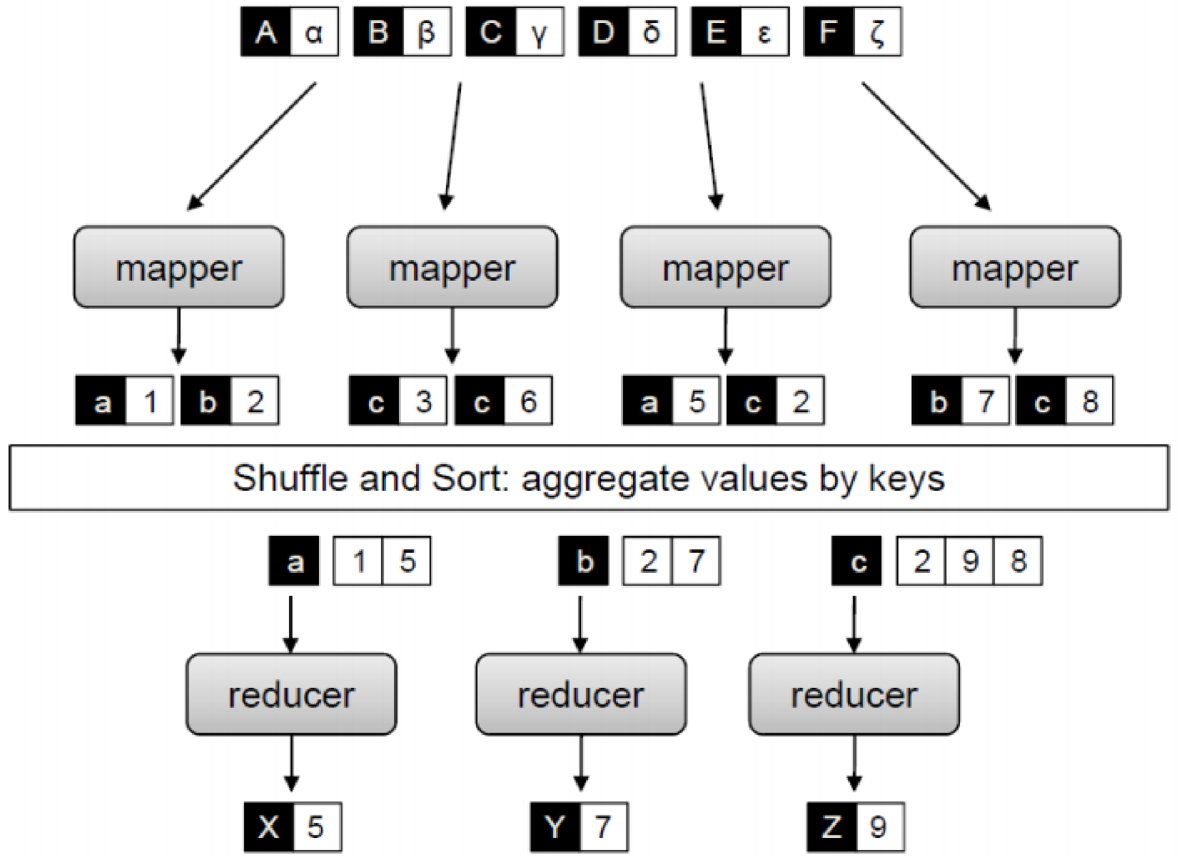
MapReduce

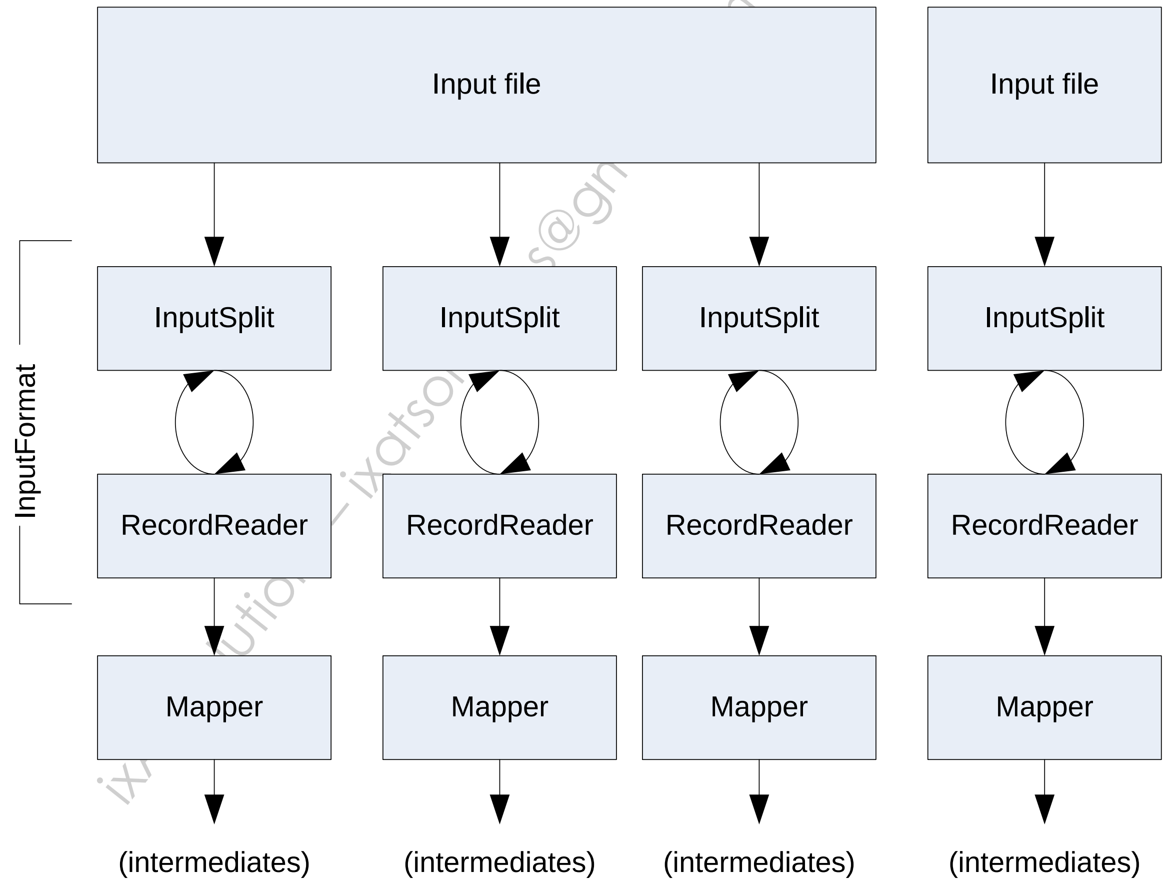
IO FORMATS, PARTITIONING AND JOB CHAINING

MR Recap



A α
B β
C γ
D δ
E ε
F ζ
mapper
mapper
mapper
mapper
a 1 b 2
c 3 c 6
a 5 c 2
b 7 c 8
Shuffle and Sort: aggregate values by keys
a 1 5
b 2 7
c 2 9 8
reducer
reducer
reducer
X 5
Y 7
Z 9

Getting Data To The Mapper



Reading Data

- 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

FileInputFormat and Friends

- *TextInputFormat* – Treats each '\n'-terminated line of a file as a value
- *KeyValueTextInputFormat* – Maps '\n'-terminated text lines of "k SEP v"
- *SequenceFileInputFormat* – Binary file of (k, v) pairs with some add'l metadata
- *SequenceFileAsTextInputFormat* – Same, but maps (k.toString(), v.toString())

Record Readers

- Each *InputFormat* provides its own *RecordReader* implementation
- *LineRecordReader* – Reads a line from a text file
- *KeyValueRecordReader* – Used by *KeyValueTextInputFormat*

The Gist

- As the Map operation is parallelized the input file set is first split to several pieces called FileSplits.
- Then a new map task is created per FileSplit.
- When an individual map task starts it will open a new output writer per configured reduce task.
- It will then proceed to read its FileSplit using the RecordReader it gets from the specified InputFormat.
- InputFormat parses the input and generates key-value pairs.
- As key-value pairs are read from the RecordReader they are passed to the configured Mapper.
- The user supplied Mapper does whatever it wants with the input pair and calls context.write with key-value pairs of its own choosing.
- The Map output is written into a SequenceFile.

OutputFormat

- Analogous to *InputFormat*
- *TextOutputFormat* – Writes "key val\n" strings to output file
- *SequenceFileOutputFormat* – Uses a binary format to pack (k, v) pairs
- *NullOutputFormat* – Discards output
  - Only useful if defining own output methods within reduce()

Combiners

- When *maps* produce many repeated keys
  - It is often useful to do a local aggregation following the *map*
  - Done by specifying a *Combiner*
  - Goal is to decrease size of the transient data
  - Combiners have the same interface as Reduces, and often are the same class
  - Combiners must ***not produce functional*** side effects, because they run an intermediate number of times
  - API ➔ *conf.setCombinerClass(Reduce.class);*

Partitioners

- Partitioners are application code that define how keys are assigned to reducers
- Default partitioning spreads keys evenly, but randomly
  - Uses *key.hashCode() % num_reduces*
- Custom partitioning is often required, for example, to produce a total order in the output
  - Should implement *Partitioner* interface
  - Set by calling conf.setPartitionerClass(MyPart.class)
  - To get a total order, sample the map output keys and pick values to divide the keys into roughly equal buckets and use that in your partitioner

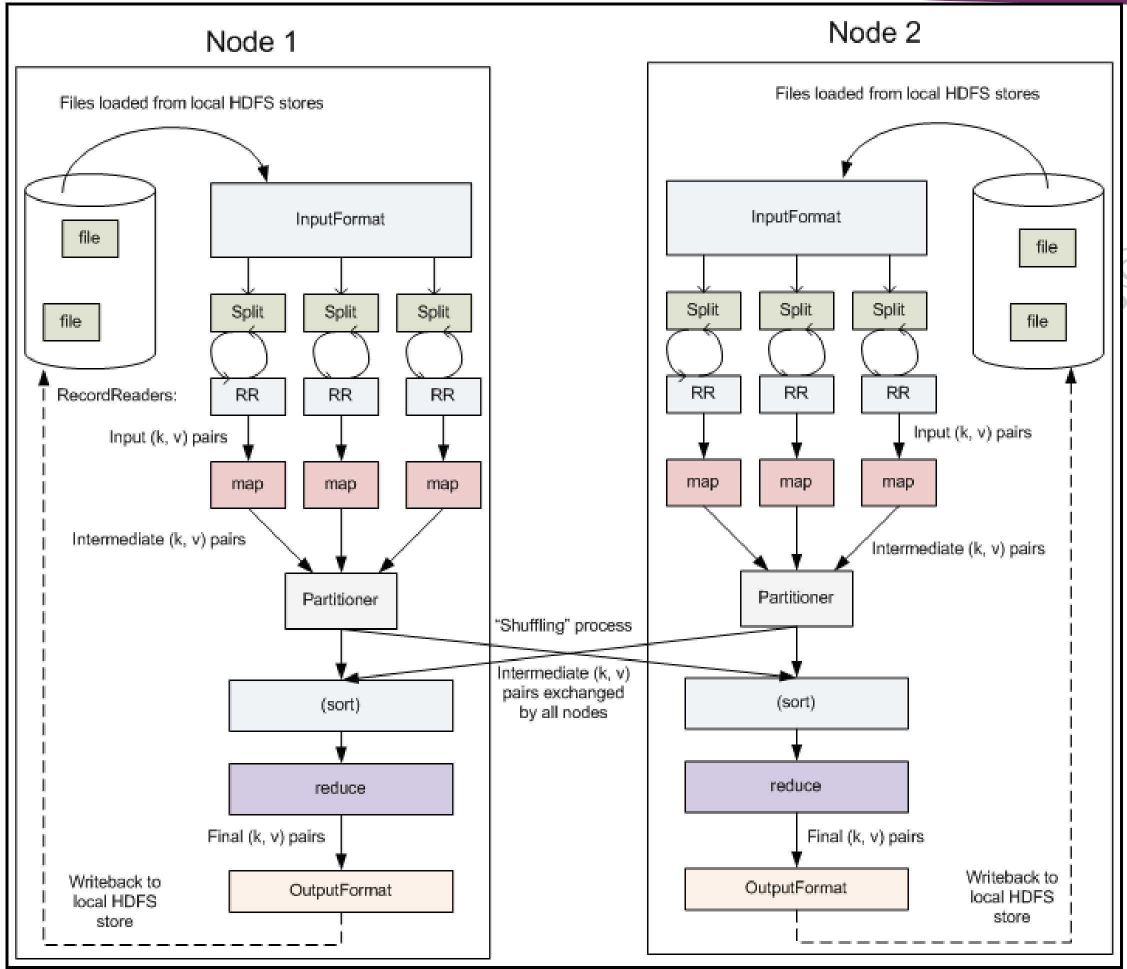
Why Partitioner

- The default partitioner only considers the key and ignores the value
- Therefore, a roughly-even partitioning of the key space may nevertheless yield large differences in the number of key-values pairs sent to each reducer
- This imbalance in the amount of data associated with each key is relatively common in many text processing applications due to the Zipfian distribution of word occurrences.

  https://en.wikipedia.org/wiki/Zipf%27s_law

- One need to implement a partitioner to balance the load across reducers.

Combiner & Partitioner and other steps in a MR app



```
Inputformat -> Map -> Combiner
                          |
                          |
                          V
                       Partitioner
                          |
                          |
                          V
  Sort <-- Shuffle <--   Sort
   |
   |
   V
 Reduce -> OutputFormat
```

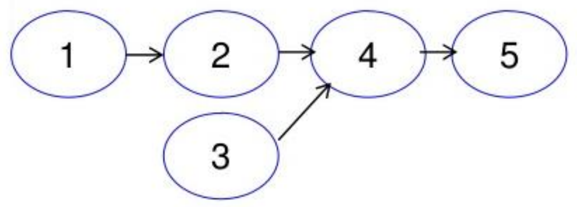
Counters

- Often Map/Reduce applications have countable events
- For example, framework counts records in to and out of Mapper and Reducer
- To define user counters:

```
enum Counter {EVENT1, EVENT2};
reporter.incrCounter(Counter.EVENT1, 1);
```

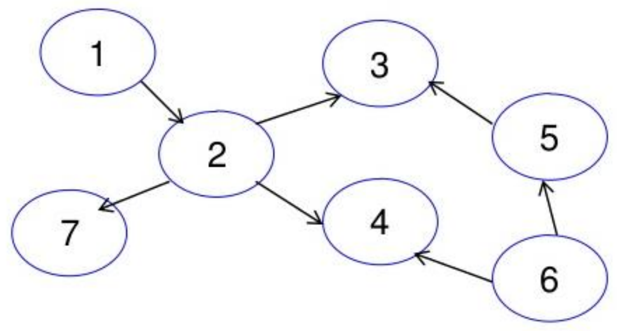
Job Chaining

- Sometimes, you may want to break down a bigger and complex problem into smaller Jobs and chain them in succession.
- Small Jobs are easy to maintain, scale and re-use
- Many of a times you may want to build a workflow out of composable and componentized jobs



Simple Dependency
or
Linear chain

VS.



Directed Acyclic Graph (DAG)

https://en.wikipedia.org/wiki/Directed_acyclic_graph

Using JobControl (Pseudo Code)

```
JobControl control = new JobControl("JobControl-Example");
Job job1 = Job.getInstance(conf);
job1.setJobName("job1-thefirst");
...
Job job2 = Job.getInstance(conf);
job2.setJobName("job2-theSecond");
...

ControlledJob step1 = new ControlledJob( job1, null);
ControlledJob step2 = new ControlledJob( job2, Arrays.asList(step1));

control.addJob(step1);
control.addJob(step2);
```

ChainMapper & ChainReducer

- The ChainMapper class allows to use multiple Mapper classes within a single Map task.
- The Mapper classes are invoked in a chained (or piped) fashion, the output of the first becomes the input of the second, and so on until the last Mapper, the output of the last Mapper will be written to the task's output.

- The key functionality of this feature is that the Mappers in the chain do not need to be aware that they are executed in a chain. This enables having reusable specialized Mappers that can be combined to perform composite operations within a single task.

ChainMapper & ChainReducer

- The ChainReducer class allows to chain multiple Mapper classes after a Reducer within the Reducer task.
- Special care has to be taken when creating chains that the key/values output by a Mapper are valid for the following Mapper in the chain. It is assumed all Mappers and the Reduce in the chain use matching output and input key and value classes as no conversion is done by the chaining code.

- Using the ChainMapper and the ChainReducer classes is possible to compose Map/Reduce jobs that look like [MAP+ / REDUCE MAP*]. And immediate benefit of this pattern is a dramatic reduction in disk IO.

ChainMapper & ChainReducer – Pseudo Code

```
conf.setJobName("chain");
conf.setInputFormat(TextInputFormat.class);
conf.setOutputFormat(TextOutputFormat.class);

JobConf mapAConf = new JobConf(false);
 ...
ChainMapper.addMapper(conf, AMap.class, LongWritable.class, Text.class, Text.class, Text.class, true, mapAConf);

JobConf mapBConf = new JobConf(false);
 ...
ChainMapper.addMapper(conf, BMap.class, Text.class, Text.class, LongWritable.class, Text.class, false, mapBConf);

JobConf reduceConf = new JobConf(false);
 ...
ChainReducer.setReducer(conf, XReduce.class, LongWritable.class, Text.class, Text.class, Text.class, true, reduceConf);
ChainReducer.addMapper(conf, CMap.class, Text.class, Text.class, LongWritable.class, Text.class, false, null);
ChainReducer.addMapper(conf, DMap.class, LongWritable.class, Text.class,LongWritable.class, LongWritable.class, true, null);

 FileInputFormat.setInputPaths(conf, inDir);
 FileOutputFormat.setOutputPath(conf, outDir);
 ...

 JobClient jc = new JobClient(conf);
 RunningJob job = jc.submitJob(conf);
 ...
```

Zero Reducers

- Frequently, we only need to run a filter on the input data
  - No sorting or shuffling required by the job
  - Set the number of reduces to 0
  - Output from maps will go directly to OutputFormat and disk