# Apache Arrow

VLDB BOSS Workshop
2019-08-30

**Wes McKinney**

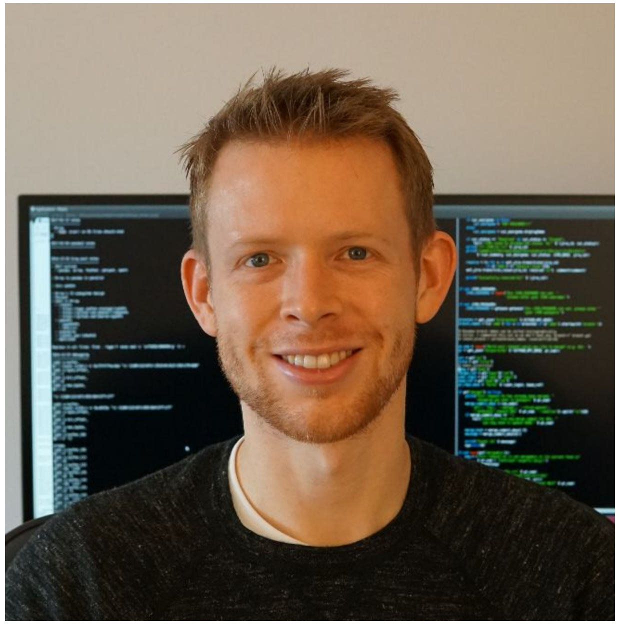
# Goals of this workshop

- Give working understanding of what Arrow is and is not
- Equip you to recognize Arrow use cases in the real world
- Solicit your involvement in the Apache Arrow community

# Arrow Workshop structure

- Arrow Use Cases Discussion
- Columnar Format and Binary Protocol
- C++ Libraries Overview
- Arrow Flight RPC Framework

# Wes McKinney



- Director of **Ursa Labs**, not-for-profit dev group working on **Apache Arrow**
- Created Python **pandas** project (~2008), lead developer/maintainer until 2013
- PMC Apache Arrow, Apache Parquet, ASF Member
- Wrote ***Python for Data Analysis*** (1e 2012, 2e 2017)
- Formerly: Two Sigma, Cloudera, DataPad, AQR

# Apache Arrow

- Open source initiative conceived in 2015
- Intersection of database systems, big data, and data science tools
- Purpose: Cross-language open standards and libraries to accelerate and simplify in-memory computing
- **https://github.com/apache/arrow**

# Apache Arrow: History and Status

- Launched in 2016, initially backed by developers of ~13 major open source data projects
- Project development status
  - Codebase 3.5 years old
  - > 300 distinct contributors
  - **14 major releases**
  - Some level of support in **11 programming languages** (C, C++, C#, Go, Java, JavaScript, MATLAB, Python, R, Ruby, Rust)

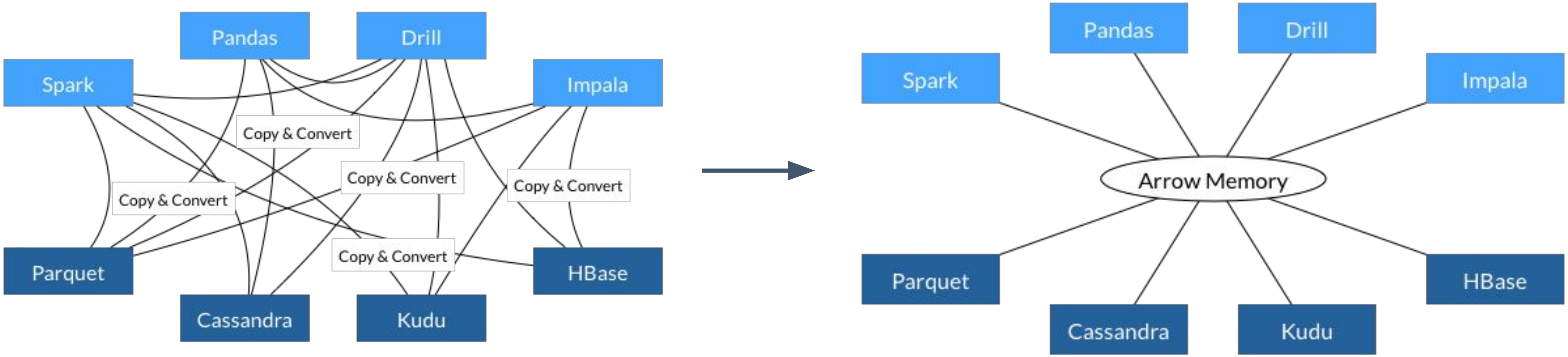
# Open standards: why do they matter?

- Simplify system architectures
- Reduce ecosystem fragmentation
- Improve interoperability
- Reuse more libraries and algorithms

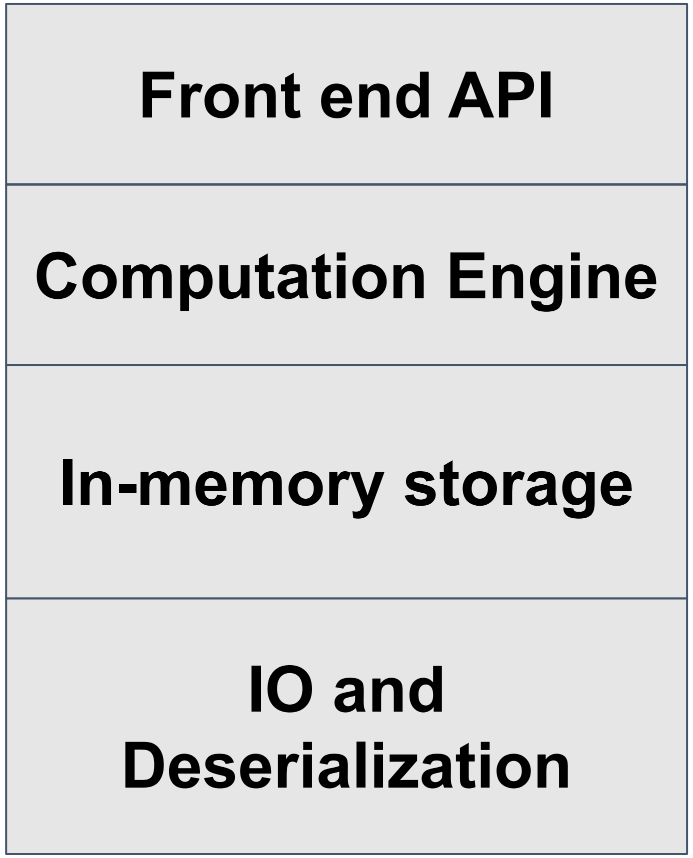
# Why create open standards for in-memory?

- Reuse computational algorithms
- Reuse IO / file format interfaces
- Move data structures without serializing
- Zero-copy shared memory access

# Defragmenting Data



# Analytic database architecture




- Vertically integrated / “Black Box”
- Internal components do not have a public API
- Users interact with front end

# Analytic database, deconstructed

**Front end API**

**Computation Engine**

**In-memory storage**

**IO and Deserialization**

- Components have public APIs
- Use what you need
- Different front ends can be developed

# Analytic database, deconstructed

~~Front end API~~

Computation Engine

In-memory storage

IO and Deserialization

**Arrow is front end agnostic**

# Some Arrow Use Cases

# Runtime data structures for analytics

- For analytical data processing systems, including databases
- Input and output of computational functions
- Examples:
  - Dremio
  - NVIDIA RAPIDS

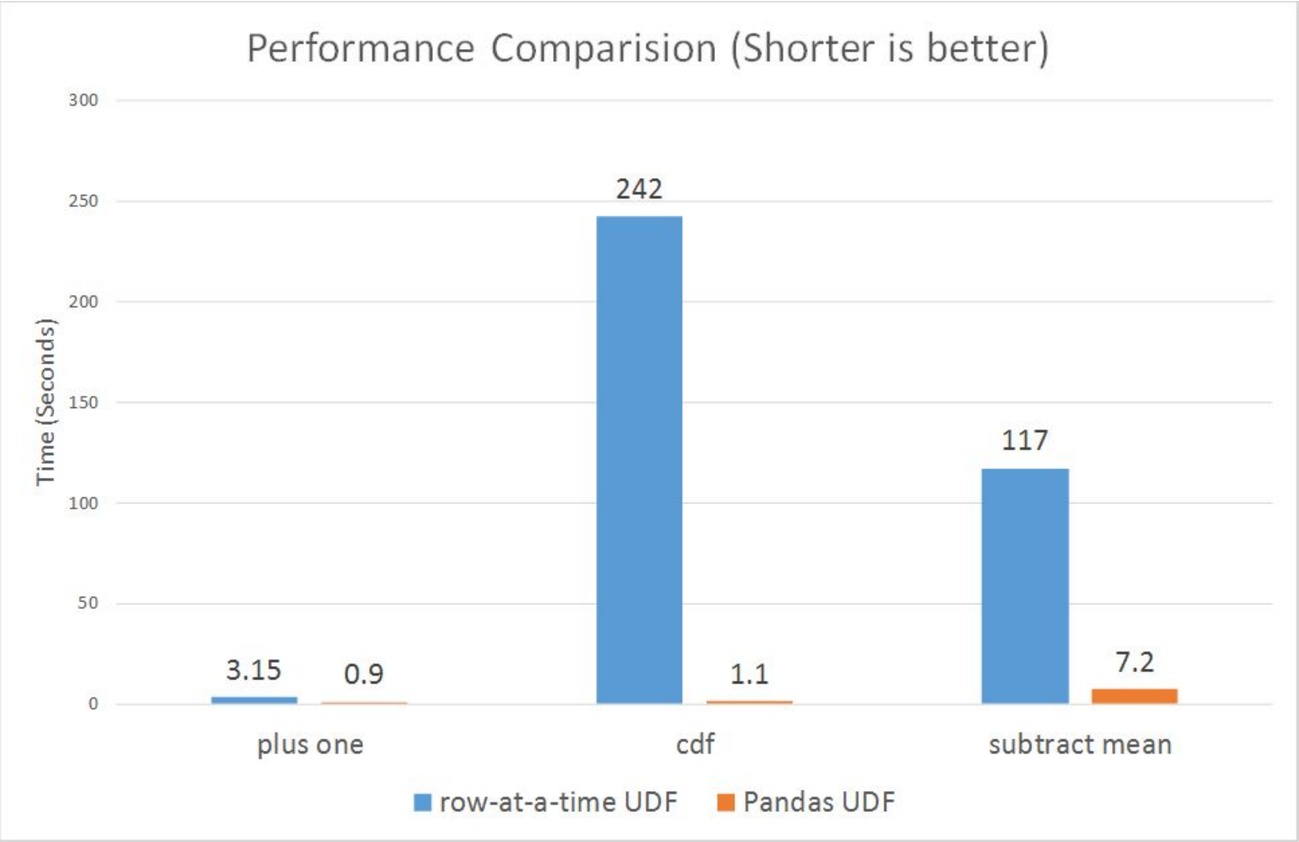
# Data Interchange (without deserialization)

- Zero-copy access through mmap
- On-wire RPC format
- Pass data structures across language boundaries in-memory without copying (e.g. Java to C++)
- Examples
  - Arrow Flight (over gRPC)
  - BigQuery Storage API
  - Apache Spark: Python / pandas user-defined functions
  - Dremio + Gandiva (Execute LLVM-compiled expressions inside Java-based query engine)

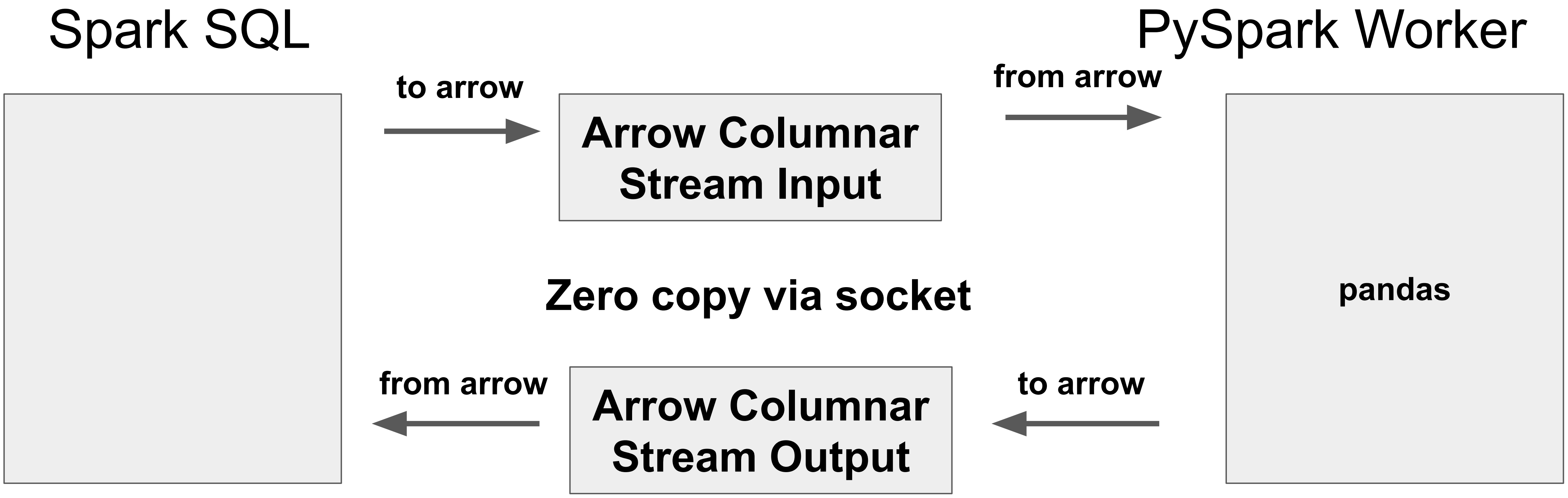
# Arrow-accelerated Python + Apache Spark

- Joint work with Li Jin from Two Sigma, Bryan Cutler from IBM
- Vectorized user-defined functions, fast data export to pandas

```
import pandas as pd
from scipy import stats
@pandas_udf('double')
def cdf(v):
    return pd.Series(stats.norm.cdf(v))
df.withColumn('cumulative_probability',
              cdf(df.v))
```

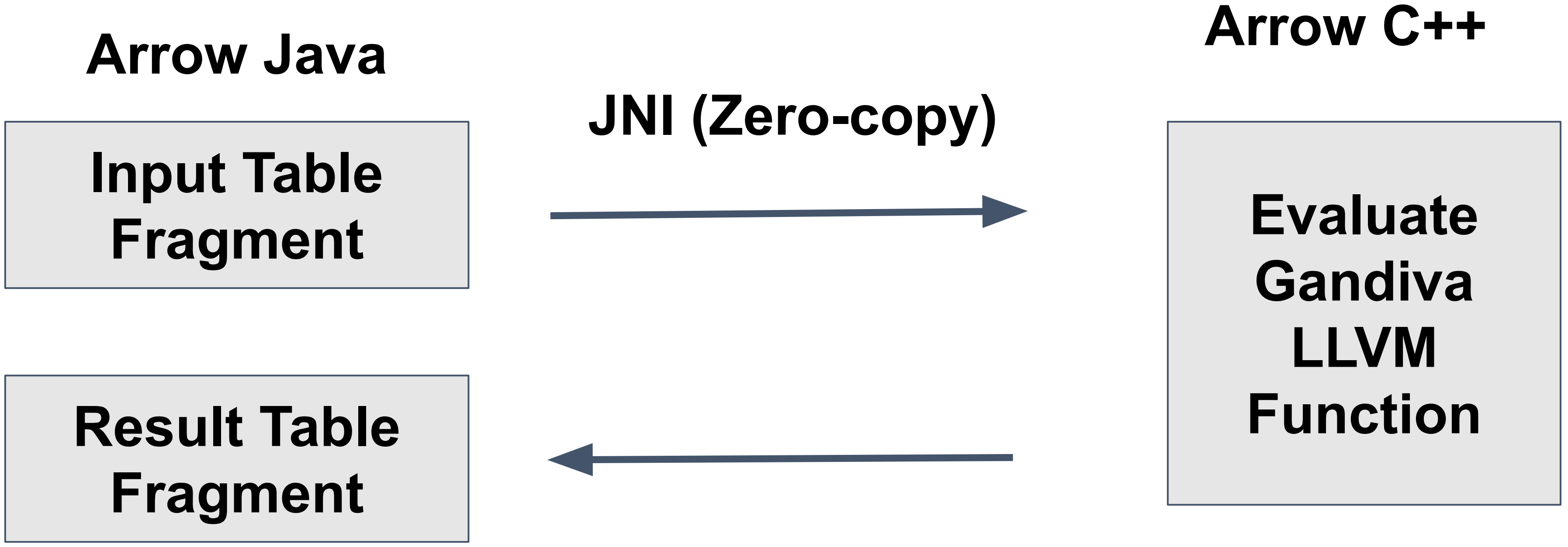


# Arrow-accelerated Python + Apache Spark



# Example: Gandiva, Arrow-LLVM compiler

```
SELECT year(timestamp), month(timestamp), …
FROM table
...
```




https://github.com/dremio/gandiva

# Caching and Memory-mapping

- Easily flush in-memory to disk without extra serialization
- RAM-resident data can be seamlessly replaced with mmap'd version

# Arrow Columnar Format and Binary Protocol

# Arrow’s Columnar Memory Format

- Runtime memory format for analytical query processing
  - Companion to serialization tech like Apache {Parquet, ORC}
- “Fully shredded” columnar, supports flat and nested schemas
- Organized for cache-efficient access on CPUs/GPUs
- Optimized for data locality, SIMD, parallel processing
- Accommodates both random access and scan workloads

# Columnar Array Data Structure

```
object Array {
  type: DataType,
  length: int64,
  null_count: int64,
  buffers: [Buffer],
  dictionary: Array or null,
  children: [Array]
}
```

A **Buffer** is a contiguous memory segment (defined by memory address and size)

# Metadata and data

- All data types are “logical”
- Data type determines physical memory layout
- Physical memory layout unconcerned with value semantics

- In-memory metadata representation is implementation-defined; metadata serialization discussed later

# Main Array Layout Types

- Fixed-size Primitive
- Variable Binary
- Variable and Fixed-size List
- Struct
- Union (“Sparse” and “Dense” varieties)
- Null

# Validity Bitmap: representing nullness

- First buffer of every array (except Null layout)
- Packed bitmap, least-significant bit ordering
- Set bit for valid (not null), not set for null

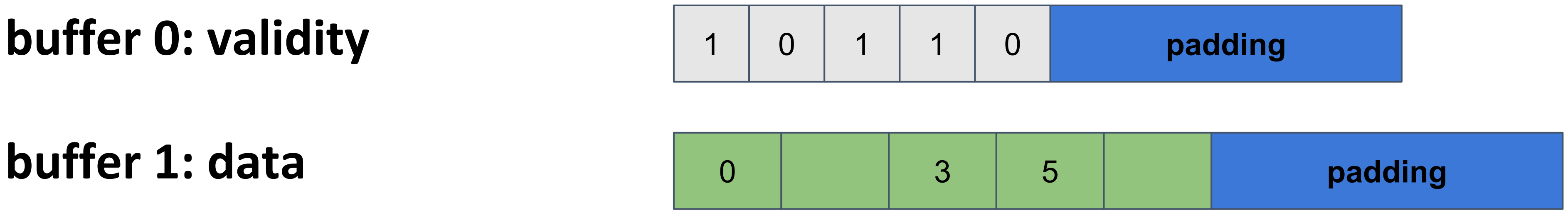
```
isvalid(i) = (validity[i / 8] >> (i % 8)) & 1
```

# Alignment and padding

- Recommend 8 or 64-byte aligned allocations and buffer padding
- **Note**: Alignment and padding in-memory implementation-dependent, but enforced when constructing protocol messages

# Fixed-Size Primitive Layout

`[0, null, 3, 5, null], type=Int32`




**Notes**

- Value bit/byte width determined by logical type
- Data in “null” slots does not have a specified value (implementations often zero this memory, though)
- Padding contents unspecified

# Variable Binary Layout

`[‘an’, null, ‘’, ‘apple’], type=String`

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| **buffer 0: validity** | 1 | 0 | 1 | 1 | | | |
| **buffer 1: offsets** | 0 | 2 | 2 | 2 | 7 | | |
| **buffer 2: data** | a | n | a | p | p | l | e |

**Notes**

- Offsets either signed int32 or int64
- Offsets in memory allowed to not start at 0, but generally are normalized to be 0-based when serializing

# Variable List Layout

[[0, 1], [], null, [5, null, 7]], type=List<Int8>

**buffer 0: validity**

| | | | |
|---|---|---|---|
| 1 | 1 | 0 | 1 |

**buffer 1: offsets**

| | | | | |
|---|---|---|---|---|
| 0 | 2 | 2 | 2 | 5 |

**child array 0:** [0, 1, 5, null 7], type=Int8

**Notes**

- Offsets either signed int32 or int64

# Fixed-Size List Layout

type=FixedSizeList<item_length=2, Int8>

[[0, 1], [2, 3], null, [6, 7]]

**buffer 0: validity**

| | | | |
|---|---|---|---|
| 1 | 1 | 0 | 1 |

**child array 0:** [0, 1, 2, 3, --, --, 6, 7], type=Int8

# Struct Layout

```
type: struct<a: int32, b: string>

[{a: 5, b: “foo”},
 {a: null, b: null},
 null,
 {a: -4, b: “”}]
```

**buffer 0: validity**

| | | | |
|---|---|---|---|
| 1 | 1 | 0 | 1 |

```
child 0 (“a”)    [5, null, null, -4]
child 1 (“b”)    [“foo”, null, null, “”]
```

**Notes**

- Null struct values need not necessarily be null in child arrays

# Sparse and Dense Union Layout

```
type: $UNION_TYPE<a: int32, b: string>
[5, “foo”, null, “bar”, “baz”]
```

**DENSE UNION**

| | | | | | |
|---|---|---|---|---|---|
| **buffer 0: validity** | 1 | 1 | 0 | 1 | 1 |
| **buffer 1: type id** | 0 | 1 | - | 1 | 1 |
| **buffer 2: offsets** | 0 | 0 | - | 1 | 2 |

```
child 0 (“a”) [5]
child 1 (“b”) [“foo”, “bar”, ‘baz’]
```

**SPARSE UNION**

| | | | | | |
|---|---|---|---|---|---|
| **buffer 0: validity** | 1 | 1 | 0 | 1 | 1 |
| **buffer 1: type id** | 0 | 1 | - | 1 | 1 |

```
child 0 (“a”) [5, -, -, -, -]
child 1 (“b”) [-, “foo”, -,
               “bar”, ‘baz’]
```

# Dictionary Encoding

```
type=dictionary<values=string, index=int8, ordered=false>
[“foo”, “bar”, null, “foo”]
```

**buffer 0: validity**

| | | | |
|---|---|---|---|
| 1 | 1 | 0 | 1 |

**buffer 1: indices**

| | | | |
|---|---|---|---|
| 0 | 1 | - | 0 |

**dictionary (type=string)** `[“foo”, “bar”]`

**Notes**

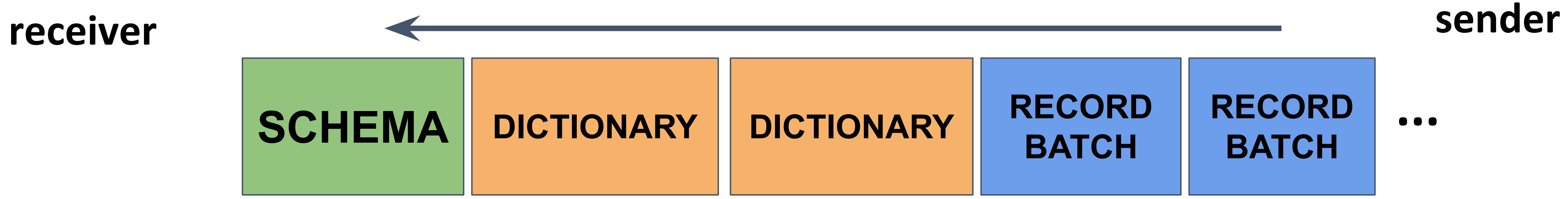
- The dictionary is data, not metadata
- Dictionary is allowed to have duplicates and nulls
- The “ordered” flag indicates that dictionary values’ order has semantic meaning

# Buffers for each layout

| **Layout** | Buffer 0 | Buffer 1 | Buffer 2 |
|---|---|---|---|
| Primitive | validity | data | |
| Dictionary-Encoded | validity | data (indices) | |
| Varbinary | validity | offsets | data |
| Variable List | validity | offsets | |
| Fixed Size List | validity | | |
| Struct | validity | | |
| Sparse Union | validity | type ids | |
| Dense Union | validity | type ids | offsets |
| Null | | | |

# Arrow Binary Protocol

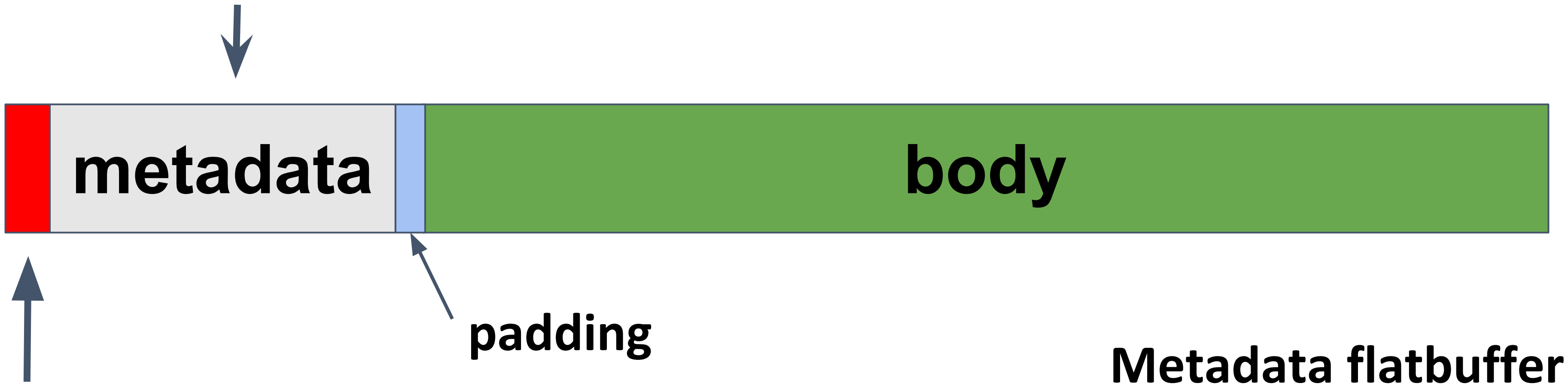
- **Record batch:** ordered collection of named arrays
- Streaming wire format for transferring record batch metadata and data between address spaces
- Often called “IPC protocol” since suitable for zero-copy shared memory interprocess communication (IPC)
- Sequence of “encapulated IPC messages”



# Encapsulated protocol ("IPC") messages

- Serialization wire format suitable for stream-based parsing

**"Message" Flatbuffer**
**(see format/Message.fbs)**




**padding**

**Metadata size or**
**end-of-stream marker**

**Metadata flatbuffer one of**

- **Schema**
- **RecordBatch**
- **DictionaryBatch**

# Metadata (schema) serialization

- Schema serialized using **Schema** Flatbuffer type defined in format/Schema.fbs
  - Contains logical type definitions (e.g. numbers, date/time types, etc.)
- Contains ordered list of Fields (names and types)
- **custom_metadata** on Schema and Field allows extensibility
- Has no “body” in the IPC message

- **“ARROW:”** is a reserved key prefix in custom_metadata. Used to implement extension types, etc.

# Record Batch serialization

```
schema {
  a: int32,
  b: list<item: binary>
}
```

- See **RecordBatch** Flatbuffer type in format/Message.fbs
- Depth-first pre-order flattened lists of Field data (lengths, null counts) and Buffer locations
- IPC message body contains buffers concatenated end-to-end
- Flatbuffer **Buffer** type records memory offset and size of each buffer, for later pointer arithmetic

| **BODY** |
|---|
| **a: buffer 0** |
| **a: buffer 1** |
| **b: buffer 0** |
| **b: buffer 1** |
| **b.item: buffer 0** |
| **b.item: buffer 1** |
| **b.item: buffer 2** |

# Dictionary Messages

- Dictionaries sent separate from Record Batches, to enable reuse across batches
- Dictionary-encoded fields assigned an “id” during schema serialization
- **DictionaryBatch** message contains **RecordBatch** with 1 field (the dictionary of interest)

```
schema {
  a: dictionary<string, int8, id=0>,
  b: dictionary<list<int32>, int32, id=1>
}
```

# Changing dictionaries

- DictionaryBatch “isDelta” flag allows dictionary appends
- Community discussing allowing dictionary replacements right now

# Extension Types

- Application-defined logical types can be defined and transmitted using special custom_metadata fields in the Schema
- Extension data stored using in a built-in logical type

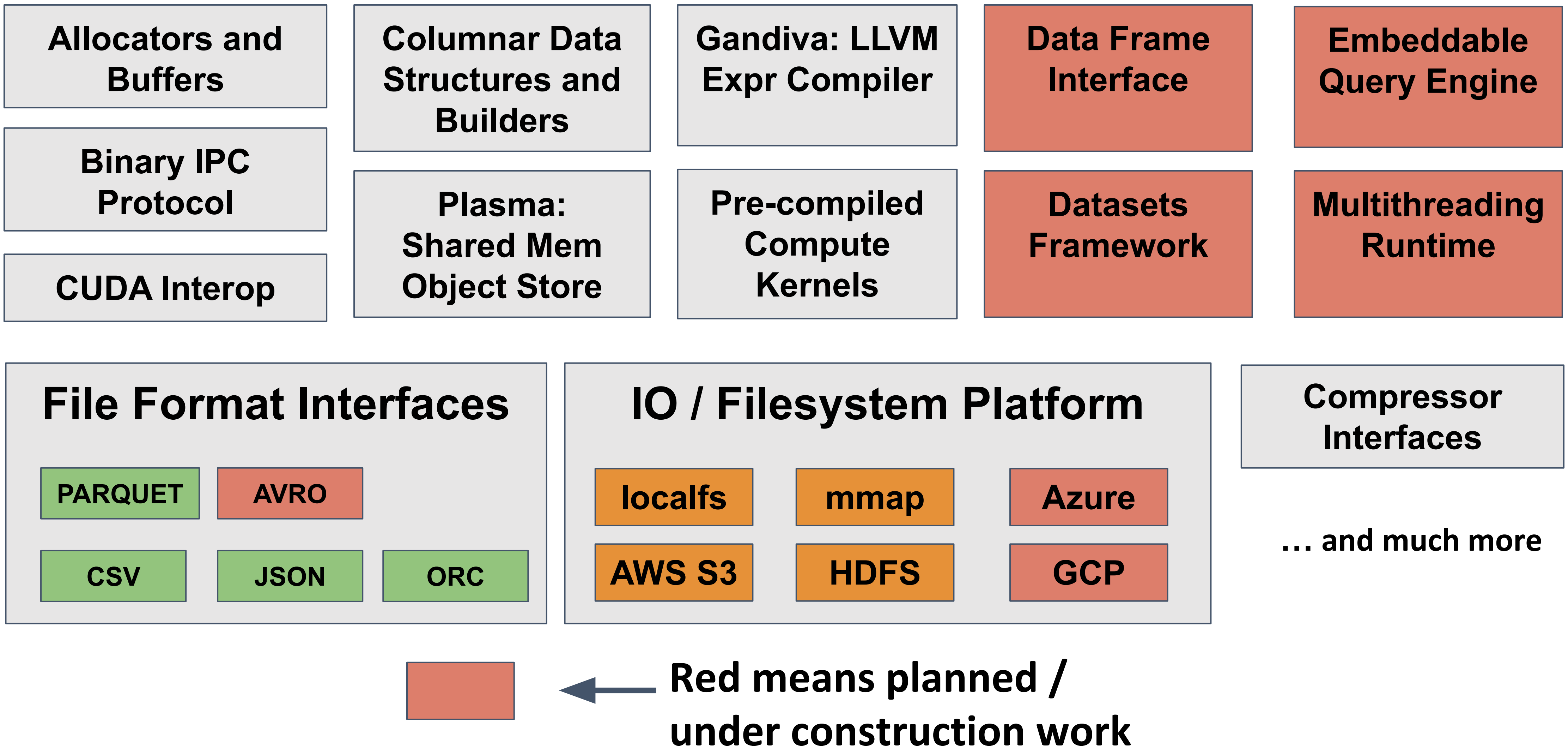
- Examples
  - **UUID** stored as **`FixedSizeBinary<16>`**
  - **LatitudeLongitude** stored as **`struct<x: double, y: double>`**

# Columnar Format Future Directions

- In-memory encoding, compression, sparseness
  - e.g. run-length encoding
  - See mailing list discussions, we need your feedback!
- Expansion of logical types

# Arrow C++ Libraries

# Arorw C++ development platform



# arrow::Buffer for memory references

- Virtual destructor allows for custom resource management
- Parent reference permits simple, copyless slicing

```cpp
class Buffer {
public:
 virtual ~Buffer();
 const uint8_t* data() const;
 int64_t size() const;
 bool is_mutable() const;
 uint8_t* mutable_data();
 std::shared_ptr<Buffer> parent() const;
};
```

# Explicit memory tracking

- All heap allocations use abstract `arrow::MemoryPool` interface
- We provide default memory pools using system allocator, jemalloc, or (coming soon) mimalloc
- MemoryPool tracks amount of “Arrow memory” that has been allocated, including peak memory use

# Common C++ data structure for all arrays

```
struct ArrayData {
  std::shared_ptr<DataType> type;
  int64_t length;
  int64_t null_count;
  int64_t offset;
  std::vector<std::shared_ptr<Buffer>> buffers;
  std::vector<std::shared_ptr<ArrayData>> child_data;
  std::shared_ptr<Array> dictionary;
}
```

# Type-specific `arrow::Array` subclasses

- Provide API conveniences for value access / unboxing
- `Array::Visit API` permits generic visitor dispatch
- `Array::Slice` allows zero-copy slicing

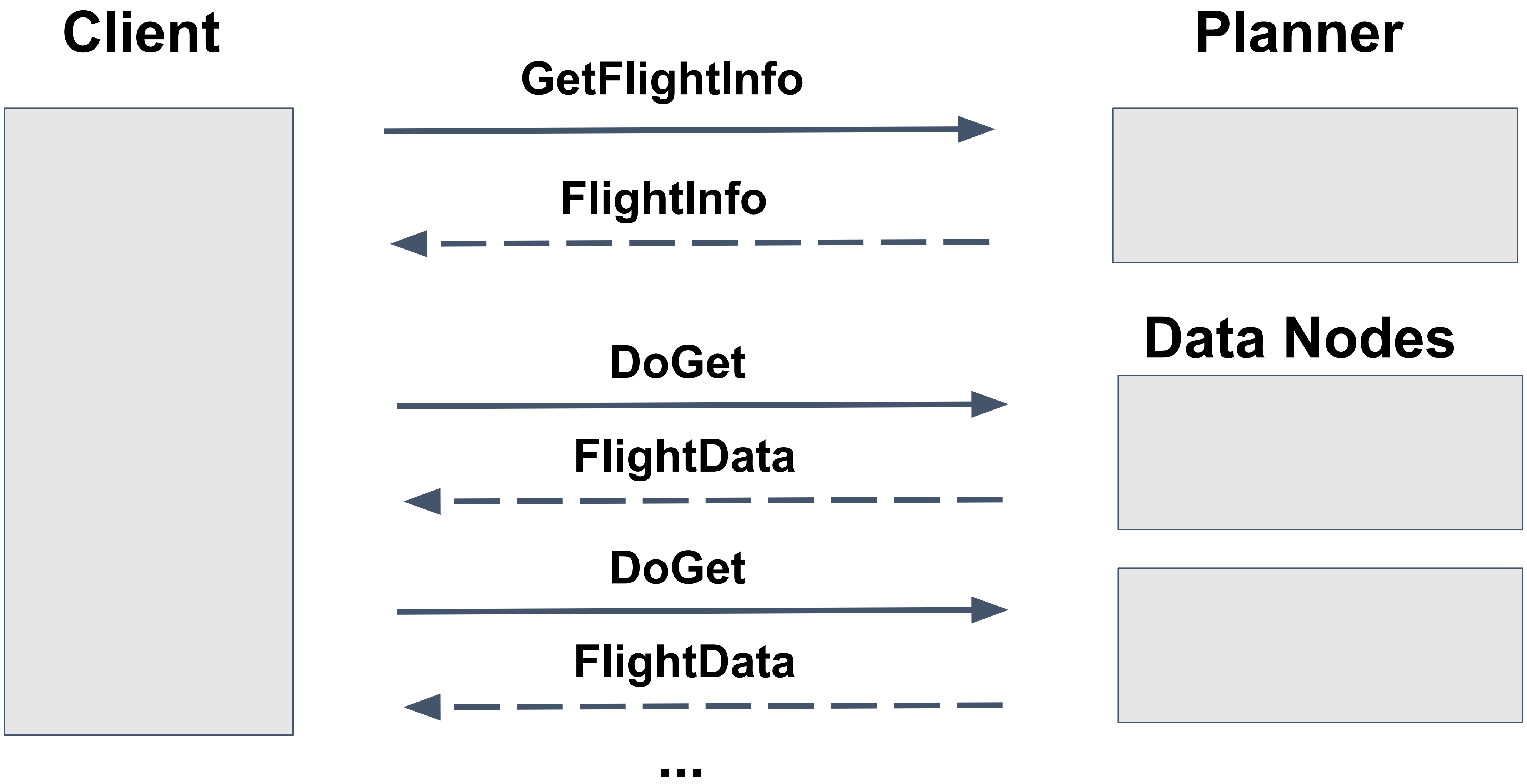
# Demo: reading/writing Arrow stream in Python

# Arrow Flight RPC Framework

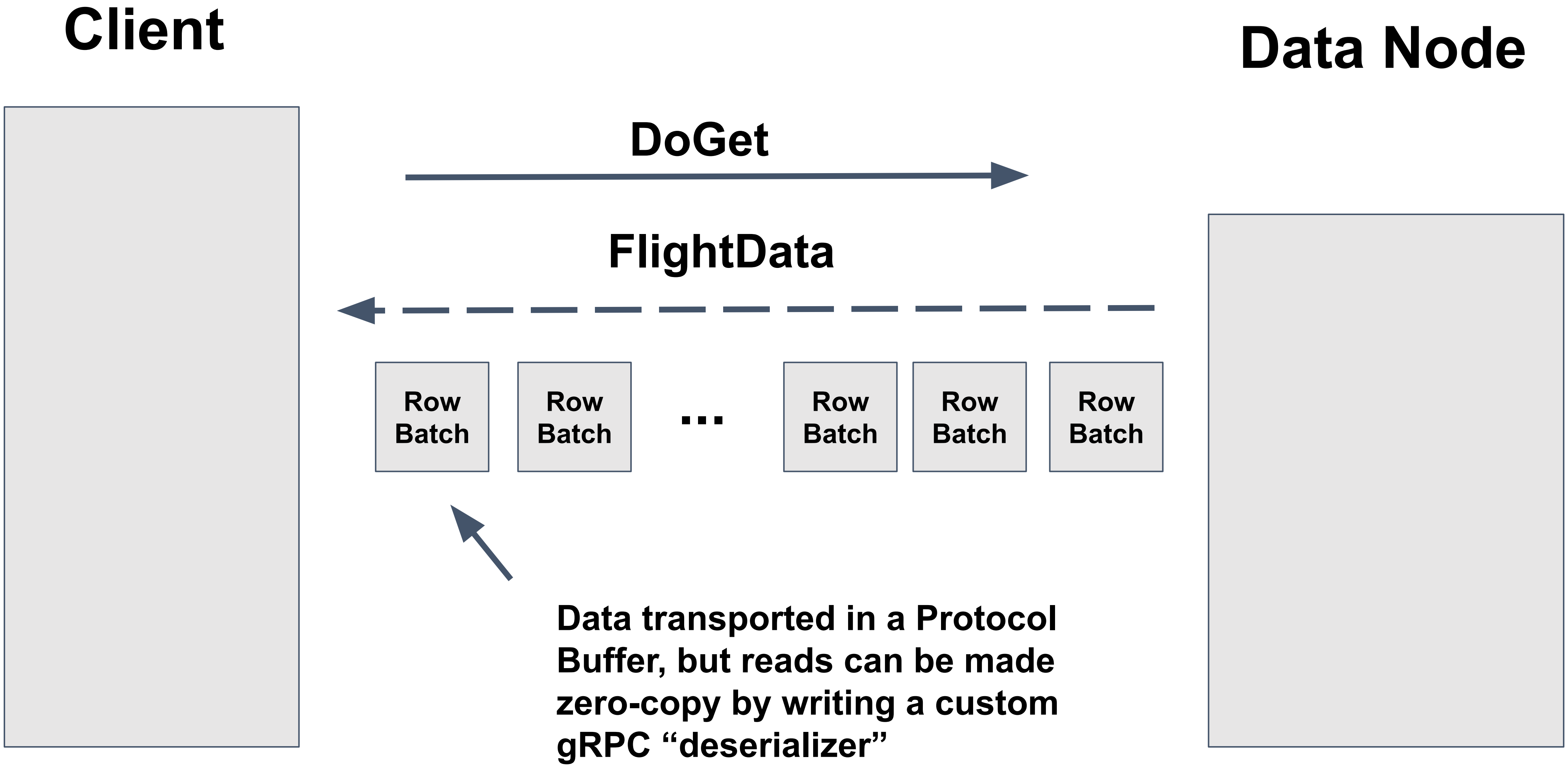
# Arrow Flight Overview

- A gRPC-based framework for defining custom data services that send and receive Arrow columnar data natively
- Uses Protocol Buffers v3 for client protocol
- Pluggable command execution layer, authentication
- Low-level gRPC optimizations
    - Write Arrow memory directly onto outgoing gRPC buffer
    - Avoid any copying or deserialization

# Arrow Flight - Parallel Get



# Arrow Flight - Efficient gRPC transport



# Flight: Static datasets and custom commands

- Support “named” datasets, and “command” datasets
- Commands are binary, and will be server-dependent
- Implement custom commands using general structured data serialization tools

**<u>Commands.proto</u>**

```
message SQLQuery {
  binary database_uri = 1;
  binary query = 2;
}
```

**GetFlightInfo RPC**

```
type: CMD
cmd: <serialized command>
```

# Flight: Custom actions

- Any request that is not a table pull or push, can be implemented as an “action”
- Actions return a stream of opaque binary data

# Demo: Build simple Flight service in Python

# Getting involved

- Join dev@arrow.apache.org
- PRs to https://github.com/apache/arrow