Data Storage and Encoding

CHIDATA

Logical v.s. Physical Representation

Logical: How the program presents the data to the analyst

‘red’: 5

‘blue’: 3

‘red’: 7

Physical: How the data is actually organized and modeled in a computer.

Index Color Number

‘red’: 5 → 00 0 101

‘blue’: 3 → 01 1 011

‘red’: 7 → 10 0 111

Color	Index	
0	00	10
1	01	

Logical v.s. Physical Representation

Logical: How the program presents the data to the analyst

```
df[df['color'] = 'red']
```

Color **Index**

0 **00** **10**

1 **01**

Index **Color** **Number**

00 0 101

01 1 011

10 0 111

Fixed Size v.s. Variable Size Data

Let's focus on single values first

Fixed-Size: Number of bits required to store the data is independent of the value. Hint: things that look like numbers

- Boolean (True/False) 1 bit
- Integer 16-bits
- Float 32-bits

Variable-Size: Number of bits required to store the data varies with the alue.

- Strings 1 byte per character + termination character.
- Arrays
- …

Fixed Size v.s. Variable Size Data

Fixed-Size: Number of bits required to store the data is independent of the value.

Integer = 4 (16 bits to store)
Integer = 631123 (16 bits to store)

Variable-Size: Number of bits required to store the data varies with the value.

String = ‘bear’ (5 bytes = 40 bits to store)
String = ‘q’ (2 bytes = 16 bits to store)

Encoding Collections of Data

Simplified storage problem: Collection of N items of same data type

```
data = [i1,i2,i3,…,iN]
```

Encoding: Turn the collection into a sequence of bits

Decoding: Turn a sequence of bits back into data

```
enc(item) => returns bits for one item
dec(bits) => returns item corresponding to bits
```

Encoding Collections of Data

Encoding: Turn the collection into a sequence of bits

```
buffer = []
for item in data:
  buffer.append(enc(item))
```

Decoding Collections of Data

Decoding Fixed-Size Data is Easy

Example. New data type called a 'Mint' 2-bit integer

[1,0,2]

↓ Encode

010010

↓ Decode

```
for i in range(2, N, 2):
    yield dec(data[i-2:i])
```

↓

[1,0,2]

Fixed Size v.s. Variable Size Data

Variable size data is a bit more complicated

['blue', 'pink' , 'green']

↓

Encode

bluepinkgreen

Can't tell where one string starts and another ends!

CHIDATA

Fixed Size v.s. Variable Size Data

[‘blue’, ‘pink’ , ‘green’]

Encode

~~bluepinkgreen~~

blue0pink0green0

Decode

```
buffer = []
for i in range(0, N):
   if data[i] == 0:
      yield buffer
      buffer = []
   else:
      buffer.append(data[i])
```

Dictionary Encoding

Leverage redundancy in data collections

```
data = ['red', 'blue', 'red', 'red']
```

Naive: Store as strings

'red' 4 bytes (32 bits)

'blue' 5 bytes (40 bits)

3*32 + 1*40 = 136 bits

Dictionary Encoding

Leverage redundancy in data collections

```
data = [‘red’, ‘blue’, ‘red’, ‘red’]
```

Dictionary Encode: Analyze data and build a lookup table with fixed size values.

```
‘red’ => 0
‘blue’ => 1

data = [0, 1, 0, 0]
```

Dictionary Encoding

Leverage redundancy in data collections

```
data = [‘red’, ‘blue’, ‘red’, ‘red’]
```

Dictionary Encode: Analyze data and build a lookup table with fixed size values.

```
‘red’ => 0
‘blue’ => 1
```

(32 + 1) + (40 + 1) bits lookup

```
data = [0, 1, 0, 0]
```

4 bits storage

76 bits total!

Dictionary Encoding

Basic insight: Data >> Domain = Profit!!

```
N strings, K distinct strings, L avg. length/string
```

Naive: s = 8*(L+1) # bit length avg
N*s #total

Dictionary: b = ceil($\log_2$(k)) # bits to rep each item
k*(s + b) # lookup table size
N*b #storage size

N*b + k(s + b) #total

CHIDATA

Dictionary Encoding

Basic insight: Data >> Domain = Profit!!

```
N strings, K distinct strings, L avg. length/string
```

Naive: N*s #total

Dictionary: N*b + k(s + b) #total

Compression Ratio: $$\frac{N*s}{N*b + k(s + b)}$$

For large N: $$\frac{s}{b}$$

Dictionary Encoding

Could it be useful for integers?

```
SAT_Scores = [1200, 1550,…, 1432]
```

Very Naive: Store as full int

16*N

Dictionary Encode: Only encode all distinct integers (0,1600)

$$\frac{s}{b} \quad \frac{16}{\mathrm{ceil}(\log_2 1600)} = 1.45$$

Dictionary Encoding

Suppose you could only get scores in "10s"

```
SAT_Scores = [1200, 1550,…, 1430]
```

Very Naive: Store as full int
16*N

Reduced Int: Store as a 11-bit int (0,2048)
11*N

Dictionary Encode: Only encode all possible values (0,1600,10)

$$\frac{s}{b} \quad \frac{16}{\mathrm{ceil}(\log_2 160)} = 2$$

Dictionary Encoding

So what is the downside?

```
SAT_Scores = [1200, 1550,…, 1432]
```

Count all SAT scores that are multiples of 30—have to fully decode to answer this question!

Trading off space for computation

Compute Directly on Encoding

"Encode" the computation

```
data = ['red', 'blue', 'red', 'red']

enc = [0, 1, 0, 0]
```

```
find_replace('red',  'blue')
find_replace(0,  1)
```

```
enc = [1, 1, 1, 1]
```

Equality operations can be translated.

Dictionary Encoding

Find all SAT scores less than 1300

```
SAT_Scores = [1200, 1550,…, 1432]
```

We can translate 1300—but it's not guaranteed to work in general!

Order preserving dictionary:

```
0 : 000000000
10 : 000000001
20 : 000000010
30 : 000000011
..
```

>,< operations can be translated in an order preserving dictionary.

Dictionary Encoding

```
SAT_Scores = [1200, 1550,…, 1432]
```

Order preserving dictionary:

```
0 : 000000000
10 : 000000001
20 : 000000010
30 : 000000011
..
```

Suppose we get a new value 1205.

Have to decode, and re-encode the whole data!

Dictionary Encoding

```
data = [‘red’, ‘blue’, ‘red’, ‘red’] + [‘black’]

‘red’ => 00
‘blue’ => 01
‘black’ => 10

data = [00, 01, 00, 00,10]
```

Regular dictionary encoding can support new values without decoding and re-encoding.

Can even leave “slack” for future new values.