Phase 3: Stripe Encoding & Encoding Profiles — Detailed Plan

Overview

Phase 3 has two interrelated goals:

1. **Implement stripe encoding** for rANS Nx16 and Range (Adaptive Arithmetic) codecs. Stripe is the only remaining unimplemented transform flag — once implemented, htsjdk supports the full CRAM 3.1 codec feature set.
2. **Introduce encoding profiles** (`fast`, `normal`, `small`, `archive`) matching htslib/samtools, giving users a simple way to trade speed for compression ratio.

These are combined into one phase because the profiles need stripe to be available for `small` and `archive` tiers, and because the profile system is the mechanism through which stripe gets surfaced to users.

Assumes: Phase 1 merged (basic CRAM 3.1 write with RANSNx16 + Name Tokeniser), Phase 2 merged (FQZComp encoding available).

Part A: Stripe Encoding

Background

Stripe (also called "multiway interleave") splits input data into N byte-columns, compresses each column independently, then concatenates the compressed sub-streams. This exploits patterns in multi-byte values — e.g., a stream of 32-bit little-endian integers becomes 4 streams of byte-0s, byte-1s, byte-2s, and byte-3s, each far more compressible individually.

Both decoders (`RANSNx16Decode.decodeStripe()` and `RangeDecode.decodeStripe()`) are already fully implemented. The encoders throw `CRAMException` when the stripe flag is set. The two `decodeStripe()` implementations are structurally identical — the encoder implementations will mirror this symmetry.

Wire format (from the decoders)

```
[1 byte]     format flags (with STRIPE_FLAG_MASK = 0x08 set)
[uint7]      uncompressed size (unless NOSZ flag is also set)
[1 byte]     N (number of interleave streams, typically 4)
[uint7 × N]  compressed byte length of each sub-stream
[sub-stream 0..N-1]  each a fully self-contained compressed block with
NOSZ flag set
```

Each sub-stream is itself a complete rANS/Range compressed blob beginning with its own format flags byte. The sub-streams have the NOSZ flag set (bit 4, `0x10`) because their uncompressed size is calculated from the total size divided by N, not stored in the stream.

Task 1: Implement RANSNx16Encode.encodeStripe()

File: `src/main/java/htsjdk/samtools/cram/compression/rans/ransnx16/RANSNx16Encode.java`

1a: Replace the throw with a call to encodeStripe()

In the `compress()` method, replace:

```
if (ransNx16Params.isStripe()) {
    throw new CRAMException("RANSNx16 Encoding with Stripe Flag is not implemented.");
}
```

with:

```
if (ransNx16Params.isStripe()) {
    return encodeStripe(inputBuffer, outBuffer, ransNx16Params, uncompressedSize);
}
```

1b: Implement the encodeStripe() method

```
private ByteBuffer encodeStripe(
        final ByteBuffer inputBuffer,
        final ByteBuffer outBuffer,
        final RANSNx16Params params,
        final int uncompressedSize) {

    // 1. Determine number of interleave streams
    //    Typically 4 (matching 32-bit word size). Could be parameterised.
    final int numStreams = 4;
    outBuffer.put((byte) numStreams);

    // 2. Transpose: split input into N column-major streams
    //    Stream j gets bytes at positions j, j+N, j+2N, ...
    final ByteBuffer[] columns = new ByteBuffer[numStreams];
    final int[] columnLengths = new int[numStreams];
    for (int j = 0; j < numStreams; j++) {
        columnLengths[j] = uncompressedSize / numStreams;
        if (j < (uncompressedSize % numStreams)) {
            columnLengths[j]++;
        }
        columns[j] =
CompressionUtils.allocateByteBuffer(columnLengths[j]);
    }
    for (int i = 0; i < uncompressedSize; i++) {
        columns[i % numStreams].put(inputBuffer.get(i));
    }
    for (int j = 0; j < numStreams; j++) {
        columns[j].rewind();
    }

    // 3. Compress each column independently
    //    Use the same flags as the parent MINUS stripe and PLUS nosz
    //    (sub-streams must not recurse into stripe, and must set NOSZ)
    final int subFlags = (params.getFormatFlags() &
~RANSNx16Params.STRIPE_FLAG_MASK)
                         | RANSNx16Params.NOSZ_FLAG_MASK;
    final ByteBuffer[] compressedColumns = new ByteBuffer[numStreams];
    for (int j = 0; j < numStreams; j++) {
        compressedColumns[j] = compress(columns[j], new
RANSNx16Params(subFlags));
    }

    // 4. Write compressed lengths (uint7 for each sub-stream)
    //    Note: lengths are written BEFORE the sub-stream data
    //    We need to reserve space — write lengths first, then data
    //    The decoder reads N lengths, then N sub-streams sequentially
    final int lengthsStart = outBuffer.position();
    for (int j = 0; j < numStreams; j++) {
        CompressionUtils.writeUint7(outBuffer,
compressedColumns[j].remaining());
    }

    // 5. Write compressed sub-stream data
    for (int j = 0; j < numStreams; j++) {
        outBuffer.put(compressedColumns[j]);
    }

    outBuffer.limit(outBuffer.position());
    outBuffer.rewind();
    return outBuffer;
}
```

Key design decisions:

- **N = 4** is the standard choice (matching htslib). The value is written to the stream, so the decoder handles any N.
- **Sub-stream flags**: Strip the STRIPE bit (prevent recursion) and set NOSZ (size not embedded). Other flags (ORDER, PACK, RLE, N32, CAT) are inherited from the parent parameters, allowing stripe to combine with other transforms.
- **The recursive call to `compress()`** is the same method that handles all other transforms. This means stripe + pack, stripe + RLE, stripe + order-1, etc. all work automatically.

Task 2: Implement RangeEncode.encodeStripe()

File: `src/main/java/htsjdk/samtools/cram/compression/range/RangeEncode.java`

2a: Replace the throw

```
if (rangeParams.isStripe()) {
    return encodeStripe(inputBuffer, outBuffer, rangeParams,
uncompressedSize);
}
```

2b: Implement encodeStripe()

Structurally identical to the rANS version — the only difference is using `RangeParams` instead of `RANSNx16Params`:

```
private ByteBuffer encodeStripe(
        final ByteBuffer inputBuffer,
        final ByteBuffer outBuffer,
        final RangeParams params,
        final int uncompressedSize) {

    final int numStreams = 4;
    outBuffer.put((byte) numStreams);

    // Transpose into columns (identical to rANS)
    final ByteBuffer[] columns = new ByteBuffer[numStreams];
    final int[] columnLengths = new int[numStreams];
    for (int j = 0; j < numStreams; j++) {
        columnLengths[j] = uncompressedSize / numStreams;
        if (j < (uncompressedSize % numStreams)) columnLengths[j]++;
        columns[j] =
CompressionUtils.allocateByteBuffer(columnLengths[j]);
    }
    for (int i = 0; i < uncompressedSize; i++) {
        columns[i % numStreams].put(inputBuffer.get(i));
    }
    for (int j = 0; j < numStreams; j++) columns[j].rewind();

    // Compress each column (strip stripe flag, add nosz)
    final int subFlags = (params.getFormatFlags() &
~RangeParams.STRIPE_FLAG_MASK)
                             | RangeParams.NOSZ_FLAG_MASK;
    final ByteBuffer[] compressedColumns = new ByteBuffer[numStreams];
    for (int j = 0; j < numStreams; j++) {
        compressedColumns[j] = compress(columns[j], new
RangeParams(subFlags));
    }
    // Write lengths then data
    for (int j = 0; j < numStreams; j++) {
        CompressionUtils.writeUint7(outBuffer,
compressedColumns[j].remaining());
    }
    for (int j = 0; j < numStreams; j++) {
        outBuffer.put(compressedColumns[j]);
    }

    outBuffer.limit(outBuffer.position());
    outBuffer.rewind();
    return outBuffer;
}
```

Note: The two `encodeStripe()` methods are nearly identical. A shared utility could be factored out, but given that they operate on different parameter types (`RANSNx16Params` vs `RangeParams`) and call different `compress()` methods, keeping them separate avoids awkward generics and is consistent with the existing decoder pattern (the two `decodeStripe()` methods are also independent).

Task 3: Enable Stripe in NameTokenisationEncode

File: `src/main/java/htsjdk/samtools/cram/compression/nametokenisation/NameTokenisationEncode.java`

The `tryCompress()` method explicitly excludes stripe from the flag sets it tries:

```
// we don't include stripe here since it's not implemented for write
```

With stripe now available, add stripe combinations to the candidate flag sets:

```
// rANS path:
final int[] ransNx16FlagsSets = {
    0,
    RANSNx16Params.RLE_FLAG_MASK,
    RANSNx16Params.PACK_FLAG_MASK,
    RANSNx16Params.PACK_FLAG_MASK | RANSNx16Params.ORDER_FLAG_MASK,
    RANSNx16Params.STRIPE_FLAG_MASK,                                     //
NEW
    RANSNx16Params.STRIPE_FLAG_MASK | RANSNx16Params.ORDER_FLAG_MASK,    //
NEW
};

// Range path:
final int[] rangeEncoderFlagsSets = {
    0,
    RangeParams.RLE_FLAG_MASK,
    RangeParams.PACK_FLAG_MASK,
    RangeParams.PACK_FLAG_MASK | RangeParams.ORDER_FLAG_MASK,
    RangeParams.STRIPE_FLAG_MASK,                                       //
NEW
    RangeParams.STRIPE_FLAG_MASK | RangeParams.ORDER_FLAG_MASK,         //
NEW
};
```

Caveat: Stripe has overhead (N length fields + N format bytes). For small sub-streams (< ~100 bytes), stripe will likely expand rather than compress. The existing size-comparison logic in `tryCompress()` will naturally reject stripe when it doesn't help — no special guard needed beyond what's already there. However, guard against stripe when the stream length isn't divisible into meaningful columns (already handled: if `remaining() < numStreams`, the column sizes degenerate and the codec handles it, but compression won't be useful).

Task 4: Update Tests for Stripe Encoding

4a: Convert stripe reject tests to round-trip tests

File: `src/test/java/htsjdk/samtools/cram/RansTest.java`

The test `testRansNx16RejectEncodeStripe()` currently asserts that encoding with stripe throws `CRAMException`. Convert this to a round-trip test:

```
@Test(dataProvider = "getRansNx16StripeParams")
public void testRansNx16EncodeStripe(final RANSNx16Params params, final
byte[] testData) {
    final ByteBuffer compressed = ransNx16Encode.compress(
        CompressionUtils.wrap(testData), params);
    final ByteBuffer decompressed = ransNx16Decode.uncompress(compressed);
    Assert.assertEquals(decompressed.array(), testData);
}
```

File: `src/test/java/htsjdk/samtools/cram/RangeTest.java`

Similarly convert `testRangeEncodeStripe()` from a reject test to a round-trip test.

4b: Enable interop tests for stripe

File: `src/test/java/htsjdk/samtools/cram/RANSInteropTest.java`

Lines 165-168 currently detect stripe params and assert `CRAMException`. Change to:

```
// Previously:
// if (ransNx16Params.isStripe()) { assertThrows(CRAMException.class, ...)
}
// Now: full round-trip
final ByteBuffer uncompressed = ransNx16Decode.uncompress(compressedData);
final ByteBuffer recompressed = ransNx16Encode.compress(uncompressed,
ransNx16Params);
final ByteBuffer decompressed = ransNx16Decode.uncompress(recompressed);
Assert.assertEquals(decompressed.array(), uncompressed.array());
```

File: `src/test/java/htsjdk/samtools/cram/RangeInteropTest.java`

Same change at lines 80-81.

4c: Test with the canonical stripe test data

The test resources include `u32.gz` — a file of 32-bit little-endian integers specifically designed to test stripe (byte lanes become highly compressible). Verify that:

- `u32.9` (rANS: STRIPE | ORDER) decodes correctly (already works)
- Encoding `u32` with stripe flag produces output that decodes to the same data
- Compression ratio with stripe is significantly better than without for this data

4d: Test stripe + other flag combinations

Test that stripe combines correctly with other transforms:

- Stripe + order-0 (flag `0x08`)
- Stripe + order-1 (flag `0x09`)
- Stripe + pack (flag `0x88`)
- Stripe + RLE (flag `0x48`)
- Edge cases: empty input, single byte, input smaller than N (4 bytes)

Part B: Encoding Profiles

Background

htslib/samtools defines four encoding profiles that control the speed/compression tradeoff:

Profile	Records/Slice	GZIP Level	Core Codec	Quality Codec	Name Codec	Tag Candidates	Extra Codecs
fast	10,000	1	GZIP (fast)	GZIP (fast)	GZIP (no tok3)	GZIP only	—
normal	10,000	default	RANSNx16	RANSNx16 order-1	Name Tokeniser	GZIP, RANSNx16	—
small	25,000	default	RANSNx16	FQZComp	Name Tokeniser	GZIP, RANSNx16, BZIP2	BZIP2 for select series

Profile	Records/Slice	GZIP Level	Core Codec	Quality Codec	Name Codec	Tag Candidates	Extra Codecs
archive	100,000	default	Range (Adaptive Arithmetic)	FQZComp	Name Tokeniser	GZIP, RANSNx16, BZIP2, LZMA, Range	LZMA, Range for select series

Key differences from `normal` (Phase 1 default):

- **fast**: Drops RANSNx16 and Name Tokeniser entirely; GZIP-only at low compression level. Essentially a faster CRAM 3.0.
- **small**: Adds FQZComp for quality scores (Phase 2), BZIP2 as a tag compression candidate, larger slices for better compression.
- **archive**: Switches core codec from RANSNx16 to Range (Adaptive Arithmetic), adds LZMA as a candidate, much larger slices.

Task 5: Add EncodingProfile Enum to CRAMEncodingStrategy

File: `src/main/java/htsjdk/samtools/cram/structure/CRAMEncodingStrategy.java`

5a: Define the profile enum

```java
public enum EncodingProfile {
    FAST,
    NORMAL,
    SMALL,
    ARCHIVE
}
```

5b: Add a profile field with a default

```java
private EncodingProfile encodingProfile = EncodingProfile.NORMAL;

public CRAMEncodingStrategy setEncodingProfile(final EncodingProfile
profile) {
    this.encodingProfile = ValidationUtils.nonNull(profile,
"encodingProfile");
    return this;
}

public EncodingProfile getEncodingProfile() {
    return encodingProfile;
}
```

5c: Apply profile defaults

When a profile is set, automatically apply the associated defaults (the user can still override individual settings afterward):

```
public CRAMEncodingStrategy setEncodingProfile(final EncodingProfile
profile) {
    this.encodingProfile = ValidationUtils.nonNull(profile,
"encodingProfile");
    switch (profile) {
        case FAST:
            this.gzipCompressionLevel = 1;
            this.readsPerSlice = 10000;
            break;
        case NORMAL:
            this.readsPerSlice = 10000;
            break;
        case SMALL:
            this.readsPerSlice = 25000;
            break;
        case ARCHIVE:
            this.readsPerSlice = 100000;
            break;
    }
    return this;
}
```

Note: The profile does NOT set a `customCompressionHeaderEncodingMap` — instead, the `CompressionHeaderEncodingMap` constructor reads the profile from the strategy and wires codecs accordingly (see Task 7).

Task 6: Surface Profiles Through User-Facing APIs

6a: Add profile to CRAMEncoderOptions (HTS plugin framework)

File: `src/main/java/htsjdk/beta/codecs/reads/cram/CRAMEncoderOptions.java`

```
private CRAMEncodingStrategy.EncodingProfile encodingProfile;

public CRAMEncoderOptions
setEncodingProfile(CRAMEncodingStrategy.EncodingProfile profile) {
    this.encodingProfile = profile;
    return this;
}

public Optional<CRAMEncodingStrategy.EncodingProfile> getEncodingProfile()
{
    return Optional.ofNullable(encodingProfile);
}
```

6b: Thread CRAMEncodingStrategy through CRAMEncoder

File: `src/main/java/htsjdk/beta/codecs/reads/cram/CRAMEncoder.java`

The `getCRAMWriter()` method currently creates a `CRAMFileWriter` without an encoding strategy. Update it to:

1. Create a `CRAMEncodingStrategy` from `CRAMEncoderOptions`
2. Pass it to the `CRAMFileWriter` constructor that accepts a strategy

```
private CRAMFileWriter getCRAMWriter(...) {
    final CRAMEncodingStrategy strategy = new CRAMEncodingStrategy();
    readsEncoderOptions.getCRAMEncoderOptions()
        .flatMap(CRAMEncoderOptions::getEncodingProfile)
        .ifPresent(strategy::setEncodingProfile);

    return new CRAMFileWriter(
        strategy,           // <-- now threaded through
        outputStream,
        referenceSource,
        samFileHeader,
        outputIdentifier);
}
```

6c: Add encoding strategy support to SAMFileWriterFactory

File: `src/main/java/htsjdk/samtools/SAMFileWriterFactory.java`

There is an existing TODO at line ~687:

```
//TODO: set encoding params
//writer.setEncodingParams(new CRAMEncodingStrategy());
```

Add a setter for `CRAMEncodingStrategy`:

```
private CRAMEncodingStrategy cramEncodingStrategy;

public SAMFileWriterFactory setCRAMEncodingStrategy(final
CRAMEncodingStrategy strategy) {
    this.cramEncodingStrategy = strategy;
    return this;
}
```

And in `createCRAMWriterWithSettings()`, use it:

```
final CRAMEncodingStrategy strategy = cramEncodingStrategy != null
    ? cramEncodingStrategy
    : new CRAMEncodingStrategy();
final CRAMFileWriter writer = new CRAMFileWriter(strategy, stream, ...);
```

Task 7: Make CompressionHeaderEncodingMap Profile-Aware

File: `src/main/java/htsjdk/samtools/cram/structure/CompressionHeaderEncodingMap.java`

This is the core change. The `initializeDefaultEncodings()` method (called from the constructor) currently hardcodes the `normal` profile codec assignments. Refactor it to consult `CRAMEncodingStrategy.getEncodingProfile()`.

7a: Pass profile through to initializeDefaultEncodings

The constructor already receives `CRAMEncodingStrategy`. Extract the profile:

```
private void initializeDefaultEncodings(final CRAMEncodingStrategy
encodingStrategy) {
    final CRAMEncodingStrategy.EncodingProfile profile =
encodingStrategy.getEncodingProfile();
    // ... profile-conditional codec assignments below
}
```

7b: Profile-conditional codec assignment

```
switch (profile) {
    case FAST:
        initializeFastProfile(encodingStrategy);
        break;
    case NORMAL:
        initializeNormalProfile(encodingStrategy);
        break;
    case SMALL:
        initializeSmallProfile(encodingStrategy);
        break;
    case ARCHIVE:
        initializeArchiveProfile(encodingStrategy);
        break;
}
```

7c: Implement each profile method

`initializeFastProfile()` — GZIP-only, no RANSNx16, no Name Tokeniser:

```
All data series → GZIP(level=1)
RN_ReadName → ByteArrayStop('\t') + GZIP(level=1)   // no tok3
QS_QualityScore → GZIP(level=1)
IN_Insertion, SC_SoftClip → ByteArrayStop('\t') + GZIP(level=1)
```

`initializeNormalProfile()` — current Phase 1 defaults (RANSNx16 + tok3):

```
AP, RI → RANSNx16 order-0
BA, BF, CF, NS, QS, RG, RL, TS → RANSNx16 order-1
RN_ReadName → ByteArrayStop + NAME_TOKENISER
Everything else → GZIP(level)
```

`initializeSmallProfile()` — adds FQZComp for quality, BZIP2 candidates:

```
Same as normal, EXCEPT:
QS_QualityScore → FQZCOMP (instead of RANSNx16 order-1)
BS_BaseSubstitutionCode → BZIP2 (or best-of GZIP/RANSNx16/BZIP2)
Other GZIP series → consider BZIP2 as candidate (via
getBestExternalCompressor)
```

`initializeArchiveProfile()` — switches to Range coding, adds LZMA:

```
AP, RI → Range (Adaptive Arithmetic) order-0
BA, BF, CF, NS, RG, RL, TS → Range order-1
QS_QualityScore → FQZCOMP
RN_ReadName → ByteArrayStop + NAME_TOKENISER
Other series → best-of GZIP/RANSNx16/Range/BZIP2/LZMA (via
getBestExternalCompressor)
```

7d: Add helper methods for Range and BZIP2/LZMA encoding

```
private void putExternalRangeOrderOneEncoding(final DataSeries dataSeries)
{
    putExternalEncoding(dataSeries,
        compressorCache.getCompressorForMethod(
            BlockCompressionMethod.ADAPTIVE_ARITHMETIC,
            RangeParams.ORDER.ONE.ordinal()));
}

private void putExternalRangeOrderZeroEncoding(final DataSeries
dataSeries) {
    putExternalEncoding(dataSeries,
        compressorCache.getCompressorForMethod(
            BlockCompressionMethod.ADAPTIVE_ARITHMETIC,
            RangeParams.ORDER.ZERO.ordinal()));
}

private void putExternalBzip2Encoding(final CRAMEncodingStrategy strategy,
final DataSeries dataSeries) {
    putExternalEncoding(dataSeries,

compressorCache.getCompressorForMethod(BlockCompressionMethod.BZIP2,
ExternalCompressor.NO_COMPRESSION_ARG));
}
```

7e: Make getBestExternalCompressor() profile-aware

This method is used for tag block compression. Extend it to try additional codecs based on the profile:

```
public ExternalCompressor getBestExternalCompressor(
        final byte[] data,
        final CRAMEncodingStrategy encodingStrategy) {

    // Always try GZIP and RANSNx16

candidates.add(compressorCache.getCompressorForMethod(BlockCompressionMeth
od.GZIP, level));

candidates.add(compressorCache.getCompressorForMethod(BlockCompressionMeth
od.RANSNx16, ORDER.ZERO.ordinal()));

candidates.add(compressorCache.getCompressorForMethod(BlockCompressionMeth
od.RANSNx16, ORDER.ONE.ordinal()));

    final EncodingProfile profile = encodingStrategy.getEncodingProfile();

    // Small and archive: also try BZIP2
    if (profile == EncodingProfile.SMALL || profile ==
EncodingProfile.ARCHIVE) {

candidates.add(compressorCache.getCompressorForMethod(BlockCompressionMeth
od.BZIP2, NO_COMPRESSION_ARG));
    }

    // Archive: also try Range and LZMA
    if (profile == EncodingProfile.ARCHIVE) {

candidates.add(compressorCache.getCompressorForMethod(BlockCompressionMeth
od.ADAPTIVE_ARITHMETIC, ORDER.ZERO.ordinal()));

candidates.add(compressorCache.getCompressorForMethod(BlockCompressionMeth
od.ADAPTIVE_ARITHMETIC, ORDER.ONE.ordinal()));
candidates.add(compressorCache.getCompressorForMethod(BlockCompressionMeth
od.LZMA, NO_COMPRESSION_ARG));
    }

    // Fast: GZIP only (remove RANS candidates)
    if (profile == EncodingProfile.FAST) {
        candidates.clear();

candidates.add(compressorCache.getCompressorForMethod(BlockCompressionMeth
od.GZIP, level));
    }

    // Pick the smallest
    return candidates.stream()
        .min(Comparator.comparingInt(c -> c.compress(data, null).length))
        .orElseThrow();
}
```

Task 8: CRAM Version Negotiation

File: `src/main/java/htsjdk/samtools/cram/structure/CompressionHeaderEncodingMap.java` (or the write path)

An important spec requirement: files that only use codecs 0-4 (GZIP, BZIP2, LZMA, RANS4x8) should be written as CRAM 3.0, not 3.1, for maximum backward compatibility. The `fast` profile uses only GZIP (codec 1), so it should produce CRAM 3.0 files.

8a: Add a method to determine the minimum required CRAM version

```
public CRAMVersion getMinimumRequiredVersion() {
    // Scan all compressors in the encoding map
    // If any use codec IDs 5-8 (RANSNx16, Range, FQZComp, NameTokeniser),
    // the file must be CRAM 3.1
    for (ExternalCompressor compressor : getAllCompressors()) {
        final int methodId = compressor.getMethod().getMethodId();
        if (methodId >= 5 && methodId <= 8) {
            return CramVersions.CRAM_v3_1;
        }
    }
    return CramVersions.CRAM_v3;
}
```

8b: Wire version negotiation into the write path

In the CRAM writer (likely `CRAMContainerStreamWriter` or `CRAMFileWriter`), after constructing the encoding map but before writing the file header:

```
final CRAMVersion effectiveVersion =
encodingMap.getMinimumRequiredVersion();
// Use effectiveVersion instead of DEFAULT_CRAM_VERSION for the file
header
```

This means:

- `fast` profile → writes CRAM 3.0 (only uses GZIP)
- `normal` profile → writes CRAM 3.1 (uses RANSNx16, Name Tokeniser)
- `small` profile → writes CRAM 3.1 (uses RANSNx16, FQZComp, Name Tokeniser)
- `archive` profile → writes CRAM 3.1 (uses Range, FQZComp, Name Tokeniser)

Task 9: Tests

9a: Profile selection unit tests

File: new `src/test/java/htsjdk/samtools/cram/structure/EncodingProfileTest.java`

- Verify that setting each profile on `CRAMEncodingStrategy` applies the correct defaults (readsPerSlice, gzipCompressionLevel)
- Verify that `CompressionHeaderEncodingMap` constructed with each profile uses the expected codecs for each data series
- Verify that individual settings can override profile defaults

9b: Profile round-trip tests

For each profile (`fast`, `normal`, `small`, `archive`):

- Write a CRAM file using the profile
- Read it back with htsjdk — verify record fidelity
- Verify the output file has the correct CRAM version (3.0 for fast, 3.1 for others)

9c: Cross-profile compatibility tests

- Write with `archive` profile → read back successfully
- Write with `fast` profile → read back successfully
- Write with each profile → read with samtools (interop, requires samtools)

9d: Version negotiation tests

- Verify `fast` profile produces CRAM 3.0 header
- Verify `normal`/`small`/`archive` produce CRAM 3.1 header
- Verify custom encoding maps with only GZIP produce 3.0
- Verify custom encoding maps with any 3.1 codec produce 3.1

9e: Stripe encoding integration tests

- Write CRAM 3.1 files that use stripe-encoded blocks
- Read back and verify fidelity
- Verify samtools can read htsjdk-written files that use stripe

9f: Compression ratio comparison tests (optional, informational)

Not correctness tests, but useful for validation:

- Compare compression ratios across profiles for the same input
- Verify that `archive` < `small` < `normal` < `fast` in output size (on typical data)
- Compare htsjdk ratios against samtools ratios for the same profiles

---

Task 10: Update Existing Tests

10a: CRAMVersionTest

File: `src/test/java/htsjdk/samtools/cram/CRAMVersionTest.java`

Update or add tests to verify that the default profile (`NORMAL`) still produces CRAM 3.1.

10b: CRAM31Tests

File: `src/test/java/htsjdk/samtools/cram/CRAM31Tests.java`

Extend the existing samtools fidelity tests to cover all profiles. The data provider should include profile as a parameter:

```
@DataProvider
public Object[][] cram31ProfileTests() {
    return new Object[][] {
        { testInput, testRef, EncodingProfile.NORMAL },
        { testInput, testRef, EncodingProfile.SMALL },
        { testInput, testRef, EncodingProfile.ARCHIVE },
        // FAST produces 3.0, test separately
    };
}
```

10c: HtsCRAMCodec31Test

File: `src/test/java/htsjdk/beta/codecs/reads/cram/HtsCRAMCodec31Test.java`

Add a test that sets the profile via `CRAMEncoderOptions`:

```
new CRAMEncoderOptions()
    .setReferencePath(referencePath)
    .setEncodingProfile(EncodingProfile.SMALL);
```

What Phase 3 Does NOT Change

- **FQZComp parameter tuning** — Phase 2 provides a single default preset. Advanced preset selection (position context, delta context, multiple param blocks, dedup) is a future enhancement.
- **NameTokenisationEncode optimizations** — DUP_PREVIOUS_STREAM and all-MATCH stream elision remain unimplemented. These are compression ratio improvements, not correctness issues.
- **Per-data-series codec override API** — Users who want fine-grained control beyond profiles must use `setCustomCompressionHeaderEncodingMap()`. A per-series override API is a possible future enhancement.
- **Compression level within profiles** — htslib allows `-1` through `-9` within each profile, giving finer control. htsjdk profiles set a single configuration; sub-levels within profiles are a future enhancement.

Dependencies

- **Phase 1** merged (CRAM 3.1 write infrastructure)
- **Phase 2** merged (FQZComp encoding — needed for `small` and `archive` profiles)
- **BZIP2 and LZMA compressors** — already implemented (`BZIP2ExternalCompressor`, `LZMAExternalCompressor`)
- **Range encoder** — already implemented (`RangeEncode.java`, minus stripe which is addressed here)

File Change Summary

Part A (Stripe) — 8 files

File	Change
`RANSNx16Encode.java`	Add `encodeStripe()`, remove throw
`RangeEncode.java`	Add `encodeStripe()`, remove throw
`NameTokenisationEncode.java`	Add stripe flag combinations to `tryCompress()`
`RansTest.java`	Convert reject test to round-trip test
`RangeTest.java`	Convert reject test to round-trip test
`RANSInteropTest.java`	Enable stripe interop round-trip
`RangeInteropTest.java`	Enable stripe interop round-trip
`StructureTestUtils.java`	Remove stripe exclusions if present

Part B (Profiles) — 10+ files

File	Change
`CRAMEncodingStrategy.java`	Add `EncodingProfile` enum, profile field, apply defaults

File	Change
`CompressionHeaderEncodingMap.java`	Profile-aware `initializeDefaultEncodings()`, profile-aware `getBestExternalCompressor()`, add Range/BZIP2/LZMA helpers
`CRAMEncoderOptions.java`	Add profile field
`CRAMEncoder.java`	Thread strategy through to `CRAMFileWriter`
`SAMFileWriterFactory.java`	Add `setCRAMEncodingStrategy()`, wire through to writer
`CRAMFileWriter.java`	Ensure strategy flows to `CompressionHeaderEncodingMap` (may already work)
`EncodingProfileTest.java` (new)	Profile unit tests
`CRAM31Tests.java`	Profile fidelity tests
`HtsCRAMCodec31Test.java`	Plugin framework profile tests
`CRAMVersionTest.java`	Version negotiation tests

Risk Assessment

Task	Effort	Risk	Notes
Task 1-2 (stripe encode)	Medium	Low	Algorithm is simple (transpose + recursive compress). Decoders are the reference.
Task 3 (stripe in tok3)	Low	Low	Adding flag combinations to existing brute-force search
Task 4 (stripe tests)	Low	Low	Converting reject tests to round-trip
Task 5 (profile enum)	Low	Low	Simple enum + defaults
Task 6 (API surface)	Medium	Medium	Touches user-facing APIs; backward compatibility matters
Task 7 (profile-aware map)	High	Medium	Core change — must wire 4 profiles correctly
Task 8 (version negotiation)	Medium	Medium	Must not break existing 3.0 consumers
Task 9-10 (tests)	Medium	Low	Many test dimensions but straightforward

Total estimated scope: ~500-800 lines for stripe, ~500-800 lines for profiles, ~500 lines of tests.