CRAM format specification

version: 1.0.1

license: Apache 2.0

by: EMBL-EBI

contacts: cram-dev@ebi.ac.uk

Overview

This specification describes the CRAM 1.0 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, the CRAM format 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. External reference sequences introduce the only external dependency into the CRAM format. When external reference sequences cannot be conveniently used the reference sequences also can be embedded within the CRAM files. However, when embedded reference sequences are used then only those reference sequence regions are preserved in CRAM that have reads aligned against them.

File structure

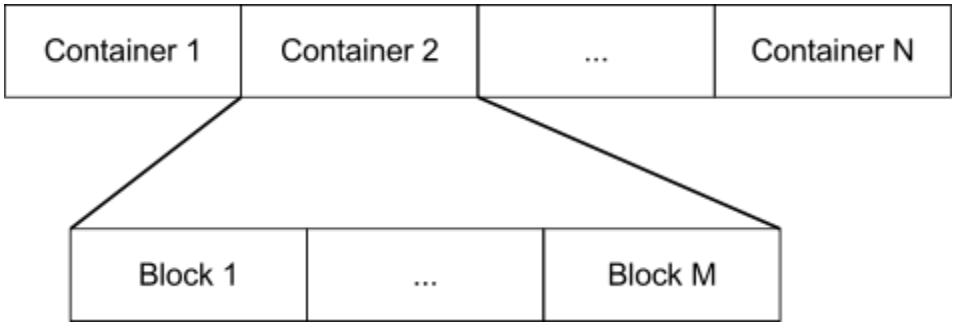
Overview

CRAM file format starts with a fixed length **file definition** followed by containers**.** The first container, the **file header container**, contains the BAM header. The file header container is followed by one or more **data containers**.

Containers and blocks

Data is stored in independent containers, each container consisting of one or more blocks. A typical block holds one or more data series while the container defines the context to which the

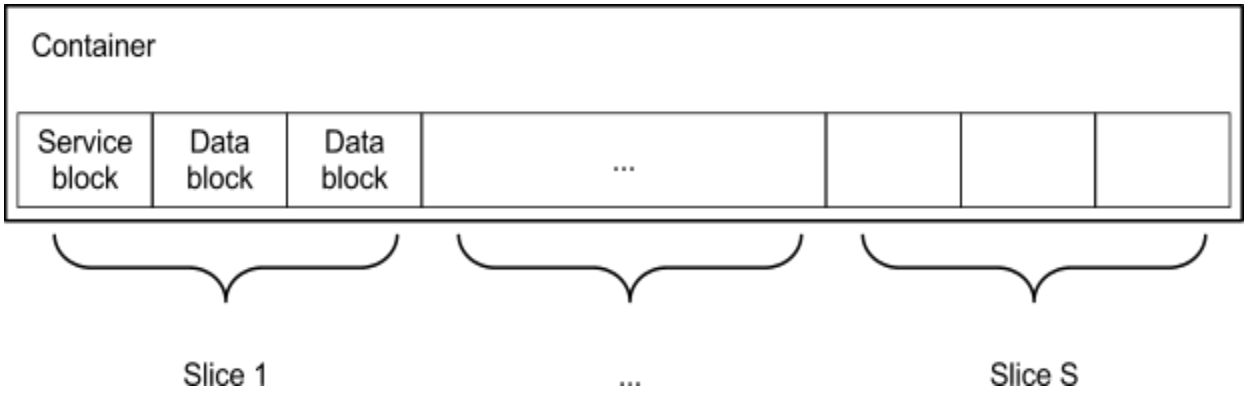
data belongs. For example, a container holding alignment data for a certain region on the reference could contain the following blocks: read names, alignment start positions, bases, quality scores and so on.



Pic.1 Organisation of containers.

Header blocks and data blocks

Data series are stored in data blocks. Header blocks specify logical union of one or more data blocks. Such logical union is called a slice containing, for example, a contiguous region of alignment data.



Pic.2 Container structure and slices.

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.

Logical data types

Boolean

Boolean is either ‘true’ or ‘false’.

Byte

Signed byte (8 bits).

Integer

Signed 32-bit integer.

Long

Signed 64-bit integer.

Array

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

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 0.x7 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.

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 (b)

Boolean is written as 1-byte with 0x0 being ‘false’ and 0x1 being ‘true’.

Short (S)

Signed 16-bit integer, written as 2 bytes in little-endian byte order.

Integer (I)

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

Long (L)

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

ITF-8 integer (i)

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.

For signed integers the first bit after the prefix defines the sign of the value.

LTF-8 long or (l)

See ITF-8 for more details. The only diffrence 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 (i), 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 (i) denotes the codec id and the second integer (i) the number of bytes in the following encoding-specific values.

Subexponential encoding example:

Value	Type	Name
0x7	i	codec id
0x3	i	number of bytes to follow
0x1	i	K parameter
0x0	l	offset
0x1	b	unary bit

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 (i) K, long (l) offset and boolean (b) unary bit:
K = 0x0 0x0 0x0 0x1 = 1
offset = 0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0 = 0
unary bit = 0x1 = true

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 (i). Keys and values are written according to their data types and are specific to each map.

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.

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. The following encodings are supported (please note that both logical and storage data types are provided):

Codec	**Codec Id**	**Parameters**	**Comment**
NULL	0	none	series not preserved
EXTERNAL	1	int (i) block content id	the block content identifier used to associate external data blocks with data series
UNARY	2	long (l) offset, boolean (b) stop bit	unary coding
BETA	3	long (l) offset, int (i) number of bits	binary coding
GAMMA	4	long (l) offset	Elias gamma coding
GOLOMB	5	int (i) M, long (l) offset	Golomb coding
GOLOMB_RICE	6	int (i) log2m, long (l) offset	Golomb-Rice coding
EXP_GOLOMB	7	int (i) k	exponential Golomb coding
SUBEXP	8	int (i) K, long (l) offset	subexponential coding
HUFFMAN_INT	9	int (i) array, int (i) array	coding with int values

BYTE_ARRAY_LEN	10	encoding<int> array length, encoding<byte> bytes	coding of byte arrays with array length
BYTE_ARRAY_STOP	11	byte stop, encoding<byte> bytes	coding of byte arrays with a stop value

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

File definition structure

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

Data type	**Name**	**Value**
byte[4]	format magick	CRAM (0x43 0x52 0x41 0x4d)
unsigned byte	major format number	1 (0x1)
unsigned byte	minor format number	0 (0x0)
byte[20]	file id	CRAM file identifier (e.g. file name or SHA1 checksum)

File header structure

The first container in a CRAM file contains the BAM header in a single block with the following additional constraints:

- 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.

File header can be padded to a certain size by specifying a bigger container size. This may be useful if modifications to the header will be required after the file has been written.

Container structure

The file definition is followed by one or more containers. We advise that each container should only refer to one reference sequence. The only exception are files with a large number of small contigs for example metagenomics sequences.

The container has the following fields:

Data type	Name	Value
i	length	byte size of the container
i	reference sequence id	reference sequence identifier or -1 for unmapped reads -2 for multiple reference sequences
i	reference sequence position	the alignment start position or 0 for unmapped reads
i	alignment length	the length of the alignment or 0 for unmapped reads
i	number of blocks	the number of blocks
byte[]	blocks	The blocks contained within the container.

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 has the following fields:

Data type	Name	Value
byte	method	the block compression method: 0: raw (none) 1: gzip 2: bzip2
byte	block content type id	the block content type identifier
i	block content id	the block content identifier used to associate external data blocks with data series
int	size in bytes	size of the block data after applying block compression
int	raw size in bytes	size of the block data before applying block compression

byte[]	data	the data stored in the block: ● bit stream of CRAM records (core data block) ● byte stream (external data block) ● additional fields (header blocks)

Block content types

CRAM blocks are divided into two groups: header blocks and data blocks. Header blocks have predefined structure and affect the interpretation of data blocks that may follow. The block content type is used to identify the header block type.

Header blocks

The following header blocks are defined in CRAM:

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	2	Slice header block: Mapped reads	See specific section
UNMAPPED_SLICE	3	Slice header block: Unmapped reads	See specific section

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.

Compression header block

The compression header block consists of 4 parts: block fields, preservation map, record encoding map and tag encoding map.

Block fields

The block fields contain general information about the block including reference sequence and alignment position details:

Data type	Name	Value
i	reference sequence id	reference sequence identifier
l	first record alignment start	alignment start position of the first record
i	number of records	the number of records
i[]	landmarks	byte offset within a container to the start of each slice

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
MI	b	mapped QS included	true if all mapped qualities are included
UI	b	unmapped QS included	true if unmapped qualities are included (mapped bit set to false and no reference given)
PI	b	unmapped placed	true if all unmapped

		QS included	qualities included (mapped bit set to false and reference and position given)
SM	byte[5]	substitution matrix	substitution matrix

Record encoding map

The record encoding map contains information about which encodings were used for which data series. It is stored as a map with byte[2] keys:

Key	**Value data type**	**Name**	**Value**
BF	encoding<int>	bit flags	see separate section
AP	encoding<long>	in-seq positions	alignment start positions
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

MQ	encoding<byte>	mapping qualities	mapping quality scores
TC	encoding<byte>	read tag counts	the number of tags in each record
TN	encoding<byte[3]>	read tag name and type	the read tags and types in each record
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<byte>	next mate bit flags	see specific section
CF	encoding<byte>	compression bit flags	see specific section

Tag encoding map

The tag encoding map contains information about which encodings were used for which tags. It is stored as a map with byte[3] keys composed of TAG two letter abbreviation followed by the tag type as defined in the SAM specification (e.g. 'OQZ' for 'OQ:Z').

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.

Slice header block

The slice header block is never compressed (block method=raw). It defines the reference sequence context of the data blocks associated with the slice.

Mapped reads

The slice header block for the mapped reads has the following fields (in addition to the generic block fields):

Data type	Name	Value
i	reference sequence id	reference sequence identifier
i	alignment start	the alignment start position
i	alignment length	the length of the alignment
i	number of records	the number of records in the slice
i	number of blocks	the number of blocks in the slice
i[]	block content ids	block content ids of the blocks in the slice
i	embedded reference bases block content id	block content id for the embedded reference sequence bases

Unmapped reads

The slice header block for the unmapped reads has the following fields (in addition to the generic block fields):

Data type	Name	Value
i	number of records	the number of records in the slice
i	number of blocks	the number of blocks in the slice
i[]	block content ids	block content ids of the blocks in the slice

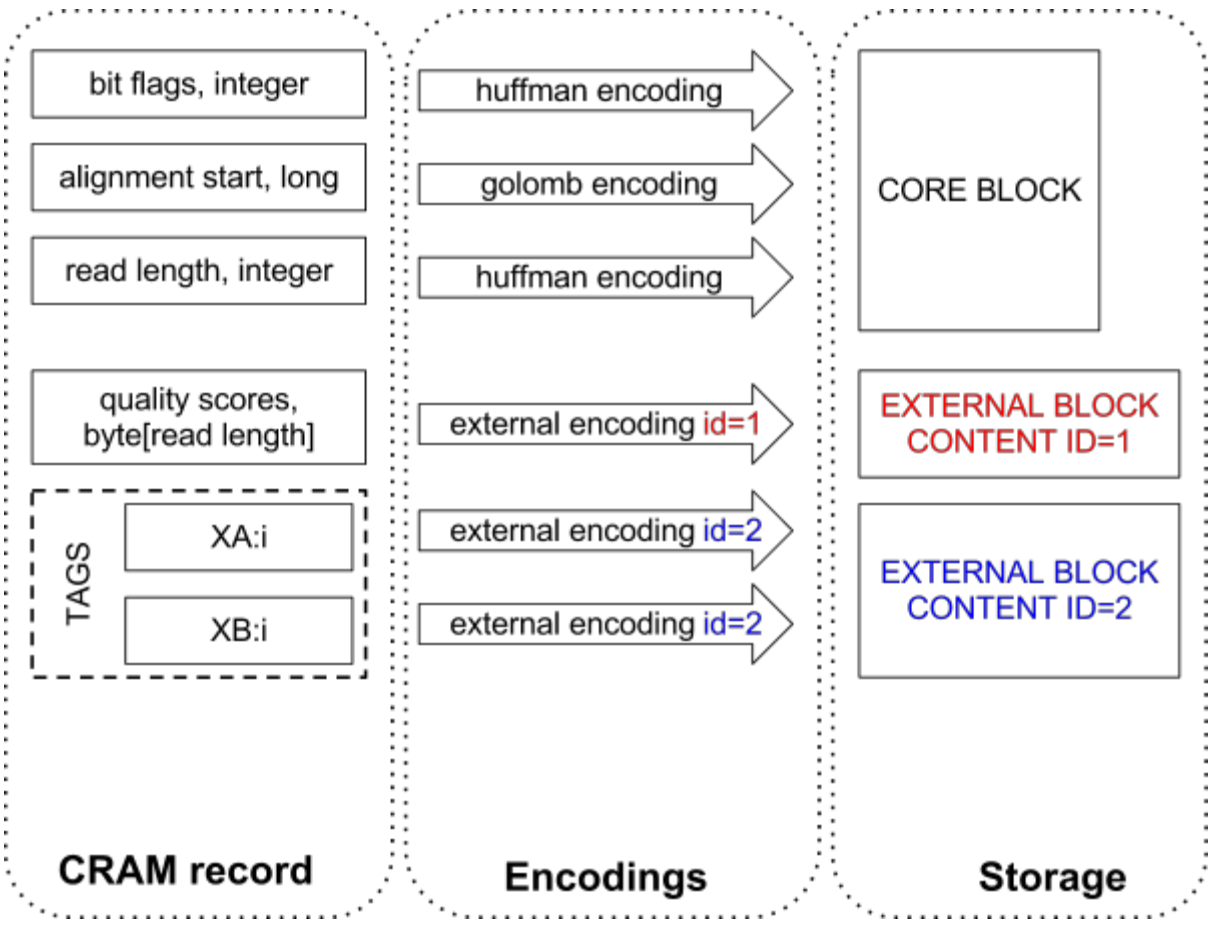
Core data block

A core data block is a bit stream 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 (in addition to the generic block fields):

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

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.

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.

CRAM record bit flags

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
0x800	service bit: next segment is beyond horizon	the next segment info is stored with this records if true

* For segments within horizon only.

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.

	Data series type	**Field**	**Description**
1	int	bit flags	see above
2	int	read length	the length of the read
3	long	alignment start	the alignment start position
4	int	read group	the read group identifier
5	byte	quality scores	quality scores are stored depending on the value of the ‘mapped QS included’ field
6	byte[]	read name	the read names (if preserved)
7	*1	mate record	*1 (if not the last record)
8	*2	tag records	*2

*1 See mate record specification below.
*2 See tag record specification below.

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

	Data series type	**Field**	**Description**
1	*1	read feature records	*1
2	byte	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	Field	Description
1	byte[read length]	bases	the read bases

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:

1	int	number of read features	the number of read features
2 *1	int	in-read-position	position of the read feature
3 *1	byte	read feature code	*2
4 *1	*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 type	Description
ReadBase	B (0x42)	<byte>,<byte>	A base and associated quality score
Substitution	S (0x53)	<byte>	base substitution codes
Insertion	I (0x49)	<byte[]>	inserted bases
Deletion	D (0x44)	<int>	number of deleted bases
InsertBase	i (0x69)	<byte>	single inserted base

BaseQualityScore	Q (0x51)	<byte>	single quality score

Base substitution codes

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.

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 out of scope (horizon), for example, is in a separate container. 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 horizon only distance is captured:

	Data series type	Name	Value
1	int	distance to next fragment	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	Name	Value
1	byte	next mate bit flags	see table below
2	byte[]	read name	the read name
3	int	mate reference	mate reference sequence identifier
4	long	mate alignment start	mate alignment start position
5	int	template size	the size of the template (insert size)

Next mate bit flags

	Data series type	Name	Value
1	bit	mate mapped bit	the bit is set if the mate is mapped
2	bit	mate negative strand bit	the bit is set if the mate is on the negative strand
3	bit	mate first bit	the bit is set if the mate is the first fragment

Read names

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

Tag records

The tag records represent SAM tags:

	Data series type	Name	Value
1	int	number or tags	the number of tags
2 *1	byte[3]	tag name and type	the name and type of the tag using BAM tag specification (e.g. OQ:Z)
3 *1	byte[]	tag value	the tag value

*1 Repeated for each tag.

Compression bit flags

The following compression flags are defined for each CRAM read record:

Bit flag	Comment	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

Indexing

it is possible to skip containers and slices because their size and alignment position are stored uncompressed in the beginning of these structures. This allows for some optimization when accessing CRAM files without indexes.
CRAM files can also be indexed using CRAM or BAM indexes.

CRAM index

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

1. Sequence name
2. Alignment start
3. Container start offset in the file

4. Block start offset in the container

Each line represents a slice in the CRAM file. Please note that some slices could be omitted from the index.

BAM index

BAM indexes are supported by using 4-byte integer pointers called landmarks that are stored in the compression header blocks. 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 the compression header block. The landmark index can be used to access the appropriate slice.

Appendix

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.

Unary coding

Definition

This is simply a way to capture a numeric value N using the same bit for N times followed by the opposite bit.

Examples

Number	Codeword

0	1
1	01
4	0000 1
10	0000 0000 001

Parameters

CRAM format defines the following parameters of unary coding:

Data type	Name	Comment
I	offset	offset is added to each value
b	stop bit	the bit (1 or 0) used as a stop

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
I	offset	offset is added to each value

i	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.

Deconding

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

Gamma encoding takes no parameters.

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* = int[*N*/*M*]
    b. remainder = *r* = *N* modulo *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=[$\log_2$(M)]
            1. If r<2^b-M code *r* as plain binary using *b*-1 bits.
            2. 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
I	offset	offset is added to each value
i	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.

Exponential-Golomb coding

Definition

Exponential-Golomb coding of order *k* is a type of universal code, parametrized by a non-negative integer *k*. To encode a non-negative integer in an order-*k* exp-Golomb code, one can use the following method:

1. Take the number in binary except for the last *k* digits and add 1 to it (arithmetically). Write this down.
2. Count the bits written, subtract one, and write that number of starting zero bits preceding the previous bit string.
3. Write the last *k* bits in binary.

Encoding

1. Determine i such that $\sum_{j=0}^{i-1} 2^{j+k} \leq l < \sum_{j=0}^{i-1} 2^{j+k}, i \geq 0.$
2. Form the prefix of i 1s.
3. Insert the separator 0.
4. Form the tail: express the value of $(l - \sum_{j=0}^{i-1} 2^{j+k})$ as a $(k + i)$-bit binary number.

Decoding

1. Let i be the number of leading 1s (prefix) in the codeword.
2. Form a run of 0s of length $\sum_{j=0}^{i-1} 2^{k+j}$.
3. Skip the next 0 (separator).
4. The next $(k + i)$ bits make up the tail. Form a run of 0s of length represented by the tail.
5. Append 1 to the run of 0s.
6. 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	100	01	001
2	101	1000	010
3	11000	1001	011
4	11001	1010	10000

5	11010	1011	10001
6	11011	110000	10010
7	1110000	110001	10011
8	1110001	110010	10100
9	1110010	110011	10101
10	1110011	110100	10110

Parameters

Data type	Name	Comment
l	offset	offset is added to each value
i	k	the order of the exponential-golomb coding

Subexponential coding

Definition

Subexponential coding is somewhat similar to Exponential-Golomb coding and also 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 $l < 2^k$, then i = 0.
    b. – if $l \geq 2^k$, then determine i such that $2^{i+k-1} \leq l < 2^{i+k}$.
2. Form the prefix of i 1s.
3. Insert the separator 0.
4. Form the tail: express the value of $(l - 2^{i+k-1})$ as a (i + k − 1)-bit binary number.

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 ≥ 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
I	offset	offset is added to each value
i	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
i[]	alphabet	list of all encoded values
i[]	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
encoding<byte>	values encoding	an encoding describing how the values are captured

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.