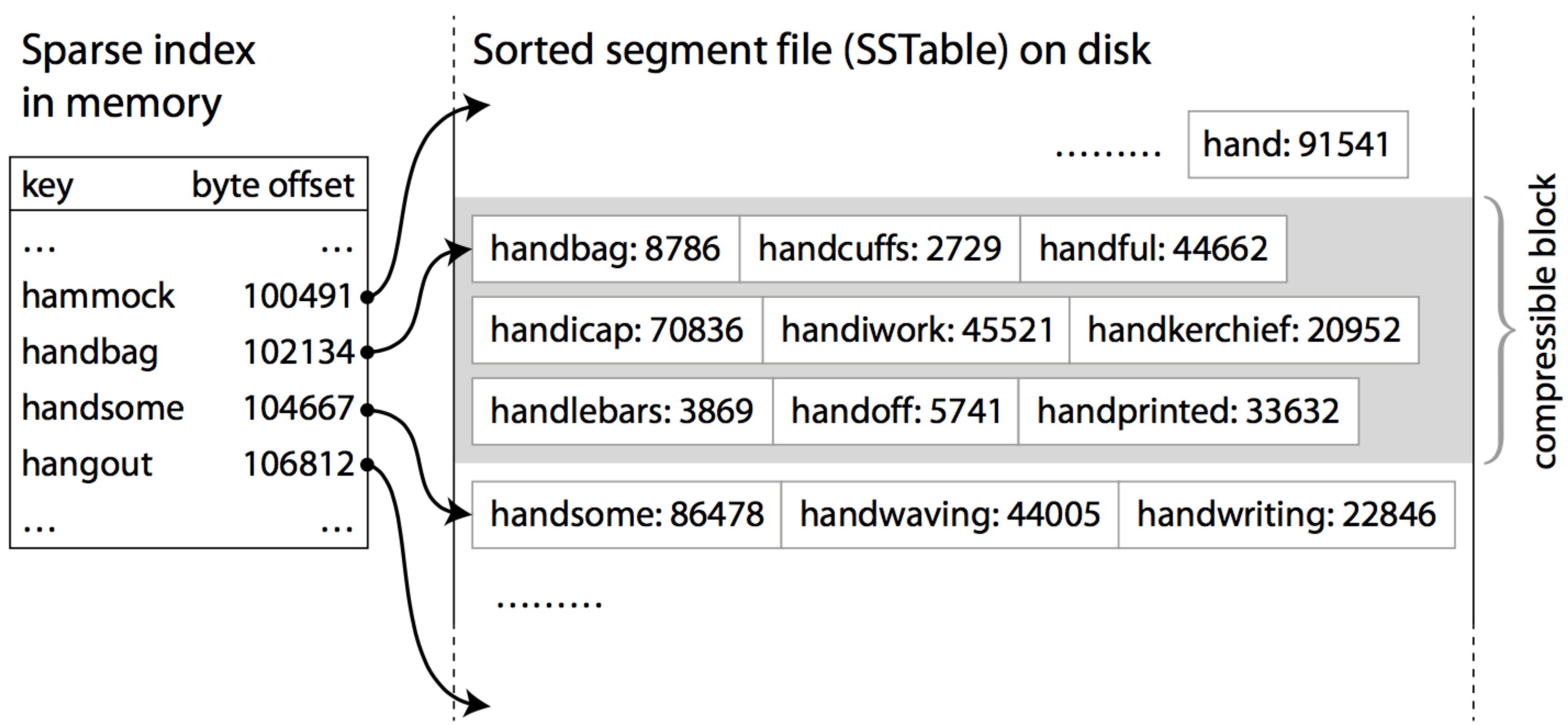
Sorted String Tables

- Sorted String Table (SSTable): the sequence of key-value pairs is sorted by key (no insertion time)
- We also require that each key only appears once within each merged segment file (the merging process already ensures that).
- SSTables have several big advantages over segments with hash indexes:
    - Merging segments is simple and efficient, even if the files are bigger than the available memory. When multiple segments contain the same key, we can keep the value from the most recent segment, and discard the values in older segments.
    - No longer need to keep an index of all the keys in memory, just use an in-memory sparse index for the the segments boundaries, then linear scan of a segment.
    - Since read requests need to scan over several key-value pairs in the requested range anyway, it is possible to group those records into a block and compress it before writing it to disk.



Storage Engine Workflow

- When a write comes in, add it to an in-memory balanced tree data structure (e.g., a Red-Black tree). This in-memory tree is sometimes called a memtable.

- When the memtable gets bigger than some threshold — typically a few megabytes — write it out to disk as an SSTable file. This can be done efficiently because the tree already maintains the key-value pairs sorted by key. The new SSTable file becomes the most recent segment of the database. When the new SSTable is ready, the memtable can be emptied.

- In order to serve a read request, first try to find the key in the memtable, then in the most recent on-disk segment, then in the next-older segment, etc.

- From time to time, run a merging and compaction process in the background to combine segment files and to discard overwritten or deleted values.

- If the database crashes, the most recent writes (which are in the memtable but not yet written out to disk) are lost. In order to avoid that problem, we can keep a separate hash indexed log on disk to which every write is immediately appended. Every time the memtable is written out to an SSTable, the corresponding log can be discarded.

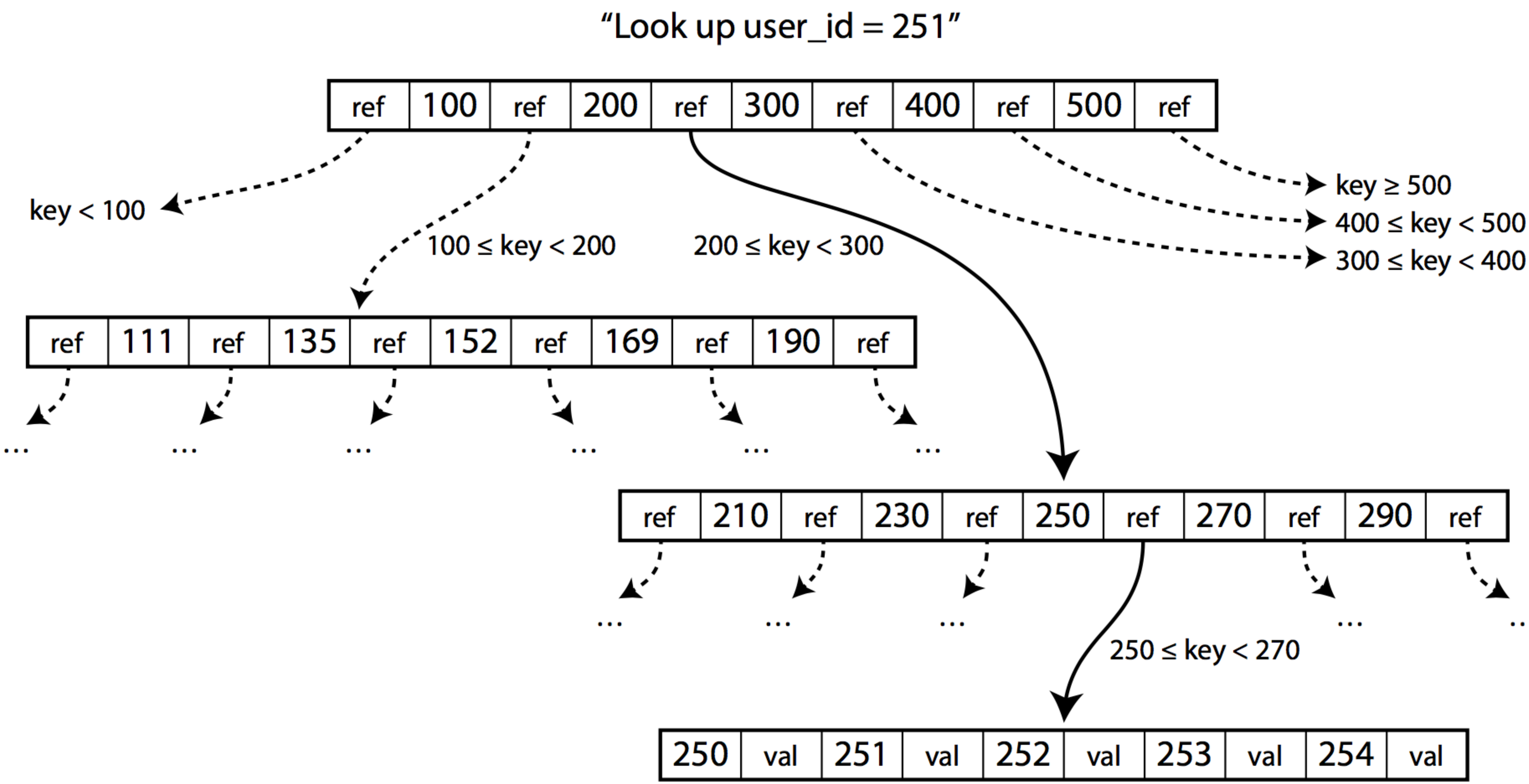
Notes

- Originally this indexing structure was names Log-Structured Merge-Tree (LSM-Tree)
    - P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil: "*The Log- Structured Merge-Tree (LSM-Tree)*", Acta Informatica, volume 33, number 4, pages 351–385, June 1996.
- BitCask implements hash indexes.
- LevelDB, RocksDB, Cassandra, Hadoop's HBase, Google's BigTable use similar approach.

B-Trees

- The most widely-used indexing structure is quite different: the B-tree.
- Like SSTables, B-trees keep key-value pairs sorted by key, which allows efficient key-value lookups and range queries.
- SSTables break the database down into variable-size segments, typically several megabytes or more in size, and always write a segment sequentially.
- B-trees break the database down into fixed-size blocks or pages, traditionally 4 kB in size, and read or write one page at a time. This corresponds more closely to the underlying hardware, as disks are also arranged in fixed-size blocks.

B-Trees



"Look up user_id = 251"
ref
100
ref
200
ref
300
ref
400
ref
500
ref
key < 100
100 ≤ key < 200
200 ≤ key < 300
key ≥ 500
400 ≤ key < 500
300 ≤ key < 400
ref
111
ref
135
ref
152
ref
169
ref
190
ref
ref
210
ref
230
ref
250
ref
270
ref
290
ref
250 ≤ key < 270
250
val
251
val
252
val
253
val
254
val

B-Trees

- **Update a value for an existing key**: search for the leaf page containing that key, and change the value that page, and write the page back to disk (any references to that page remain valid).

- **Add a new key**: find the page whose range encompasses the new key, and add it to that page. If there isn't enough free space in the page to accommodate the new key, it is split into two half-full pages, and the parent page is updated to account for the new subdivision of key ranges

- This algorithm for key additions ensures that the tree remains balanced: a B-tree with *n* keys always has a height of *O(log n)*.

- B-tree implementations normally include a write-ahead log (WAL a.k.a. redo log). This is an append-only file to which every B-tree modification must be written before it can be applied to the pages of the tree itself. When the database comes back up after a crash, this log is used to restore the B-tree back to a consistent state.

- In-place updates required locks to avoid that a thread sees the tree in an inconsistent state.

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:	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
product_sk = 30:	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0
product_sk = 31:	0	0	0	0	0	1	1	1	1	0	0	0	1	1	1	0	0	0
product_sk = 68:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
product_sk = 69:	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	1	1
product_sk = 74:	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, BISON, and Smile, to name a few)

JSON MessagePack Example

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

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 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
```

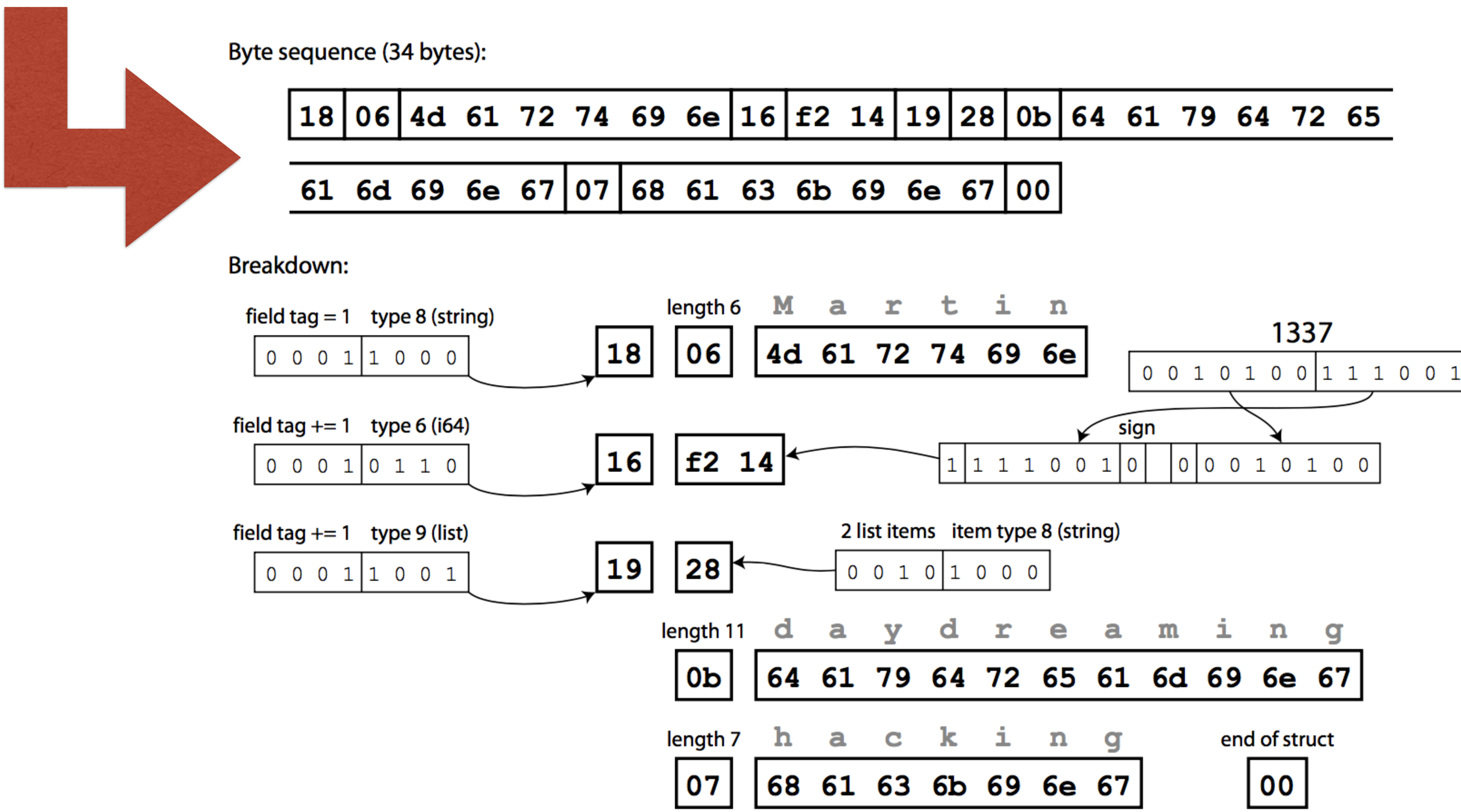
Breakdown:

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
		00 00 00 07	68 61 63 6b 69 6e 67
			end of struct
			00

Thrift Compact Protocol

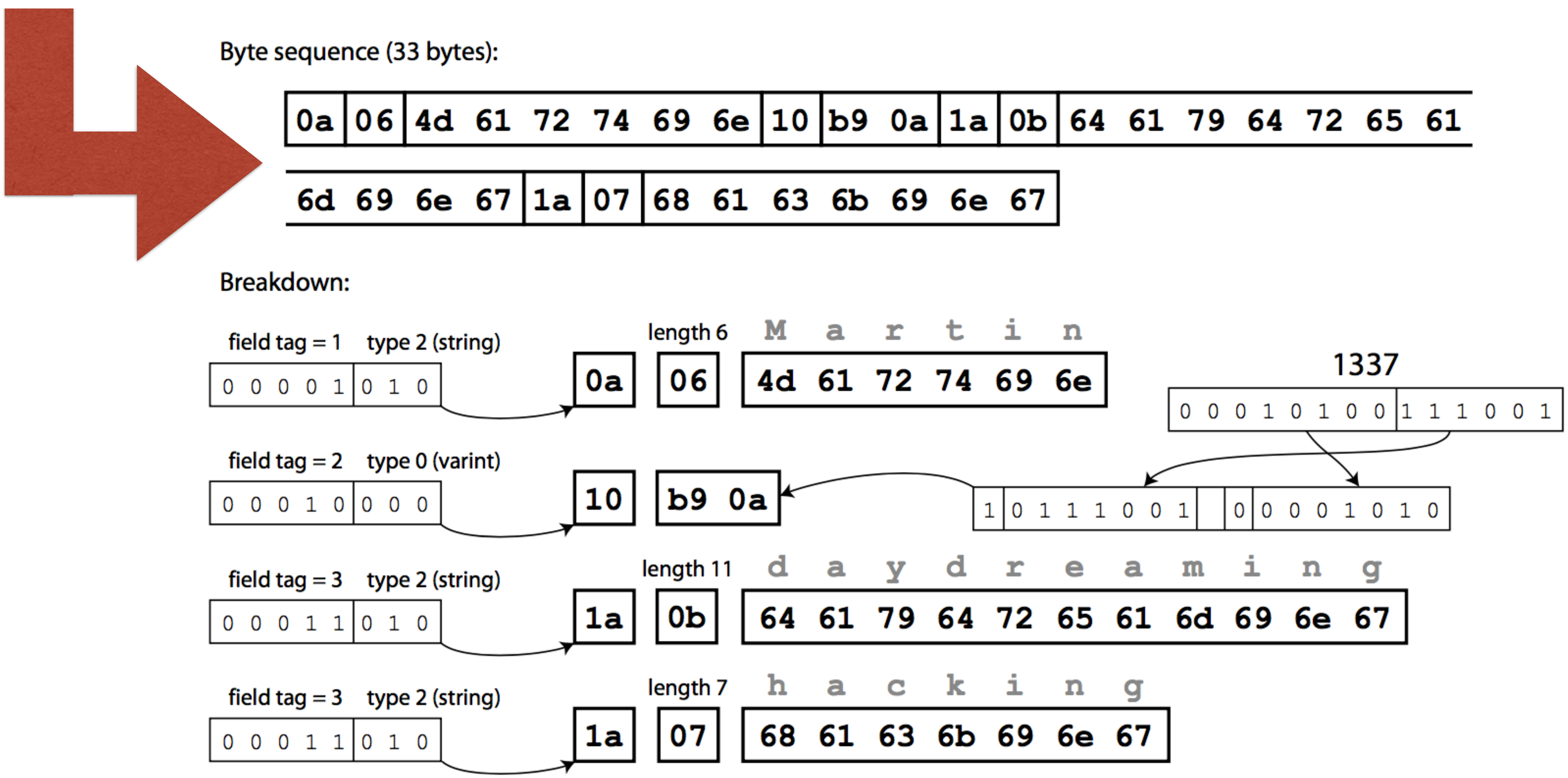
```
{
    "userName": "Martin",
    "favoriteNumber": 1337,
    "interests": ["daydreaming", "hacking"]
}
```

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



Protocol Buffers

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



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 <u>if you add a new field, you cannot make it required</u>. 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.

- <u>Removing a field is just like adding a field</u>, 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

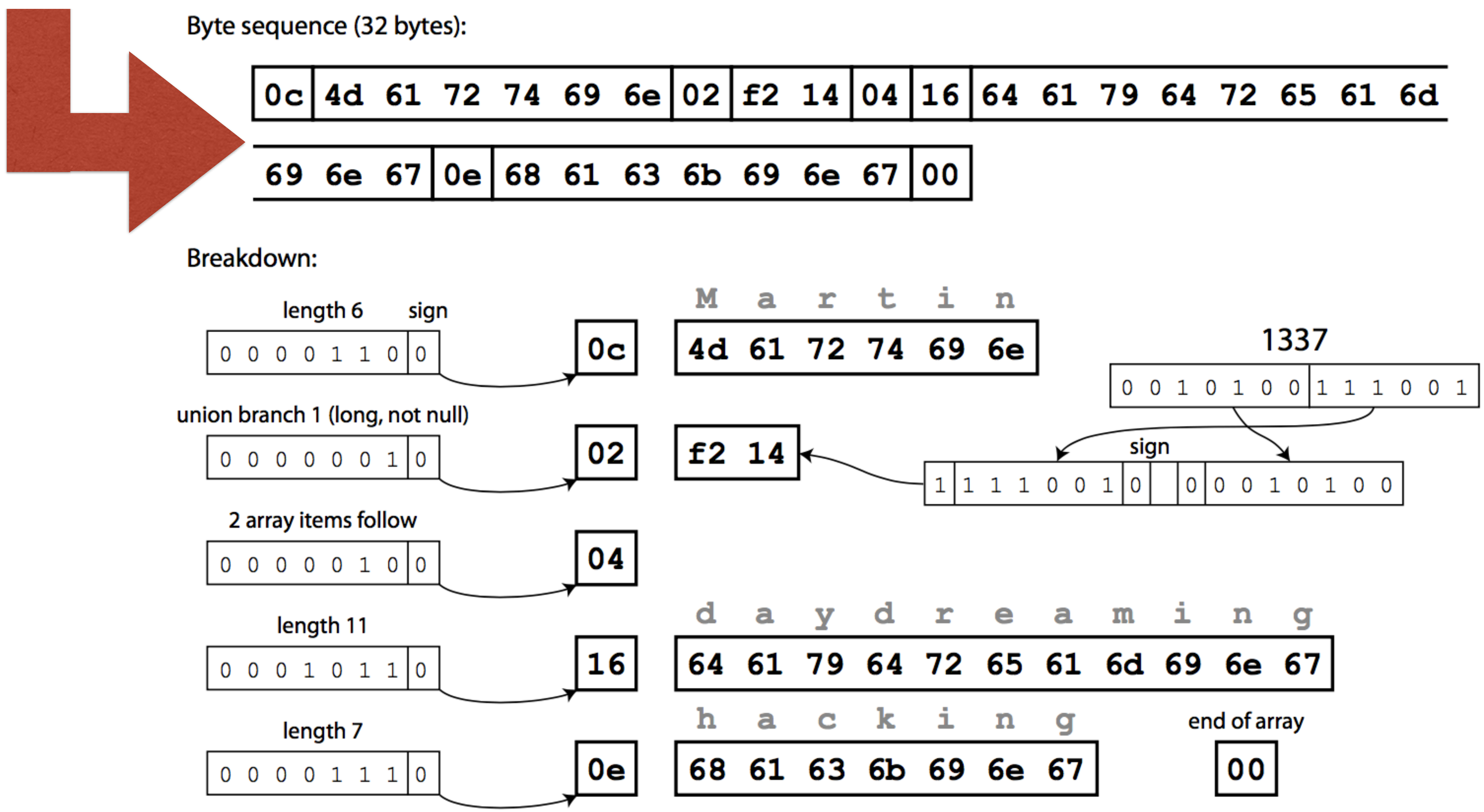
- Apache Avro is another binary encoding format started in 2009 as a sub-project of Hadoop.

- 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;
}
```



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.