# CS523 - BDT
# Big Data Technologies

## Apache Avro

**(Hadoop's Cross-language Data Serialization System)**

# Serialization

- ***Serialization*** - process of turning structured objects into a byte stream for transmission over a network or for writing to persistent storage.
- ***Deserialization*** **-** reverse process of turning a byte stream back into a series of structured objects.
- **Use of Serialization -** for interprocess communication and for persistent storage.
- In Hadoop, interprocess communication between nodes in the system is implemented using *remote procedure calls* (RPCs).
- The RPC protocol uses serialization to render the message into a binary stream to be sent to the remote node, which then deserializes the binary stream into the original message.

# Desired Characteristics of an RPC Serialization Format

- ***Compact –*** making the best use of n/w bandwidth, which is the most scarce resource in a data center.
- ***Fast -*** Interprocess communication forms the backbone for a distributed system, so it is essential that there is as little performance overhead as possible for the serialization and deserialization process.
- ***Extensible -*** Protocols change over time to meet new requirements, so it should be straightforward to evolve the protocol in a controlled manner for clients and servers.
  - For example, it should be possible to add a new argument to a method call and have the new servers accept messages in the old format (without the new argument) from old clients.
- ***Interoperable -*** For some systems, it is desirable to be able to support clients that are written in different languages to the server, so the format needs to be designed to make this possible.

# Desired Characteristics for Persistent Storage Format

- The data format chosen for persistent storage would have different requirements from a serialization framework.
  - After all, the lifespan of an RPC is less than a second, whereas persistent data may be read years after it was written.
- But it turns out, the four desirable properties of an RPC's serialization format are also crucial for a persistent storage format.
- We want the storage format to be **compact** (to make efficient use of storage space), **fast** (so the overhead in reading or writing terabytes of data is minimal), **extensible** (so we can transparently read data written in an older format), and **interoperable** (so we can read or write persistent data using different languages).

# Java Serialization

- Java comes with its own serialization mechanism, called Java Object Serialization (often referred to simply as "Java Serialization"), that is tightly integrated with the language.
- In Big Data processing scenarios, usually need to have a precise control over exactly how objects are written and read.
- With Serialization you can get some control, but you have to fight for it. The logic for not using RMI was similar.
- Effective, high-performance inter-process communications are critical to Hadoop. Need to precisely control how things like connections, timeouts and buffers are handled, and RMI gives you little control over those.
- The problem is that Java Serialization doesn't meet the criteria for a serialization format listed earlier: compact, fast, extensible, and interoperable.

# Java Serialization Problems

- Java Serialization is not compact: it writes the classname of each object being written to the stream.
- Subsequent instances of the same class write a reference handle to the first occurrence, which occupies only 5 bytes.
- However, reference handles don't work well with random access, since the referent class may occur at any point in the preceding stream—that is, there is state stored in the stream.
- Even worse, reference handles play havoc with sorting records in a serialized stream, since the first record of a particular class is distinguished and must be treated as a special case.

# Java Serialization Problems

- Java Serialization is a general-purpose mechanism for serializing graphs of objects, so it necessarily has some overhead for serialization and deserialization operations.
- What's more, the deserialization procedure creates a new instance for each object deserialized from the stream!

To avoid all these problems, Hadoop uses it's own Writables!

# Advantage of Hadoop over Java Serialization

- In java serialization process, it writes meta data about the object which includes the class name, field name and types and its super class.
- Where as in Hadoop Serialization mechanism, while defining '*Writable*' we(applications) know the expected class.
- So Writables do not store their type in the serialized representation as while deserializing.
  - For example: if the input key is LongWritable instance , so an empty LongWritable instance is asked to populate itself from the input data stream.
- As no Meta info need to be stored this results in more compact binary files, random access and high performance.

# Advantage of Hadoop over Java Serialization

- Hadoop's Writable-based serialization is capable to reduce the object-creation overhead by reusing the Writable objects, which is not possible with the Java's native serialization framework.
- For example, for a MapReduce job, which at its core serializes and deserializes billions of records of just a handful of different types, the savings gained by not having to allocate new objects are significant.

# Disadvantages of Hadoop Serialization

- To serialize Hadoop data, there are two ways:
  - You can use the **Writable** classes, provided by Hadoop's native library.
  - You can also use **Sequence Files** which store the data in binary format.
- The main drawback of these two mechanisms is that Writables and SequenceFiles have only a Java API and they cannot be written or read in any other language.
- Therefore any of the files created in Hadoop with above two mechanisms cannot be read by any other third language!
- To address this drawback, Doug Cutting created Avro, which is a language independent data structure.
- Only Writable or Avro objects can be serialized or deserialized out of the box in Hadoop.

# Apache Avro


- Avro is a preferred tool to serialize data in Hadoop which allows to serialize data in a format that has a schema built in.
- Including schemas with the Avro messages allows any application to deserialize the data.
- The serialized data is in a compact binary format that doesn't require proxy objects or code generation. But if required you can use Avro tools to generate proxy objects in Java to easily work with the objects.
- Avro datafiles are widely supported across components in the Hadoop ecosystem (Pig, Hive, Spark,etc.), so they are a good default choice for a binary format. Avro is also the best choice for Kafka.
- Avro currently supports languages such as Java, C, C++, C#, Python, and Ruby.

# Apache Avro


- Main uses of Apache Avro are for:
  - Data serialization/deserialization
  - Data exchange (RPC)

# Avro Schema (.avsc)

- Avro data format (wire format and file format) is defined by Avro schemas.
- Avro, being a schema-based serialization utility, accepts schemas as input. (.avsc)
- Avro schemas are usually written in **JSON**, and data is usually encoded using a binary format, but there are other options, too.
- There is a higher-level language called *Avro IDL* for writing schemas in a C-like language.
- Avro allows every data to be written with no prior knowledge of the schema.
- Schemas are composed of primitive types (null, boolean, int, long, float, double, bytes, and string) and complex types (record, enum, array, map, union, and fixed).

# Avro Schema

- Avro is defined by a schema (schema is written in JSON)

```
// a simple three-element record
{"name": "Block", "type": "record":,
 "fields": [
   {"name": "id", "type": "string"},
   {"name": "length", "type": "integer"},
   {"name": "hosts", "type":
     {"type": "array:, "items": "string"}}
 ]
}

// a linked list of strings or ints
{"name": "MyList", "type": "record":,
 "fields": [
   {"name": "value", "type": ["string", "int"]},
   {"name": "next", "type": ["MyList", "null"]}
 ]
}
```

# Avro Data Files (.avro)

- An Avro *datafile* (.avro) is language neutral and has a metadata section where the schema is stored, which makes the file self-describing.
- Avro datafiles are like sequence files in that they are designed for largescale data processing—they are compact and splittable—but they are portable across different programming languages.
- The Avro files include markers that can be used for splitting large datasets into subsets (HDFS blocks) which makes it very suitable for MapReduce processing.
- Objects stored in Avro datafiles are described by a schema, rather than in the Java code of the implementation of a Writable object (as is the case for sequence files), making the Writable very Java-centric.
- The system can then generate types for different languages, which is good for interoperability.
- Avro datafiles also support compression.
- Avro datafiles store only records (not key/value pairs).

# Avro Data Types

- Avro provides rich data structures. E.g., you can create a record that contains an array, an enumerated type, and a sub record. These datatypes can be created in any language, can be processed in Hadoop, and the results can be fed to a third language.
- Each Avro language API has a representation for each Avro type that is specific to the language.
- For example, Avro's double type is represented in C, C++, and Java by a double, in Python by a float, and in Ruby by a Float.
- All languages support a dynamic mapping, which can be used even when the schema is not known ahead of runtime.
- **Java calls this the *Generic* mapping.**

# Avro Data Types

- In addition, the Java and C++ implementations can generate code to represent the data for an Avro schema.
- Code generation, which is called the ***Specific*** mapping in Java, is an optimization that is useful when you have a copy of the schema before you read or write data.
- Generated classes also provide a more domain-oriented API for user code than Generic ones.
- Java has a third mapping, the ***Reflect*** mapping, which maps Avro types onto preexisting Java types using reflection.
  - It is slower than the Generic and Specific mappings but can be a convenient way of defining a type, since Avro can infer a schema automatically.

# Avro Primitive Data Types

- Avro defines a small number of primitive data types, which can be used to build application-specific data structures by writing schemas.
- For interoperability, implementations must support all Avro types.

| Type | Description | Schema |
|---|---|---|
| null | The absence of a value | "null" |
| boolean | A binary value | "boolean" |
| int | 32-bit signed integer | "int" |
| long | 64-bit signed integer | "long" |
| float | Single-precision (32-bit) IEEE 754 floating-point number | "float" |
| double | Double-precision (64-bit) IEEE 754 floating-point number | "double" |
| bytes | Sequence of 8-bit unsigned bytes | "bytes" |
| string | Sequence of Unicode characters | "string" |

# Avro Complex Data Types

- **array**
  - An ordered collection of objects. All objects in a particular array must have the same schema.
- **Schema example**

```
{
    "type": "array",
    "items": "long"
}
```

- **map**
  - An unordered collection of key-value pairs. Keys must be strings and values may be any type, although within a particular map, all values must have the same schema.
- **Schema example**

```
{
    "type": “map",
    “values": “string"
}
```

# Avro Complex Data Types

- **enum**
  - A set of named values.
- **Schema example**

```
{
   "type": "enum",
   "name": "Cutlery",
   "doc": "An eating utensil.",
   "symbols": ["KNIFE",
               "FORK","SPOON"]
}
```

- **fixed**
  - A fixed number of 8-bit unsigned bytes.
  - **Schema example**

```
{
   "type": "fixed",
   "name": "Md5Hash",
   "size": 16
 }
```

# Avro Complex Data Types

- **record**
  - A collection of named fields of any type.
  - **Schema example**

```
{
    "type"      : "record",
    "name"      : "WeatherRecord",
    "doc"       : "A weather reading.",
    "fields": [
                {"name": "year",  "type": "int"},
                {"name": "temperature",  "type": "int"},
                {"name": "stationId",   "type": "string"}
              ]
 }
```

# Avro Complex Data Types

- **union**
  - A union of schemas. A union is represented by a JSON array, where each element in the array is a schema.
  - Data represented by a union must match one of the schemas in the union.
  - **Schema example**

```
[
   "null",
   "string",
   {"type": "map",
    "values": "string"}
]
```

# General Working of Avro

To use Avro, you need to follow the given workflow:

- **Step 1:** Create schemas. Here you need to design Avro schema according to your data.
- **Step 2:** Read the schemas into your program. It is done in two ways:
  - **By Generating a Class Corresponding to Schema** – Compile the schema using Avro. This generates a class file corresponding to the schema. (code generation)
  - **By Using Parsers Library –** You can directly read the schema using parsers library.
- **Step 3:** Serialize the data using the serialization API provided for Avro, which is found in the package org.apache.avro.generic.
- **Step 4:** Deserialize the data using deserialization API provided for Avro, which is found in the package org.apache.avro.generic.

# Use of Class AvroJob
## (org.apache.avro.mapreduce.AvroJob)

- AvroJob class has utility methods for configuring jobs that work with Avro.
- When using Avro data as MapReduce keys and values, data must be wrapped in a suitable **AvroWrapper** implementation.
- MapReduce keys must be wrapped in an **AvroKey** object, and MapReduce values must be wrapped in an **AvroValue** object.
- E.g. in wordcount program, if instead of using a *Text* and *IntWritable* output value, you would like to use Avro data with a schema of "string" and "int", respectively, you may parameterize your reducer with *AvroKey<String>* and *AvroValue<Integer>* types.
- Then, use the **setOutputKeySchema()** and **setOutputValueSchema()** methods of *AvroJob* to set writer schemas for the records you will generate.

# Station-Temp-Year Example

- The output needs to be in the following format:

| StationId | Temp | Year |
|---|---|---|
| 029720-99999 | 317 | 1901 |
| 227070-99999 | 244 | 1902 |
| ..... | | |

- Create an Avro schema in the above record format and pass Avro Key from the mapper adhering to this schema.

# Weather.avsc

```
{
    "type"      : "record",
    "name"      : "WeatherRecord",
    "doc"       : "A weather reading.",
    "fields"    : [
            {"name": "stationId",   "type": "string"},
            {"name": "temperature", "type": "int"},
            {"name": "year",  "type": "int"}
            ]
 }
```

# Compiling the Schema (with code generation)

- Code generation allows us to automatically create classes based on our previously-defined schema.
- Once we have defined the relevant classes, there is no need to use the schema directly in our programs.

- We use the avro-tools jar to generate code as follows:

```
avro-tools compile schema
/home/cloudera/cs523/Examples/Avro/weather.avsc
/home/cloudera/cs523/Examples/Avro
```

# Without Code Generation

- Data in Avro is always stored with its corresponding schema, meaning we can always read a serialized item regardless of whether we know the schema ahead of time.
- This allows us to perform serialization and deserialization without code generation.
- First, we use a Parser to read our schema definition and create a *Schema* object.

```
Schema SCHEMA = new Schema.Parser().parse(new File("weather.avsc"));
```

- Using this ***SCHEMA***, we'll create weather records from the given input file.

# Without Code Generation

- Since we're not using code generation, we use interface *GenericRecord* (org.apache.avro.generic) to represent weather records.
- *GenericRecord* uses the schema to verify that we only specify valid fields. If we try to set a non-existent field, we'll get an *AvroRuntimeException* when we run the program.
- Create the object of GenericRecord interface, by instantiating *GenericData.Record* class. Pass the schema object to its constructor.

```
GenericRecord record = new GenericData.Record(SCHEMA);

record.put("year", parser.getYearInt());
record.put("temperature", parser.getAirTemperature());
record.put("stationId", parser.getStationId());

context.write(new AvroKey<Integer>(parser.getYearInt()), new
AvroValue<GenericRecord>(record));
```

# Schema Evolution

- In a fast changing environment it should be possible to
  - write data to file with a schema
  - change the schema
    - Add extra fields
    - Delete fields
    - Rename fields
  - and still read the written file with changed schema
- Avro has rich *schema resolution* capabilities. The schema used to read data need not be identical to the schema that was used to write the data.
- This is the mechanism by which Avro supports *schema evolution*.
- Note that when you change schema, the new field must be given a default value. This prevents errors when clients using an old version of the schema create new values that will be missing the new field.

# Schema Evolution

- For example, a new, optional field may be added to a record by declaring it in the schema used to read the old data.
- New and old clients alike will be able to read the old data, while new clients can write new data that uses the new field.
- Conversely, if an old client sees newly encoded data, it will gracefully ignore the new field and carry on processing as it would have done with old data.