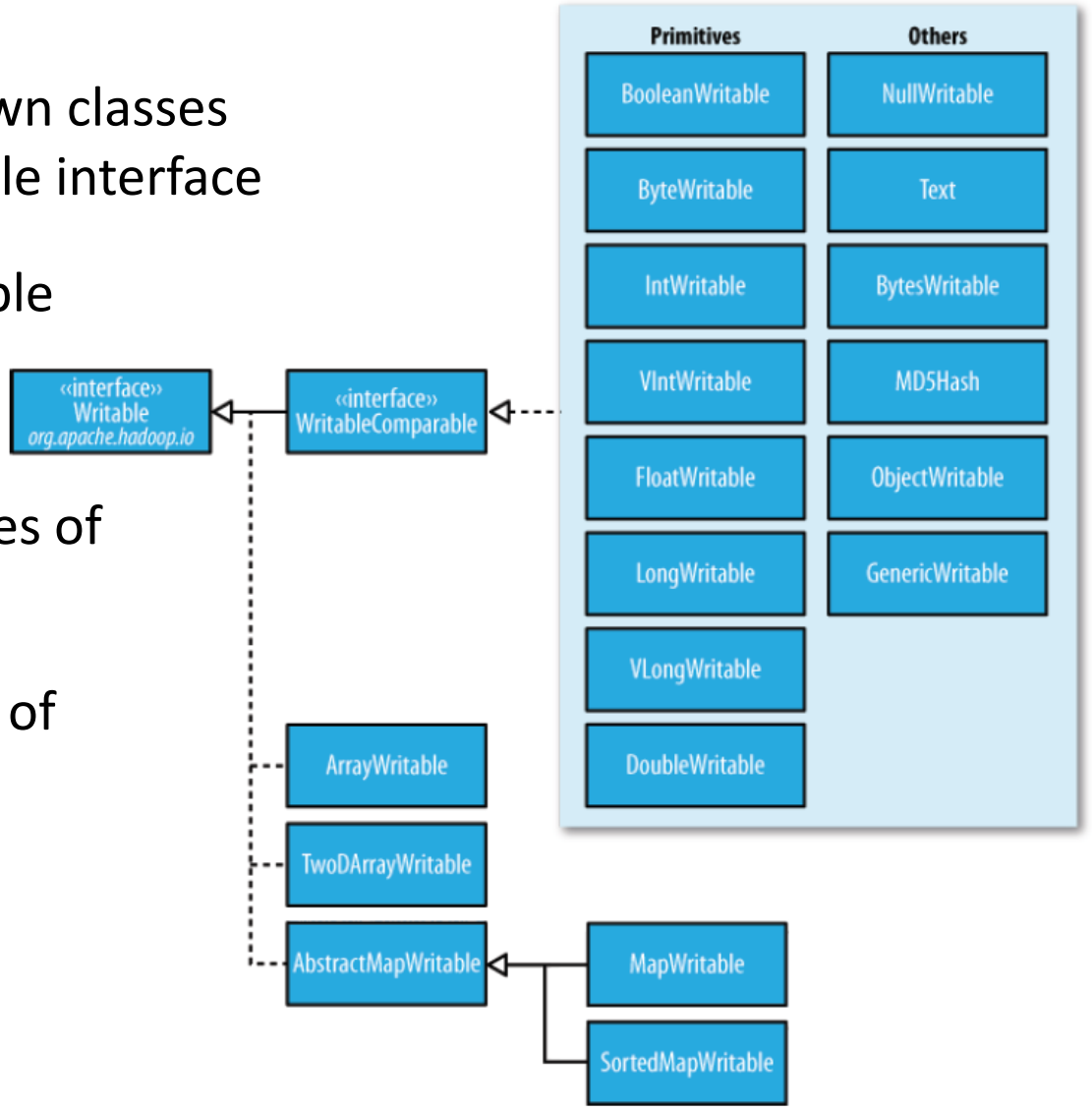
CMSC 691
High Performance Distributed Systems

Apache Hadoop Programming

Dr. Alberto Cano
Assistant Professor
Department of Computer Science
acano@vcu.edu

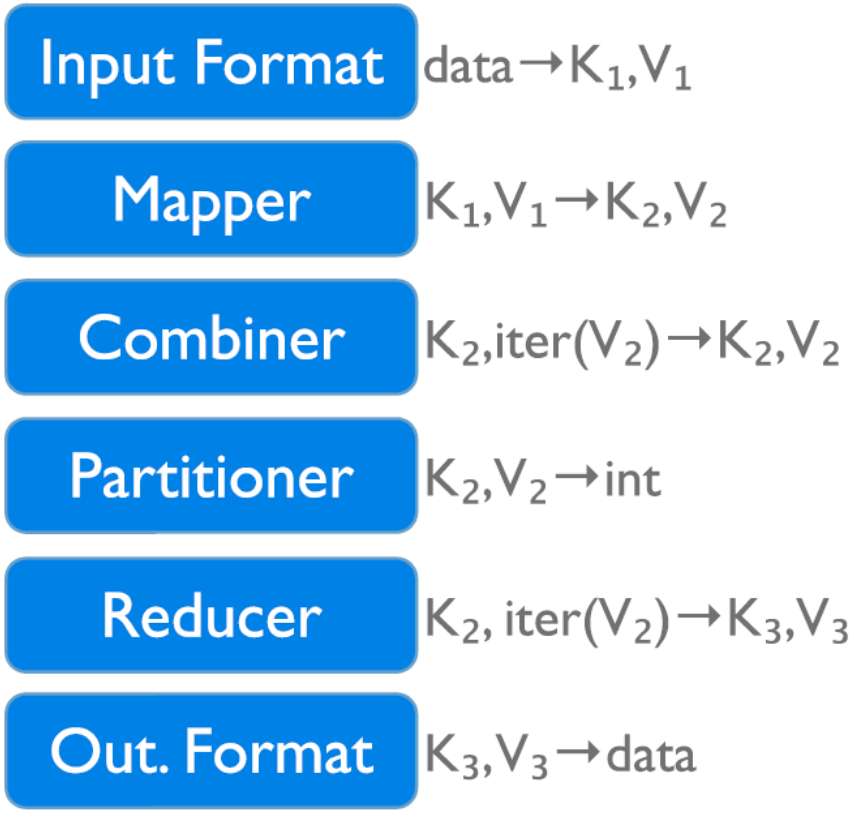
Hadoop data types

- Hadoop defines its own classes implementing Writable interface
- Writeable is serializable
- All values are instances of Writable
- All keys are instances of WritableComparable



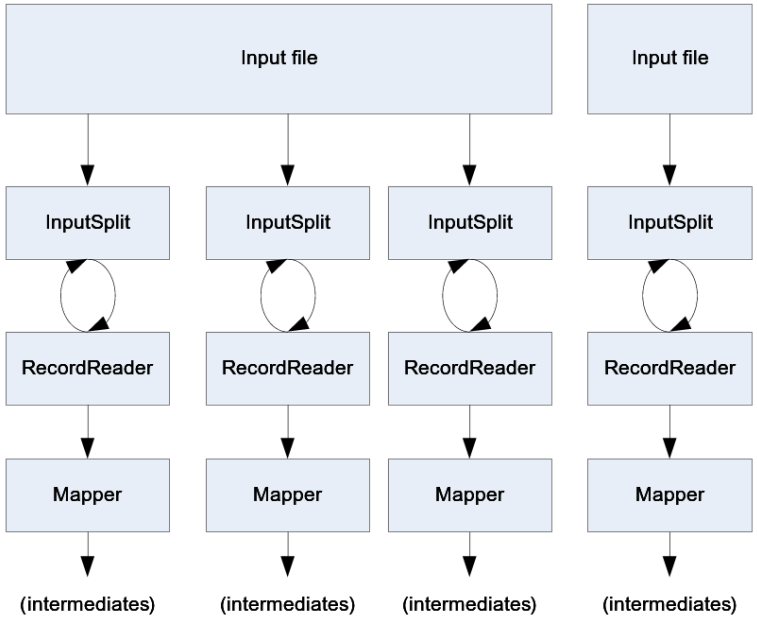
Data flow in Hadoop MapReduce

- InputFormat
- Map function
- Partitioner
- Sorting and merging
- Combiner
- Shuffling
- Merging
- Reduce function
- OutputFormat



Reading input data and feeding the Mapper

- Data sets are specified by InputFormats
- Defines input data (e.g., a directory)
- Identifies partitions of the data that form an InputSplit
- Factory for RecordReader objects to extract (k, v) records from the input source



InputFormat<K,V>

- InputFormat describes the input-specification for a job
- Validate the input-specification of the job
- Split-up the input files into logical InputSplits, each of which is then assigned to an individual Mapper
- Provide the RecordReader implementation to be used to collect input records from the logical InputSplit for Mapper processing

InputSplit[] getSplits(JobConf job, int chunks)

Logically split the set of input files for the job.
We may override how this is done.

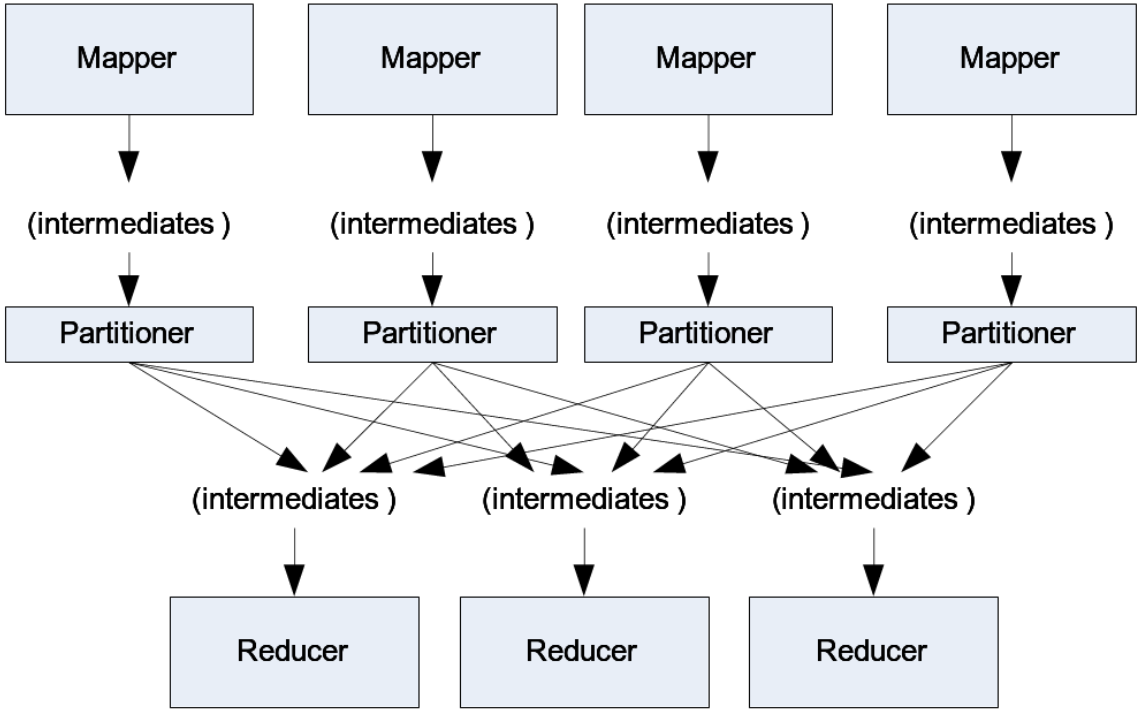
Mapper <KeyInput, ValueInput, KeyOutput, ValueOutput>

- Maps input key/value pairs to a set of intermediate keys/values
- The transformed intermediate records are not required to be of the same type as the input records. A given input pair may map to zero or many output pairs
- The Hadoop Map-Reduce framework spawns one map task for each *InputSplit* generated by the *InputFormat* for the job
- Results are returned via context.write(KeyOutput, ValueOutput)

```
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);
        }
    }
```

Output of the Mapper as Input of the Reducer

- Combiner (optional): mini-reducers that run in memory after the map phase, combining pairs with shared keys
- Partitioner (optional): translate Mapper outputs to Reducers



Partitioner<K,V>

- Partitioner controls partitioning of the intermediate map-outputs
- 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 reduce tasks the intermediate key is sent for reduction
- HashPartitioner used by default based on key.hashCode() % NRed

```
public class MyPartitioner implements Partitioner<IntWritable,Text> {

    public int getPartition(IntWritable key, Text value, int numPartitions) {
        int keyVal = key.get();
        return keyVal % numPartitions;
    }

    public void configure(JobConf arg0) {
    }
}
```

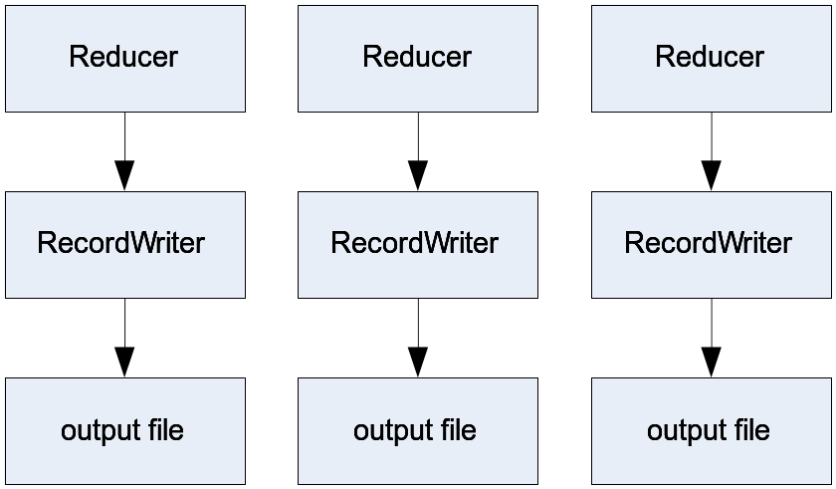
Reducer <KeyInput, ValueInput, KeyOutput, ValueOutput>

- Reduces a set of intermediate values which share a key to a smaller set of values
- Reducer has 3 primary phases:
  - Shuffle: input the grouped output of a Mapper
  - Sort: group by input key
  - Reduce: in this phase the reduce(Object, Iterator, OutputCollector, Reporter) method is called for each <key, (list of values)> pair in the grouped inputs.
- The shuffle and sort phases occur simultaneously, while outputs are being fetched they are merged

Reducer <KeyInput, ValueInput, KeyOutput, ValueOutput>

```
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
        int sum = 0;
        for (IntWritable val : values) {
            sum += val.get();
        }
        result.set(sum);
        context.write(key, result);
    }
}
```

Writing the output: OutputFormat



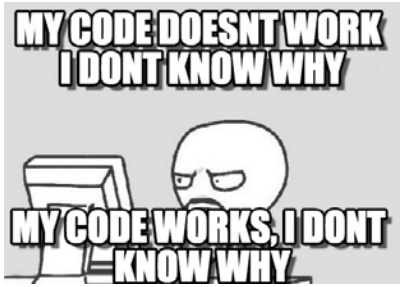
Job

- Allows the user to configure the job, submit it, control its execution, and query the state (monitor its progress)

```
public static void main(String[] args) throws Exception {
  Configuration conf = new Configuration();
  Job job = new Job(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(otherArgs[0]));
  FileOutputFormat.setOutputPath(job, new Path(otherArgs[1]));

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

Everything else is automatic

- The execution framework handles everything else
  - Scheduling: assigns workers to map and reduce tasks
  - Data distribution: moves processes to data
  - Synchronization: gathers, sorts, and shuffles intermediate data
  - Errors and faults: detects worker failures and restarts
- Limited control over data and execution flow
  - Algorithms: Mapper, Reducer, Combiner, Partitioner
- You don't know:
  - Where mappers and reducers run
  - When a mapper or reducer begins or finishes
  - Which input a particular mapper is processing
  - Which intermediate key a particular reducer is processing



But … we can override many things

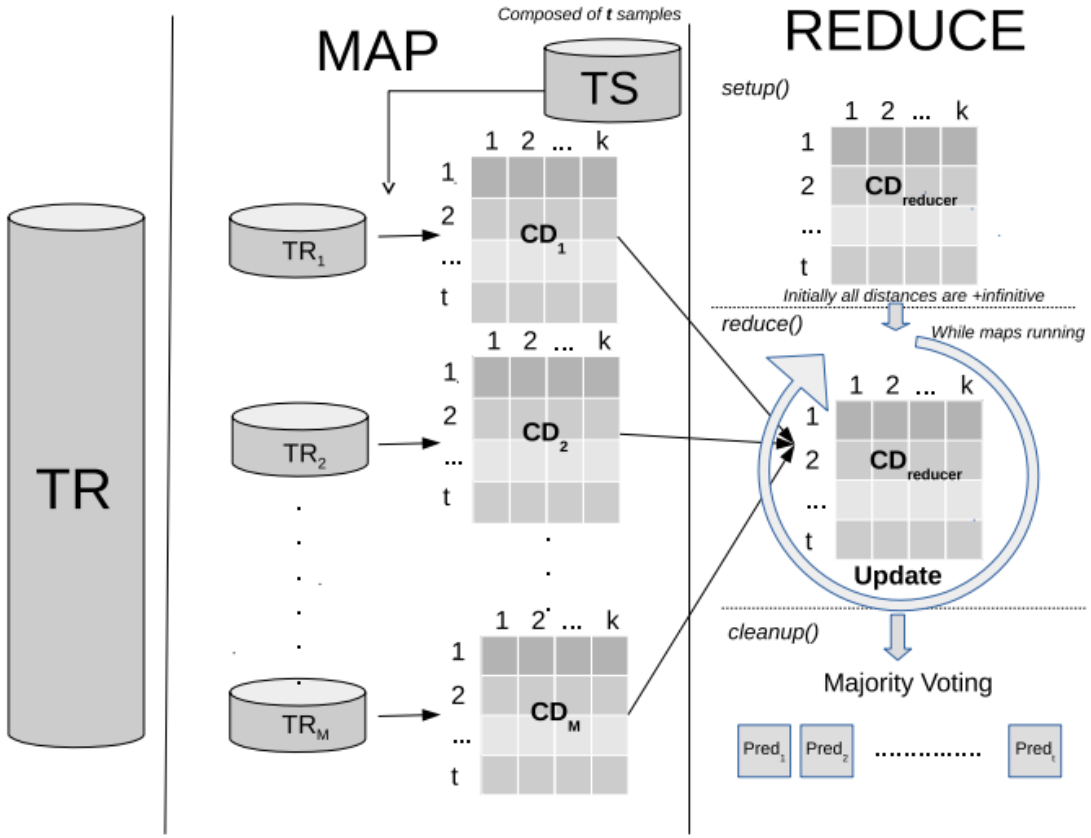
- setup() and cleanup() are methods you can override, if you choose, and they are there for you to initialize and clean up your map/reduce tasks
- setup() and cleanup() methods are simply hooks to have a chance to do something before and after your map/reduce tasks
- The lifecycle of a map/reduce task is:

  setup -> map -> cleanup

  setup -> reduce -> cleanup
- setup() and cleanup() are called once for each **task**

Hands on!

- KNN in Hadoop. Read the following paper and implement it.
- A MapReduce-based k-Nearest Neighbor Approach for Big Data Classification



CMSC 691
High Performance Distributed Systems

Apache Hadoop Programming

Dr. Alberto Cano
Assistant Professor
Department of Computer Science
acano@vcu.edu