Hadoop I/O

Data integrity in HDFS

HDFS transparently checksums all data for every io.bytes.per.checksum bytes of data (512 bytes). When writing or reading to a pipeline of datanodes, the last verifies checksum.

HDFS can "heal" corrupted blocks with the replicas. It is also possible to disable verification passing `false` to the `FileSystem.setVerifyChecksum()` or throught the command line using `-ignoreCrc`

LocalFileSystem

Hadoop creates a hidden *.crc file using ChecksumFileSystem everytime it writes a file with a chunk size set in io.bytes.per.checksum

Compression

When compressed, inputs will be decompressed automatically by MapReduce. To compress the output, set the `mapred.output.compress` to true and the `mapred.output.compression.codec` to the classname of the compression format you want to use

Compression format	Tool	Algorithm	Extension	Splittable
DEFLATE	N/A	DEFLATE	.deflate	No
gzip	gzip	DEFLATE	.gz	No
bzip2	bzip2	bzip2	.bz2	Yes
LZO	lzop	LZO	.lzo	Yes/No
Snappy	N/A	Snappy	.snappy	No

Compression and Input Splits

Splittable compression formats allow to search inside 16-blocks file when having only one of them locally. Depending on the application, one format is better suited than others. When storage costs are critical, you should use the highest compression even if it doesn't split and the same mapper has to process the 16 blocks. It can even compress the map output

Property name	Type	Default value	De
mapred.output.compress	boolean	false	Cor outp
mapred.output.compression.codec	Class name	org.apache.hadoop.io.compress.DefaultCodec	The com cod for
mapred.output.compression.type	String	RECORD	The com to u Sec

Serialization

Is turning structured objects into a byte stream for transmission over a network or writing to persistent storage. Deserialization is turning a byte stream back into a series of structured objects. Is used for interprocess communication (RPC) and for persistent storage.

RPC must be:

1. Compact
2. Fast
3. Extensible (evolve straightforward)
4. Interoperable (support clients written in diff languages)

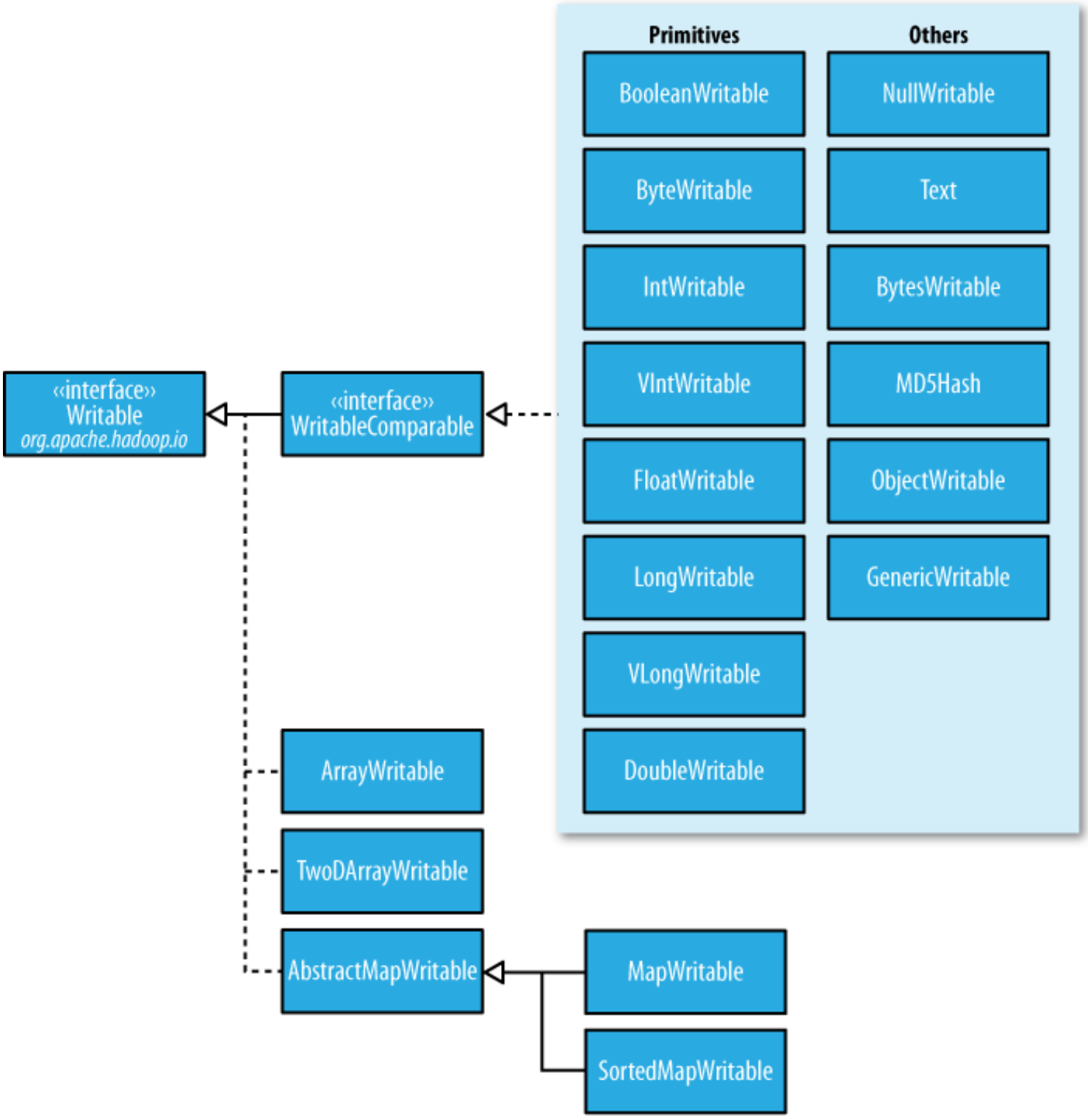
The Writable Interface

Writable and WritableComparable

2 methods for Writable interface: **DataOutput** and **DataInput**. WritableComparable is a subinterface of

Writable and java.lang.Comparable. Comparisons are crucial for sorting.

Writable Classes



When encoding integers you can use the fixed length (IntWritable) or variable (VIntWritable). Most numeric variables tend to have non uniform distributions so on average they will save space.

Text

Equivalent to Java String, maximum value is 2gb as it stores the number of bytes in an int.

BytesWritable

Wrapper for an array of binary data serialized as an 4-byte integer that specifies the bytes to follow and the bytes themselves. For example, length=2 (4 byte integer 00000002) with values 3 and 5 (03 and 05) == 000000020305

NullWritable

Zero length, singleton

ObjectWritable and GenericWritable

Wrapper for String, enum, Writable and null or arrays of any of them. **ObjectWritable** is useful when a field can be more than one type, for example in a SequenceFile with multiple types. If the number of types is known in advance, you can write an array of the classes and use **GenericWritable** instead

Writable Collections

- ArrayWritable and TwoDArrayWritable, for arrays and 2d Writables. They must all be the same instances of the same class.
- ArrayPrimitiveWritable wrapper for Java primitives arrays.
- MapWritable (java.utilMap) and SortedMapWritable (java.util.SortedMap) EnumSetWritable.

Implementing a Custom Writable

1. It's needed to extend **WritableComparable** to have the **compareTo()**
2. Have an empty constructor
3. Populates the fields calling **readFields()
4. Have a **write()** method
5. Override **hashCode(), equals()** and **toString()**

Serialization Frameworks

A Serialization defines a mapping from types to Serializer instances (for turning an object into a byte stream)

Avro

Resume in here: http://blog.cloudera.com/blog/2009/11/avro-a-new-format-for-data-interchange/

Language-neutral serialization system to be processed by many languages (C, C++, Python...). Uses *schemas* in JSON but code generation is optional (you can read data that conforms to a given schema even

if your code has not seen that schema before) and provides API's for des/serialization

It has common data types (null, boolean, int, bytes, string...) and some complex types (array, map, enum...). Typical Avro schema with types:

```
{
    "type":"record",
    "name":"StringPair",
    "doc": "A pair of strings.",
    "fields":{
        {"name":"left", "type":"string"},
        {"name":"right", "type":"string"}
    }
}
```

Then to load, use and serialize a record in Java (the use will be similar in Python, for example):

```
//Loads
Schema.Parser parser = new Schema.Parser();
Schema schema = parser.parse(getClass().getResourceAsStream("StringPair.avsc"));

//Creates Avro instance
GenericRecord datum = new GenericData.Record(schema);
datum.put("left", "L");
datum.put("right", "R");

//Serialize to an output stream
ByteArrayOutputStream out = new ByteArrayOutputStream();
DatumWriter<GenericRecord> writer = new GenericDatumWriter<GenericRecord>(schema);
Encoder encoder = EncoderFactory.get().binaryEncoder(out, null);
writer.write(datum, encoder);
encoder.flush();
out.close();
```

TODO?

File-Based Data Structures

SequenceFile

- **Writing**: Good to persist binary data structure for key-value pairs. To write a SequenceFile use a **createWriter()** method which returns a SequenceFile.Writer where we use the **append()** method (optional arguments include compression and codec).

- **Reading**: With a **SequenceFile.Reader** and iterating with **next()** if they are Writable types or also **getCurrentValue(Object val)** if using non-Writable.
- **Seeking / finding**: Using **seek()** to retrieve a given position in a sequence file.

Command line actions

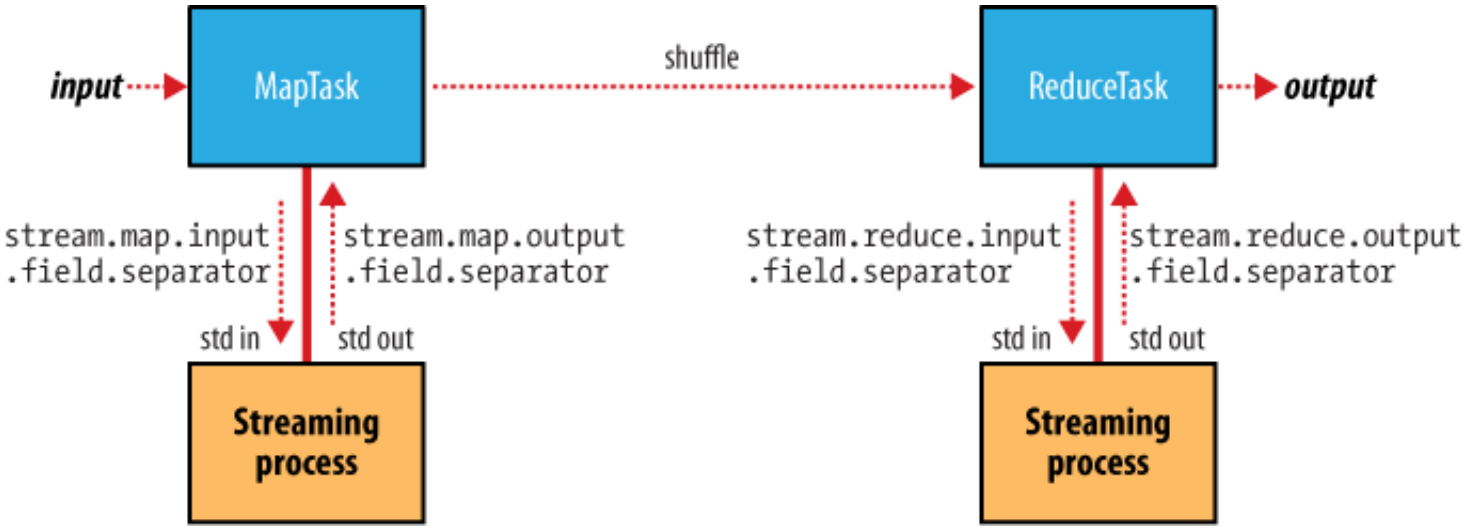
If a SequenceFile has a meaningful text representation it can be read as follows:

```
hadoop fs -text numbers.seq | head
```

And can be sorted using MapReduce job

The SequenceFile format

Consists of a header followed by one or more records, it also contains other fields including the *sync* marker that allow a reader to synchronize to a record boundary.



MapFile

Is a sorted SequenceFile with an index por permit lookups by key. Is written with an instance of **MapFile.Writer** and calling **append()** to add entries in order.