CRAM 3.1 Cross-Library Assessment

Background

CRAM 3.1 extends CRAM 3.0 purely by adding four new block-level compression codecs (IDs 5-8). There are no structural changes to containers, slices, or blocks. The version byte in the file header indicates which codecs are available. Files using only codecs 0-4 should be written as CRAM 3.0 for maximum compatibility.

New Codecs in CRAM 3.1

Method ID	Codec	Purpose
5	rANS Nx16	16-bit renormalization rANS with stripe, RLE, and pack transforms
6	Adaptive Arithmetic Coding (Range)	Byte-wise range coder with adaptive probabilities
7	fqzcomp	Context-adaptive quality score compression
8	Name Tokeniser (tok3)	Column-wise read name compression

1. Is CRAM 3.1 Reading Implemented in All Three Libraries?

Yes - all three libraries can read CRAM 3.1 files.

Library	CRAM 3.1 Read	Evidence
htsjdk	Yes	Commit `c6729bb30` ("Wire up CRAM 3.1 codecs for reading"). `CRAMVersion(3,1)` registered in `supportedCRAMVersions`. Decoders for all 4 new codecs are implemented. Tests in `CRAM31Tests.java` round-trip against samtools-produced CRAM 3.1 files across multiple profiles (fast, normal, small, archive).
htslib	Yes	Default output version is already 3.1 (`minor_version = 1` in `cram_io.c:5260`). All codec IDs 0-8 are handled in both `cram_uncompress_block` and `cram_compress_block3`.
noodles	Yes	Version is transparently detected from the 2-byte header. All compression methods 0-8 are registered in the block reader. Read support since v0.12.0 (Feb 2022).

htsjdk Reading Detail

- **Version constants**: `CRAM_v3_1` defined as `new CRAMVersion(3, 1)` in `CramVersions.java`
- **Codec registration**: `CRAMCodecV3_1.java` detects CRAM 3.1 magic bytes (`CRAM\3\1`)
- **Decoder**: `CRAMDecoderV3_1.java` extends base `CRAMDecoder`
- **All codec decoders implemented**:
  - `RANSNx16Decode.java` - rANS Nx16 (order-0, order-1, all transform flags)
  - `RangeDecode.java` - Adaptive Arithmetic (order-0, order-1, all transform flags)
  - `FQZCompDecode.java` - Quality score decompression
  - `NameTokenisationDecode.java` - Name tokeniser decompression
- **Test coverage**: `CRAM31Tests.java` and `HtsCRAMCodec31Test.java` with comprehensive round-trip fidelity tests

htslib Reading Detail

- Version handling via `CRAM_MAJOR_VERS(fd->version)` and `CRAM_MINOR_VERS(fd->version)` macros
- All codec IDs handled in `cram_uncompress_block` switch statement
- Codec implementations sourced from the `htscodecs` library (rANS_static4x16, arith_dynamic, fqzcomp_qual, tokenise_name3)

noodles Reading Detail

- Version transparently decoded as `Version::new(buf[0], buf[1])` in `format_version.rs`
- Block compression methods decoded in `compression_method.rs` with all IDs 0-8 registered
- Full codec implementations in `noodles-cram/src/codecs/` for all CRAM 3.1 codecs

---

2. How Close Is htsjdk to Partial CRAM 3.1 Writing?

Very close - approximately 80-85% of the infrastructure is in place.

Per-Codec Write Status

Codec (ID)	Decode	Encode	Write Gap
rANS Nx16 (5)	Full	**Partial**	Stripe flag not implemented. Order-0, Order-1, Pack, RLE, CAT all work.
Range/Adaptive Arithmetic (6)	Full	**Partial**	Stripe flag not implemented. Order-0, Order-1, Pack, RLE, CAT, EXT (bzip2 fallback) all work.
fqzcomp (7)	Full	**Not implemented**	`FQZCompEncode.java` throws `CRAMException("FQZComp compression is not implemented")`. This is the single biggest gap.
Name Tokeniser (8)	Full	**Implemented**	`NameTokenisationEncode.java` is functional (described as "naive" - no duplicate stream detection, no all-match optimization).

What's Blocking the Write Path Today

1. **Hard block in encoder constructor**: `CRAMEncoderV3_1.java:25` throws `CRAMException("CRAM v3.1 encoding is not yet supported")`
2. **No 3.1 codecs in default encoding map**: `CompressionHeaderEncodingMap.java` only uses GZIP and rANS 4x8 for writing
3. **Default version still 3.0**: `DEFAULT_CRAM_VERSION` in `CramVersions.java` is `CRAM_v3`
4. **Best compressor selection ignores 3.1 codecs**: `getBestExternalCompressor()` only considers GZIP, rANS4x8 order-0, and rANS4x8 order-1

Existing Development Work

Development branches already contain commits enabling partial CRAM 3.1 writing:

- `881163d75` - "Enable a naive CRAM 3.1 write profile" (8 files changed, 67 insertions, 48 deletions)
  - Changes `DEFAULT_CRAM_VERSION` to `CRAM_v3_1`
  - Removes write exception from `CRAMEncoderV3_1`
  - Updates `CompressionHeaderEncodingMap` for 3.1
- `0135312a4` - "CRAM 3.1 write tests and code cleanup" (40 additional test cases)

These have **not** been merged to master.

Comparison with htslib's Normal Profile

A "naive" partial 3.1 write (equivalent to htslib's `normal` profile) is achievable **without fqzcomp**. htslib's `normal` profile doesn't use fqzcomp either - only the `small` and `archive` profiles do. The `normal` profile uses rANS (including Nx16) and tok3 for names, which htsjdk already has encoders for.

---

3. What Is Needed for Full CRAM 3.1 Writing in htsjdk?

Phase 1: Enable "Normal" Profile Writing (Lowest Risk, Highest Impact)

Gets htsjdk writing valid CRAM 3.1 files using the codecs that already have working encoders.

1.1 - Remove the write block in `CRAMEncoderV3_1`

- Remove the exception throw in the constructor
- Set `DEFAULT_CRAM_VERSION` to `CRAM_v3_1`
- Files: `CRAMEncoderV3_1.java`, `CramVersions.java`

1.2 - Update `CompressionHeaderEncodingMap` default encoding strategy

- Replace rANS 4x8 with rANS Nx16 for data series currently using rANS (e.g., `QS_QualityScore`, `BA_Base`, `BF_BitFlags`, etc.)
- Use Name Tokeniser (codec 8) for `RN_ReadName` instead of ByteArrayStopTab + GZIP
- Keep GZIP for data series where it's still appropriate
- File: `CompressionHeaderEncodingMap.java`

1.3 - Update `getBestExternalCompressor()` to include 3.1 codecs

- Add rANS Nx16 order-0 and order-1 to the candidate compressor set
- Add Range coding as a candidate
- Add Name Tokeniser for appropriate block types
- File: `CompressionHeaderEncodingMap.java`

1.4 - Interoperability tests

- Write CRAM 3.1 from htsjdk, read with samtools/htslib
- Write CRAM 3.1 from htsjdk, read with noodles
- Verify existing samtools-produced CRAM 3.1 files still round-trip
- Extend `CRAM31Tests.java`

Phase 2: Implement fqzcomp Encoding (Highest Complexity)

The biggest gap, enabling the `small` and `archive` profiles.

2.1 - Implement `FQZCompEncode.compress()`

- Complex, context-adaptive statistical encoder for quality scores
- Uses 16-bit context addressing with up to 65,536 models
- Context depends on: previous quality values, sequence position, read pairing, quality change frequency
- Reference implementations: htslib's `htscodecs` library (C), noodles' `fqzcomp/encode.rs` (Rust)
- File: `FQZCompEncode.java`
- Supporting files already exist: `FQZParam.java`, `FQZParams.java`, `FQZModels.java`, `FQZState.java`

2.2 - Integrate fqzcomp into the encoding strategy

- Wire `FQZCompExternalCompressor` into the `CompressionHeaderEncodingMap` for `QS_QualityScore`
- Add fqzcomp to `getBestExternalCompressor()` candidate set for quality-score blocks
- Requires `CRAMCodecModelContext` to propagate read metadata (position, pairing info) to the encoder - currently a placeholder class

2.3 - fqzcomp parameter tuning

- htslib provides multiple presets (`FQZ`, `FQZ_b`, `FQZ_c`, `FQZ_d`) with different compression vs speed tradeoffs
- Need to decide on parameter presets for htsjdk (or allow user configuration via `CRAMEncodingStrategy`)

Phase 3: Implement Stripe Support for rANS Nx16 and Range Coding

3.1 - Implement stripe encoding for rANS Nx16

- Currently throws `CRAMException("RANSNx16 Encoding with Stripe Flag is not implemented.")`
- Stripe interleaves byte streams for better SIMD performance
- Not strictly required for correctness, but improves compression ratios for certain data types
- File: `RANSNx16Encode.java`

3.2 - Implement stripe encoding for Range coding

- Same gap as rANS Nx16 - throws on stripe flag
- File: `RangeEncode.java`

Phase 4: Encoding Strategy & Profiles

4.1 - Introduce write profiles similar to htslib

htslib defines four profiles with different codec combinations:

Profile	Codecs Used	Records/Slice
fast	GZIP level 1, no tok3	10,000
normal	rANS Nx16, tok3 for names	default
small	+ bzip2, + fqzcomp	25,000
archive	+ adaptive arithmetic, + lzma at high levels	100,000

- Extend `CRAMEncodingStrategy.java` with a profile concept
- Allow users to select profiles or configure individual codecs

4.2 - Populate `CRAMCodecModelContext`

- Currently empty placeholder class
- Needs to carry read metadata for context-aware codecs (fqzcomp needs read position, pairing info, quality change count)
- Wire through the CRAM write path

Phase 5: Improve Name Tokeniser Quality

5.1 - Optimize the "naive" encoder

Current implementation notes it doesn't:

- Detect duplicate streams (set `DUP_PREVIOUS_STREAM_FLAG_MASK`)
- Detect all-match streams (spec states these can be discarded)

Adding these optimizations improves compression ratio but is not a correctness issue.

Phase 6: Validation & Performance

6.1 - Comprehensive cross-library round-trip testing

- htsjdk write -> htslib read (all profiles)
- htsjdk write -> noodles read (all profiles)
- htslib write -> htsjdk read (all profiles) - already done
- noodles write -> htsjdk read

6.2 - Performance benchmarking

- Compare compression ratios across profiles against htslib and noodles
- Measure encode/decode speed
- Profile memory usage (particularly for fqzcomp's 65K models)

6.3 - Spec conformance testing

- Use hts-specs interop test files
- Verify version negotiation: files using only codecs 0-4 should write as 3.0

Risk Assessment & Prioritization

Phase	Effort	Risk	Impact	Recommendation
Phase 1 (Normal profile)	Low	Low	High	**Do first** - most code exists on dev branches
Phase 2 (fqzcomp)	High	Medium	Medium	Second priority - needed for archival quality
Phase 3 (Stripe)	Medium	Low	Low	Optional - improves compression margins
Phase 4 (Profiles)	Medium	Low	Medium	Third priority - usability improvement
Phase 5 (Name tok improvements)	Low	Low	Low	Nice-to-have optimization
Phase 6 (Validation)	Medium	Low	High	Ongoing throughout all phases

Key Files Reference

htsjdk - Version & Codec Infrastructure

File	Purpose
`src/main/java/htsjdk/samtools/cram/common/CramVersions.java`	Version constants
`src/main/java/htsjdk/samtools/cram/common/CRAMVersion.java`	Version class

File	Purpose
src/main/java/htsjdk/beta/codecs/reads/cram/cramV3_1/CRAMEncoderV3_1.java	3.1 encoder (blocked)
src/main/java/htsjdk/samtools/cram/structure/CompressionHeaderEncodingMap.java	Encoding strategy map
src/main/java/htsjdk/samtools/cram/structure/CRAMEncodingStrategy.java	Encoding parameters
src/main/java/htsjdk/samtools/cram/structure/CRAMCodecModelContext.java	Context model (empty)
src/main/java/htsjdk/samtools/cram/compression/ExternalCompressor.java	Compressor factory
src/main/java/htsjdk/samtools/cram/structure/block/BlockCompressionMethod.java	Codec ID enum

htsjdk - Codec Implementations

Directory	Codec
src/main/java/htsjdk/samtools/cram/compression/rans/rans4x8/	rANS 4x8 (CRAM 3.0)
src/main/java/htsjdk/samtools/cram/compression/rans/ransnx16/	rANS Nx16 (CRAM 3.1)
src/main/java/htsjdk/samtools/cram/compression/range/	Range/Adaptive Arithmetic (CRAM 3.1)
src/main/java/htsjdk/samtools/cram/compression/fqzcomp/	fqzcomp (CRAM 3.1)
src/main/java/htsjdk/samtools/cram/compression/nametokenisation/	Name Tokeniser (CRAM 3.1)

htsjdk - Tests

File	Purpose
src/test/java/htsjdk/samtools/cram/CRAM31Tests.java	Round-trip fidelity tests
src/test/java/htsjdk/beta/codecs/reads/cram/HtsCRAMCodec31Test.java	Codec-level reading tests

Summary

- **Reading**: Fully implemented across htsjdk, htslib, and noodles. All three libraries can read CRAM 3.1 files produced by any of the others.
- **Partial writing**: htsjdk is very close. Development branches already demonstrate a working "normal" profile write. The main gaps are the hard block in `CRAMEncoderV3_1` (trivial to remove) and the default encoding map not using 3.1 codecs.
- **Full writing**: The single biggest engineering effort is implementing fqzcomp encoding. Everything else (stripe support, profiles, name tokeniser optimization) is incremental improvement on an already-functional foundation.

Phase 1 could likely be merged quickly since the development branch commits already demonstrate it working. The recommended approach is to ship Phase 1 first (enabling "normal" profile writing), then tackle fqzcomp encoding as a

separate, focused effort.