Avro 1.2.0 Specification

Table of contents

1. Introduction

This document defines Avro. It is intended to be the authoritative specification. Implementations of Avro must adhere to this document.

2. Schema Declaration

A Schema is represented in JSON by one of:

- A JSON string, naming a defined type.
- A JSON object, of the form:
  ```
  {"type": "typeName" ...attributes...}
  ```
  where *typeName* is 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, representing a union of embedded types.

2.1. Primitive Types

The set of primitive type names is:

- `string`: unicode character sequence
- `bytes`: sequence of 8-bit bytes
- `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
- `boolean`: a binary value
- `null`: no value

Primitive types have no specified attributes.

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

```
{"type": "string"}
```

2.2. Complex Types

Avro supports six kinds of complex types: records, enums, arrays, maps, unions and fixed.

2.2.1. Records

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

- `name`: a JSON string providing the name of the record (required).
- `fields`: a JSON array, listing fields (required). Each field is a JSON object with the following attributes:
  - `name`: a JSON string providing the name of the field (required), and
  - `type:` A JSON object defining a schema, or a JSON string naming a record definition (required).
  - `default:` A default value for this field, used when reading instances that lack this field (optional). Permitted values depend on the field's schema type, according to the table below. Default values for union fields correspond the first schema in the union.

avro type	json type	example
string	string	"foo"
bytes	string	"\u00FF"
int,long	integer	1
float,double	number	1.1
boolean	boolean	true
null	null	null
record	object	{"a": 1}
enum	string	"FOO"
array	array	[1]
map	object	{"a": 1}
fixed	string	"\u00ff"

Table 1: field default values

  - `order:` specifies how this field impacts sort ordering of this record (optional). Valid values are "ascending" (the default), "descending", or "ignore". For more details on how this is used, see the the sort order section below.

For example, a linked-list of 64-bit values may be defined with:

```
{
  "type": "record",
  "name": "LongList",
  "fields" : [
    {"name": "value", "type": "long"},            // each element has a
long
    {"name": "next", "type": ["LongList", "null"]} // optional next element
  ]
```

```
}
```

2.2.2. Enums

Enums use the type name "enum" and support the following attributes:

- `name`: a JSON string providing the name of the enum (required).
- `symbols`: a JSON array, listing symbols, as JSON strings (required).

For example, playing card suits might be defined with:

```
{ "type": "enum",
  "name": "Suit",
  "symbols" : ["SPADES", "HEARTS", "DIAMONDS", "CLUBS"]
}
```

2.2.3. Arrays

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"}
```

2.2.4. Maps

Maps use the type name `"map"` and support one attribute:

- `values`: the schema of the map's values.

Map keys are assumed to be strings.

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

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

2.2.5. Unions

Unions, as mentioned above, are represented using JSON arrays. For example, `["string", "null"]` declares a schema which may be either a string or null.

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

Unions may not immediately contain other unions.

2.2.6. Fixed

Fixed uses the type name `"fixed"` and supports two attributes:

- `name`: the name of the fixed (required).
- `size`: an integer, specifying the number of bytes per value (required).

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

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

2.3. Identifiers

Record, field and enum names must:

- start with `[A-Za-z_]`
- subsequently contain only `[A-Za-z0-9_]`

3. Data Serialization

Avro data is always serialized with its schema. Files that store Avro data should always also include the schema for that data in the same file. Avro-based remote procedure call (RPC) systems must also guarantee that remote recipients of data have a copy of the schema used to write that data.

Because the schema used to write data is always available when the data is read, Avro data itself is not tagged with type information. The schema is required to parse data.

In general, both serialization and deserialization proceed as a depth-first, left-to-right traversal of the schema, serializing primitive types as they are encountered.

3.1. Encodings

Avro specifies two serialization encodings: binary and JSON. Most applications will use the binary encoding, as it is smaller and faster. But, for debugging and web-based applications, the JSON encoding may sometimes be appropriate.

3.2. Binary Encoding

3.2.1. Primitive Types

Primitive types are encoded in binary as follows:

- a `string` is encoded as a `long` followed by that many bytes of UTF-8 encoded character data.

  For example, the three-character string "foo" would be encoded as the long value 3 (encoded as hex `06`) followed by the UTF-8 encoding of 'f', 'o', and 'o' (the hex bytes `66 6f 6f`):

  ```
  06 66 6f 6f
  ```
- `bytes` are encoded as a `long` followed by that many bytes of data.
- `int` and `long` values are written using variable-length zig-zag coding. Some examples:

value	hex
0	00
-1	01
1	02
-2	03
2	04
...	
-64	7f
64	80 01
...	

- a `float` is written as 4 bytes. The float is converted into a 32-bit integer using a method equivalent to Java's floatToIntBits and then encoded in little-endian format.
- a `double` is written as 8 bytes. The double is converted into a 64-bit integer using a method equivalent to Java's doubleToLongBits and then encoded in little-endian format.
- a `boolean` is written as a single byte whose value is either `0` (false) or `1` (true).
- `null` is written as zero bytes.

3.2.2. Complex Types

Complex types are encoded in binary as follows:

3.2.2.1. Records

A record is encoded by encoding the values of its fields in the order that they are declared. In other words, a record is encode as just the concatenation of its field's encodings. Field values are encoded per their schema.

For example, the record schema

```
{
"type": "record",
"name": "test",
"fields" : [
{"name": "a", "type": "long"},
{"name": "b", "type": "string"}
]
}
```

An instance of this record whose `a` field has value 27 (encoded as hex `36`) and whose `b` field has value "foo" (encoded as hex bytes `0C 66 6f 6f`), would be encoded simply as the concatenation of these, namely the hex byte sequence:

```
36 0C 66 6f 6f
```

3.2.2.2. Enums

An enum is encoded by a `int`, representing the zero-based position of the symbol in the schema.

For example, consider the enum:

```
{"type": "enum", "name": "Foo", "symbols": ["A", "B", "C",
"D"] }
```

This would be encoded by an `int` between zero and three, with zero indicating "A", and 3 indicating "D".

3.2.2.3. Arrays

Arrays are encoded as a series of *blocks*. Each block consists of a `long` *count* value, followed by that many array items. A block with count zero indicates the end of the array. Each item is encoded per the array's item schema.

If a block's count is negative, then the count is followed immediately by a `long` block *size*, indicating the number of bytes in the block. The actual count in this case is the absolute value of the count written.

For example, the array schema

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

an array containing the items 3 and 27 could be encoded as the long value 2 (encoded as hex 04) followed by long values 3 and 27 (encoded as hex `06 36`) terminated by zero:

```
04 06 36 00
```

The blocked representation permits one to read and write arrays larger than can be buffered in memory, since one can start writing items without knowing the full length of the array. The optional block sizes permit fast skipping through data, e.g., when projecting a record to a subset of its fields.

3.2.2.4. Maps

Maps are encoded as a series of *blocks*. Each block consists of a `long` *count* value, followed by that many key/value pairs. A block with count zero indicates the end of the map. Each item is encoded per the map's value schema.

If a block's count is negative, then the count is followed immediately by a `long` block *size*, indicating the number of bytes in the block. The actual count in this case is the absolute value of the count written.

The blocked representation permits one to read and write maps larger than can be buffered in memory, since one can start writing items without knowing the full length of the map. The optional block sizes permit fast skipping through data, e.g., when projecting a record to a subset of its fields.

NOTE: Blocking has not yet been fully implemented and may change. Arbitrarily large objects must be easily writable and readable but until we have proven this with an implementation and tests this part of the specification should be considered draft.

3.2.2.5. Unions

A union is encoded by first writing a `long` value indicating the zero-based position within the union of the schema of its value. The value is then encoded per the indicated schema within the union.

For example, the union schema `["string","null"]` would encode:

- `null` as the integer 1 (the index of "null" in the union, encoded as hex `02`):
  ```
  02
  ```
- the string `"a"` as zero (the index of "string" in the union), followed by the serialized string:
  ```
  00 02 61
  ```

3.2.2.6. Fixed

Fixed instances are encoded using the number of bytes declared in the schema.

3.3. JSON Encoding

Except for unions, the JSON encoding is the same as is used to encode field default values.

The value of a union is encoded in JSON as follows:

- if its type is `null`, then it is encoded as a JSON null;
- otherwise it is encoded as a JSON object with one name/value pair whose name is the type's name and whose value is the recursively encoded value. For Avro's named types (record, fixed or enum) the user-specified name is used, for other types the type name is used.

For example, the union schema `["null","string","Foo"]`, where Foo is a record name, would encode:

- `null` as `null`;
- the string `"a"` as `{"string": "a"}`; and
- a Foo instance as `{"Foo": {...}}`, where `{...}` indicates the JSON encoding of a Foo instance.

Note that a schema is still required to correctly process JSON-encoded data. For example, the JSON encoding does not distinguish between `int` and `long`, `float` and `double`, records and maps, enums and strings, etc.

4. Sort Order

Avro defines a standard sort order for data. This permits data written by one system to be efficiently sorted by another system. This can be an important optimization, as sort order comparisons are sometimes the most frequent per-object operation. Note also that Avro binary-encoded data can be efficiently ordered without deserializing it to objects.

Data items may only be compared if they have identical schemas. Pairwise comparisons are implemented recursively with a depth-first, left-to-right traversal of the schema. The first mismatch encountered determines the order of the items.

Two items with the same schema are compared according to the following rules.

- `int`, `long`, `float` and `double` data is ordered by ascending numeric value.
- `boolean` data is ordered with false before true.
- `null` data is always equal.
- `string` data is compared lexicographically. Note that since UTF-8 is used as the binary encoding of strings, sorting by bytes and characters is equivalent.
- `bytes` and `fixed` data are compared lexicographically by byte.
- `array` data is compared lexicographically by element.
- `enum` data is ordered by the symbol's position in the enum schema. For example, an enum whose symbols are `["z", "a"]` would sort `"z"` values before `"a"` values.

- `union` data is first ordered by the branch within the union, and, within that, by the type of the branch. For example, an `["int", "string"]` union would order all int values before all string values, with the ints and strings themselves ordered as defined above.
- `record` data is ordered lexicographically by field. If a field specifies that its order is:
  - `"ascending"`, then the order of its values is unaltered.
  - `"ascending"`, then the order of its values is reversed.
  - `"ignore"`, then its values are ignored when sorting.
- `map` data may not be compared. It is an error to attempt to compare data containing maps unless those maps are in an `"order":"ignore"` record field.

5. Object Container Files

Avro includes a simple object container file format. A file has a schema, and all objects stored in the file must be written according to that schema. Objects are stored in blocks that may be compressed. Syncronization markers are used between blocks to permit efficient splitting of files for MapReduce processing.

Files may include arbitrary user-specified metadata.

A file consists of:

- A *header, followed by*
- one or more *blocks*.

There are two kinds of blocks, *normal* and *metadata*. All files must contain at least one metadata block. A file terminates with its last metadata block. Any data after the last metadata block is ignored.

A header consists of:

- Four bytes, ASCII 'O', 'b', 'j', followed by zero.
- A 16-byte sync marker.

A metadata block consists of:

- The file's 16-byte sync marker.
- A long with value -1, identifying this as a metadata block.
- A long indicating the size in bytes of this block.
- A long indicating the number of metadata key/value pairs.
- For each pair, a string key and bytes value.
- The size in bytes of this block as a 4-byte big-endian integer.
  When a file is closed normally, this terminates the file and permits one to efficiently seek to the start of the metadata. If the sync marker there does not match that at the start of the file, then one must scan for the last metadata in the file.

The following metadata properties are reserved:

- **schema** contains the schema of objects stored in the file, as a string.
- **count** contains the number of objects in the file as a decimal ASCII string.
- **codec** the name of the compression codec used to compress blocks, as a string. The only value for codec currently supported is "null" (meaning no compression is performed). If codec is absent, it is assumed to be "null".
- **sync** the 16-byte sync marker used in this file, as a byte sequence.

A normal block consists of:

- The file's 16-byte sync marker.
- A long indicating the size in bytes of this block in the file.
- The serialized objects. If a codec is specified, this is compressed by that codec.

Note that this format supports appends, since multiple metadata blocks are permitted.

To be robust to application failure, implementations can write metadata periodically to limit the amount of the file that must be scanned to find the last metadata block.

6. Protocol Declaration

Avro protocols describe RPC interfaces. Like schemas, they are defined with JSON text.

A protocol is a JSON object with the following attributes:

- *name*, string, to distinguish it from other protocols;
- *namespace*, a string which qualifies the name;
- *types*, a list of record, enum and error definitions. An error definition is just like a record definition except it uses "error" instead of "record". Note that forward references to records, enums and errors are not currently supported.
- *messages*, a JSON object whose keys are message names and whose values are objects whose attributes are described below. No two messages may have the same name.

6.1. Messages

A message has attributes:

- a *request*, a list of named, typed *parameter* schemas (this has the same form as the fields of a record declaration);
- a *response* schema; and
- an optional union of *error* schemas.

A request parameter list is processed equivalently to an anonymous record. Since record field

lists may vary between reader and writer, request parameters may also differ between the caller and responder, and such differences are resolved in the same manner as record field differences.

6.2. Sample Protocol

For example, one may define a simple HelloWorld protocol with:

```
{
  "namespace": "com.acme",
  "protocol": "HelloWorld",

  "types": [
    {"name": "Greeting", "type": "record", "fields": [
      {"name": "message", "type": "string"}]},
    {"name": "Curse", "type": "error", "fields": [
      {"name": "message", "type": "string"}]}
  ],

  "messages": {
    "hello": {
      "request": [{"name": "greeting", "type": "Greeting" }],
      "response": "Greeting",
      "errors": ["Curse"]
    }
  }
}
```

7. Protocol Wire Format

7.1. Message Transport

Messages may be transmitted via different *transport* mechanisms. For example, one might use the HTTP, raw sockets, or SSL, etc. This document specifies formats for request and response message data, but it does not yet specify any details of how message data is encapsulated in different transports.

To the transport, a *message* is an opaque byte sequence.

A transport is a system that supports:

- **transmission of request messages**
- **receipt of corresponding response messages**

  Servers will send a response message back to the client corresponding to each request message. The mechanism of that correspondance is transport-specific. For example, in

HTTP it might be implicit, since HTTP directly supports requests and responses. But a transport that multiplexes many client threads over a single socket would need to tag messages with unique identifiers.

7.2. Message Framing

Avro messages are *framed* as a list of buffers.

Framing is a layer between messages and the transport. It exists to optimize certain operations.

The format of framed message data is:

- a series of *buffers*, where each buffer consists of:
  - a four-byte, big-endian *buffer length*, followed by
  - that many bytes of *buffer data*.
- A message is always terminated by a zero-lenghted buffer.

Framing is transparent to request and response message formats (described below). Any message may be presented as a single or multiple buffers.

Framing can permit readers to more efficiently get different buffers from different sources and for writers to more efficiently store different buffers to different destinations. In particular, it can reduce the number of times large binary objects are copied. For example, if an RPC parameter consists of a megabyte of file data, that data can be copied directly to a socket from a file descriptor, and, on the other end, it could be written directly to a file descriptor, never entering user space.

A simple, recommended, framing policy is for writers to create a new segment whenever a single binary object is written that is larger than a normal output buffer. Small objects are then appended in buffers, while larger objects are written as their own buffers. When a reader then tries to read a large object the runtime can hand it an entire buffer directly, without having to copy it.

7.3. Handshake

RPC requests and responses are prefixed by handshakes. The purpose of the handshake is to ensure that the client and the server have each other's protocol definition, so that the client can correctly deserialize responses, and the server can correctly deserialize requests. Both clients and servers should maintain a cache of recently seen protocols, so that, in most cases, a handshake will be completed without extra round-trip network exchanges or the transmission of full protocol text.

The handshake process uses the following record schemas:

```
{
  "type": "record",
  "name": "HandshakeRequest", "namespace":"org.apache.avro.ipc",
  "fields": [
    {"name": "clientHash",
     "type": {"type": "fixed", "name": "MD5", "size": 16}},
    {"name": "clientProtocol", "type": ["null", "string"]},
    {"name": "serverHash", "type": "MD5"},
    {"name": "meta", "type": ["null", {"type": "map", "values": "bytes"}]}
  ]
}
{
  "type": "record",
  "name": "HandshakeResponse", "namespace": "org.apache.avro.ipc",
  "fields": [
    {"name": "match",
     "type": {"type": "enum", "name": "HandshakeMatch",
              "symbols": ["BOTH", "CLIENT", "NONE"]}},
    {"name": "serverProtocol",
     "type": ["null", "string"]},
    {"name": "serverHash",
     "type": ["null", {"type": "fixed", "name": "MD5", "size": 16}]},
    {"name": "meta",
     "type": ["null", {"type": "map", "values": "bytes"}]}
  ]
}
```

- A client first prefixes each request with a `HandshakeRequest` containing just the hash of its protocol and of the server's protocol (`clientHash!=null, clientProtocol=null, serverHash!=null`), where the hashes are 128-bit MD5 hashes of the JSON protocol text. If a client has never connected to a given server, it sends its hash as a guess of the server's hash, otherwise it sends the hash that it previously obtained from this server.
- The server responds with a `HandshakeResponse` containing one of:
  - `match=BOTH, serverProtocol=null, serverHash=null` if the client sent the valid hash of the server's protocol and the server knows what protocol corresponds to the client's hash. In this case, the request is complete and the response data immediately follows the HandshakeResponse.
  - `match=CLIENT, serverProtocol!=null, serverHash!=null` if the server has previously seen the client's protocol, but the client sent an incorrect hash of the server's protocol. The request is complete and the response data immediately follows the HandshakeResponse. The client must use the returned protocol to process the response and should also cache that protocol and its hash for future interactions with this server.
  - `match=NONE, serverProtocol!=null, serverHash!=null` if the

server has not previously seen the client's protocol and the client sent an incorrect hash of the server's protocol.

In this case The client must then re-submit its request with its protocol text (`clientHash!=null, clientProtocol!=null, serverHash!=null`) and the server should respond with with a successful match (`match=BOTH, serverProtocol=null, serverHash=null`) as above.

The `meta` field is reserved for future handshake enhancements.

7.4. Call Format

A *call* consists of a request message paired with its resulting response or error message. Requests and responses contain extensible metadata, and both kinds of messages are framed as described above.

The format of a call request is:

- *request metadata*, a map with values of type `bytes`
- the *message name*, an Avro string, followed by
- the message *parameters*. Parameters are serialized according to the message's request declaration.

The format of a call response is:

- *response metadata*, a map with values of type `bytes`
- a one-byte *error flag* boolean, followed by either:
  - if the error flag is false, the message *response*, serialized per the message's response schema.
  - if the error flag is true, the *error*, serialized per the message's error union schema.

8. Schema Resolution

A reader of Avro data, whether from an RPC or a file, can always parse that data because its schema is provided. But that schema may not be exactly the schema that was expected. For example, if the data was written with a different version of the software than it is read, then records may have had fields added or removed. This section specifies how such schema differences should be resolved.

We call the schema used to write the data as the *writer's* schema, and the schema that the application expects the *reader's* schema. Differences between these should be resolved as follows:

- It is an error if the two schemas do not *match*.

  To match, one of the following must hold:

  - both schemas are arrays whose item types match
  - both schemas are maps whose value types match
  - both schemas are enums whose names match
  - both schemas are fixed whose sizes and names match
  - both schemas are records with the same name
  - either schema is a union
  - both schemas have same primitive type
  - the writer's schema may be *promoted* to the reader's as follows:
    - int is promotable to long, float, or double
    - long is promotable to float or double
    - float is promotable to double

- **if both are records:**
  - the ordering of fields may be different: fields are matched by name.
  - schemas for fields with the same name in both records are resolved recursively.
  - if the writer's record contains a field with a name not present in the reader's record, the writer's value for that field is ignored.
  - if the reader's record schema has a field that contains a default value, and writer's schema does not have a field with the same name, then the reader should use the default value from its field.
  - if the reader's record schema has a field with no default value, and writer's schema does not have a field with the same name, then the field's value is unset.
- **if both are enums:**

  if the writer's symbol is not present in the reader's enum, then the enum's value is unset.
- **if both are arrays:**

  This resolution algorithm is applied recursively to the reader's and writer's array item schemas.
- **if both are maps:**

  This resolution algorithm is applied recursively to the reader's and writer's value schemas.
- **if both are unions:**

  The first schema in the reader's union that matches the selected writer's union schema is recursively resolved against it. if none match, an error is signalled.
- **if reader's is a union, but writer's is not**

  The first schema in the reader's union that matches the writer's schema is recursively

resolved against it. If none match, an error is signalled.

- **if writer's is a union, but reader's is not**
  If the reader's schema matches the selected writer's schema, it is recursively resolved against it. If they do not match, an error is signalled.