Prof R. Madana Mohana

BIG DATA ANALYTICS

MapReduce Types & Formats

Input & Output Formats

https://www.youtube.com/c/RASINENIMADANAMOHANA

Input and Output Formats: *Outline*

Input Formats:

- Input Splits and Records
- Text Input

Output Formats:

- **Text Output**

Input Formats

Input Formats

- **Hadoop** can process many different types of data formats, from flat text files to databases.
- **Input Splits and Records**
  - FileInputFormat
  - FileInputFormat input paths
  - FileInputFormat input splits
  - Small files and CombineFileInputFormat
  - Preventing splitting
  - File information in the mapper
  - Processing a whole file as a record

Input Formats

- **Text Input**
  a. TextInputFormat
  b. KeyValueTextInputFormat
  c. NLineInputFormat
  d. XML

Input Formats:

Input Splits and Records

Input Splits and Records

- An input split is a chunk of the input that is processed by a single **map**.
- Each **map** processes a single split.
- Each split is divided into records, and the **map** processes each record-a **key-value** pair-in turn.
- Splits and records are logical: there is nothing that requires them to be tied to files.

Input Splits and Records

- In a database context, a split might correspond to a range of rows from a table and a record to a row in that range (this is precisely the case with **DBInputFormat**, which is an input format for reading data from a relational database).

Input Splits and Records

- Input splits are represented by the Java class **InputSplit** (which, like all of the classes mentioned in the **org.apache.hadoop.mapreduce** package):

```
public abstract class InputSplit {
  public abstract long getLength() throws
IOException, InterruptedException;
  public abstract String[] getLocations() throws
IOException, InterruptedException;
}
```

Input Splits and Records

- An **InputSplit** has a length in bytes and a set of storage locations, which are just hostname strings.
- Notice that a split doesn't contain the input data; it is just a reference to the data.
- The storage locations are used by the **MapReduce** system to place **map** tasks as close to the split's data as possible, and the size is used to order the splits so that the largest get processed *first*, in an attempt to minimize the job *runtime* (this is an instance of a greedy approximation algorithm).

Input Splits and Records

- As a **MapReduce** application writer, we don't need to deal with **InputSplits** directly, as they are created by an **InputFormat** (an **InputFormat** is responsible for creating the input splits and dividing them into records).

Input Splits and Records

- Example of **InputFormats**, how it is used in **MapReduce**. Here's the interface:

```
public abstract class InputFormat<K, V> {
  public abstract List<InputSplit> getSplits(JobContext
context) throws IOException, InterruptedException;
  public abstract RecordReader<K, V>
createRecordReader(InputSplit split, TaskAttemptContext
context) throws IOException, InterruptedException;
}
```

Input Splits and Records

- Mapper's **run()** method:

```
public void run(Context context) throws IOException,
InterruptedException {
   setup(context);
   while (context.nextKeyValue()) {
map(context.getCurrentKey(), context.getCurrentValue(),
context);
   }
   cleanup(context);
}
```

Input Splits and Records

- After running **setup()**, the **nextKeyValue()** is called repeatedly on the **Context** (which delegates to the identically named method on the **RecordReader**) to populate the key and value objects for the **mapper**.
- The key and value are retrieved from the **RecordReader** by way of the **Context** and are passed to the **map()** method for it to do its work.
- When the reader gets to the end of the stream, the **nextKeyValue()** method returns false, and the **map** task runs its **cleanup()** method and then completes.

Input Splits and Records

- Finally, **note** that the **Mapper**'s **run()** method is public and may be customized by users.
- **MultithreadedMapper** is an implementation that runs mappers concurrently in a configurable number of threads (set by **mapreduce.mapper.multithreadedmapper.threads**).
- For most data processing tasks, it confers no advantage over the default implementation.
- However, for mappers that spend a long time processing each record -because they contact external servers, for example-it allows multiple mappers to run in one **JVM** with little contention.

Input Splits and Records: *FileInputFormat*

- **FileInputFormat** is the base class for all implementations of **InputFormat** that use files as their data source.

Input Splits and Records: *FileInputFormat*

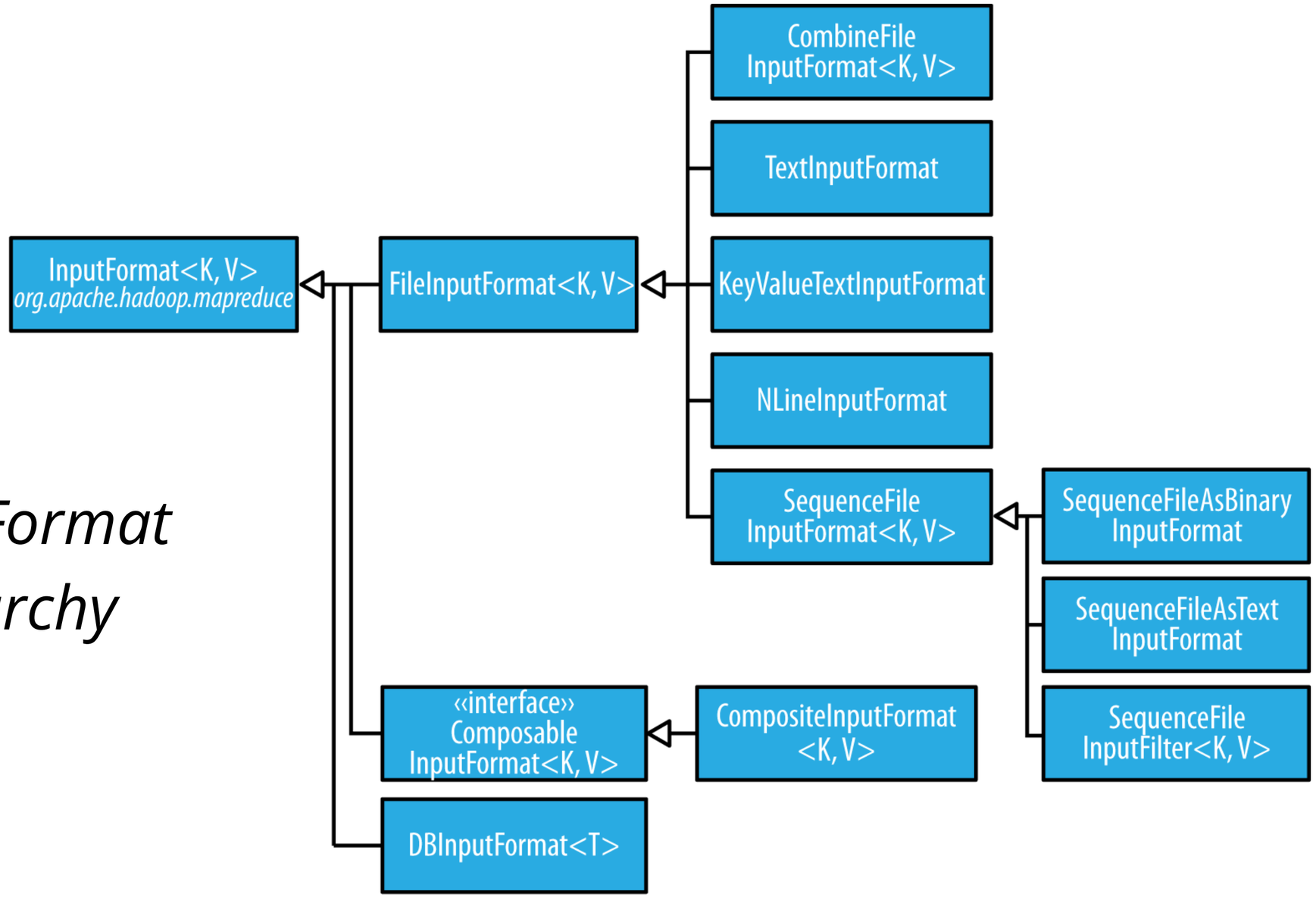

Fig: *InputFormat class hierarchy*

Input Splits and Records: *FileInputFormat*

It provides two things:

1. A place to define which files are included as the input to a job, and an implementation for generating splits for the input files.
2. The job of dividing splits into records is performed by subclasses.

Input Splits and Records: *FileInputFormat input paths*

- The input to a job is specified as a collection of paths, which offers great flexibility in constraining the input.
- **FileInputFormat** offers four static convenience methods for setting a **Job**'s input paths:

```
1. public static void addInputPath(Job job, Path path)
2. public static void addInputPaths(Job job, String
   commaSeparatedPaths)
3. public static void setInputPaths(Job job, Path...
   inputPaths)
4. public static void setInputPaths(Job job, String
   commaSeparatedPaths)
```

Input Splits and Records: *FileInputFormat input paths*

- The **addInputPath()** and **addInputPaths()** methods add a path or paths to the list of inputs. We can call these methods repeatedly to build the list of paths.
- The **setInputPaths()** methods set the entire list of paths in one go (replacing any paths set on the Job in previous calls).
- A path may represent a file, a directory, or, by using a glob, a collection of files and directories.
- A path representing a directory includes all the files in the directory as input to the job.

Input Splits and Records: *FileInputFormat input paths*

- The add and set methods allow files to be specified by inclusion only.
- To exclude certain files from the input, we can set a filter using the **setInputPathFilter()** method on **FileInputFormat**.

Input Splits and Records: *FileInputFormat input paths*

- Even if we don't set a filter, **FileInputFormat** uses a default filter that excludes hidden files (those whose names begin with a dot or an underscore).
- If we set a filter by calling **setInputPathFilter()**, it acts in addition to the default filter. In other words, only nonhidden files that are accepted by our filter get through.

Input Splits and Records: *FileInputFormat input paths*

- Paths and filters can be set through configuration properties, too (Shown in *below Table*), which can be handy for Streaming jobs.
- Setting paths is done with the **-input** option for the Streaming interface, so setting paths directly usually is not needed.

Input Splits and Records: *FileInputFormat input paths*

Table. *Input path and filter properties*

Property name	Type	Default value	Description
mapreduce.input.fileinputformat.inputdir	Comma-separated paths	None	The input files for a job. Paths that contain commas should have those commas escaped by a backslash character. For example, the glob **{a,b}** would be escaped as **{a\,b}.**
mapreduce.input.pathFilter.class	PathFilter classname	None	The filter to apply to the input files for a job.

Input Splits and Records: *FileInputFormat input splits*

Given a set of files, how does **FileInputFormat** *turn them into splits?*

- **FileInputFormat** splits only large files-here, **"large"** means larger than an **HDFS block**.
- The split size is normally the size of an **HDFS block**, which is appropriate for most applications; however, it is possible to control this value by setting various **Hadoop properties**, as shown in *below Table*.

Input Splits and Records: *FileInputFormat input splits*

Table. *Properties for controlling split size*

Property name	Type	Default value	Description
mapreduce.input.fileinputformat.split.minsize	Int	1	The smallest valid size in bytes for a file split.
mapreduce.input.fileinputformat.split.maxsize	long	Long.MAX_VALUE (i.e., 9223372036854775807)	The largest valid size in bytes for a file split.
dfs.blocksize	long	128 MB (i.e., 134217728)	The size of a block in HDFS in bytes.

Input Splits and Records: *FileInputFormat input splits*

- The split size is calculated by the following formula:

```
max(minimumSize, min(maximumSize, blockSize))
```

- and by default:

```
minimumSize < blockSize < maximumSize
```

- so the split size is blockSize.
- Various settings for these parameters and how they affect the final split size are illustrated in *below Table*:

Input Splits and Records: *FileInputFormat input splits*

Table. *Examples of how to control the split size*

Minimum split size	Maximum split size	Block size	Split size	Comment
1 (default)	Long.MAX_VALUE (default)	128 MB (default)	128 MB	By default, the split size is the same as the default block size.
1 (default)	Long.MAX_VALUE (default))	256 MB	256 MB	The most natural way to increase the split size is to have larger blocks in HDFS, either by setting **dfs.blocksize** or by configuring this on a per-file basis at file construction time.
256 MB	Long.MAX_VALUE (default)	128 MB (default)	256 MB	Making the minimum split size greater than the block size increases the split size, but at the cost of locality.
1 (default)	64 MB	128 MB (default)	64 MB	Making the maximum split size less than the block size decreases the split size.

Input Splits and Records: *Small files and CombineFileInputFormat*

- **Hadoop** works better with a small number of large files than a large number of small files.
- One reason for this is that **FileInputFormat** generates splits in such a way that each split is all or part of a single file.

Input Splits and Records: *Small files and CombineFileInputFormat*

- If the file is very small (**"small"** means significantly smaller than an **HDFS block**) and there are a lot of them, each **map task** will process very little input, and there will be a lot of them (one per file), each of which imposes extra bookkeeping overhead.
- Compare a **1 GB** file broken into eight **128 MB** blocks with **10,000** or so **100 KB** files.
- The **10,000** files use one map each, and the job time can be tens or hundreds of times slower than the equivalent one with a single input file and eight map tasks.

Input Splits and Records: *Small files and CombineFileInputFormat*

- The situation is reduced somewhat by **CombineFileInputFormat**, which was designed to work well with small files.
- Where **FileInputFormat** creates a split per file, **CombineFileInputFormat** packs many files into each split so that each mapper has more to process.
- Crucially, **CombineFileInputFormat** takes node and rack locality into account when deciding which blocks to place in the same split, so it does not compromise the speed at which it can process the input in a typical **MapReduce job**.

Input Splits and Records: *Preventing splitting*

- Some applications don't want files to be split, as this allows a single mapper to process each input file in its entirety.
- For example, a simple way to check if all the records in a file are sorted is to go through the records in order, checking whether each record is not less than the preceding one. Implemented as a map task, this algorithm will work only if one map processes the whole file.

Input Splits and Records: *Preventing splitting*

- There are a couple of ways to ensure that an existing file is not split.
- The *first* (quickand-dirty) way is to increase the minimum split size to be larger than the largest file in our system. Setting it to its maximum value, **Long.MAX_VALUE**, has this effect.
- The *second* is to subclass the concrete subclass of **FileInputFormat** that we want to use, to override the **isSplitable()** method to return **false**.

Input Splits and Records: *Preventing splitting*

For example, here's a nonsplittable **TextInputFormat**:

```
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.mapreduce.JobContext;
import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;
public class NonSplittableTextInputFormat extends
TextInputFormat {
  @Override
  protected boolean isSplitable(JobContext context, Path
file) {
    return false;
  }
}
```

Input Splits and Records: *File information in the mapper*

- A **mapper** processing a file input split can find information about the split by calling the **getInputSplit()** method on the **Mapper**'s **Context** object.
- When the input format derives from **FileInputFormat**, the **InputSplit** returned by this method can be cast to a **FileSplit** to access the file information listed in *below Table*.

Input Splits and Records: *File information in the mapper*

Table. *File split properties*

FileSplit method	Property name	Type	Description
getPath()	mapreduce.map.input.file	Path/ String	The path of the input file being processed.
getStart()	mapreduce.map.input.start	long	The byte offset of the start of the split from the beginning of the file.
getLength()	mapreduce.map.input.length	long	The length of the split in bytes.

Input Splits and Records: *File information in the mapper*

- In the **old MapReduce API**, and the Streaming interface, the same file split information is made available through properties that can be read from the **mapper**'s configuration. (In the **old MapReduce API** this is achieved by implementing **configure()** in our **Mapper** implementation to get access to the **JobConf** object.)

Input Splits and Records: *Processing a whole file as a record*

- A related requirement that sometimes crops up is for **mappers** to have access to the full contents of a file.
- Not splitting the file gets part of the way there, but also need to have a **RecordReader** that delivers the file contents as the value of the record.

Input Splits and Records: *Processing a whole file as a record*

The listing for **WholeFileInputFormat** shown in *below Example-1*:

An InputFormat for reading a whole file as a record

```
public class WholeFileInputFormat extends
FileInputFormat<NullWritable, BytesWritable> {
 @Override
 protected boolean isSplitable(JobContext context,
Path file) {
    return false;
}
```

Input Splits and Records: *Processing a whole file as a record*

Example-1: **An InputFormat for reading a whole file as a record**

```
 @Override
 public RecordReader<NullWritable, BytesWritable>
createRecordReader(InputSplit split, TaskAttemptContext
context) throws IOException,
InterruptedException {
WholeFileRecordReader reader = new
WholeFileRecordReader();
reader.initialize(split, context);
   return reader;
}
}
```

Input Splits and Records: *Processing a whole file as a record*

Example-2*: *The RecordReader used by WholeFileInputFormat for reading a whole file as a record

```
class WholeFileRecordReader extends
RecordReader<NullWritable, BytesWritable> {
private FileSplit fileSplit;
private Configuration conf;
private BytesWritable value = new BytesWritable();
private boolean processed = false;
```

Input Splits and Records: *Processing a whole file as a record*

Example-2*: *The RecordReader used by WholeFileInputFormat for reading a whole file as a record

```
@Override
public void initialize(InputSplit split,
TaskAttemptContext context) throws IOException,
InterruptedException {
this.fileSplit = (FileSplit) split;
this.conf = context.getConfiguration();
}
```

Input Splits and Records: *Processing a whole file as a record*

Example-2*: *The RecordReader used by WholeFileInputFormat for reading a whole file as a record

```
@Override
public boolean nextKeyValue() throws IOException,
InterruptedException {
  if (!processed) {
    byte[] contents = new byte[(int)
fileSplit.getLength()];
    Path file = fileSplit.getPath();
    FileSystem fs = file.getFileSystem(conf);
    FSDataInputStream in = null;
```

Input Splits and Records: *Processing a whole file as a record*

Example-2*: *The RecordReader used by WholeFileInputFormat for reading a whole file as a record

```
    try {
      in = fs.open(file);
      IOUtils.readFully(in, contents, 0, contents.length);
      value.set(contents, 0, contents.length);
    } finally {
      IOUtils.closeStream(in);
    }
    processed = true;
    return true;
} return false; }
```

Input Splits and Records: *Processing a whole file as a record*

Example-2*: *The RecordReader used by WholeFileInputFormat for reading a whole file as a record

```
  @Override
  public NullWritable getCurrentKey() throws IOException,
InterruptedException {
    return NullWritable.get();
  }
  @Override
  public BytesWritable getCurrentValue() throws
IOException, InterruptedException {
    return value;
  }
```

Input Splits and Records: *Processing a whole file as a record*

Example-2*: *The RecordReader used by WholeFileInputFormat for reading a whole file as a record

```
  @Override
  public float getProgress() throws IOException {
    return processed ? 1.0f : 0.0f;
  }
  @Override
  public void close() throws IOException {
    // do nothing
  }
}
```

Input Splits and Records: *Processing a whole file as a record*

***Example-3*: A MapReduce program for packaging a collection of small files as a single SequenceFile**

```
public class SmallFilesToSequenceFileConverter extends
Configured implements Tool {
   static class SequenceFileMapper extends
Mapper<NullWritable, BytesWritable, Text,
BytesWritable> {
    private Text filenameKey;
```

Input Splits and Records: *Processing a whole file as a record*

Example-3*: *A MapReduce program for packaging a collection of small files as a single SequenceFile

```
    @Override
    protected void setup(Context context) throws
IOException, InterruptedException {
     InputSplit split = context.getInputSplit();
     Path path = ((FileSplit) split).getPath();
     filenameKey = new Text(path.toString());
    }
```

Input Splits and Records: *Processing a whole file as a record*

Example-3*: *A MapReduce program for packaging a collection of small files as a single SequenceFile

```
    @Override
    protected void map(NullWritable key, BytesWritable
value, Context context) throws IOException,
InterruptedException {
        context.write(filenameKey, value);
    }
}
```

Input Splits and Records: *Processing a whole file as a record*

Example-3*: *A MapReduce program for packaging a collection of small files as a single SequenceFile

```
    @Override
    public int run(String[] args) throws Exception {
      Job job = JobBuilder.parseInputAndOutput(this,
getConf(), args);
      if (job == null) {
        return -1;
      }
```

Input Splits and Records: *Processing a whole file as a record*

Example-3*: *A MapReduce program for packaging a collection of small files as a single SequenceFile

```
job.setInputFormatClass(WholeFileInputFormat.class);
job.setOutputFormatClass(SequenceFileOutputFormat.class);
job.setOutputKeyClass(Text.class);
job.setOutputValueClass(BytesWritable.class);
job.setMapperClass(SequenceFileMapper.class);
return job.waitForCompletion(true) ? 0 : 1;
}
```

Input Splits and Records: *Processing a whole file as a record*

Example-3*: *A MapReduce program for packaging a collection of small files as a single SequenceFile

```
public static void main(String[] args) throws Exception
{
   int exitCode = ToolRunner.run(new
SmallFilesToSequenceFileConverter(), args);
   System.exit(exitCode);
}
}
```

Input Splits and Records: *Processing a whole file as a record*

Example-3*: *A MapReduce program for packaging a collection of small files as a single SequenceFile

Here's a run on a few small files. We've chosen to use two reducers, so we get two output sequence files:

```
% hadoop jar hadoop-examples.jar
SmallFilesToSequenceFileConverter \
-conf conf/hadoop-localhost.xml -D mapreduce.job.reduces=2
\ input/smallfiles output
```

Input Splits and Records: *Processing a whole file as a record*

Example-3*: *A MapReduce program for packaging a collection of small files as a single SequenceFile

Two part files are created, each of which is a sequence file. We can inspect these with the **-text** option to the filesystem shell:

```
% hadoop fs -conf conf/hadoop-localhost.xml -text
output/part-r-00000
hdfs://localhost/user/tom/input/smallfiles/a 61 61 61 61
61 61 61 61 61 61
hdfs://localhost/user/tom/input/smallfiles/c 63 63 63 63
63 63 63 63 63 63
hdfs://localhost/user/tom/input/smallfiles/e
```

Input Splits and Records: *Processing a whole file as a record*

Example-3*: *A MapReduce program for packaging a collection of small files as a single SequenceFile

Two part files are created, each of which is a sequence file. We can inspect these with the **-text** option to the filesystem shell:

```
% hadoop fs -conf conf/hadoop-localhost.xml -text
output/part-r-00001
hdfs://localhost/user/tom/input/smallfiles/b 62 62 62 62 62
62 62 62 62 62
hdfs://localhost/user/tom/input/smallfiles/d 64 64 64 64 64
64 64 64 64 64
hdfs://localhost/user/tom/input/smallfiles/f 66 66 66 66 66
66 66 66 66 66
```

Input Formats:

Text Input

Text Input

- **Hadoop** excels at processing *unstructured text*.
- In this, we discuss the different **InputFormats** that **Hadoop** provides to process ***text***.

Text Input: *TextInputFormat*

- **TextInputFormat** is the default **InputFormat**.
- Each record is a line of input.
- The ***key***, a **LongWritable**, is the byte offset within the file of the beginning of the line.
- The ***value*** is the contents of the line, excluding any line terminators (e.g., *newline* or *carriage return*), and is packaged as a **Text** object.

Text Input: *TextInputFormat*

For Example, a file containing the following text:

```
On the top of the Crumpetty Tree
The Quangle Wangle sat,
But his face you could not see,
On account of his Beaver Hat.
```

is divided into one split of four records. The records are interpreted as the following ***key-value*** pairs:

```
(0, On the top of the Crumpetty Tree)
(33, The Quangle Wangle sat,)
(57, But his face you could not see,)
(89, On account of his Beaver Hat.)
```

Text Input: *TextInputFormat*

- Clearly, the **keys** are ***not*** line numbers.
- This would be impossible to implement in general, in that a file is broken into splits at byte, not line, boundaries.
- Splits are processed independently.
- Line numbers are really a sequential notion.
- We have to keep a count of lines as we consume them, so knowing the line number within a split would be possible, but not within the file.

Text Input: *TextInputFormat*

The Relationship Between Input Splits and HDFS Blocks:

- The logical records that **FileInputFormats** define usually do not fit neatly into **HDFS blocks.**
- For example, a **TextInputFormat's** logical records are lines, which will cross **HDFS** boundaries more often than not. This has no bearing on the functioning of our program-lines are not missed or broken, for example-but it's worth knowing about because it does mean that data-local maps (that is, maps that are running on the same host as their input data) will perform some remote reads. The slight overhead this causes is not normally significant.

Text Input: *TextInputFormat*

The Relationship Between Input Splits and HDFS Blocks:

The *following Figure* shows an example.

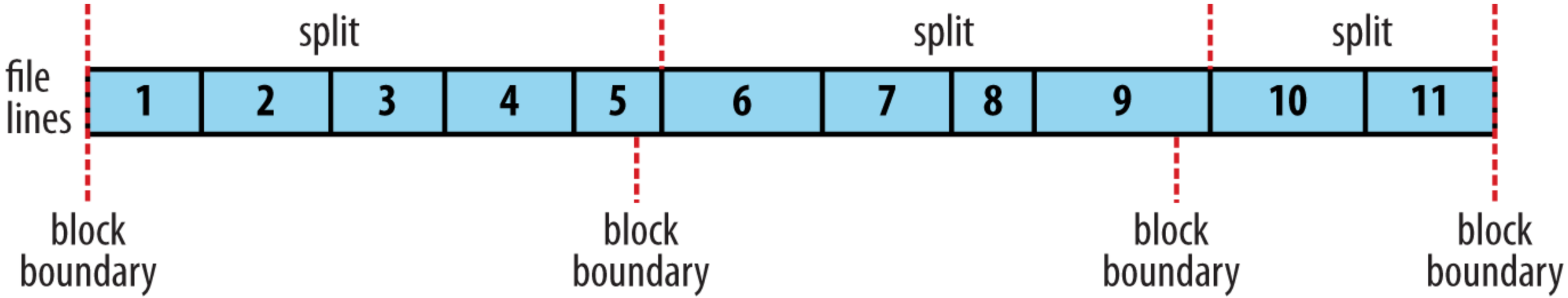

Figure. *Logical records and HDFS blocks for TextInputFormat*

Text Input: *TextInputFormat*

Controlling the maximum line length:

- If we are using one of the text input formats, we can set a maximum expected line length to safeguard against corrupted files.
- Corruption in a file can manifest itself as a very long line, which can cause *out-of memory errors* and then task failure.
- By setting **mapreduce.input.linerecordreader.line.maxlength** to a value in bytes that fits in memory (and is comfortably greater than the length of lines in our input data), we ensure that the record reader will skip the (long) corrupt lines without the task failing.

Text Input: *KeyValueTextInputFormat*

- **TextInputFormat**'s keys, being simply the offsets within the file, are not normally very useful.
- It is common for each line in a file to be a ***key-value*** pair, separated by a delimiter such as a tab character.
- For example, this is the kind of output produced by **TextOutputFormat**, Hadoop's default **OutputFormat**.
- To interpret such files correctly, **KeyValueTextInputFormat** is appropriate.

Text Input: *KeyValueTextInputFormat*

- We can specify the separator via the **mapreduce.input.keyvaluelinerecordreader.key.value.separator** property.
- It is a tab character by default.
- Consider the following *input file*, where → represents a (horizontal) tab character:

Text Input: *KeyValueTextInputFormat*

```
line1→On the top of the Crumpetty Tree
line2→The Quangle Wangle sat,
line3→But his face you could not see,
line4→On account of his Beaver Hat.
```

Like in the **TextInputFormat** case, the input is in a single split comprising four records, although this time the ***keys*** are the Text sequences before the tab in each line:

```
(line1, On the top of the Crumpetty Tree)
(line2, The Quangle Wangle sat,)
(line3, But his face you could not see,)
(line4, On account of his Beaver Hat.)
```

Text Input: *NLineInputFormat*

- With **TextInputFormat** and **KeyValueTextInputFormat**, each **mapper** receives a variable number of lines of input.
- The number depends on the size of the split and the length of the lines.
- If we want our **mappers** to receive a fixed number of lines of input, then **NLineInputFormat** is the **InputFormat** to use.
- Like with **TextInputFormat**, the ***keys*** are the byte offsets within the file and the ***values*** are the lines themselves.

Text Input: *NLineInputFormat*

- **N** refers to the number of lines of input that each **mapper** receives.
- With **N** set to **1** (the default), each **mapper** receives exactly one line of input.
- The **mapreduce.input.lineinputformat.linespermap** property controls the value of **N**.

Text Input: *NLineInputFormat*

Example, consider these four lines again:

```
On the top of the Crumpetty Tree
The Quangle Wangle sat,
But his face you could not see,
On account of his Beaver Hat.
```

If, for example, **N** is **2**, then each split contains two lines.

One **mapper** will receive the first two **key-value** pairs:

```
(0, On the top of the Crumpetty Tree)
(33, The Quangle Wangle sat,)
```

Text Input: *NLineInputFormat*

And another **mapper** will receive the second two **key-value** pairs:

```
(57, But his face you could not see,)
(89, On account of his Beaver Hat.)
```

The **keys** and **values** are the same as those that **TextInputFormat** produces. The difference is in the way the splits are constructed.

Text Input: *NLineInputFormat*

- Usually, having a **map** task for a small number of lines of input is inefficient (due to the overhead in task setup), but there are applications that take a small amount of input data and run an extensive (i.e., **CPU-intensive**) computation for it, then release their output.
- Simulations are a good example.
- By creating an input file that specifies input parameters, one per line, you can perform a ***parameter sweep***: run a set of simulations in parallel to find how a model varies as the parameter changes.

Text Input: *XML*

- Most **XML** parsers operate on whole **XML** documents, so if a large **XML** document is made up of multiple input splits, it is a challenge to parse these individually.
- Of course, we can process the entire **XML** document in one **mapper** (if it is not too large) using the technique in **"Processing a whole file as a record"** discussed previously.

Text Input: *XML*

- Large **XML** documents that are composed of a series of **"records"** (**XML** document fragments) can be broken into these records using simple string or regular-expression matching to find the start and end tags of records.
- This reduce the problem when the document is split by the framework because the next start tag of a record is easy to find by simply scanning from the start of the split, just like **TextInputFormat** finds newline boundaries.

Text Input: *XML*

- **Hadoop** comes with a class for this purpose called **StreamXmlRecordReader** (which is in the **org.apache.hadoop.streaming.mapreduce** package, although it can be used outside of Streaming).
- We can use it by setting our input format to **StreamInputFormat** and setting the **stream.recordreader.class** property to **org.apache.hadoop.streaming.mapreduce.StreamXmlRecordReader**.
- The reader is configured by setting job configuration properties to tell it the patterns for the start and end tags.

Text Input: *XML*

- **Example:** **Wikipedia** provides dumps of its content in **XML** form, which are appropriate for processing in parallel with **MapReduce** using this approach.
- The data is contained in one large **XML** wrapper document, which contains a series of elements, such as page elements that contain a page's content and associated metadata.
- Using **StreamXmlRecordReader**, the page elements can be interpreted as records for processing by a **mapper**.

Output Formats

Output Formats

- **Hadoop** has **output** data formats that correspond to the **input** formats covered previously.
- The **OutputFormat** class hierarchy is shown in *below Figure*:

Output Formats

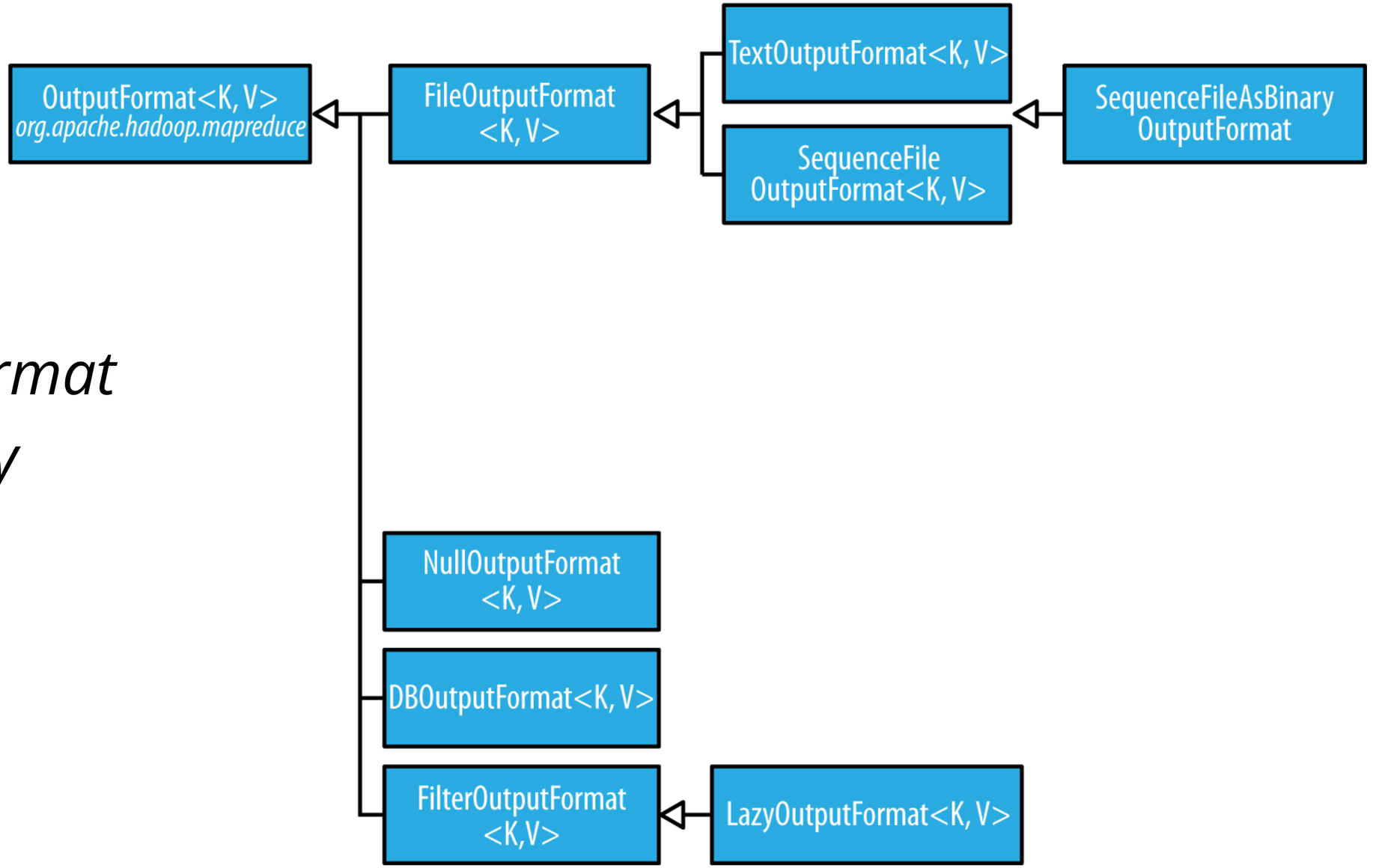

Figure. *OutputFormat class hierarchy*

Output Formats:

Text Output

Output Formats: *Text Output*

- The *default* **output format**, **TextOutputFormat**, writes records as lines of text.
- Its ***keys*** and ***values*** may be of any type, since **TextOutputFormat** turns them to strings by calling **toString()** on them.
- Each ***key-value*** pair is separated by a tab character, although that may be changed using the **mapreduce.output.textoutputformat.separator** property.

Output Formats: *Text Output*

- The counterpart to **TextOutputFormat** for reading in this case is **KeyValueTextInputFormat**, since it breaks lines into **key-value** pairs based on a configurable separator.
- We can suppress the **key** or the **value** from the **output** (or both, making this output format equivalent to **NullOutputFormat**, which release nothing) using a **NullWritable** type.

Output Formats: *Text Output*

- This also causes no separator to be written, which makes the **output** suitable for reading in using **TextInputFormat**.

Developing a MapReduce Application

Developing a MapReduce Application

Refer Laboratory Experiment:

Analysis of Weather Dataset using Mapper Reducer on single node Cluster using Hadoop

(JAVA Implementation)