Sharing is caring

Efficient Data Exchange with PyArrow

By: Alenka Frim, Raúl Cumplido and Rok Mihevc

Contents

Who are we?

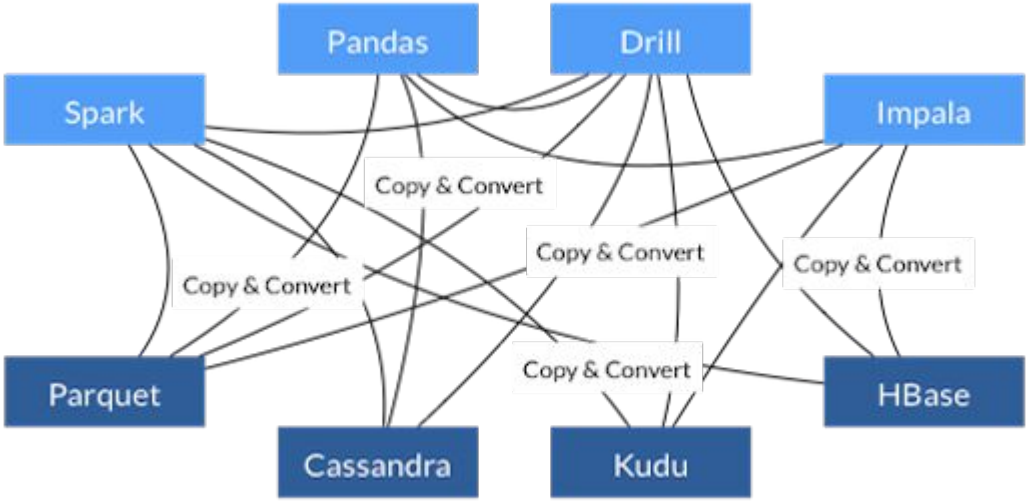
- Rok Mihevc
- Independent (Arctos Alliance)
- Arrow C++, Parquet

- Raúl Cumplido
- QuantStack
- PyArrow, Arrow C++, CI and general project maintenance

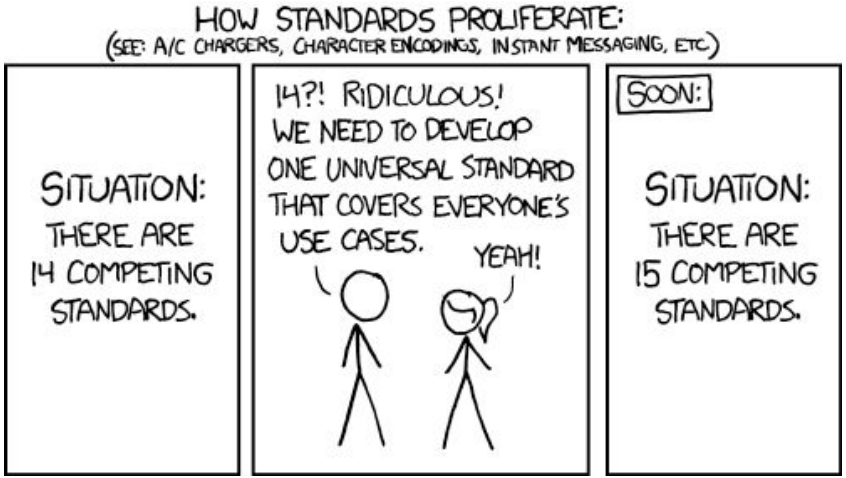
- Alenka Frim
- Independent (Arctos Alliance)
- PyArrow and general project maintenance

What is Apache Arrow?

The initial Problem

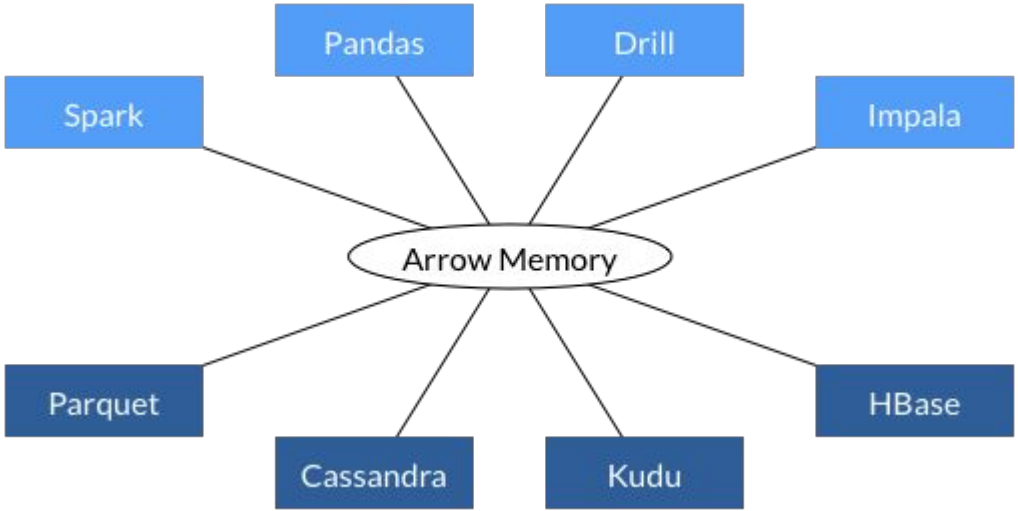


The initial idea



https://xkcd.com/927/

The idea becomes reality

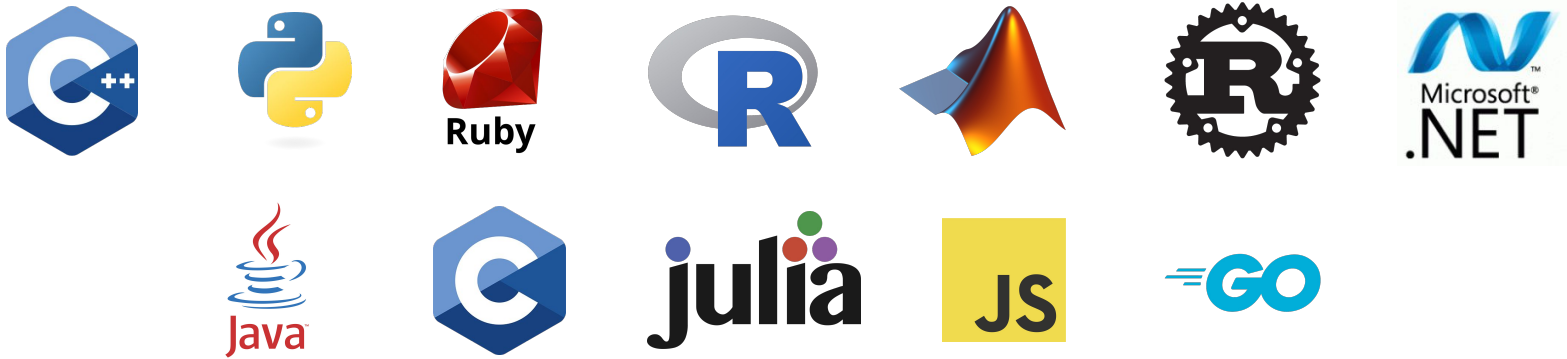


What is Arrow?

Apache Arrow is a multi-language toolbox for building high performance applications that process and transport large data sets. It is designed to both improve the performance of analytical algorithms and the efficiency of moving data from one system or programming language to another.

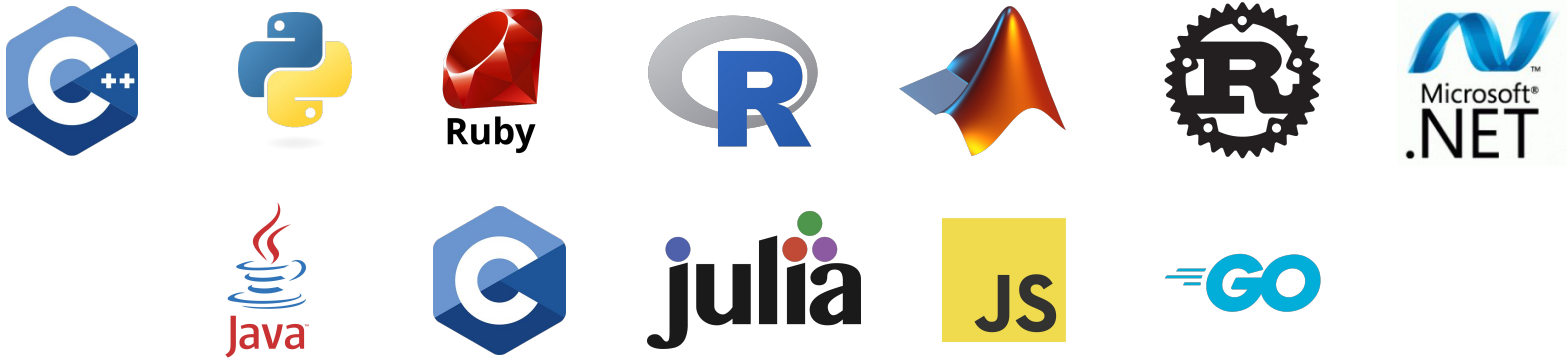
What is Arrow?

Apache Arrow is a **multi-language** toolbox for building high performance applications that process and transport large data sets. It is designed to both improve the performance of analytical algorithms and the **efficiency of moving data from one** system or **programming language to another**.



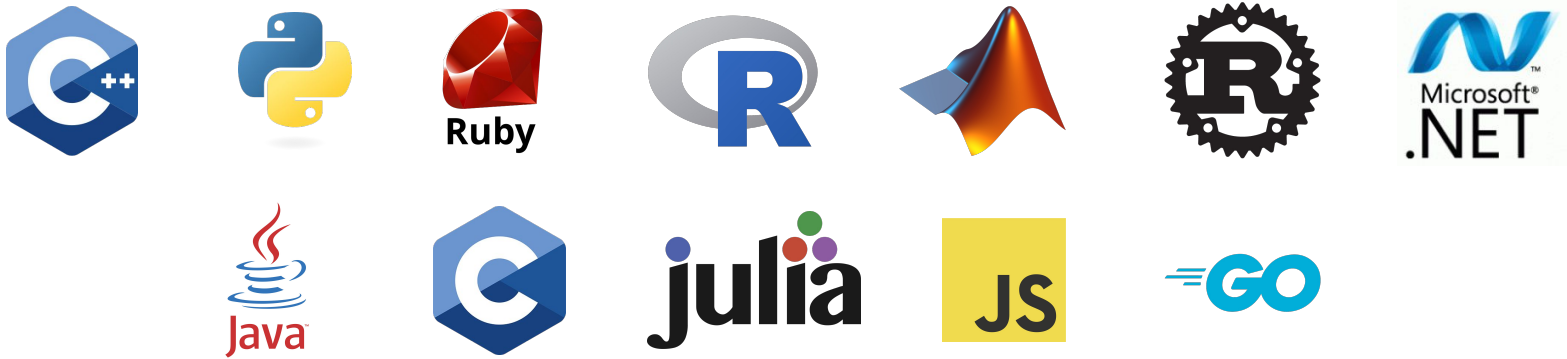
What is Arrow?

Apache Arrow is a multi-language **toolbox** for building high performance applications that process and transport large data sets. It is designed to both improve the performance of analytical algorithms and the efficiency of moving data from one system or programming language to another.



What is Arrow?

Apache Arrow is a multi-language toolbox **for building high performance applications that process and transport large data sets**. It is designed to both improve the performance of analytical algorithms and the efficiency of moving data from one system or programming language to another.

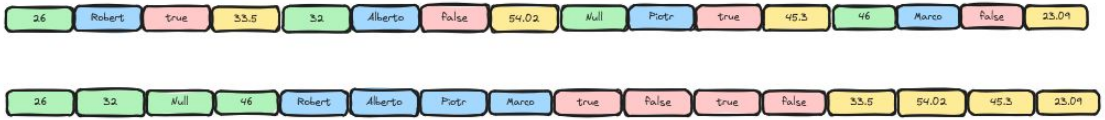


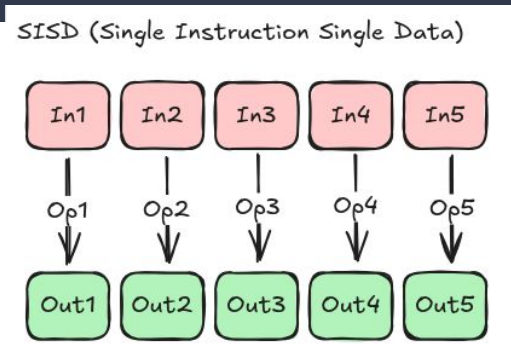
Columnar vs Row

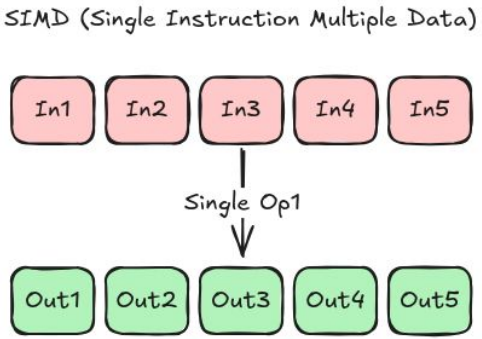
Age	Name	subscribed	spent
26	Robert	true	33.5
32	Alberto	false	54.02
Null	Piotr	true	45.3
46	Marco	false	23.09

Other advantages of columnar format

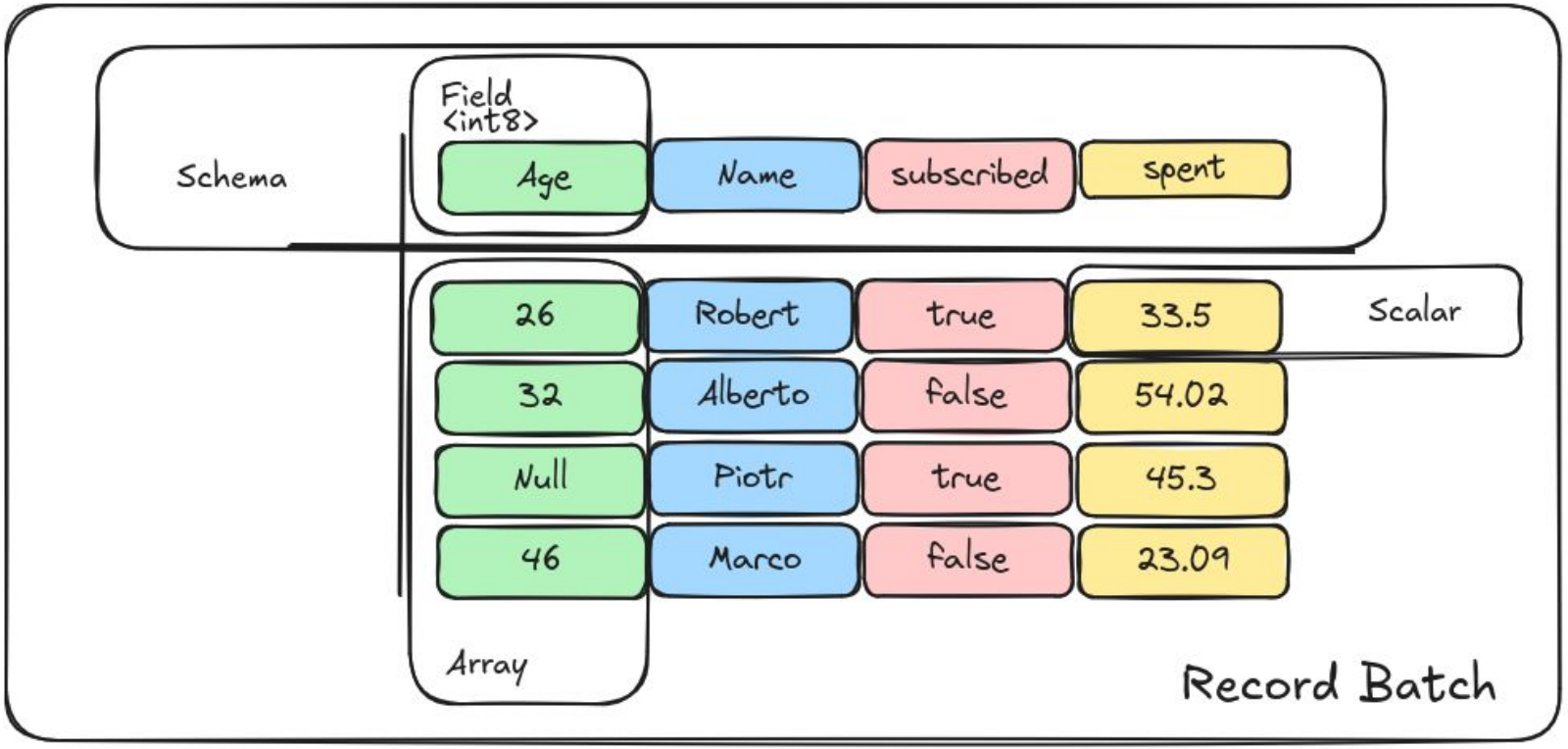
- SIMD Optimizations:
  - Allow us to perform an operation over multiple data
- Better Compression Algorithms







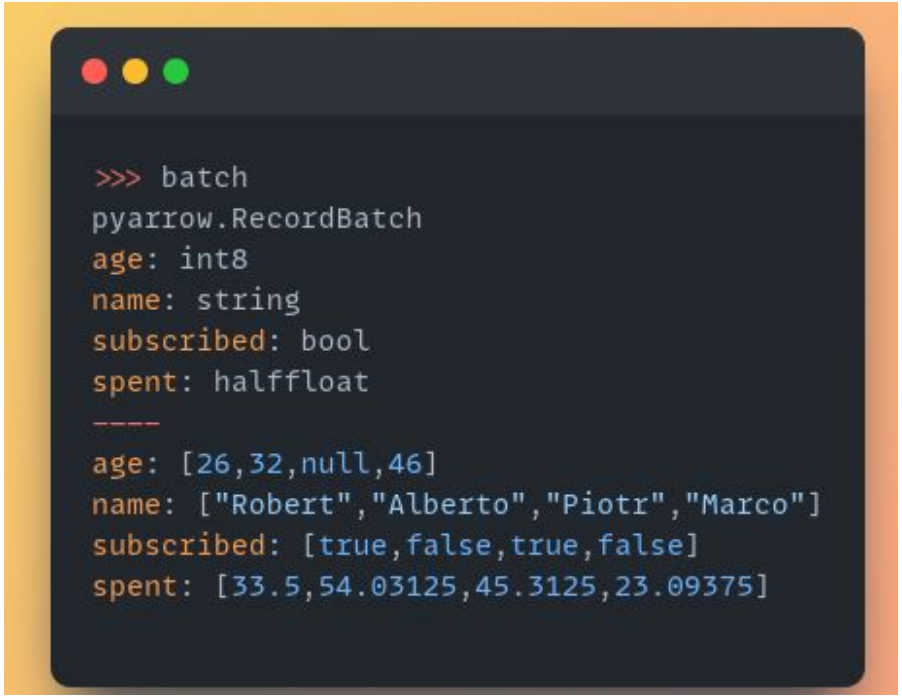
Record batch and Array



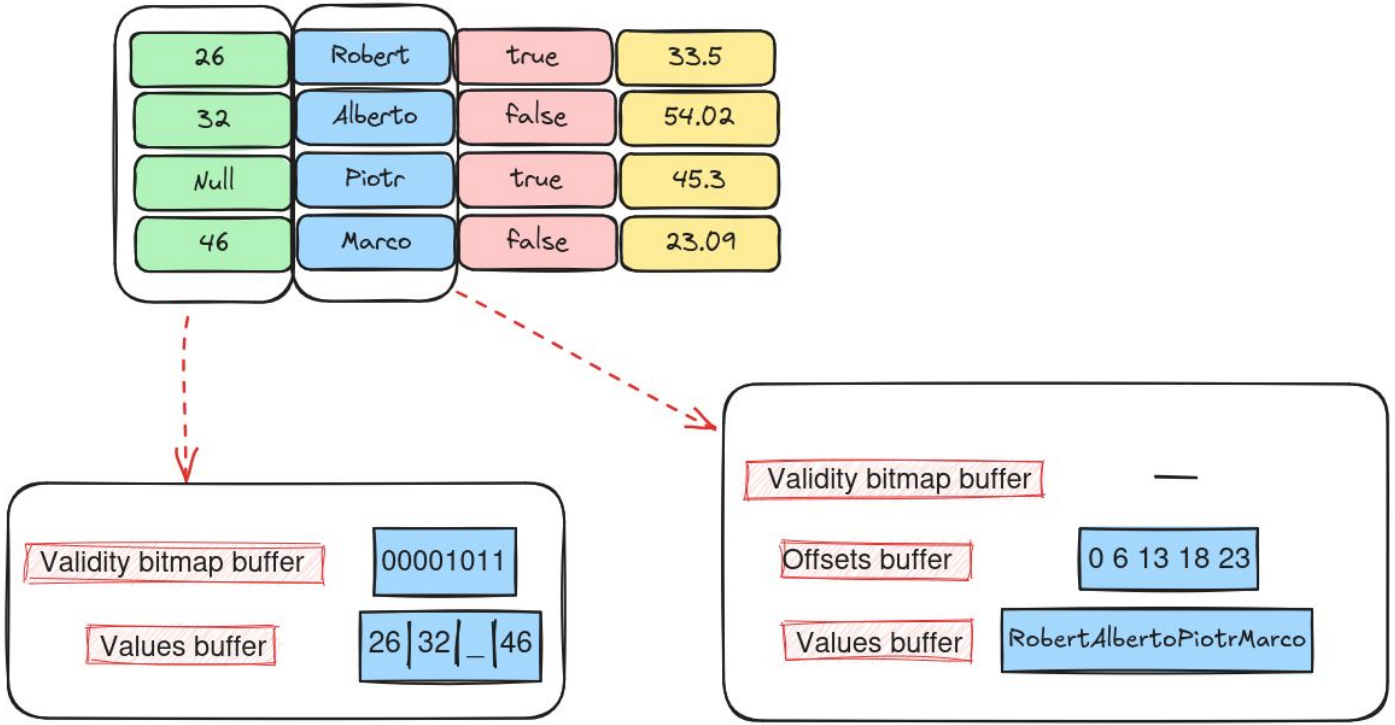
Record batch and Array

```
schema = pa.schema([pa.field('age', pa.int8()),
                    pa.field('name', pa.string()),
                    pa.field('subscribed', pa.bool_()),
                    pa.field('spent', pa.float16())])
a0 = pa.array([26, 32, None, 46], type=pa.int8())
a1 = pa.array(["Robert", "Alberto", "Piotr", "Marco"], type=pa.string())
a2 = pa.array([True, False, True, False], type=pa.bool_())
a3 = pa.array([33.5, 54.02, 45.3, 23.09], type=pa.float16())
batch = pa.record_batch([a0, a1, a2, a3], schema=schema)
```

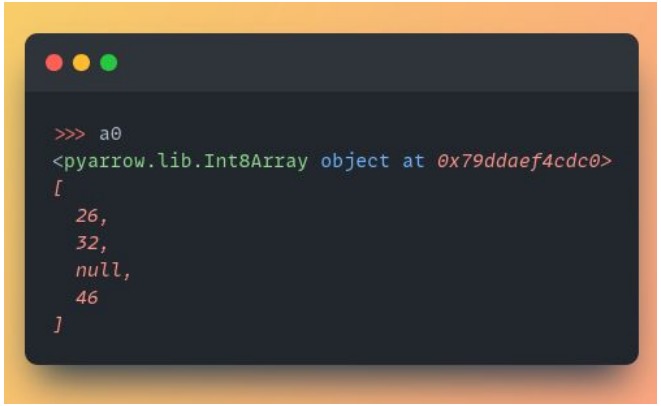
Record batch and Array

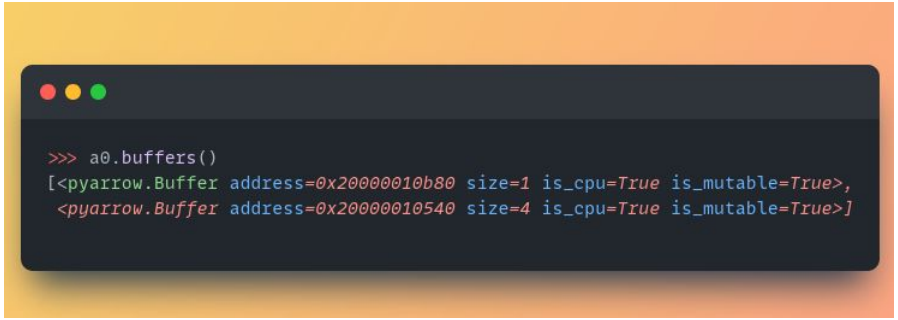


Arrow format specification

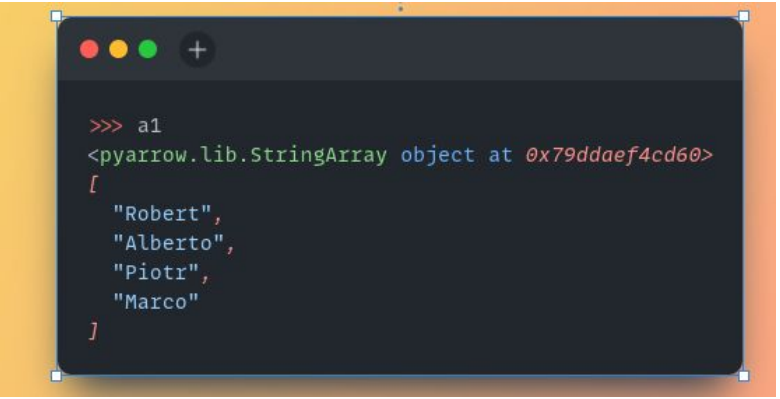


Array and buffers





Array and buffers



```
>>> a1
<pyarrow.lib.StringArray object at 0x79ddaef4cd60>
[
  "Robert",
  "Alberto",
  "Piotr",
  "Marco"
]
```

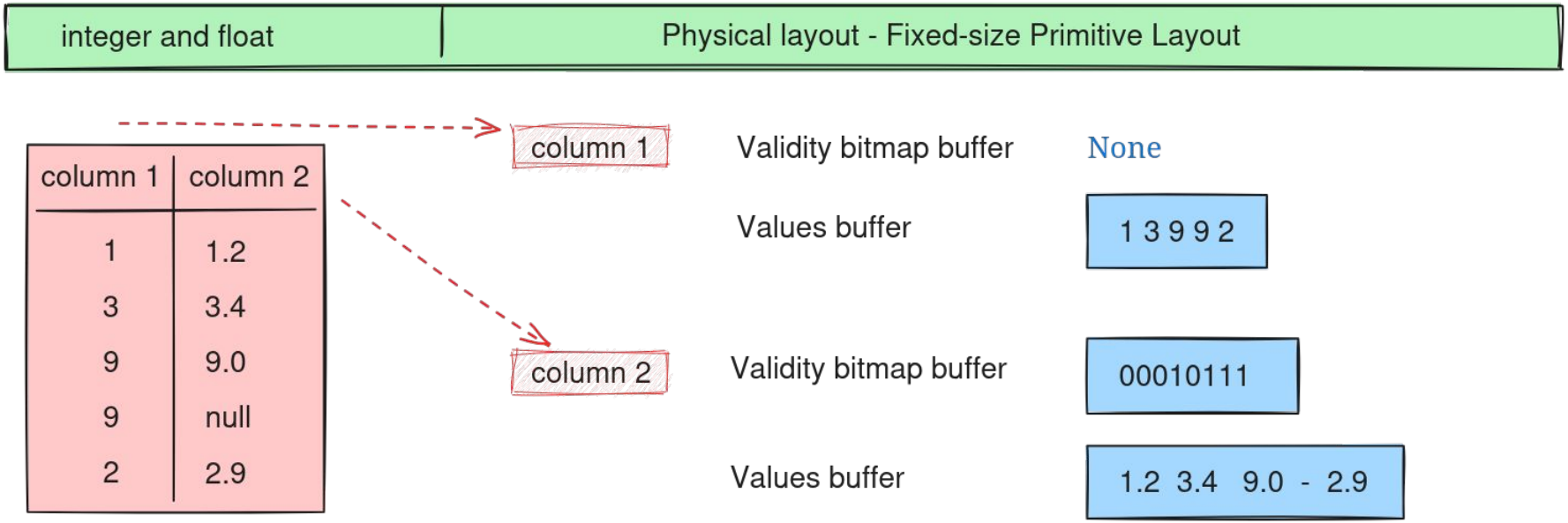
```
>>> a1.buffers()
[None,
 <pyarrow.Buffer address=0x20000010a40 size=20 is_cpu=True is_mutable=True>,
 <pyarrow.Buffer address=0x20000010200 size=23 is_cpu=True is_mutable=True>]
```

Array and buffers

```
>>> na.array(a0).inspect()
<ArrowArray int8>
- length: 4
- offset: 0
- null_count: 1
- buffers[2]:
  - validity <bool[1 b] 11010000>
  - data <int8[4 b] 26 32 0 46>
- dictionary: NULL
- children[0]:
```

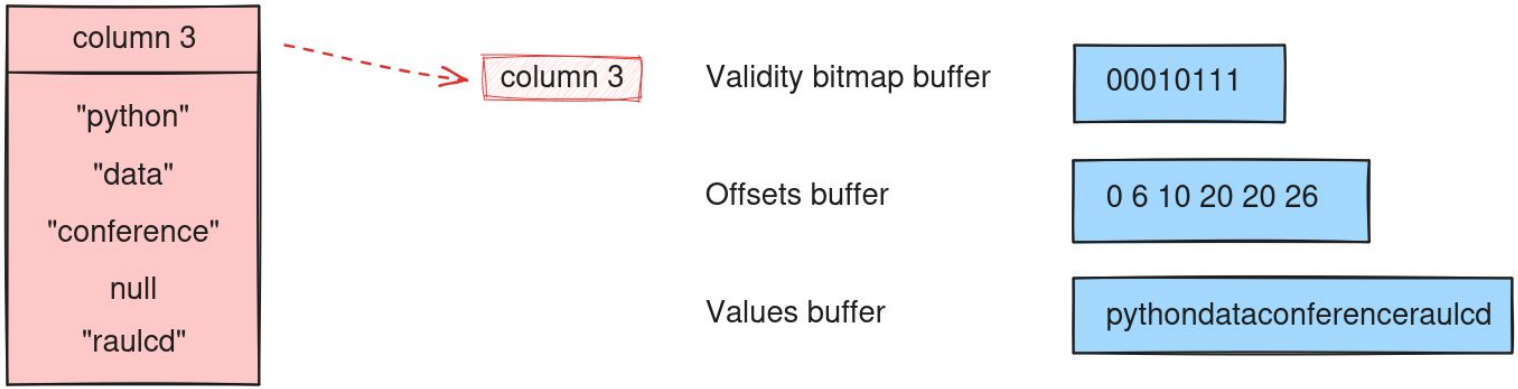
```
>>> na.array(a1).inspect()
<ArrowArray string>
- length: 4
- offset: 0
- null_count: 0
- buffers[3]:
  - validity <bool[0 b] >
  - data_offset <int32[20 b] 0 6 13 18 23>
  - data <string[23 b] b'RobertAlbertoPiotrMarco'>
- dictionary: NULL
- children[0]:
```

The In-memory format specification

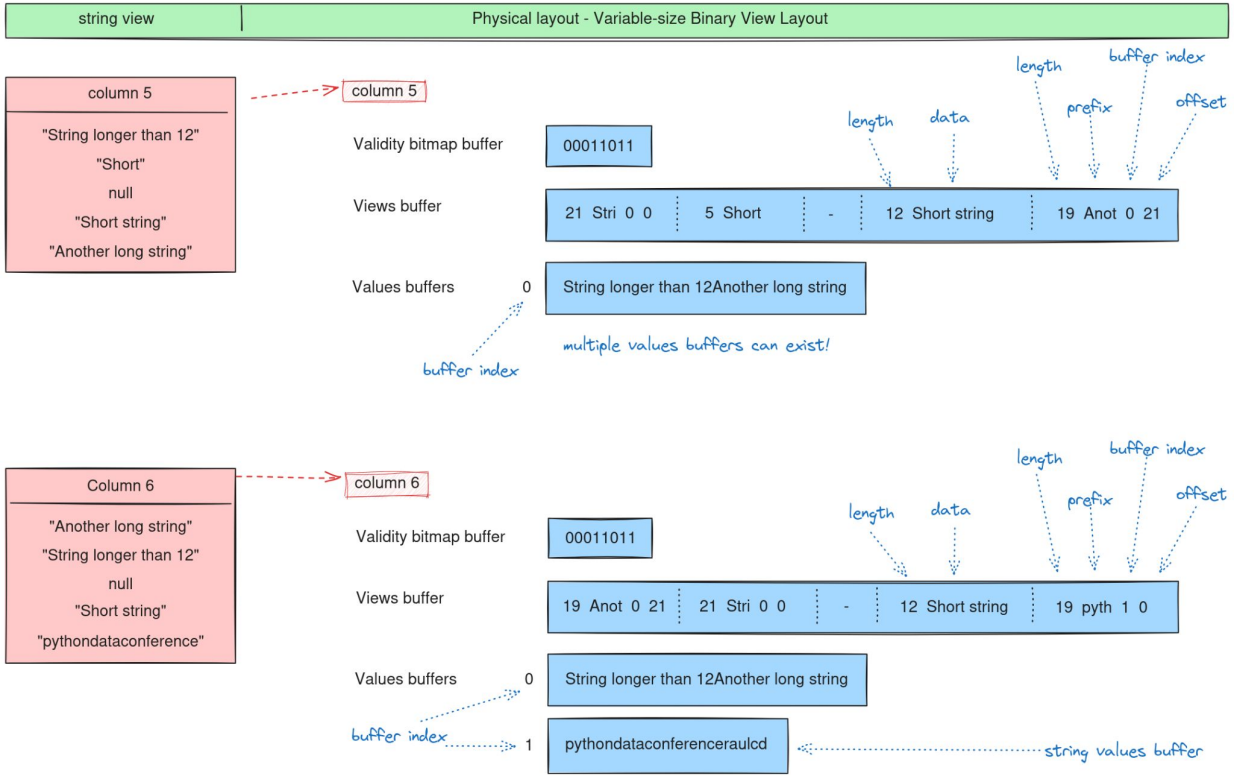


The In-memory format specification

`binary (utf8), large_binary`
`string, large_string`

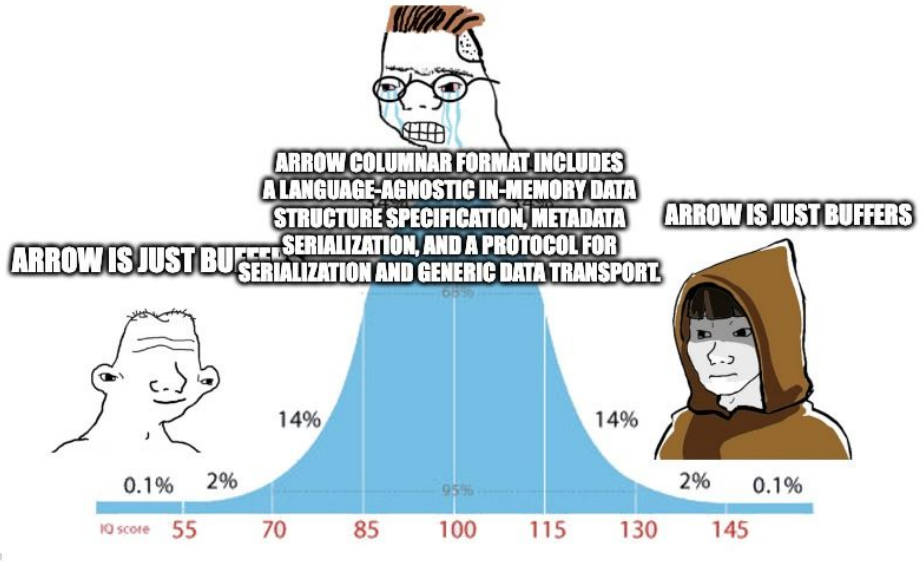


The In-memory format specification



To sum up

- Arrow defines how arrays and tables look like in memory
- Arrow implementations provide a toolset to work with such columnar data structure

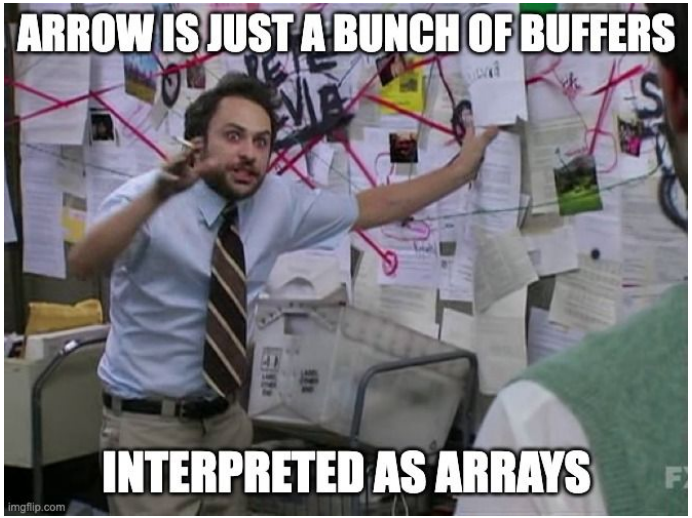


To sum up

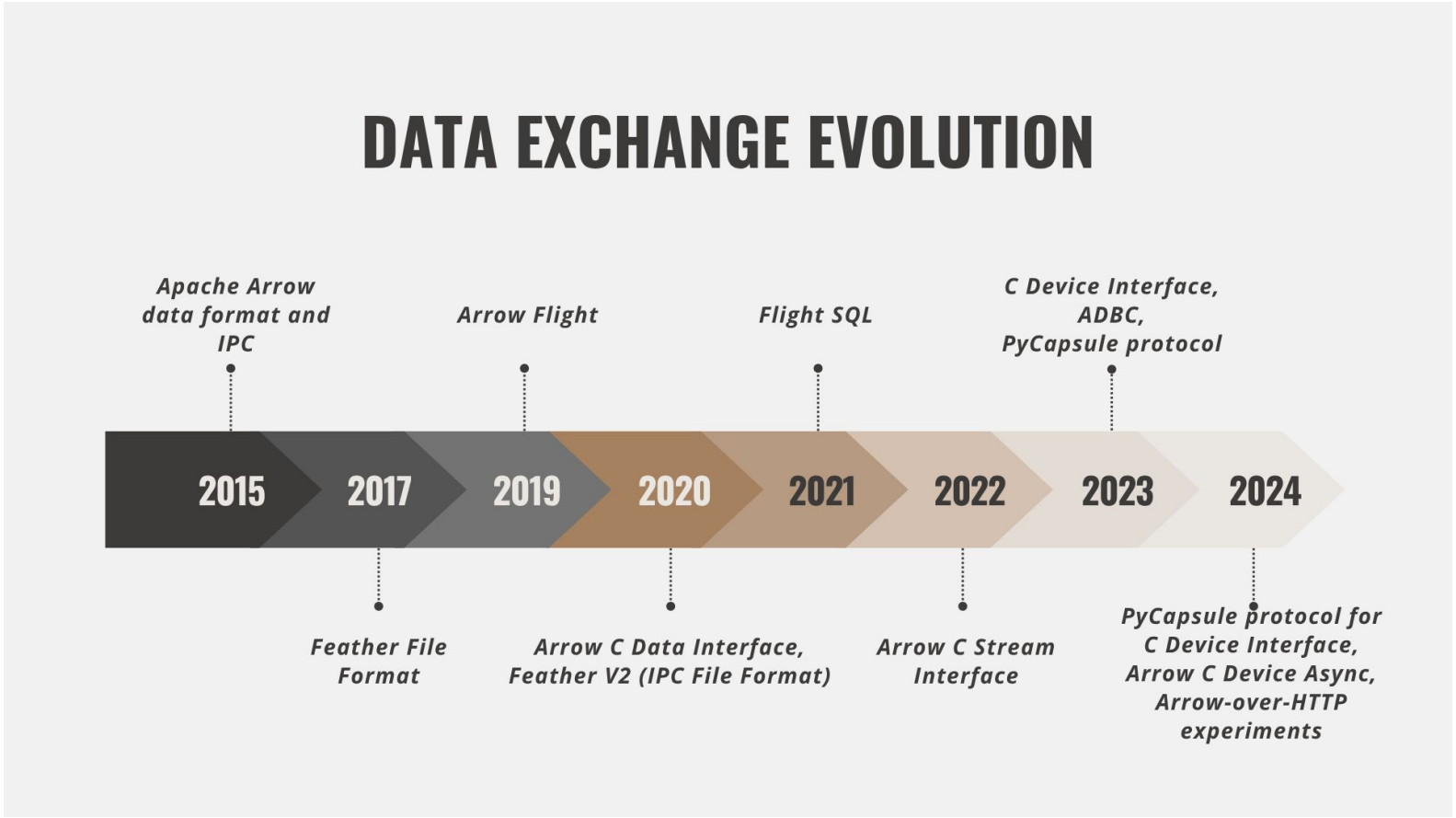
- Arrow defines how arrays and tables look like in memory
- Arrow implementations provide a toolset to work with such columnar data structure

To sum up

- Arrow defines how arrays and tables look like in memory
- Arrow implementations provide a toolset to work with such columnar data structure

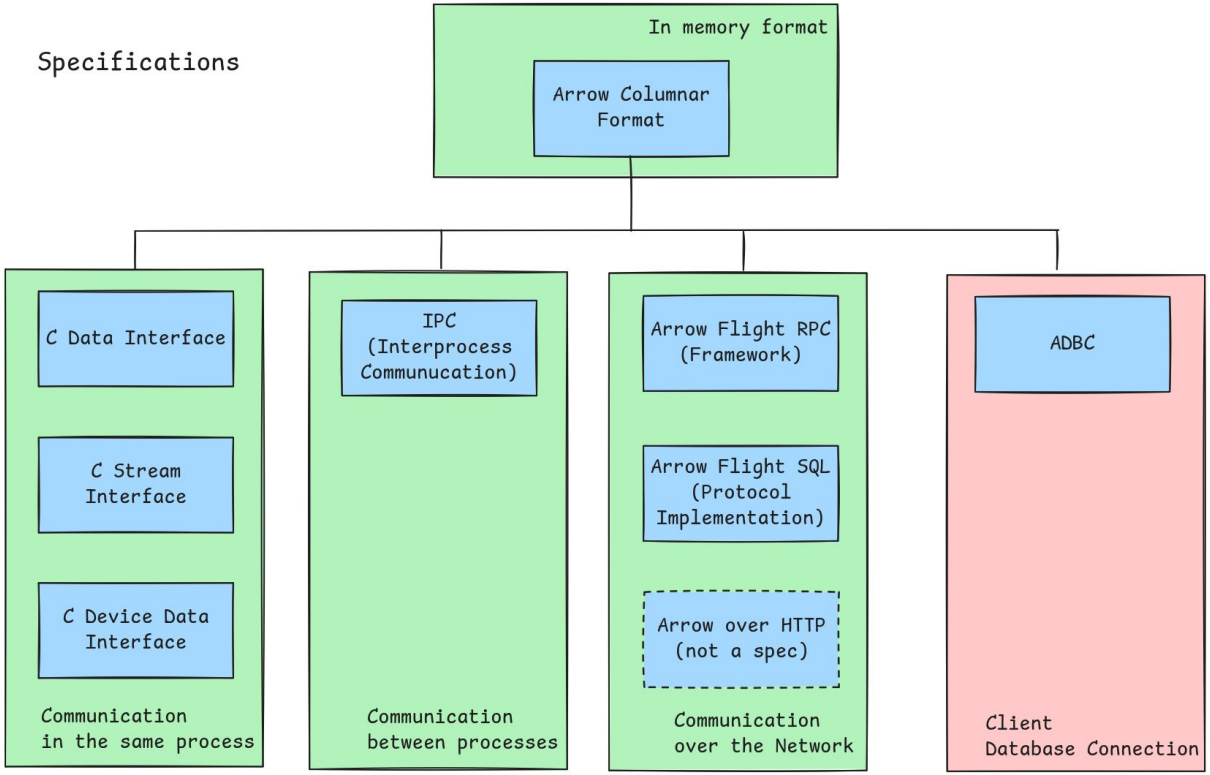


Arrow data Exchange

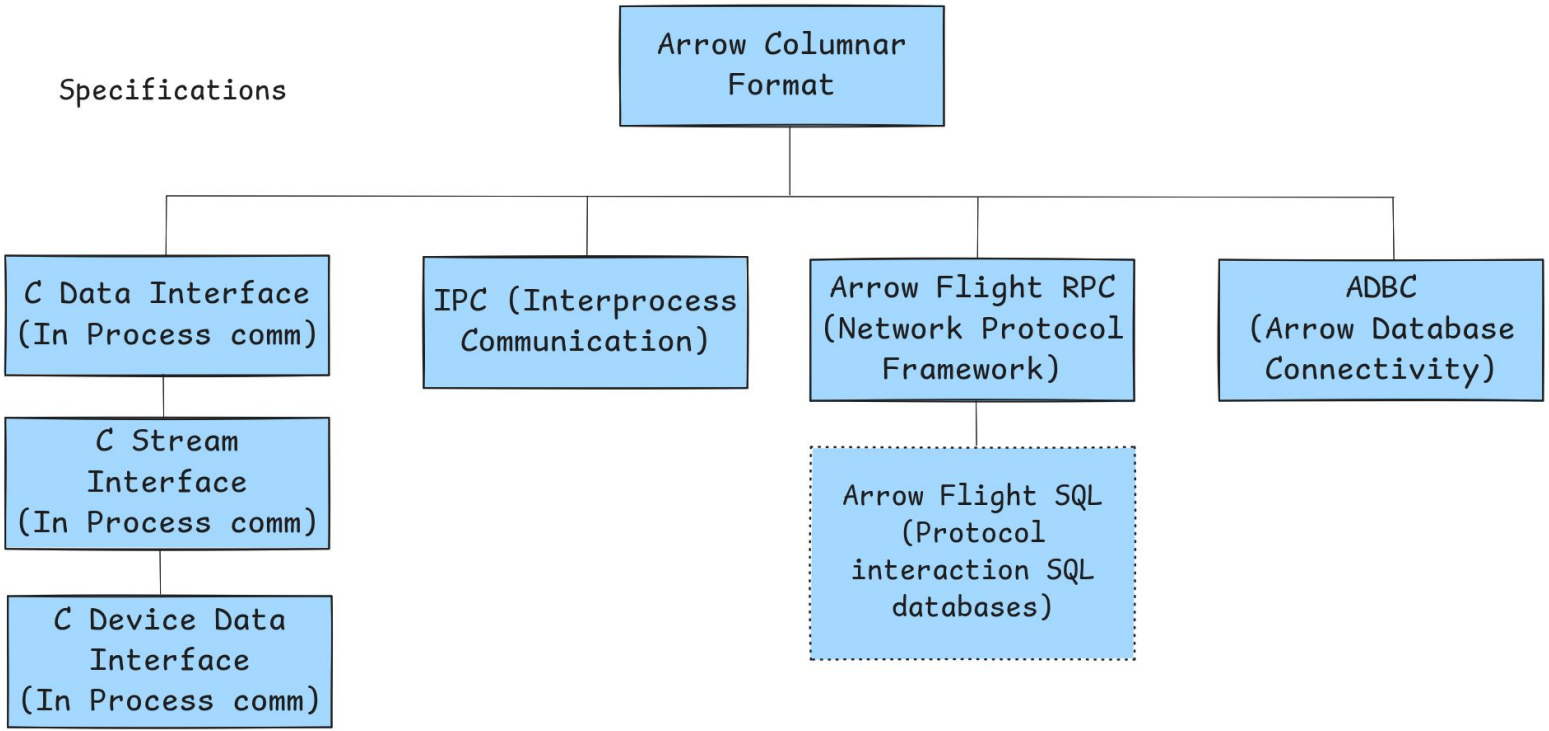


Evolution of the Arrow project around the data exchange

Overview


Specifications
In memory format
Arrow Columnar Format
C Data Interface
C Stream Interface
C Device Data Interface
Communication in the same process
IPC (Interprocess Communucation)
Communication between processes
Arrow Flight RPC (Framework)
Arrow Flight SQL (Protocol Implementation)
Arrow over HTTP (not a spec)
Communication over the Network
ADBC
Client Database Connection

Overview


Specifications
Arrow Columnar Format
C Data Interface (In Process comm)
IPC (Interprocess Communication)
Arrow Flight RPC (Network Protocol Framework)
ADBC (Arrow Database Connectivity)
C Stream Interface (In Process comm)
C Device Data Interface (In Process comm)
Arrow Flight SQL (Protocol interaction SQL databases)

Serialization and Interprocess Communication (IPC)

IPC (Interprocess Communication)

A protocol for

- turning Arrow record batches into a one-way stream of binary messages,
- and then rebuilding them on the other side
- — all without copying data in memory.

https://arrow.apache.org/docs/format/Columnar.html#serialization-and-interprocess-communication-ipc

IPC (Interprocess Communication)

Arrow IPC comes in two flavors

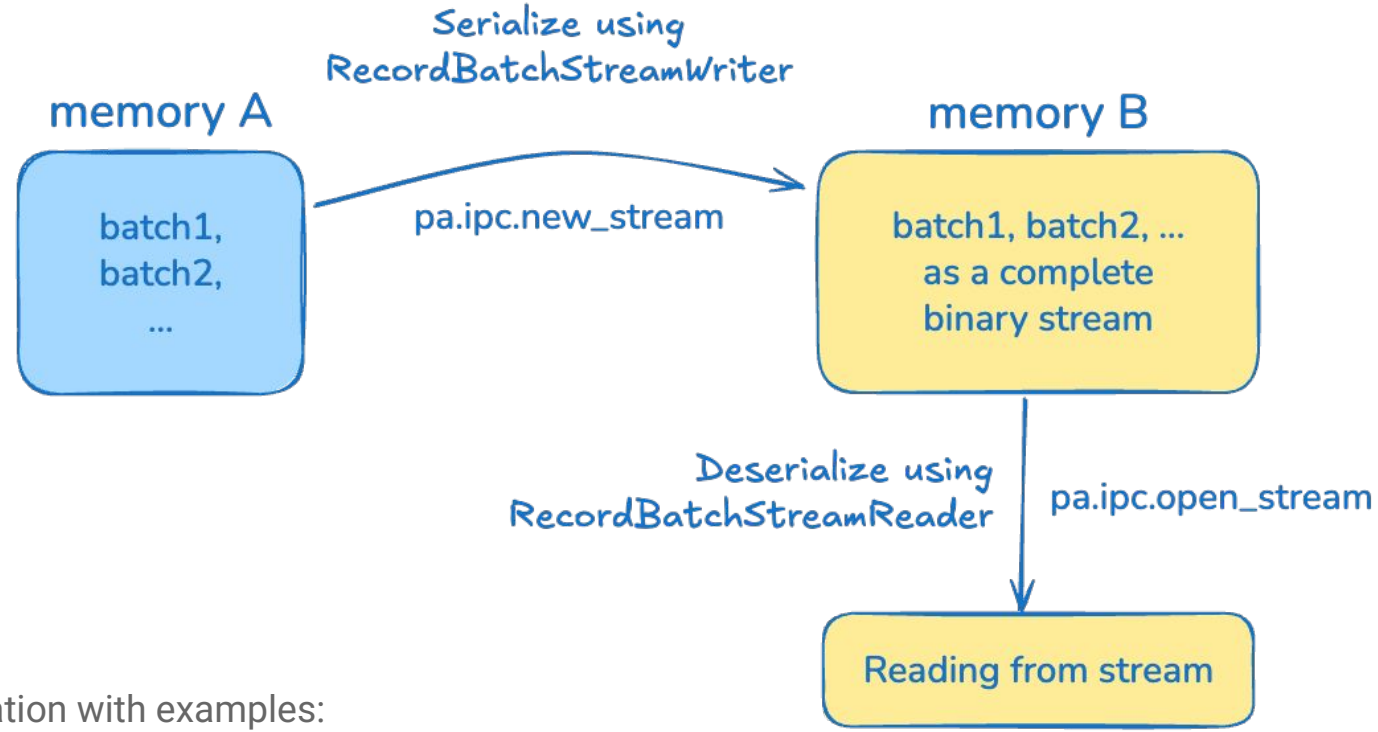
Stream format

- Sending an arbitrary length sequence of record batches
- For transferring data over a network

File format

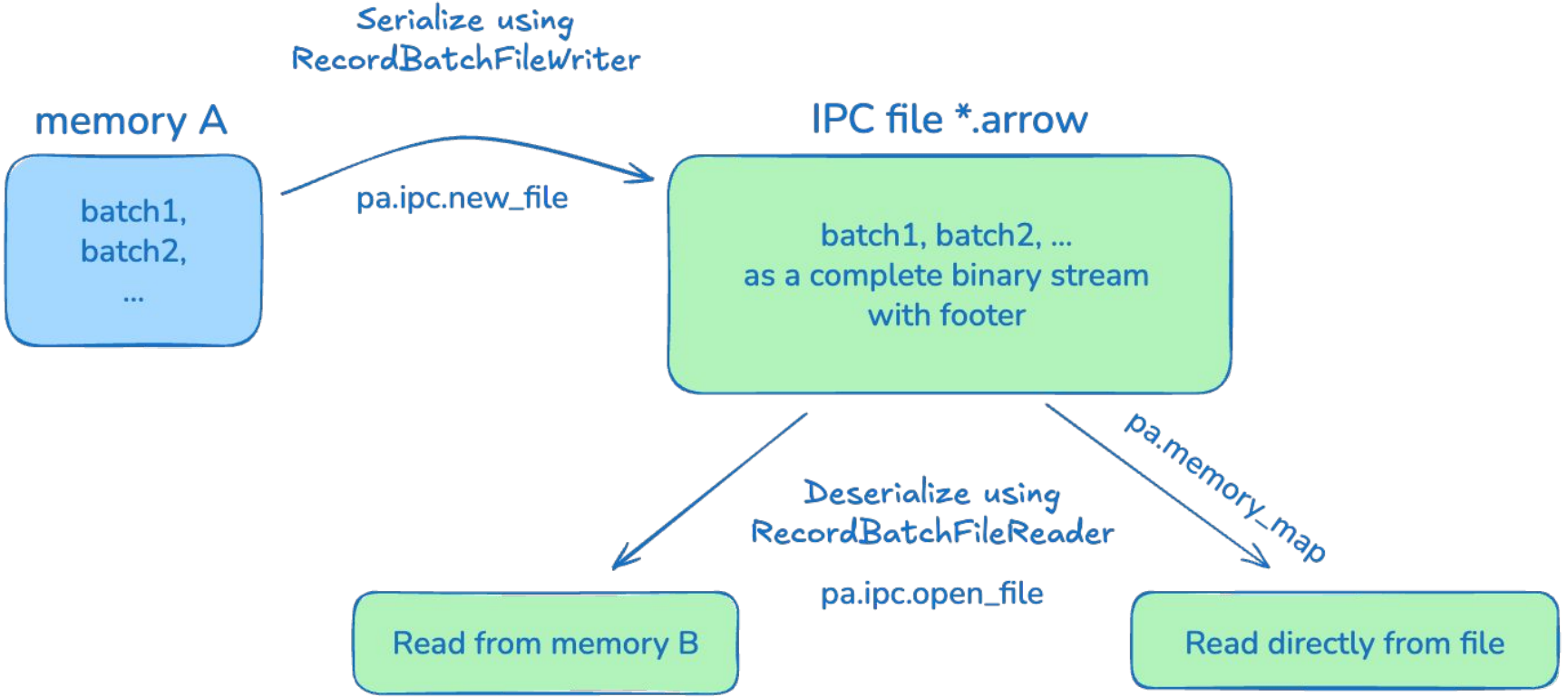
- Serializing a fixed number of record batches
- Includes an index (footer) for random access

IPC Streaming Format



Documentation with examples:
https://arrow.apache.org/docs/python/ipc.html

IPC File Format



IPC Example

```
>>> # Create two record batches
... batch1 = pa.record_batch(
...     [pa.array([0, 0]), pa.array(["first", "batch"])],
...     names=["id", "label"])
... batch2 = pa.record_batch(
...     [pa.array([1, 1]), pa.array(["second", "batch"])],
...     names=["id", "label"])
```

IPC Stream Example

We are using an in-memory Arrow buffer stream (sink).
But this can be a socket or some other IO sink.

```
>>> sink = pa.BufferOutputStream()
...
... with pa.ipc.new_stream(sink, batch1.schema) as writer:
...     writer.write_batch(batch1)
...     writer.write_batch(batch2)
...
```

IPC Stream Example

```
>>> # buf contains the complete stream as an in-memory byte buffer
... buf = sink.getvalue()
...
... with pa.ipc.open_stream(buf) as reader:
...         # reader is a RecordBatchStreamReader
...         reader.read_next_batch()["label"].to_pylist()
...         reader.read_next_batch()["label"].to_pylist()
...
['first', 'batch']
['second', 'batch']
```

IPC File and Memory Map Example

```
>>> with pa.OSFile('ipc_example.arrow', 'wb') as sink:
...     with pa.ipc.new_file(sink, batch1.schema) as writer:
...         writer.write(batch1)
...         writer.write(batch2)
...
```

IPC File and Memory Map Example

Using memory mapping without any memory allocation or copying

```
>>> with pa.memory_map("ipc_example.arrow", "r") as source:
...     # reader is a RecordBatchFileReader
...     with pa.ipc.open_file(source) as reader:
...         reader.get_batch(1)
...
pyarrow.RecordBatch
id: int64
label: string
----
id: [1,1]
label: ["second","batch"]
```

C Data Interface and Its Extensions: Stream and Device Interfaces

C Data Interface

- C ABI interface (C Application Binary Interface)
  Interface to compiled code (“API for compiled code”)

- For zero-copy interchange of Arrow columnar data structures
- At runtime
- In the same process
- Without the need to link to Arrow libraries

- https://arrow.apache.org/docs/format/CDataInterface.html
- https://arrow.apache.org/blog/2020/05/03/introducing-arrow-c-data-interface/
- https://willayd.com/leveraging-the-arrow-c-data-interface.html

C Data Interface vs. IPC

https://arrow.apache.org/docs/format/CDataInterface.html#comparison-with-the-arrow-ipc-format

IPC (Interprocess Communication)	C Data Interface
across processes and machines	in-memory, inter-language
serialized data	zero-copy
binary stream format	C structs
needs Arrow IPC reader and writer	Only C ABI
Example: Arrow Flight	Example: PyArrow ↔ Pandas

C Data interface: free-standing definitions

```c
struct ArrowArray {
  // Array data description
  int64_t length;
  int64_t null_count;
  int64_t offset;
  int64_t n_buffers;
  int64_t n_children;
  const void** buffers;
  struct ArrowArray** children;
  struct ArrowArray* dictionary;

  // Release callback
  void (*release)(struct ArrowArray*);
  // Opaque producer-specific data
  void* private_data;
};
```

```c
struct ArrowSchema {
  // Array type description
  const char* format;
  const char* name;
  const char* metadata;
  int64_t flags;
  int64_t n_children;
  struct ArrowSchema** children;
  struct ArrowSchema* dictionary;

  // Release callback
  void (*release)(struct ArrowSchema*);
  // Opaque producer-specific data
  void* private_data;
};
```

Expanding C Data Interface

C Data Interface	Zero-copy sharing of Arrow columnar data in memory	Base layer ● C structs ● ArrowArray, ArrowSchema
C Stream Interface	Share streams of data batches	Built on top of C Data Interface with ArrowArrayStream struct ● Data chunks, same schema ● Blocking pull-style
C Device Interface (Device Stream, Async)	Extend to non-CPU memory	Expands C Data Interface ● to non-CPU memory

The Arrow PyCapsule Interface

Arrow PyCapsule Protocol

- C Data structs wrapped into a PyCapsule
- Capsules are a part of the Python C API
- Instead of returning raw integer pointers on export, PyCapsule is created using standardized “dunder” methods

Arrow PyCapsule Protocol

- C Data structs wrapped into a PyCapsule
- Capsules are a part of the Python C API
- Instead of returning raw integer pointers on export, PyCapsule is created using standardized “dunder” methods

- All C Data/C Device Interface benefits
- More robust
- No PyArrow dependence

- https://arrow.apache.org/docs/format/CDataInterface/PyCapsuleInterface.html
- https://docs.python.org/3/c-api/capsule.html

PyCapsule Protocol - example from PyArrow

```
>>> import pyarrow as pa
>>> arr = pa.array([17, 7, 2025])
>>> arr
<pyarrow.lib.Int64Array object at 0x117b5f760>
[
  17,
  7,
  2025
]
>>> pyarrow_add = arr.buffers()[1].address
>>> pyarrow_add
3312292397248
```

PyCapsule Protocol - example to Polars

```
>>> polars_series = pl.Series(arr)
>>> polars_series
shape: (3,)
Series: '' [i64]
[
        17
        7
        2025
]
>>> polars_add = polars_series._get_buffer_info()[0]
>>> polars_add
3312292397248
>>> polars_add == pyarrow_add
True
```

PyCapsule Protocol - example to Pandas

```
>>> import pandas as pd
>>> pandas_series = pd.Series(polars_series)
>>> pandas_series
0      17
1       7
2    2025
dtype: int64
>>> pandas_add = pandas_series.values.ctypes.data
>>> pandas_add
3312292397248
>>> pandas_add == polars_add == pyarrow_add
True
```

PyCapsule Protocol - also to ...

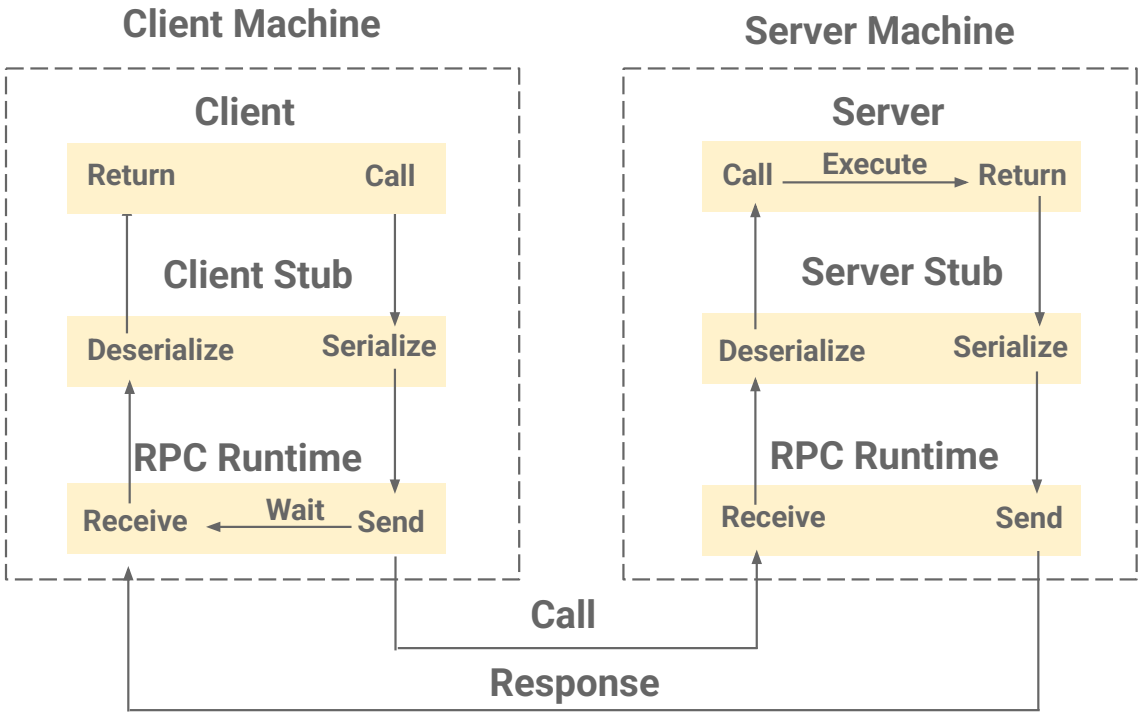
... Ibis, arro3, GDAL, narwhals, quak, DataFusion, DuckDB, GeoPandas, cuDF, ...

https://github.com/apache/arrow/issues/39195#issuecomment-2245718008

Flight RPC

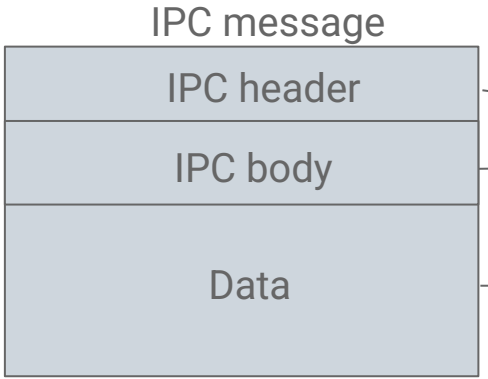
Remote procedure call (RPC) frameworks

- Frameworks for distributed computing that allow executing routines on other machines
- Most RPCs have a language to describe interfaces (stubs) - think of it as REST where you can define your own methods
- Note that network traffic is here needs to be serialized for sending



Flight RPC - what is flight

- What is it
  - RPC framework for high-performance data services
  - IPC format “serialized” with protobuf and sent over gRPC
  - Designed for zero-copy serialization and parallel data transfer




```
message FlightData {
  FlightDescriptor flight_descriptor = 1;
  bytes data_header = 2;
  bytes app_metadata = 3;
  bytes data_body = 1000;
}
```

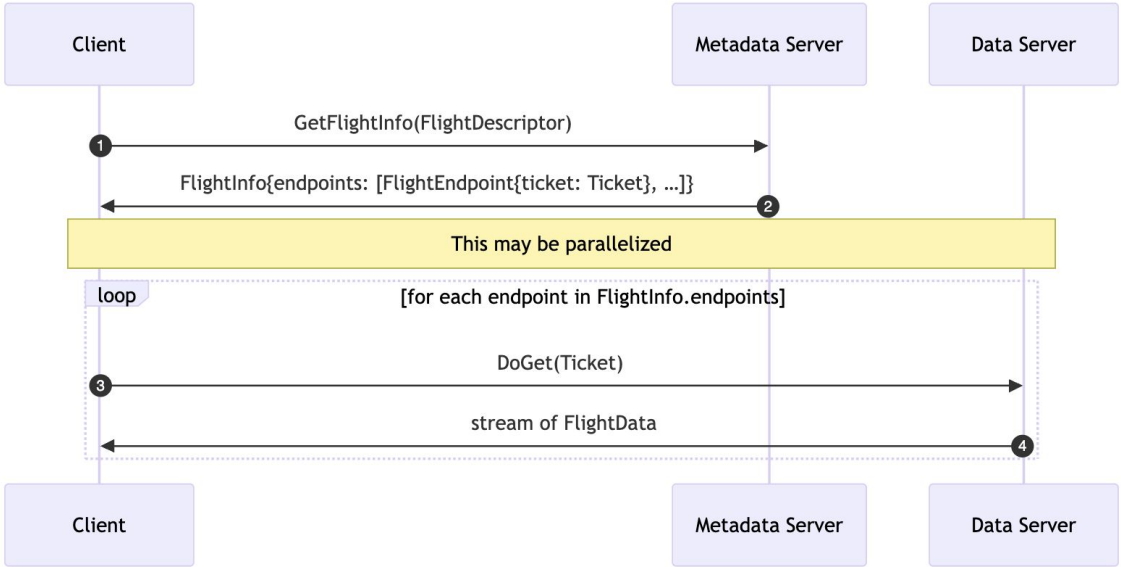
Flight RPC - when to use it

- When does it make sense to use it
  - Server and client use Arrow layout data
  - Moving large batches over the network

- When does it not make sense to use it
  - Row-oriented data
  - Small volume sent which can’t be batched

Flight RPC - how it works

- Server implements a set of RPC endpoints, e.g. GetFlightInfo
- FlightInfo message includes schema and endpoints where data can be retrieved from



Retrieving data via `DoGet`.

Arrow over HTTP

See https://github.com/apache/arrow-experiments/tree/main/http

Arrow over HTTP - client

To receive record batches as a client:

- Sends an HTTP GET request to a server.
- Receives an HTTP 200 response from the server, with the response body containing an Arrow IPC stream of record batches.
- Adds the record batches to a list as they are received.

```
import urllib.request
import pyarrow as pa

url = 'http://localhost:8008'
response = urllib.request.urlopen(url)

batches = []

with pa.ipc.open_stream(response) as reader:
    schema = reader.schema
    batches = [b for b in reader]
```

Arrow over HTTP - server

To send record batches from server:

- Write record batches as IPC into buffers
- Serve buffers via an HTTP endpoint as a

```
def generate_bytes(schema, batches):
    with pa.RecordBatchReader.from_batches(schema, batches) as source, \
            io.BytesIO() as sink, \
            pa.ipc.new_stream(sink, schema) as writer:
        for batch in source:
            sink.seek(0)
            writer.write_batch(batch)
            sink.truncate()
            with sink.getbuffer() as buffer:
                yield buffer

        sink.seek(0)
        writer.close()
        sink.truncate()
        with sink.getbuffer() as buffer:
            yield buffer
```

ADBC

Arrow Database Connectivity

ADBC

At a high level, ADBC is the standard for Arrow-native access to databases.
At a lower level, ADBC is two separate but related things:

- An abstract API for working with databases and Arrow data.
- A set of concrete implementations of that abstract API in different languages and drivers for different databases

ADBC

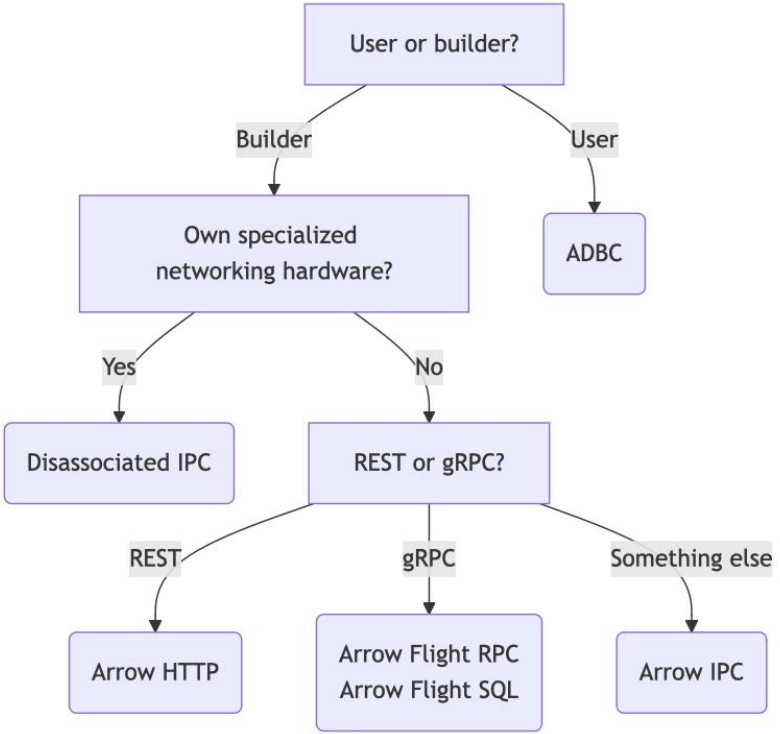
Driver	Supported Languages	Implementation Language	Status
Apache DataFusion	Rust	Rust	Experimental
BigQuery (C#)	C#	C#	Experimental
BigQuery (Go)	C/C++	Go	Experimental
DuckDB [1]	C/C++	C++	Stable
Flight SQL (Go)	C/C++, C# [3]	Go	Stable
Flight SQL (Java)	Java	Java	Experimental
JDBC Adapter	Java	Java	Experimental
PostgreSQL	C/C++	C++	Stable
SQLite	C/C++	C	Stable
Snowflake	C/C++, Rust [3]	Go	Stable
Thrift protocol-based [2]	C#	C#	Experimental

[1] DuckDB is developed and provided by a third party. See the DuckDB documentation for details.

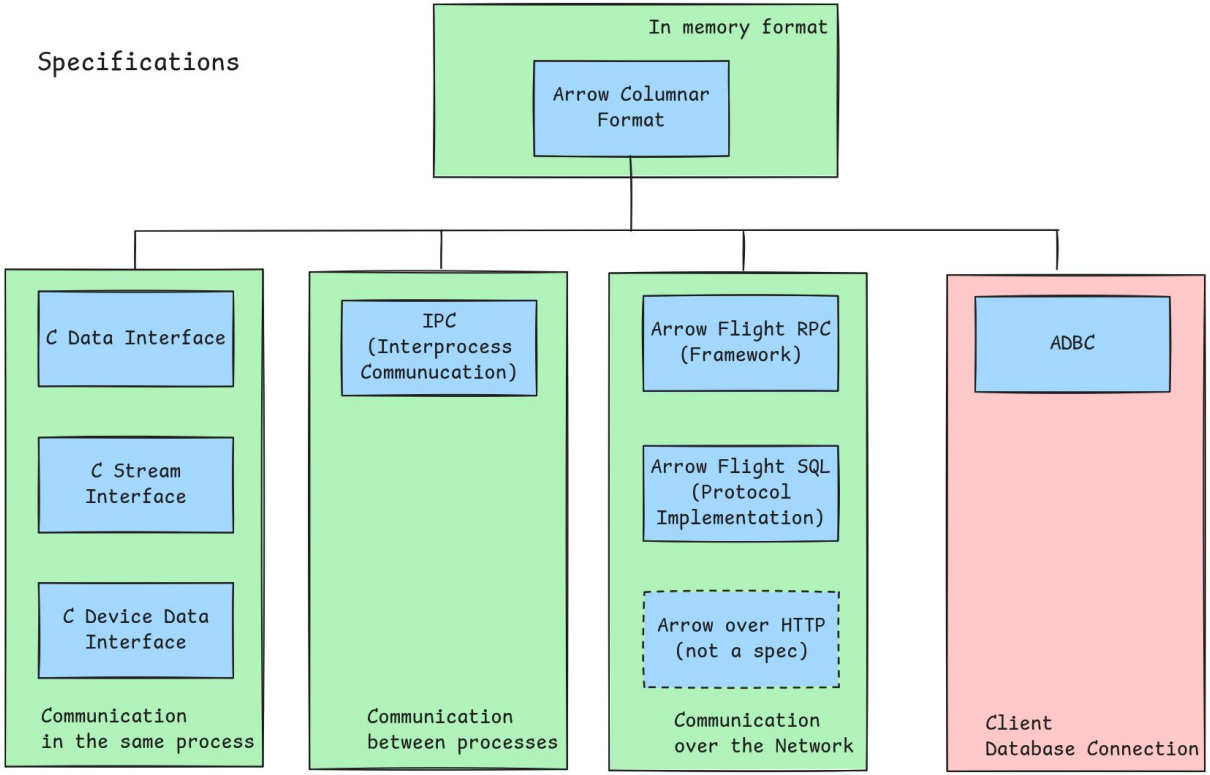
[2] Supports Apache Hive/Impala/Spark.

[3] (1,2) Listed separately because a wrapper package is provided that combines the driver and the bindings for you.

Summary


User or builder?
Builder
User
Own specialized networking hardware?
ADBC
Yes
No
Disassociated IPC
REST or gRPC?
REST
gRPC
Something else
Arrow HTTP
Arrow Flight RPC
Arrow Flight SQL
Arrow IPC

Summary


Specifications
In memory format
Arrow Columnar Format
C Data Interface
C Stream Interface
C Device Data Interface
Communication in the same process
IPC (Interprocess Communucation)
Communication between processes
Arrow Flight RPC (Framework)
Arrow Flight SQL (Protocol Implementation)
Arrow over HTTP (not a spec)
Communication over the Network
ADBC
Client Database Connection