THE

APACHE™

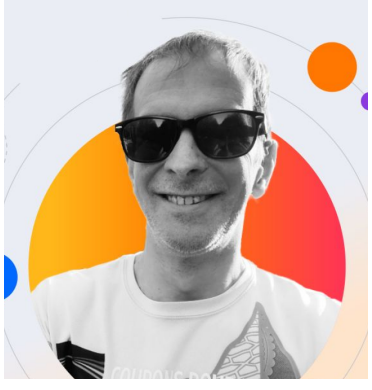
SOFTWARE FOUNDATION

Dive into Avro

Everything a Data Engineer needs to know

Who are we?

Ryan Skraba



aiven

RyanSkraba

RyanSkraba

rskraba@apache.org

Ismaël Mejía

Microsoft

iemejia

iemejia

iemejia@apache.org

iemejia

What is Avro?

- A File Format? A data format?
- A data model?
- A code generator?
- A serializer for my existing code?
- "Apache Avro™ is the leading serialization format for record data, and first choice for streaming data pipelines."

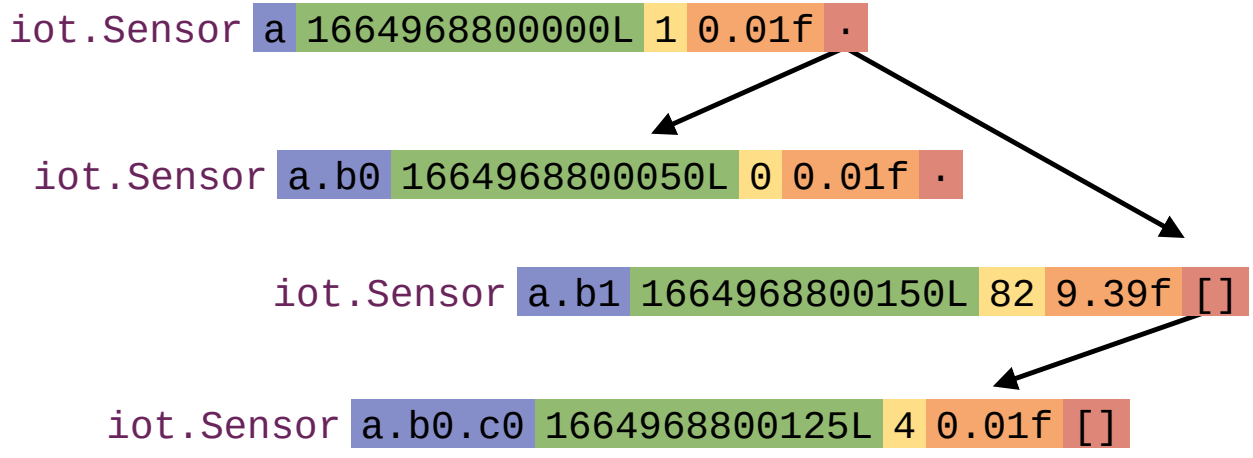
structure

```
iot.Sensor
  id: STRING
  start_ms: LONG
  defects: INT
  deviation: FLOAT
  subsensors: LIST<iot.Sensor>
```

JSON

```
{
  "id": "a",
  "start_ms": 1664968800000L,
  "defects": 1,
  "deviation": 0.01f,
  "subsensors": [
    ...
  ]
}
```

in-memory



Serializing data with Avro

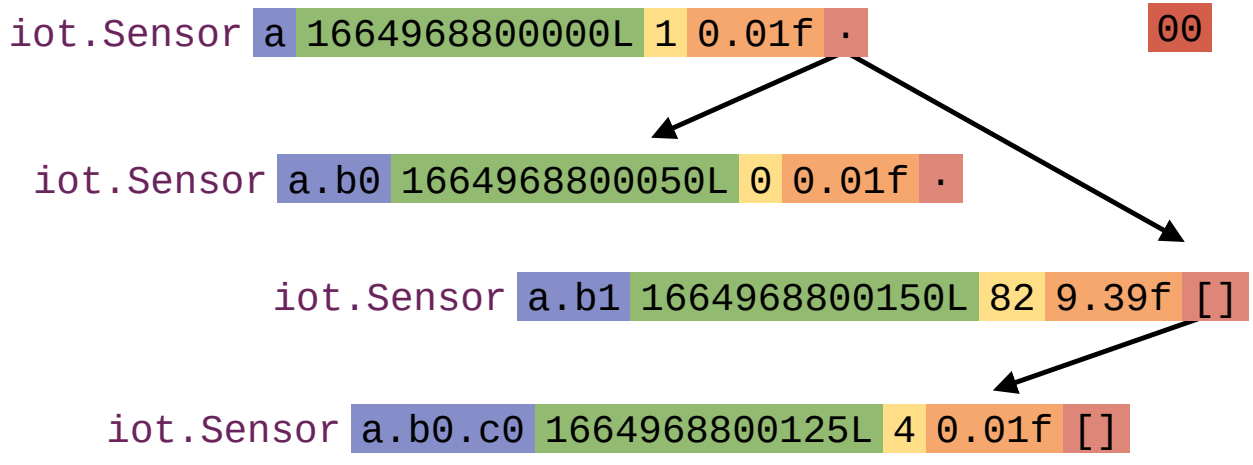
schema

```
iot.Sensor
  id: STRING
  start_ms: LONG
  defects: INT
  deviation: FLOAT
  subsensors: LIST<iot.Sensor>
```

binary

```
02 61 80 DC FD FD F4 60 02 0A
D7 23 3C 04 08 61 2E 62 30 E4
DC FD FD F4 60 00 0A D7 23 3C
00 08 61 2E 62 31 AC DE FD FD
F4 60 A4 01 89 41 16 41 02 0E
61 2E 62 30 2E 63 30 FA DD FD
FD F4 60 08 0A D7 23 3C 00 00
00
```

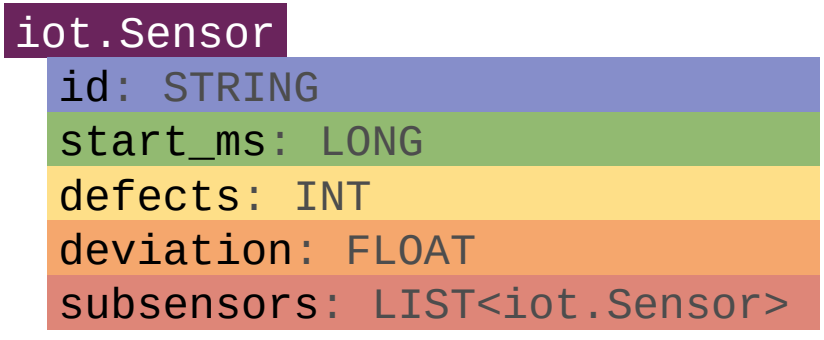
datum



PART I

The Binary Serialization Story

The Schema




```
{
  "type" : "record",
  "name" : "Sensor",
  "namespace" : "iot",
  "fields" : [
    {"name" : "id", "type" : "string"},
    {"name" : "start_ms", "type" : "long"},
    {"name" : "defects", "type" : "int"},
    {"name" : "deviation", "type" : "float"},
    {"name" : "subsensors",
     "type" : {"type" : "array",
               "items" : "Sensor"}}
  ]
}
```

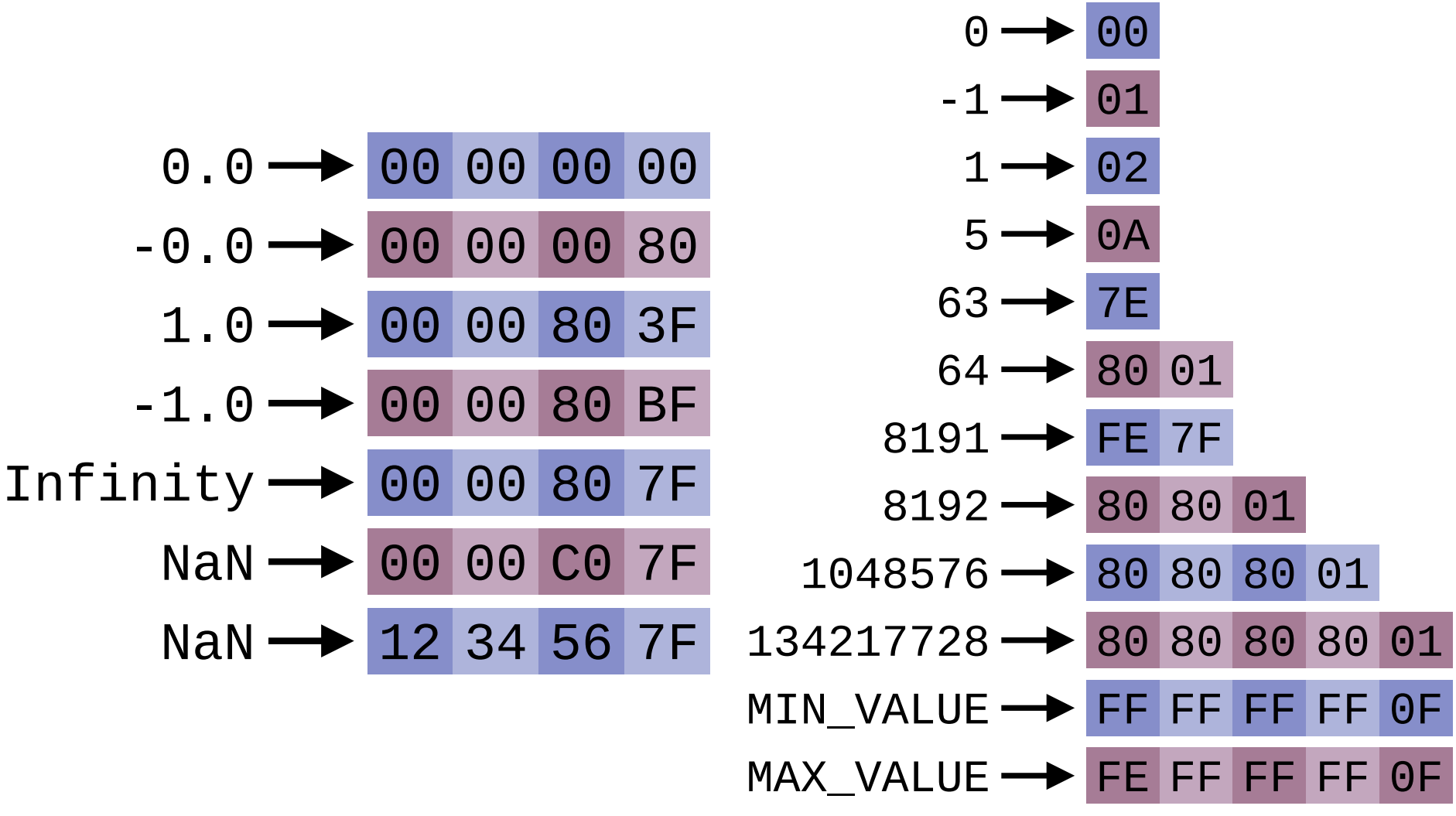
Serializing the primitives

```
iot.Sensor
  id: STRING
  start_ms: LONG
  defects: INT
  deviation: FLOAT
  subsensors: LIST<iot.Sensor>
```

```
02 61 80 DC FD FD F4 60 02 0A
D7 23 3C 04 08 61 2E 62 30 E4
DC FD FD F4 60 00 0A D7 23 3C
00 08 61 2E 62 31 AC DE FD FD
F4 60 A4 01 89 41 16 41 02 0E
61 2E 62 30 2E 63 30 FA DD FD
FD F4 60 08 0A D7 23 3C 00 00
00
```

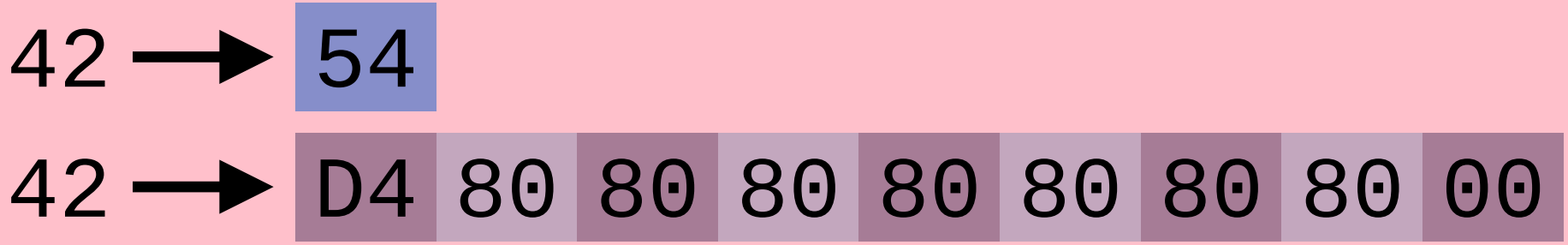
```
"a" → 02 61
1664968800000L → 80 DC FD FD F4 60
1 → 02
0.01 → 0A D7 23 3C
[b0, b1] → 04 .. .. .. .. 00

"a.b0" → 08 61 2E 62 30
0L → 00
82 → A4 01
9.93f → 89 41 16 41
[] → 00
```

Floating point versus integers



Gotcha: Determinism

- **Q**: Does `serialize(42)` always equal `serialize(42)`?
- **A**: Usually, but…



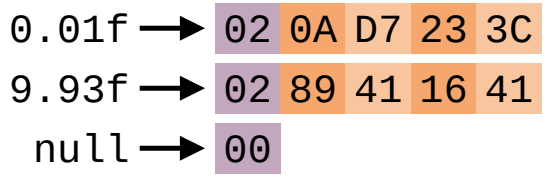
⚠️ Attention when using serialized bytes as keys in big data!

Gotcha: Precision

- An 8-byte IEEE-754 floating point number has *nearly* 16 decimal digits of precision
- An 8-byte LONG integer has *nearly* 19
- Usually not a problem, except… **JSON**!

UNIONs and NULLs

```
{
  "type" : "record",
  "name" : "Sensor",
  "namespace" : "iot",
  "fields" : [
    {"name" : "id", "type" : "string"},
    {"name" : "start_ms", "type" : "long"},
    {"name" : "defects", "type" : "int"},
    {"name" : "deviation",
     "type" : ["null", "float"]},
    {"name" : "subsensors",
     "type" : {"type" : "array",
               "items" : "Sensor"}}
  ]
}
```

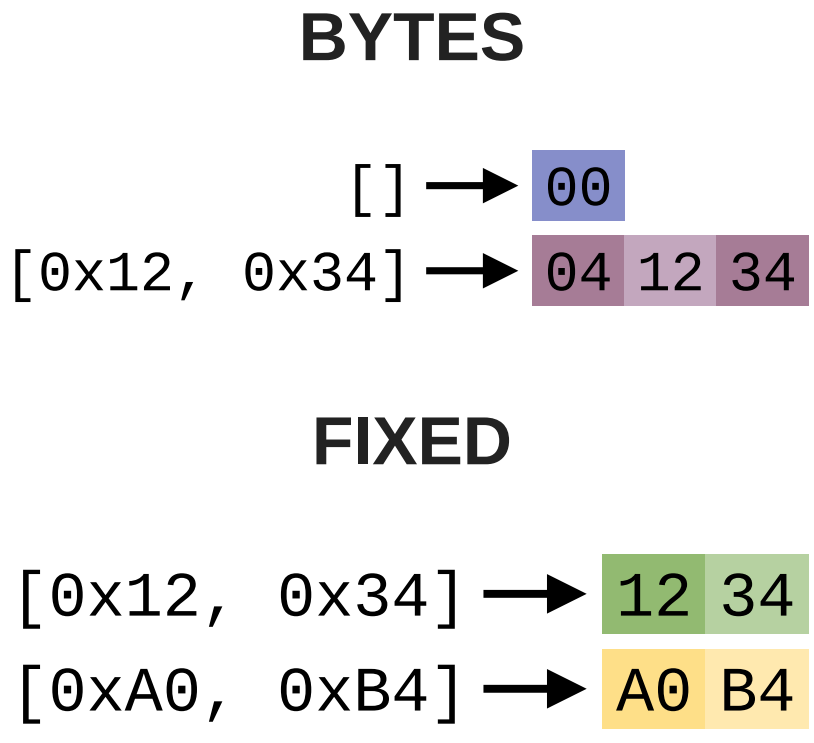


Gotcha: Meditation on the nature of nothing

- **Q**: How many zero byte datum (NULL) can you read from a zero byte buffer?
- **A**: … a bit more than infinity.

All the types

```
# Eight primitives
"null"
"boolean"
"int", "long"
"float", "double"
"bytes"
"string"

# Two collections
{"type", "array", "item": "iot.Sensor"}
{"type", "map", "item": "iot.Sensor"}

# Three named
{"type", "fixed", "name": "F2",
  "size": 2}
{"type", "enum", "name": "E2",
  "symbols": ["Z", "Y", "X", "W"]}
{"type", "record", "name": "iot.Sensor" ...}
# Plus union
```



Gotcha: Nondeterministic maps

```
[4, 5, 6] → 06 08 0A 0C 00
[4, 5, 6] → 02 08 02 0A 02 0C 00
[4, 5, 6] → 01 02 08 01 02 0A 01 02 0C 00
```

```
{"Hello":4,"Bye":5}
   → 04 0A 48 65 6C 6C 6F 08 06 42 79 65 0A 00
   → 04 06 42 79 65 0A 0A 48 65 6C 6C 6F 08 00
   → 02 0A 48 65 6C 6C 6F 08 02 06 42 79 65 0A 00
   → 01 10 0A 48 65 6C 6C 6F 08 01 1A 06 42 79 65 0A 00
```

⚠️ Don’t use MAPs in partition keys

Gotcha: Inherited namespaces

```
# Unqualified name
# Null or default namespace
{"type":"fixed","size":2,
 "name":"FX"}

# Qualified with a namespace
{"type":"fixed","size":3,
 "name":"size3.FX"}
{"type":"fixed","size":3
 "namespace":"size3",
 "name":"FX"}

{"type":"fixed","size":4,
 "name":"size4.FX"}
```

Namespaces are inherited inside a RECORD

```
{"type", "record",
  "name": "Record",
  "namespace": "size4", ⚠️(inherited)
  "fields": [
    {"name": "field0", "type": "size3.FX"}
    {"name": "field1", "type": "FX"} 🔴
  ]
}
```

⚠️ Best practice: use a namespace!

Gotcha: UTF-8 names

```
# Sometimes works
{"type":"fixed","size":12,
 "name":"utf.Durée"}

# Better
{"type":"fixed","size":12,
 "name":"utf.Dur",
 "i18n.fr_FR": "Durée"**}
```

Avro names are never found in serialized data!

`[_a-zA-Z][_a-zA-Z0-9]`

⚠️ Best practice: Follow the spec here.

Gotcha: Defaults are unused

```
{
  "type" : "record"
  "name" : "Sensor",
  "namespace" : "iot",
  "fields" : [
    {"name" : "id", "type" : "string"},
    {"name" : "start_ms", "type" : "long"},
    {"name" : "defects", "type" : "int"},
    {"name" : "deviation", "type" : "float"},
    {"name" : "temp",
     "type" : ["null", "double"],
     "aliases": ["tmep", "temperature"],
     "default": null},
    {"name" : "subsensors",
     "type" : {"type" : "array",
               "items" : "Sensor"}}
  ]
}
```

Some attributes don’t take part in simple serialization.

Except when they do…

⚠️ Best practice: Check your SDK

Gotcha: (Java-only) CharSequence or String, ByteBuffer or byte[]?

- By default, Avro reads `STRING` into `Utf8` instances

```
{"type": "string", "avro.java.string": "String"}
```

- Avro reads `BYTES` into `ByteBuffer` instances.
  - Be kind, `rewind()`

Gotcha: Embedding a schema in a schema

Succinct

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

Verbose

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

Nope

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

Gotcha: {"type": {"type": ... }}

```
{
  "type" : "record",
  "name" : "ns1.Simple",
  "fields" : [ {
    "name" : "id",
    "type" : {"type" : "long"},
  }, {
    "name" : "name",
    "type" : "string"
  } ]
}
```

- Conceptually `fieldType` and `fieldName`
- Important when adding arbitrary properties!

Parsing Canonical Form

- Primitives use the simple form.
- Remove all unnecessary attributes and all user properties.
- Use full `names`, remove all `namespace`
- List attributes in a specific order.
- Normalize JSON strings, replace any `\uXXXX` by UTF-8.
- Remove quotes and zero padding in numbers.
- Remove unnecessary whitespace.

Parsing Canonical Form

```
{
  "type" : "record"
  "name" : "Sensor",
  "namespace" : "iot",
  "fields" : [
    {"name" : "id", "type" : "string"},
    {"name" : "start_ms", "type" : "long"},
    {"name" : "defects", "type" : "int"},
    {"name" : "deviation", "type" : "float"},
    {"name" : "temp",
     "type" : ["null", "double"],
     "aliases": ["tmep", "temperature"],
     "default": null},
    {"name" : "subsensors",
     "type" : {"type" : "array",
               "items" : "Sensor"}}
  ]
}
```

```
**{"name":"iot.Sensor","type":"record",
{"name":"id","type":"string"},{"name":"
","type":"long"},{"name":"defects","typ
},{"name":"deviation","type":"float"},{
temp","type":["null","double"]},{"name"
sors","type":{"type":"array","items":"i
r"}}]}**
```

- **64bit Fingerprint**: 5614D05749C8743

What have we learned?

- Deep dive into ***HOW***, which answers a couple of ***WHY***:
  - Why isn't there a `SHORT` type? A `UINT`?
- The basic Avro data model and how to write a schema
- Tagless, streamlined binary encoding

PART II

Using Avro

Schema evolution: A new field

API V1 (aka **Actual**) (aka **Writer**)

```
iot.Sensor
  id: STRING
  start_ms: LONG
  defects: INT
  deviation: FLOAT
  subsensors: LIST<iot.Sensor>
```

```
02 61 80 DC FD FD F4 60 02 0A
D7 23 3C 04 08 61 2E 62 30 E4
DC FD FD F4 60 00 0A D7 23 3C
00 08 61 2E 62 31 AC DE FD FD
F4 60 A4 01 89 41 16 41 02 0E
61 2E 62 30 2E 63 30 FA DD FD
FD F4 60 08 0A D7 23 3C 00 00
00
```

API V2 (aka **Expected**) (aka **Reader**)

```
iot.Sensor
  id: STRING
  start_ms: LONG
  defects: INT
  temperature: [NULL,DOUBLE] = NULL
  deviation: FLOAT
  subsensors: LIST<iot.Sensor>
```

Every Sensor V2 `temperature` field will be read with NULL

Schema registries

- Store and order the schemas for a subject, and enforces compatibility guarantees when introducing a new schema.
- **`BACKWARDS`**: V4 ➜ V5 guaranteed; **consumers** update first
- **`FORWARDS`**: V5 ➜ V4 guaranteed; **producers** update first
- **`FULL`**: V4 ⬌ V5 guaranteed
- Suffix **`_TRANSITIVE`** to apply to ALL older schemas

Record evolution: By name

API V1 (aka **Actual**) (aka **Writer**)

```
iot.Sensor
  id: STRING
  start_ms: LONG
  defects: INT
  deviation: FLOAT
  subsensors: LIST<iot.Sensor>
```

```
02 61 80 DC FD FD F4 60 02 0A
D7 23 3C 04 08 61 2E 62 30 E4
DC FD FD F4 60 00 0A D7 23 3C
00 08 61 2E 62 31 AC DE FD FD
F4 60 A4 01 89 41 16 41 02 0E
61 2E 62 30 2E 63 30 FA DD FD
FD F4 60 08 0A D7 23 3C 00 00
00
```

API V3 Drop a field

```
iot.Sensor
  id: STRING
  start_ms: LONG
  defects: INT
  subsensors: LIST<iot.Sensor>
```

API V4 Reorder fields

```
iot.Sensor
  id: STRING
  subsensors: LIST<iot.Sensor>
  start_ms: LONG
  deviation: FLOAT
  defects: INT
```

Ungotcha: Renaming

```
iot.SensorV2
    name: STRING
    time: LONG
    errors: INT
    pressure: FLOAT
    sublist: LIST<iot.Sensor>
```

⚠️**This one trick drives schema registries crazy!**

- Avro binary data never has names in it.
- Don’t use a schema pair 😱
- Stick the new name in the same position in the "writer" schema.
- Let us never speak of this again.

Schema evolution: Primitives

1. Primitives can be widened / promoted:
    - `INT` ➜ to `LONG`, `FLOAT`, or `DOUBLE`
    - `LONG` ➜ to `FLOAT`, or `DOUBLE`
    - `FLOAT` ➜ to `DOUBLE`
    - `STRING` ⬌ `BYTES` are interchangeable

⚠️ `LONG` to `DOUBLE` loses precision but you're asking for it

Schema evolution: other types

- if both **`array`** ➜ the `item` type matches
- if both **`map`** ➜ the `value` type matches
- if both **`fixed`** ➜ the `size` is the same
- if named ➜ the **unqualified name** is the same
- if both **`enum`** ➜ symbols are resolved by name
- if unknown **`enum`** symobl, use `default`
- otherwise an error at **deserialization time**

Schema evolution: Unions

1. Making a field **optional**:

   `LONG` ➜ to `UNION<NULL,LONG>`

2. Making a field **required**: ⚠️

   `UNION<NULL,LONG>` ➜ to `LONG`

3. **Adding** a message payload:

   `UNION<A,B,C,D,E>` ➜ to `UNION<A,B,C,D,E,F>`

4. **Removing** a message payload: ⚠️

   `UNION<A,B,C,D,E>` ➜ to `UNION<A,B,D,E>`

Avro message format

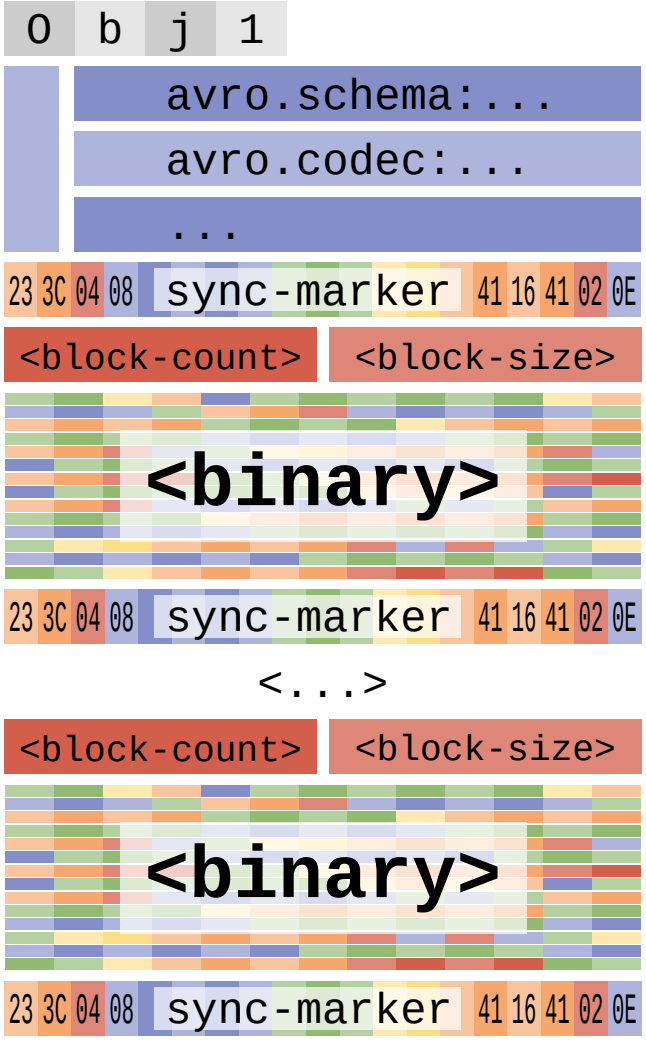
```
iot.Sensor
  id: STRING
  start_ms: LONG
  defects: INT
  deviation: FLOAT
  subsensors: LIST<iot.Sensor>
```

```
C3 01 05 61 4D 05 74 9C 87 43
02 61 80 DC FD FD F4 60 02 0A
D7 23 3C 04 08 61 2E 62 30 E4
DC FD FD F4 60 00 0A D7 23 3C
00 08 61 2E 62 31 AC DE FD FD
F4 60 A4 01 89 41 16 41 02 0E
61 2E 62 30 2E 63 30 FA DD FD
FD F4 60 08 0A D7 23 3C 00 00
00
```

- `0xC301` + 8 byte fingerprint
- Doesn't specify how to store, fetch, resolve, decode…

⚠️ PCF drops evolution attributes!

Avro file format



- `Obj1` + metadata
- 16 byte sync marker
- Splittable
- Compressable
- Appendable

Built-in Logical types

- `date`: on `INT` (days since 1970)
- `time-millis`: on `INT` (ms since midnight)
- `time-micros`: on `LONG` (µs since midnight)
- `timestamp-millis`: on `LONG` (ms since 0h00 1970 UTC)
- `timestamp-micros`: on `LONG` (µs since 0h00 1970 UTC)
- `duration`: on `FIXED(12)` for three little-endian UINT32: months, days, ms
- `decimal`: on `FIXED` and `BYTES`, `scale`, `precision`
- `uuid`: on `STRING`

Future Avro Training subjects

- Generic / Specific / Reflect
- Code generation
- Avro IDL

Part III

Avro in the Big Data Ecosystem

What Avro solved

- Byte Representation **Consistency**
- Multi-language **Interoperability**
- **Distributed-Data Friendly**: Splittable, Compressable, Appendable

Uses of Avro

1. Data (Record) Exchange Format (Serialization)
2. File Format
3. IDL / RPC

1. Avro for Data Exchange

Most popular serialization format for Streaming Systems

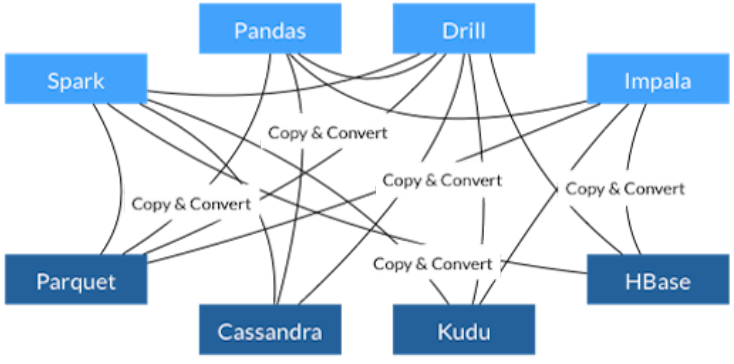
Why Avro excels for this use case?

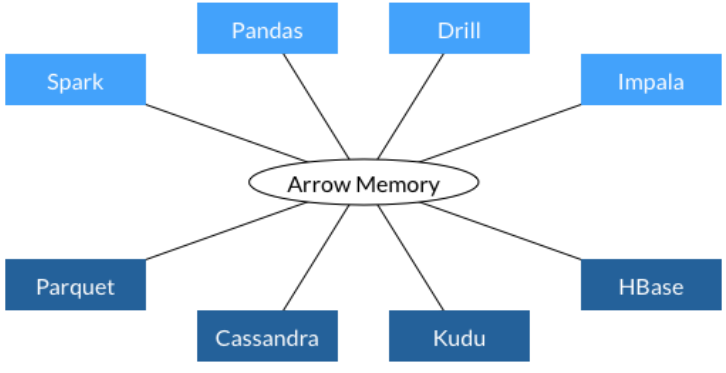
Efficient format and **Schema Evolution**

Alternatives to Avro for Data Exchange

- **Language-based serialization** (e.g. Java, Pickle): Not consistent, Language specific
- **JSON**: Schemaless
- **CSV**: Verbose, Inconsistent
- **XML**: Too verbose
- **Protocol Buffers**: IDL/RPC oriented (gRPC)
- **Apache Thrift**: Protocol definition format
- **Flatbuffers**: No need to decode in memory

Disgression 1: What about in-memory exchange formats?





- **Apache Arrow:** Memory layout format designed for language-agnostic interoperability.
- **Arrow2** (not Apache) can represent Avro records as Arrow records.

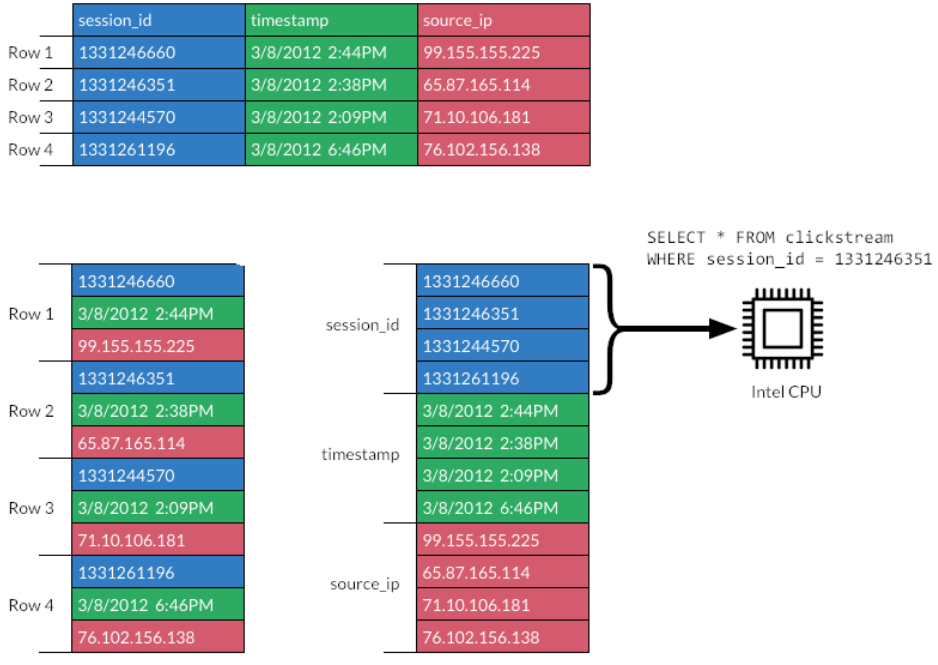
2. Avro as a File Format

Supported by all **Big Data Frameworks** and **(Cloud) Data-Warehouse**

Alternatives to Avro File Format

- **JSON, XML**: Non-splittable
- **CSV**: Not a standard
- **Protobuf**: Low ecosystem support for the file use case.
- **Parquet**: Column-based, efficient for Analytics.

Disgression 2: Columnar formats: Parquet



Columnar formats are better suited for analytics: column pruning, vectorization, stats).

Disgression 3: What about Table Formats?

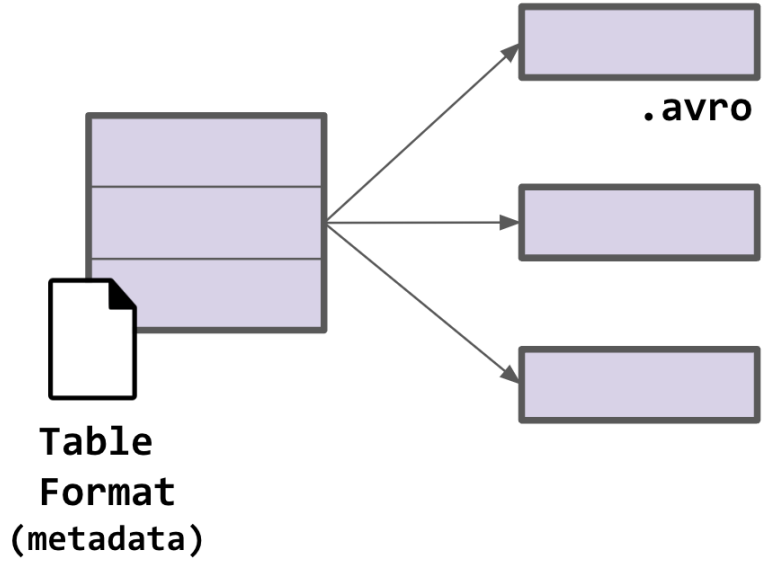


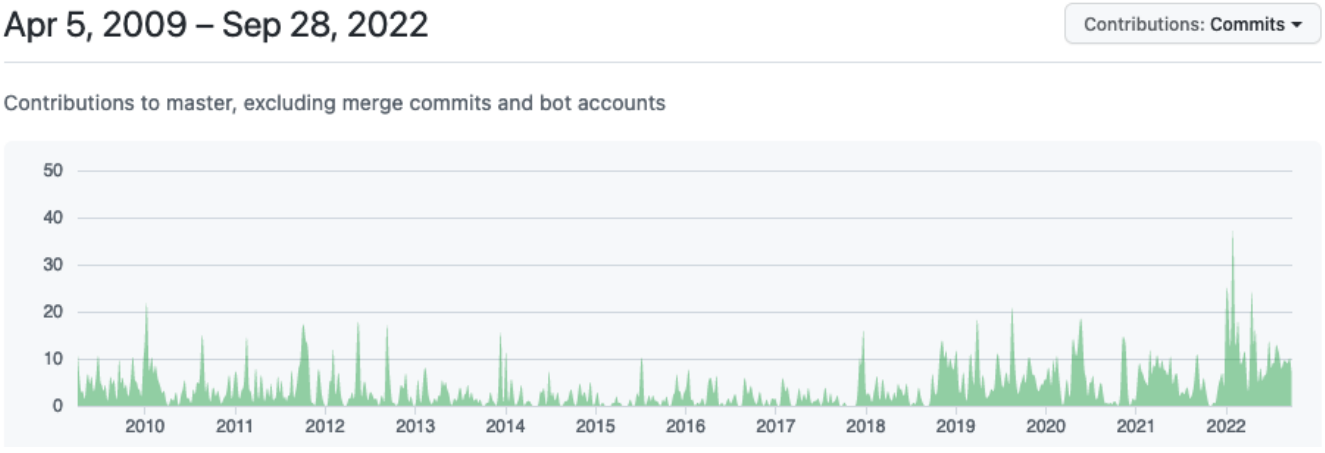
Table formats are metadata to represent all the files that compose a dataset as a “table”.

Avro can be one of those formats (i.e. Iceberg).

Advantages of Avro

- **Ecosystem Support**
- **Active community**
- **Stable Specification**, not broken since 1.3 (12 years!). Logical Types in 1.8 are backwards/forwards compatible.
- **Language Support**
  Apache: C, C++, C#, Java, Javascript, Perl, PHP, Python, Ruby, Rust
  Non-Apache: Python (FastAvro), Go, Haskell, Erlang, etc.

State of the Project



- Improved Release Cadence
- Improved Contributor Experience: Github Actions, Docker, Codespaces
- New Website
- Rust Implementation (donated by Yelp)

The Future of Avro

Format

- Is it worth to break the spec at this point?
- Is there room for improvement given the stability constraints?

Implementations

- Semantic Versioning. How to do it with so many languages?
- Automation specially for the release.
- Performance and Interoperability Tests
- Improved documentation

Conclusion / Q&A

No format is perfect for everything, there are different use cases where different formats excel.

Use Avro where it fits.

Join us?

https://avro.apache.org