DORA DATA FORMAT

Typed messages using Apache Arrow

Philipp Oppermann
os2edu
2023-11-10

Agenda

- Motivation:
  - Passing data using shared memory → avoid copying data
  - Serializing typed data
  - Avoiding serialization overhead → use platform-independent data format (e.g. Cap'n Proto)
  - Simplify building and data processing → use Apache Arrow
- Arrow Data Format
  - Introduction
  - Data Representation
  - `RecordBatch` and IPC

Passing data in Dora

- Dora nodes run as separate processes
  - outputs need to cross process boundaries
  - pass data using inter process communication (IPC), e.g. TCP stream
- Output messages can be large, e.g. when sending image data
  - for good performance, we want to avoid copying the data if possible
  - TCP stream: cross-platform, but copies all data
  - → Use *shared memory* to send data

Zero Copy using Shared Memory

- Shared memory gives other processes direct access to some data
  - Without any copying
  - Possible on Linux, Windows, and macOS

Zero Copy using Shared Memory

- Shared memory gives other processes direct access to some data
  - Without any copying
  - Possible on Linux, Windows, and macOS
- Example:
  - Process A

```
let shmem = shared_memory::ShmemConf::new().size(4096).create()?;

// write some data
unsafe { *shmem.as_ptr() = 42 };

let id = shmem.get_os_id().to_owned();
send_id_to_proc_b(id)?; // e.g. through a TCP message
```

  - Process B

```
let id = receive_id_from_proc_a()?;
let shmem = shared_memory::ShmemConf::new().os_id(id).open()?;
let data = unsafe { *shmem.as_ptr() };
```

Dora: Passing data in shared memory

In sender:

1. Calculate required memory size
2. Allocate shared memory
3. Write data into shared memory
    - or construct it there directly
4. Send shared memory address + metadata to receiver
    - via other IPC method (e.g. TCP stream)

In receiver:

1. Map received shared memory region
2. Read and/or process data

Passing typed data

- Challenge: Shared memory is raw binary data → how can we send typed data?
- We need a way to represent higher-level data
    - For example, floating point values, arrays, or structs

Passing typed data

- Challenge: Shared memory is raw binary data → how can we send typed data?
- We need a way to represent higher-level data
  - For example, floating point values, arrays, or structs
- Just use native data representation? → not possible
  - Native representation varies across languages and architectures
  - Rust ABI is not stable yet → might change between Rust versions

Passing typed data

- Challenge: Shared memory is raw binary data → how can we send typed data?
- We need a way to represent higher-level data
  - For example, floating point values, arrays, or structs

- Just use native data representation? → not possible
  - Native representation varies across languages and architectures
  - Rust ABI is not stable yet → might change between Rust versions

→ use *serialization* to encode typed data to raw binary data

Serialization

- Serialization encodes data as raw bytes or as string
  - For example, JSON can be used as a serialization format

```json
{
    "name": "John Doe",
    "age": 43
}
```

- Receiver can *deserialize* the data again → to native representation, e.g. Rust struct or Python object

Serialization

- Serialization encodes data as raw bytes or as string
  - For example, JSON can be used as a serialization format

```
{
    "name": "John Doe",
    "age": 43
}
```

- Receiver can *deserialize* the data again → to native representation, e.g. Rust struct or Python object
- In Rust, the `serde` crate makes serialization and deserialization easy

```rust
#[derive(serde::Serialize, serde::Deserialize)]
struct Person {
    name: String,
    age: u32
}
```

```rust
let person = Person { name: "John Doe".to_string(), age: 43 };
let serialized_data: Vec<u8> = serde_json::to_vec(&person).unwrap();
```

Serialization: Drawbacks

- Requires additional encoding/decoding work
- Creates an additional copy of all the data

→ Can be costly for large messages (e.g. image data)

Serialization: Drawbacks

- Requires additional encoding/decoding work
- Creates an additional copy of all the data

→ Can be costly for large messages (e.g. image data)

Avoiding Serialization Overhead

- Use a serialization format that is close to native data representation
  - Examples: `protobuf` or Rust `bincode` crate
- Skip the serialization/deserialization completely by using custom, platform-independent format
  - Instead of native data format
  - Example: Cap’n Proto

The Cap’n Proto Format

- Schema files that describe message types

```
struct Person {
    name @0 :Text;
    email @1 :Text;
}
```

- Use `capnp` tool to compile schema into Rust/C++/Python interface
  - generates builder functions to create messages
  - generates accessor methods to read and write message data

The Cap’n Proto Format

- Schema files that describe message types

```
struct Person {
    name @0 :Text;
    email @1 :Text;
}
```

- Use `capnp` tool to compile schema into Rust/C++/Python interface
  - generates builder functions to create messages
  - generates accessor methods to read and write message data
- Accessing data is only possible through generated accessor methods
- Serialization/deserialization is a no-op
  - Data is already in a stable, platform-independent format

Cap’n Proto Example: Serialization

```
let mut message = ::capnp::message::Builder::new_default();
{
    let address_book = message.init_root::<address_book::Builder>();
    let mut people = address_book.init_people(1);
    {
        let mut alice = people.reborrow().get(0);
        alice.set_id(123);
        alice.set_name("Alice".into());
        alice.set_email("alice@example.com".into());
    }
}
capnp::serialize::write_message(&mut shared_memory, &message);
```

Challenges:

- Sender must use special `init` and `set` methods
- Does not work well with Rust's borrowing rules → additional `{}` scopes and `reborrow` calls are needed

Cap’n Proto Example: Deserialization

```rust
let message_reader = serialize::read_message(&mut shared_memory, ::capnp::message::ReaderOptions::new())?;

let address_book = message_reader.get_root::<address_book::Reader>()?;
for person in address_book.get_people()? {
    println!("{}: {}",
        person.get_name()?.to_str()?,
        person.get_email()?.to_str()?
    );
}
```

Challenges:

- Cap’n Proto is not self-describing → receiver needs to know message type to understand it
- Receiver must use special accessor methods to read data → this can make data processing difficult

Cap’n Proto Evaluation

Advantages:

- No serialization/deserialization cost
- No data copy required
- Works across languages

Drawbacks:

- Pre-compile step makes build process more complex
  - Especially for interpreted languages such as Python
- Cap’n Proto is not self-describing → receiver needs to know message type to understand it
- Receiver must use special accessor methods to read data → this can make data processing difficult

Cap’n Proto Evaluation

Advantages:

- No serialization/deserialization cost
- No data copy required
- Works across languages

Drawbacks:

- Pre-compile step makes build process more complex
  - Especially for interpreted languages such as Python
- Cap’n Proto is not self-describing → receiver needs to know message type to understand it
- Receiver must use special accessor methods to read data → this can make data processing difficult

⇒ use the **Apache Arrow** format to avoid the drawbacks

Apache Arrow

- Defines a cross-platform, cross-language data format (similar to Cap'n Proto)
  - Self-describing format → receiver can deduce message type from data
  - No schema files or pre-compilation necessary
  - Provides official bindings for various languages, including Rust and Python
- Data format designed with zero-copy in mind
  - Allows slicing and reordering data without copying
  - Python: zero-copy conversion to numpy and pandas → easier processing

```python
# numpy to arrow
data = numpy.arange(10, dtype='int16')
arr = pyarrow.array(data)

# arrow to numpy
arr = pyarrow.array([4, 5, 6], type=pyarrow.int32())
view = arr.to_numpy()
```

Basics of Arrow Format

- Base type: `Array`
  - specifies item type and length
- Array data is stored in one or multiple *buffers*
  - a buffer represents a memory region, e.g. on the heap or in shared memory
- Arrays can have children to create more complex types
  - Example: an array of `Vec<Vec<u8>>` can be represented by an offsets buffer and a child `u8` array
- Efficient representation of `null` entries
  - Arrays support a *validity bitmap* to indicate entries that are `null`

Arrow: Primitive Array Representation

Example: `Int32` array

```
[1, null, 2, 4, 8]
```

Representation:

```
- Length: 5, Null count: 1
- Validity bitmap buffer:

  | Byte 0 (validity bitmap) | Bytes 1-63            |
  |--------------------------|-----------------------|
  | 00011101                 | 0 (padding)           |

- Value Buffer:

  | Bytes 0-3   | Bytes 4-7   | Bytes 8-11  | Bytes 12-15 | Bytes 16-19 | Bytes 20-63           |
  |-------------|-------------|-------------|-------------|-------------|-----------------------|
  | 1           | unspecified | 2           | 4           | 8           | unspecified (padding) |
```

Arrow: Representing single primitives?

- Arrow is an array-based format → no special representation for single values
- → Store them as an array with length 1

Arrow: Representing single primitives?

- Arrow is an array-based format → no special representation for single values
- → Store them as an array with length 1

Convenience Functions in Dora

- We provide an `IntoArrow` trait in Dora → convert various Rust types to arrow array
  - e.g. `1i64.into_arrow()` creates an `Int64` arrow array with a single entry
  - e.g. `vec![0u8, 1, 2, 3].into_arrow()` creates an `UInt8` array with 4 entries
- There are matching `TryFrom` implementations for the receiver
  - e.g. `i64::try_from(&arrow_array)`

Arrow: String Arrays

Example: **Layout of** `[ 'foo', null, null, 'test' ]`

```
- Length: 4, Null count: 2
- Validity bitmap buffer:

  | Byte 0 (validity bitmap) | Bytes 1-63            |
  |--------------------------|-----------------------|
  | 00001001                 | 0 (padding)           |

- Offsets buffer:

  | Bytes 0-19      | Bytes 20-63           |
  |-----------------|-----------------------|
  | 0, 3, 3, 3, 7    | unspecified (padding) |

 - Value buffer:

  | Bytes 0-6       | Bytes 7-63            |
  |-----------------|-----------------------|
  | footest          | unspecified (padding) |
```

Reading the offsets buffer

- field i contains start offset
- field i+1 contains end offset
- length can be zero

Arrow: Struct Arrays

Example: **Layout of** `[ {'os', 1}, {'', 2}, {'edu', 4} ]`

- `Length: 3`
- `Buffers: []`
- `Children:`
  - `field-0 array`
    - `Length: 3`
    - `Offsets Buffer:` `0 | 2 | 2 | 5`
    - `Value Buffer:` `o | s | e | d | u`
  - `field-0 array`
    - `Length: 3`
    - `Value Buffer:` `1 | 2 | 4`

Struct Representation

- one child array for each field
- child arrays do not need to be adjacent in memory
  - allows creating smaller slices without copying

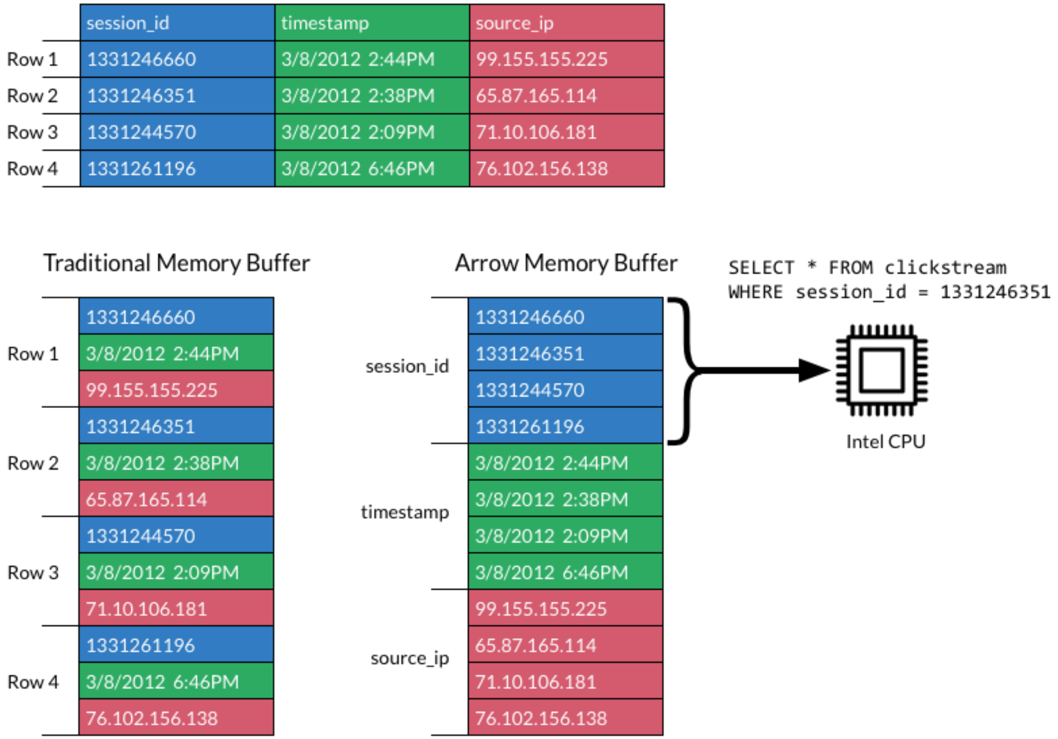
Reading the offsets buffer

- field `i` contains start offset
- field `i+1` contains end offset
- length can be zero

Arrow: Columnar Layout

- Columnar layout leads to better performance
  - iterating a single columns is fast, e.g. for a `find` operation
  - enables vectorization using SIMD operations
  - columns can be ignored without extra cost
- …, but it also makes mutation more expensive
  - adding a row might require reallocation if buffers are adjacent
  - data needs to be reordered on insert

→ most bindings treat Arrow arrays as immutable
→ copy data when it's modified



Arrow: `RecordBatch` and IPC

- A `RecordBatch` is a dataset of multiple arrays that have the same length
  - Example: array of data plus array of corresponding timestamps
  - used for data serialization, e.g. writing an event log to a file
- `RecordBatch` and `StructArray` are similar
  - both are a collection of fields/arrays with same length
  - there are `From` implementations to convert between the two types
  - differences
    - `StructArray` can be nested and its fields can be `null`
    - `RecordBatch` can additional top-level metadata and schema information
- The `arrow-ipc` crate can write a `RecordBatch` into file (or other writer)
  - using `arrow_ipc::writer::FileWriter` (or `StreamWriter`)
  - useful for passing arrow data in shared memory

Arrow: RecordBatch Example

```
// define schema
let schema = Schema::new(vec![
    Field::new("id", DataType::Int32, false),
    Field::new("nested", DataType::Struct(Fields::from(vec![
        Field::new("a", DataType::Utf8, false),
        Field::new("b", DataType::Float64, false),
        Field::new("c", DataType::Float64, false),
    ])), false, ),
]);

// create some data
let id = Int32Array::from(vec![1, 2, 3, 4, 5]);
let nested = StructArray::from(vec![
    (Arc::new(Field::new("a", DataType::Utf8, false)),
        Arc::new(StringArray::from(vec!["a", "b", "c", "d", "e"])) as Arc<dyn Array>,),
    (Arc::new(Field::new("b", DataType::Float64, false)),
        Arc::new(Float64Array::from(vec![1.1, 2.2, 3.3, 4.4, 5.5])),),
    (Arc::new(Field::new("c", DataType::Float64, false)),
        Arc::new(Float64Array::from(vec![2.2, 3.3, 4.4, 5.5, 6.6])),),
]);
// build a record batch
let batch = RecordBatch::try_new(Arc::new(schema), vec![Arc::new(id), Arc::new(nested)])?;
```

Full example: See

`arrow/examples/dynamic_types.rs`

in github.com/apache/arrow-rs

```
+----+-------------------------+
| id | nested                  |
+----+-------------------------+
| 1  | {a: a, b: 1.1, c: 2.2} |
| 2  | {a: b, b: 2.2, c: 3.3} |
| 3  | {a: c, b: 3.3, c: 4.4} |
| 4  | {a: d, b: 4.4, c: 5.5} |
| 5  | {a: e, b: 5.5, c: 6.6} |
+----+-------------------------+
```

Summary

- Shared memory enables zero-copy message passing
- Arrow defines a self-describing, platform-independent data format
    - no serialization necessary
    - no precompilation necessary
    - makes zero-copy processing easier (e.g. through numpy/pandas conversions)
- Arrow uses arrays as base type
    - Columnar Layout
    - Arrays are backed by (multiple) buffers
    - Complex nested types are possible

→ Arrow data format is a fit for Dora