RBC — a binary JSON format

Mahdi Yousefi and Ramesh Subramonian

February 14, 2023

Abstract

This paper specifies the binary JSON grammar

1 Grammar to parse binary data

An RBC must have one of the following types, called a `jtype`

1. null
2. boolean
3. number
4. string
5. array
6. object (or map)

Given the start of an RBC, one must be able to determine its length.

1.1 qtypes

```
typedef enum {
  Q0, // mixed  must be first one
  I1,
  I2,
  I4,
  I8,
  F4,
  F8,
  UI1,
  UI2,
  UI4,
  UI8,
  SC,  // constant length strings
  SV,  // variable length strings
```

```
  TM,  // time struct
  HL,  // holiday bit mask
  NUM_QTYPES // must be last one
} qtype_t;
```

1.2 Notations and Definitions

Notation 1 *poff = parent offset, currently unused* $\boxed{\textbf{\textit{TO BE COMPLETED}}}$

Notation 2 *The term "padding" means increasing the width of a datum to be a multiple of 8, with any additional bytes set to 0.*

Definition 1 *An array is said to be "uniform" when all elements have the same qtype*

2 null

1. Byte 0 contains jtype.
2. Bytes 1 — 3 unused
3. Bytes 4 — 7 contains poff
4. Length is 8 bytes

3 boolean

1. Byte 0 contains jtype.
2. Bytes 1 contains value (0 or 1)
3. Bytes 2 — 3 unused
4. Bytes 4 — 7 contains poff
5. Length is 8 bytes

4 number

1. Byte 0 contains jtype
2. Byte 1 contains qtype
3. Bytes 2 — 3 unused
4. Bytes 4 — 7 contains poff
5. Byte 8-15 contains value (number)
6. Length is 16 bytes

5 date

This is actually `typedef struct _tm_t` specified in qtypes

1. Byte 0 contains jtype
2. Byte 1 contains qtype
3. Bytes 2 — 3 unused
4. Bytes 4 — 7 contains poff
5. Byte 8-15 contains value (`tm_t`)
6. Length is 16 bytes

6 string

1. Byte 0 contains jtype
2. Byte 1 contains qtype, set to SC
3. Bytes 2 — 3 unused
4. Bytes 4 — 7 contains poff
5. Byte 8-11 contains length, n
6. Byte 12-15 contains len_alloc, s
   (a) s is a multiple of 8
   (b) $s \geq n + 1$.
      i. The reason for the +1 is to have null characater termination
      ii. The reason for allocationg more than we need is to allow in-place updates without having to move things around.
7. Length is 16 bytes plus size

7 array

1. Byte 0 contains jtype
2. Byte 1 contains qtype
3. Byte 2 contains width if uniform array; 0, otherwise
4. Bytes 3 – 3 unused
5. Bytes 4 — 7 contains poff
6. Byte 8-11 contains number of elements resident n
7. Byte 12-15 contains number of elements allocated, s
8. Byte 16-19 contains length of RBC, l
9. Byte 20-23 unused

7.1 uniform array

1. $s \times w$ bytes, padded.

7.2 mixed array

1. n offsets stored as 4-byte integers, with padding applied. Call this array O. Then, $O[i]$ tells us the location of the ith element of the array as an offset from the start of this item.
2. concatenation of the binary representation of each item. Since an item must be a multiple of 8, this blob is also a multiple of 8.

8 object

1. Byte 0 contains jtype
2. Byte 1 contains qtype **if** all items same type
3. Bytes 2 — 3 unused
4. Bytes 4 — 7 contains poff
5. Byte 8 — 11 contains number of elements, n
6. Byte 12 — 15 unused
7. Byte 16 — 19 contains length of RBC, l
8. Byte 20 — 23 unused
9. $4 \times n$ bytes for key offsets, padded.
10. $4 \times n$ bytes for val offsets, padded
11. $8 \times n$ bytes for (hash/idx), explained in Section 8.1
12. K bytes representing a concatenation of all keys, padded. If the lengths of the keys are $\{l_1, l_2, \dots l_n\}$, then $K = n + \sum_{i=1}^{i=n} l_i$, with a null character between strings.
13. the values

8.1 Fast lookup of keys in object

To reduce the time to locate a key in an object, we store n hash/index pairs, each of which is a 64-bit unsigned integer. The top 48 bits represent a hash of the key and the bottom 16 bits represent the index of the key as stored. This limit the number of keys to 65536 — we do not think this is an unreasonable limitation. Let us assume that an object has 3 keys, x, y, z stored in that order. Let us assume that $h(x) = 30, h(y) = 10, h(z) = 20$. In that case, we store $\{(10, 1), (20, 2), (30, 3)\}$. Note that thee keys are storted in ascending order.

At run time, when we are given a key, we compute its hash. We perform a binary search on the hash/idx array and we have of 2 outcomes

1. the hash does not exist and we inform the user accordingly
2. the hash does exist. We pick up the index i and using the key offset, compare the i^{th} key withe one provided by the user. Once again, we have 2 outcomes.
   (a) If we get a match, use the value offsets and i to return the correesponding value.
   (b) If not, we know that the hash/idx structure has failed us and we perform a linear search of the keys until we either find what we want or report the key to be missing