Sequence Alignment/Map (SAM) Format

Version 0.1.2-draft (20090416)

1. Introduction

1.1. Purpose

This specification aims to define a generic nucleotide alignment format, SAM, that describes the alignment of query sequences or sequencing reads to a reference sequence or assembly, and:

- Is flexible enough to store all the alignment information generated by various alignment programs;
- Is simple enough to be easily generated by alignment programs or converted from existing alignment formats;
- Is compact in file size;
- Allows most of operations on the alignment to work on a stream without loading the whole alignment into memory;
- Allows the file to be indexed by genomic position to efficiently retrieve all reads aligning to a locus.

The document also describes the format of the binary equivalent to SAM and the format of alignment index.

1.2. Scope

This document specifies the formats of the text and binary alignment files and describes the indexing algorithm and the format of index files. It does not specify any application programming interfaces (APIs) or language bindings.

1.3. Nomenclature

Reference sequence. An existing sequence typically from previous studies. A reference sequence can be, but not is restricted to, a chromosome, a supercontig/scaffold or a contig from *de novo* assembly.

Query sequence. A sequence that is aligned to the reference sequences. A query sequence can be, but is not restricted to, a sequencing read, a cDNA or a contig. Typically, a query sequence is shorter than a target sequence.

Alignment. An alignment record describes a relationship between one query and one reference sequence. Insertions and deletions are allowed on either sequence. A query or a target sequence can be present in more than one alignment records.

1.4. Rationale

1.4.1. Text vs. binary format

SAM is a TAB-delimited text format. It is easy to understand, easy to parse, easy to generate and easy to check for errors. However, SAM is a bit slow to parse. Therefore we introduce a binary equivalent to SAM, called BAM, for intensive data processing. We envision that BAM will be used in most production pipelines, but that SAM, which is simpler to parse and can be produced by streaming from BAM, may be useful for interconversion with external applications and for exploratory analyses.

1.4.2. Flexibility: storing optional fields

Different alignment programs may produce different information which may be useful to the downstream analyses. A generic alignment format should allow for such information to be stored conveniently. In SAM, each alignment must contain a fixed number of mandatory fields that describe the key information about the alignment (such as coordinate detailed alignment and sequences) and may contain a variable number of optional fields which are less important or aligner specific.

1.4.3. Flexibility: storing various types of alignments

SAM is able to store clipped alignments, spliced alignments, multi-part alignments, padded alignments and alignments in color space. The extended CIGAR string is the key to describing these types of alignments.

Clipped alignment. In Smith-Waterman alignment, a sequence may not be aligned from the first residue to the last one. Subsequences at the ends may be clipped off. We introduce operation 'S' to describe (softly) clipped alignment. Here is an example. Suppose the clipped alignment is:

```
 REF: AGCTAGCATCGTGTCGCCCGTCTAGCATACGCATGATCGACTGTCAGCTAGTCAGACTAGTCGATCGATGTG
READ:        gggGTGTAACC-GACTAGgggg
```

where on the read sequence, bases in uppercase are matches and bases in lowercase are clipped off. The CIGAR for this alignment is: `3S8M1D6M4S`.

Spliced alignment. In cDNA-to-genome alignment, we may want to distinguish introns from deletions in exons. We introduce operation 'N' to represent long skip on the reference sequence. Suppose the spliced alignment is:

```
 REF: AGCTAGCATCGTGTCGCCCGTCTAGCATACGCATGATCGACTGTCAGCTAGTCAGACTAGTCGATCGATGTG
READ:           GTGTAACCC................................TCAGAATA
```

where '...' on the read sequence indicates the intron. The CIGAR for this alignment is: `9M32N8M`.

Multi-part alignment. One query sequence may be aligned to multiple places on the reference genome, either with or without overlaps. In SAM, we keep multiple hits as multiple alignment records. To avoid presenting the full query sequence multiple times for non-overlapping hits, we introduce operation 'H' to describe hard clipped alignment. Hard clipping (H) is similar to soft clipping (S). They are different in that hard clipped subsequence is not present in the alignment record. The example alignment in "clipped alignment" can also be represented with CIGAR: `3H8M1D6M4H`, but in this case, the sequence stored in SAM is "`GTGTAACCGACTAG`", instead of "`GGGGTGTAACCGACTAGGGGG`" if soft clipping is in use.

Padded alignment. Most sequence aligners only give the sequences inserted to the reference genome, but do not present how these inserted sequences are aligned against each other. Alignment with inserted sequences fully aligned is called padded alignment. Padded alignment is always produced by *de novo* assemblers and is important for an alignment viewer to display the alignment properly. To store padded alignment, we introduce operation 'P' which can be considered as a silent deletion from padded reference sequence. In the following example, GA on READ1 and A on READ2 are inserted to the reference. With unpadded CIGAR, we would not be able to distinguish the following padded multi-alignments:

```
  REF: CACGATCA**GACCGATACGTCCGA          REF: CACGATCA**GACCGATACGTCCGA
READ1:   CGATCAGAGACCGATA              READ1:   CGATCAGAGACCGATA
READ2:     ATCA*AGACCGATAC             READ2:     ATCAA*GACCGATAC
READ3:    GATCA**GACCG                 READ3:    GATCA**GACCG
```

The padded CIGAR are different:

```
READ1: 6M2I8M                          READ1: 6M2I8M
READ2: 4M1P1I9M                        READ2: 4M1I1P9M
READ3: 5M2P5M                          READ3: 5M2P5M
```

Note that it is hard to convert unpadded CIGAR to padded one. Fully resolving the alignment between inserted sequences would essentially require a *de novo* assembler. However, it is easy vice versa. By simply removing all P operations we get the CIGAR without padding.

Alignments in color space. Color alignments are stored as normal nucleotide alignments with additional tags describing the raw color sequences, qualities and color-specific properties.

1.4.4. Storing paired-end reads

A mapped read pair is stored in two (or more if multiple hits are stored) separate alignment records. The two reads in the pair have identical read pair name and are distinguished by their flag field (Section 2.2.2). The mate coordinate and the inferred insert size are recommended (not required) to be present. A tool is also provided to reconstruct mating information from BAM, although this is done at the cost of intensive computation and large disk space.

If in a read pair one read is mapped but the other not, the unmapped read can be absent from the alignment file or may be stored in two optional ways. The first method is to record no coordinate for the unmapped read (i.e. reference name = "*"). When using this method, flag bit 0x08 must be set on the mapped mate. The second method is to give the unmapped read a coordinate for sorting/indexing purposes only (this is generally the coordinate of the mapped mate). When using the second method, flag bit 0x4 must be set on the unmapped read, flag bit 0x08 must be set on the mapped mate.

1.4.5. File compression and random access in a compressed file

Typically, the size of a BAM file can be reduced by nearly a factor of four (to ~27%) under gzip/zlib compression. This compression ratio is significant. To achieve smaller file size, we always compress a BAM file with the BGZF library, developed by Bob Handsaker. BGZF is a stand-alone library that achieves similar compression ratio to gzip/zlib while supporting random access using virtual file offsets. A file compressed with BGZF is also gzip/zlib compatible in that we can use gzip/zlib to decompress the compressed file, although random file access is not supported in this case.

1.4.6. Ordering the alignments

An SAM/BAM file can be sorted by the reference coordinates, by query names, or unsorted. However, most operations on the alignments only work on a BAM sorted by the leftmost reference coordinate. Such an order is crucial to data processing on a stream and to indexing. A command-line tool is provided to sort an unsorted BAM in the required order.

1.4.7. Indexing alignments

Indexing paves the way for quick retrieval of alignments overlapping with a specified region. As BAM is supposed to work with spliced alignments, indexing must be efficient for alignments spanning long distance on the reference genome. A binning index as is used in the UCSC Genome Browser suits this goal better than a linear index alone. The binning index is further improved by being coupled with a simple linear index. For short read alignments, one seek call is needed in most cases to retrieve alignments.

1.5. Format implementation

SAM/BAM is implemented in two forms: a development library and a command-line tool. The library provides developers with basic I/O on SAM/BAM as well as routines on manipulating the alignment, such as merging, sorting, indexing and viewing. The command-line tool is built upon the library and is more convenient to non-developers. However, describing implementation details is out of the scope of this document.

1.6. Contributions

The major contributors to this specification are (in no particular order):

- Heng Li (Sanger Institute)
- Bob Handsaker (Broad Institute)
- Jue Ruan (Beijing Genomics Institute)
- Richard Durbin (Sanger Institute)
- Gabor Marth (Boston College)
- Michael Stromberg (Boston College)
- Fiona Hyland (Applied Biosystems)
- Goncalo Abecasis (University of Michigan)
- Richa Agarwala (NCBI)

2. SAM Format Specification

The SAM format consists of one header section and one alignment section. The whole header section can be absent, but keeping the header is recommended.

Here is an example of an SAM file:

```
@HD    VN:1.0
@SQ    SN:chr20 AS:HG18 LN:62435964
@RG    ID:L1 PU:SC_1_10 LB:SC_1 SM:NA12891
@RG    ID:L2 PU:SC_2_12 LB:SC_2 SM:NA12891
read_28833_29006_6945 99 20 chr20 28833 10M1D25M = 28993 195 \
       AGCTTAGCTAGCTACCTATATCTTGGTCTTGGCCG <<<<<<<<<<<<<<<<<<<<<<:<9/,&,22;;<<< \
       MF:i:130 NM:i:1 H0:i:0 H1:i:0 RG:Z:L1
read_28701_28881_323b 147 30 chr20 28834 35M = 28701 -168 \
       ACCTATATCTTGGCCTTGGCCGATGCGGCCTTGCA <<<<<;<<<<7;:<<<6;<<<<<<<<<<<<7<<<< \
       MF:i:18 NM:i:0 H0:i:1 H1:i:0 RG:Z:L2
```

2.1. Header section

Each header line begins with character '@' followed by a two-letter record type code. In the header, each line is TAB-delimited and each data field has an explicit field tag, which is represented using two ASCII characters, as is described below. The field type defines the content and format of the data in the field.

The following table give the defined record types and tags. Tags with '*' are required when the record type is present.

Type	Tag	Description
`HD - header`	`VN*`	File format version.
	`SO`	Sort order. Valid values are: *unsorted*, *queryname* or *coordinate*.
	`GO`	Group order (full sorting is not imposed in a group). Valid values are: *none*, *query* or *reference*.
`SQ - Sequence dictionary`	`SN*`	Sequence name. Unique among all sequence records in the file. The value of this field is used in alignment records.
	`LN*`	Sequence length.
	`AS`	Genome assembly identifier. Refers to the reference genome assembly in an unambiguous form. Example: HG18.
	`M5`	MD5 checksum of the sequence in the uppercase (gaps and space are removed)
	`UR`	URI of the sequence
	`SP`	Species.
`RG - read group`	`ID*`	Unique read group identifier. The value of the ID field is used in the RG tags of alignment records.
	`SM*`	Sample (use pool name where a pool is being sequenced)
	`LB`	Library
	`DS`	Description
	`PU`	Platform unit (e.g. lane for Illumina or slide for SOLiD); should be a full, unambiguous identifier
	`PI`	Predicted median insert size (maybe different from the actual median insert size)
	`CN`	Name of sequencing center producing the read.
	`DT`	Date the run was produced (ISO 8601 date or date/time).
	`PL`	Platform/technology used to produce the read.
`PG - Program`	`ID*`	Program name
	`VN`	Program version
	`CL`	Command line

2.2. Alignment Section

2.2.1. Overview

The alignment section consists of multiple TAB-delimited lines with each line describing an alignment. Each line is:

```
<QNAME> <FLAG> <RNAME> <POS> <MAPQ> <CIGAR> <MRNM> <MPOS> <ISIZE> <SEQ> <QUAL> \
  [<TAG>:<VTYPE>:<VALUE> [...]]
```

The format of each field is explained in the following table. More detailed descriptions are given in the sections below.

Field	Regular expression	Range	Description
QNAME	`[^ \t\n\r]+`		Query pair NAME if paired; or Query NAME if unpaired [2]
FLAG	`[0-9]+`	$[0,2^{16}-1]$	bitwise FLAG (Section 2.2.2)
RNAME	`[^ \t\n\r@=]+`		Reference sequence NAME [3]
POS	`[0-9]+`	$[0,2^{29}-1]$	1-based leftmost POSition/coordinate of the clipped sequence
MAPQ	`[0-9]+`	$[0,2^{8}-1]$	MAPping Quality (phred-scaled posterior probability that the mapping position of this read is incorrect) [4]
CIGAR	`([0-9]+[MIDNSHP])+|\*`		extended CIGAR string
MRNM	`[^ \t\n\r@]+`		Mate Reference sequence NaMe; "=" if the same as `<RNAME>` [3]
MPOS	`[0-9]+`	$[0,2^{29}-1]$	1-based leftmost Mate POSition of the clipped sequence
ISIZE	`-?[0-9]+`	$[-2^{29},2^{29}]$	inferred Insert SIZE [5]
SEQ	`[acgtnACGTN.=]+|\*`		query SEQuence; "=" for a match to the reference; n/N/. for ambiguity; cases are not maintained [6,7]
QUAL	`[!-~]+|\*`		query QUALity; `ASCII-33` gives the Phred base quality [6,7]
TAG	`[A-Z][A-Z0-9]`		TAG
VTYPE	`[AifZH]`		Value TYPE
VALUE	`[^\t\n\r]+`		match `<VTYPE>` (space allowed)

Notes:

1. `QNAME` and `FLAG` are required for all alignments. If the mapping position of the query is not available, `RNAME` and `CIGAR` are set as "*", and `POS` and `MAPQ` as 0. If the query is unpaired or pairing information is not available, `MRNM` equals "*", and `MPOS` and `ISIZE` equal 0. `SEQ` and `QUAL` can both be absent, represented as a star "*". If `QUAL` is not a star, it must be of the same length as `SEQ`.
2. The name of a pair/read is required to be unique in the SAM file, but one pair/read may appear multiple times in different alignment records, representing multiple or split hits. The maximum string length is 254.
3. If `SQ` is present in the header, `RNAME` and `MRNM` must appear in an `SQ` header record.
4. Field `MAPQ` considers pairing in calculation if the read is paired. Providing `MAPQ` is recommended. If such a calculation is difficult, 255 should be applied, indicating the mapping quality is not available.
5. If the two reads in a pair are mapped to the same reference, `ISIZE` equals the difference between the coordinate of the 5'-end of the mate and of the 5'-end of the current read; otherwise `ISIZE` equals 0 (by the "5'-end" we mean the 5'-end of the original read, so for Illumina short-insert paired end reads this calculates the difference in mapping coordinates of the outer edges of the original sequenced fragment). `ISIZE` is negative if the mate is mapped to a smaller coordinate than the current read.
6. Color alignments are stored as normal nucleotide alignments with additional tags describing the raw color sequences, qualities and color-specific properties (see also Note 5 in section 2.2.4).
7. All mapped reads are represented on the forward genomic strand. The bases are reverse complemented from the unmapped read sequence and the quality scores and cigar strings are recorded consistently with the bases. This applies to information in the mate tags (`R2`, `Q2`, `S2`, etc.) and any other tags that are strand sensitive. The strand bits in the flag simply indicates whether this reverse complement transform was applied from the original read sequence to obtain the bases listed in the SAM file.

2.2.2. The `<flag>` field

Field `<flag>` is a bitwise flag. The meaning of predefined bits is shown in the following table:

Flag	Description
`0x0001`	the read is paired in sequencing, no matter whether it is mapped in a pair
`0x0002`	the read is mapped in a proper pair (depends on the protocol, normally inferred during alignment) [1]
`0x0004`	the query sequence itself is unmapped
`0x0008`	the mate is unmapped [1]
`0x0010`	strand of the query (0 for forward; 1 for reverse strand)

Flag	Description
`0x0020`	strand of the mate [1]
`0x0040`	the read is the first read in a pair [1,2]
`0x0080`	the read is the second read in a pair [1,2]
`0x0100`	the alignment is not primary (a read having split hits may have multiple primary alignment records)
`0x0200`	the read fails platform/vendor quality checks
`0x0400`	the read is either a PCR duplicate or an optical duplicate

Notes:

1. Flag 0x02, 0x08, 0x20, 0x40 and 0x80 are only meaningful when flag 0x01 is present.
2. If in a read pair the information on which read is the first in the pair is lost in the upstream analysis, flag 0x01 should be present and 0x40 and 0x80 are both zero.

2.2.3. Extended CIGAR format

A CIGAR string is comprised of a series of operation lengths plus the operations. The conventional CIGAR format allows for three types of operations: M for match or mismatch, I for insertion and D for deletion. The extended CIGAR format further allows four more operations, as is shown in the following table, to describe clipping, padding and splicing:

op	Description
`M`	Match or mismatch
`I`	Insertion to the reference
`D`	Deletion from the reference
`N`	Skipped region from the reference
`S`	Soft clip on the read (clipped sequence present in `<seq>`)
`H`	Hard clip on the read (clipped sequence NOT present in `<seq>`)
`P`	Padding (silent deletion from the padded reference sequence)

2.2.4. Format of optional fields

Optional fields are in the format: `<TAG>:<VTYPE>:<VALUE>`. Each tag is encoded in two alphanumeric characters and appears only once for an alignment. The `<VTYPE>` follows Perl's rule (see also perldoc -f pack). Valid types in SAM are:

Type	Description
`A`	Printable character
`i`	Signed 32-bit integer
`f`	Single-precision float number
`Z`	Printable string
`H`	Hex string

Predefined tags are shown in the following table. You can freely add new tags, and if a new tag may be of general interest, you can email samtools-help@lists.sourceforge.net to add the new tag to the specification. Note that tags started with 'X', 'Y' and 'Z' are reserved for local use and will not be formally defined in any future version of this specification.

Tag	Type	Description
`X?`	`?`	Reserved fields for end users (together with Y? and Z?)
`RG`	`Z`	Read group. Value matches the header `RG-ID` tag if `@RG` is present in the header.
`LB`	`Z`	Library. Value should be consistent with the header `RG-LB` tag if `@RG` is present.
`PU`	`Z`	Platform unit. Value should be consistent with the header `RG-PU` tag if `@RG` is present.
`PG`	`Z`	Program that generates the alignment; match the header `PG-ID` tag if `@PG` is present.
`AS`	`i`	Alignment score generated by aligner
`SQ`	`H`	Encoded base probabilities for the suboptimal bases at each position [1]
`MQ`	`i`	The mapping quality score the mate alignment

Tag	Type	Description
`NM`	`i`	Number of nucleotide differences (i.e. edit distance to the reference sequence) [2]
`H0`	`i`	Number of perfect hits [2]
`H1`	`i`	Number of 1-difference hits (an in/del counted as a difference) [2]
`H2`	`i`	Number of 2-difference hits (an in/del counted as a difference) [2]
`UQ`	`i`	Phred likelihood of the read sequence, conditional on the mapping location being correct [5]
`PQ`	`i`	Phred likelihood of the read pair, conditional on both the mapping locations being correct [5]
`NH`	`i`	Number of reported alignments that contains the query in the current record
`IH`	`i`	Number of stored alignments in SAM that contains the query in the current record
`HI`	`i`	Query hit index, indicating the alignment record is the *i*-th one stored in SAM
`MD`	`Z`	String for mismatching positions in the format of `[0-9]+(([ACGTN]|\^[ACGTN]+)[0-9]+)*` [2,3]
`CS`	`Z`	Color read sequence on the same strand as the reference [4]
`CQ`	`Z`	Color read quality on the same strand as the reference; encoded in the same way as <`QUAL`> [4]
`CM`	`i`	Number of color differences [2]
`GS`	`Z`	Sequence in the overlap [6]
`GQ`	`Z`	Quality in the overlap encoded in the same way as the `QUAL` field [6]
`GC`	`Z`	CIGAR-like string describing the overlaps in the format of `[0-9]+[SG]` [6]
`R2`	`Z`	Sequence of the mate.
`Q2`	`Z`	Phred quality for the mate (encoding is the same as <`QUAL`>).
`S2`	`H`	Encoded base probabilities for the other 3 bases for the mate-pair read. Same encoding as `SQ` [1]
`CC`	`Z`	Reference name of the next hit; "=" for the same chromosome
`CP`	`i`	Leftmost coordinate of the next hit
`SM`	`i`	Mapping quality if the read is mapped as a single read rather than as a read pair
`AM`	`i`	Smaller single-end mapping quality of the two reads in a pair
`MF`	`i`	MAQ pair flag (MAQ specific)

Notes:

1. In the `SQ` field, the highest 2 bits give the second most likely base; the next 6 bits give the phred scaled log likelihood ratio of the second to the third most likely base calls.
2. Mismatches/insertions/deletions in clipped sequences are not counted.
3. The `MD` field aims to achieve SNP/indel calling without looking at the reference. SOAP and Eland SNP callers prefer such information. For example, a string "`10A5^AC6`" means from the leftmost reference base in the alignment, there are 10 matches followed by an A on the reference which is different from the aligned read base; the next 5 reference bases are matches followed by a 2bp deletion from the reference; the deleted sequence is AC; the last 6 bases are matches. The `MD` field should match the CIGAR string, although an SAM parser may not check this optional field.
4. On a raw SOLiD read, the first nucleotide is the primer base and the first color is the one between the primer base and the first nucleotide from the sample being sequenced. The primer base and the first color must be present in `CS`.
5. On the assumption that each base on a read is independent, `UQ` equals the sum of phred scores at mismatch positions, plus a term for the indels in the alignment. Tag `PQ` further includes the phred likelihood of insert size.
6. Some bases from reads generated by Complete Genomics may come from the same nucleotide. The `SEQ` and `QUAL` fields always store the flattened sequence and quality in that bases and qualities from the same nucleotide are collapsed to one. The three optional tags `GS`/`GQ`/`GC` describes how to generate the raw read. For example, given a raw read `AAACGCGAAAA`, '`CG`' starting from 4th and 6th position come from the same oligonucleotide. Suppose this read is mapped without gaps. In SAM, the read alignment is stored as: `SEQ=AAACGAAAA`, `CIGAR=9M`, `GS:Z:CGCG`, and `GC:Z:3S2G4S`, where `GS` keeps the bases in the overlap and `GC` says that to get the raw read sequence, we need to copy 3 bases from `SEQ`, copy 2+2 bases from `GS` and then copy 4 bases from the `SEQ` field again.

3. BAM Format Specification

BAM is compressed in the BGZF format. All integers in BAM are little-endian, regardless of the machine endianness. The whole format is formally described in the following table (values in [] are the default when the corresponding information is not available):

Field			Description	Type	Value
magic			BAM magic number	char[4]	BAM\1
l_text			Length of the header text, including any zero padding [1]	int32_t	
text			Plain header text in SAM; not necessarily zero terminated [1]	char[l_text]	
n_ref			# reference sequences	int32_t	
List of reference information (n = n_ref)					
	l_name		Length of the reference name plus 1 (including NULL)	int32_t	
	name		Name; NULL terminated	char[l_name]	
	l_ref		Length of the reference sequence	int32_t	
List of alignments (until the end of the file)					
	block_size		Length of the remainder of the block	int32_t	
	rID		Reference sequence ID (`-1≤rID<n_ref`)	int32_t	[-1]
	pos		0-based leftmost coordinate	int32_t	[-1]
	bin_mq_nl		`bin<<16|mapQual<<8|read_name_len` (including NULL) [2]	uint32_t	
	flag_nc		`flag<<16|cigar_len`	uint32_t	
	read_len		Length of the read	int32_t	
	mate_rID		Mate reference sequence ID (`-1≤mate_rID<n_ref`)	int32_t	[-1]
	mate_pos		Leftmost coordinate of the mate	int32_t	[-1]
	ins_size		Insert size of the read pair (if paired)	int32_t	[0]
	read_name		Read name, null terminated	char[read_name_len]	
	cigar		Cigar: `op_len<<4|op`. Op: `MIDNSHP=>0123456`	uint32_t[cigar_len]	
	seq		4-bit encoded read: `=ACGTN=>0,1,2,4,8,15`; the earlier base is stored in the high-order 4 bits of the byte.	char[(read_len+1)/2]	
	qual		Phred base quality (`0xFF` if absent)	uint8_t[read_len]	
	List of auxiliary data (until the end of this alignment block)				
		tag	Two-character tag	char[2]	
		val_type	Value type: AcCsSiIfZH. An integer may be stored as cCsSiI, depending on the magnitude of the integer. [3,4]	char	
		value	Content	by val_type	

Notes:

1. In the current version of specification, the header text in SAM is literally copied to BAM without parsing. In the future version, we may replace the "text" field with multiple dictionaries. The "l_text" field will indicate the total length of these dictionaries on disk. Having "l_text" guarantees that the extra structure in "text" will not break an old BAM parser.
2. "Bin" is used by indexing. Given an alignment in [beg,end), the bin is calculated by function `reg2bin()` described in section 4.3. Although bin can be calculated on the fly, precalculating it accelerates the retrieval of alignments in a specified region.
3. In BAM, an integer may be stored as a signed 8-bit integer (c), unsigned 8-bit integer (C), signed short (s), unsigned short (S), signed 32-bit (i) or unsigned 32-bit integer (I), depending on the signed magnitude of the integer. However, in SAM, all types of integers are presented as type 'i'. Having multiple precision on integers helps to reduce the disk space.

4. BGZF Compression

BGZF is block compression implemented on top of the standard gzip file format. The goal of BGZF is to provide good compression while allowing efficient random access to the BAM file for indexed queries. The BGZF format is "gunzip compatible", in the sense that a compliant gunzip utility can decompress a BGZF compressed file.

A BGZF archive is a series of concatenated BGZF blocks. Each BGZF block is itself a spec-compliant gzip archive which contains an "extra field" in the format described in RFC1952. The gzip file format allows the inclusion of application-specific extra fields and these are ignored by compliant decompression implementation. The gzip specification also allows gzip files to be concatenated. The result of decompressing concatenated gzip files is the concatenation of the uncompressed data.

Each BGZF block contains a standard gzip file header with the following standard-compliant extensions:

a) The F.EXTRA bit in the header is set to indicate that extra fields are present.
b) The extra field used by BGZF uses the two subfield ID values 66 and 67 (ascii 'BC').
c) The length of the BGZF extra field payload (field LEN in the gzip specification) is 2 (two bytes of payload).
d) The payload of the BGZF extra field is a 16-bit unsigned integer in little endian format.
   This integer gives the size of the containing BGZF block minus one.

On disk, a full BGZF file is (all integers are little endian as is required by RFC1952):

Field			Description	Type	Value
List of compression blocks (until the end of the file)					
	`ID1`		gzip IDentifier1	`uint8_t`	31
	`ID2`		gzip IDentifier2	`uint8_t`	139
	`CM`		gzip Compression Method	`uint8_t`	8
	`FLG`		gzip FLaGs	`uint8_t`	4
	`MTIME`		gzip Modification TIME	`uint32_t`	
	`XFL`		gzip eXtra FLags	`uint8_t`	
	`OS`		gzip Operating System	`uint8_t`	
	`XLEN`		gzip eXtra LENgth	`uint16_t`	
	Extra subfield(s) (total size=XLEN)				
		Additional RFC1952 extra subfields if present			
		`SI1`	Subfield Identifier1	`uint8_t`	66
		`SI2`	Subfield Identifier2	`uint8_t`	67
		`SLEN`	Subfield LENgth	`uint16_t`	2
		`BSIZE`	total Block SIZE minus 1	`uint16_t`	
		Additional RFC1952 extra subfields if present			
	`CDATA`		Compressed DATA by zlib::deflate()	`uint8_t[BSIZE-XLEN-19]`	
	`CRC32`		CRC-32	`uint32_t`	
	`ISIZE`		Input SIZE (length of uncompressed data)	`uint32_t`	

BGZF files support random access through the BAM file index. To achieve this, the BAM file index uses virtual file offsets into the BGZF file. Each virtual file offset is 64 bits, divided as follows:

coffset : 48 | uoffset : 16

The most significant 48 bits (coffset) are an unsigned byte offset into the BGZF file to the beginning of a BGZF block. The least significant 16 bits (uoffset) are an unsigned byte offset into the uncompressed data stream represented by that BGZF block. Virtual file offsets can be compared, but subtraction between virtual file offsets and addition between a virtual offset and an integer are both disallowed.

4.1. Implementation Note

There is a known bug in the Java GZIPInputStream class that concatenated gzip archives cannot be successfully decompressed by this class. BGZF files can be created and manipulated using the built-in Java util.zip package, but naive use of GZIPInputStream on a BGZF file will not work due to this bug.

5. Indexing BAM

Indexing aims to achieve fast retrieval of alignments overlapping a specified region without going through the whole alignments. BAM must be sorted by the reference ID and then the leftmost coordinate before indexing.

5.1. Algorithm

5.1.1. Basic binning index

The UCSC binning scheme was suggested by Richard Durbin and Lincoln Stein and is explained by Kent et al. (2002). In this scheme, each bin represents a contiguous genomic region which is either fully contained in or non-overlapping with another bin; each alignment is associated with a bin which represents the smallest region containing the entire alignment. The binning scheme is essentially a representation of R-tree. A distinct bin uniquely corresponds to a distinct internal node in a R-tree. Bin A is a child of Bin B if the region represented by A is contained in B.

To find the alignments that overlap a specified region, we need to get the bins that overlap the region, and then test each alignment in the bins to check overlap. To quickly find alignments associated with a specified bin, we can keep in the index the start file offsets of chunks of alignments which all have the bin. As alignments are sorted by the leftmost coordinates, alignments having the same bin tend to be clustered together on the disk and therefore usually a bin is only associated with a few chunks. Traversing all the alignments having the same bin usually needs a few seek calls. Given the set of bins that overlap the specified region, we can visit alignments in the order of their leftmost coordinates and stop seeking the rest when an alignment falls outside the required region. This strategy saves half of the seek calls in average.

In BAM, each bin may span 2^{29}, 2^{26}, 2^{23}, 2^{20}, 2^{17} or 2^{14} bp. Bin 0 spans a 512Mbp region, bins 1-8 span 64Mbp, 9-72 8Mbp, 73-584 1Mbp, 585-4680 128Kbp and bins 4681-37449 span 16Kbp regions.

5.1.2. Reducing small chunks

Around the boundary of two adjacent bins, we may see many small chunks with some having a shorter bin while the rest having a larger bin. To reduce the number of seek calls, we may join two chunks having the same bin if they are close to each other. After this process, a joined chunk will contain alignments with different bins. We need to keep in the index the file offset of the end of each chunk to identify its boundaries.

5.1.3. Combining with linear index

For an alignment starting beyond 64Mbp, we always need to seek to some chunks in bin 0, which can be avoided by using a linear index. In the linear index, for each tiling 16384bp window on the reference, we record the smallest file offset of the alignments that start in the window. Given a region [rbeg,rend), we only need to visit a chunk whose end file offset is larger than the file offset of the 16kbp window containing rbeg.

With both binning and linear indices, we can retrieve alignments in most of regions with just one seek call.

5.1.4. A conceptual example

Suppose we have a genome shorter than 144kbp. we can design a binning scheme which consists of three types of bins: bin 0 spans 0-144kbp, bin 1, 2 and 3 span 48kbp and bins from 4 to 12 span 16kbp each:

0 (0-144kbp)								
1 (0-48kbp)			2 (48-96kbp)			3 (96-144kbp)		
4 (0-16k)	5 (16-32k)	6 (32-48k)	7 (48-64k)	8 (64-80k)	9 (80-96k)	10	11	12

An alignment starting at 65kbp and ending at 67kbp would have a bin number 8, which is the smallest bin containing the alignment. Similarly, an alignment starting at 51kbp and ending at 70kbp would go to bin 2, while an alignment between [40k,49k] to bin 0. Suppose we want to find all the alignments overlapping region [65k,71k). We first calculate that bin 0, 2 and 8 overlap with this region and then traverse the alignments in these bins to find the required alignments. With a binning index alone, we need to visit the alignment at [40k,49k] as it belongs to bin 0. But with a linear index, we know that such an alignment stops before 64kbp and cannot overlap the specified region. A seek call can thus be saved.

5.2. Alignment Index Format

Field				Description	Type	Value
magic				Magic string	char[4]	BAI\1
n_ref				# reference sequences	int32_t	
List of indices (n = n_ref)						
	n_bin			# distinct bins (for the binning index)	int32_t	
	List of distinct bins (n = n_bin)					
		bin		Distinct bin	uint32_t	
		n_chunk		# chunks	int32_t	
		List of chunks (n = n_chunk)				
			chunk_beg	(Virtual) file offset of the start of the chunk	uint64_t	
			chunk_end	(Virtual) file offset of the end of the chunk	uint64_t	
	n_intv			# 16Kbp intervals (for the linear index)	int32_t	
	List of intervals (n = n_intv)					
		ioffset		(Virtual) file offset of the first alignment in the interval	uint64_t	

5.3. Useful Codes/Pseudo-codes

In the following, each bin may span 2^{29}, 2^{26}, 2^{23}, 2^{20}, 2^{17} or 2^{14} bp. Bin 0 spans a 512Mbp region, bins 1-8 span 64Mbp, 9-72 8Mbp, 73-584 1Mbp, 585-4680 128Kbp and bins 4681-37449 span 16Kbp regions.

```
/* calculate the bin given an alignment in [beg,end) */
int reg2bin(int beg, int end)
{
        --end;
        if (beg>>14 == end>>14) return ((1<<15)-1)/7 + (beg>>14);
        if (beg>>17 == end>>17) return ((1<<12)-1)/7 + (beg>>17);
        if (beg>>20 == end>>20) return  ((1<<9)-1)/7 + (beg>>20);
        if (beg>>23 == end>>23) return  ((1<<6)-1)/7 + (beg>>23);
        if (beg>>26 == end>>26) return  ((1<<3)-1)/7 + (beg>>26);
        return 0;
}
/* calculate list of bins that may overlap with region [rbeg,rend) */
#define MAX_BIN (((1<<18)-1)/7)
int reg2bins(int rbeg, int rend, uint16_t list[MAX_BIN])
{
        int i = 0, k;
        --rend;
        list[i++] = 0;
        for (k =    1 + (rbeg>>26); k <=    1 + (rend>>26); ++k) list[i++] = k;
        for (k =    9 + (rbeg>>23); k <=    9 + (rend>>23); ++k) list[i++] = k;
        for (k =   73 + (rbeg>>20); k <=   73 + (rend>>20); ++k) list[i++] = k;
        for (k =  585 + (rbeg>>17); k <=  585 + (rend>>17); ++k) list[i++] = k;
        for (k = 4681 + (rbeg>>14); k <= 4681 + (rend>>14); ++k) list[i++] = k;
        return i; // #elements in list[]
}
```

A. Genotype Likelihood Format version 3 (GLFv3)

The GLF format stores the probability of a genotype given data. It is not part of the SAM/BAM format, but is one of the outputs of the SAMtools utilities. Its specification is described here as appendix.

Like in BAM, all integers in GLF are stored in the little-endian byte order.

Field			Description	Type	Value
magic			GLFv3 magic number	char[4]	GLF\3
l_text			Length of the header text, including any zero padding	int32_t	
text			Text	char[l_text]	
List of reference information until the end of the file					
	l_name		Length of the reference sequence name plus 1 (including NULL)	int32_t	
	name		Name; NULL terminated	char[l_name]	
	ref_len		length of the reference sequence	uint32_t	
	List of sites until a record with rtype==0				
	rtype_ref		`record_type<<4|ref_base; 0..15=>XACMGRSVTWYHKDBN`	uint8_t	
	if rtype ==1	offset	offset from the precedent record[1]	uint32_t	
		min_depth	`min_lk<<24|read_depth` (min_lk capped at 255)	uint32_t	
		rmsMapQ	RMS of mapping qualities of reads covering the site	uint8_t	
		lk	likelihood of each genotype in the order of AA..AT..CC..CT..GG..TT	uint8_t[10]	
	if rtype ==2	offset	offset from the precedent record[1,2]	uint32_t	
		min_depth	`min_lk<<24|read_depth`	uint32_t	
		rmsMapQ	RMS of mapping qualities of reads covering the site	uint8_t	
		lkHom1	likelihood of the first homozygous indel allele (capped at 255)	uint8_t	
		lkHom2	likelihood of the second homozygous indel allele (capped at 255)	uint8_t	
		lkHet	likelihood of a heterozygote (capped at 255)	uint8_t	
		indelLen1	length of the first indel allele (positive=ins; negative=del; zero=no-indel)	int16_t	
		indelLen2	length of the second indel allele	int16_t	
		indelSeq1	sequence of the first indel allele	char[indelLen1]	
		indelSeq2	sequence of the second indel allele	char[indelLen2]	
	if 0	endMarker	end of this chromosome; no data in this record	(null)	

Notes:

1. Field offset equals the zero-based coordinate of the current record minus the coordinate of the precedent record. For the first record in a reference sequence, the coordinate of the precedent record is assumed to be zero. Offset is non-negative.
2. If a sequence is inserted between position [x,x+1] on the reference sequence, the coordinate of this record is x; if the sequence between [x,y] on the reference is deleted, the coordinate of this record is x.