MAP REDUCE

Senthil Kumar A

Introduction

- A High level abstracted framework for distributed processing of large datasets
  - Fault Tolerant , Parallelization
- Computation consists of two phases
  - Map
  - Reduce
- A Master-Slaves architecture
- Computations occurs in multiple slave nodes
- And it tries to provide data locality as much as possible.

Daemons

- JobTracker
  - Client submits the computation to JobTracker
  - Assign a task to the TaskTracker who has free slots and where data is stored if possible
    - It tries to provide data locality as much as possible.
- TaskTracker
  - Spawns a JVM process for each input split as directed by Job Tracker
  - Send periodic heartbeats to Job Tracker

Terminology

- Job
  - A complete user defined computation or program
- Tasks
  - A subset of computation
  - Can be either execution of MAP or REDUCE
- Task Attempt
  - An attempt to run a task.
  - If an attempt fails, Job Tracker tries to start an another task attempt for the same task.
  - By Default, total number of task attempts for a task is four

Anatomy of MR code

- Mapper - a Java class to be extended by the developer
  - Methods – setup, map, run, cleanup
  - Map method takes a key value and can emit zero or more intermediate key value pairs depending upon the logic implemented by the developer
  - A JVM running Mapper is launched for each input split.
- Reducer – a Java class to be extended by the developer
  - Methods – setup, reduce, run, cleanup
  - Reduce method takes a (intermediate key-list of values) and can emit zero or more key value pairs depending upon the logic implemented by the developer
- Driver
  - Configures the job and submits the job to the cluster from the client.

Writables

What is Serialization??

the process of converting a data structure or object state into a format that can be stored (for example, in a file or memory buffer, or transmitted across a network connection link) and "resurrected" later in the same or another computer environment

- Writables in Hadoop are present for serialization

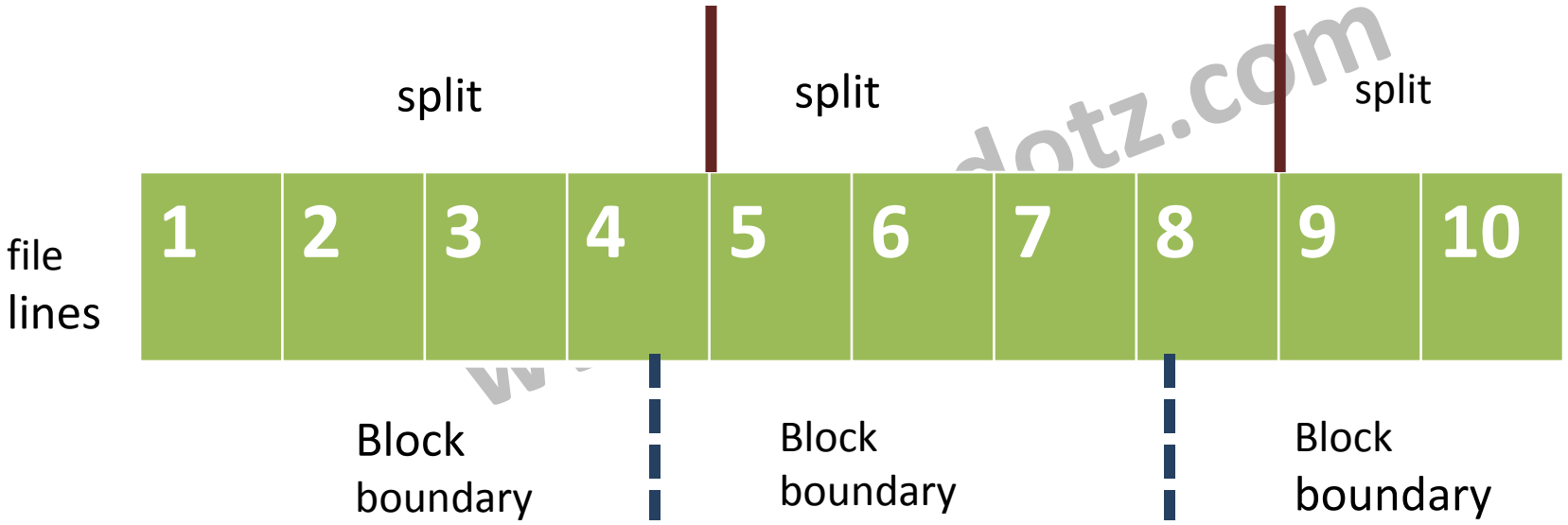
Why a separate framework instead of java serialization?

1. *Compact*
2. *Fast*
3. *Extensible*
4. *Interoperable*

java	Writable implementation	Serialized size (bytes)
byte	ByteWritable	1
Boolean	BooleanWritable	1
Int	IntWritable	4
	VIntWritable	1-5
Float	FloatWritable	4
Double	DoubleWritable	8

NullWritable --> usage *NullWritable.get()*
ArrayWritable
ArrayPrimitiveWritable
TwoDArrayWritable
MapWritable - used Widely ... Look @ Map or HashMap
SortedMapWritable
EnumSetWritable

Block vs. split



Partitioner

- It decides which key (with its associated value) goes to reducer.
- By default its HashPartitioner.
  - hashes a record's key to determine which partition (and thus which reducer) the record belongs in

```
public class HashPartitioner<K, V> extends Partitioner<K, V> {

    public int getPartition(K key, V value, int numReduceTasks) {
        return (key.hashCode() & Integer.MAX_VALUE) % numReduceTasks;
    }
}
```

- Number of Partitions is equal to the number of reducers.

Custom Partitioner

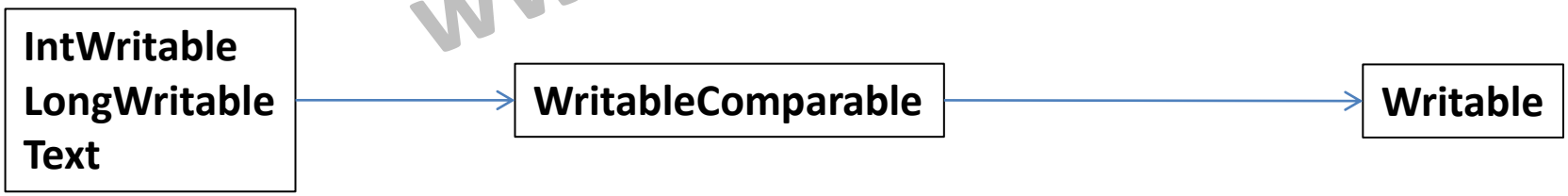
- Need Customer partitioner for better load balancing(performance)
- To Write a partitioner, follow the steps
  - Extend **partitioner** class
  - Override the method **getPartition**
    - input – key,value,number of Reducers
    - Output –0 to n-1(where n - number of reducer)
  - job.setPartitionerClass(<yourclassname.class>)

Combiner

- To Reduce the intermediate data from mapper to reducer
  - To reduce Network IO and Disk IO
- Runs on a single mapper output(like a mini-reducer)
- Extends the Reducer class (new API)
- Job.setCombinerClass(*.class);
- Combiner may or maynot run
- Its better to use identical combiner and reducer when both are commutative as well as associative functions

WritableComparable

- A WritableComparable is a Writable which is also Comparable
  - Two WritableComparables can be compared against each other to determine their 'order'
  - Keys must be WritableComparables because they are passed to the Reducer in sorted order
  - We will talk more about WritableComparable later



Defines a sort order. All keys must be WritableComparable

Defines a de/serialization protocol. Every data type in Hadoop is a Writable

WritableComparable

- WritableComparable is a sub-interface of Writable
  - Writable is an interface and must inherit following methods
    - readFields(DataInput in);
    - write(DataOutput out);
  - Must implement compareTo, hashCode, equals methods
  - All keys in MapReduce must be WritableComparable
  - compareTo method compares the keys in the mapper out to provide sorted
    - It deserializes the keys and compare the values in java
    - It is time consuming .instead you can use comparators

Streaming API

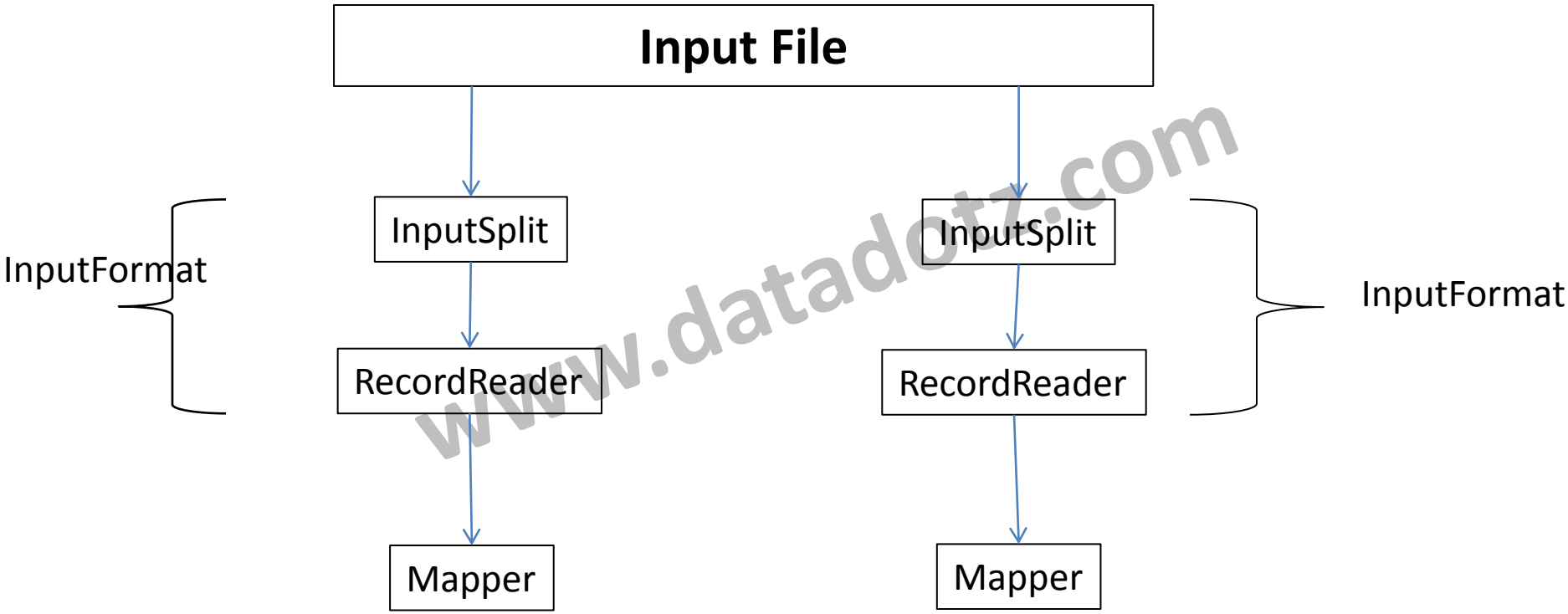
- To use other languages for writing MR
- Python, perl, ruby
- RAD, use of existing libraries
- Use stdin, stdout for input & output respectively
- TextInputFormat –default -- without key
- MR emits Key(tab)value
- No iterators as that of Java- developers have to make sure to detect change in key

bin/hadoop jar contrib/streaming/hadoop-streaming-1.0.4.jar -file /hadoop/hadoop-docs/mapper.py -mapper /hadoop/hadoop-docs/mapper.py -file /hadoop/hadoop-docs/reducer.py -reducer /hadoop/hadoop-docs/reducer.py -input /data_30lac.txt -output /python_out

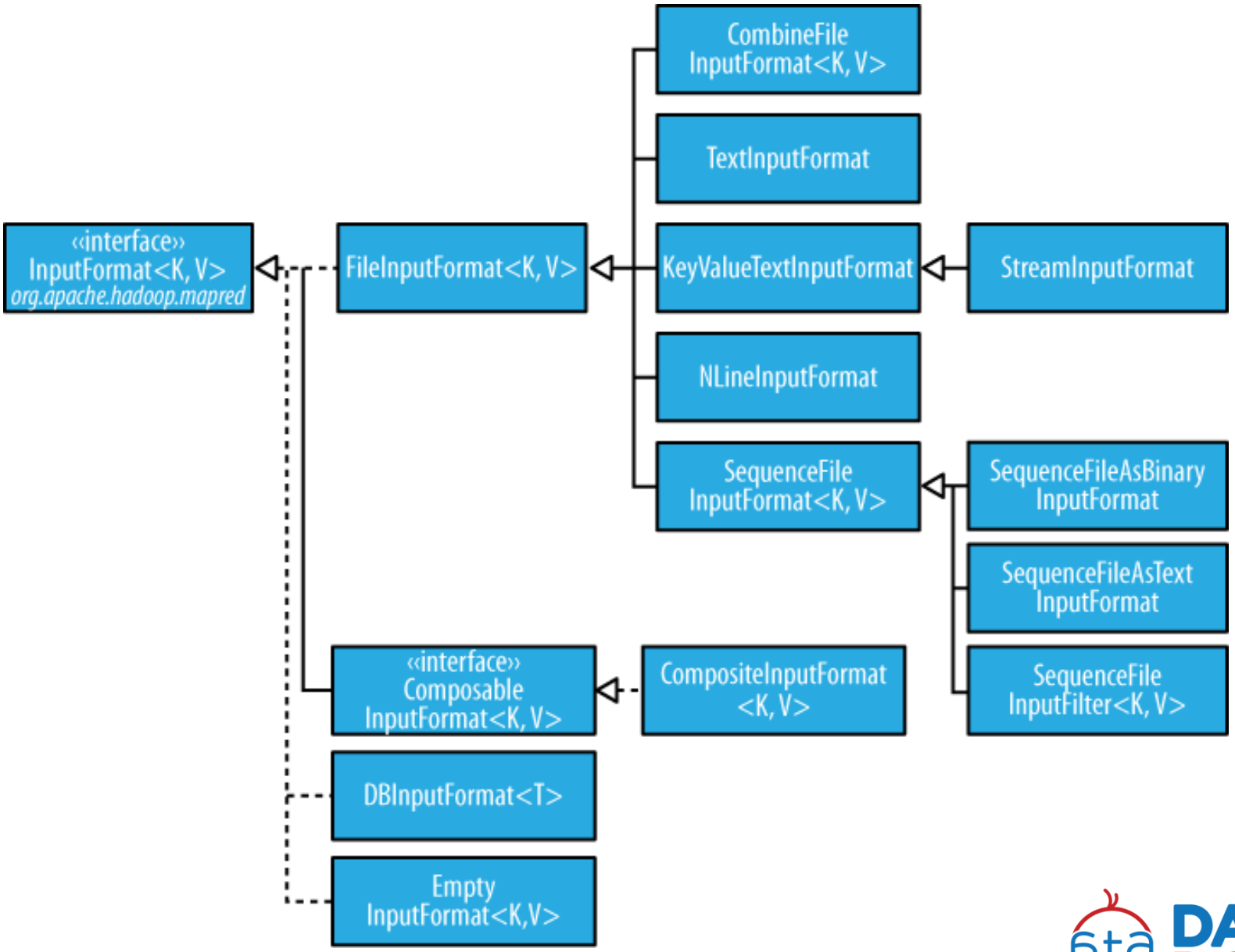
To Change Input Split Size

mapred.min.split.size	mapred.max.split.size	dfs.block.size	Split Size
1 (default)	Long.MAX_VALUE (default)	64 MB (default)	64 MB
1 (default)	Long.MAX_VALUE (default)	128 MB	128 MB
128 MB	Long.MAX_VALUE (default)	64 MB (default)	128 MB
1 (default)	10 MB	64 MB (default)	10 MB

InputFormat



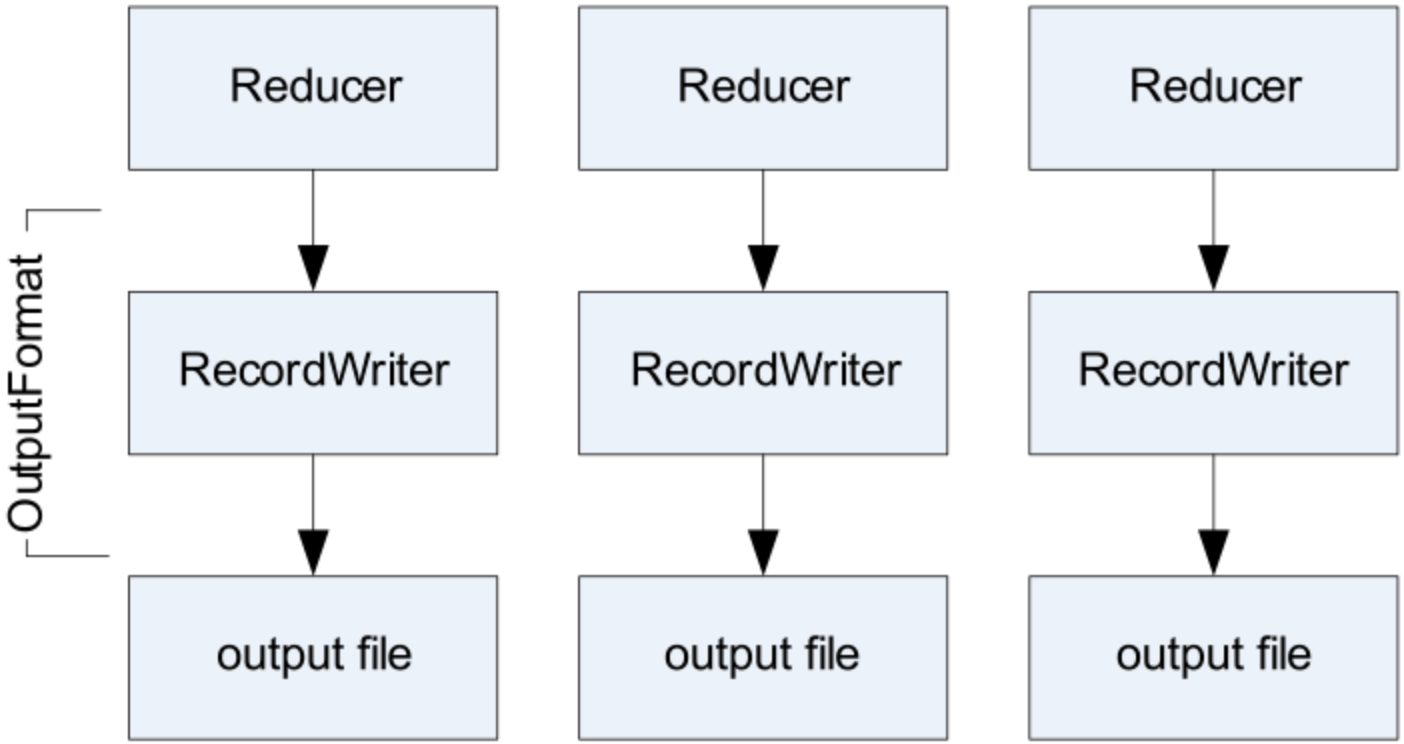
Responsible for creating inputsplit and dividing them into records

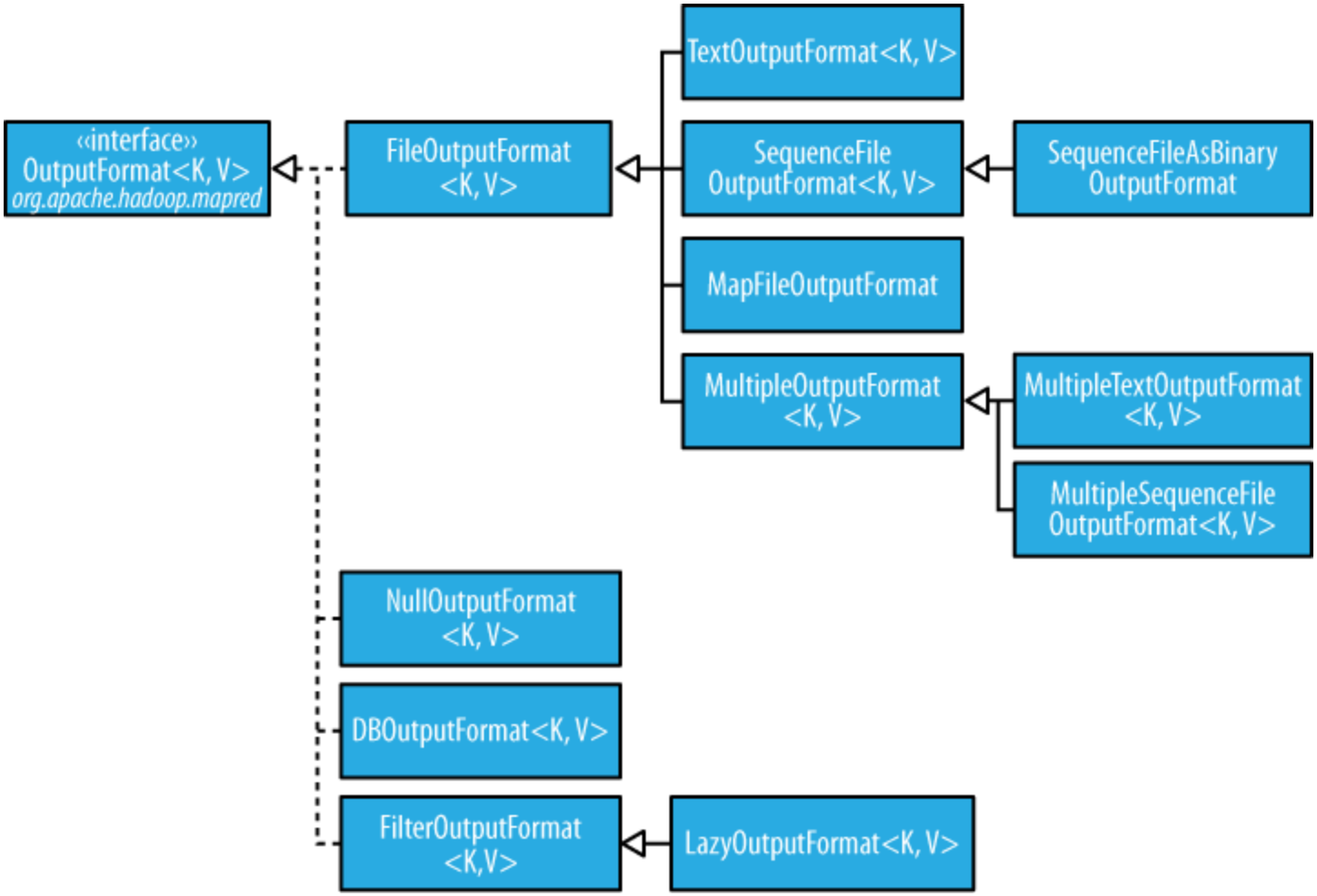

«interface»
InputFormat<K, V>
org.apache.hadoop.mapred
FileInputFormat<K, V>
CombineFile
InputFormat<K, V>
TextInputFormat
KeyValueTextInputFormat
StreamInputFormat
NLineInputFormat
SequenceFile
InputFormat<K, V>
SequenceFileAsBinary
InputFormat
SequenceFileAsText
InputFormat
SequenceFile
InputFilter<K, V>
«interface»
Composable
InputFormat<K, V>
CompositeInputFormat
<K, V>
DBInputFormat<T>
Empty
InputFormat<K,V>

InputFormat	Key	Value
TextInputFormat	Byte offset of the line	Line contents
KeyValueInputFormat	Everything up to the first tab charcter	The remainder of the line
SequenceFileInputFormat	User-defined	User-defined
WholeFileInputFormat	NullWritable	file contents
NLineInputFormat	Byte offset of the line	*n* Number of lines
MultipleInputs	Per path basis	Per Path Basis
TableInputFormat (HBase)	Rowkey	Value

mapred.line.input.format.linespermap

OutPutFormat




«interface»
OutputFormat<K, V>
org.apache.hadoop.mapred
FileOutputFormat
<K, V>
TextOutputFormat<K, V>
SequenceFile
OutputFormat<K, V>
SequenceFileAsBinary
OutputFormat
MapFileOutputFormat
MultipleOutputFormat
<K, V>
MultipleTextOutputFormat
<K, V>
MultipleSequenceFile
OutputFormat<K, V>
NullOutputFormat
<K, V>
DBOutputFormat<K, V>
FilterOutputFormat
<K,V>
LazyOutputFormat<K, V>

CustomInputFormat

- Use FileInputFormat as a starting point
  - – Extend it
- Write your own custom RecordReader
- Override getRecordReader method in FileInputFormat
- Override isSplittable if you don't want input files to be split

Some More info

- IdentityMapper
  - mapping inputs directly to outputs
- IdentityReducer
- Performs no reduction, writing all input values directly to the output.
- Single Reducer
  - Use when complete sort order is required
- Zero Reducer
  - SetNumReduceTasks to 0
  - Output from maps will go directly to OutputFormat and disk
  - No Sorting and Shuffling

Counters

- for gathering statistics about the job
  - for quality control
  - for application level-statistics
- Classified into two counters
  - Built In Counters
    - Task counters
    - Job Counters
  - Custom Counters

Built in Counter for JOB

	Counter	Map	Reduce	Total
Job Counters	SLOTS_MILLIS_MAPS	0	0	4,425,106
	Total time spent by all reduces waiting after reserving slots (ms)	0	0	0
	Total time spent by all maps waiting after reserving slots (ms)	0	0	0
	Rack-local map tasks	0	0	189
	Launched map tasks	0	0	337
	Data-local map tasks	0	0	148
	SLOTS_MILLIS_REDUCES	0	0	0
File Output Format Counters	Bytes Written	0	0	0
FileSystemCounters	HDFS_BYTES_READ	88,533	0	88,533
	FILE_BYTES_WRITTEN	33,492,676	0	33,492,676
	HDFS_BYTES_WRITTEN	13,932	0	13,932
File Input Format Counters	Bytes Read	0	0	0
	Map input records	774	0	774
	Physical memory (bytes) snapshot	34,978,086,912	0	34,978,086,912
	Spilled Records	0	0	0

Custom Counter

```
public static enum CUSTOMCOUNTER {
        ERROR_COUNT
}
```

To insert the code in JOB class

To increase the counter value in mapper class

```
context.getCounter(CUSTOMCOUNTER.ERROR_COUNT).increment(1);
```

To display the output of the counter in job after completion

```
job.getCounters().findCounter(CUSTOMCOUNTER.ERROR_COUNT).getValue()
```

Side Data distribution

- To keep some read only data available for all the tasks
- Can be achieved using following two ways:
  - Configuration Object
  - DistributedCache

```
Configuration conf = new Configuration();
conf.set("personName","kumar");
```

Distributed Cache

- Cache files and archive to the task nodes
- Usage
  - As
  - DistributedCache API
    - *public void addCacheFile(URI uri)*
    - *public void addCacheArchive(URI uri)*
    - *public void setCacheFiles(URI[] files)*
    - *public void setCacheArchives(URI[] archives)*
    - *public void addFileToClassPath(Path file)*
    - *public void addArchiveToClassPath(Path archive)*
    - *public void createSymlink()*

GenericOptionsParser, Tool, and ToolRunner

- class that interprets common Hadoop command-line options
- implement the Tool interface and run your application with the ToolRunner
- GenericOptionsParser
  - -D*property=value*
  - *-conf file*
  - *-fs uri*
  - *-jt host:port*
  - *-files file1 file2*
  - *-archives archive1 archive2*
  - *-libjars jar1*

Joins

- To join data from Multiple datasets
- Please try to use PIG or HIVE join if you are using text based files
- Two varieties or approaches
  - Map-side Join
  - Reduce side Join

Map Side Join

- **Basic idea for Map-side joins:**
  - Load one set of data into memory, stored in an associative array
  - Key of the associative array is the join key
  - Map over the other set of data, and perform a lookup on the associative array using the join key
  - If the join key is found, you have a successful join
  - Otherwise, do nothing

Reduce Side Join

- Use the same key for the mapper output
- Can be performed in two ways
  - MultipleInputs
  - Secondary Sort

MultipleInputs.addInputPath(job, InputPath, TextInputFormat.class, CustomMapper.class);

Sorting

- In MR, keys from all mappers are sent to the reducers in sorting order
- Total sorting can be obtained using single reducer
  - Reduces the performance
- Partial sorting can be obtained using partition
- For benchmarking , terasort has been widely used

Secondary Sorting

- In MR sorting , the keys are sorted and not the values
- To achieve the secondary sorting
  - Make the key a composite of the natural key and the natural value
  - The sort comparator should order by the composite key, that is, the natural key *and* natural value.
  - The partitioner and grouping comparator for the composite key should consider only the natural key for partitioning and grouping.
- Example

Searching

Assignment

- Input
  - A set of input files containing lines
  - A pattern
- OutPut
  - Pattern with list of filename containing the pattern

Solution:

1. Set the pattern using Configuration Object
2. In Mapper(with TextInputFormat), verify for the pattern
3. If Pattern matches, emit (pattern, filename)
4. If No pattern , emit nothing
5. Zero reducer

map.input.file

LocalJobRunner

- only designed for simple testing of MapReduceprograms
- can't run more than one reducer
- *bin/hadoop jar job.jar com.example.wordcount -D mapred.job.tracker=local -D fs.default.name=file:/// (args)*

- *Other tips*
  - Set *keep.failed.task.files* to true
  - *Use the isolationRunner to run just the failed tasks*

Thank you

DATA DOTZ