BIG DATA SERIALIZATION FORMAT

Characteristics when selecting a data serialization format

- *Code generation*—Some serialization formats are accompanied by libraries with code-generation abilities that allow you to generate rich objects, making it easier for you to interact with your data. The generated code also provides the added benefit of type-safety to make sure that your consumers and producers are working with the right data types.

- *Schema evolution*—Data models evolve over time, and it's important that your data formats support your need to modify your data models. Schema evolution allows you to add, modify, and in some cases delete attributes, while at the same time providing backward and forward compatibility for readers and writers.

- *Language support*—It's likely that you'll need to access your data in more than one programming language, and it's important that the mainstream languages have support for a data format.

Characteristics when selecting a data serialization format (Contd)

- *Transparent compression*—Data compression is important given the volumes of data you'll work with, and a desirable data format has the ability to internally compress and decompress data on writes and reads. It's a much bigger headache for you as a programmer if the data format doesn't support compression, because it means that you'll have to manage compression and decompression as part of your data pipeline (as is the case when you're working with text-based file formats).

- *Splittability*—Newer data formats understand the importance of supporting multiple parallel readers that are reading and processing different chunks of a large file. It's crucial that file formats contain synchronization markers (and thereby support the ability for a reader to perform a random seek and scan to the start of the next record).

- *Support in MapReduce and the Hadoop ecosystem*—A data format that you select must have support in MapReduce and other critical Hadoop ecosystem projects, such as Hive. Without this support, you'll be responsible for writing the code to make a file format work with these systems.

Feature comparison of data serialization frameworks

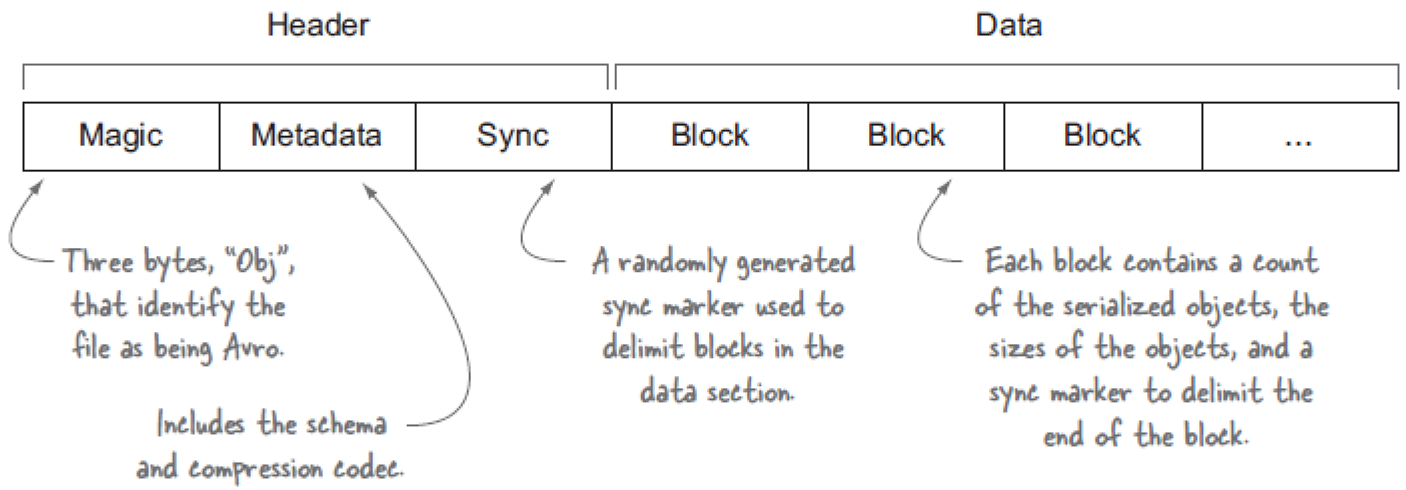
Library	Code generation	Schema evolution	Language support	Transparent compression	Splittable	Native support in MapReduce	Pig and Hive support
Sequence-File	No	No	Java, Python	Yes	Yes	Yes	Yes
Protocol Buffers	Yes (optional)	Yes	C++, Java, Python, Perl, Ruby	No	No	No	No
Thrift	Yes (mandatory)	Yes	C, C++, Java, Python, Ruby, Perl	No[a]	No	No	No
Avro	Yes (optional)	Yes	C, C++, Java, Python, Ruby, C#	Yes	Yes	Yes	Yes
Parquet	No	Yes	Java, Python (C++ planned in 2.0)	Yes	Yes	Yes	Yes

[a] Thrift does support compression, but not in the Java library.

Avro

- Doug Cutting created Avro, a data serialization and RPC library, to help improve data interchange, interoperability, and versioning in MapReduce
- Avro utilizes a compact binary data format—which you have the option to compress—that results in fast serialization Times
- It embeds the schema in the container file format, allowing for dynamic discovery and data interactions.

Avro container file format


Header
Data
Magic
Metadata
Sync
Block
Block
Block
...
Three bytes, "Obj", that identify the file as being Avro.
Includes the schema and compression codec.
A randomly generated sync marker used to delimit blocks in the data section.
Each block contains a count of the serialized objects, the sizes of the objects, and a sync marker to delimit the end of the block.

- The schema is serialized as part of the header, which makes deserialization simple and loosens restrictions around users having to maintain and access the schema outside of the Avro data files being interacted with.
- Each data block contains a number of Avro records, and by default is 16 KB in size.
- Data serialization supports code generation, versioning, and compression, schema evolution and has a high level of integration with MapReduce

Java program to read and write Avro data from and to streams

Simple Avro schema for representing a pair of strings as a record

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

Load this schema using the following

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

Java program to read and write Avro data from and to streams (Contd)

```
create an instance of an Avro record using the Generic API as follows:
GenericRecord datum = new GenericData.Record(schema);
datum.put("left", "L");
datum.put("right", "R");


Next, we serialize the record 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();
```

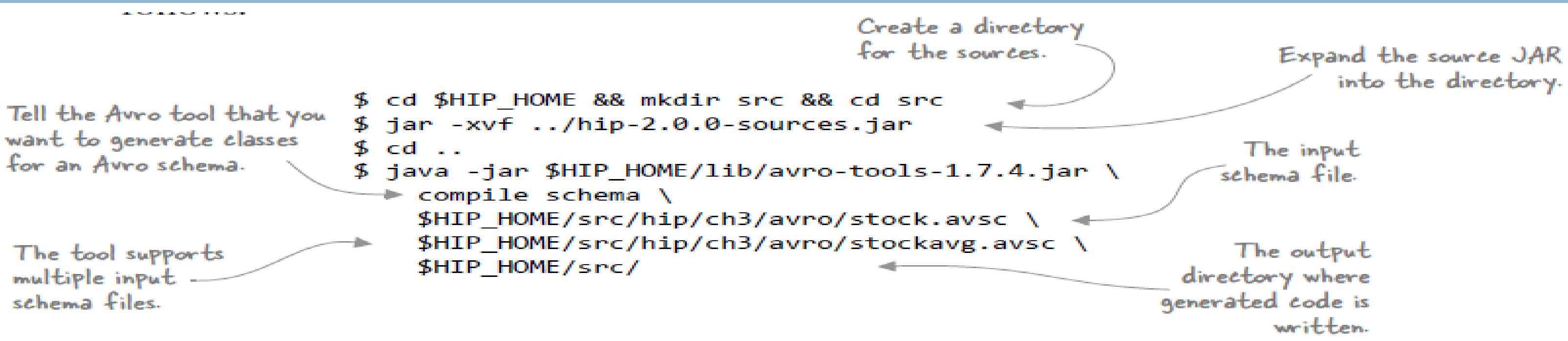
- A DatumWriter translates data objects into the types understood by an Encoder, which the latter writes to the output stream
- We pass a null to the encoder factory because we are not reusing a previously constructed encoder here

Java program to read and write Avro data from and to streams (Contd)

We can reverse the process and read the object back from the byte buffer

```
DatumReader<GenericRecord> reader = new GenericDatumReader<GenericRecord>(schema);
Decoder decoder = DecoderFactory.get().binaryDecoder(out.toByteArray(),null);
GenericRecord result = reader.read(null, decoder);
assertThat(result.get("left").toString(), is("L"));
assertThat(result.get("right").toString(), is("R"));
```

Using SpecificDatumReader



```
$ cd $HIP_HOME && mkdir src && cd src
$ jar -xvf ../hip-2.0.0-sources.jar
$ cd ..
$ java -jar $HIP_HOME/lib/avro-tools-1.7.4.jar \
    compile schema \
    $HIP_HOME/src/hip/ch3/avro/stock.avsc \
    $HIP_HOME/src/hip/ch3/avro/stockavg.avsc \
    $HIP_HOME/src/
```

Create a directory for the sources.
Expand the source JAR into the directory.
Tell the Avro tool that you want to generate classes for an Avro schema.
The input schema file.
The tool supports multiple input schema files.
The output directory where generated code is written.

```
StringPair datum = new StringPair();
datum.setLeft("L");
datum.setRight("R");
ByteArrayOutputStream out = new ByteArrayOutputStream();
DatumWriter<StringPair> writer=new
SpecificDatumWriter<StringPair>(StringPair.class);
Encoder encoder = EncoderFactory.get().binaryEncoder(out, null);
writer.write(datum, encoder);
encoder.flush();
out.close();
DatumReader<StringPair> reader = new
SpecificDatumReader<StringPair>(StringPair.class);
Decoder decoder = DecoderFactory.get().binaryDecoder(out.toByteArray(),null);
StringPair result = reader.read(null, decoder);
assertThat(result.getLeft(), is("L"));
assertThat(result.getRight(), is("R"));
```