File (diagram) path: `/mnt/data/a7f15157-d103-4776-a7c2-c4e2a27c82dc.png`

Nice constraint — targeting low RAM makes the system much more production-realistic. Below I give a practical, engineer-grade plan (with formulas, concrete parameters, and exact code/design changes you should make) so your pipeline stays **≤ 2GB** for ≤100k docs and **≤ 4GB** for bigger corpora.

Short answer (summary)

Yes — you can meet those memory targets by:

- keeping compact numeric tables (IDF, DL) in memory,
- never keeping full forward/inverted indexes fully in RAM,
- writing/reading posting lists on disk in **barrels** and using an **LRU cache** sized by the target RAM,
- using 32-bit integers and compression (delta + vbyte) for postings,
- streaming processing (write segments to disk and free memory),
- using `mmap` or `seek+read` to avoid copying.

Below: exact steps, formulas, and recommended parameter values.

---

1) Memory model & quick formulas

Estimate the big contributors:

- `T` = number of distinct terms
- `N` = number of documents
- `avgDL` = average tokens/doc

- `P` = total postings (sum of dfs) ≈ average df * T

Table sizes (in bytes):

- IDF array (double) ≈ `T * 8` bytes (use float32 to half it → `T * 4`)
- DL array (int32) ≈ `N * 4`
- Term→barrel mapping (struct with two int32) ≈ `T * 8`
- In-memory cached postings (variable) — this is the big one.

Example numbers (practical):

- Suppose `T = 200k` terms, `N = 100k` docs.
  - IDF (float32): 200k * 4 = **0.8 MB**
  - DL (int32): 100k * 4 = **0.4 MB**
  - Lexicon extra arrays: ~ a few MB
  - So **core fixed tables << 10 MB**. The dominant RAM is caches.

Posting entry size (uncompressed)

- doc_id (4 bytes) + tf (2-4 bytes) + pos_count (4) + positions (pos_count * 4)
- On average if you store doc_id+tf only ≈ 8 bytes per posting. For 20M postings → 160 MB uncompressed.

Takeaways: fixed tables are tiny; the cache is where you control memory.

---

2) Design choices to hit RAM targets

A — Never keep whole ForwardIndex or InvertedIndex in RAM

Change: Stage 2 should **write forward index segments to disk immediately** (you already segment). Do **not** keep `documents` vector for all docs — flush segments and free memory.
Change: Stage 3 should **stream forward segments** and **write posting lists directly into barrels** (or write partial inverted segments to disk and barrelize) — do not accumulate full inverted index in memory.

B — Compact in-memory tables only

Keep in memory:

- `idf[]` (float32)
- `dl[]` (uint32)
- `lexicon` minimal arrays: term -> (term_id, df, barrel_id, posting_ptr) (store term strings only if needed; otherwise keep reverse map on disk)
  These together are under **~10–50 MB** for typical T,N.

C — Use barrels + LRU cache

- Partition inverted index into barrels (term-range).
- Keep an **LRU cache** of decompressed posting lists or whole barrels.
- Make cache size configurable; set defaults:
  - For datasets ≤100k docs: cache cap = **512 MB** (allows frequent terms to be hot)
  - For datasets >100k: cache cap = **1.5–2.5 GB** depending on target ≤4GB total.
  - Keep some headroom for program + OS (e.g., if limit is 2GB, cap cache at 1.6–1.8GB).

D — Use 32-bit integers and float32

- Use `int32` for doc_id, positions, and `float` (32-bit) for IDF and BM25 intermediate values. This halves memory vs 64-bit.

E — Compression for postings

- Delta encode docIDs then **vbyte** encode.
- For positions, store deltas per document and vbyte encode.
- Typical compression reduces posting bytes by 5–10× for docIDs, and smaller for positions.

F — Partial read & mmap

- When reading a posting, either `seek`+`read` the exact byte range (using `posting_ptr` and length) or `mmap` the barrel file and parse in-place. `mmap` avoids copying and lets OS page in only needed ranges.

---

3) Concrete parameter recommendations (to meet your caps)

For ≤100k docs (target ≤2GB RAM):

- Use float32 for IDF.
- Use int32 for DL.
- Barrel term-range size: `T_per_barrel = 10k` terms → with 200k terms → 20 barrels.
- LRU cache size: **512 MB** (default). Limit max open barrel pages to maybe 4–8.
- Compression: always write postings compressed (delta + vbyte).
- Max in-memory decompressed postings at a time: cap to e.g., 50–200 MB per posting; prefer storing compressed and decode on demand.

For >100k docs (target ≤4GB RAM, e.g., 200k docs):

- LRU cache size: **1.5 GB** (or configurable), with rest of memory for process/OS.
- Barrel size: increase to `T_per_barrel = 20k` if T grows, to reduce number of files. But avoid extremely large barrels that don't fit in cache.

4) Exact code/design changes to your pipeline

Stage2 (ForwardIndex.build)

- Currently: you push each `ForwardDoc` into `documents` vector.
- **Change**: after building a document's arrays, immediately serialize it to a forward segment file (binary), then `clear()` the per-doc vectors. Only keep a small in-memory buffer (segment_size) before flushing.
- Advantage: memory limited to `segment_size` docs. Use `segment_size = 1k` (already used).

Stage3 (InvertedIndex.build)

- Current design builds all postings in `index[term_id]` in RAM.
- **Change**: implement *external merging* pattern:
    1. Read forward segments one-by-one.
    2. For each doc, write temporary posting fragments per term to small per-term temp files or an on-disk buffer sorted by term id (or use a map of small memory buffers flushed to disk when big).
    3. Merge fragments for each term sequentially and write directly into **barrel files** (append mode). Update lexicon with barrel_id + offset immediately.
    4. Free memory of fragment buffers as soon as merged.
- Alternatively: build inverted index term-by-term by scanning forward index multiple times (slower but low memory).

Stage6 (Barrels)

- Implement barrel writer to write compressed posting lists.

- When writing per-term posting list, write compressed bytes and record `byte_len`. Store `posting_ptr` and `posting_len` (useful to read exact bytes).

Query Engine (Stage5)

- Load `idf[]`, `dl[]`, and lexicon minimal arrays into memory.
- For each query term: find lexicon entry -> `barrel_id`, `posting_ptr`, `posting_len`.
- Request `BarrelReader` to read compressed bytes (seek `posting_ptr` for `posting_len` bytes) and decompress into postings. Use the LRU to keep recently decompressed posting lists.

---

5) LRU cache design (practical)

- Cache keyed by `barrel_id` or by `(term_id)` (store decompressed posting lists).
- Use an approximate size counter (total bytes of decompressed postings) and evict least recently used entries until under cap.

Implementation structure:

struct CacheEntry { int term_id; size_t size_bytes; vector<Posting> postings; };
unordered_map<int, list_iter> table;
list<CacheEntry> lru_list; // front = most recent
size_t current_bytes;

-
- For barrel-level caching, store raw compressed bytes to keep memory lower (decompress on demand if needed).

---

6) Testing & validation steps (must do)

1. **Unit tests**: small dataset → build pipeline with streaming and barrels, verify posting equality with your previous in-memory build.
2. **Memory stress test**:
   - Run pipeline on sample medium sizes (10k, 50k, 100k, 200k). Monitor Resident Set Size (RSS).
   - On Linux: `time` + `ps`/`top`; on Windows Task Manager or `Get-Process` PowerShell.
3. **Query-time RAM test**:
   - Run many queries (mix hot and cold terms) and record memory usage; ensure it stays under cap.
   - If memory spikes, tighten cache cap.
4. **I/O test**:
   - Measure bytes read per query with barrel vs reading whole inverted set.

---

7) Practical config defaults to include in code (config file or CLI)

```
# engine.conf
barrel_size_terms = 10000
cache_max_bytes = 536870912      # 512 MB default for <=100k
cache_max_bytes_large = 1610612736 # 1.5 GB for >100k
use_mmap = true
compression = delta_vbyte
idf_type = float32
dl_type = uint32
```

Let your `main.cpp` or a config reader set cache based on `N` (document count) at startup.

---

8) Monitoring & fail-safe

- Expose a `/metrics` endpoint (Prometheus) with:
  - `mem_used_bytes`, `cache_hits`, `cache_misses`, `open_barrels`, `bytes_read_per_query`.
- If `mem_used_bytes` approaches cap, start evicting more aggressively or disable preloading.

---

9) Quick checklist to implement right away

1. Replace in-memory inverted-index builder with streaming/barrel-writing approach.
2. Always write postings compressed (delta+vbyte). Record `posting_len`.
3. Update lexicon entries to include `barrel_id`, `posting_ptr`, `posting_len`. Save lexicon binary+json.
4. Implement `BarrelReader` that `seek`+`read` exact bytes and decompress. Optionally `mmap`.
5. Implement LRU cache with configurable cap. Default: 512MB for ≤100k, 1.5GB for >100k.
6. Instrument memory and I/O metrics and test on 10k/50k/100k.

---

10) Final practical numbers & goal check

- With compression and streaming, for **≤100k documents**, target **RAM ≤2GB** is conservative and achievable even with a fairly large cache (512MB–1GB) if you:
  - keep lexicon + IDF + DL in memory (~<20MB),
  - keep an LRU cache bounded to 512MB,
  - and stream everything else from disk.

- For **>100k docs**, increase cache to ~1.5–2.5GB depending on total RAM; still achievable to stay ≤4GB.

If you want, I will:

1. Produce the **concrete C++ changes** to Stage2/Stage3 to stream & barrelize (diff-ready).
2. Implement an **LRU cache** and `BarrelReader` code you can drop into your repo.
3. Add a **small benchmark harness** that runs queries and prints RSS and p95 latency.

Say **“generate streaming + barrels code”** or **“generate LRU + BarrelReader”** and I’ll write the exact header + `.cpp` files and show the minimal changes to `stage2`/`stage3` to meet the RAM constraints.