APACHE Spark™

Loading + Saving Data

Loading & Saving Data

1. So far we
    a. either converted in-memory data
    b. Or used the HDFS file

Loading & Saving Data

1. So far we
    a. either converted in-memory data
    b. Or used the HDFS file
2. Spark supports wide variety of datasets
3. Can access data through InputFormat & OutputFormat
    a. The interfaces used by Hadoop
    b. Available for many common file formats and storage systems (e.g., S3, HDFS, Cassandra, HBase, etc.).

Common Data Sources

File formats	Stores

Common Data Sources

File formats	**Stores**
● Text, JSON, SequenceFiles, Protocol buffers. ● We can also configure compression	

Common Data Sources

File formats	**Stores**
● Text, JSON, SequenceFiles, Protocol buffers. ● We can also configure compression	**Filesystems** ● Local, NFS, HDFS, Amazon S3 **Databases and key/value stores** ● For Cassandra, HBase, Elasticsearch, and JDBC databases.

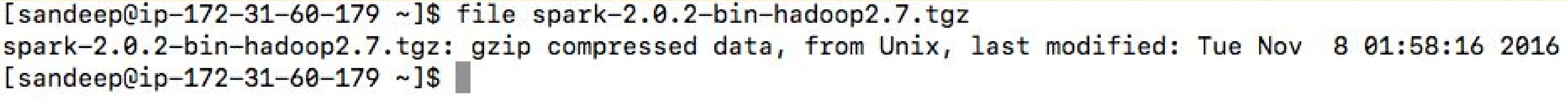
Loading & Saving Data

Structured data sources through Spark SQL aka Data Frames

- Efficient API for structured data sources, including JSON and Apache Hive
- Covered later

Common supported file formats

- + A file could be in any format
- + If we know upfront, we can read it and load it
- + Else we can use “file” command like tool



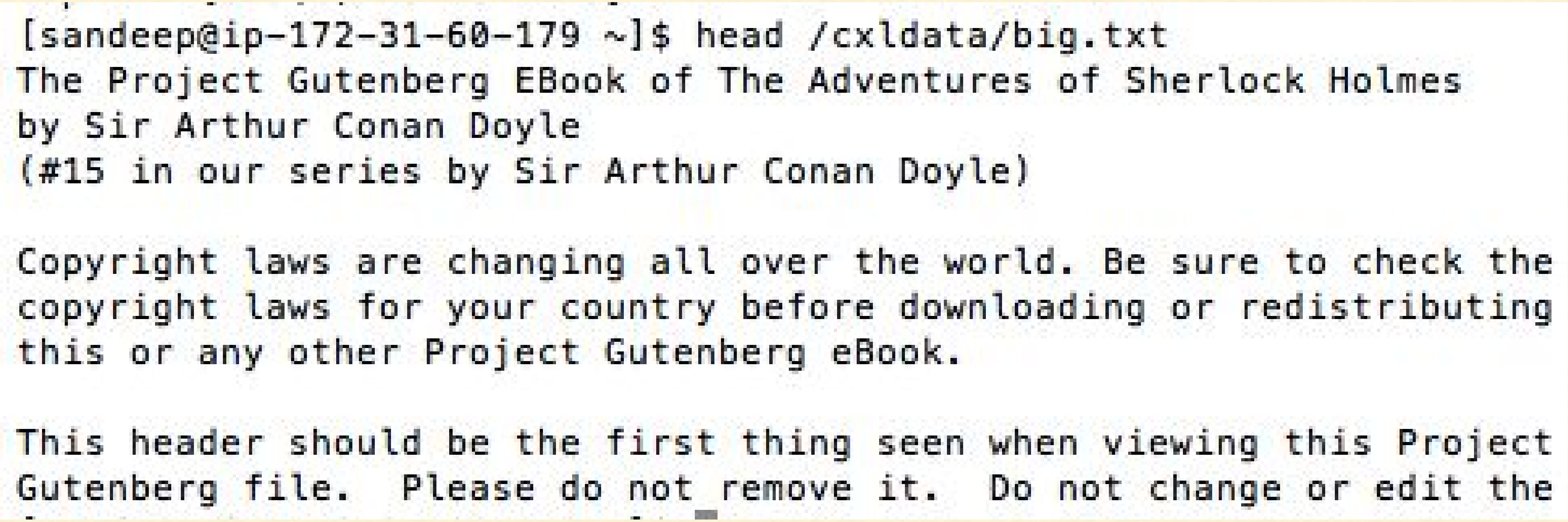
```
[sandeep@ip-172-31-60-179 ~]$ file spark-2.0.2-bin-hadoop2.7.tgz
spark-2.0.2-bin-hadoop2.7.tgz: gzip compressed data, from Unix, last modified: Tue Nov  8 01:58:16 2016
[sandeep@ip-172-31-60-179 ~]$ 
```

Common supported file formats

Text files

- Very common. Plain old text files. Printable chars
- Records are assumed to be one per line.
- Unstructured Data

Example:



```
[sandeep@ip-172-31-60-179 ~]$ head /cxldata/big.txt
The Project Gutenberg EBook of The Adventures of Sherlock Holmes
by Sir Arthur Conan Doyle
(#15 in our series by Sir Arthur Conan Doyle)

Copyright laws are changing all over the world. Be sure to check the
copyright laws for your country before downloading or redistributing
this or any other Project Gutenberg eBook.

This header should be the first thing seen when viewing this Project
Gutenberg file.  Please do not remove it.  Do not change or edit the
```

Common supported file formats

JSON files

- + Javascript Object Notation
- + Common text-based format
- + Semistructured; most libraries require one record per line.

Example:

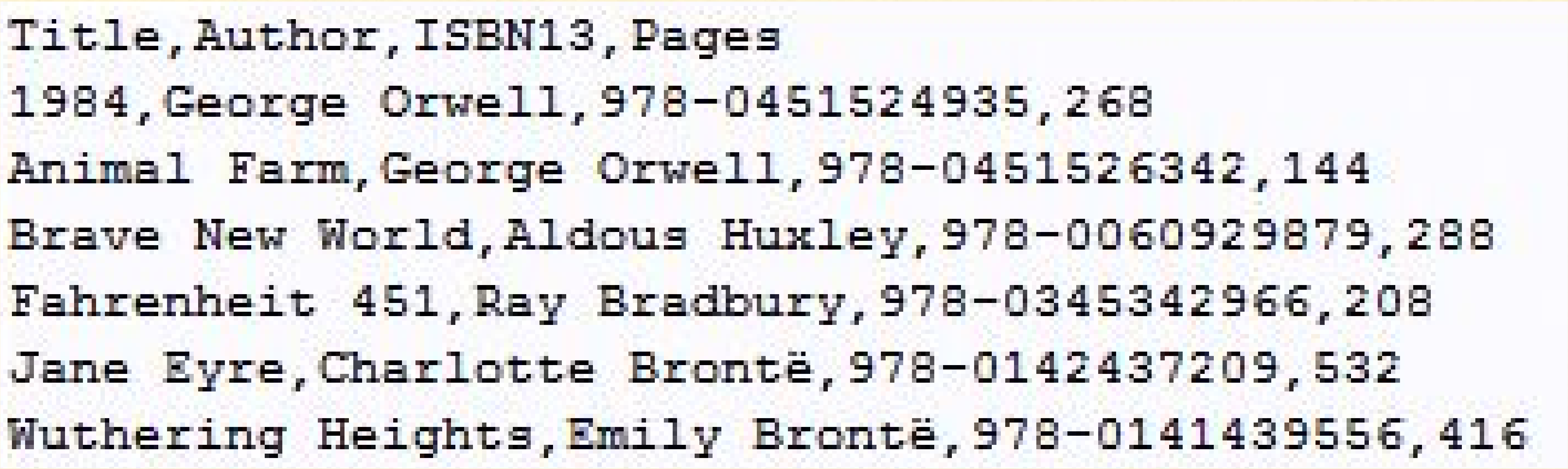
```
{
   "name" : "John",
   "age" : 31,
   "knows" : [“C”, “C++”]
}
```

Common supported file formats

CSV files

- Very common text-based format
- Often used with spreadsheet applications.
- Comma separated Values

Example:

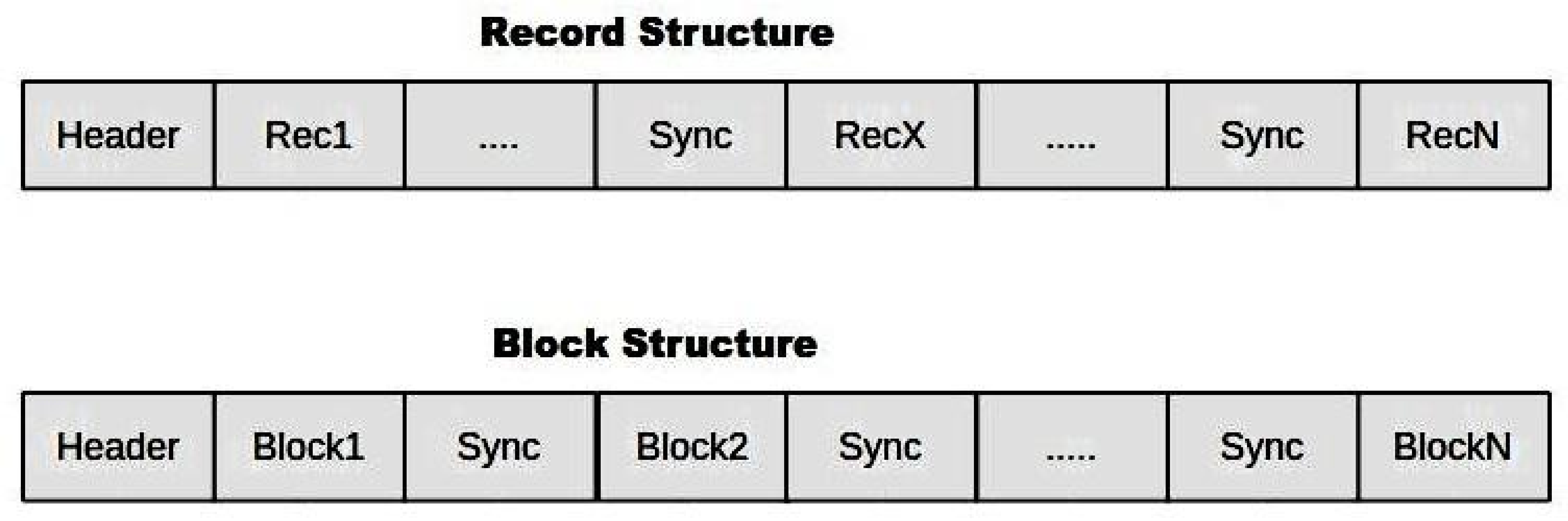


```
Title,Author,ISBN13,Pages
1984,George Orwell,978-0451524935,268
Animal Farm,George Orwell,978-0451526342,144
Brave New World,Aldous Huxley,978-0060929879,288
Fahrenheit 451,Ray Bradbury,978-0345342966,208
Jane Eyre,Charlotte Brontë,978-0142437209,532
Wuthering Heights,Emily Brontë,978-0141439556,416
```

Common supported file formats

Sequence files

- + Compact Hadoop file format used for key/value data.
- + Key and values can be binary data
- + To bundle together many small files



Record Structure

Header	Rec1	....	Sync	RecX	.....	Sync	RecN

Block Structure

Header	Block1	Sync	Block2	Sync	.....	Sync	BlockN

See More at https://wiki.apache.org/hadoop/SequenceFile

Common supported file formats

Protocol buffers

- A fast, space-efficient multilanguage format.
- More compact than JSON.

```
message Person {
  required string name = 1;
  required int32 id = 2;
  optional string email = 3;
}
```

See More at https://developers.google.com/protocol-buffers/

Common supported file formats

Object Files

+ For data from a Spark job to be consumed by another
+ Breaks if you change your classes - Java Serialization.

Handling Text Files - scala

Loading Files

```
var input = sc.textFile("/data/ml-100k/u1.test")
```

Handling Text Files - scala

Loading Files

```
var input = sc.textFile("/data/ml-100k/u1.test")
```

Loading Directories

```
var input = sc.wholeTextFiles("/data/ml-100k");
var lengths = input.mapValues(x => x.length);
lengths.collect();
```

```
[(u'hdfs://ip-172-31-53-48.ec2.internal:8020/data/ml-100k/mku.sh', 643),
(u'hdfs://ip-172-31-53-48.ec2.internal:8020/data/ml-100k/u.data', 1979173),
(u'hdfs://ip-172-31-53-48.ec2.internal:8020/data/ml-100k/u.genre', 202),
(u'hdfs://ip-172-31-53-48.ec2.internal:8020/data/ml-100k/u.info', 36) …]
```

Handling Text Files - scala

Loading Files

```
var input = sc.textFile("/data/ml-100k/u1.test")
```

Loading Directories

```
var input = sc.wholeTextFiles("/data/ml-100k");
var lengths = input.mapValues(x => x.length);
lengths.collect();
```

```
[(u'hdfs://ip-172-31-53-48.ec2.internal:8020/data/ml-100k/mku.sh', 643),
(u'hdfs://ip-172-31-53-48.ec2.internal:8020/data/ml-100k/u.data', 1979173),
(u'hdfs://ip-172-31-53-48.ec2.internal:8020/data/ml-100k/u.genre', 202),
(u'hdfs://ip-172-31-53-48.ec2.internal:8020/data/ml-100k/u.info', 36) …]
```

Saving Files

```
lengths.saveAsTextFile(outputDir)
```

Comma / Tab -Separated Values (CSV / TSV)

1. Records are stored one per line,
2. Fixed number of fields per line
3. Fields are separated by a comma (tab in TSV)
4. We get row number to detect header etc.

Loading CSV - Sample Data

Data: /data/spark/temps.csv

```
20, NYC, 2014-01-01
20, NYC, 2015-01-01
21, NYC, 2014-01-02
23, BLR, 2012-01-01
25, SEATLE, 2016-01-01
21, CHICAGO, 2013-01-05
24, NYC, 2016-5-05
```

Loading CSV - Simple Approach

```
var lines = sc.textFile("/data/spark/temps.csv");
var recordsRDD = lines.map(line => line.split(","));
recordsRDD.take(10);
```

```
Array(
    Array(20, " NYC", " 2014-01-01"),
    Array(20, " NYC", " 2015-01-01"),
    Array(21, " NYC", " 2014-01-02"),
    Array(23, " BLR", " 2012-01-01"),
    Array(25, " SEATLE", " 2016-01-01"),
    Array(21, " CHICAGO", " 2013-01-05"),
    Array(24, " NYC", " 2016-5-05")
)
```

Loading CSV - Example

spark-shell --packages net.sf.opencsv:opencsv:2.3
Or
Add this to sbt: ***libraryDependencies += "net.sf.opencsv" % "opencsv" % "2.3"***

```
import au.com.bytecode.opencsv.CSVParser

var a = sc.textFile("/data/spark/temps.csv");
var p = a.map(
    line => {
        val parser = new CSVParser(',')
        parser.parseLine(line)
     })
p.take(1)
//Array(Array(20, " NYC", " 2014-01-01"))
```

https://gist.github.com/girisandeep/b721cf93981c338665c328441d419253

Loading CSV - Example Efficient

https://gist.github.com/girisandeep/fddf49ef97fde429a0d3256160b257c1

Loading CSV - Example Efficient

```
import au.com.bytecode.opencsv.CSVParser
var linesRdd = sc.textFile("/data/spark/temps.csv");
```

https://gist.github.com/girisandeep/fddf49ef97fde429a0d3256160b257c1

Loading CSV - Example Efficient

```
import au.com.bytecode.opencsv.CSVParser
var linesRdd = sc.textFile("/data/spark/temps.csv");
def parseCSV(itr:Iterator[String]):Iterator[Array[String]] = {
    val parser = new CSVParser(',')
    for(line <- itr)
        yield parser.parseLine(line)
}
```

https://gist.github.com/girisandeep/fddf49ef97fde429a0d3256160b257c1

Loading CSV - Example Efficient

```
import au.com.bytecode.opencsv.CSVParser
var linesRdd = sc.textFile("/data/spark/temps.csv");
def parseCSV(itr:Iterator[String]):Iterator[Array[String]] = {
    val parser = new CSVParser(',')
    for(line <- itr)
        yield parser.parseLine(line)
}
//Check with simple example
val x = parseCSV(Array("1,2,3","a,b,c").iterator)
val result = linesRdd.mapPartitions(parseCSV)
```

https://gist.github.com/girisandeep/fddf49ef97fde429a0d3256160b257c1

Loading CSV - Example Efficient

```
import au.com.bytecode.opencsv.CSVParser
var linesRdd = sc.textFile("/data/spark/temps.csv");
def parseCSV(itr:Iterator[String]):Iterator[Array[String]] = {
    val parser = new CSVParser(',')
    for(line <- itr)
        yield parser.parseLine(line)
}
//Check with simple example
val x = parseCSV(Array("1,2,3","a,b,c").iterator)
val result = linesRdd.mapPartitions(parseCSV)
result.take(1)
//Array[Array[String]] = Array(Array(20, " NYC", " 2014-01-01"))
```

https://gist.github.com/girisandeep/fddf49ef97fde429a0d3256160b257c1

Tab Separated Files

Similar to csv:

```
    val parser = new CSVParser('\t')
```

SequenceFiles

- Popular Hadoop format
  - For handling small files
  - Create InputSplits without too much transport

SequenceFiles

- Popular Hadoop format
    - For handling small files
    - Create InputSplits without too much transport
- Composed of flat files with key/value pairs.
- Has Sync markers
    - Allow to seek to a point
    - Then resynchronize with the record boundaries
    - Allows Spark to efficiently read in parallel from multiple nodes

Loading SequenceFiles

```
val data = sc.sequenceFile(inFile,
"org.apache.hadoop.io.Text", "org.apache.hadoop.io.IntWritable")
data.map(func)
…
```

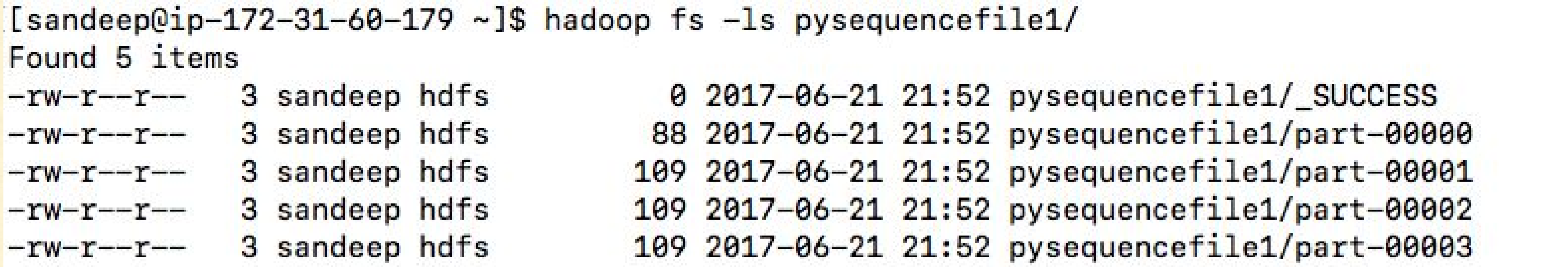
```
data.saveAsSequenceFile(outputFile)
```

Saving SequenceFiles - Example

```
var rdd = sc.parallelize(Array(("key1", 1.0), ("key2", 2.0), ("key3", 3.0)))
rdd.saveAsSequenceFile("pysequencefile1")
```

Saving SequenceFiles - Example

```
var rdd = sc.parallelize(Array(("key1", 1.0), ("key2", 2.0), ("key3", 3.0)))
rdd.saveAsSequenceFile("pysequencefile1")
```



```
[sandeep@ip-172-31-60-179 ~]$ hadoop fs -ls pysequencefile1/
Found 5 items
-rw-r--r--   3 sandeep hdfs          0 2017-06-21 21:52 pysequencefile1/_SUCCESS
-rw-r--r--   3 sandeep hdfs         88 2017-06-21 21:52 pysequencefile1/part-00000
-rw-r--r--   3 sandeep hdfs        109 2017-06-21 21:52 pysequencefile1/part-00001
-rw-r--r--   3 sandeep hdfs        109 2017-06-21 21:52 pysequencefile1/part-00002
-rw-r--r--   3 sandeep hdfs        109 2017-06-21 21:52 pysequencefile1/part-00003
```

Loading SequenceFiles - Example

```
import org.apache.hadoop.io.DoubleWritable
import org.apache.hadoop.io.Text
```

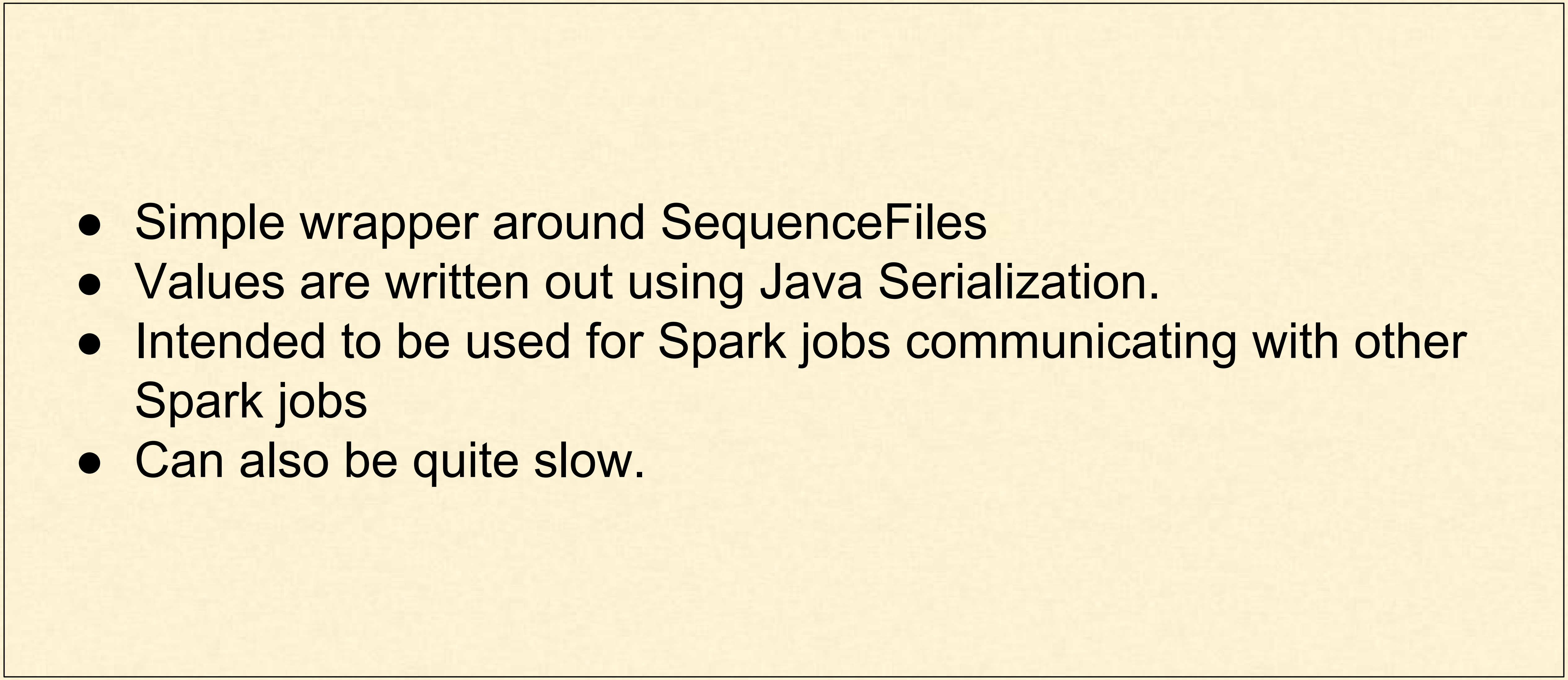
Loading SequenceFiles - Example

```
import org.apache.hadoop.io.DoubleWritable
import org.apache.hadoop.io.Text

val myrdd = sc.sequenceFile(
    "pysequencefile1",
    classOf[Text], classOf[DoubleWritable])
```

Loading SequenceFiles - Example

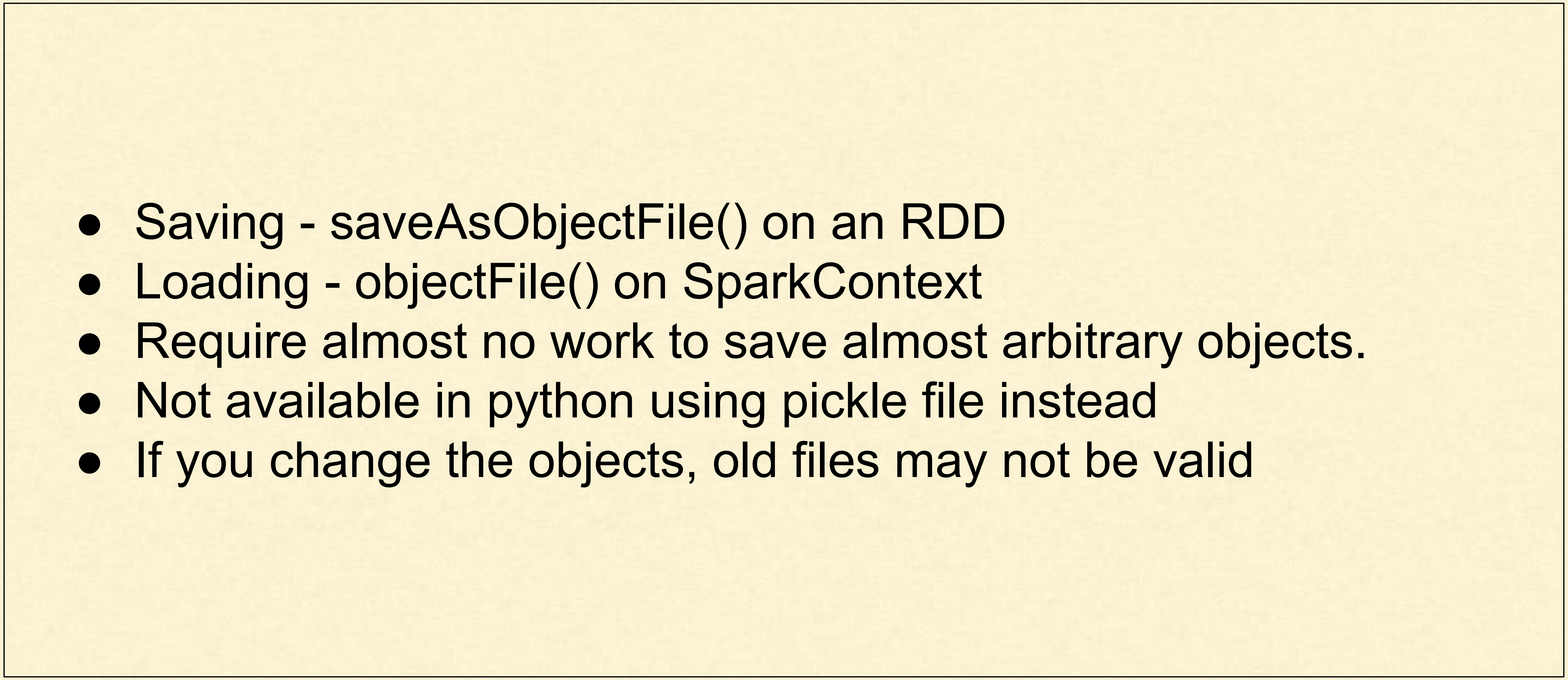
```
import org.apache.hadoop.io.DoubleWritable
import org.apache.hadoop.io.Text

val myrdd = sc.sequenceFile(
    "pysequencefile1",
    classOf[Text], classOf[DoubleWritable])

val result = myrdd.map{case (x, y) => (x.toString, y.get())}
result.collect()

//Array((key1,1.0), (key2,2.0), (key3,3.0))
```

Object Files



- Simple wrapper around SequenceFiles
- Values are written out using Java Serialization.
- Intended to be used for Spark jobs communicating with other Spark jobs
- Can also be quite slow.

Object Files



- Saving - saveAsObjectFile() on an RDD
- Loading - objectFile() on SparkContext
- Require almost no work to save almost arbitrary objects.
- Not available in python using pickle file instead
- If you change the objects, old files may not be valid

Pickle File

- Python way of handling object files
- Uses Python’s pickle serialization library
- Saving - saveAsPickleFile() on an RDD
- Loading - pickleFile() on SparkContext
- Can also be quite slow as Object Fiels

Non-filesystem data sources - hadoopFile

- Access Hadoop-supported storage formats
- Many key/value stores provide Hadoop input formats
- Example providers:HBase, MongoDB

- Older: hadoopFile() / saveAsHadoopFile()
- Newer: newAPIHadoopDataset() / saveAsNewAPIHadoopDataset()
- Takes a Configuration object on which you set the Hadoop properties

Hadoop Input and Output Formats - Old API

hadoopFile(path, inputFormatClass, keyClass, valueClass, keyConverter=None, valueConverter=None, conf=None, batchSize=0)

Read an ‘old’ Hadoop InputFormat with arbitrary key and value class from HDFS, a local file system (available on all nodes), or any Hadoop-supported file system URI. The mechanism is the same as for sc.sequenceFile.

A Hadoop configuration can be passed in as a Python dict. This will be converted into a Configuration in Java.

Parameters:
path – path to Hadoop file
inputFormatClass – fully qualified classname of Hadoop InputFormat (e.g. “org.apache.hadoop.mapred.TextInputFormat”)
keyClass – fully qualified classname of key Writable class (e.g. “org.apache.hadoop.io.Text”)
valueClass – fully qualified classname of value Writable class (e.g. “org.apache.hadoop.io.LongWritable”)
keyConverter – (None by default)
valueConverter – (None by default)
conf – Hadoop configuration, passed in as a dict (None by default)
batchSize – The number of Python objects represented as a single Java object. (default 0, choose batchSize automatically)

Hadoop Input and Output Formats - New API

newAPIHadoopFile(path, inputFormatClass, keyClass, valueClass, keyConverter=None, valueConverter=None, conf=None, batchSize=0)

Read a ‘new API’ Hadoop InputFormat with arbitrary key and value class from HDFS, a local file system (available on all nodes), or any Hadoop-supported file system URI. The mechanism is the same as for sc.sequenceFile.

A Hadoop configuration can be passed in as a Python dict. This will be converted into a Configuration in Java

Parameters:

- path – path to Hadoop file
- inputFormatClass – fully qualified classname of Hadoop InputFormat (e.g. “org.apache.hadoop.mapreduce.lib.input.TextInputFormat”)
- keyClass – fully qualified classname of key Writable class (e.g. “org.apache.hadoop.io.Text”)
- valueClass – fully qualified classname of value Writable class (e.g. “org.apache.hadoop.io.LongWritable”)
- keyConverter – (None by default)
- valueConverter – (None by default)
- conf – Hadoop configuration, passed in as a dict (None by default)
- batchSize – The number of Python objects represented as a single Java object. (default 0, choose batchSize automatically)

Hadoop Input and Output Formats - Old API

Loading Data from mongodb

```
See https://databricks.com/blog/2015/03/20/using-mongodb-with-spark.html
# set up parameters for reading from MongoDB via Hadoop input format
config = {"mongo.input.uri": "mongodb://localhost:27017/marketdata.minbars"}
inputFormatClassName = "com.mongodb.hadoop.MongoInputFormat"

keyClassName = "org.apache.hadoop.io.Text"
valueClassName = "org.apache.hadoop.io.MapWritable"

# read the 1-minute bars from MongoDB into Spark RDD format
minBarRawRDD = sc.newAPIHadoopRDD(inputFormatClassName, keyClassName,
valueClassName, None, None, config)
```

Protocol buffers

- Developed at Google for internal RPCs
- Open sourced
- Structured data - fields & types of fields defined
- Fast for encoding and decoding (20-100x than XML)
- Take up the minimum space (3-10x than xml)
- Defined using a domain-specific language
- Compiler generates accessor methods in variety of languages
- Consist of fields: optional, required, or repeated
- While parsing
  - A missing optional field => success
  - A missing required field => failure
- So, make new fields as optional (remember object file failures?)

Protocol buffers - Example

```
package tutorial;
message Person {
  required string name = 1;
  required int32 id = 2;
  optional string email = 3;
  enum PhoneType {
    MOBILE = 0;
    HOME = 1;
    WORK = 2;
  }
  message PhoneNumber {
    required string number = 1;
    optional PhoneType type = 2 [default = HOME];
  }
  repeated PhoneNumber phone = 4;
}

message AddressBook {
  repeated Person person = 1;
}
```

Protocol buffers - Steps

1. Download and install protocol buffer compiler
2. pip install protobuf
3. protoc -I=$SRC_DIR --python_out=$DST_DIR $SRC_DIR/addressbook.proto
4. create objects
5. Convert those into protocol buffers
6. See this project

File Compression

1. To Save Storage & Network Overhead
2. With most hadoop output formats we can specify compression codecs
3. Compression should not require the whole file at once
4. Each worker can find start of record => splitable
5. You can configure HDP for LZO using Ambari: http://docs.hortonworks.com/HDPDocuments/Ambari-2.2.2.0/bk_ambari_reference_guide/content/_configure_core-sitexml_for_lzo.html

File Compression Options

Characteristics of compression:

- Splittable
- Speed
- Effectiveness on Text
- Code

File Compression Options

Format	**Splittable**	**Speed**	**Effectiveness on text**	**Hadoop compression codec**	**comments**
gzip	N	Fast	High	org.apache.hadoop.io.com.GzipCodec	
lzo	*Y*	*V. Fast*	*Medium*	*com.hadoop.compression.lzo.LzoCodec*	*LZO requires installation on every worker node*
bzip2	*Y*	*Slow*	*V. High*	*org.apache.hadoop.io.com.BZip2Codec*	*Uses pure Java for splittable version*
zlib	N	Slow	Medium	org.apache.hadoop.io.com.DefaultCodec	Default compression codec for Hadoop
Snappy	N	V. Fast	Low	org.apache.hadoop.io.com.SnappyCodec	There is a pure Java port of Snappy but it is not yet available in Spark/ Hadoop

Handling LZO

1. Enable in HADOOP by updating the conf of hadoop
http://docs.hortonworks.com/HDPDocuments/Ambari-2.2.2.0/bk_ambari_reference_guide/content/_configure_core-sitexml_for_lzo.html

2. Create data:

```
$ bzip2 --stdout file.bz2 | lzop -o file.lzo
```

3. Update Spark-env.sh with

```
export SPARK_CLASSPATH=$SPARK_CLASSPATH:hadoop-lzo-0.4.20-SNAPSHOT.jar
```

In your code, use:

```
conf.set("io.compression.codecs”, "com.hadoop.compression.lzo.LzopCodec”);
```

Ref: https://gist.github.com/zedar/c43cbc7ff7f98abee885

Loading + Saving Data: File Systems

Local/“Regular” FS

1. rdd = sc.textFile("**file**:///home/holden/happypandas.gz")
2. The path has to be available on all nodes.
   Otherwise, load it locally and distribute using sc.parallelize

Amazon S3

1. Popular option
2. Good if nodes are inside EC2
3. Use path in all input methods (textFile, hadoopFile etc)

    s3n://bucket/path-within-bucket

4. Set Env. Vars: AWS_ACCESS_KEY_ID AWS_SECRET_ACCESS_KEY

More details: https://cloudxlab.com/blog/access-s3-files-spark/

HDFS

1. The Hadoop Distributed File System
2. Spark and HDFS can be collocated on the same machines
3. Spark can take advantage of this data locality to avoid network overhead
4. In all i/o methods, use path: hdfs://master:port/path
5. Use only the version of spark w.r.t HDFS version

Spark - Loading + Saving Data

Thank you!