CSE535 Asynchronous Systems

MapReduce Types and Formats

YoungMin Kwon

MapReduce Types

- map: (K1, V1) -> list(K2, V2)
- reduce: (K2, list(V2)) -> list(K3, V3)

- Because Mapper and Reducer are separate classes, the type parameters have different scope
  - Mapper output and Reducer input must match, but Java compiler does not enforce it

```
//
// MapReduce Signatures
//
public class Mapper<KEYIN, VALUEIN, KEYOUT, VALUEOUT> {
    public class Context extends MapContext<KEYIN,  VALUEIN,
                                            KEYOUT, VALUEOUT> {
        //...
    }
    protected void map(KEYIN key, VALUEIN value, Context context)
        throws IOException, InterruptedException{
        //...
    }
}

public class Reducer<KEYIN, VALUEIN, KEYOUT, VALUEOUT> {
    public class Context extends ReducerContext<KEYIN,  VALUEIN,
                                                KEYOUT, VALUEOUT> {
        //...
    }
    protected void reduce(KEYIN key, Iterable<VALUEIN> values,
                          Context context)
        throws IOEXception, InterruptedException {
        //...
    }
}
```

APIs for Configuring MapReduce Types

JobConf setter method	Input types		Intermediate types		Output types	
	K1	V1	K2	V2	K3	V3
setInputFormat()	•	•				
setMapOutputKeyClass()			•			
setMapOutputValueClass()				•		
setOutputKeyClass()					•	
setOutputValueClass()						•

Input Formats

- An input split is a single chunk of the input processed by a single map.
  - Each map processes a single split
  - Each split is divided into records. The mapper processes each record (key-value pair) in turn

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

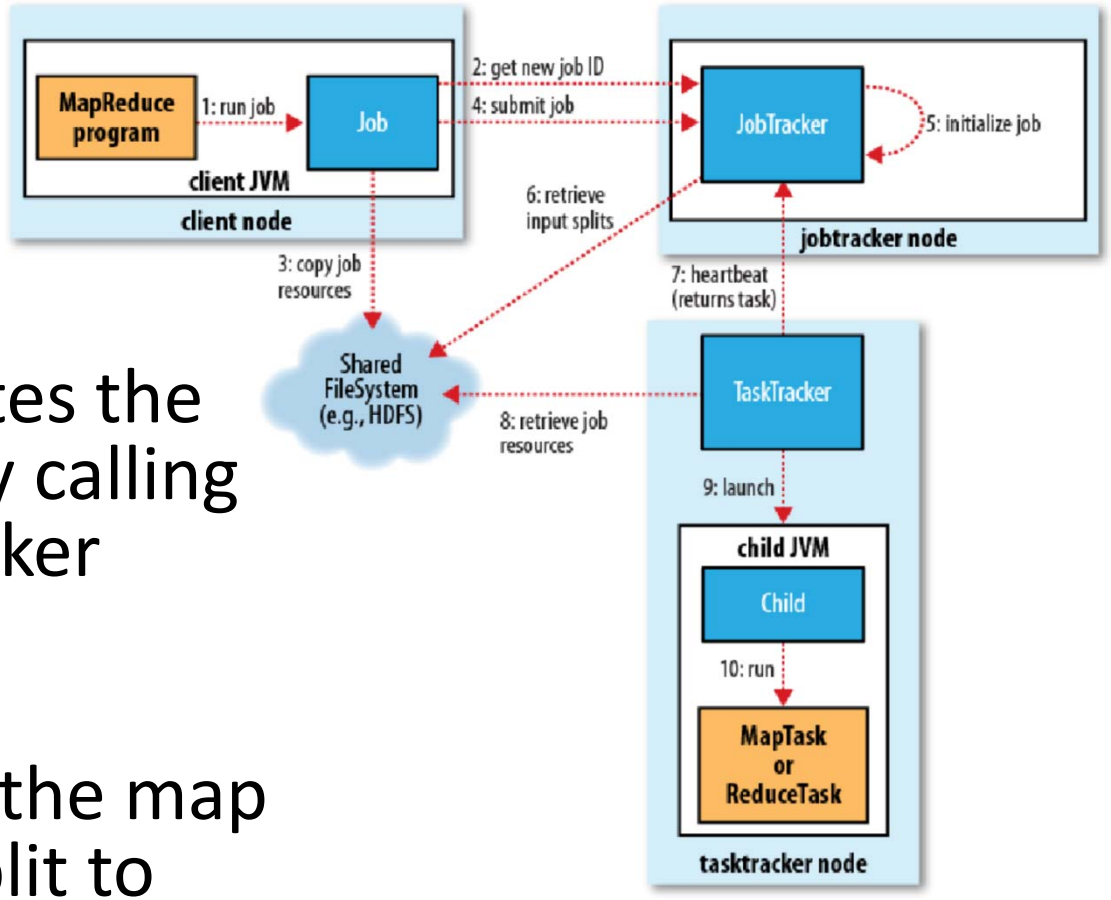
Input Formats

- InputFormat is responsible for
  - Creating InputSplits
  - Dividing the InputSplits into Records

```
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;
};
```

InputFormats



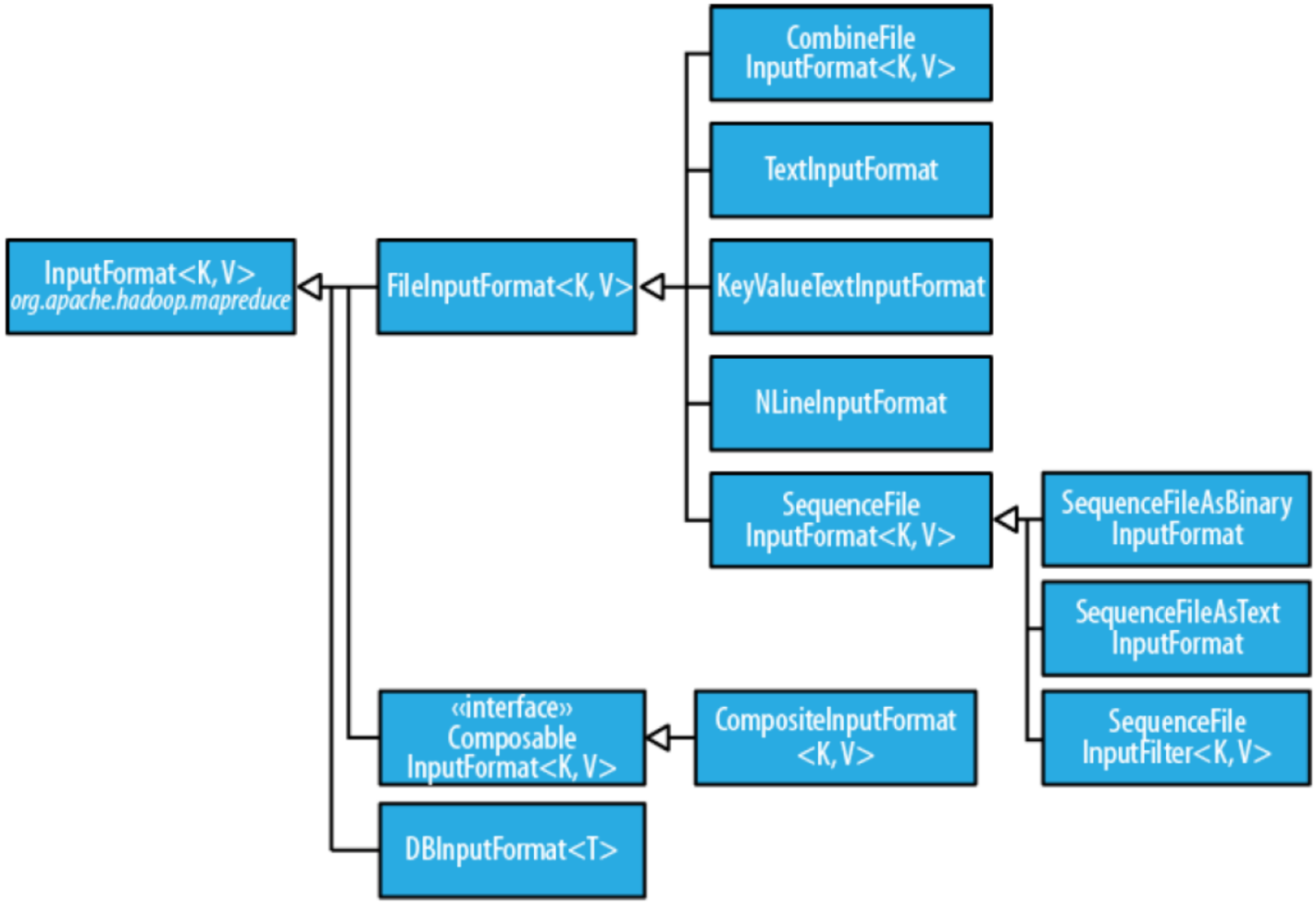
- The client calculates the spilt for the job by calling getSplits. JobTracker retrieves them
- On a TaskTracker, the map task passes the split to createRecordReader

InputFormats

- Using RecordReader, a mapper's run method reads the key and value pairs

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

InputFormat Class Hierarchy



FileInputFormat

- Base class for all InputFormats that use files as their data source
- Two roles
  - Define which files are included as the input to a job
  - Generating splits for the input files
- Four methods to set input paths

```
public static void addInputPath(Job job, Path path);
public static void addInputPaths(Job job, String commaSeparatedPaths);
public static void setInputPaths(Job job, Path... inputPaths);
public static void setInputPaths(Job job, String commaSeparatedPaths);
```

FileInputFormat

- Given a set of files, how FileInputFormat turns them into splits?
  - FileInputFormat splits only large files (larger than HDFS block)
- Split size is calculated by
  - max(minimumSize, min(maximumSize, blockSize))
  - by default minumSize < blockSize < maximumSize

Property name	Type	Default value	Description
mapred.min.split.size	int	1	The smallest valid size in bytes for a file split
mapred.max.split.size[a]	long	Long.MAX_VALUE, that is, 9223372036854775807	The largest valid size in bytes for a file split
dfs.block.size	long	64 MB, that is,67108864	The size of a block in HDFS in bytes

FileInputFormat

Minimum split size	Maximum split size	Block size	Split size	Comment
1 (default)	Long.MAX_VALUE (default)	64 MB (default)	64 MB	By default, the split size is the same as the default block size.
1 (default)	Long.MAX_VALUE (default)	128 MB	128 MB	The most natural way to increase the split size is to have larger blocks in HDFS, either by setting `dfs.block size` or on a per-file basis at file construction time.
128 MB	Long.MAX_VALUE (default)	64 MB (default)	128 MB	Making the minimum split size greater than the block size increases the split size, but at the cost of locality.
1 (default)	32 MB	64 MB (default)	32 MB	Making the maximum split size less than the block size decreases the split size.

Split Size = max(minimumSize, min(maximumSize, blockSize))

FileInputFormat

- CombineFileInputFormat
  - Pack many small files into each split
  - Each mapper has more to process
- Preventing Splitting
  - Set minimumSize to Long.MAX_VALUE
  - Override isSplitable() to return false

```
public class NonSplitableTextInputFormat extends TextInputFormat {
    @Override
    protected boolean isSplitable(JobContext context, Path file) {
        return false;
    }
}
```

FileInputFormat

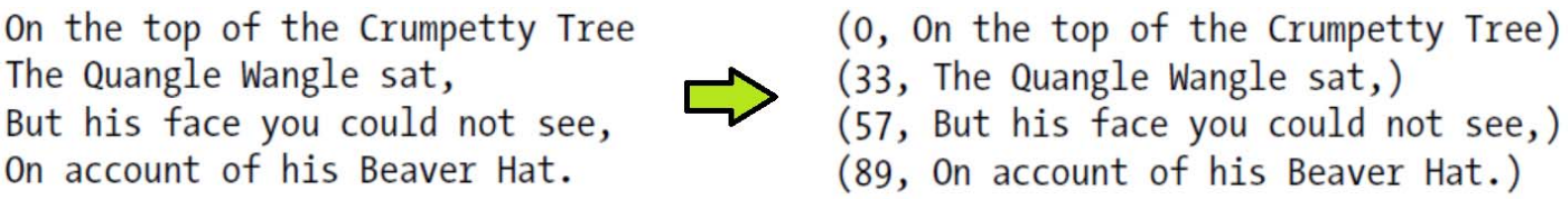
Processing a whole file as a record

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

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

TextInput

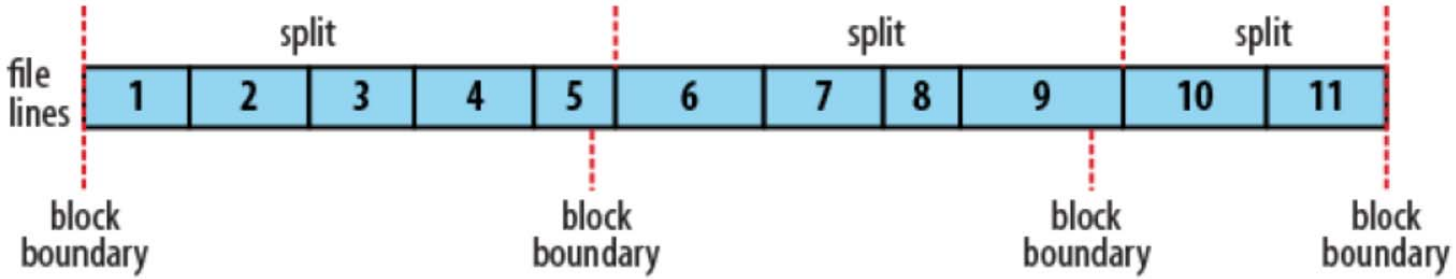
- TextInputFormat



  - Key: LongWritable, byte offset within the file
  - Value: Text, the content of the line

TextInput

- Line boundaries do not correspond with the HDFS boundaries. Splits honor logical record boundaries (lines)

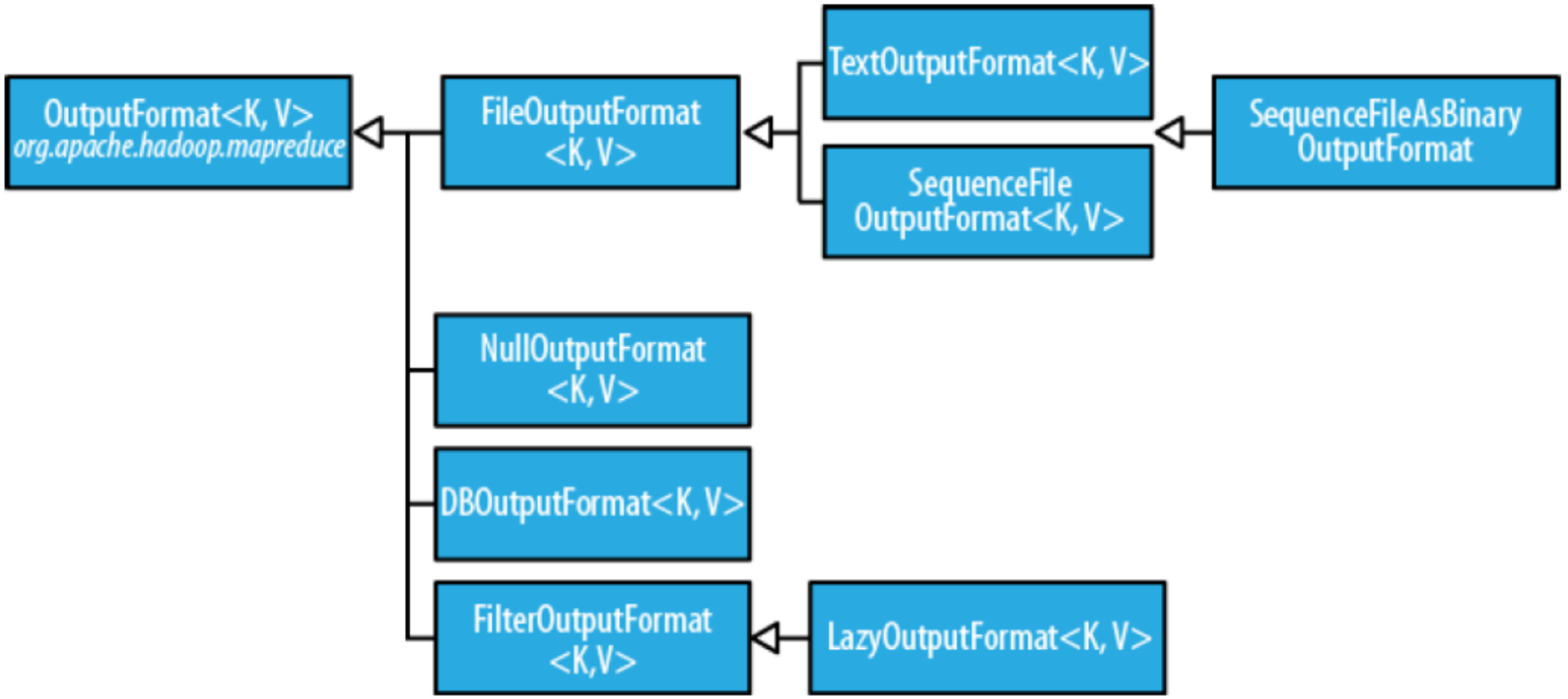


Other InputFormat

- NLineInputFormat
  - Each mapper receives N lines
  - mapreduce.input.lineinputformat.linespermap property controls the value of N
- XML
  - StreamXmlRecordReader class
- DBInputFormat
  - Reading data from a relational database, using JDBC
- Multiple Inputs (different formats)

```
MultipleInputs.addInputPath(job, ncdInputPath,
    TextInputForamt.class, MaxTemperatureMapper.class);
MultipleInputs.addInputPath(job, metOfficeInputPath,
    TextInputForamt.class, MetOfficeMaxTemperatureMapper.class);
```

Output Formats



WholeFileInputFormat.java

```
import java.io.*;
import org.apache.hadoop.fs.*;
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapreduce.*;
import org.apache.hadoop.mapreduce.lib.input.*;

public class WholeFileInputFormat
    extends FileInputFormat<NullWritable, BytesWritable> {

    @Override
    protected boolean isSplitable(JobContext context, Path file) {
        return false;
    }

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

WholeFileRecordReader.java

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

public class WholeFileRecordReader
    extends RecordReader<NullWritable, BytesWritable> {

    private FileSplit fileSplit;
    private Configuration conf;
    private BytesWritable value = new BytesWritable();
    private boolean processed = false;

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

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

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

    @Override
    public BytesWritable getCurrentValue()
        throws IOException, InterruptedException {
        return value;
    }

    @Override
    public float getProgress() throws IOException {
        return processed ? 1.0f : 0.0f;
    }

    @Override
    public void close() throws IOException {}
}
```

SmallFilesToSequenceFileConverter.java

```
import java.io.*;
import org.apache.hadoop.conf.*;
import org.apache.hadoop.fs.*;
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapreduce.*;
import org.apache.hadoop.mapreduce.lib.input.*;
import org.apache.hadoop.mapreduce.lib.output.*;
import org.apache.hadoop.util.*;
public class SmallFilesToSequenceFileConverter
    extends Configured implements Tool {
    static class SequenceFileMapper
        extends Mapper<NullWritable, BytesWritable, Text, BytesWritable> {
        private Text filenameKey;
        @Override
        protected void setup(Context context)
            throws IOException {
            InputSplit split = context.getInputSplit();
            Path path = ((FileSplit) split).getPath();
            filenameKey = new Text(path.toString());
        }
        @Override
        protected void map(NullWritable key,
                           BytesWritable value, Context context)
            throws IOException, InterruptedException {
            context.write(filenameKey, value);
        }
    }
```

```
    @Override
    public int run(String[] args) throws Exception {
        Job job = new Job(this.getConf());
        job.setJarByClass(this.getClass());

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

        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;
    }

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

Makefile

```
all:
    javac *.java
    jar cf SmallFiles.jar *.class
run:
    hadoop jar SmallFiles.jar SmallFilesToSequenceFileConverter\
                -D mapreduce.job.reduces=2\
                /input/smallfiles /output/sample2
    hadoop dfs -cat /output/sample2/part-r-00000
    hadoop dfs -cat /output/sample2/part-r-00001
init:
    hadoop dfs -rm /input/sample2.txt; true
    hadoop dfs -put sample2.txt /input/smallfiles/; true
    hadoop dfs -put sample2.txt /input/smallfiles/sample1.txt; true
    hadoop dfs -rm /output/sample2/*; true
    hadoop dfs -rmdir /output/sample2; true


*** Try, make all; make init; make run
```