Attune University

Getting Started with Apache Avro

By Reeshu Patel

Introduction Apache Avro

Apache Avro is a remote procedure call and serialization framework developed with Apache's Hadoop project. This is uses JSON for defining data types and protocols, and tend to serializes data in a compact binary format. In other words, Apache Avro is a data serialization system. Its frist native use is in Apache Hadoop, where it's provide both a serialization format for persistent data, and a correct format for communication between Hadoop nodes, and from client programs to the apache Hadoop services.

Avro is a data serialization system.It'sprovides:

- Rich data structures.
- A compact, fast, binary data format.
- A container file, to store persistent data.
- Remote procedure call .
- It's easily integration with dynamic languages. Code generation is not mendetory to read or write data files nor to use or implement Remote procedure call protocols. Code generation is as an optional optimization, only worth implementing for statically typewritten languages.

Schemas of Apache Avro

When Apache avro data is read, the schema use when writing it's always present. This permits every datum to be written in no per-value overheads, creating serialization both fast and small. It also facilitates used dynamic, scripting languages, and data, together with it's schema, is fully itself-describing.

When Apache avro data is storein a file, it's schema is store with it, so that files may be processe later by any program. If the program is reading the data expects a different schema this can be simply resolved, since twice schemas are present.

When Avro is used in Remote procedure call, the client and server exchange schemas in between the connection handshake. (This may be optimized so that, for most calls, no schemas is completely transmitted.) Since both client and server both have the another's full schema, correspondence in between same named field, missing fields and extra fields, etc. can all be simply resolved

How to Comparison Apache Avro with other systems

Apache avro provides functionality same as to systems such as Thrift, Protocol Buffers, etc. Apache avro different from these systems in the following fundamental way.

- Dynamic typing: Avro does't require that code be create. Data always accompanied by a schema that permits full processing of that data without generationing cod, static datatypes, etc. it facilitates construction of generic data-processing systems and languages.
- Untagged data: Since the schema is present when It's data read, considerably minimum type information requeired be encoded with data, finalizing in small serialization size.
- No manually-assigned field IDs: When a schema modifly, both the old and new schema are always present when It'sprocessing data, so differ may be resolved symbolically, it's using field names.

Download Apache Avro

Apache avro implementations for C, C++, C#, Java, PHP, Python, and Ruby may be downloaded from the Avro Releases page. Here we uses Avro 1.7.5, the New version at the time of writing. For In this example you can download avro-1.7.5.jar and avro-tools-1.7.5.jar. The Apache avro Java Installation also depends on the Jackson JSON library. From the Jackson download page, Now download the core-asl and mapper-asl jars. And add avro-1.7.5.jar and the Jackson jars to our project's classpath (avro-tools can be use for code generation).

Thought, if you will be using Maven, then add the following dependency to your POM:

```
<dependency>

  <groupId>org.apache.avro</groupId>

  <artifactId>avro</artifactId>

  <version>1.7.5</version>

</dependency>
```

How to Define a Avro schema useing JSON

Apache avro schema is defined within JSON. Schema is composed of primitive types (null, boolean, int, long, float, double, bytes, and string) and complex types (record, enum, array, map, union, and fixed). You may learn more about Apache avro schemas and types from the specification, but for now we can start with a easy schema example to user.avsc file

```
{"namespace": "example.avro",

"type": "record",

"name": "User",

"fields": [

{"name": "name", "type": "string"},

{"name": "favorite_number", "type": ["int", "null"]},

{"name": "favorite_color", "type": ["string", "null"]}

]

}
```

Here this schema defines a record showing a hypothetical user. (Note that a schema file may only contain a single schema definition.) At less, a record definition must include it is type ("type": "record"), a name ("name": "User"), fields, and case name, favorite_number, with favorite_color. We can also define a namespace ("namespace": "example.avro"), which we together with the name attribute defines the "full name" of the schema (example.avro.User in this case).

Serializing and deserializing with code generation

Compiling the schema

Code generation allows us to itselfe create classes based on our frist-defined schema. Once we have defined the relevant classes, there is no required to use the schema directly in your programs. We used the avro-tools jar to create code as follows:

```
java -jar /path/to/avro-tools-1.7.5.jar compile schema <schema file> <destination>
```

This will be generating the appropriate source files in a package based on the schemas snamespace in the provid end folder. For instance, to generate a User class in a package with example.avro from the schema defined as run

```
java -jar /path/to/avro-tools-1.7.5.jar compile schema user.avsc .
```

Note that if you are using the Apache avro Maven plugin, there is no required to manually invoke the schema compiler .the plugin generationing code on any type .avsc files present in the configured with source directory.

Creating Users

Now that we have completed the code creation, now we create some users, serialize with them to a data file on disk, and then read back again the file and deserialize the User objects.

First we will create some Users and set their fields.

```
User user1 = new User();

user1.setName("Alyssa");
```

```
user1.setFavoriteNumber(256);

// Leave favorite color null



// Alternate constructor

User user2 = new User("Ben", 7, "red");



// Construct via builder

User user3 = User.newBuilder()

          .setName("Charlie")

          .setFavoriteColor("blue")

          .setFavoriteNumber(null)

          .build();
```

As you can see in above example, Apache avro objects can be create either by invoking a constructor directly or with in a builder. Unlike constructors, builders can automatically set any default values specifie in the schema. Thought , builders validate the data as it set, whereas objects constructed directly will not cause an error until the object iwill be serialized. thought, using constructors directly generally offers good performance, as builders make a copy of the datastructure before it's written.

Serializing

Now we can serialize your Users to disk.

```
// Serialize user1 and user2 to disk

File file = new File("users.avro");

DatumWriter<User> userDatumWriter = new
SpecificDatumWriter<User>(User.class);

DataFileWriter<User> dataFileWriter = new
DataFileWriter<User>(userDatumWriter);

dataFileWriter.create(user1.getSchema(), new File("users.avro"));

dataFileWriter.append(user1);

dataFileWriter.append(user2);

dataFileWriter.append(user3);

dataFileWriter.close();
```

We make a DatumWriter, which converts Java objects into an in-memory serialized format. The Specific DatumWriter class is use with createed classes and extracts the schema from the specifie created type.

Next we make a DataFileWriter, which writes and the serialized records, as well as the schema, to the file specified in the dataFileWriter.create call. Now

We write our users to the file via calls to the dataFileWriter.append method. When we will be done writing, we close the our data file.

Deserializing

Next, we tend to use the DataFileReader to iterate through the serialized users and print the deserialized object to stdout. Note however we tend to perform the iteration: we tend to produce one GenericRecord object that we tend to store this deserialized user in, and pass this record object to each call of dataFileReader.next. this can be a performance improvement that enables the DataFileReader to reprocess a similar record object instead of allocating a brand new GenericRecord for each iteration, which may be t very expensive in terms of object allocation and trash collection if we tend to deserialize an outsized record. While its technique is the currect way to iterate through a data file, it is also possible to use for (GenericRecord user : dataFileReader) if performance is not a concern.

Finally, we will ll deserialize the data file we just created.

```
// Deserialize users from disk

DatumReader<GenericRecord> datumReader = new
GenericDatumReader<GenericRecord>(schema);

DataFileReader<GenericRecord> dataFileReader = new
DataFileReader<GenericRecord>(file, datumReader);

GenericRecord user = null;

while (dataFileReader.hasNext()) {

// Reuse user object by passing it to next(). This saves us from
// allocating and garbage collecting many objects for files with

// many items.

user = dataFileReader.next(user);

System.out.println(user);
```

This outputs:

```
{"name": "Alyssa", "favorite_number": 256, "favorite_color": null}

{"name": "Ben", "favorite_number": 7, "favorite_color": "red"}
```

Deserializing is incredibly just like serializing. we tend to produce a GenericDatumReader, analogous to the GenericDatumWriter we tend to employed in serialisation, that converts in-memory serialized things into GenericRecords. we tend to pass the DatumReader and also the antecedently created File to a DataFileReader, analogous to the DataFileWriter, that reads the info file on disk.

Apache Avro Specification

Here Now we tend to defines Avro. It's intended to be the authoritative specification. Implementations of Apache avro must adhere to this document.

Schema Declaration

It's Schema epresented in JSON by one of:

- A JSON string, naming a defined type.
- A JSON object, of the form:

  {"type": "typeName" ...attributes...}

- where It's typeName either a primitive or derived type name, as defined below. Attributes not defined in this document are permitted as metadata, but must not affect the format of serialized data.

  A JSON array, represent one of a union of embedded types.

Primitive Types

The set of primitive type names is:

- null: no value
- boolean: a binary value
- int: 32-bit signed integer
- long: 64-bit signed integer
- float: single precision (32-bit) IEEE 754 floating-point number
- double: double precision (64-bit) IEEE 754 floating-point number
- bytes: sequence of 8-bit unsigned bytes
- string: unicode character sequence

Primitive types have't specified attributes.

Primitive type names are also defined type names. Thus, for example, the schema "string" is equivalent to:

{"type": "string"}

Complex Types

Apache Avro supports 6 kinds of complex types records, enums and arrays, maps and unions fixed.

• Records

Records use the type name "record" and support three attributes:

- name: a JSON string providing the name of the record (required).
- namespace, a JSON string that qualifies the name;

• Enums

For example, playing card suits might be defined with:

```
{ "type": "enum",

  "name": "Suit",

  "symbols" : ["SPADES", "HEARTS", "DIAMONDS", "CLUBS"]

}
```

• Array

Arrays use the type name "array" and support a single attribute:

- items: the schema of the array's items.

  For example, an array of strings is declared with:

  {"type": "array", "items": "string"}

- ## Maps

  For example, a map from string to long is declared with:

  {"type": "map", "values": "long"}

- ## Unions

- We have mentioned Unions, above, are represent using JSON arrays. For example, ["string", "null"] declares a schema which can be either a string or null.

- Unions can not contain more than one schema with the same type and except for the named types record, fixed and enum. For example, unions containing twic array types or twic map types are not permitted, but two types with different names are permitted. (Names permit efficient resolution when reading and writing unions.)

- ## Fixed

  For example, 16-byte quantity may be declared with:

  {"type": "fixed", "size": 16, "name": "md5"}

Data Serialization

Avro data is awlways serialized with its schema. Files that store Avro data should always embody the schema for that data within the same file. Avro-based remote procedure decision (RPC) systems should guarantee that remote recipients data have a copy of the schema used write that data.

Because the schema used write data is always accessible once the data is read, Avro data itself isn't labeled with kind data. The schema is needed to analyse data.In general, each serialization and deserialization proceed as a depth-first, left-to-right traversal of the schema, serializing primitive varieties as they're encountered.

Apache Avro Java API

Though theoretically any language could use Avro, the following languages have APIs written for them:

- Java
- Scala
- C#[2][3][4]
- C
- C++
- Python
- Ruby

Avro IDL

In addition to supporting JSON for kind and protocol definitions, Avro includes experimental support for associate alternate interface description language (IDL) syntax known as avro IDL. Previously known to as GenAvro, this format is intended to ease adoption by users familiar with a lot of ancient IDLs and programming languages, with a syntax kind of like C/C++, Protocol Buffers and an others.

Here we are going to define Avro IDL, a higher-level language for authoring Avro schemata. Before reading this document, you must have familiarity with the ideas of schemata and protocols, additionally because the varied primitive and sophisticated varieties accessible in Avro.

Why we used Avro IDL

The aim of the Avro IDL language is to start developers to author schemata in a way that feels a lot of kind of like common programming languages like Java, C++, or Python. to boot, the Avro IDL language might feel a lot of familiar for those users who have previously used the interface description languages (IDLs) in different frameworks like Thrift, Protocol Buffers, or CORBA.

Each Avro IDL file defines one Avro Protocol, and so generates as its output a JSON-format Avro Protocol file with extension .avpr

To convert a .avdl file into a .avpr file, it may be processed by the idl tool. For example:

```
$ java -jar avroj-tools.jar idl src/test/idl/input/namespaces.avdl /tmp/namespaces.avpr

$ head /tmp/namespaces.avpr

{

  "protocol" : "TestNamespace",

  "namespace" : "avro.test.protocol"
```

The idl tool can also process input to and from stdin and stdout. See idl --help for full usage information.

A Maven plugin is also provided to compile .avdl files. To use it, add something like the following to your pom.xml:

```
<build>

 <plugins>

  <plugin>

   <groupId>org.apache.avro</groupId>

   <artifactId>avro-maven-plugin</artifactId>

   <executions>

    <execution>

     <goals>

      <goal>idl-protocol</goal>

     </goals>

    </execution>

   </executions>

  </plugin>
```

```
                </plugins>

              </build>
```

How to Define a Protocol in Avro IDL

An Avro IDL file consists of correctly one protocol definition. The less protocol is defined by the following code:

```
protocol MyProtocol {

}
```

This is equivalent to (and generates) the following JSON protocol definition:

```
{

"protocol" : "MyProtocol",

"types" : [ ],

"messages" : {

}

}
```

The namespace of the protocol may be changed using the @namespace annotation:

```
@namespace("mynamespace")
```

```
protocol MyProtocol {

}
```

In this notation we used throughout Avro IDL as a way of specifying properties for the annotated element, as can be described later in this document.

Protocols in Avro IDL may contain the following items:

- Imports of external protocol and schema files.
- Definitions of named schemata, including records, errors, enums, and fixeds.
- Definitions of RPC messages

Defining an Enumeration

Enums are defined in Avro IDL using a syntax similar to C or Java:

```
enum Suit {

SPADES, DIAMONDS, CLUBS, HEARTS

}
```

Note that, unlike the JSON format, anonymous enums cannot be defined.

Conclusion

Affter learning In this content you can esily understand what is apache avro.we will be learning also about JSON and JSON file .