
(k1,v1)
Mapper 1
Mapper 2
Mapper 3
Map
Map
Map
Output
list(k2,v2)
Collect output, spill to disk when buffer verges to overflow
map1-spill1.out
map1-spill2.out
map1-spill3.out
map2-spill1.out
map2-spill2.out
map2-spill3.out
map3-spill1.out
map3-spill2.out
map3-spill3.out
Shuffle
map1-spill1.out part1
map3-spill1.out part1
map1-spill2.out part1
map3-spill2.out part1
map1-spill2.out part1
map3-spill3.out part1
Sort
map1-spill1.out part2
map3-spill1.out part2
map1-spill2.out part2
map3-spill2.out part2
map1-spill2.out part2
map3-spill3.out part2
Input
(k2,list(v2))
Reduce
Reduce
Reduce
Output
list(k3,v3)
Reducer 1
Reducer 2
DFS
Output Part 1
Output Part 2

YARN

- **Apache Hadoop NextGen MapReduce (YARN)**
  - MapReduce has undergone a complete overhaul in hadoop-0.23 and we now have, what we call, MapReduce 2.0 (MRv2) or YARN.
  - The fundamental idea of MRv2 is to split up the two major functionalities of the JobTracker, resource management and job scheduling/monitoring, into separate daemons.
  - The idea is to have a global ResourceManager (*RM*) and per-application ApplicationMaster (*AM*).
  - An application is either a single job or a DAG of jobs.

YARN


Client
Client
Resource Manager
Node Manager
Container
App Mstr
Node Manager
App Mstr
Container
Node Manager
Container
Container
MapReduce Status
Job Submission
Node Status
Resource Request

YARN

- The ResourceManager has two main components: Scheduler and ApplicationsManager.
  - The Scheduler is responsible for allocating resources to the various running applications subject to familiar constraints of capacities, queues etc.
  - The Scheduler performs its scheduling function based the resource requirements of the applications;
    - it does so based on the abstract notion of a resource *Container* which incorporates elements such as memory, cpu, disk, network etc. In the first version, only memory is supported.
  - The Scheduler has a pluggable policy plug-in, which is responsible for partitioning the cluster resources among the various queues, applications etc.
    - The current Map-Reduce schedulers such as the CapacityScheduler and the FairScheduler would be some examples of the plug-in.

YARN

- The ApplicationsManager is responsible for
  - accepting job-submissions,
  - negotiating the first container for executing the application specific ApplicationMaster
  - and provides the service for restarting the ApplicationMaster container on failure.

- The NodeManager is the per-machine framework agent
  - who is responsible for containers,
  - monitoring their resource usage (cpu, memory, disk, network)
  - and reporting the same to the ResourceManager/Scheduler.

- The per-application ApplicationMaster has the responsibility of
  - negotiating appropriate resource containers from the Scheduler, tracking their status and monitoring for progress.

References

- Hadoop: Setting up a Single Node Cluster
  - https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-common/SingleCluster.html
- MapReduce Tutorial
  - https://hadoop.apache.org/docs/stable/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html
- 关于 Mac 中 ssh: connect to host localhost port 22: Connection refused
  - https://blog.csdn.net/u011068475/article/details/52883677
- **Hadoop3.x 启动后无法访问 9870**
  - https://blog.csdn.net/weixin_43867016/article/details/116855522
- Apache Hadoop YARN
  - https://hadoop.apache.org/docs/stable/hadoop-yarn/hadoop-yarn-site/YARN.html
- Hadoop: The Definitive Guide, By Tom White, O'Reilly Publishing
  - https://github.com/tomwhite/hadoop-book/
- 使用 intellij idea 在本地开发 hadoop 程序
  - https://www.aboutyun.com/thread-22749-1-2.html

Thank You!