

OLAP

Querying in Data Warehouses

- In most OLTP databases, storage is laid out in a **row-oriented fashion**: all the values from one row of a table are stored next to each other.
- Document databases are similar: an entire document is typically stored as one contiguous sequence of bytes.
- In order to process a query, you may have indexes on columns, which tell the storage engine where to find all the sales for a particular date or for a particular product. But then, a row-oriented storage engine still needs to load all of those rows (each consisting of over 100 attributes) from disk into memory, parse them, and filter out those that don't meet the required conditions. That can take a long time.
- The idea behind **column-oriented storage** is simple: don't store all the values from one row together, but store all the values from each column together instead. If each column is stored in a separate file, a query only needs to read and parse those columns that are used in that query, which can save a lot of work.

Rows vs. Columns

fact_sales table

date_key	product_sk	store_sk	promotion_sk	customer_sk	quantity	net_price	discount_price
140102	69	4	NULL	NULL	1	13.99	13.99
140102	69	5	19	NULL	3	14.99	9.99
140102	69	5	NULL	191	1	14.99	14.99
140102	74	3	23	202	5	0.99	0.89
140103	31	2	NULL	NULL	1	2.49	2.49
140103	31	3	NULL	NULL	3	14.99	9.99
140103	31	3	21	123	1	49.99	39.99
140103	31	8	NULL	233	1	0.99	0.99

Columnar storage layout:

date_key file contents: 140102, 140102, 140102, 140102, 140103, 140103, 140103, 140103
product_sk file contents: 69, 69, 69, 74, 31, 31, 31, 31
store_sk file contents: 4, 5, 5, 3, 2, 3, 3, 8
promotion_sk file contents: NULL, 19, NULL, 23, NULL, NULL, 21, NULL
customer_sk file contents: NULL, NULL, 191, 202, NULL, NULL, 123, 233
quantity file contents: 1, 3, 1, 5, 1, 3, 1, 1
net_price file contents: 13.99, 14.99, 14.99, 0.99, 2.49, 14.99, 49.99, 0.99
discount_price file contents: 13.99, 9.99, 14.99, 0.89, 2.49, 9.99, 39.99, 0.99

Columns Compression

Column values:

product_sk:

69	69	69	69	74	31	31	31	31	29	30	30	31	31	31	68	69	69
----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----

Bitmap for each possible value:

product_sk = 29:	<table border="1"><tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr></table>	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0		
product_sk = 30:	<table border="1"><tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr></table>	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0		
product_sk = 31:	<table border="1"><tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td><td>1</td><td>1</td><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td><td>1</td><td>0</td><td>0</td><td>0</td></tr></table>	0	0	0	0	0	1	1	1	1	0	0	0	1	1	1	0	0	0
0	0	0	0	0	1	1	1	1	0	0	0	1	1	1	0	0	0		
product_sk = 68:	<table border="1"><tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td><td>0</td></tr></table>	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0		
product_sk = 69:	<table border="1"><tr><td>1</td><td>1</td><td>1</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td></tr></table>	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	1	1
1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	1	1		
product_sk = 74:	<table border="1"><tr><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr></table>	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0		

Run-length encoding:

product_sk = 29:	9, 1	(9 zeros, 1 one, rest zeros)
product_sk = 30:	10, 2	(10 zeros, 2 ones, rest zeros)
product_sk = 31:	5, 4, 3, 3	(5 zeros, 4 ones, 3 zeros, 3 ones, rest zeros)
product_sk = 68:	15, 1	(15 zeros, 1 one, rest zeros)
product_sk = 69:	0, 4, 12, 2	(0 zeros, 4 ones, 12 zeros, 2 ones)
product_sk = 74:	4, 1	(4 zeros, 1 one, rest zeros)

Querying with Columns

- `WHERE product_sk IN (30, 68, 69):`

Load the three bitmaps for `product_sk = 30`, `product_sk = 68` and `product_sk = 69`, and calculate the **bitwise OR** of the three bitmaps, which can be done very efficiently.

- `WHERE product_sk = 31 AND store_sk = 3:`

Load the bitmaps for `product_sk = 31` and `store_sk = 3`, and calculate the **bitwise AND**. This works because the columns contain the rows in the same order, so the k^{th} bit in one column's bitmap corresponds to the same row as the k^{th} bit in another column's bitmap.

DATA ENCODING

Data Flows

- If two processes **don't share memory**, we need one process to **send some data** to another process
- There are many ways **how data can flow** from one process to another:
 - via databases (data survives code)
 - via calls to services (Web services, REST, RPC)
 - via asynchronous message-passing systems (message brokers)
- We need to **encode** a message as a sequence of bytes

Evolvability

- Applications inevitably **change over time**. In most cases, a **change of application** features also requires a **change to data** that it stores.
- When a data format or schema changes, a corresponding change to application code often needs to happen
- In a large application, code changes often cannot happen instantaneously. This means that **old and new versions** of the code, and old and new data formats, may potentially all **coexist** in the system at the same time
- We need to maintain compatibility in both directions:
 - **Backward compatibility**: Newer code can read data that was written by older code. [EASY]
 - **Forward compatibility**: Older code can read data that was written by newer code. [TRICKY]

Terminology

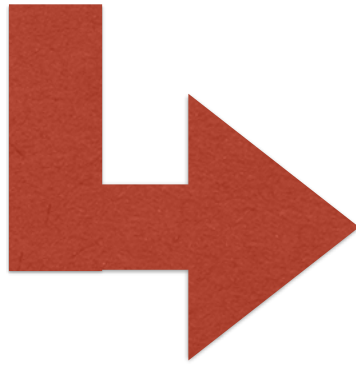
- In **memory**, data is kept in objects, structs, lists, arrays, hash tables, trees and so on. These data structures are optimized for efficient access and manipulation by the CPU (typically using pointers).
- When you want to write data to a **file**, or send it over the **network**, you have to encode it as some kind of self-contained sequence of bytes (for example, a JSON document). Since a pointer wouldn't make sense to any other process, this sequence-of-bytes representation looks quite different from the data structures that are normally used in memory.
- Thus, we need some kind of translation between the two representations. The translation from the in-memory representation to a byte sequence is called **encoding** (also known as **serialization** or **marshalling**), and the reverse is called **decoding** (**parsing**, **deserialization**, **unmarshalling**).
- As this is such a common problem, there are a myriad different libraries and encoding formats to choose from.

Typical Approaches

- Language-specific formats (e.g., `java.io.Serializable`): minimal additional code, but tied to a particular programming language, versioning data is often not a problem, efficiency is not a major problem.
- Textual formats (e.g., JSON, XML, CSV): verbosity issue, lot of ambiguity around the encoding of numbers (what is a real number or a string), good support for Unicode character strings but no support binary strings, quite complicate schema languages (no schema for CSV)
- Binary formats: used for data that is used only internally within your organization, necessary when dealing with terabytes of data, several choices for JSON (MessagePack, BSON, BJSON, UBJSON, BJSON, and Smile, to name a few)

$$\{ \}$$

}



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														

object (3 entries)	string (length 8)	user Name		string (length 6)	Martin	
83	a8	75	73 65 72 4e 61 6d 65	a6	4d 61 72 74 69 6e	
	string (length 14)	favorite Number				
	ae	66	61 76 6f 72 69 74 65 4e 75 6d 62 65 72			
	uint16	1337	string (length 9)	interests		
	cd	05 39	a9	69	6e 74 65 72 65 73 74 73	
array (2 entries)	string (length 11)	daydreaming				
92	ab	64	61 79 64 72 65 61 6d 69 6e 67			
	string (length 7)	hacking				
	a7	68	61 63 6b 69 6e 67			

Thrift and Protocol Buffers

- Apache Thrift (Facebook) and Protocol Buffers (Google) are binary encoding libraries that are based on the same principle. Both were made open source in 2007-08.
- Both Thrift and Protocol Buffers require a schema for any data that is encoded.

Thrift

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

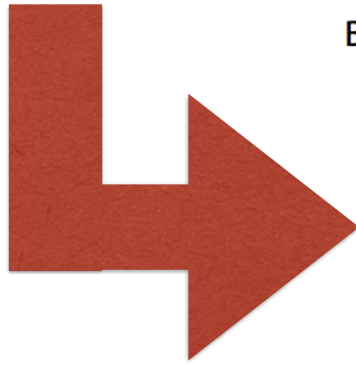
Protocol Buffers

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

Thrift Binary Protocol

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

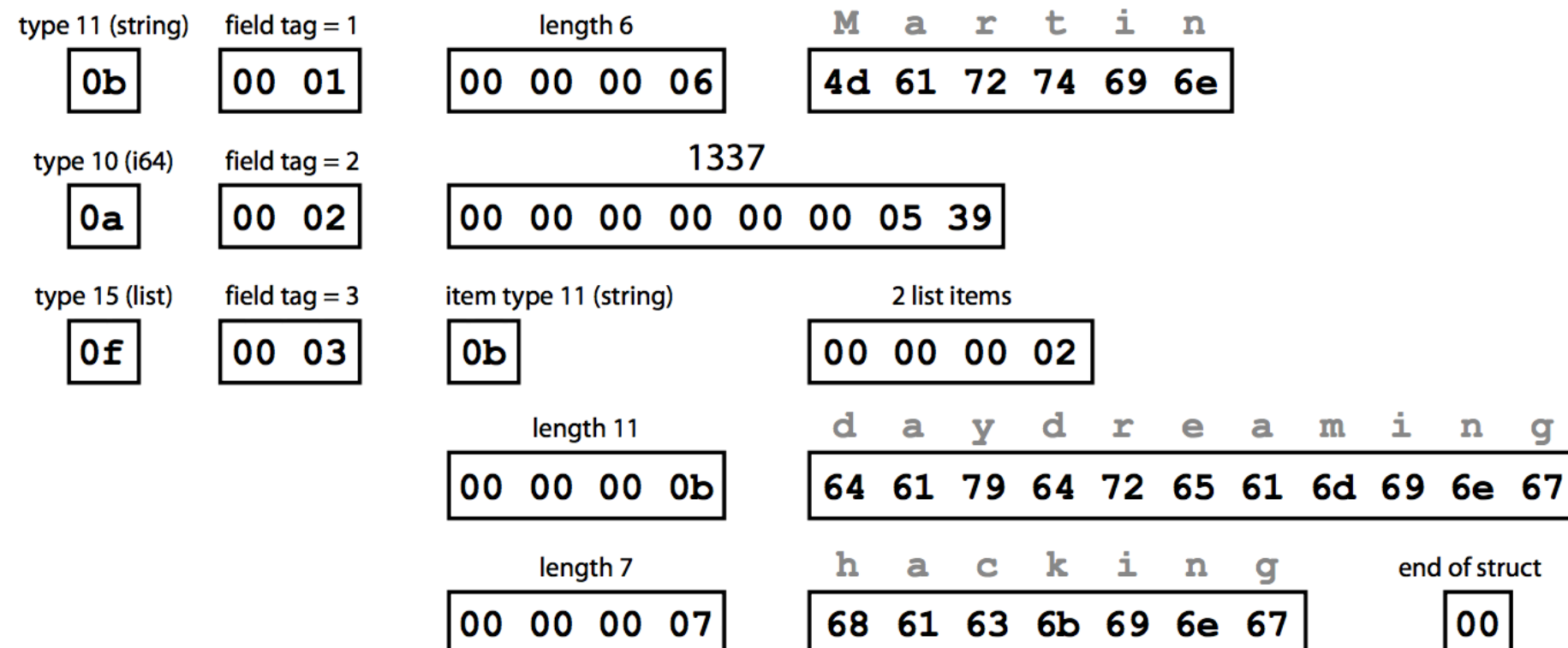
```
struct Person {  
  1: required string      userName,  
  2: optional i64         favoriteNumber,  
  3: optional list<string> interests  
}
```



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			

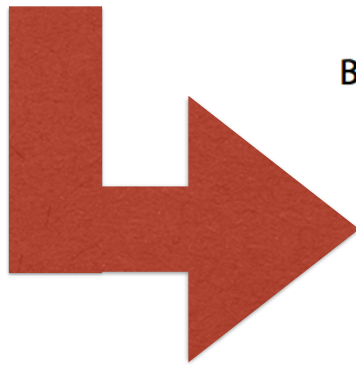
Breakdown:



Thrift Compact Protocol

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

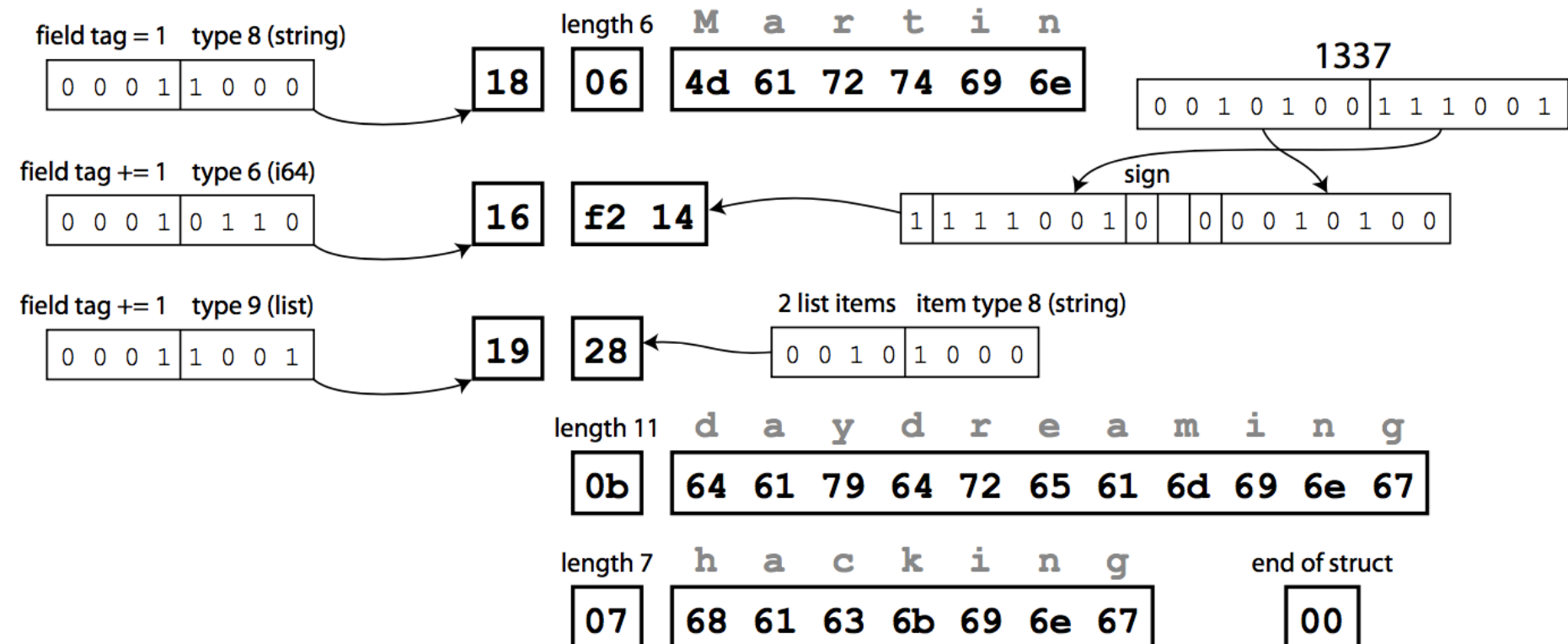
```
struct Person {  
  1: required string      userName,  
  2: optional i64         favoriteNumber,  
  3: optional list<string> interests  
}
```



Byte sequence (34 bytes):

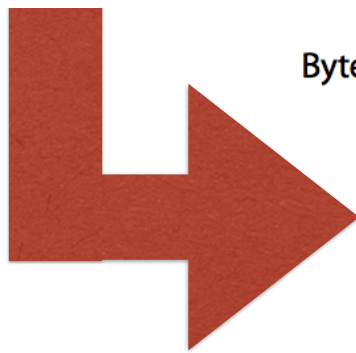
18	06	4d	61	72	74	69	6e	16	f2	14	19	28	0b	64	61	79	64	72	65
61	6d	69	6e	67	07	68	61	63	6b	69	6e	67	00						

Breakdown:



Protocol Buffers

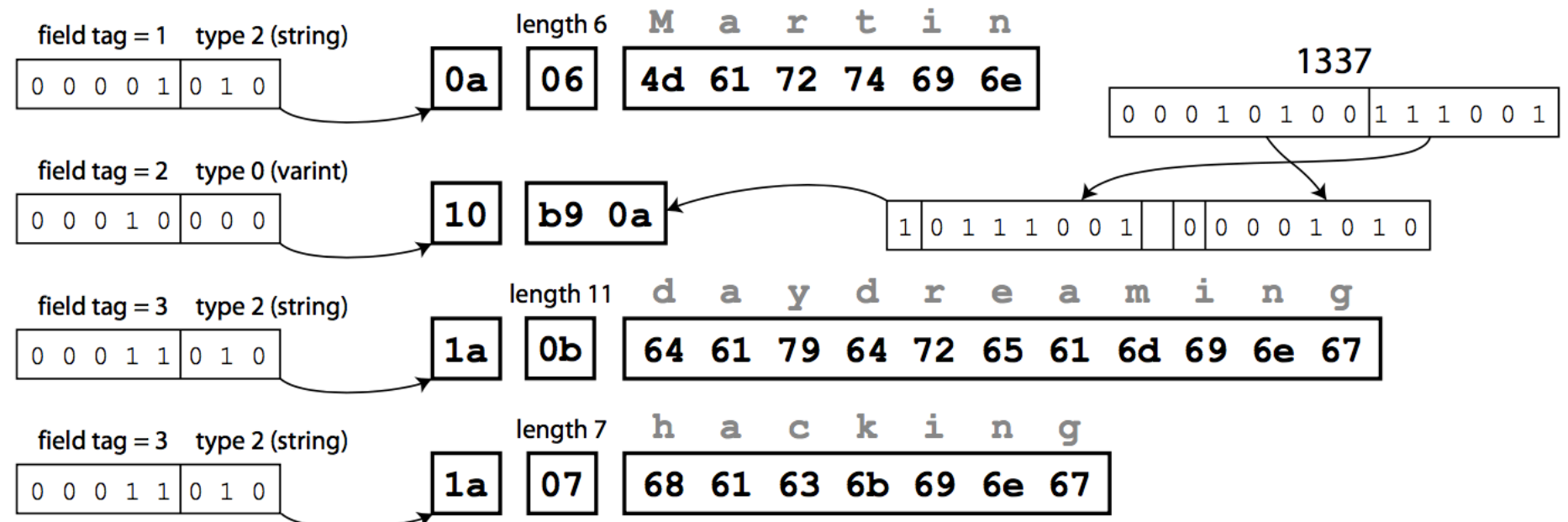
```
message Person {  
  required string user_name      = 1;  
  optional int64  favorite_number = 2;  
  repeated string interests      = 3;  
}
```



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:



Encoded Fields

- An encoded record is just the concatenation of its encoded fields.
- Each field is identified by its tag number (the numbers 1, 2, 3 in the schemas above), and annotated with a datatype (e.g. string or integer).
- If a field value is not set, it is simply omitted from the encoded record.
- You can change the name of a field in the schema, since the encoded data never refers to field names, but you cannot change a field's tag, since that would make all existing encoded data invalid.
- You can add new fields to the schema, provided that you give each field a new tag number.

Schema Evolution

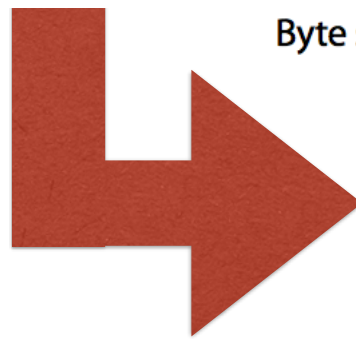
- How do Thrift and Protocol Buffers handle schema changes while keeping backward and forward compatibility?
- **Forward compatibility** (old code can read records that were written by new code): If old code (which doesn't know about the new tag numbers you added) tries to read data written by new code, including new a field with a tag number it doesn't recognize, it can simply ignore that field. The datatype annotation allows the parser to determine how many bytes it needs to skip.
- **Backward compatibility** (new code can read records that were written by old code): As long as each field has a unique tag number, new code can always read old data, because the tag numbers still have the same meaning.
- The only detail is that if you add a new field, you cannot make it required. If you were to add a field and make it required, that check would fail if new code reads data written by old code, because the old code did not write the new field that you added.
- Removing a field is just like adding a field, with backward and forward compatibility concerns reversed. That means you can only remove a field that is optional (a required field can never be removed), and you can never use the same tag number again (because you may still have data written somewhere that includes the old tag number, and that field must be ignored by new code).

Avro

- Apache Avro is another binary encoding format started in 2009 as a sub-project of Hadoop.
- Avro also uses a schema to specify the structure of the data being encoded. It has two schema languages: one for human editing, and one more easily machine-readable.
- There are no tag numbers in the schema!

Avro

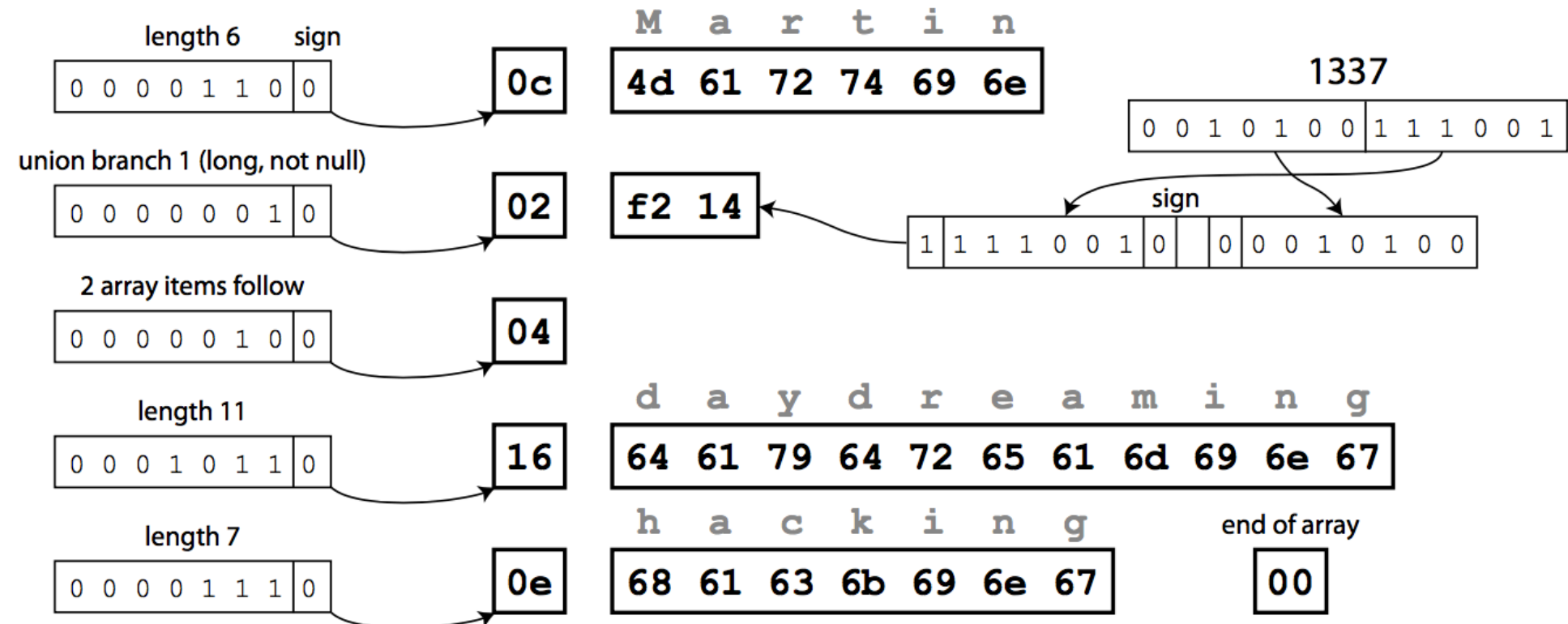
```
record Person {  
  string      userName;  
  union { null, long } favoriteNumber = null;  
  array<string> interests;  
}
```



Byte sequence (32 bytes):

0c	4d	61	72	74	69	6e	02	f2	14	04	16	64	61	79	64	72	65	61	6d
69	6e	67	0e	68	61	63	6b	69	6e	67	00								

Breakdown:



Reader's and writer's schemas

Writer's schema for Person record

Datatype	Field name
string	userName
union {null, long}	favoriteNumber
array<string>	interests
string	photoURL

Reader's schema for Person record

Datatype	Field name
long	userID
union {null, int}	favoriteNumber
string	userName
array<string>	interests

- How does the reader know the writer's schema with which a particular piece of data was encoded?
 1. Large file with lots of records (e.g., Hadoop): include the writer's schema once at the beginning of the file.
 2. Database with individually written records: include a version number at the beginning of every encoded record, and to keep a list of schema versions in your database
 3. Sending records over a network connection: when two processes are communicating over a bidirectional network connection, they can negotiate the schema version on connection setup, and then continue using that schema for the lifetime of the connection.