Intro to Hadoop SerDe

Formats, compression and serialization/deserialization

What I learned about SerDe* from backpacking

Food - > freezedried -> packs well, is small and light -> rehydrated, still recognizable

analogous to

Data -> serialized and compressed -> packs well, is small -> decompressed and deserialized, still recognizable

Being able to recognize your data is important

*SerDe stands for serialization/deserialization

How do we “freeze-dry” and “rehydrate” files?

Formats

- Text
- SequenceFile
- Avro
- Parquet
- ORC

Compression

- Snappy
- LZ4
- LZF
- LZO
- ZLIB (Deflate)
- BZip2 (splittable)

Agenda

- Native support
- Compressible
- Splittable
- Self-describing

	Avro	**SequenceFile**	**ORC**	**Parquet**
Origin	Hadoop	Hadoop	Hortonworks	Twitter Clouder
Rationale	Portable replacement for Writables	Need splittable, compressible input to MR2	Improved RCFile	Similar to ORC but portable
Applicability	General	General	Hive	General
Storage	Row	Row	Columnar	Columnar
Schema evolution	Yes	No	Yes	Yes
Block size (MB)	128	128	256	128/256**

Serialization/Deserialization in different applications

- Serializable types in Spark
  - **java.io.Serializeable** objects
  - faster serialization with **Kryo**
  - requires **Kryo** *registration*

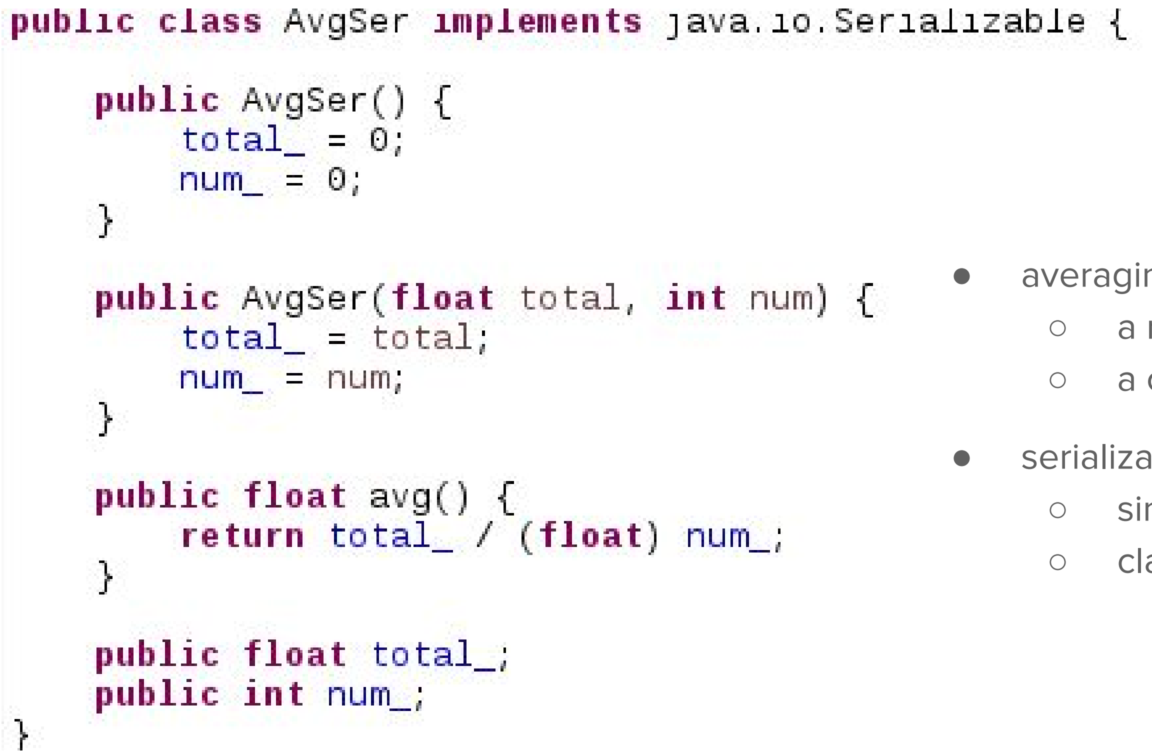
- Serializable types in MapReduce 2:
  - hadoop.io.**Writables**
  - apache.**Avro**

- Serialization classes in Hive
  - Parquet - ParquetSerDe
  - Avro - AvroSerDe
  - ORC - ORCSerDe
  - Text
    - JSON format - JsonSerDe
    - “Regular” format - RegExSerDe
    - Table format - OpenCSVSerde
  - Thrift - ThriftSerDe

Serialization in Spark with an example

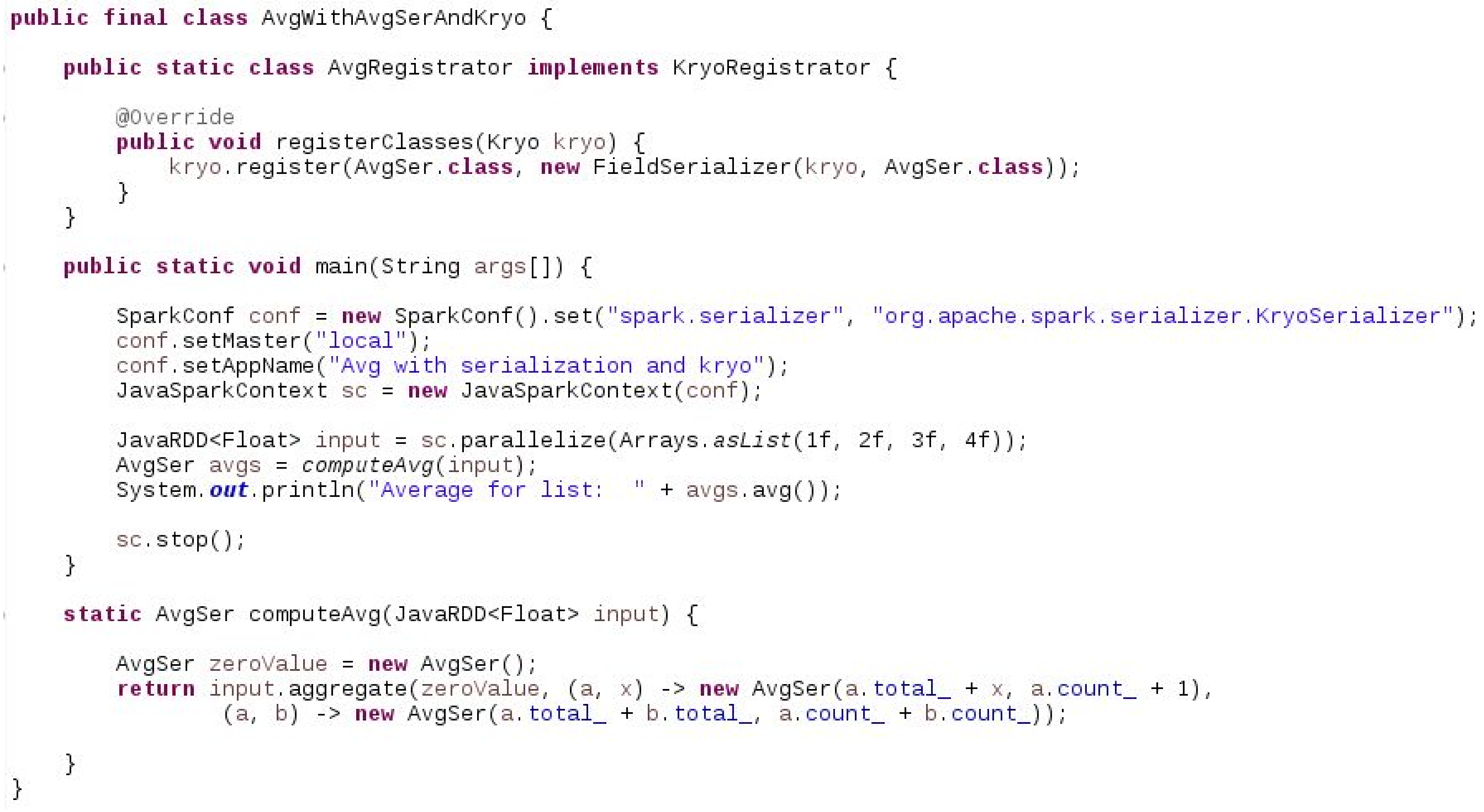
- averaging with aggregate requires us to keep
  - a running total of input values
  - a count of input values
- working with aggregation, simple or complex, can be simpler if we have objects that contain the information and operations needed
  - simplifies the lambda functions
  - clarifies the operations
- we can create a data class for aggregation, but it must be serializable
- if we create a data class, we should register it with Kryo

Serializable for aggregate when averaging



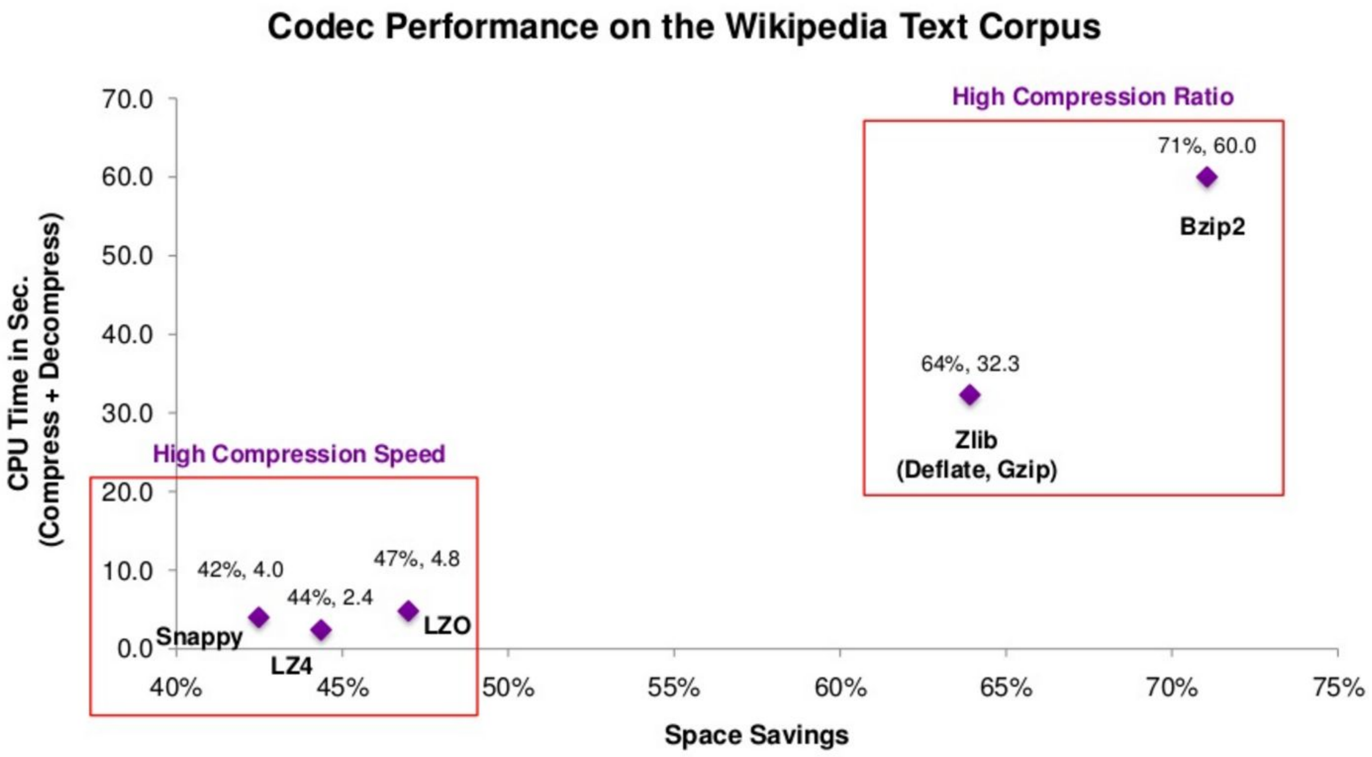
```
public class AvgSer implements java.io.Serializable {

    public AvgSer() {
        total_ = 0;
        num_ = 0;
    }

    public AvgSer(float total, int num) {
        total_ = total;
        num_ = num;
    }

    public float avg() {
        return total_ / (float) num_;
    }

    public float total_;
    public int num_;
}
```

- averaging requires us to keep
  - a running total of input values
  - a count of input values
- serializable data encapsulate info and ops
  - simplify lambda functions
  - clarify the ops



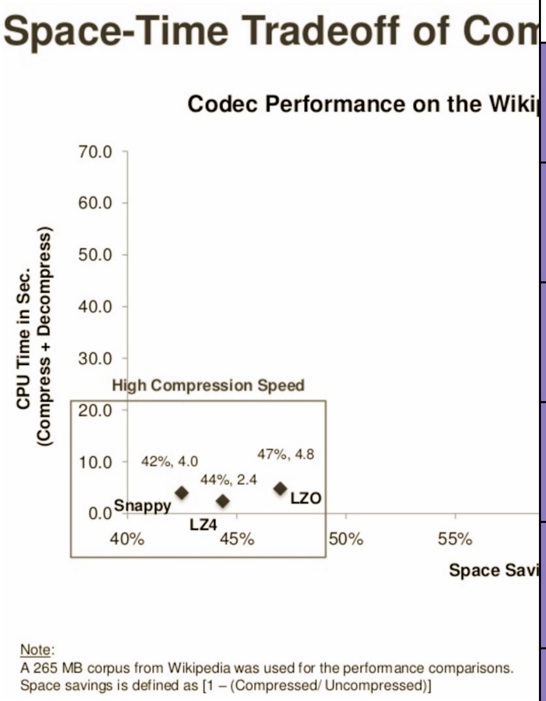
```
public final class AvgWithAvgSerAndKryo {

    public static class AvgRegistrator implements KryoRegistrator {

        @Override
        public void registerClasses(Kryo kryo) {
            kryo.register(AvgSer.class, new FieldSerializer(kryo, AvgSer.class));
        }
    }

    public static void main(String args[]) {

        SparkConf conf = new SparkConf().set("spark.serializer", "org.apache.spark.serializer.KryoSerializer");
        conf.setMaster("local");
        conf.setAppName("Avg with serialization and kryo");
        JavaSparkContext sc = new JavaSparkContext(conf);

        JavaRDD<Float> input = sc.parallelize(Arrays.asList(1f, 2f, 3f, 4f));
        AvgSer avgs = computeAvg(input);
        System.out.println("Average for list:  " + avgs.avg());

        sc.stop();
    }

    static AvgSer computeAvg(JavaRDD<Float> input) {

        AvgSer zeroValue = new AvgSer();
        return input.aggregate(zeroValue, (a, x) -> new AvgSer(a.total_ + x, a.count_ + 1),
                (a, b) -> new AvgSer(a.total_ + b.total_, a.count_ + b.count_));

    }
}
```

Space-Time Tradeoff of Compression Options


Codec Performance on the Wikipedia Text Corpus

Other performance comparisons: [Blog comparing snappy, LZF, LZO and ZLIB]()

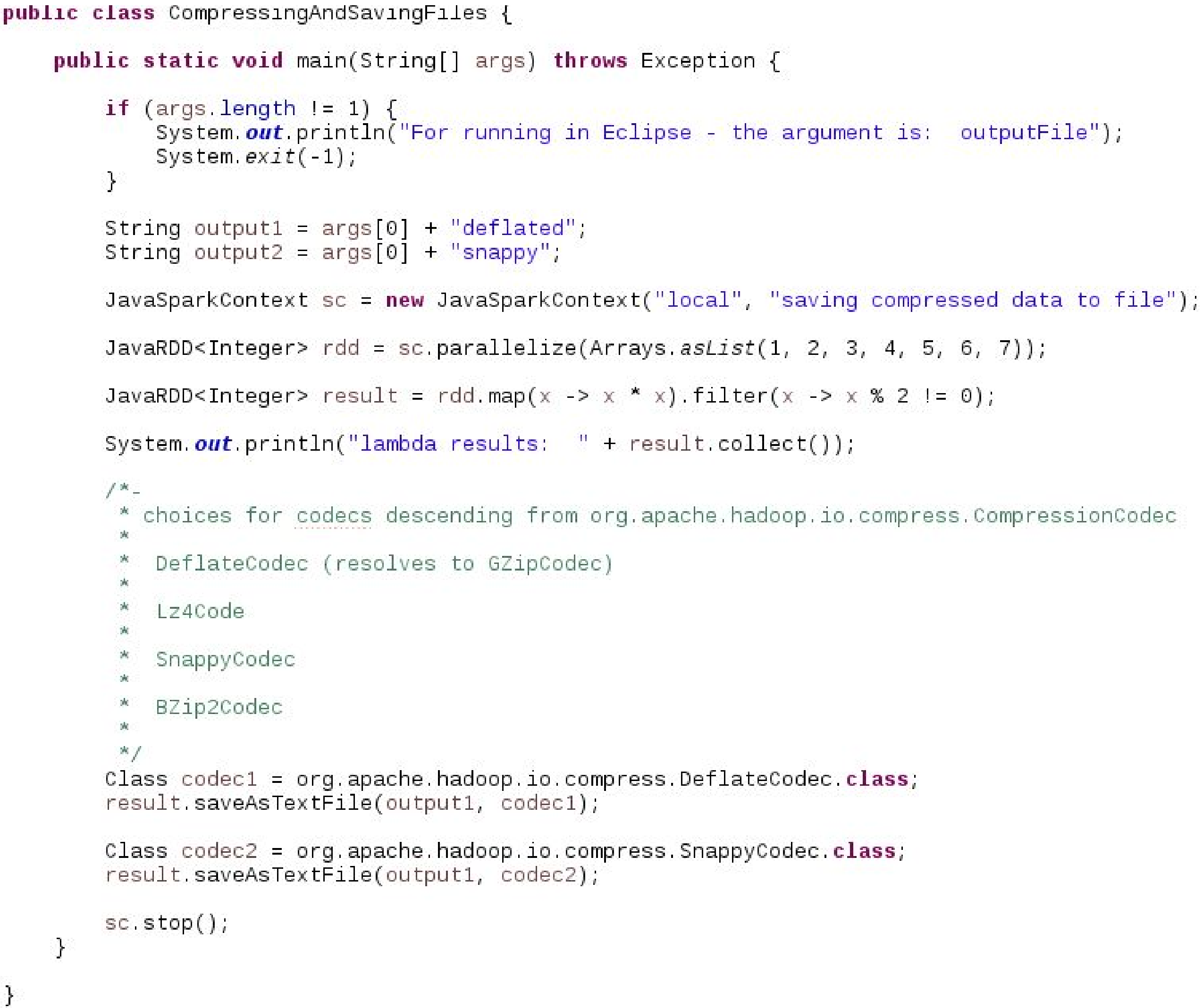
Note:
A 265 MB corpus from Wikipedia was used for the performance comparisons.
Space savings is defined as [1 – (Compressed/ Uncompressed)]



Space-Time Tradeoff of Com

Note:
A 265 MB corpus from Wikipedia was used for the performance comparisons.
Space savings is defined as [1 – (Compressed/ Uncompressed)]

Algorithm	Compression Ratio	Speed	Splittable
Snappy	~40%	4.0	NO
LZ4	~45%	2.4	NO
LZF	~45%	NA	NO
LZO	~45%	4.8	NO
ZLIB (Deflate)	~60%	32.8	NO
BZip2	~71%	60.0	YES



```
public class CompressingAndSavingFiles {

    public static void main(String[] args) throws Exception {

        if (args.length != 1) {
            System.out.println("For running in Eclipse - the argument is:  outputFile");
            System.exit(-1);
        }

        String output1 = args[0] + "deflated";
        String output2 = args[0] + "snappy";

        JavaSparkContext sc = new JavaSparkContext("local", "saving compressed data to file");

        JavaRDD<Integer> rdd = sc.parallelize(Arrays.asList(1, 2, 3, 4, 5, 6, 7));

        JavaRDD<Integer> result = rdd.map(x -> x * x).filter(x -> x % 2 != 0);

        System.out.println("lambda results:  " + result.collect());

        /*-
         * choices for codecs descending from org.apache.hadoop.io.compress.CompressionCodec
         *
         *  DeflateCodec (resolves to GZipCodec)
         *
         *  Lz4Code
         *
         *  SnappyCodec
         *
         *  BZip2Codec
         *
         */
        Class codec1 = org.apache.hadoop.io.compress.DeflateCodec.class;
        result.saveAsTextFile(output1, codec1);

        Class codec2 = org.apache.hadoop.io.compress.SnappyCodec.class;
        result.saveAsTextFile(output1, codec2);

        sc.stop();
    }

}
```

Hey, wait, what about…

..writing files using

- Spark RDDs, DataFrames, DataSets
- Hadoop IO standalone

What about the wrappers for Hadoop IO in Spark?

How do we DO anything with the serialization framework…

Ah, the glories of transitions and evolving open-source.

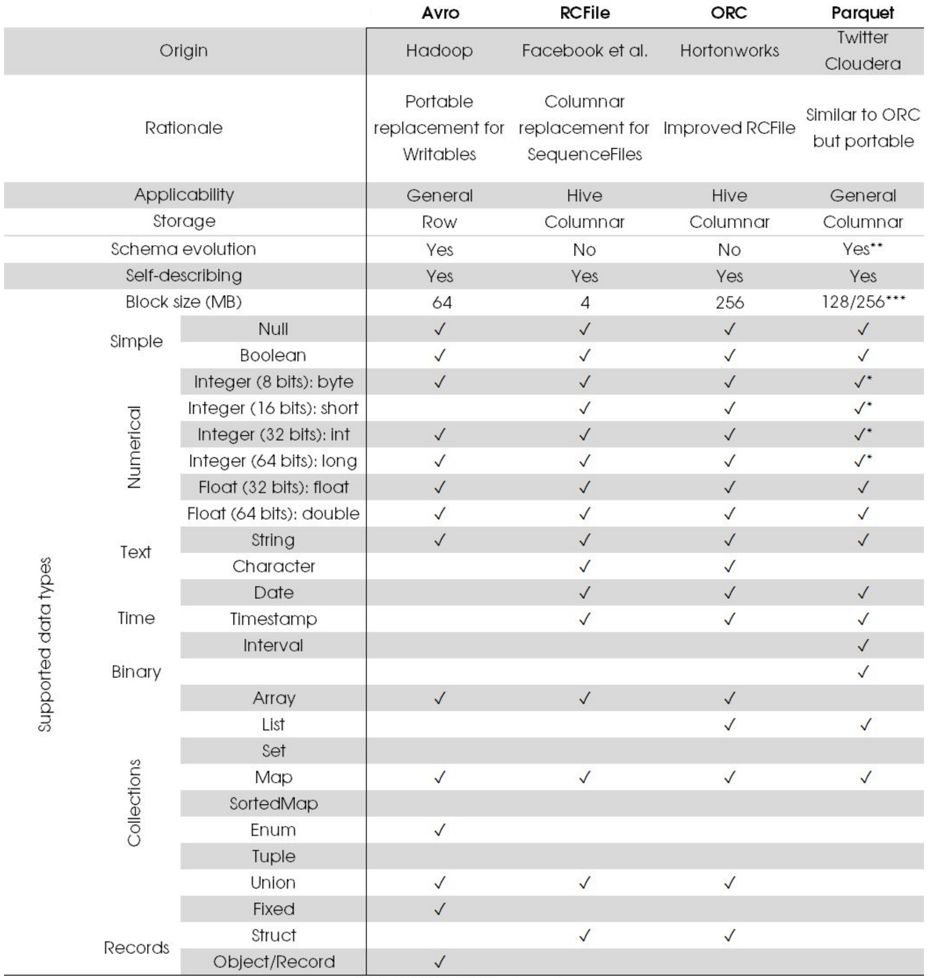
Reference slides

			Writables	Thrift	Protocol Buffers	Avro	RCFile	ORC	Parquet
Origin			Hadoop	Facebook	Google	Hadoop	Facebook et al.	Hortonworks	Twitter Cloudera
Rationale				Language-independent interfaces (IDL) to services	Data exchange between services through IDL	Portable replacement for Writables	Columnar replacement for SequenceFiles	Improved RCFile	Similar to ORC but portable
Applicability			Java	General	General	General	Hive	Hive	General
Storage			Row	Row	Row	Row	Columnar	Columnar	Columnar
Native MR support			Yes	No	No	Yes	Yes	Yes	Yes
Compressible			Yes	No	No	Yes	Yes	Yes	Yes
Splittable			Yes	No	No	Yes	Yes	Yes	Yes
Schema evolution			No	Yes	Yes	Yes	No	No	Yes**
Self-describing			No	No	No	Yes	Yes	Yes	Yes
Block size (MB)			-	-	-	64	4	256	128/256***
Supported data types	Simple	Null	✓		✓	✓	✓	✓	✓
		Boolean	✓	✓	✓	✓	✓	✓	✓
	Numerical	Integer (8 bits): byte	✓	✓		✓	✓	✓	✓*
		Integer (16 bits): short	✓	✓	✓*		✓	✓	✓*
		Integer (32 bits): int	✓	✓	✓*	✓	✓	✓	✓*
		Integer (64 bits): long	✓	✓	✓*	✓	✓	✓	✓*
		Float (32 bits): float	✓		✓*	✓	✓	✓	✓
		Float (64 bits): double	✓	✓	✓*	✓	✓	✓	✓
	Text	String	✓	✓	✓	✓	✓	✓	✓
		Character					✓	✓	
	Time	Date					✓	✓	✓
		Timestamp					✓	✓	✓
		Interval							✓
	Binary			✓					✓
	Collections	Array	✓			✓	✓	✓	
		List		✓				✓	✓
		Set		✓					
		Map	✓	✓	✓	✓	✓	✓	✓
		SortedMap	✓						
		Enum	✓		✓	✓			
		Tuple	✓						
		Union				✓	✓	✓	
		Fixed				✓			
	Records	Struct		✓			✓	✓	
		Object/Record	✓		✓	✓			

* Including unsigned data types / ** Append only / *** 256 MB in Impala

Compression types for different file formats

FileType	Compression
Text	LZO
Avro	Snappy, GZIP, deflate, BZIP2
Parquet	Snappy, GZIP, deflate, BZIP2
ORC	Snappy, ZLib
RCFile	Snappy, GZIP, deflate, BZIP2
SequenceFile	Snappy, GZIP, deflate, BZIP2



			Avro	RCFile	ORC	Parquet
Origin			Hadoop	Facebook et al.	Hortonworks	Twitter Cloudera
Rationale			Portable replacement for Writables	Columnar replacement for SequenceFiles	Improved RCFile	Similar to ORC but portable
Applicability			General	Hive	Hive	General
Storage			Row	Columnar	Columnar	Columnar
Schema evolution			Yes	No	No	Yes**
Self-describing			Yes	Yes	Yes	Yes
Block size (MB)			64	4	256	128/256***
Supported data types	Simple	Null	✓	✓	✓	✓
		Boolean	✓	✓	✓	✓
	Numerical	Integer (8 bits): byte	✓	✓	✓	✓*
		Integer (16 bits): short		✓	✓	✓*
		Integer (32 bits): int	✓	✓	✓	✓*
		Integer (64 bits): long	✓	✓	✓	✓*
		Float (32 bits): float	✓	✓	✓	✓
		Float (64 bits): double	✓	✓	✓	✓
	Text	String	✓	✓	✓	✓
		Character		✓	✓	
	Time	Date		✓	✓	✓
		Timestamp		✓	✓	✓
		Interval				✓
	Binary					✓
	Collections	Array	✓	✓	✓	
		List			✓	✓
		Set				
		Map	✓	✓	✓	✓
		SortedMap				
		Enum	✓			
		Tuple				
		Union	✓	✓	✓	
		Fixed	✓			
	Records	Struct		✓	✓	
		Object/Record	✓			

Each format supports different data types

Hadoop IO - loading data with input formats in Spark

To use TextInputFormat:

```
lines = sc.textFile(pathStr);
```

To use SequenceFileInputFormat:

```
records = sc.sequenceFile(pathStr, key.class, value.class);
```

EXAMPLE: READING A SEQUENCE FILE

```
import org.apache.hadoop.io.IntWritable

import org.apache.hadoop.io.Text

JavaPairRDD rdd = sc.sequenceFile(“mySequenceFile”), IntWritable.class,

Text.class)
```

Hadoop IO - saving in different formats in spark

To use TextInputFormat:

```
sc.saveAsTextFile(rdd);
```

To use SequenceFileInputFormat:

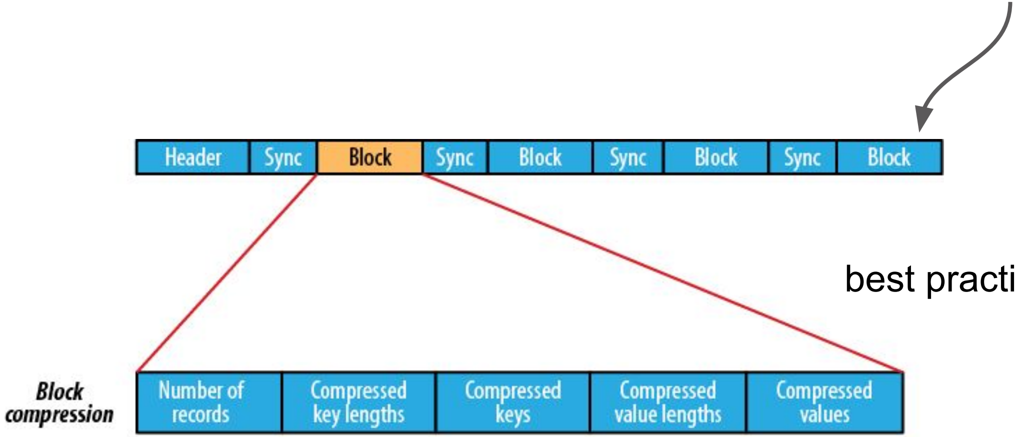
```
sc.saveAsHadoopFile(fileName,
    <key>.class,
    <value>.class,
    SequenceFileOutputFormat.class);
```

key and value classes must be Writables

Sequence files

- Store key, value pairs as records
- Compressible, splittable
- Self-aware header

writer.append(key, value);



Header | Sync | Block | Sync | Block | Sync | Block | Sync | Block

Block compression

Number of records | Compressed key lengths | Compressed keys | Compressed value lengths | Compressed values

best practice: create with block compression

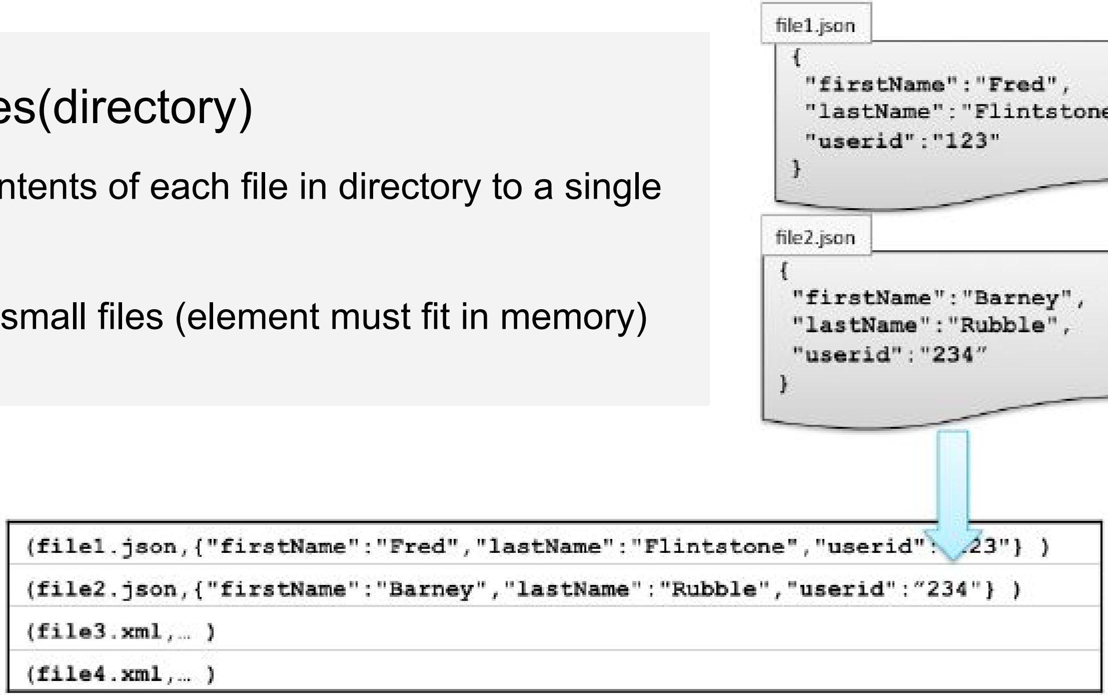
Examples of reading and writing SequenceFiles

See class-examples-io

- SaveAsSequenceFile.java - run this first, it will create a sequence file
  - To run, in Run Configurations, specify the name of the file. For example:
    data/panda.seq
  -
- LoadSequenceFile.java - run this next, it will load the sequence file and show you the contents

using wholeTextFiles: Handling small files

- sc.wholeTextFiles(directory)
    - Maps entire contents of each file in directory to a single RDD element
    - Works only for small files (element must fit in memory)



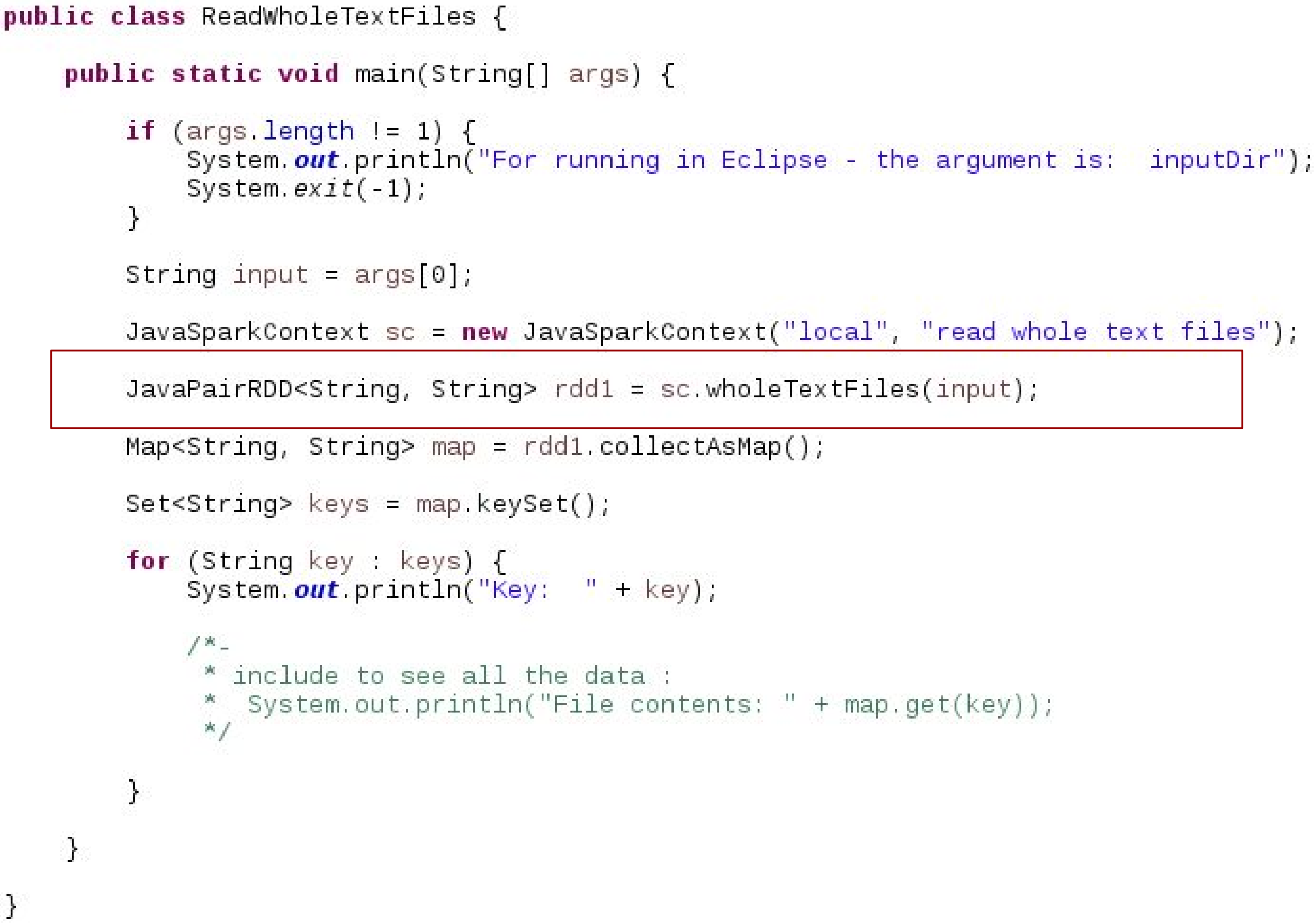
file1.json

```
{
 "firstName":"Fred",
 "lastName":"Flintstone",
 "userid":"123"
}
```

file2.json

```
{
 "firstName":"Barney",
 "lastName":"Rubble",
 "userid":"234”
}
```

```
(file1.json,{"firstName":"Fred","lastName":"Flintstone","userid":"123"} )
(file2.json,{"firstName":"Barney","lastName":"Rubble","userid":”234"} )
(file3.xml,… )
(file4.xml,… )
```



```java
public class ReadWholeTextFiles {

    public static void main(String[] args) {

        if (args.length != 1) {
            System.out.println("For running in Eclipse - the argument is:  inputDir");
            System.exit(-1);
        }

        String input = args[0];

        JavaSparkContext sc = new JavaSparkContext("local", "read whole text files");

        JavaPairRDD<String, String> rdd1 = sc.wholeTextFiles(input);

        Map<String, String> map = rdd1.collectAsMap();

        Set<String> keys = map.keySet();

        for (String key : keys) {
            System.out.println("Key:  " + key);

            /*-
             * include to see all the data :
             *  System.out.println("File contents: " + map.get(key));
             */

        }

    }

}
```

Examples of reading WholeTextFiles

See class-examples-io

- ReadWholeTextFiles.java - run this first with a directory of small files
  - To run, in Run Configurations, specify the name of the directory.
  - For example: **data/stock-sample-tiny**

  This will read each file in the directory and show you the files it has read.

Hadoop / Spark Input formats (1)

Avro files

```
newAPIHadoopFile(path, InputFormat class, key.class, value.class, [conf])
```

```
JavaPairRDD avroRecords = sc.newAPIHadoopFile(input,
AvroKeyInputFormat.class, Text.class, NullWritable.class,
new Configuration());
```

Hadoop /Spark Input formats (2)

KeyValueText files

`newAPIHadoopFile(path, InputFormat class, key.class, value.class, [conf])`

```
Configuration conf = new Configuration();
conf.set(“mapreduce.input.keyvaluelinerecordreader.key.value.separator”,
    ”:”);
JavaPairRDD keyValues = sc.newAPIHadoopFile(input,
    KeyValueTextInputFormat.class,Text.class, IntWritable.class,
    conf)
```

Hadoop / Spark input and output formats

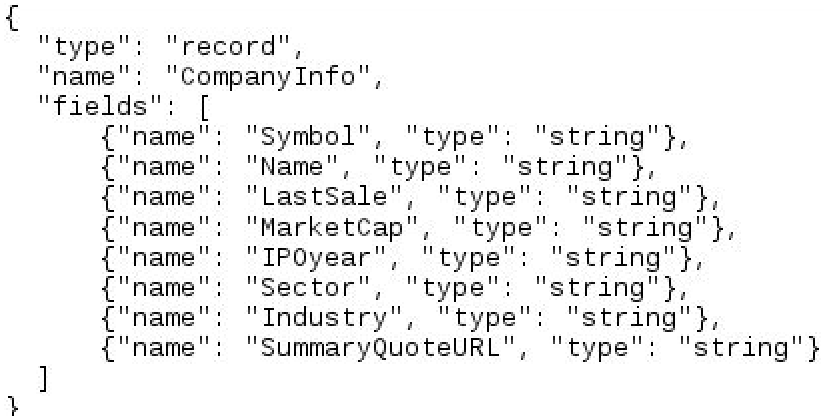
OK, I’m in the 3rd circle of Hell…

I hope I’m not alone.

Avro data

- Useful for evolving binary data, JSON based.
  - Schema evolution – Avro requires schemas written in JSON. Can use different schemas for serialization and deserialization.
  - Untagged data – Allows each datum be written without overhead: more compact data encoding, and faster data processing.
  - Dynamic typing – serialization and deserialization without code generation.

Avro file - companyInfo.avsc



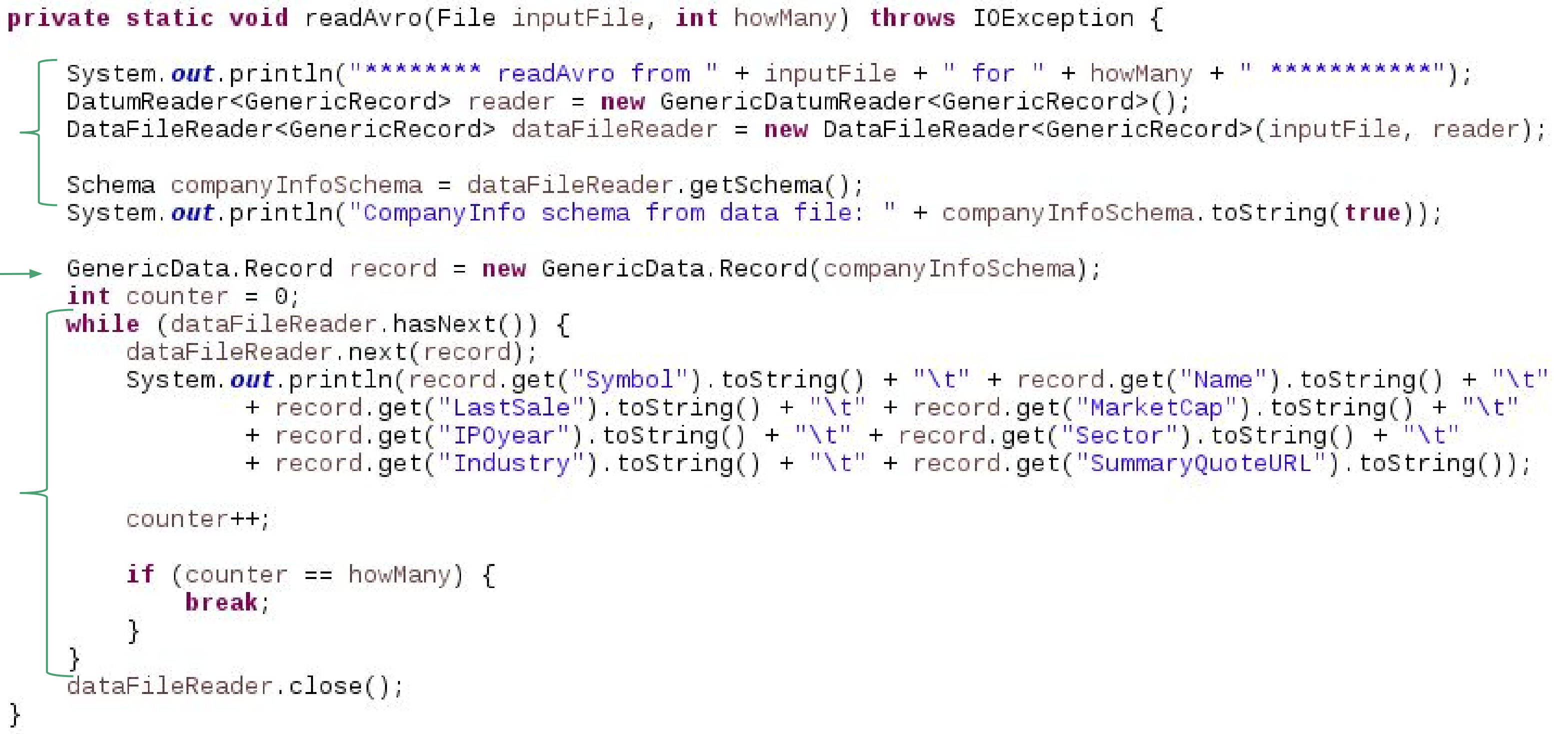
```
{
  "type": "record",
  "name": "CompanyInfo",
  "fields": [
      {"name": "Symbol", "type": "string"},
      {"name": "Name", "type": "string"},
      {"name": "LastSale", "type": "string"},
      {"name": "MarketCap", "type": "string"},
      {"name": "IPOyear", "type": "string"},
      {"name": "Sector", "type": "string"},
      {"name": "Industry", "type": "string"},
      {"name": "SummaryQuoteURL", "type": "string"}
  ]
}
```

This file is in src/main/resources/avro

Create reader for Avro file

Create record

Write records to System.out



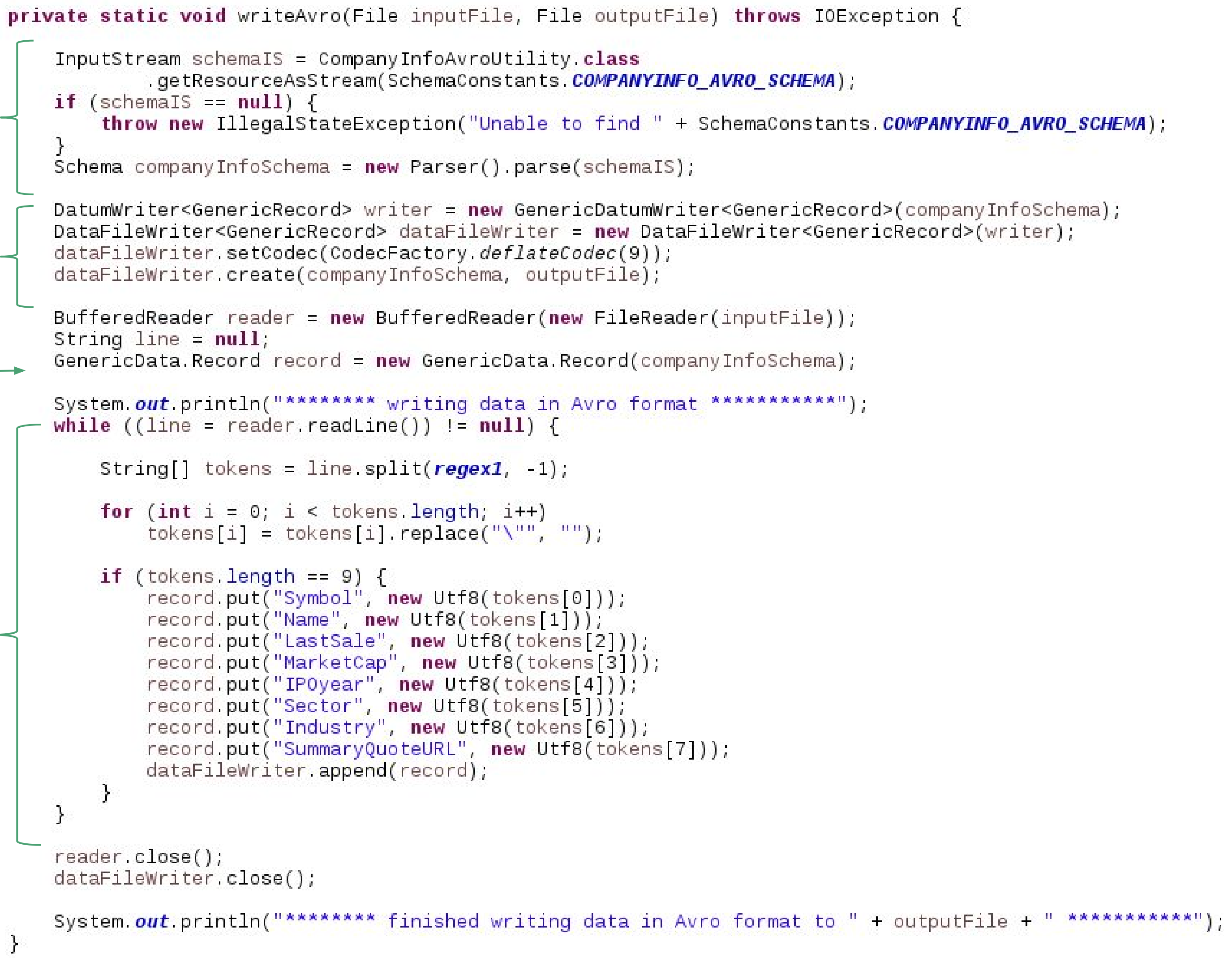
```
private static void readAvro(File inputFile, int howMany) throws IOException {

    System.out.println("******** readAvro from " + inputFile + " for " + howMany + " ***********");
    DatumReader<GenericRecord> reader = new GenericDatumReader<GenericRecord>();
    DataFileReader<GenericRecord> dataFileReader = new DataFileReader<GenericRecord>(inputFile, reader);

    Schema companyInfoSchema = dataFileReader.getSchema();
    System.out.println("CompanyInfo schema from data file: " + companyInfoSchema.toString(true));

    GenericData.Record record = new GenericData.Record(companyInfoSchema);
    int counter = 0;
    while (dataFileReader.hasNext()) {
        dataFileReader.next(record);
        System.out.println(record.get("Symbol").toString() + "\t" + record.get("Name").toString() + "\t"
                + record.get("LastSale").toString() + "\t" + record.get("MarketCap").toString() + "\t"
                + record.get("IPOyear").toString() + "\t" + record.get("Sector").toString() + "\t"
                + record.get("Industry").toString() + "\t" + record.get("SummaryQuoteURL").toString());

        counter++;

        if (counter == howMany) {
            break;
        }
    }
    dataFileReader.close();
}
```

Read and parse schema (.avsc)

Create writer

Create record

Read text fields and write to record



```
private static void writeAvro(File inputFile, File outputFile) throws IOException {

    InputStream schemaIS = CompanyInfoAvroUtility.class
            .getResourceAsStream(SchemaConstants.COMPANYINFO_AVRO_SCHEMA);
    if (schemaIS == null) {
        throw new IllegalStateException("Unable to find " + SchemaConstants.COMPANYINFO_AVRO_SCHEMA);
    }
    Schema companyInfoSchema = new Parser().parse(schemaIS);

    DatumWriter<GenericRecord> writer = new GenericDatumWriter<GenericRecord>(companyInfoSchema);
    DataFileWriter<GenericRecord> dataFileWriter = new DataFileWriter<GenericRecord>(writer);
    dataFileWriter.setCodec(CodecFactory.deflateCodec(9));
    dataFileWriter.create(companyInfoSchema, outputFile);

    BufferedReader reader = new BufferedReader(new FileReader(inputFile));
    String line = null;
    GenericData.Record record = new GenericData.Record(companyInfoSchema);

    System.out.println("******** writing data in Avro format ***********");
    while ((line = reader.readLine()) != null) {

        String[] tokens = line.split(regex1, -1);

        for (int i = 0; i < tokens.length; i++)
            tokens[i] = tokens[i].replace("\"", "");

        if (tokens.length == 9) {
            record.put("Symbol", new Utf8(tokens[0]));
            record.put("Name", new Utf8(tokens[1]));
            record.put("LastSale", new Utf8(tokens[2]));
            record.put("MarketCap", new Utf8(tokens[3]));
            record.put("IPOyear", new Utf8(tokens[4]));
            record.put("Sector", new Utf8(tokens[5]));
            record.put("Industry", new Utf8(tokens[6]));
            record.put("SummaryQuoteURL", new Utf8(tokens[7]));
            dataFileWriter.append(record);
        }
    }

    reader.close();
    dataFileWriter.close();

    System.out.println("******** finished writing data in Avro format to " + outputFile + " ***********");
}
```

To run example:

click on `CompanyInfoAvroUtility.java`

select “Run As” -> select “Run Configurations”

Arguments:

- To convert files:

  ```
  convert data/companylistNASDAQ.csv data/companylistNASDAQ.avro
  ```

- To read 25 files:

  ```
  read data/companylistNASDAQ.avro 25
  ```

For a deeper (but possibly out-of-date??) analysis of data flows in Hadoop

Hadoop Application Architectures

Designing Real-World Big Data Applications

By Mark Grover, Ted Malaska, Jonathan Seidman, Gwen Shapira
Publisher: O'Reilly Media
Final Release Date: June 2015
Pages: 400

O'Reilly link - to buy or browse

Still to discuss: Parquet

Serialization in Hadoop

Hadoop IO can serialize and deserialize app data of the following types:

- Java Serializable, with Kryo (Spark only)
- Avro
- Parquet
- ORC
- Writable
- SequenceFile

Used by Hadoop (internally) but not by apps in Spark, Hive or MR2

- Thrift
- Protocol buffers