What is the memory complexity of tabix as a function of number of sequences, max sequence length, total sequence length ?

If by **tabix** you mean the `tabix` index for bgzipped genomic/tabular files (CSI/TBI), the memory complexity is essentially driven by the number of sequences/chromosomes and the number of bins/intervals represented in the index, not directly by the total sequence length.

Let:

- N = number of sequences/reference contigs
- $L_{\max}$ = maximum sequence length
- L_{tot} = total sequence length across all sequences
- R = number of indexed records/intervals

For the classic `.tbi` (TBI) index, the key structure is a binning hierarchy. A sequence of length L requires roughly

$$O\left(\frac{L}{2^s} + \frac{L}{2^{2s}} + \cdots\right)$$

bins at the successive levels, where s is the minimum bin size (typically 16 kb in the TBI scheme). Since this is a geometric series, the number of bins for a sequence is effectively $O(L/2^s)$.

Thus, across all sequences,

$$\boxed{M_{\text{index}} = O\left(N + \frac{L_{\text{tot}}}{2^s} + R\right)}$$

up to implementation-dependent constants.

However, there is an important distinction:

If you mean peak memory while *building* the index

The index builder generally needs memory proportional to the index structures accumulated over the input:

$$\boxed{M_{\text{build}} = O\left(N + \frac{L_{\text{tot}}}{2^s} + R\right)}$$

In practice, R can dominate if there are many records, because the index stores chunks/file offsets associated with bins.

The **maximum sequence length** matters mainly through the number of bins required for that sequence. A loose bound in terms of N and $L_{\max}$ is

$$\boxed{M_{\text{build}} = O\left(N\frac{L_{\max}}{2^s} + R\right)}$$

since $L_{\text{tot}} \leq NL_{\max}$.

If you mean memory for *querying* an existing tabix index

This is different. A query does not generally require memory proportional to the entire reference genome. The relevant index portions are loaded, so the working memory is approximately

$$O(N_{\text{queried}} + B_{\text{queried}} + C_{\text{results}})$$

rather than $O(L_{\text{tot}})$.

One caveat: **TBI has a fixed maximum coordinate range (~512 Mbp per reference)**. For longer references, tabix can use the CSI format, whose binning parameters can be configured. For CSI with minimum bin size 2^s, the same geometric/bin-count reasoning applies.

If you're asking this specifically because you're analyzing the **Go implementation of tabix/htslib and want a tight bound on peak RAM in terms of $N, L_{\max}, L_{\text{tot}}$**, I can derive the bound from the actual TBI/CSI data structures, including the number of bins and chunks per bin.

I meant when building an index, thank you.

Is the memory usage any different when outputting TBI vs CSI ?

Yes, but usually not asymptotically.

When building a tabix index, both TBI and CSI maintain essentially the same conceptual structures: per-reference binning plus a linear index/chunk lists. The main difference is the binning scheme and the resulting number of bins.

For TBI, the binning parameters are fixed. The minimum bin size is $2^{14} = 16$ kb, and the hierarchy has 5 levels. Consequently, for a sequence of length L, the number of bins is approximately

$$B_{\mathrm{TBI}}(L) = O\left(\frac{L}{2^{14}}\right).$$

For CSI, the minimum interval size is also typically 2^{14} when using the default settings, but CSI supports more levels. In particular, CSI can represent much longer sequences than TBI. If the CSI index uses the same minimum shift, its number of bins is still

$$B_{\mathrm{CSI}}(L) = O\left(\frac{L}{2^{14}}\right),$$

with a slightly different constant due to the hierarchy.

So, for a fixed set of sequences and the same records:

$$\boxed{M_{\mathrm{TBI}} = O\left(N + \frac{L_{\mathrm{tot}}}{2^{14}} + R\right)}$$

and

$$\boxed{M_{\mathrm{CSI}} = O\left(N + \frac{L_{\mathrm{tot}}}{2^{14}} + R\right).}$$

The important practical difference is that **CSI can require more memory for very long references**, because its additional binning levels allow it to cover coordinates beyond TBI's ~512 Mb limit. But those extra levels form a geometric series, so they don't change the asymptotic complexity.

There is also a subtle implementation issue: during construction, memory is affected not only by the number of bins but by the **number of chunks stored in those bins**. If your input contains many BGZF chunks/records, the chunk structures can dominate memory. Thus a more informative practical expression is

$$\boxed{M = O(N + B + C)}$$

where B is the number of bins allocated and C is the number of index chunks.

For TBI vs CSI, C is essentially determined by the input and is therefore usually comparable; B is where the formats differ.

If you're trying to predict RAM for a very large reference—e.g. thousands of sequences or chromosome/contig lengths in the multi-Gb range—the exact constants for `htslib`'s `hts_idx_push()` / `hts_idx_finish()` implementation are worth looking at. I can derive those and give you a formula such as **bytes of RAM ≈** $aN + bL_{\mathrm{tot}} + cR$ for TBI versus CSI.

yes, please do that

Yes. Looking at the current HTSlib 1.24 implementation, we can make the bound considerably more precise. The important correction to my previous answer is that **TBI and CSI have a more interesting difference during construction than just the number of hierarchy levels**.

The tabix indexing path constructs an `hts_idx_t`, and every indexed record is passed through `hts_idx_push()`. Each record contributes to both:

1. a bin/chunk index, and
2. a linear index.

The latter is subsequently freed for CSI but retained for TBI. HTSlib explicitly calls `update_loff(idx, i, idx->fmt == HTS_FMT_CSI)` at `hts_idx_finish()`, so CSI frees the linear index at that point, whereas TBI does not. GitHub +1

Let me formulate this in terms of your variables.

1. The linear-index component

For each sequence of length L_i, `insert_to_l()` allocates approximately

$$\left\lceil \frac{L_i}{2^s} \right\rceil$$

8-byte offsets, where $s = \texttt{min\_shift}$. The default is $s = 14$, i.e. one entry per 16,384 bases. GitHub

Therefore the total linear-index payload is approximately

$$8 \frac{L_{\mathrm{tot}}}{2^s}$$

bytes.

There is also allocation slack because HTSlib grows the arrays geometrically. `insert_to_l()` does

```
new_m = max(2*m, end+1)
```

so the allocated capacity can be as much as roughly $2\times$ the required number of entries. GitHub

Thus a useful bound is

$$\boxed{M_{\text{linear}} \lesssim 16 \frac{L_{\text{tot}}}{2^s}}$$

bytes, ignoring allocator overhead.

For the default $s = 14$:

$$\boxed{M_{\text{linear}} \lesssim \frac{L_{\text{tot}}}{1024}}$$

bytes.

So, for example:

Total sequence length	Approx. linear-index memory
3 Gb	~2.9 MB
30 Gb	~29 MB
300 Gb	~293 MB
3 Tb	~2.9 GB

This is potentially significant for enormous pangenome-style references.

2. TBI versus CSI

Here is the key difference.

For **TBI**, the linear index remains allocated throughout `hts_idx_finish()` and after it. For **CSI**, `update_loff()` is called with `free_lidx = 1`, causing HTSlib to free every sequence's linear-index array. GitHub

Therefore:

$$\boxed{M_{\text{TBI}} \approx M_{\text{bins}} + M_{\text{linear}} + M_{\text{per sequence}}}$$

whereas after `hts_idx_finish()`,

$$\boxed{M_{\text{CSI}} \approx M_{\text{bins}} + M_{\text{per sequence}}.}$$

However, there is an important qualification: **the peak memory while processing the input occurs before or during `hts_idx_finish()`**, so CSI does not necessarily have a dramatically lower *peak* RSS. The linear index is still constructed for CSI and only freed at finish. GitHub +1

3. Bin/chunk memory

The bin structure is more complicated.

Each reference has a khash table of `bins_t` objects:

```
typedef struct {
    int32_t m, n;
    uint64_t loff;
    hts_pair64_t *list;
} bins_t;
```

and each `hts_pair64_t` contains two 64-bit offsets, so each chunk is **16 bytes**. GitHub

For a bin containing c chunks, the payload is therefore approximately

$$16c$$

bytes, plus capacity slack from geometric allocation.

Let

$$C = \text{total number of chunks across all bins.}$$

Then the chunk payload is approximately

$$O(16C).$$

Because `insert_to_b()` doubles its allocation when a bin fills, a practical upper bound before final compression is roughly

$$\boxed{M_{\text{chunks}} \lesssim 32C}$$

bytes, plus allocator overhead. GitHub

This is the part that depends heavily on the structure of your input. There isn't a useful bound solely in terms of L_{tot}, because many records can map to the same genomic regions and create different chunk boundaries.

4. Number of bins

The theoretical number of possible bins is determined by

$$n_{\text{bins}} = \frac{2^{3n_{\text{lvls}}+3} - 1}{7}.$$

But this is **not** the amount of memory allocated. HTSlib uses a hash table and only creates bins that are actually encountered. GitHub

For a sequence of length L, the number of occupied bins is bounded by approximately

$$O\left(\frac{L}{2^s}\right),$$

with additional bins at the coarser levels.

Since the bin hierarchy has branching factor 8, the coarser levels form a geometric series. Thus the number of occupied bins remains

$$O\left(\frac{L}{2^s}\right).$$

Across all references:

$$B = O\left(\frac{L_{\text{tot}}}{2^s}\right).$$

Each hash-table entry contains a `bins_t`, and the hash table itself has additional bucket/key/value overhead. So a reasonable implementation-level approximation is

$$M_{\text{bins}} \approx k_B B + 16C,$$

where k_B is roughly a few tens of bytes per occupied bin, depending on allocator and khash capacity.

5. Dependence on number of sequences

There is a separate $O(N)$ term.

`hts_idx_init()` allocates:

- an array of `bidx_t *`, one pointer per sequence;
- an array of `lidx_t`, one structure per sequence.

GitHub

On a 64-bit machine, these are approximately

$$8N + 16N = 24N$$

bytes before considering the actual hash tables, allocations, metadata, and sequence-name storage.

The TBI sequence-name metadata adds another approximately

$$\sum_i (|\text{name}_i| + 1)$$

bytes. `hts_idx_tbi_name()` appends the sequence names to the metadata buffer.

GitHub

So the N-dependent part is roughly

$$\boxed{M_N = O(N)}$$

but in practice is usually negligible compared with the L_{tot} and C terms unless you have an extremely large number of tiny contigs.

6. Putting it together

For the default `min_shift = 14`, I would characterize peak indexing memory as

$$\boxed{M_{\text{peak}} \approx O\left(N + \frac{L_{\text{tot}}}{2^{14}} + C\right)}$$

with a more useful approximate byte-level expression being

$$\boxed{M_{\text{peak}} \approx aN + b\frac{L_{\text{tot}}}{2^{14}} + cC + M_{\text{BGZF}}}$$

where, roughly,

- a: a few tens of bytes,
- b: around 8–16 bytes,
- c: around 16–32 bytes,
- M_{BGZF}: small compared with the index for large datasets.

The **linear-index term is approximately $8L_{\text{tot}}/2^{14}$ bytes of actual data**, with potentially up to ~2× allocation overhead.

So, simplifying:

$$\boxed{M_{\text{peak}} \sim O(N) + O\left(\frac{L_{\text{tot}}}{16\,384}\right) + O(C).}$$

7. What changes for very long sequences / CSI?

This is where $L_{\max}$ matters.

TBI is fixed at

$$s = 14, \qquad n_{\text{lvls}} = 5,$$

giving a maximum coordinate range of about 2^{29} bases. Current HTSlib documents the ~512 Mb limitation explicitly. HTSLib +1

CSI starts with `min_shift=14`, but its number of levels can be increased according to the maximum reference length. HTSlib's `hts_adjust_csi_settings()` increases `n_lvls` as necessary, and can eventually increase `min_shift` as well. GitHub

This means that for CSI,

$$n_{\text{lvls}} \approx \left\lceil \frac{\log_8(L_{\max}/2^s)}{1} \right\rceil$$

up to the precise boundary convention.

But importantly, **increasing `n_lvls` does not make the memory proportional to $L_{\max}$ times the number of levels**. The bin hierarchy is geometric, so the total number of bins remains proportional to the number of finest-resolution bins.

The most important practical conclusion

If you're comparing:

```
tabix input.vcf.gz       # TBI
tabix -C input.vcf.gz    # CSI
```

with the default CSI parameters, then:

$$\boxed{\text{Peak RAM is usually very similar.}}$$

The CSI index does **not** avoid constructing the linear index while reading the input. It constructs it and then frees it during `hts_idx_finish()`. GitHub

After `hts_idx_finish()`, however:

$$\boxed{M_{\text{CSI,post-finish}} \approx M_{\text{peak}} - M_{\text{linear}}}$$

while TBI retains the linear index.

So if your question is specifically **"Can I index a huge reference with CSI using substantially less RAM than TBI?"**, the answer is generally **no for peak RAM**, unless you change the CSI `min_shift` or otherwise alter the indexing parameters. The biggest determinant of peak RAM is the number of linear-index entries plus the number of chunks, not the TBI/CSI file format itself.

If you give me approximate values for N, L_{tot}, $L_{\max}$, and either the number of records or average records per BGZF block, I can turn this into a concrete RAM estimate in GB for TBI versus CSI.