Apache Avro

1) Avro is a binary encoding format. Encodes data in binary which makes it fast when transferred in networks.
2) It uses a schema to specify the structure of data that is being encoded, so decoding it is easier.
3) Avro has two schema languages (Avro IDL) and one based on JSON.
4) Avro IDL -> Avro Interface Definition Language.

Example data:

```
{
 "username": "Martin",
 "favouriteNumber": 1337,
 "interests": ["dayDreaming", "hacking"]
}
```

Example Schema in Avro IDL

```
record Person{
        string  username;
        union {null, long} favouriteNumber;
        array<string>  interests;
}
```

Equivalent JSON representation

```
{
 "type": "record",
 "name": "Person",
 "fields": [
  {"name":  "username", "type": "string"},
  {"name":  "favouriteNumber",
    "type": ["null", "long"],
    "default": null
  },
  {"name":  "interests", "type": {"type":  "array", "items":  "string"}}
 ]}
```

5) In the encoding , there is nothing to identify the fields or their datatypes.
6) The encoding simply consists of values concatenated together.
7) A String is just a length prefix followed by UTF-8 bytes.
8) An integer is encoded using a variable length encoding.
9) To parse the binary data, we have to go through the fields in the order in which they appear in the schema and use the schema to tell what is the data type.
10) This means that the binary data can only be decoded correctly only if the code reading the data uses exactly the same schema as the code that wrote the data.
11) Writers Schema -> The schema that is used to write or encode the data and write it to file, database or send it over the network.
12) Readers Schema -> Schema that is used to decode the data or read the data from the file,database
13) The key idea with Avro is that the readers and the writers schema don't have to be the same- they only have to be compatible.
14) Avro library resolves the differences by looking at the writer's schema and reader's schema side by side and translating the data from the writer's schema into the reader's schema.
15) Fields in the reader and writer schema can be in different orders because the schema resolution matches the fields by the field names.
16) If the code reading the data encounters a field that is in the writer's schema but not in the reader's schema, it is ignored.
17) If the code reading the data expects a field but that field is not present in the writer's schema, then the value of that field is filled with the default value present in the reader's schema.

Schema Evolution Rules

Forward Compatibility:
Older code can read data that was written by newer code.

New Version of schema as a writer.
Old Version of schema as a reader.

Backward Compatibility:

Newer code can read data that was written by older code.

New Version of schema as a reader.
Old Version of schema as a writer.

To Maintain Compatibility:

1) Only add or remove fields that have default values.

   If you add a field without a default value, the new readers won't be able to read data written by old writers thus breaking backward compatibility.

   If you remove a field without a default value, old readers won't be able to read data written by new writers.

2) We can change data types as long as avro can convert it.

Nulls

By default, null will not be acceptable. If you want a field to be null, have to use union type for example union {"string", "null"}.

Specifying Schema.

1) In Large files with million of records, the writer will include the writers schema
2) In a database, different records may be written at different point of time using different writer's schema.

The solution to this is to include a version number at the beginning of every encoded record, and to keep a list of schema versions in your database
A reader can fetch the record, extract the version number then fetch the writer schema using that version number and decode the data.

3) While sending records over the network, they can negotiate over the schema version and use that schema for the connection lifetime.

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, using binary encoding.

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 *file header*, followed by
- one or more *file data blocks*.

A file header consists of:

- Four bytes, ASCII 'O', 'b', 'j', followed by 1.
- *file metadata*, including the schema.
- The 16-byte, randomly-generated sync marker for this file.

File metadata is written as if defined by the following map schema:

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

All metadata properties that start with "avro." are reserved. The following file metadata properties are currently used:

- **avro.schema** contains the schema of objects stored in the file, as JSON data (required).
- **avro.codec** the name of the compression codec used to compress blocks, as a string. Implementations are required to support the following codecs: "null" and "deflate". If codec is absent, it is assumed to be "null". The codecs are described with more detail below.

A file header is thus described by the following schema:

```
{"type": "record", "name": "org.apache.avro.file.Header",
 "fields" : [
   {"name": "magic", "type": {"type": "fixed", "name": "Magic", "size": 4}},
   {"name": "meta", "type": {"type": "map", "values": "bytes"}},
   {"name": "sync", "type": {"type": "fixed", "name": "Sync", "size": 16}},
  ]
}
```

A file data block consists of:

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

Thus, each block's binary data can be efficiently extracted or skipped without deserializing the contents. The combination of block size, object counts, and sync markers enable detection of corrupt blocks and help ensure data integrity.

Required Codecs

null

The "null" codec simply passes through data uncompressed.

deflate

The "deflate" codec writes the data block using the deflate algorithm as specified in RFC 1951, and typically implemented using the zlib library. Note that this format (unlike the "zlib format" in RFC 1950) does not have a checksum.

Optional Codecs

bzip2

The "bzip2" codec uses the bzip2 compression library.

snappy

The "snappy" codec uses Google's Snappy compression library. Each compressed block is followed by the 4-byte, big-endian CRC32 checksum of the uncompressed data in the block.

xz

The "xz" codec uses the XZ compression library.

zstandard

The "zstandard" codec uses Facebook's Zstandard compression library.

Python with Avro
https://avro.apache.org/docs/1.10.2/gettingstartedpython.html
https://fastavro.readthedocs.io/en/latest/

Spark With Avro
https://spark.apache.org/docs/latest/sql-data-sources-avro.html