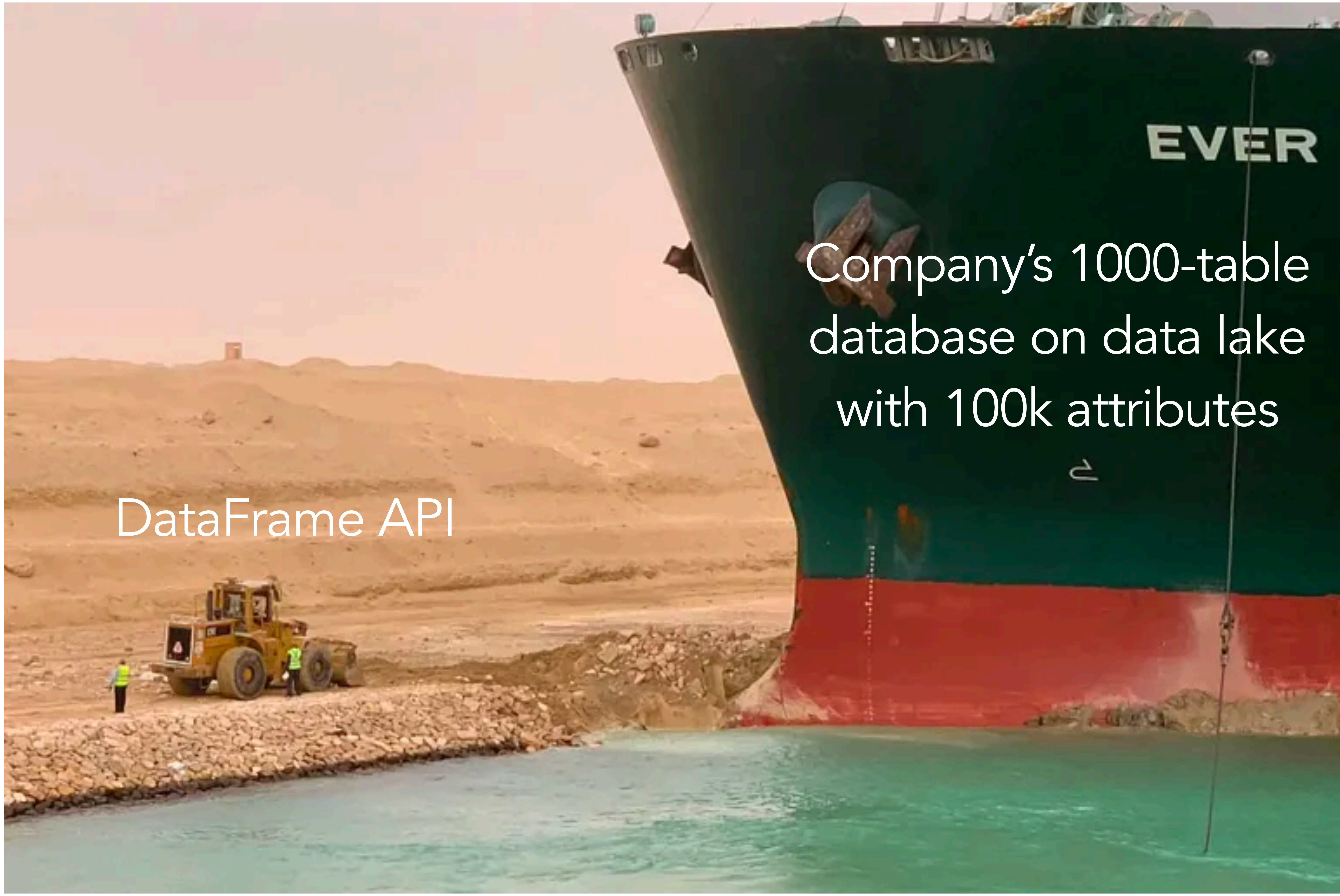
DSC 204a Scalable Data Systems

- Haojian Jin


Company's 1000-table database on data lake with 100k attributes
DataFrame API
EVER

Meme idea credit: https://datasystemsfun.tumblr.com/

Where are we in the class?

Foundations of Data Systems (2 weeks)

- Digital representation of Data → Computer Organization → Memory hierarchy → Process → Storage

Scaling Distributed Systems (3 weeks)

- Cloud → Network → Distributed storage → Parallelism → Partition and replication

Data Processing and Programming model (5 weeks)

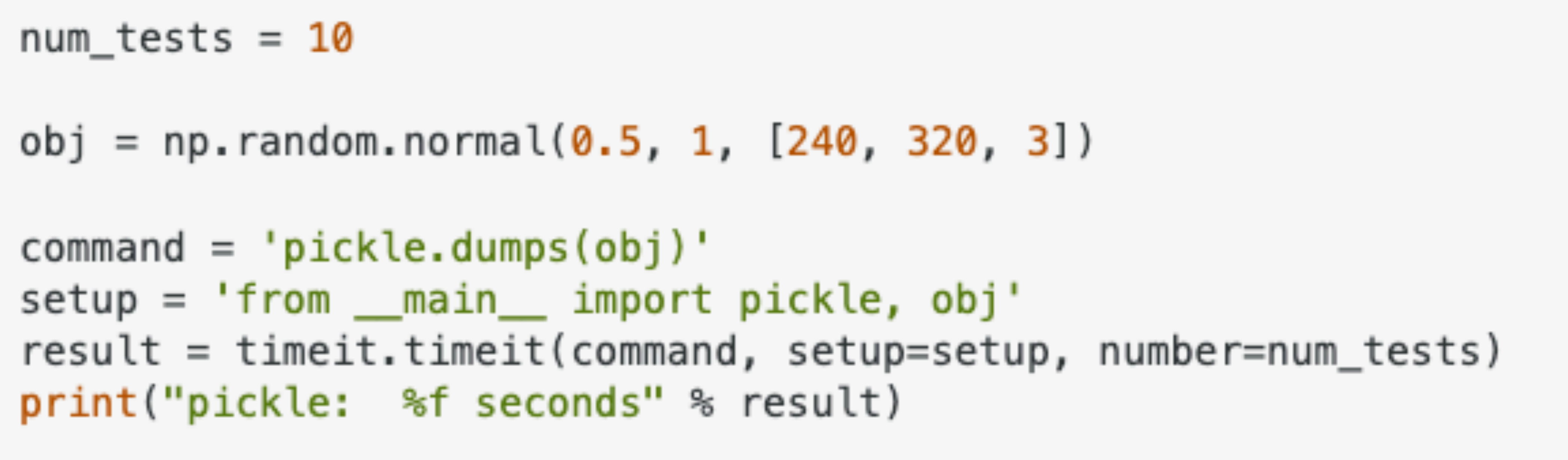
- Data Models evolution → **Data encoding evolution** → IO & Unix Pipes → Batch processing (MapReduce) → Stream processing (Spark)

Today’s topic: Data Encoding

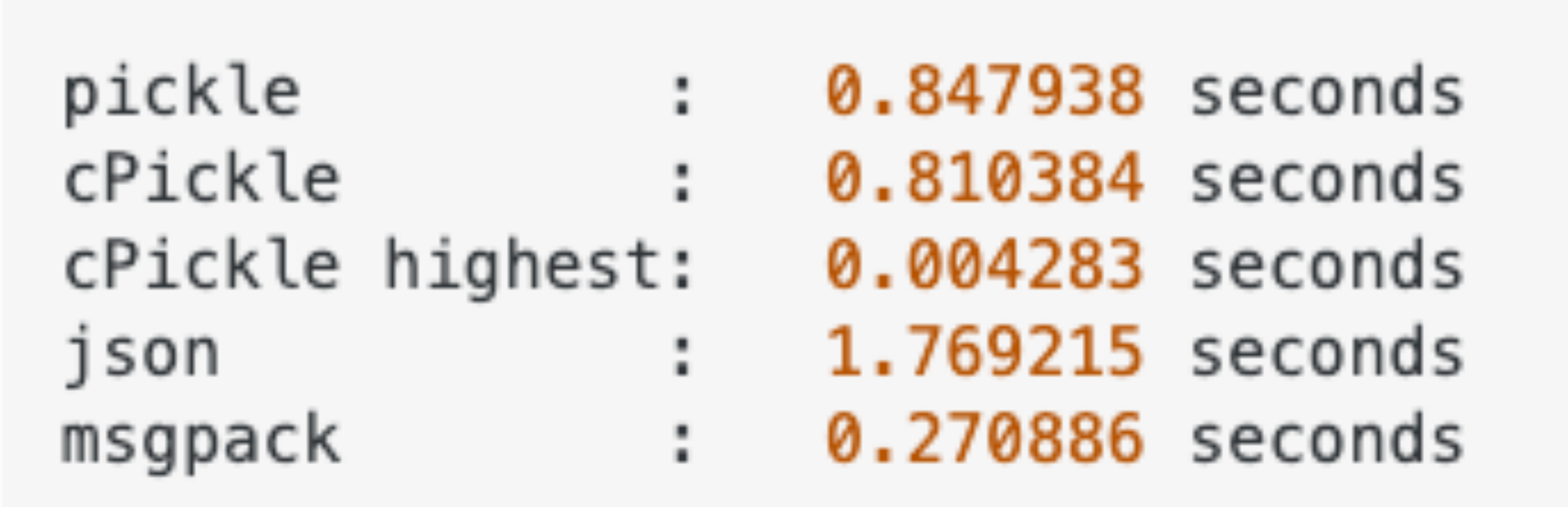
- Formats for Encoding Data
  - Language-Specific Formats
  - JSON, XML, and Binary Variants
  - BINARY ENCODING
- Modes of dataflow
  - Database
  - REST
  - RPC
  - GraphQL
- Summary

Why encoding?

- Data in memory
  - e.g., objects, structs, lists, arrays, hash tables, trees
  - Efficient access and manipulation by the CPU (typically using pointers)
  - Why pointers? => Random address access
- Data in storage or network
  - No pointers.
  - Self-contained sequence of bytes.



```python
num_tests = 10

obj = np.random.normal(0.5, 1, [240, 320, 3])

command = 'pickle.dumps(obj)'
setup = 'from __main__ import pickle, obj'
result = timeit.timeit(command, setup=setup, number=num_tests)
print("pickle:  %f seconds" % result)
```



```
pickle         :   0.847938 seconds
cPickle        :   0.810384 seconds
cPickle highest:   0.004283 seconds
json           :   1.769215 seconds
msgpack        :   0.270886 seconds
```

https://stackoverflow.com/questions/2259270/pickle-or-json

Language-Specific Formats

- Java: java.io.Serializable;
- Python has pickle;
- Pros:
  - Convenient: in-memory objects to be saved and restored.
- Cons:
  - Tied to a programming language.
  - Decoding may lead to over-privileged behaviors.
    - e.g., remote execution.
  - Versioning, forward and backward compatibility
  - Efficiency
- Summary: quick, dirty, small individual projects

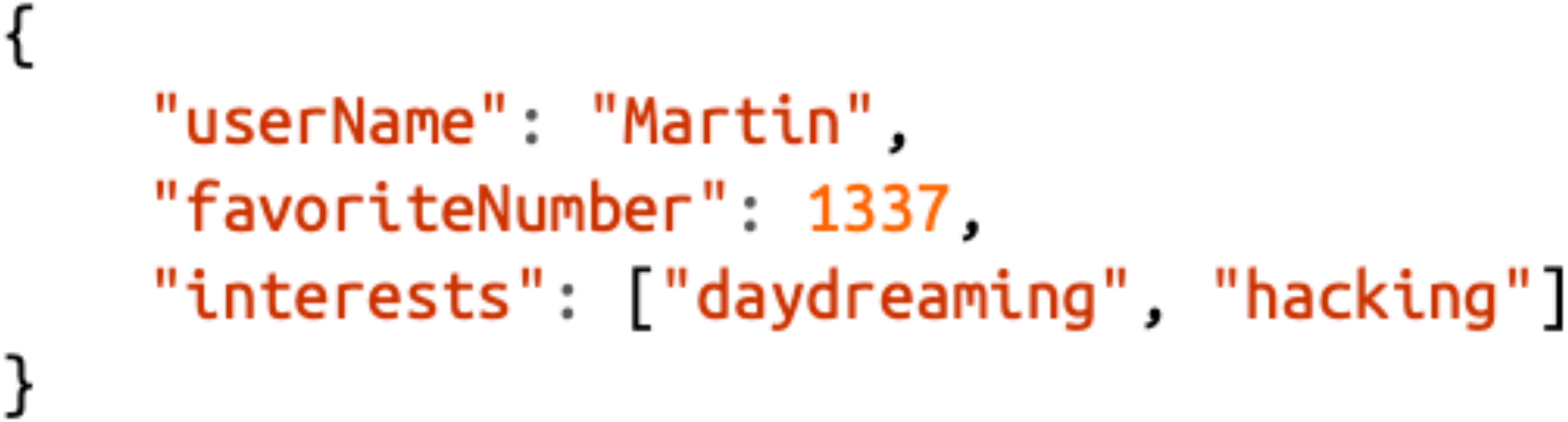
JSON, XML, CSV

- Python: Json dump.
- JSON, XML, CSV: human-readable but verbose.
- JSON: browser friendly and simple
- Common cons:
  - too verbose and unnecessarily complicated
  - ambiguity around the encoding of numbers
    - XML and CSV don’t distinguish a number and a string that happens to consist of digits
    - JSON doesn’t distinguish integers and floating-point numbers, and it doesn’t specify a precision.

JSON, XML, CSV

- Python: Json dump.
- Common cons:
  - JSON and XML have good support for Unicode character strings.
  - There is optional schema support for both XML and JSON
  - CSV does not have any schema,

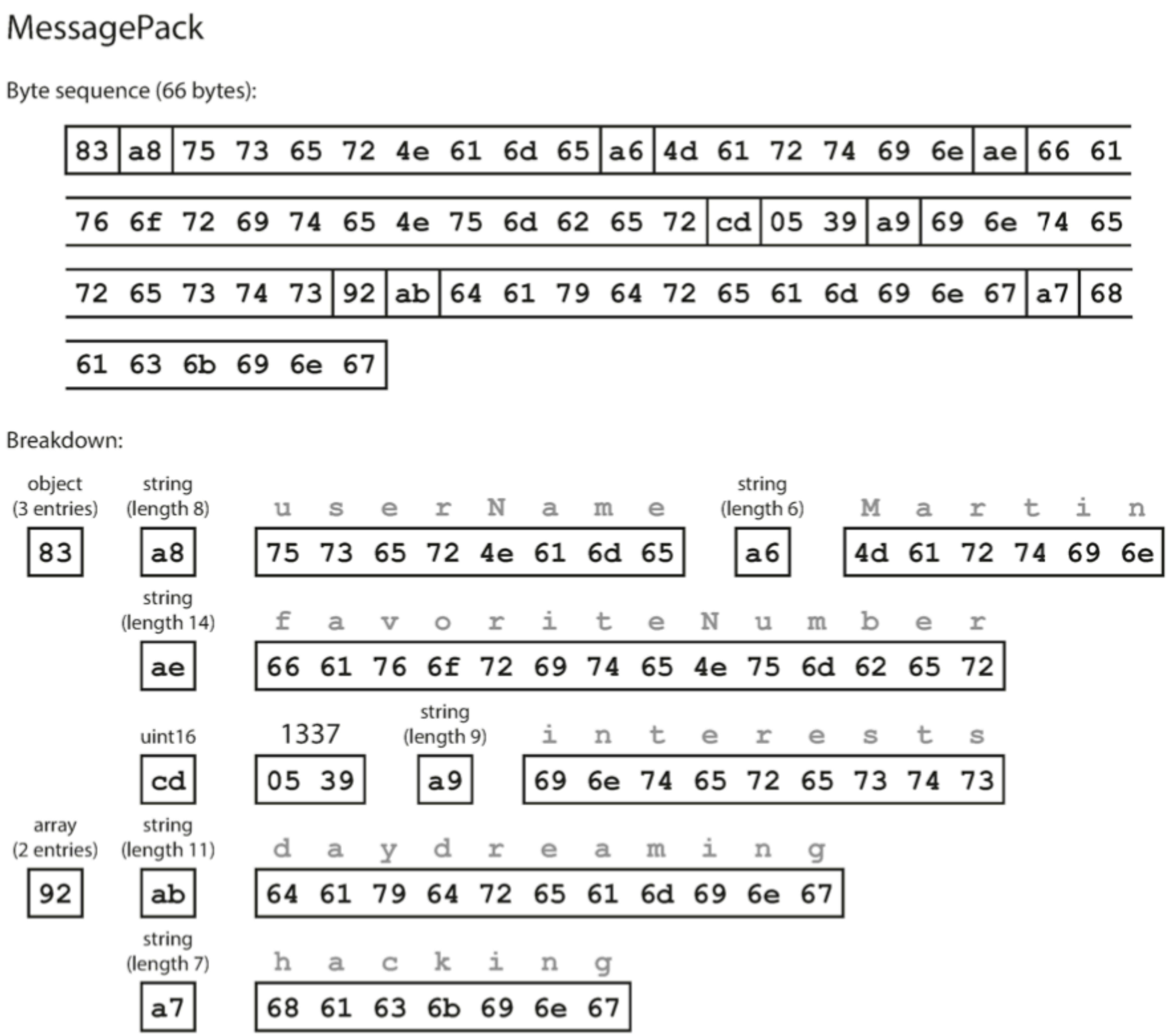
Example



```
{
    "userName": "Martin",
    "favoriteNumber": 1337,
    "interests": ["daydreaming", "hacking"]
}
```

MessagePack

```
{
    "userName": "Martin",
    "favoriteNumber": 1337,
    "interests": ["daydreaming", "hacking"]
}
```



MessagePack

Byte sequence (66 bytes):

```
83 a8 75 73 65 72 4e 61 6d 65 a6 4d 61 72 74 69 6e ae 66 61
76 6f 72 69 74 65 4e 75 6d 62 65 72 cd 05 39 a9 69 6e 74 65
72 65 73 74 73 92 ab 64 61 79 64 72 65 61 6d 69 6e 67 a7 68
61 63 6b 69 6e 67
```

Breakdown:

object (3 entries) 83

string (length 8) a8 — u s e r N a m e — 75 73 65 72 4e 61 6d 65

string (length 6) a6 — M a r t i n — 4d 61 72 74 69 6e

string (length 14) ae — f a v o r i t e N u m b e r — 66 61 76 6f 72 69 74 65 4e 75 6d 62 65 72

uint16 cd — 1337 — 05 39

string (length 9) a9 — i n t e r e s t s — 69 6e 74 65 72 65 73 74 73

array (2 entries) 92

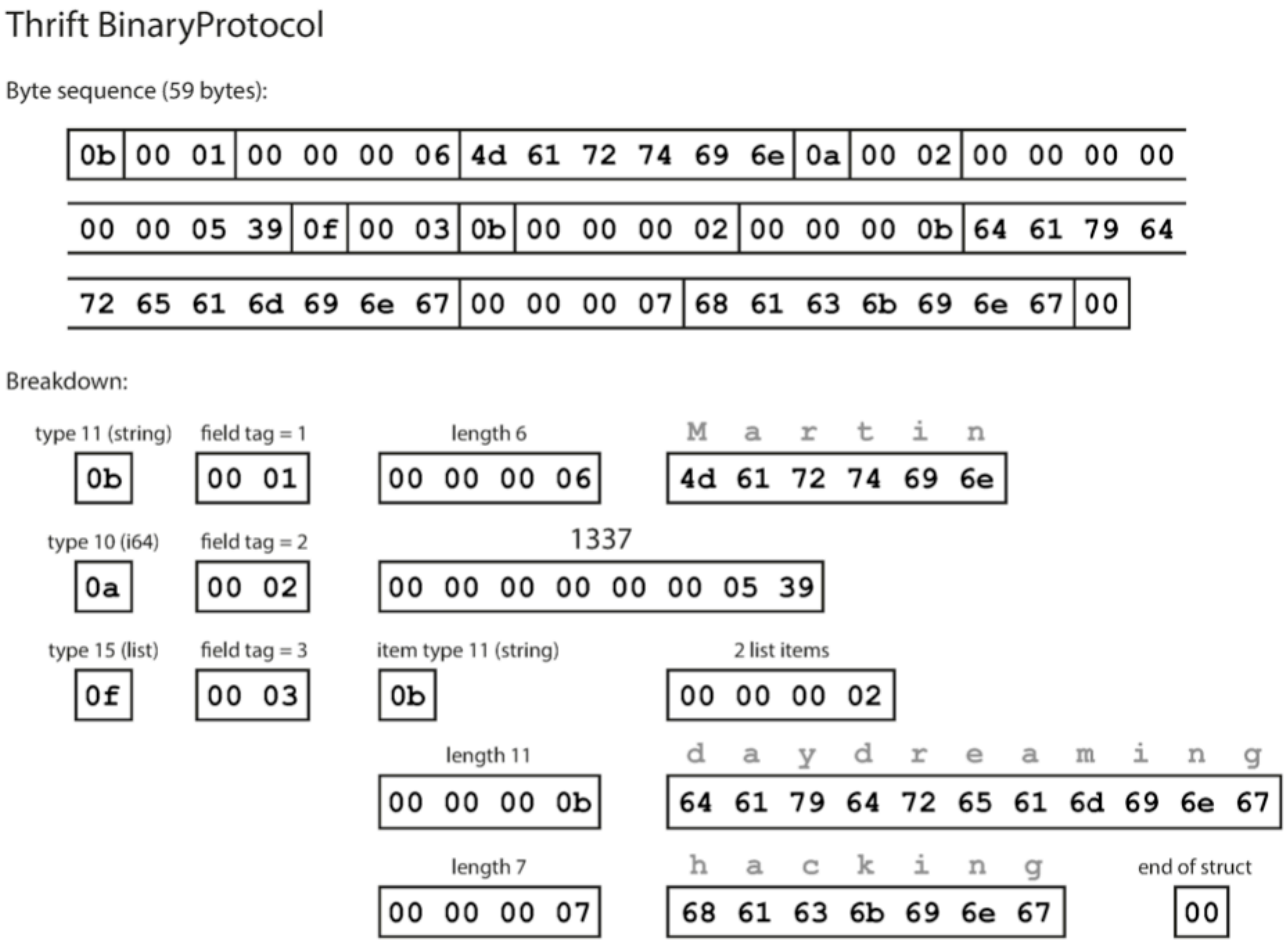
string (length 11) ab — d a y d r e a m i n g — 64 61 79 64 72 65 61 6d 69 6e 67

string (length 7) a7 — h a c k i n g — 68 61 63 6b 69 6e 67

Thrift BinaryProtocol

```
{
    "userName": "Martin",
    "favoriteNumber": 1337,
    "interests": ["daydreaming", "hacking"]
}
```

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



Thrift BinaryProtocol

Byte sequence (59 bytes):

```
0b|00 01|00 00 00 06|4d 61 72 74 69 6e|0a|00 02|00 00 00 00
00 00 05 39|0f|00 03|0b|00 00 00 02|00 00 00 0b|64 61 79 64
72 65 61 6d 69 6e 67|00 00 00 07|68 61 63 6b 69 6e 67|00
```

Breakdown:

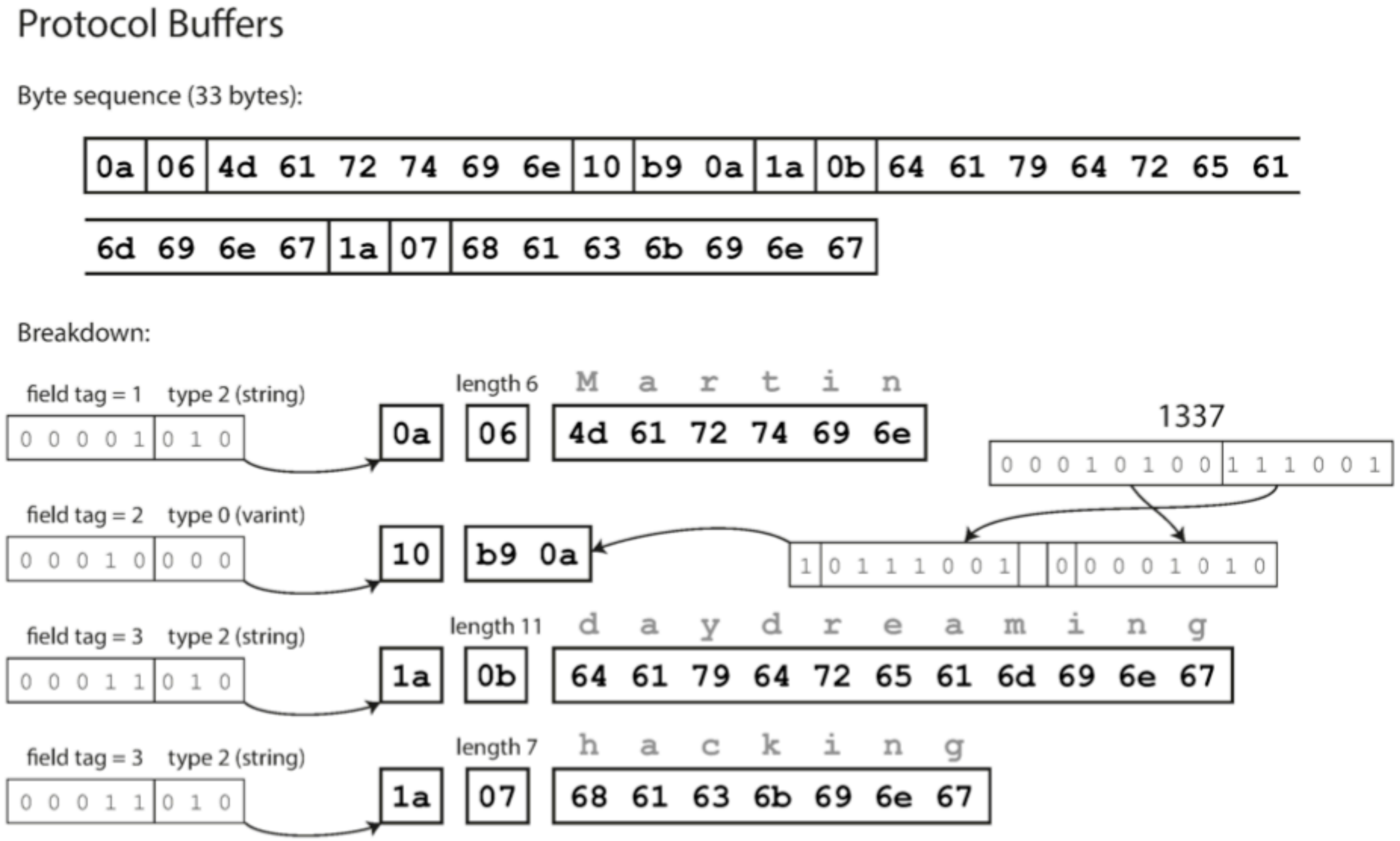
```
type 11 (string)  field tag = 1   length 6              M  a  r  t  i  n
0b                00 01           00 00 00 06           4d 61 72 74 69 6e

type 10 (i64)     field tag = 2                 1337
0a                00 02           00 00 00 00 00 00 05 39

type 15 (list)    field tag = 3   item type 11 (string)  2 list items
0f                00 03           0b                     00 00 00 02

                                  length 11             d  a  y  d  r  e  a  m  i  n  g
                                  00 00 00 0b           64 61 79 64 72 65 61 6d 69 6e 67

                                  length 7              h  a  c  k  i  n  g           end of struct
                                  00 00 00 07           68 61 63 6b 69 6e 67          00
```

Thrift Binary Protocol v.s. MessagePack

- Same:
    - each field has a type annotation
    - a length indication
    - strings also encoded as ASCII
- Diff:
    - there are no field names
    - Instead, contains field tags
    - 59 bytes vs. 81 bytes

More system performance

- Protobuff, Thrift CompactProtocol
- Key ideas:
  - Packing the field type and tag number into a single byte
  - Using variable-length integer.



Protocol Buffers

Byte sequence (33 bytes):

```
0a 06 4d 61 72 74 69 6e 10 b9 0a 1a 0b 64 61 79 64 72 65 61
6d 69 6e 67 1a 07 68 61 63 6b 69 6e 67
```

Breakdown:

```
field tag = 1  type 2 (string)            length 6  M  a  r  t  i  n
0 0 0 0 1|0 1 0            0a   06        4d 61 72 74 69 6e

field tag = 2  type 0 (varint)                                  1337
0 0 0 1 0|0 0 0            10   b9 0a     0 0 0 1 0 1 0 0|1 1 1 0 0 1
                                          1|0 1 1 1 0 0 1   0|0 0 0 1 0 1 0

field tag = 3  type 2 (string)            length 11  d  a  y  d  r  e  a  m  i  n  g
0 0 0 1 1|0 1 0            1a   0b        64 61 79 64 72 65 61 6d 69 6e 67

field tag = 3  type 2 (string)            length 7  h  a  c  k  i  n  g
0 0 0 1 1|0 1 0            1a   07        68 61 63 6b 69 6e 67
```

Schema evolution:

- Field tags
  - to maintain backward compatibility, every field you add after the initial deployment of the schema must be optional or have a default value.
  - only remove a field that is optional (a required field can never be removed)
  - never use the same tag number again
- Data types:
  - Possible but huge cost. May lose precision or get truncated.
  - Many language specific tricks.

The Merits of Schemas

- more compact than the “binary JSON” variants => omit field names.
- The schema & documentation.
- Hard to manually maintained.
- Keeping a database of schemas allows you to check forward and backward compatibility of schema changes.
- Code generation.

Today's topic: Data Encoding

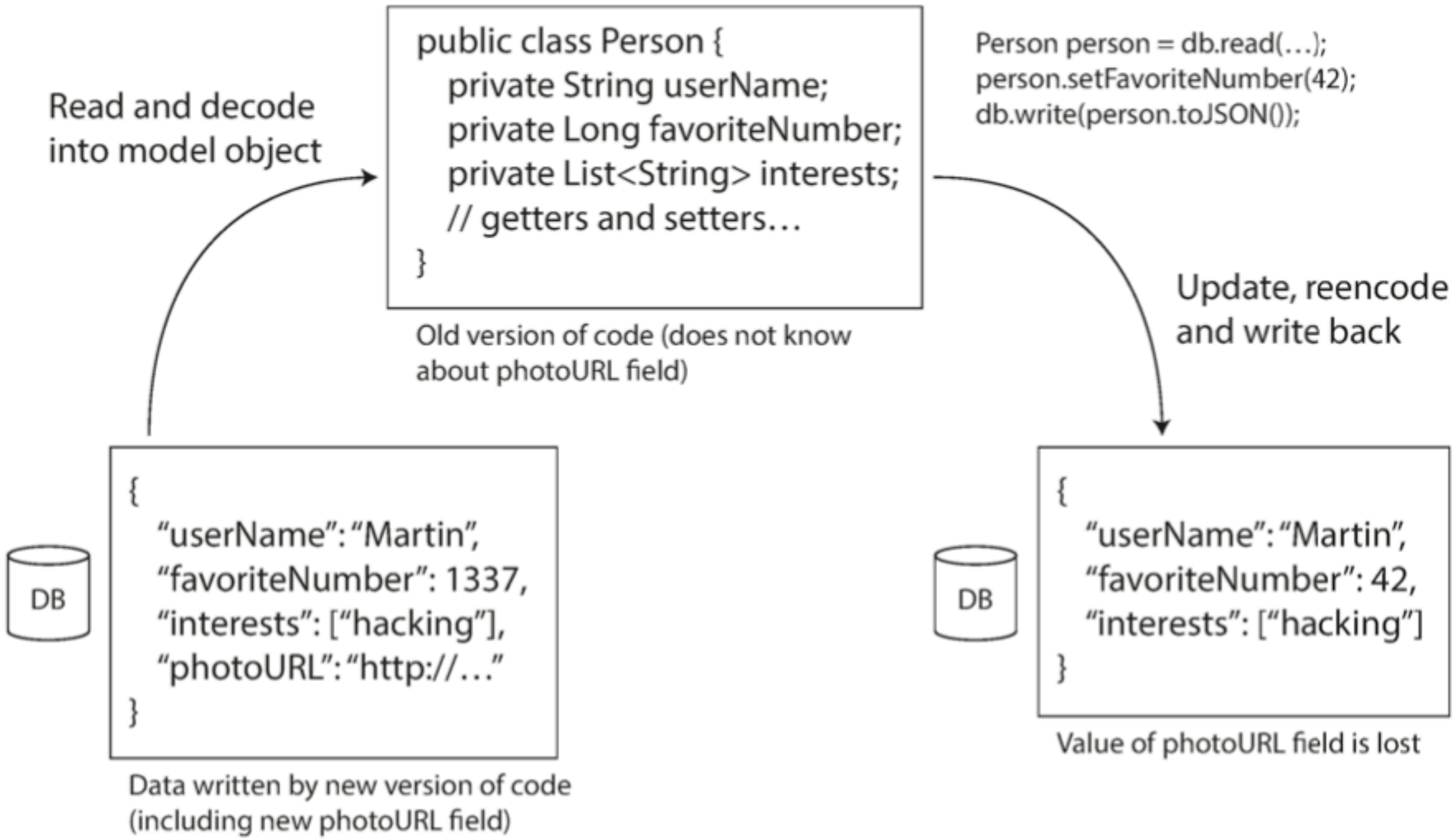
- Formats for Encoding Data
  - Language-Specific Formats
  - JSON, XML, and Binary Variants
  - BINARY ENCODING
- Modes of dataflow
  - Database
  - REST, RPC, GraphQL
  - Message-passing
- Summary

Modes of dataflow

- Dataflow:
  - Encoding + Sharing (flowing) + Decoding
- Via databases (see “Dataflow Through Databases”)
- Via service calls (see “Dataflow Through Services: REST and RPC”)
- Via asynchronous message passing (see “Message-Passing Dataflow”)

Dataflow Through Databases

- The write process encodes data
- The read process decodes data
- Can be the same process.



Read and decode into model object

```
public class Person {
   private String userName;
   private Long favoriteNumber;
   private List<String> interests;
   // getters and setters…
}
```

Old version of code (does not know about photoURL field)

```
Person person = db.read(…);
person.setFavoriteNumber(42);
db.write(person.toJSON());
```

Update, reencode and write back

DB

```
{
   "userName": "Martin",
   "favoriteNumber": 1337,
   "interests": ["hacking"],
   "photoURL": "http://…"
}
```

Data written by new version of code (including new photoURL field)

DB

```
{
   "userName": "Martin",
   "favoriteNumber": 42,
   "interests": ["hacking"]
}
```

Value of photoURL field is lost