CRAM format specification (version 3.0)

samtools-devel@lists.sourceforge.net

26 Mar 2015

The master version of this document can be found at `https://github.com/samtools/hts-specs`.
This printing is version cbd60fe from that repository, last modified on the date shown above.

1 Overview

This specification describes the CRAM 3.0 format.

CRAM has the following major objectives:

1. Significantly better lossless compression than BAM
2. Full compatibility with BAM
3. Effortless transition to CRAM from using BAM files
4. Support for controlled loss of BAM data

The first three objectives allow users to take immediate advantage of the CRAM format while offering a smooth transition path from using BAM files. The fourth objective supports the exploration of different lossy compression strategies and provides a framework in which to effect these choices. Please note that the CRAM format does not impose any rules about what data should or should not be preserved. Instead, CRAM supports a wide range of lossless and lossy data preservation strategies enabling users to choose which data should be preserved.

Data in CRAM is stored either as CRAM records or using one of the general purpose compressors (gzip, bzip2). CRAM records are compressed using a number of different encoding strategies. For example, bases are reference compressed (Hsi-Yang Fritz, et al. (2011) Genome Res. 21:734-740) by encoding base differences rather than storing the bases themselves.

2 Data types

CRAM specification uses logical data types and storage data types; logical data types are written as words (e.g. int) while physical data types are written using single letters (e.g. i). The difference between the two is that storage data types define how logical data types are stored in CRAM. Data in CRAM is stored either as as bits or as bytes. Writing values as bits and bytes is described in detail below.

2.1 Logical data types

Byte
: Signed byte (8 bits).

Integer
: Signed 32-bit integer.

Long
Signed 64-bit integer.

Array
An array of any logical data type: <type>[ ]

2.2 Writing bits to a bit stream

A bit stream consists of a sequence of 1s and 0s. The bits are written most significant bit first where new bits are stacked to the right and full bytes on the left are written out. In a bit stream the last byte will be incomplete if less than 8 bits have been written to it. In this case the bits in the last byte are shifted to the left.

Example of writing to bit stream

Let's consider the following example. The table below shows a sequence of write operations:

Operation order	Buffer state before	Written bits	Buffer state after	Issued bytes
1	0x0	1	0x1	-
2	0x1	0	0x2	-
3	0x2	11	0xB	-
4	0xB	0000 0111	0x7	0xB0

After flushing the above bit stream the following bytes are written: 0xB0 0x70. Please note that the last byte was 0x7 before shifting to the left and became 0x70 after that:

```
> echo "obase=16; ibase=2; 00000111" | bc
7

> echo "obase=16; ibase=2; 01110000" | bc
70
```

And the whole bit sequence:

```
> echo "obase=2; ibase=16; B070" | bc
1011000001110000
```

When reading the bits from the bit sequence it must be known that only 12 bits are meaningful and the bit stream should not be read after that.

Note on writing to bit stream

When writing to a bit stream both the value and the number of bits in the value must be known. This is because programming languages normally operate with bytes (8 bits) and to specify which bits are to be written requires a bit-holder, for example an integer, and the number of bits in it. Equally, when reading a value from a bit stream the number of bits must be known in advance. In case of prefix codes (e.g. Huffman) all possible bit combinations are either known in advance or it is possible to calculate how many bits will follow based on the first few bits. Alternatively, two codes can be combined, where the first contains the number of bits to read.

2.3 Writing bytes to a byte stream

The interpretation of byte stream is straightforward. CRAM uses little endiannes for bytes when applicable and defines the following storage data types:

Boolean (bool)
Boolean is written as 1-byte with 0x0 being 'false' and 0x1 being 'true'.

Integer (int32)
Signed 32-bit integer, written as 4 bytes in little-endian byte order.

Long (int64)
Signed 64-bit integer, written as 8 bytes in little-endian byte order.

ITF-8 integer (itf8)

This is an alternative way to write an integer value. The idea is similar to UTF-8 encoding and therefore this encoding is called ITF-8 (Integer Transformation Format - 8 bit).

The most significant bits of the first byte have special meaning and are called 'prefix'. These are 0 to 4 true bits followed by a 0. The number of 1's denote the number of bytes the follow. To accommodate 32 bits such representation requires 5 bytes with only 4 lower bits used in the last byte 5.

LTF-8 long or (ltf8)

See ITF-8 for more details. The only difference between ITF-8 and LTF-8 is the number of bytes used to encode a single value. To do so 64 bits are required and this can be done with 9 byte at most with the first byte consisting of just 1s or 0xFF value.

Array ([ ])

Array length is written first as integer (itf8), followed by the elements of the array.

Encoding

Encoding is a data type that specifies how data series have been compressed. Encodings are defined as encoding<type> where the type is a logical data type as opposed to a storage data type.

An encoding is written as follows. The first integer (itf8) denotes the codec id and the second integer (itf8) the number of bytes in the following encoding-specific values.

Subexponential encoding example:

Value	Type	Name
0x7	itf8	codec id
0x2	itf8	number of bytes to follow
0x0	itf8	offset
0x1	itf8	K parameter

The first byte "0x7" is the codec id.

The second 4 bytes "0x0 0x0 0x0 0xD" denote the length of the bytes to follow (13).

The subexponential encoding has 3 parameters: integer (itf8) K, int (itf8) offset and boolean (bool) unary bit:

K = 0x1 = 1

offset = 0x0 = 0

Map

A map is a collection of keys and associated values. A map with N keys is written as follows:

size in bytes	N	key 1	value 1	key ...	value ...	key N	value N

Both the size in bytes and the number of keys are written as integer (itf8). Keys and values are written according to their data types and are specific to each map.

2.4 Strings

Strings are represented as byte arrays using UTF-8 format. Read names, reference sequence names and tag values with type 'Z' are stored as UTF-8.

3 Encodings

Encoding is a data structure that captures information about compression details of a data series that are required to uncompress it. This could be a set of constants required to initialize a specific decompression algorithm or statistical properties of a data series or, in case of data series being stored in an external block, the block content id.

Encoding notation is defined as the keyword 'encoding' followed by its data type in angular brackets, for example 'encoding<byte>' stands for an encoding that operates on a data series of data type 'byte'.

Encodings may have parameters of different data types, for example the external encoding has only one parameter, integer id of the external block. The following encodings are defined:

Codec	ID	Parameters	Comment
NULL	0	none	series not preserved
EXTERNAL	1	int block content id	the block content identifier used to associate external data blocks with data series
GOLOMB	2	int offset, int M	Golomb coding
HUFFMAN_INT	3	int array, int array	coding with int values
BYTE_ARRAY_LEN	4	encoding<int> array length, encoding<byte> bytes	coding of byte arrays with array length
BYTE_ARRAY_STOP	5	byte stop, int external block content id	coding of byte arrays with a stop value
BETA	6	int offset, int number of bits	binary coding
SUBEXP	7	int offset, int K	subexponential coding
GOLOMB_RICE	8	int offset, int log2m	Golomb-Rice coding
GAMMA	9	int offset	Elias gamma coding

A more detailed description of all the above coding algorithms and their parameters can be found in the Codings section.

4 Checksums

The checksumming is used to ensure data integrity. The following checksumming algorithms are used in CRAM.

4.1 CRC32

This is a cyclic redundancy checksum 32-bit long with the polynomial 0x04C11DB7. Please refer to ITU-T V.42 for more details. The value of the CRC32 hash function is written as an integer.

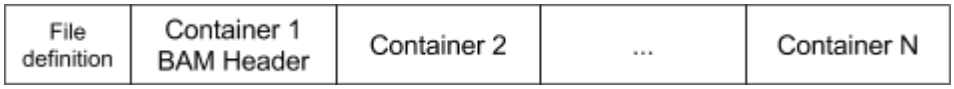
4.2 CRC32 sum

CRC32 sum is a combination of CRC32 values by summing up all individual CRC32 values modulo 2^{32}.

5 File structure

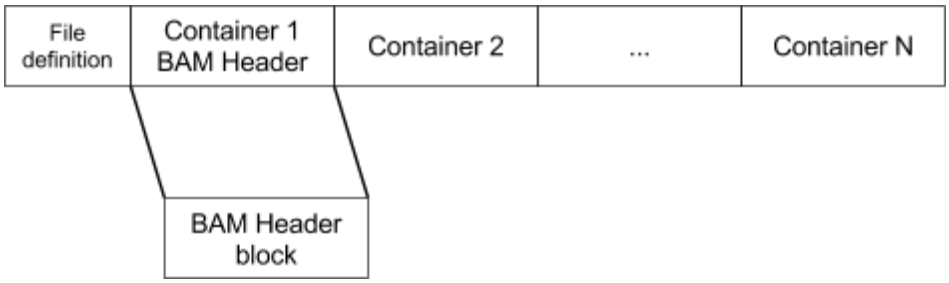
The overall CRAM file structure is described in this section. Please refer to other sections of this document for more detailed information.

A CRAM file starts with a fixed length file definition followed by one or more containers. The BAM header is stored in the first container.



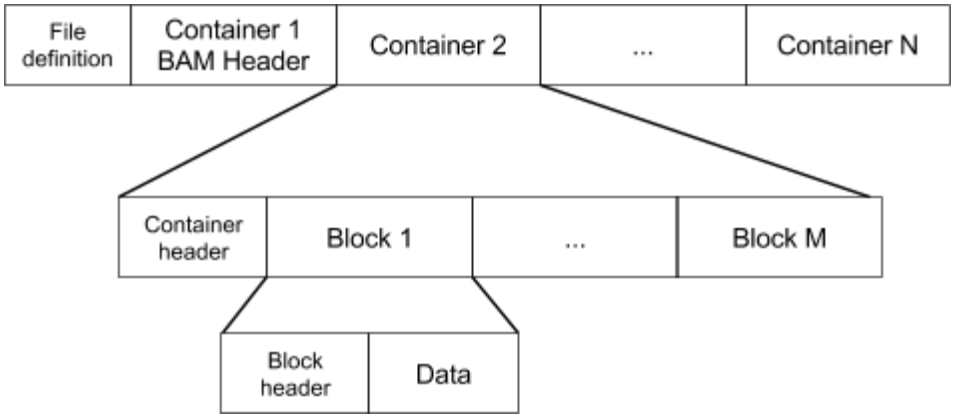
Pic.1 CRAM file starts with a file definition followed by the BAM header and other containers.

Containers consist of one or more blocks. By convention, the BAM header is stored in the first container within a single block. This is known as the BAM header block.



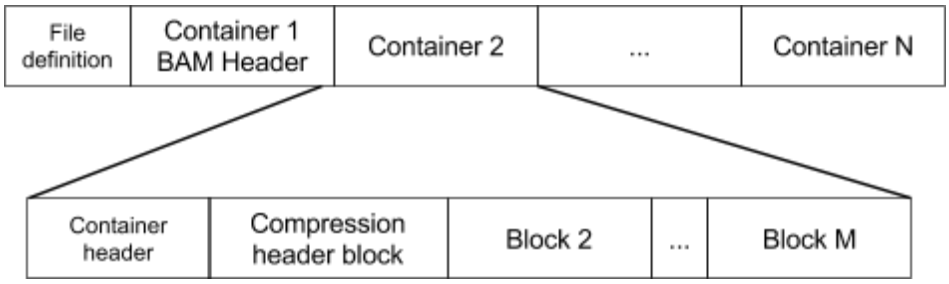
Pic.2 The BAM header is stored in the first container.

Each container starts with a container header followed by one or more blocks. Each block starts with a block header. All data in CRAM is stored within blocks after the block header.



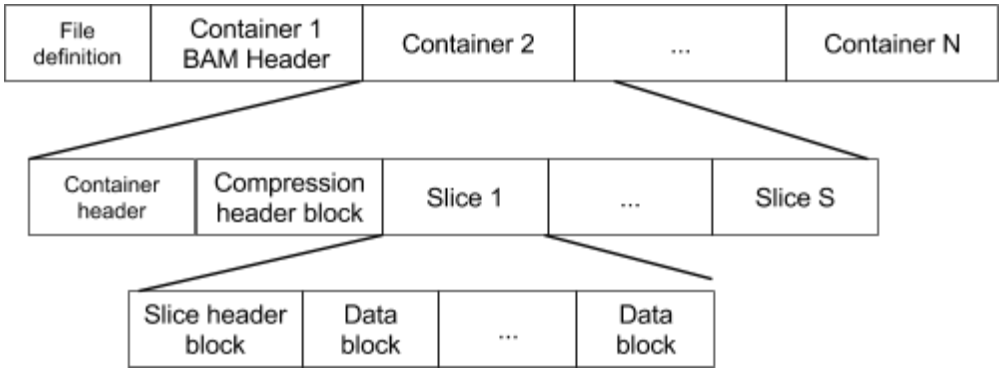
Pic.3 Container and block structure. All data in CRAM files is stored in blocks.

The first block in each container is the compression header block:



Pic.4 Compression header is the first block in the container.

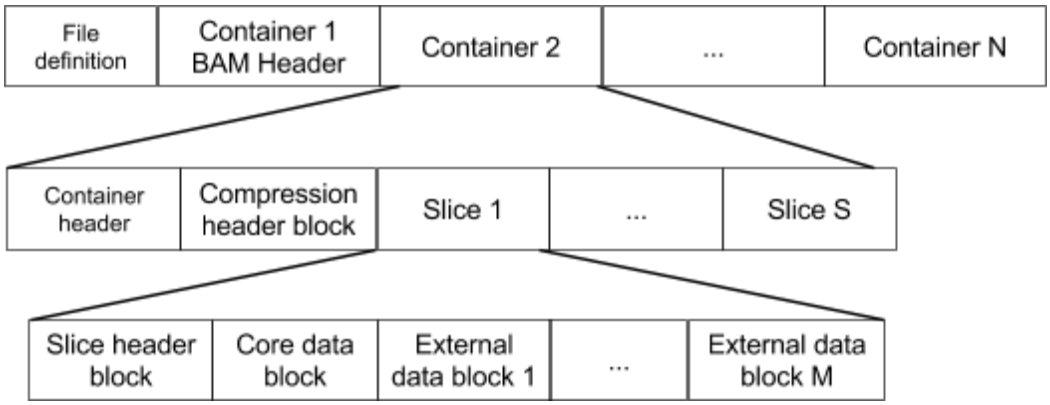
The blocks after the compression header are organised logically into slices. One slice may contain, for example, a contiguous region of alignment data. Slices begin with a slice header block and are followed by one or more data blocks:



Pic.5 Containers are logically organised into slices.

Data blocks are divided into core and external data blocks. Each slice must have at least one core data block immediately after the slice header block. The core data block may be followed by one or more external data

blocks.



Pic.5 Data blocks are divided into core and external data blocks.

6 File definition

Each CRAM file starts with a fixed length (26 bytes) definition with the following fields:

Data type	Name	Value
byte[4]	format magic number	CRAM (0x43 0x52 0x41 0x4d)
unsigned byte	major format number	2 (0x2)
unsigned byte	minor format number	0 (0x0)
byte[20]	file id	CRAM file identifier (e.g. file name or SHA1 checksum)

7 Container structure

The file definition is followed by one or more containers with the following header structure where the container content is stored in the 'blocks' field:

Data type	Name	Value
int32	length	byte size of the container data (blocks)
itf8	reference sequence id	reference sequence identifier or -1 for unmapped reads -2 for multiple reference sequences
itf8	starting position on the reference	the alignment start position or 0 for unmapped reads
itf8	alignment span	the length of the alignment or 0 for unmapped reads
itf8	number of records	number of records in the container
ltf8	record counter	1-based sequential index of records in the file/stream.
ltf8	bases	number of read bases
itf8	number of blocks	the number of blocks
itf8[ ]	landmarks	Each integer value of this array is a byte offset into the blocks byte array. Landmarks are used for random access indexing.
int	crc32	CRC32 hash of the all the preceding bytes in the container.
byte[ ]	blocks	The blocks contained within the container.

7.1 CRAM header in the first container

The first container in the CRAM file contains the BAM header in an optionally compressed block. Only gzip is allowed as compression method for this block. BAM header is terminated with \0 byte and any extra bytes in the block can be used to expand the BAM header. For example when updating @SQ records additional space

may be required for the BAM header. It is recommended to reserve 50% more space in the CRAM header block than it is required by the BAM header.

8 Block structure

Containers consist of one or more blocks. Block compression is applied independently and in addition to any encodings used to compress data within the block. The block have the following header structure with the data stored in the 'block data' field:

Data type	Name	Value
byte	method	the block compression method: 0: raw (none)* 1: gzip 2: bzip2 3: lzma 4: rans
byte	block content type id	the block content type identifier
itf8	block content id	the block content identifier used to associate external data blocks with data series
itf8	size in bytes*	size of the block data after applying block compression
itf8	raw size in bytes*	size of the block data before applying block compression
byte[ ]	block data	the data stored in the block: • bit stream of CRAM records (core data block) • byte stream (external data block) • additional fields (header blocks)
byte[4]	CRC32	CRC32 hash value for all preceding bytes in the block

* Note on raw method: both compressed and raw sizes must be set to the same value.

8.1 Block content types

CRAM has the following block content types:

Block content type	Block content type id	Name	Contents
FILE_HEADER	0	BAM header block	BAM header
COMPRESSION_HEADER	1	Compression header block	See specific section
MAPPED_SLICE_HEADER	2	Slice header block	See specific section
	3		reserved
EXTERNAL_DATA	4	external data block	data produced by external encodings
CORE_DATA	5	core data block	bit stream of all encodings except for external

8.2 Block content id

Block content id is used to distinguish between external blocks in the same slice. Each external encoding has an id parameter which must be one of the external block content ids. For external blocks the content id is a positive integer. For all other blocks content id should be 0. Consequently, all external encodings must not use content id less than 1.

Data blocks

Data is stored in data blocks. There are two types of data blocks: core data blocks and external data blocks.The difference between core and external data blocks is that core data blocks consist of data series that are compressed

using bit encodings while the external data blocks are byte compressed. One core data block and any number of external data blocks are associated with each slice.

Writing to and reading from core and external data blocks is organised through CRAM records. Each data series is associated with an encoding. In case of external encoding the block content id is used to identify the block where the data series is stored. Please note that external blocks can have multiple data series associated with them; in this case the values from these data series will be interleaved.

8.3 BAM header block

The BAM header is stored in a single block within the first container.

The following constraints apply to the BAM header:

- The SQ:MD5 checksum is required unless the reference sequence has been embedded into the file.
- At least one RG record is required.
- The HD:SO sort order is always POS.

8.4 Compression header block

The compression header block consists of 3 parts: preservation map, data series encoding map and tag encoding map.

Preservation map

The preservation map contains information about which data was preserved in the CRAM file. It is stored as a map with byte[2] keys:

Key	Value data type	Name	Value
RN	bool	read names included	true if read names are preserved for all reads
AP	bool	AP data series delta	true if AP data series is delta, false otherwise
RR	bool	reference required	true if reference sequence is required to restore the data completely
SM	byte[5]	substitution matrix	substitution matrix
TD	byte[ ]	tag ids dictionary	a list of lists of tag ids, see tag encoding section
BD	int	base digest	sum over all reads of CRC32 hashes of bases.
SD	int	score digest	sum over all reads of CRC32 hashes of quality scores.

Data series encodings

Each data series has an encoding. These encoding are stored in a map with byte[2] keys:

Key	Value data type	Name	Value
BF	encoding<int>	bit flags	see separate section
AP	encoding<int>	in-seq positions	0-based alignment start delta from previous record *
FP	encoding<int>	in-read positions	positions of the read features
RL	encoding<int>	read lengths	read lengths
DL	encoding<int>	deletion lengths	base-pair deletion lengths
NF	encoding<int>	distance to next fragment	number of records to the next fragment*
BA	encoding<byte>	bases	bases
QS	encoding<byte>	quality scores	quality scores
FC	encoding<byte>	read features codes	see separate section
FN	encoding<int>	number of read features	number of read features in each record
BS	encoding<byte>	base substitution codes	base substitution codes
IN	encoding<byte[ ]>	insertion	inserted bases
RG	encoding<int>	read groups	read groups. Special value '-1' stands for no group.
MQ	encoding<int>	mapping qualities	mapping quality scores
TL	encoding<int>	tag ids	list of tag ids, see tag encoding section
RN	encoding<byte[ ]>	read names	read names
NS	encoding<int>	next fragment reference sequence id	reference sequence ids for the next fragment
NP	encoding<int>	next mate alignment start	alignment positions for the next fragment
TS	encoding<int>	template size	template sizes
MF	encoding<int>	next mate bit flags	see specific section
CF	encoding<int>	compression bit flags	see specific section
TM	encoding<int>	test mark	a prefix expected before every record, for debugging purposes.
RI	encoding<int>	reference id	record reference id from the BAM file header
RS	encoding<int>	reference skip length	number of skipped bases for the 'N' read feature
PD	encoding<int>	padding	number of padded bases
HC	encoding<int>	hard clip	number of hard clipped bases
SC	encoding<byte[ ]>	soft clip	soft clipped bases

* The data series is reset for each slice.

Encoding tags

The TL (tag list) data series represents combined information about the number of tags in a record and their ids.

Let $L_i = \{T_{i0}, T_{i1}, \ldots, T_{ix}\}$ be sorted list of all tag ids for a record R_i, where i is the sequential record index and T_{ij} denotes j-th tag id in the record. We recommend alphabetical sort order. The list of unique L_i is assigned sequential integer numbers starting with 0. These integer numbers represent the TL data series. The sorted list of unique L_i is stored as the TD value in the preservation map. Using TD, an integer from the TL data series can be mapped back into a list of tag ids.

The TD is written as byte array consisting of L_i values separated with \0. Each L_i value is written as a sequence of 3 bytes: tag id followed by tag value type. For example AMiOQz\0OQz\0, where the TD consists of just two values: integer 0 for tags {AM:i,OQ:z} and 1 for tag {OQ:z}.

Encoding tag values

The encodings used for different tags are stored in a map. The map has integer keys composed of the two letter tag abbreviation followed by the tag type as defined in the SAM specification, for example 'OQZ' for 'OQ:Z'.

The three bytes form a big endian integer and are written as ITF8. For example, 3-byte representation of OQ:Z is {0x4F, 0x51, 0x5A} and these bytes are intepreted as the integer 0x004F515A. The integer is finally written as ITF8.

Key	Value data type	Name	Value
TAG NAME 1:TAG TYPE 1	encoding<byte[ ]>	read tag 1	tag values (names and types are available in the data series code)
...		...	...
TAG NAME N:TAG TYPE N	encoding<byte[ ]>	read tag N	...

Note that tag values are encoded as array of bytes. The routines to convert tag values into byte array and back are the same as in BAM with the exception of value type being captured in the tag key rather in the value.

8.5 Slice header block

The slice header block is never compressed (block method=raw). For reference mapped reads the slice header also defines the reference sequence context of the data blocks associated with the slice. Mapped and unmapped reads can be stored within the same slice similarly to BAM file. Slices with unsorted reads must not contain any other types of reads.

The slice header block contains the following fields.

Data type	Name	Value
itf8	reference sequence id	reference sequence identifier or -1 for unmapped or unsorted reads
itf8	alignment start	the alignment start position or -1 for unmapped or unsorted reads
itf8	alignment span	the length of the alignment or 0 for unmapped or unsorted reads
itf8	number of records	the number of records in the slice
ltf8	record counter	1-based sequential index of records in the file/stream
itf8	number of blocks	the number of blocks in the slice
itf8[ ]	block content ids	block content ids of the blocks in the slice
itf8	embedded reference bases block content id	block content id for the embedded reference sequence bases or -1 for none
byte[16]	reference md5	MD5 checksum of the reference bases within the slice boundaries or 16 \0 bytes for unmapped or unsorted reads

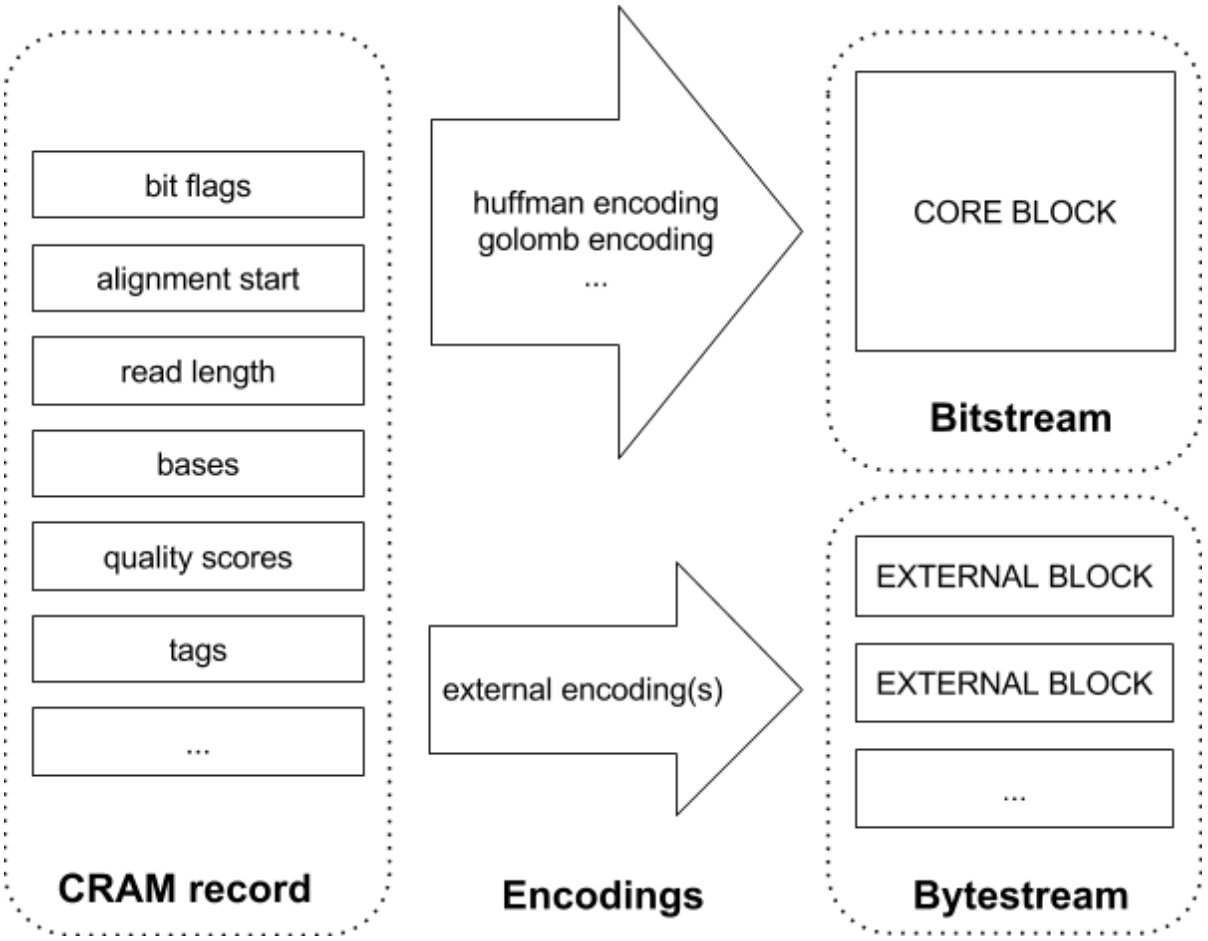
8.6 Core data block

A core data block is a bit stream (most significant bit first) consisting of one or more CRAM records. Please note that one byte could hold more then one CRAM record as a minimal CRAM record could be just a few bits long. The core data block has the following fields:

Data type	Name	Value
bit[ ]	CRAM record 1	The first CRAM record
...	...	...
bit[ ]	CRAM record N	The Nth CRAM record

8.7 External data block

Relationship between core data block and external data blocks is shown in the following picture:



Pic.3 Relationship between core data block and external data blocks.

The picture shows how a CRAM record (on the left) is partially written to core data block while the other fields are stored in two external data blocks. The specific encodings are presented only for demonstration purposes, the main point here is to distinguish between bit encodings whose output is always stored in core data block and the external encoding which simply stored the bytes into external data blocks.

9 End of file marker

A special container is used to mark the end of a file or stream. It is required in version 3 or later. The idea is to provide an easy and a quick way to detect that a CRAM file or stream is complete. The marker is basically an empty container with ref seq id set to -1 (unaligned) and alignment start set to 4542278.

Here is a complete content of the EOF container explained in detail:

hex bytes	data type	decimal value	field name
Container header			
0f 00 00 00	integer	15	size of blocks data
ff ff ff ff f0	itf8	-1	ref seq id
e0 45 4f 46	itf8	4542278	alignment start
00	itf8	0	alignment span
00	itf8	0	nof records
00	itf8	0	global record counter
00	itf8	0	bases
01	itf8	1	block count
00	array	0	landmarks
05 bd d9 4f	integer	1339669765	container header CRC32
Compression header block			
00	byte	0 (RAW)	compression method
01	byte	1 (COMPRESSION_HEADER)	block content type
00	itf8	0	block content id
06	itf8	6	compressed size
06	itf8	6	uncompressed size
Compression header			
01	itf8	1	preservation map byte size
00	itf8	0	preservation map size
01	itf8	1	encoding map byte size
00	itf8	0	encoding map size
01	itf8	1	tag encoding byte size
00	itf8	0	tag encoding map size
ee 63 01 4b	integer	1258382318	block CRC32

When compiled together the EOF marker is 38 bytes long and in hex representation is:

0f 00 00 00 ff ff ff ff 0f e0 45 4f 46 00 00 00 00 01 00 05 bd d9 4f 00 01 00 06 06 01 00 01 00 01 00 ee 63 01 4b

10 Record structure

CRAM record is based on the SAM record but has additional features allowing for more efficient data storage. In contrast to BAM record CRAM record uses bits as well as bytes for data storage. This way, for example, various coding techniques which output variable length binary codes can be used directly in CRAM. On the other hand, data series that do not require binary coding can be stored separately in external blocks with some other compression applied to them independently.

10.1 CRAM record

Both mapped and unmapped reads start with the following fields. Please note that the data series type refers to the logical data type and the data series name corresponds to the data series encoding map.

	Data series type	Data series name	Field	Description
1	int	BF	CRAM bit flags	see CRAM record bit flags
2	int	CF	compression bit flags	see compression bit flags
3	int	RI	ref id	reference sequence id, not used for single reference slices, reserved for future multiref slices.
4	int	RL	read length	the length of the read
5	int	AP	alignment start	the alignment start position *1
6	int	RG	read group	the read group identifier
7	byte	QS	quality scores	quality scores are stored depending on the value of the 'mapped QS included' field
8	byte[ ]	RN	read name	the read names (if preserved)
9	*2	*2	mate record	*2 (if not the last record)
10	int	TL	tag ids	tag ids *3
11	byte[ ]	-	tag values	tag values *3

*1 The AP data series is delta encoded for reads mapped to a single reference slice and normal integer value in all other cases.

*2 See mate record section.

*3 See tag encoding section.

The CRAM record structure for mapped reads has the following additional fields:

	Data series type	Data series name	Field	Description
1	*1	*1	read feature records	*1
2	byte	MQ	mapping quality	read mapping quality

*1 See read feature record specification below.

The CRAM record structure for unmapped reads has the following additional fields:

	Data series type	Data series name	Field	Description
1	byte[read length]	BA	bases	the read bases

10.2 Read bases

CRAM format supports ACGTN bases only. All non-ACGTN read bases must be replaced with N (unknown) base. In case of mismatching non-ACGTN read base and non-ACGTN reference base a ReadBase read feature should be used to capture the fact that the read base should be restored as N base.

10.3 CRAM record bit flags (BF data series)

The following flags are defined for each CRAM read record:

Bit flag	Comment	Description
0x1	! 0x40 && ! 0x80	template having multiple segments in sequencing
0x2		each segment properly aligned according to the aligner
0x4		segment unmapped
0x8	calculated* or stored in the mate's info	next segment in the template unmapped
0x10		SEQ being reverse complemented
0x20	calculated* or stored in the mate's info	SEQ of the next segment in the template being reversed
0x40		the first segment in the template
0x80		the last segment in the template
0x100		secondary alignment
0x200		not passing quality controls
0x400		PCR or optical duplicate

* For segments within the same slice.

10.4 Read feature records

Read features are used to store read details that are expressed using read coordinates (e.g. base differences respective to the reference sequence). The read feature records start with the number of read features followed by the read features themselves:

	Data series type	Data series name	Field	Description
1	int	FN	number of read features	the number of read features
2*1	int	FP	in-read-position	position of the read feature
3*1	byte	FC	read feature code	*2
4*1	*2	*2	read feature data	*2

*1 Repeated for each read feature.

*2 See read feature codes below.

Read feature codes

The following codes are used to distinguish variations in read coordinates:

Feature code	Id	Data series type	Data series name	Description
Bases	b (0x62)	byte[ ]	BA	a stretch of bases
Scores	q (0x71)	byte[ ]	QS	a stretch of scores
Bases and scores	A (0x41)	byte[ ],byte[ ]	BA,QS	A a stretch of bases and quality scores score
Read base	B (0x42)	byte,byte	BA,QS	A base and associated quality score
Substitution	X (0x58)	byte	BS	base substitution codes, SAM operators X, M and =
Insertion	I (0x49)	byte[ ]	IN	inserted bases, SAM operator I
Deletion	D (0x44)	int	DL	number of deleted bases, SAM operator D
Insert base	i (0x69)	byte	BA	single inserted base, SAM operator I
Quality score	Q (0x51)	byte	QS	single quality score
Reference skip	N (0x4E)	int	RS	number of skipped bases, SAM operator N
Soft clip	S	byte[ ]	SC	soft clipped bases, SAM operator S
Padding	P	int	PD	number of padded bases, SAM operator P
Hard clip	H	int	HC	number of hard clipped bases, SAM operator H

Base substitution codes (BS data series)

A base substitution is defined as a change from one nucleotide base (reference base) to another (read base) including N as an unknown or missing base. There are 5 possible bases ACGTN, 4 possible substitutions for each base and 20 substitutions in total. Substitutions for the same reference base are assigned integer codes from 0 to 3 inclusive. To restore a base one would need to know its substitution code and the reference base.

A base substitution matrix assigns integer codes to all possible substitutions.

Substitution matrix is written as follows. Substitutions for a given reference base are sorted by their frequencies in descending order then assigned numbers from 0 to 3. Same-frequency ties are broken using alphabetical order. For example, let us assume the following substitution frequencies for base A:

AC: 15%

AG: 25%

AT: 55%

AN: 5%

Then the substitution codes are:

AC: 2

AG: 1

AT: 0

AN: 3

and they are written as a single byte, 10 01 00 11 = 147 decimal or 0x93 in this case. The whole substitution matrix is written as 5 bytes, one for each reference base in the alphabetical order: A, C, G, T and N.

Note: the last two bits of each substitution code are redundant but still required to simplify the reading.

10.5 Mate record

There are two ways in which mate information can be preserved in CRAM: number of records downstream (distance) to the next fragment in the template and a special mate record if the next fragment is not in the current slice. Combination of the two approaches allows to fully restore BAM level mate information and efficiently store it in the CRAM file.

For mates within the slice only the distance is captured:

	Data series type	**Data series name**	**Description**
1	int	NF	the number of records to the next fragment

If the next fragment is not found within the horizon then the following structure is included into the CRAM record:

	Data series type	**Data series name**	**Description**
1	byte	MF	next mate bit flags, see table below
2	byte[ ]	RN	the read name
3	int	NS	mate reference sequence identifier
4	long	NP	mate alignment start position
5	int	TS	the size of the template (insert size)

Next mate bit flags (MF data series)

The next mate bit flags expressed as an integer represent the MF data series. The following bit flags are defined:

Bit flag	**Name**	**Description**
0x1	mate negative strand bit	the bit is set if the mate is on the negative strand
0x2	mate mapped bit	the bit is set if the mate is mapped

Read names (RN data series)

Read names can be preserved in the CRAM format. However, it is anticipated that in the majority of cases original read names will not be preserved and sequential integer numbers will be used as read names. Read names may also be used to associate fragments into templates when the fragments are too far apart to be referenced by the number of CRAM records. In this case the read names are not required to be the same as the original ones. Their only two requirements are:

- read name must be the same for all fragments of the same template
- read name of a template must be unique within a file

10.6 Compression bit flags (CF data series)

The compression bit flags expressed as an integer represent the CF data series. The following compression flags are defined for each CRAM read record:

Bit flag	**Name**	**Description**
0x1	quality scores stored as array	quality scores can be stored as read features or as an array similar to read bases.
0x2	detached	the next segment is out of horizon
0x4	has mate downstream	tells if the next segment should be expected further in the stream
0x8	decode sequence as "*"	informs the decoder that the sequence is unknown and that any encoded reference differences are present only to recreate the CIGAR string.

11 Reference sequences

CRAM format is natively based upon usage of reference sequences even though in some cases they are not required. In contrast to BAM format CRAM format has strict rules about reference sequences.

1. M5 (sequence MD5 checksum) field of @SQ sequence record in the BAM header is required and UR (URI for the sequence fasta optionally gzipped file) field is strongly advised. The rule for calculating MD5 is to remove any non-base symbols (like \n, sequence name or length and spaces) and upper case the rest. Here are some examples:

   ```
   > samtools faidx human_g1k_v37.fasta 1 | grep -v '^>' | tr -d '\n' | tr a-z A-Z | md5sum
   -
   1b22b98cdeb4a9304cb5d48026a85128 -
   > samtools faidx human_g1k_v37.fasta 1:10-20 |grep -v '^>' |tr -d '\n' |tr a-z A-Z |md5sum
   -
   0f2a4865e3952676ffad2c3671f14057 -
   ```

   Please note that the latter calculates the checksum for 11 bases from position 10 (inclusive) to 20 (inclusive) and the bases are counted 1-based, so the first base position is 1.

2. All CRAM reader implementations are expected to check for reference MD5 checksums and report any missing or mismatching entries. Consequently, all writer implementations are expected to ensure that all checksums are injected or checked during compression time.

3. In some cases reads may be mapped beyond the reference sequence. All out of range reference bases are all assumed to be 'N'.

4. MD5 checksum bytes in slice header should be ignored for unmapped or multiref slices.

12 Indexing

General notes

Please note that CRAM indexing is external to the file format itself and may change independently of the file format specification in the future. For example, a new type of index files may appear.

Individual records are not indexed in CRAM files, slices should be used instead as a unit of random access. Another important difference between CRAM and BAM indexing is that CRAM container header and compression header block (first block in container) must always be read before decoding a slice. Therefore two read operations are required for random access in CRAM.

Indexing a CRAM file is deemed to be a lightweight operation because it does not require any CRAM records to be read. All indexing information can be obtained from container headers, namely sequence id, alignment start and span, container start byte offset and slice byte offset inside the container.

CRAM index

A CRAM index is a gzipped tab delimited file containing the following columns:

1. Sequence id
2. Alignment start
3. Alignment span
4. Container start byte offset in the file
5. Slice start byte offset in the container data ('blocks')
6. Slice bytes

Each line represents a slice in the CRAM file. Please note that all slices must be listed in index file.

BAM index

BAM indexes are supported by using 4-byte integer pointers called landmarks that are stored in container header. BAM index pointer is a 64-bit value with 48 bits reserved for the BAM block start position and 16 bits reserved for the in-block offset. When used to index CRAM files, the first 48 bits are used to store the CRAM container start position and the last 16 bits are used to store the index of the landmark in the landmark array stored in container header. The landmark index can be used to access the appropriate slice.

The above indexing scheme treats CRAM slices as individual records in BAM file. This allows to apply BAM indexing to CRAM files, however it introduces some overhead in seeking specific alignment start because all preceding records in the slice must be read and discarded.

13 Appendix

13.1 External encoding

External encoding operates on bytes only. Therefore any data series must be translated into bytes before sending data into an external block. The following agreements are defined.

Integer values are written as ITF8, which then can be translated into an array of bytes.

Strings, like read name, are translated into bytes according to UTF8 rules. In most cases these should coincide with ASCII, making the translation trivial.

13.2 rANS codec

rANS is the range-coder variant of the Asymmetric Numerical System[1].

The structure of the external rANS codec consists of several components: meta-data consisting of compression-order, and compressed and uncompressed sizes; normalised frequencies of the alphabet systems to be encoded, either in Order-0 or Order-1 context; and the rANS encoded byte stream itself.

Here "Order" refers to the number of bytes of context used in computing the frequencies. It will be 0 or 1. Ignoring punctuation and space, an Order-0 analysis of English text may observe that 'e' is the most common letter (12-13%), and that 'u' occurs only around 2.5% of the time. If instead we consider the frequency of a letter in the context of one previous letter (Order-1) then these statistics change considerably; we know that if the previous letter was 'q' then 'e' becomes a rare letter while 'u' is the most likely.

These observed frequencies are directly related to the amount of storage required to encode a symbol (e.g. an alphabet letter)[2].

13.2.1 rANS compressed data structure

A compressed data block consists of the following logical parts:

[1] J. Duda, *Asymmetric numeral systems: entropy coding combining speed of Huffman coding with compression rate of arithmetic coding*, `http://arxiv.org/abs/1311.2540`

[2] C.E. Shannon, *A Mathematical Theory of Communication*, Bell System Technical Journal, vol. 27, pp. 379-423, 623-656, July, October, 1948

Value data type	Name	Description
byte	order	the order of the codec, either 0 or 1
int	compressed size	the size in bytes of frequency table and compressed blob
int	data size	raw or uncompressed data size in bytes
byte[]	frequency table	byte frequencies of input data written using RLE
byte[]	compressed blob	compressed data

13.2.2 Frequency table

The alphabet used here is simply byte values, so a maximum of 256 symbols as some values may not be present.

The symbol frequency table indicates which symbols are present and what their relative frequencies are. The total sum of symbol frequencies are normalised to add up to 4095.

Formally, this is an ordered alphabet $\mathbb{A}$ containing symbols s where s_i with the i-th symbol in $\mathbb{A}$, occuring with the frequency $freq_i$.

Order-0 encoding

The normalised symbol frequencies are then written out as {symbol, frequency} pairs in ascending order of symbol (0 to 255 inclusive). If a symbol has a frequency of 0 then it is omitted.

To avoid storing long consecutive runs of symbols if all are present (eg a-z in a long piece of English text) we use run-length-encoding on the alphabet symbols. If two consecutive symbols have non-zero frequencies then a counter of how many other non-zero frequency consecutive symbols is output directly after the second consecutive symbol, with that many symbols being subsequently omitted.

For example for non-zero frequency symbols ‘a’, ‘b’, ‘c’, ‘d’ and ‘e’ we would write out symbol ‘a’, ‘b’ and the value 3 (to indicate ‘c’, ‘d’ and ‘e’ are also present).

The frequency is output after every symbol (whether explicit or implicit) using ITF8 encoding. This means that frequencies 0-127 are encoded in 1 byte while frequencies 128-4095 are encoded in 2 bytes.

Finally the symbol 0 is written out to indicate the end of the symbol-frequency table.

As an example, take the string `abracadabra`.

Symbol frequency:

Symbol	Frequency
a	5
b	2
c	1
d	1
r	2

Normalised to sum to 4095:

Symbol	Frequency
a	1863
b	744
c	372
d	372
r	744

Encoded as:

```
0x61      0x87 0x47      # ‘a’           <1863>
0x62 0x02 0x82 0xe8      # ‘b’ <+2: c,d>  <744>
          0x81 0x74      # ‘c’ (implicit) <372>
          0x81 0x74      # ‘d’ (implicit) <372>
0x72      0x82 0xe8      # ‘r’            <744>
0x00                     # <0>
```

Order-1 encoding

To encode Order-1 statistics typically requires a larger table as for an N sized alphabet we need to potentially store an NxN matrix. We store these as a series of Order-0 tables.

We start with the outer context byte, emitting the symbol if it is non-zero frequency. We perform the same run-length-encoding as we use for the Order-0 table and end the contexts with a nul byte. After each context byte we emit the Order-0 table relating to that context.

One last caveat is that we have no context for the first byte in the data stream (infact for 4 equally spaced starting points, see "interleaving" below). We use the ASCII value ('\0') as the starting context and so need to consider this in our frequency table.

Consider `abracadabraabracadabraabracadabraabracadabra` as example input.

Observed Order-1 frequencies:

Context	Symbol	Frequency
\0	a	4
a	a	3
	b	8
	c	4
	d	4
b	r	8
c	a	4
d	a	4
r	a	8

Normalised (per Order-0 statistics):

Context	Symbol	Frequency
\0	a	4095
a	a	646
	b	1725
	c	862
	d	862
b	r	4095
c	a	4095
d	a	4095
r	a	4095

Encoded as:

```
0x00                # '\0' context
0x61      0x8f 0xff  # a  <4095>
0x00                # end of Order-0 table

0x61                # 'a' context
0x61      0x82 0x86  # a            <646>
0x62 0x02 0x86 0xbd  # b <+2: c,d> <1725>
          0x83 0x5e  # c (implicit) <862>
          0x83 0x5e  # d (implicit) <862>
0x00                # end of Order-0 table

0x62 0x02           # 'b' context, <+2: c, d>
0x72      0x8f 0xff  # r <4095>
0x00                # end of Order-0 table

                    # 'c' context (implicit)
0x61      0x8f 0xff  # a <4095>
0x00                # end of Order-0 table

                    # 'd' context (implicit)
0x61      0x8f 0xff  # a <4095>
0x00                # end of Order-0 table

0x72                # 'r' context
0x61      0x8f 0xff  # a <4095>
0x00                # end of Order-0 table

0x00                # end of contexts
```

13.2.3 rANS entropy encoding

The encoder takes a symbol s and a current state x (initially zero) to produce a new state x' with function C.

$$x' = C(s, x)$$

The decoding function D is the inverse of C such that $C(D(x)) = x$.

$$D(x') = (s, x)$$

The entire encoded message can be viewed as a series of nested C operations, with decoding yielding the symbols in reverse order, much like popping items off a stack. This is where the asymmetric part of ANS comes from.

As we encode into x the value will grow, so for efficiency we ensure that it always fits within known bounds. This is governed by

$$L \leq x < bL - 1$$

where b is the base and L is the lower-bound.

We ensure this property is true before every use of C and after every use of D. Finally to end the stream we flush any remaining data out by storing the end state of x.

Implementation specifics

We use an unsigned 32-bit integer to hold x. In encoding it is initialised to zero. For decoding it is read little-endian from the input stream.

Recall $freq_i$ is the frequency of the i-th symbol s_i in alphabet $\mathbb{A}$. We define $cfreq_i$ to be cumulative frequency of all symbols up to but not including s_i:

$$cfreq_i = \begin{cases} 0 & \text{if } i < 1 \\ cfreq_{i-1} + freq_{i-1} & \text{if } i \geq 1 \end{cases}$$

We have a reverse lookup table $cfreq\_to\_sym_c$ from 0 to 4095 (0xfff) that maps a cumulative frequency c to a symbol s.

$$cfreq\_to\_sym_c = s_i \quad | \quad c: \ cfreq_i \leq c < cfreq_i + freq_i$$

The $x' = C(s, x)$ function used for the $i - thsymbol$s is:

$$x' = (x/freq_i) * \texttt{0x1000} + cfreq_i + (x\%freq_i)$$

The $D(x') = (s, x)$ function used to produce the i-th symbol s and a new state x is:

$$c = x' \& \texttt{0xfff}$$
$$s_i = cfreq\_to\_sym_c$$
$$x = freq_i(x'/\texttt{0x1000}) + c - cfreq_i$$

Most of these operations can be implemented as bit-shifts and bit-AND, with the encoder modulus being implemented as a multiplication by the reciprocal, computed once only per alphabet symbol.

We use $L = \texttt{0x800000}$ and $b = 256$, permitting us to flush out one byte at a time (encoded and decoded in reverse order).

Before every encode $C(s, x)$ we renormalise x, shifting out the bottom 8 bits of x until $x < \texttt{0x80000} * freq_i$. After finishing encoding we flush 4 more bytes (lowest 8-bits first) from x.

After every decoded $D(x')$ we renormalise x', shifting in the bottom 8 bits until $x \geq \texttt{0x800000}$.

Interleaving

For efficiency, we interleave 4 separate rANS codecs at the same time[3]. For the Order-0 codecs these simply encode or decode the 4 neighbouring bytes in cyclic fashion using interleaved codec 1, 2, 3 and 4, sharing the same output buffer (so the output bytes get interleaved).

For the Order-1 codec we cannot do this as we need to know the previous byte value as the context for the next byte. Therefore split the input data into 4 approximately equal sized fragments[4] starting at 0, $\lfloor len/4 \rfloor * 1$, $\lfloor len/4 \rfloor * 2$ and $\lfloor len/4 \rfloor * 3$. Each Order-1 codec operates in a cyclic fashion as with Order-0, all starting with 0 as their state and sharing the same output buffer. Any remainder, when the input buffer is not divisible by 4, is encoded at the end by the 4th encoder.

We do not permit Order-1 encoding of data streams smaller than 4 bytes.

[3] F. Giesen, *Interleaved entropy coders*, `http://arxiv.org/abs/1402.3392`

[4] This was why the '\0' → 'a' context in the example above had a frequency of 4 instead of 1.

13.3 Codings

Introduction

The basic idea for codings is to efficiently represent some values in binary format. This can be achieved in a number of ways that most frequently involve some knowledge about the nature of the values being encoded, for example, distribution statistics. The methods for choosing the best encoding and determining its parameters are very diverse and are not part of the CRAM format specification, which only describes how the information needed to decode the values should be stored.

Offset

Most of the codings listed below encode positive integer numbers. An integer offset value is used to allow any integer numbers and not just positive ones to be encoded. It can also be used for monotonically decreasing distributions with the maximum not equal to zero. For example, given offset is 10 and the value to be encoded is 1, the actually encoded value would be offset+value=11. Then when decoding, the offset would be subtracted from the decoded value.

Beta coding

Definition

Beta coding is a most common way to represent numbers in binary notation.

Examples

Number	Codeword
0	0
1	1
2	10
4	100

Parameters

CRAM format defines the following parameters of beta coding:

Data type	Name	Comment
itf8	offset	offset is added to each value
itf8	length	the number of bits used

Gamma coding

Definition

Elias gamma code is a prefix encoding of positive integers. This is a combination of unary coding and beta coding. The first is used to capture the number of bits required for beta coding to capture the value.

Encoding

1. Write it in binary.
2. Subtract 1 from the number of bits written in step 1 and prepend that many zeros.
3. An equivalent way to express the same process:
4. Separate the integer into the highest power of 2 it contains ($2N$) and the remaining N binary digits of the integer.

5. Encode N in unary; that is, as N zeroes followed by a one.
6. Append the remaining N binary digits to this representation of N.

Decoding

1. Read and count 0s from the stream until you reach the first 1. Call this count of zeroes N.
2. Considering the one that was reached to be the first digit of the integer, with a value of $2N$, read the remaining N digits of the integer.

Examples

Value	Codeword
1	1
2	010
3	011
4	00100

Parameters

Data type	Name	Comment
itf8	offset	offset is added to each value

Golomb coding

Definition

Golomb encoding is a prefix encoding optimal for representation of random positive numbers following geometric distribution.

1. Fix the parameter M to an integer value.
2. For N, the number to be encoded, find
   (a) quotient $q = \lfloor N/M \rfloor$
   (b) remainder $r = N \bmod M$
3. Generate Codeword
   (a) The Code format : <Quotient Code><Remainder Code>, where
   (b) Quotient Code (in unary coding)
      i. Write a q-length string of 1 bits
      ii. Write a 0 bit
   (c) Remainder Code (in truncated binary encoding)
      i. If M is power of 2, code remainder as binary format. So $log_2(M)$ bits are needed. (Rice code)
      ii. If M is not a power of 2, set $b = \lceil log_2(M) \rceil$
         A. If $r < 2^b - M$ code r as plain binary using $b - 1$ bits.
         B. If $r \geq 2^b$ code the number $r + 2^b$ in plain binary representation using b bits.

Examples

Number	Codeword, M=10
0	0000
4	0100
10	10000
42	11110010

Parameters

Golomb coding takes the following parameters:

Data type	Name	Comment
itf8	offset	offset is added to each value
itf8	M	the golomb parameter (number of bins)

Golomb-Rice coding

Golomb-Rice coding is a special case of Golomb coding when the M parameter is a power of 2. The reason for this coding is that the division operations in Golomb coding can be replaced with bit shift operators.

Subexponential coding

Definition

Subexponential coding is parametrized by a non-nengative integer k. The main feature of the subexponential code is its length. For integers $n < 2k + 1$ the code length increases linearly with n, but for larger n it increases logarithmically.

Encoding

1. Determine the group index i using the following rules:
   (a) if $n < 2^k$, then $i = 0$.
   (b) if $n \geq 2^k$, then determine i such that $2^{i+k-1} \leq n < 2^{i+k}$
2. Form the prefix of i 1s.
3. Insert the separator 0.
4. Form the tail: express the value of $(n - 2^{i+k-1})$ as a $(i + k - 1)$-bit binary number if $i > 0$ and n as a k-bit binary number otherwise.

Decoding

1. Let i be the number of leading 1s (prefix) in the codeword.
2. Form a run of 0s of length
   (a) 0, if $i = 0$
   (b) 2^{i+k-1}, otherwise
3. Skip the next 0 (separator).
4. Compute the length of the tail, c_{tail} as
   (a) k, if $i = 0$
   (b) $k + i - 1$, if $i \geq 1$
5. The next c_{tail} bits are the tail. Form a run of 0s of length represented by the tail.
6. Append 1 to the run of 0s.
7. Go to step 1 to process the next codeword.

Examples

Number	Codeword, k=0	Codeword, k=1	Codeword, k=2
0	0	00	000
1	10	01	001
2	1100	100	010
3	1101	101	011
4	111000	11000	1000
5	111001	11001	1001
6	111010	11010	1010
7	111011	11011	1011
8	11110000	1110000	110000
9	11110001	1110001	110001
10	11110010	1110010	110010

Parameters

Data type	Name	Comment
itf8	offset	offset is added to each value
itf8	k	the order of the subexponential coding

Huffman coding

CRAM uses canonical huffman coding, which requires only bit-lengths of codewords to restore data. The canonical huffman code follows two additional rules: the alphabet has a natural sort order and codewords are sorted by their numerical values. Given these rules and a codebook containing bit-lengths for each value in the alphabet the codewords can be easily restored.

Important note: for alphabets with only one value there is no output bits at all.

Code computation

• Sort the alphabet ascending using bit-lengths and then using numerical order of the values.

• The first symbol in the list gets assigned a codeword which is the same length as the symbol's original codeword but all zeros. This will often be a single zero ('0').

• Each subsequent symbol is assigned the next binary number in sequence, ensuring that following codes are always higher in value.

• When you reach a longer codeword, then after incrementing, append zeros until the length of the new codeword is equal to the length of the old codeword.

Parameters

Data type	Name	Comment
itf8[ ]	alphabet	list of all encoded values
itf8[ ]	bit-lengths	array of bit-lengths for each symbol in the alphabet

Byte array coding

Often there is a need to encode an array of bytes. This can be optimized if the length of the encoded arrays is known. For such cases BYTE_ARRAY_LEN and BYTE_ARRAY_STOP codings can be used.

BYTE_ARRAY_LEN

Byte arrays are captured length-first, meaning that the length of every array is written using an additional encoding. For example this could be a golomb encoding. The parameter for BYTE_ARRAY_LEN are listed

below:

Data type	Name	Comment
encoding<int>	lengths encoding	an encoding describing how the arrays lengths are captured
encoding<byte>	values encoding	an encoding describing how the values are captured

BYTE_ARRAY_STOP

Byte arrays are captured as a sequence of bytes teminated by a special stop byteFor example this could be a golomb encoding. The parameter for BYTE_ARRAY_STOP are listed below:

Data type	Name	Comment
byte	stop byte	a special byte treated as a delimiter
itf8	external id	id of an external block containing the byte stream

13.4 Choosing the container size

CRAM format does not constrain the size of the containers. However, the following should be considered when deciding the container size:

- Data can be compressed better by using larger containers
- Random access performance is better for smaller containers
- Streaming is more convenient for small containers
- Applications typically buffer containers into memory

We recommend 1MB containers. They are small enough to provide good random access and streaming performance while being large enough to provide good compression. 1MB containers are also small enough to fit into the L2 cache of most modern CPUs.

Some simplified examples are provided below to fit data into 1MB containers.

Unmapped short reads with bases, read names, recalibrated and original quality scores

We have 10,000 unmapped short reads (100bp) with read names, recalibrated and original quality scores. We estimate 0.4 bits/base (read names) + 0.4 bits/base (bases) + 3 bits/base (recalibrated quality scores) + 3 bits/base (original quality scores) =~ 7 bits/base. Space estimate is (10,000 * 100 * 7) / 8 / 1024 / 1024 =~ 0.9 MB. Data could be stored in a single container.

Unmapped long reads with bases, read names and quality scores

We have 10,000 unmapped long reads (10kb) with read names and quality scores. We estimate: 0.4 bits/base (bases) + 3 bits/base (original quality scores) =~ 3.5 bits/base. Space estimate is (10,000 * 10,000 * 3.5) / 8 / 1024 / 1024 =~ 42 MB. Data could be stored in 42 x 1MB containers.

Mapped short reads with bases, pairing and mapping information

We have 250,000 mapped short reads (100bp) with bases, pairing and mapping information. We estimate the compression to be 0.2 bits/base. Space estimate is (250,000 * 100 * 0.2) / 8 / 1024 / 1024 =~ 0.6 MB. Data could be stored in a single container.

Embedded reference sequences

We have a reference sequence (10Mb). We estimate the compression to be 2 bits/base. Space estimate is (10000000 * 2 / 8 / 1024 / 1024) =~ 2.4MB. Data could be written into three containers: 1MB + 1MB + 0.4MB.