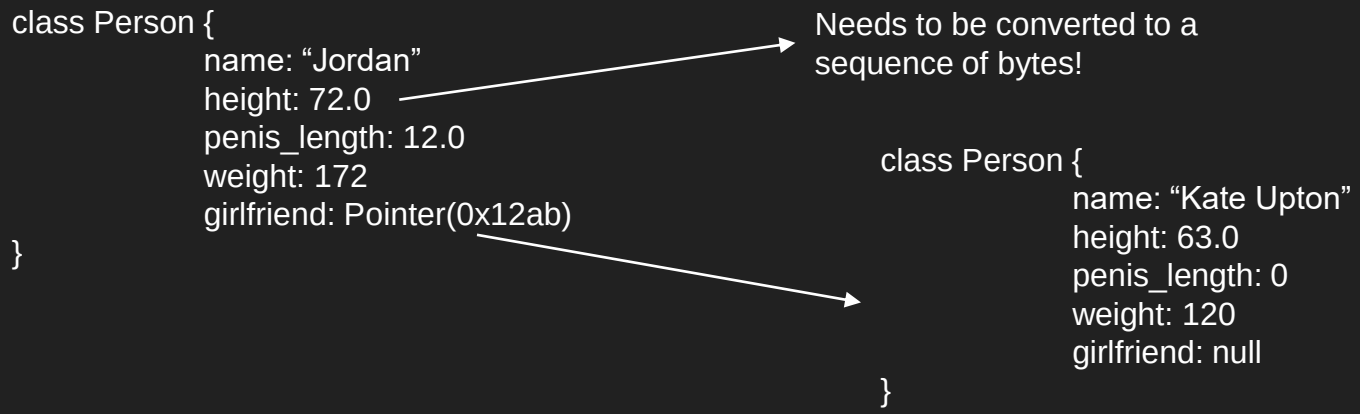
Data Serialization

Background

In most distributed systems, we need a way of turning in-memory data into byte sequences for transport. This could be either to files on disk, or over the network to other computers. However, as we will see, some formats for storing data in a transferable format are not only more efficient, but more maintainable for long term development and data schema evolution, than others.



Naive Approach to serialization

Language Specific Serialization Frameworks:

- Pickle for Python
- Marshal for Ruby
- Serializable for Java

While these libraries are convenient, they lock you into using a single language, and additionally, they often will deserialize the data into arbitrary classes which can lead to security vulnerabilities! Additionally, they do not care about versioning data, and are relatively bad performance wise.

Standardized Encodings

Extremely popular, formats like JSON/XML:

- Useful because everyone is familiar with them, hence why they are great for communication between different organizations
- Issues determining between numbers and strings (XML) and integers and floats (JSON)
- No support for binary strings (only unicode ones)
- Not particularly compact, thus introducing extra network load and leaving room for performance improvements (field names need to be sent unless there is a schema involved)
    - Binary encoding can improve these issues but still suffer from above problem regarding field names

Thrift and Protocol Buffers

- Binary encoding libraries that use a schema to greatly reduce the size of serialized messages
  - Once a schema is provided these libraries can generate classes representing those datatypes in most popular languages
  - Since each field has a numbered tag we can greatly decrease the amount of space required for encoding, as we do not need a full string to represent the field name

```
struct Person {
  1: required string       userName,
  2: optional i64          favoriteNumber,
  3: optional list<string> interests
}
```

Thrift

```
message Person {
    required string user_name       = 1;
    optional int64  favorite_number = 2;
    repeated string interests       = 3;
}
```

Protocol Buffers

Schema Evolution

Important to ensure that the schemas that we set are both forwards and backwards compatible. Rolling updates, as well as slow to update clients are always a possibility!

Backwards compatibility: Newer code can read data written by older code.

Forwards compatibility: Older code can read data written by newer code.

Schema Evolution

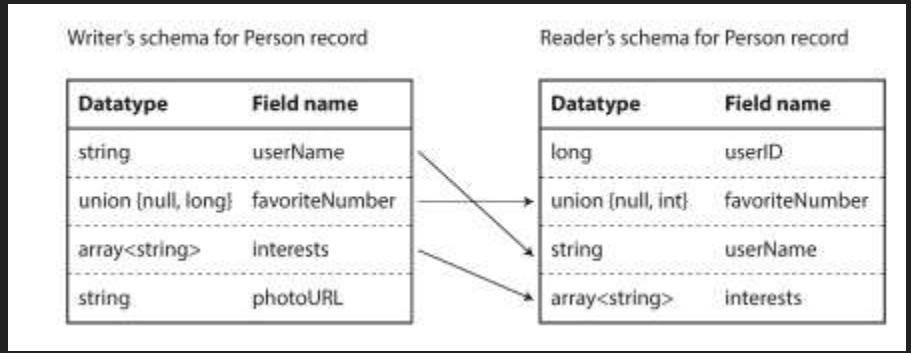
- Cannot change existing field tags as it would make all existing encodings invalid
  - But you can add new fields with unique tag numbers (cannot be set as required)
    - Ensures forwards compatibility, as old code can ignore new fields
    - Ensures backwards compatibility, as new code still knows how to read all fields contained in old serializations (hence new field cannot be required, old code won’t contain it)
- Depending on situation may be able to change data type of field

Avro

- Similar to ProtoBuf and Thrift, but created to better suit use cases for Hadoop (dumping lots of possibly unstructured data files)
- Again declare a schema, but no field tags this time
  - Use the schema itself to decode the data, must be in the same order as the fields
  - However, the schema can still evolve!

Writer vs. Reader Schema

- Fields matched up by field name in writer and reader's schema
- If fields present in writer schema and not in reader's schema they are ignored
- If fields present in reader schema and not in writer's schema we use the default value for them



This means that fields can only be added or removed if they have a default value!

Optimizing Network Bandwidth

So far, we have implied that in Avro every single record needs to also include the writer's schema, which would use a ton of unnecessary network I/O.

Remember that Avro is most useful for Hadoop:

- If we have a large file with many records encoded the same way, only need to send the writer's schema once
- If we have records written many different ways, include a version number with each record that corresponds to one writer's schema

Why Field Tags are Hard

If we want to export all rows from a database to something like HDFS, we want to be able to fully automate this process. With Thrift or ProtoBuf, we would have to have somebody manually convert the database schema to a serialization schema to follow the rules of only increasing tag numbers (the database is not aware of this). With Avro, we can more easily automate the schema generation process because we can just create it from the existing columns of the database.

Very useful for ETL processes.

Schema Evolution in Databases

Data outlives code:

- Database holds data written by multiple different schemas
- It is upto the serialization framework to be able to rectify differences in data written at different times
  - Make sure that processes that are reading data encoded in an older format do not crash
  - Make sure that processes with an older version of the schema do not lose columns if performing a read-modify-update cycle

Conclusion

Ultimately, these text serialization libraries are extremely useful for providing some format to data that may otherwise be unstructured. Not only do they further compress byte sequences that need to be transferred, keeping a record/database of the schemas over time acts as an effective method of documenting the representation of the data through the progression of the application. Finally, in statically typed applications, the ability to generate code/classes to interact with is extremely useful.