CS286 Solving Big Data Problems – Exam #1 Study Guide

By: Zayd Hammoudeh

Lecture #01 – Introduction to Big Data

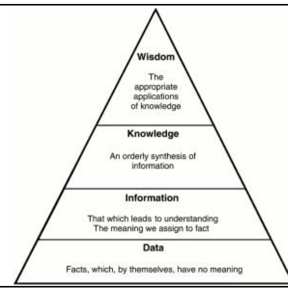
Data Categories	
Quantitative • Observable and **measureable** • Structured and **objective** • Numerical **Example:** Income, Height	**Qualitative** • Observable but not **measureable** • Unstructured and subjective • **Descriptive** **Example:** Favorite Color

Data – Raw values

Information – Set of data with meaning

Knowledge –Interpretation of the data with meaning.

Wisdom – Appropriate application of knowledge.



Storage Terminology

Directly Attached Storage (DAS)	Network Attached Storage (NAS)	Relational Database Management System (RDBMS)	Storage Area Network (SAN)	Network File System (NFS)
• Storage attached directly to the processing node. • Lowest capacity • Minimal data sharing • **Highest Speed.**	• Storage accessible via a network connection. • **Capable of using NFS**	• Traditional database providers. • **Examples:** Oracle, MySQL, IBM DB2	• Storage accessible via a network connection. • Uses different protocols than NAS.	Allows a computer to view and store data on remote disk as if that disk was directly attached to the local computer. **Access Transparency** – Access data the same way whether it is remote or local.

Data Analysis Categories	
Descriptive • **Backward** looking. • Hindsight • Explain a previous phenomenon. • **Analysis**	**Predictive** • **Forward** looking • Foresight • Investigate future trends. • **Mining**

Four Steps in Traditional Data Mining

1. Problem Definition
2. Data gathering and preparation
3. Model building and evaluation
4. Knowledge Deployment

Process is **cyclical** and **may repeat multiple times**.

Big Data

Big Data – Data whose scale, diversity, and complexity require new architecture, techniques, algorithms, and analytics to manage it and to extract value and hidden knowledge from it.

3 V's of Big Data		
Volume – The amount of data is too large for traditional database software tools to cope with. **Example:** Image server	**Velocity** – The data is being produced at a rate that is beyond the performance limits of traditional systems. **Example:** Social media site	**Variety** – Data lacks the structure to make it suitable for storage and analysis in traditional databases and data warehouses. **Example:** Data organization variety.

Data Organization		
Structured – Every piece of data and its format is known. Fits in a database. **Example:** RDBMS	**Semi-structured** – For some fields, data may not exist and some fields can have different formats. Not in a typical database but has structure. **Example:** XML, CSV, JSON	**Unstructured** – Does not fit into a database well. Most data is in this category. **Examples:** Text document, multimedia content.

Scaling to Process Big Data		
Scale Up **Limitations:** • Large capital and operating expense. • Lower availability and scalability. **Example:** Monolithic Database	**Scale Out** **Limitations:** • Synchronization overhead • Programming Complexity • Specialized hardware. **Example:** Grid Cluster	**Sampling** **Limitation:** • Lower accuracy and precision. **Example:** Any approach

Exploiting Locality of Reference – In Big Data, accessing the data can be very time consuming. **Solution:** Keep the data and program close together.

Distribute Data and Computation – Map the data to multiple nodes and the program with it to decrease execution time.

Three Laws of Big Data		
Moore's Law – Every two years, the number of transistors per chip doubles. **Kryder's Law** – Every two years, storage capacity doubles. (**Storage version of Moore's Law**)	**Amdahl's Law** – The extent to which a program's execution can be sped up is dependent on its level of parallelism.	**Murphy's Law** – What can go wrong will go wrong. **Big data must be resistant to failures.**

Hadoop

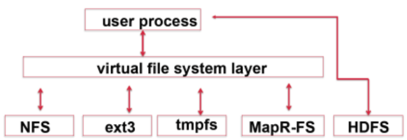
Summary of the Hadoop Strategy					
Distribute Data Processing nodes share no data.	**Distribute Computation** Achieve parallelism without synchronization.	**Tolerate Failures** Eliminate single points of failure.	**Core of Hadoop** 1. Hadoop File System (HDFS) 2. Map Reduce	**Name Node** Key component in HDFS that stores the location of distributed data in the file system.	**Job Tracker** Manages computation tasks in the Hadoop system.

Lecture #02 – Introduction to HDFS and MapR-FS

File System	**Storage in a File System**	Block Structure in an ext2 File System				
Like a database. A system to store data so that the data can be accessed later. **Typical Structure:** A rooted tree.	**Data** – Actual file in the FS. **Metadata** – Information about the data/file. **Example:** Size, location	**Hadoop Block Size:** 64MB	**Inode** – Data structure used to represent a file system object. This includes the location of the disk block location.	**Direct Block** – File block location pointed to directly by the inode.	**Indirect Block** – Block pointed to by the inode through exactly one intermediary block.	**Double Indirect Block** – Block pointed to by the inode through exactly two intermediary blocks.

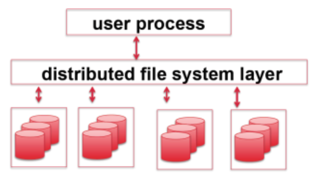
Virtual File System

- Transition layer between a generic file system and a real file system.
- Virtualizes different file system types into a single common interface.
- Enables standard POSIX file access.
- HDFS is not compatible with a virtual file system while MapR-FS is.

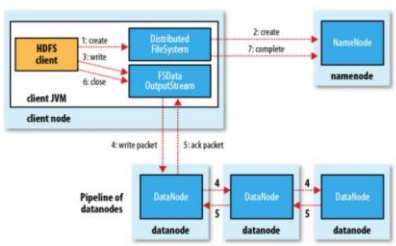


Distributed File System

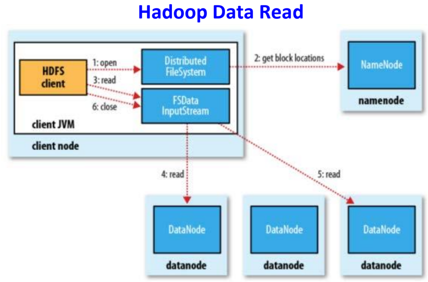
- Centrally stores metadata (e.g. name node) and distributes actual data (e.g. data node)
- Overcomes space, performance, and availability limitations of a single machine.
- Location Transparency – Abstracts data locality from client access.



Hadoop Data Write



Hadoop Data Read



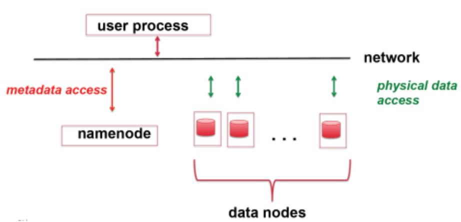
Hadoop Write Pipeline – Before a write can be acknowledged to the client, it must be acknowledged by the name node.

- Each replicate write is sequential through a pipeline where one data node writes to the next.

Sequential Block Reading – Each file block is read sequentially even if the blocks reside on multiple data nodes and could theoretically be read in parallel.

- Block size: 64MB

Hadoop Distributed File System (HDFS) Architecture



User Process

- Connected to HDFS through the network.
- Communicates with the name node to know where to read and write data.

Name Node – Master

- Manages file names and locations on disk. **Provides metadata information**
- All data is persisted in memory (RAM)
- May have a secondary name node used to offload processing (e.g. writing logs) off the primary. Secondary is not for high availability.
- All writes must be acknowledged by the name node before they can be acknowledged to the user process.

Data Node – Slave

- Persistent storage disks for the data.
- Data is replicated across multiple data nodes if possible across multiple racks.

Limitations of HDFS

Mutability	**Block Size**	**POSIX Semantics**	**Availability**	**Scalability**	**Performance**
Data is write once, read many.	Single block size (e.g. 64MB) for disk I/O, replication and sharding	Must use the command "`hadoop fs`" to access the data. Example POSIX Commands: Open, close, read, write.	No snapshot or built-in mirroring capability.	Name node only scales to 100M files. This is due to the single name node persisting all data in RAM.	Written in Java and runs on a block device

Overview of MapR File System (MapR-FS)

Physical Disk – A single hard drive. **Storage Pool** – Three striped physical disks. Striping is used to increase write performance.	**Node** – A set of storage pools. **Topology** – A set of nodes.	**Container** – Unit of shared storage. It is the size of replicated data. A storage pool has multiple containers. Each container belongs to only one volume.	**Volume** – A tree of files and directories grouped for the purpose of applying a policy or set of policies.	

MapR-FS Volume Features

Topologies	Compression	Mirroring	Snapshots	Quotas	Permissions	Replication
Provide data placement policies.	Compress data as it is written to disk.	Copy data locally or remotely for protection **in real time for load balancing, backup, and disaster readiness.**	Maintain point-in-time data and updates.	Restrict total capacity per-user or per-group.	Restrict access to users or groups.	Replicate containers in a volume across the cluster

Differences between MapR-FS and HDFS

Block Size	Mutability	Access	POSIX Support	Availability	Scalability	Performance
MapR-FS supports different block sizes for sharding, replication, and performing I/O.	MapR-FS has full read write capability.	MapR-FS volumes can be NFS-mounted.	MapR-FS supports native OS commands to access data.	MapR-FS supports snapshots and local/remote mirroring support.	No limit to the number of files.	**MapR-FS is written in C** and **runs on a raw device (i.e. no filesystem overhead).**

Block Size Comparison between HDFS and MapR-FS

Storage Unit	HDFS	MapR-FS
Unit of Sharding	Block=64MB	Chunk=256MB
Unit of Replication	Block=64MB	Container = 16-32GB
Unit of I/O	Block=64MB	Block=8KB

MapR-FS allows for different storage unit sizes to optimize performance.

Role of a Single Sharding Unit (e.g. **Block/Chunk**) – In Map Reduce, **each mapper is assigned a single shard** (e.g. block/chunk) to analyze.

Relationship between Container and Volume – In MapR-FS, a container is assigned to a single volume and a volume is made up of one or more containers.

Example Block/Chunk Count Calculation: If a Map Reduce file has 300MB of data, it will required 5 blocks in HDFS and 2 chunks in MapR-FS.

Using the "`hadoop fs`" Command Line Interface (CLI)

Format:

```
hadoop fs -<command> [args]
```

Examples:

```
hadoop fs -mkdir newDirectory

hadoop fs -rm my_file.txt
```

Not Supported Command:

```
hadoop fs -cd …
```

This command has no directory state so must use absolute path.

Lecture #03 – Introduction to Map Reduce

Map Reduce Underlying Principle
Divide and Conquer

Derives from Lisp

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

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

Reduce is called one on each key NOT each partition.

Key Methods

EmitIntermediate – Output of the mapper function. Writes an intermediary key-value pair to be analyzed by a reducer.

Emit – Outputs the result of the reducer.

Three Phases of Map Reduce

1. **Map**
2. **Sort/Shuffle/Merge**
3. **Reduce**

Map

- One mapper per input split. The **"map" function is called once for each key-value pair** (i.e. **record**).
- Each mapper processes a local data set and **outputs a set of intermediary key value pairs**.
- "**Send the compute to where the data is.**"
- **Outputs zero or more key value pairs.**

Sort/Shuffle/Merge

- Transfer results from mappers to reducers.
- Creates ***n*** partitions where ***n*** is equal to the number of reducers.
- Divides intermediary key value pairs into the ***n*** partitions.
- May run a "**Combiner**" function to merge results from the Map stage to reduce the amount of data to transfer over the network.
- After keys are partitioned and merge, the keys in the partition are sorted.
- Partitions are sent over the network to the reducers. Hadoop uses HTTP while MapR-FS uses RPC.

Reduce

- One reducer per input partition. The "**reduce" method is called once per key**.
- **Outputs zero or more key value pairs.**
- Reads one list of values for each key.
- **No data locality exploitation in reduce**.

Responsibilities of the Map Reduce Framework

- **Split the incoming input file and read the records.**
- **Schedules, runs, and reruns map/reduce tasks.**
- **Transfers map outputs to reduce inputs.**
- **Collects and writes status and results.**

Map Reduce Block and Record Splitting

- The Map Reduce framework divides an input file to one or more **splits\block**.
- A split\block contains one or more (typically many) **records**. **Default record delimiter is "\n".**
- **The map function is called once per record.**

Map Record Key-Value Format

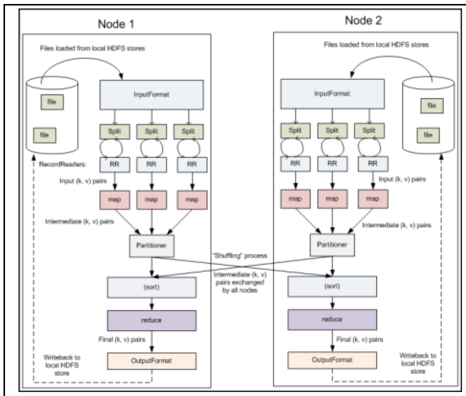
- **key** – **Byte offset** for start of record
- **value** – Record data in the split.

Typical Map Reduce Workflow

1. **Load the data into the cluster.**
   - **HDFS** – Uses WORM (write once read many). Preload only.
   - **MapR-FS** – POSIX + network file system (NFS) access. Preload or persistent storage.
2. **Analyze the data**
3. **Store the results in the cluster** (e.g. in HDFS/MapR-FS)

Read the results from the cluster.

Map Reduce Complete Flow



Map Reduce Complete Flow

1. Data is loaded into HDFS
2. The job decides the input format of the data.
3. Data is split between different mappers running on all the nodes.
4. **Record readers** (RR) parse out the data key-value pairs serve as inputs into the map() methods.

Map Reduce Complete Flow

5. The map() method produces key-value pairs that are sent to the **partitioner**.
6. When there are multiple reducers, the partition mapper creates one partition for each reduce task.
7. The key-value pairs are **sorted** by key within each partition.

Map Reduce Complete Flow

8. The reduce() method is take the intermediary key value pairs in the partition and reduces them to a **final list of key value pairs**.
9. The job defines the output format of the data.

Example Partition Function

$$Part\# = hash(key)\ \%\ \#Partitions$$

Hadoop Classes

`InputFormat`

- Checks if the input file exists.
- Splits the input file into one or more `InputSplit` objects.
- Instantiates `RecordReader` to partition splits into records which are turned into key-value pairs.
    - **Key is byte offset** of the start of the record.

`Mapper`

- Implements the map() method.
- One `Mapper` object is created for each input split.
- Processes keys and/or values.
- Updates status in reporter.
- Writes output.

`Partitioner`

- Takes the output(s) generated by the map() method and **creates partitions based on the hashed key**.
- Each partition is assigned to a single reducer.
- **All records with the same key are assigned to the same partition.**

`Combiner` (Optional)

- Has **no default behavior**.
- **Motivation**: Reduce the intermediate values of the mappers before they are sent over the network.
- **Often the reducer can be repurposed as a combiner.**

`Reducer`

- Implements the reduce() method.
- Each `Reducer` object is assigned one partition.
- Executes the reduce method on each key in the partition.
- Updates status in reporter.
- Writes output.

Outputs of a MapReduce Job

- **_SUCCESS** – Empty file indicating the job was completed successfully.
- **part-m-00000** – First intermediate results output file from a map task.
- **part-r-00000** – First intermediate results output file from a single reducer.

Hadoop Job Execution Framework

JobClient

- Instantiated by the client. Submits job to the JobTracker. Runs inside a JVM.

JobTracker

- Instantiates a Job object which gets sent to the TaskTracker(s). Runs inside a JVM.
- Reschedules tasks on failed TaskTrackers to other TaskTrackers.

TaskTracker

- Launches a child process that runs a **MapTask** or a **ReduceTask**.
- **HeartBeat Messages** to JobTracker include:
    - **Task Status**
    - **Task Counter**
    - **Data read/write status**

Hadoop Schedulers

- **Fair Scheduler** (**default**) – Resources shared evenly among pools.
    - Each user has a pool. Custom pools can be created. Supports Pre-emption.
- **Capacity Scheduler** – Resources shared among queues. Admin creates hierarchical queues. **Supports soft and hard capacity limits to users within a queue.**

Hadoop Fair Scheduler

- **Pool** – Set of jobs.
- **User configures priority of jobs within a pool.**
- Default of one user per pool.
- "Over-using" users can be preempted.
- **Developed at Facebook.**

Scheduling Algorithm

- Divide each pool's min maps and reduces among jobs.
- When a slot is free, allocate a job that is below its minimum share (i.e. most starved).
- Preempt long running jobs to meet minimum guarantees.

Hadoop Capacity Scheduler

- **Queue** – Set of Jobs
- Queues may be **hierarchically organized** (i.e. **a queue is made of other queues**).
- **Shares assigned to queues as a percentage of total resources.**
- Per-Queue and Per-User configurations.
- **Developed at Yahoo**.

Scheduling Algorithm

- Allocate slots to queues based on percentage of shares.
- **FIFO scheduling within each queue.**

MCS – MapR Control System

CLDB – Container Location Database.

Limitations of the Hadoop Execution Framework

Scalability	Availability	Inflexibility	Scheduler Optimization	Program Support
Single JobTracker restricts job throughput.	Only one JobTracker and one NameNode introduces single points of failure (SPOF).	Map and reduce jobs are not interchangeable.	Framework does not optimize scheduling of jobs.	Framework is limited to Map and Reduce programs.

Inflexibility and program support are addressed in Map Reduce version 2 (also known as **YARN**)

Lecture #04 – Installing MapR

Disk Provisioning	Network Configuration		Joining Data
• **Dynamic** – Thin provisioning • **Fixed** – Thick provisioning	• **NAT** – The VM does not have a separate IP from the host. Rather a separate private network is setup on the host machine and the VM gets an address in that network. Network traffic looks as though it came from the host PC.	• **Bridged** – Replicates another node on the physical network and the VM gets its own IP. • **Host-Only** – The nested VM's network is within the host computer only.	Join can be done in the map and reduce stages.

Lecture #05 – Writing a MapReduce Program

Common Map Reduce Applications

Summarizing Data	**Filtering Data**	**Organizing Data**	**Joining Data** Join can be done in the map and reduce stages.

MapReduce Program Imports

org.apache.hadoop.mapreduce.* Includes the definition of the "**Mapper**", "**Reducer**", "**Job**", and "**Context**" classes.	**org.apache.hadoop.io.*** Includes the definition of the "**Text**", "**LongWritable**", and "**IntWritable**" classes.	**org.apache.hadoop.conf.*** Includes the definition of the "**Configured**" and "**Configuration**" classes.	**org.apache.hadoop.util.*** Includes the definition of the "**Tool**" interface and "**ToolRunner**" class.
org.apache.hadoop.mapreduce.lib.input.* Includes the definition of the "**TextInputFormat**" and "**FileInputFormat**" classes.	**org.apache.hadoop.fs.*** Includes the definition of the "**Path**" class.	**java.util.*** Includes the definition of the "**StringTokenizer**" class.	**java.io.*** Includes the definition of the "**IOException**" class.
org.apache.hadoop.mapreduce.lib.output.* Includes the definition of the "**FileOutputFormat**" class.			

MapReduce Class Definitions

Mapper Class Definition

```
import java.util.*;
import java.io.*;
import org.apache.hadoop.mapreduce.*;
import org.apache.hadoop.io.*;

public class MyMapper
        extends Mapper <InputKeyClassName,
                        InputValueClassName,
                        OutputKeysClassName,
                        OutputValuesClassName> {

}
```

Must override the "**map**" method.

InputFormat – **TextInputFormat**
Key Class – **LongWritable**
Value Class – **Text**

Reducer Class Definition

```
import java.util.*;
import java.io.*;
import org.apache.hadoop.mapreduce.*;
import org.apache.hadoop.io.*;

public class MyReducer
        extends Reducer <InputKeyClassName,
                         InputValueSClassName,
                         OutputKeysClassName,
                         OutputValuesClassName> {

}
```

Must override the "**reduce**" method.

The input key and value types for the Reducer must match the output key and value types for the associated Mapper.

Driver Class Definition

```
import org.apache.hadoop.conf.*;
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapreduce.*;
import org.apache.hadoop.mapreduce.lib.input.*;
import org.apache.hadoop.util.*;

public class ReceiptsDriver extends Configured
                implements Tool {

...
  public static void main(String[] args) throws Exception{
    Configuration conf = new Configuration();
    System.exit(ToolRunner.run(conf, new ReceiptsDriver(), args));
  }

}
```

- Must implement the "**run**" method.
- **Specifies whether the job is run synchronously or asynchronously** via the "**waitForCompletion**" command.
- **Specifies class types for mapper and reducer.**
- Verifies function input arguments.

MapReduce Class Method Definitions

`map` Function Format

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

    StringTokenizer strToken = new StringTokenizer(value,
                                                    splitCriteria);

    // Iterate through all the tokens in the record
    while(strToken.hasTokens()){
        String myStr = strToken.nextToken();
        …
    }

    // Emit any intermediate <key, value> pairs
    // Optional to emit any pairs.
    context.write(new OutputKeysClassName(…),
                new OutputValuesClassName (…));
}
```

First two arguments in the `map` method are the input key and record value.

`Map` is **called once per input record**.

`reduce` Function Format

```java
@Override
public void reduce(Text key, Iterable<Text> value, Context context)
                throws IOException, InterruptedException {

    // Parse the Iterable object
    for(Text value: values)
    }

    // Emit any intermediate <key, value> pairs
    // Optional to emit any pairs.
    context.write(new OutputKeysClassName(…),
                new OutputValuesClassName (…));
}
```

`Reduce` is **called once per intermediate key**.

`run` Function Format

```java
@Override
public int run(String[] args) throws Exception {
    if(args.length != 2){
        System.err.printf("usage: %s [general options] <inputfile> <outputfile>\n",
                            getClass().getSimpleName());
        System.exit(1);
    }

    // Configure the job
    Job job = new Job ( getConf(), "job name");

    job.setJarByClass(MyDriver.class);
    job.setMapperClass(MyMapper.class);
    job.setReducerClass(MyReducer.class);

    // Define input file's format (e.g. text file)
    job.setInputFormatClass(TextInputFormat.class);

    // Setup the mapper output classes.
    // Mapper Input class are a LongWritable by default and Text
    job.setMapOutputKeyClass(MapperOutputKeysClassName.class);
    job.setMapOutputValueClass(MapperOutputValuesClassName.class);

    // Set the reducer's output class.
    job.setOutputKeyClass(ReduceOutputKeysClassName.class);
    job.setOutputKeyClass(ReduceOutputValuesClassName.class);

    // Set the reducer's output class.
    FileInputFormat.addInputFormat (job, new Path(<inputfilepath>);
    FileOutputFormat.setOutputFormat (job, new Path(<outputfolderpath>);


    // Wait for the job to finish.
    return job.waitForCompletion(true) ? 0 : 1;
}
```

MapReduce Environment Variables

HADOOP_HOME
- **Path:** /opt/mapr/hadoop/Hadoop-0.20.2
- Not required.
- **Useful when defining other environment variables.**

LD_LIBRARY_PATH
- **Path:** $HADOOP_HOME/lib/native/Linux-amd64-64
- Not required.
- **Enables the use of libraries specifically compiled for MapR.**

PATH
- **Path:** $HADOOP_HOME/bin**:**$PATH
- Not required.
- **Order in PATH variable is important** as earlier items in the list take precedence.
- **Provides path to Hadoop executables so user does not need to specify the absolute path.**

CLASSPATH
- **Path:** /opt/mapr/hadoop/Hadoop-0.20.2
- Not required.
- **Points to all jars in the Hadoop distribution required to run a program.**

HADOOP_CLASSPATH
- **Path:** $CLASSPATH
- Not required.
- **Makes it easier to run MapReduce applications from the `hadoop` command.**

Command Line Instructions

`javac`
- **Compiles a Java class from ASCII to byte code.**
- **Example:**

  javac –d <*FolderName*> <*ClassName*>.java

- **-d** – Allows for a custom output directory to be used.

`hadoop jar`
- Launches a Hadoop job.
- **Example:**

  hadoop jar <*JarNameAndPath*>.jar <*DriverClass*> file://<*InputPathAndFile*> <*outputDirectory*>

- Arguments in the call correspond to the `args` argument in the driver.

`hadoop fs`
- Enables POSIX style commands on HDFS
- **Example:**

  hadoop fs -<CommandName> [args]

- **Must precede POSIX command (e.g. ls, cat, rm, etc.) with a hyphen.**

`jar`
- **Combines the different class files into a single Java Archive (JAR) File.**
- **Example #1: Create a New JAR File**

  jar –cvf <jarname>.jar –C <classfolder>/.

- **-c** – Create a new JAR file.
- **-v** – Generate a verbose output.
- **-f** – Specifies that the command includes the output JAR's file name.
- **-C** – Specifies the location of the source .class files.

- **Example #1: Updating an Existing JAR**

  jar –cvf <jarname>.jar –C <classfolder>/.

- **-U** – Update a JAR.

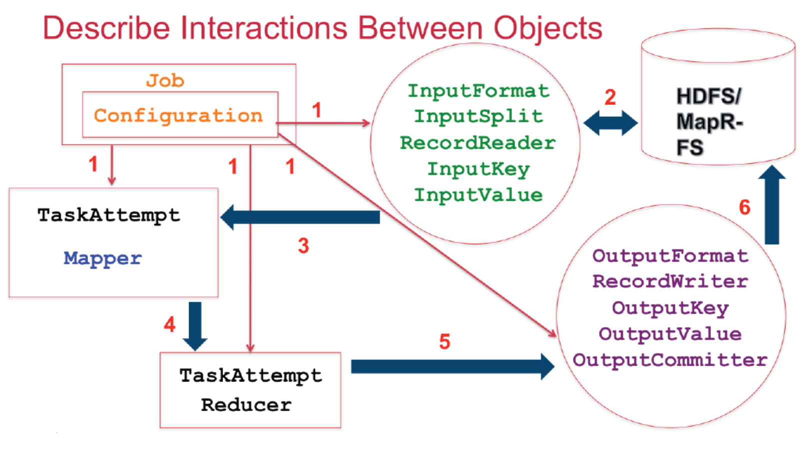
Lecture #06 – Using the mapreduce API

Hadoop and Mapr	Setting HADOOP_HOME PATH
• MapR currently ships with version **0.20.2** of Hadoop.	HADOOP_HOME = /opt/mapr/hadoop/hadoop-0.20.2

Comparison of the mapreduce and mapred Libraries

	Supported on MapR	Deprecated	YARN-Compatible	Types	Objects
mapred	Yes	No	Yes	Interfaces	`OutputCollector, Reporter, JobConf`
mapreduce	Yes	No	Yes	Abstract Classes	`Context`

	Methods	Output Files	Reducer Input Values	Import Command
mapred	`map()`, `reduce()`	part-xxxxx	`java.lang.Iterable`	`import org.apache.hadoop.mapred.*`
mapreduce	`map()`, `reduce()`, `cleanup()`, `setup()`, `run()`	part-m-xxxxx (Mapper) part-r-yyyyy (Reducer)	`Java.lang.iterator`	`import org.apache.hadoop.mapreduce.*`



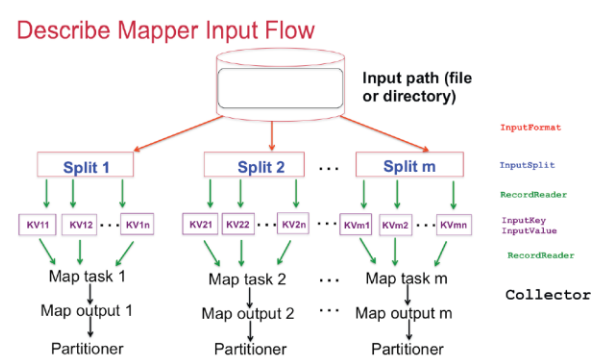
Writable Types

- **All Key/Value types must implement the Writable Interface.**
- Used to serialize keys/values before they are written to disk.
- All Java primitives must have a wrapper class to be able to return/pass from map/reduce calls.
- Do not support commands on equivalent Java primitives. Example: cannot use "+" to add to LongWritables.

Writable Interface Methods

- `void write(DataOutput out) throws IOException`
- `void readFields(DataInput in) throws IOException`

Java Primitive	Hadoop Writable Type
boolean	BooleanWritable
long	LongWritable new LongWritable(1)
double	DoubleWritable
string	Text (UTF-8 Format) new Text("my String")
N/A	BytesWritable (Writable Binary)



WritableComparable

- **All keys must implement the Writable and Comparable Interfaces.**

Comparable Interface

`int compareTo(WriteComparable o)`

- **`compareTo`** is used to provide a total ordering of keys in the Sort/Shuffle/Merge stage.
- Returns -1 if implicit parameter should be order first.
- Returns 0 if they are equal.
- Returns 1 if explicit parameter should be ordered first.

InputFormat Class

- Valid input files/directories exists.
- Partitions the input file into splits.
- Instantiates `RecordReader` for parsing records in the splits.
- Throws `IOException`

Methods

```
public abstract List<InputSplit> getSplits(JobContext)

public abstract RecordReader<K,V> createRecordReader(InputSplit split, TaskAttemptContext context)
```

Common Implementations

- `TextInputFormat` – Single Line Record Text Files. **Terminated by newline characters.**
- `SequenceFileInputFormat` – Binary Files

InputSplit Class

- Object that encapsulates a single file split.
- Logical representation of a subset of the data.
- **Split size is defined by**:

$$\max(minSplitSize, \max(maxSplitSize, blockSize))$$

Methods

`public abstract long getLength()`

`public abstract String[] getLocations()` – Gets a list of host names where the split is located.

Common Implementations

- `FileSplit`

Split Versus Block Size

- Split Size is configurable in Hadoop and Mapr.
- A split may be smaller, larger, or the same size as a block.

Record Boundaries – Two Possibilities

- **Last Record Boundary Falls On Split Boundary** – Read whole first record in the next split.
- **Last Record Boundary Falls in the Next Split** – Record reader reads the next split until the end of the record.

RecordReader Interface

- **Breaks up the data in an input split into Key-Value pairs**
- **Handles incomplete records**
  - **Discards first record in a split after the first split**
  - **Reads ahead to first delimiter in the next split (except the last split).**

Methods

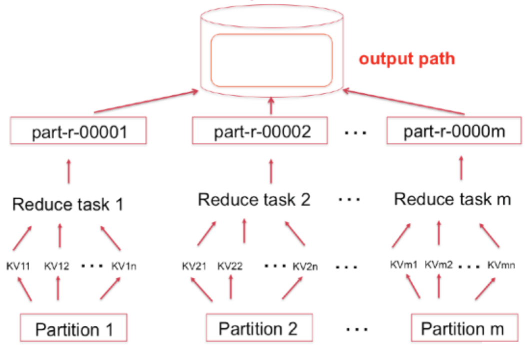
```
boolean next(K key, V value)
K createKey()
V createValue()
long getPos()
public void close()
float getProgress()
```

Common Implementations

- `LineRecordReader` – Used for text files. Key is byte offset and text is the line.
- `SequenceFileRecordReader` – Binary input files

Reducer Output Classes

Describe Reducer Output Flow



`OutputFormat` Class

- Valid output file specifications via method **checkOutputSpecs**.
- Provide `RecordWriter` to write output files.

Methods

```
public abstract RecordWriter<K,V>
getRecordWrite(TaskAttemptContext
context)

public abstract void
checkOutputSpecs(JobContext context)

public abstract OutputCommitter
getOuputCommitter(TaskAttemptContext
context)
```

Common Implementations

- `FileOutputFormat` – Wrapper of OutputFormat.
- `TextOutputFormat` – Plain text file.
- `NullOutputFormat` – Send all outputs to /dev/null
- `SequenceFileOutputFormat` – Binary Files

`RecordWriter` Class

- Writes the key value pairs to the output files.
- Can automatically compress the output streams as they are written to disk.

Methods

```
public abstract void write(K key, V
value)

public abstract void
close(TaskAttemptContext)
```

Common Implementations

- `TextOutputFormat.LineRecordWriter` – Writes Key-Value pairs to plain text files.

`OutputCommitter` Class

- Initializes the Job at job start (in `setupJob`())
- Cleans up the job upon job completion (in `cleanJob`()).
- Sets up the task temporary outputs (in `setupTask`())
- Checks whether a tasks needs to be committed (in `needsTaskCommit`())
- Commit of the task output (in `commitTask`())
- Discard the task commit (in `abortTask`())

Common Implementations

- `FileOutputCommitter` – Commits files to job output directory.

`Mapper` Class

- Based off **Java Generics** since keys and value types are generic.
- Primary method to override is **map**.
- `Context` object is used to output to intermediate files.
- `run` method calls `setup`, `map`, and `cleanup`.
- `setup` is called before `map` and `cleanup` is called after `map`.

Methods

```
protected void cleanup(Context context)

protected void map(KEYIN key, VALUEIN
value, Context context)

void run(Context context)

protected void setup(Context context)
```

Mapper and Reducer `run` Method

```
public void run(Context context){

   try{
      setup()
      while(context.nextKey()){
         map(context.getCurrentKey(),
               context.getCurrentValue(),
               context);
      }
   }
   finally{
      cleanup()
   }
}
```

`Reducer` Class

- Based off **Java Generics** since keys and value types are generic.
- If no `Reducer` class is specified, then `Mapper` outputs are sent directly as final outputs **after sorting by key**.
- Primary method to override is **reduce**.
- `Context` object is used to output to final files.
- `run` method calls `setup`, `reduce`, and `cleanup`.
- `setup` is called before `reduce` and `cleanup` is called after `reduce`. (**Similar to Mapper**)

Methods

```
protected void cleanup(Context context)

protected void map(KEYIN key,
Iterable<ValueIn> values, Context context)

void run(Context context)
```

- `protected void setup(Context context)`

`Job` Class

- Allows a user to configure and submit a job, control its execution, and query its state.
- To get a job's configuration, you use the **getConf**() method.

Constructors

```
Job()
Job(Configuration conf)
Job(Configuration conf, String jobName)
```

Example Usage

```
Configuration conf = new Configuration();
Job job1 = new Job(conf, "Job1");

   Job job2 = new Job(getConf(), "Job2");
```

Methods

`void failTask(TaskAttemptID taskID)` – Indicate task with specified ID failed.

`String getJar()` – Gets the Job's JAR file pathname.

`boolean isComplete()` – Gets whether the job has completed.

`boolean isSuccessful()` – Returns whether the job completed successfully.

`void killJob()` – Kills the job.

`void killTask(TaskAttemptID taskID)` – Kills the task with the specified ID failed.

`float mapProgress()` – Gets progress of the map tasks. Between 0 and 1.

`float reduceProgress()` – Gets progress of the reduce tasks. Between 0 and 1.

More Job Class Methods

`void setJarByClass(Class cls)` – Specifies the driver class.

`void setInputFormat(Class cls)` – Sets the `InputFormat` type for the job.

`void setMapperClass(Class cls)` – Sets the class type for the `Mapper`.

`void setMapOutputKeyClass(Class cls)` – Sets the class type for the `Mapper` output key(s).

`void setMapOutputValueClass(Class cls)` – Sets the class type for the `Mapper` output value(s).

`void setOutputFormat(Class cls)` – Sets the `OutputFormat` type for the job.

`void setReducerClass(Class cls)` – Sets the class type for the `Reducer`.

`void setOutputKeyClass(Class cls)` – Sets the class type for the `Reducer` output key(s).

`void setOutputValueClass(Class cls)` – Sets the class type for the `Reducer` output value(s).

`void submit()` – Submit the job to the cluster and return immediately.

`void waitForCompletion(boolean verbose)` – Submit the job to the cluster and wait for it to finish. **Often called within System.exit() with a ternary operator**.

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

Implementing the Driver

```
public class MyDriver extends Configured
                      implements Tool{
  public static void main(){
    Configuration conf = new Configuration();
    System.exit(ToolRunner.run(conf,
                               new MyDriver(),
                               args);
  }

  public int run(String[] args) throws Exception{
    Job job = new Job(getConf(), "My Job");
    ...
    return job.waitForCompletion(True) ? 0 : 1;
  }
}
```

User ToolRunner to execute driver code.

Job Configuration Code Example

```
Job job = new Job(getConf(), "myJob");

job.setJarByClass(MyDriver.class);
job.setMapperClass(MyMapper.class);
job.setReducerClass(MyReducer.class);

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

job.setInputFormatClass(TextInputFormat.class);
job.setOutputFormatClass(TextOutputFormat.class);

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

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

Drawback: Cannot be dynamically configured.

Levels of MapReduce Configuration Priority

Highest Priority

1. **Driver Code**
2. **Command Line Parameters**
3. **Local XML Files**
4. **Global XML Files (i.e. within the Global Map Reduce folder)**
5. **Hadoop Framework Modifications**

Lowest Priority