Lab: CPU RAG on SLURM — Build, Infer, & Monitor

Assistant Lecturer: Dr. Ahmed Métwalli • Last updated: August 9, 2025

(Goal & Rationale)

Goal (45–60 min). Implement a *minimal, CPU-only* Retrieval-Augmented Generation (RAG) pipeline that you can run in a SLURM cluster:

1. **Train the retriever** by fitting a TF–IDF index on a small corpus (this is the "training" step for retrieval).
2. **Generate answers** by conditioning a small T5 model on retrieved passages (observe how retrieval changes answers).
3. **Parallelize inference** across processes with `torchrun` and **monitor** CPU/RAM in real time (understand scheduling and resource use).

Why. RAG is a core pattern in practical LLM systems: instead of memorizing all facts, we *retrieve* relevant context at query time and *ground* generations on it. This lab shows the end-to-end mechanics *without GPUs*, focusing on: reproducibility, resource allocation, and measurable speed/quality trade-offs.

Today's stack (CPU-friendly):

- **Corpus**: a slice of `ag_news` (passages).
- **Retriever**: TF–IDF (scikit-learn) with a token-overlap fallback.
- **Generator**: `google/flan-t5-small`.
- **Orchestration**: `salloc` and optional `torchrun`.

Background: TF–IDF and Modern Retrieval Alternatives

What is TF–IDF and why do we still use it?

Term Frequency–Inverse Document Frequency (TF–IDF) is a classic, lightweight way to score how important a term is to a document in a collection. It remains popular because it is:

- **Fast & CPU-friendly:** No neural model is required; vectorization and scoring are sparse linear algebra.
- **Explainable:** Scores come from transparent statistics (term counts and document frequencies).
- **Zero-training overhead:** Fitting the vectorizer is just counting; great for demos, baselines, and constrained compute.
- **Strong lexical baseline:** Works well whenever answers share *words* with queries.

Its main limitation is that it is *lexical*: it struggles with paraphrase and semantic similarity ("car" vs. "automobile").

Equations

Let t be a term, d a document, and N the number of documents. Define:

$$\mathrm{tf}(t,d) \;=\; \text{count of } t \text{ in } d \qquad \text{or} \qquad \mathrm{tf}(t,d) \;=\; 1+\log\big(\mathrm{count}(t,d)\big)$$

$$\mathrm{idf}(t) \;=\; \log\left(\frac{N+1}{\mathrm{df}(t)+1}\right)+1$$

where $\mathrm{df}(t)$ is the number of documents containing t. The TF–IDF weight is

$$w(t,d) \;=\; \mathrm{tf}(t,d)\cdot \mathrm{idf}(t).$$

Representing d and a query q as vectors of $w(t,\cdot)$, we rank documents for q by cosine similarity:

$$\mathrm{sim}(q,d) \;=\; \frac{\mathbf{v}_q \cdot \mathbf{v}_d}{\|\mathbf{v}_q\|\,\|\mathbf{v}_d\|}.$$

A tiny TF–IDF pipeline (conceptual)

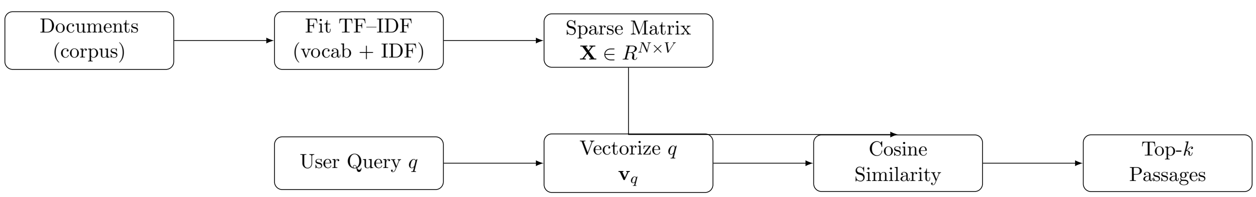

Figure 1: TF–IDF retrieval: fit once on the corpus, then score queries via cosine similarity.

How it compares to modern (heavier) alternatives

Family	**Example**	**Compute Profile**	**Notes / When to Use**
Lexical (sparse)	TF–IDF, BM25	CPU, very light	Strong baseline when lexical overlap is expected; easy to cache & ship.
Learned sparse	SPLADE, uniCOIL	Train + GPU helpful; CPU at query time	Sparse but semantic; better recall than pure lexical, still index-friendly.
Dense bi-encoders	DPR, SBERT, E5, bge	GPU to build embeddings; ANN at scale	Captures semantics; needs FAISS/HNSW/vector DB; good for paraphrase.
Late-interaction	ColBERT / v2	Heavier index; GPU helpful	High accuracy via token-level matching; larger storage/compute.
Cross-encoders (re-rank)	MiniLM-XE, MonoT5	Expensive per pair (often GPU)	Best precision for top-k re-ranking; low throughput; use after recall.

Takeaways.

- TF–IDF remains a **fast, transparent** baseline for RAG demos, CPU clusters, and sanity checks.

- For paraphrase/semantic matches, **dense retrieval** (bi-encoders) with **ANN indexes** (e.g., FAISS/HNSW) offers higher recall, at the cost of **GPU time** to embed corpora and **larger memory/storage**.
- High-precision pipelines often combine stages: *(lexical/dense) recall → cross-encoder re-rank.*

1 One-time Prep (5 min)

A. Activate your venv

```
source ~/llamaenv_local/bin/activate
python --version
```

B. Install minimal deps

```
pip install --upgrade pip
pip install "transformers==4.*" "datasets==2.*" "scikit-learn==1.*" "sentencepiece
    ==0.2.*"
```

C. Pre-cache models + dataset (faster runs)

```
python - <<'PY'
import os, datasets, transformers
os.environ["HF_HOME"] = os.path.expanduser("~/.cache/hf_rag")
datasets.load_dataset("ag_news", split="train[:2000]",
                      cache_dir=f"{os.environ['HF_HOME']}/ds")
transformers.AutoTokenizer.from_pretrained(
    "google/flan-t5-small", cache_dir=f"{os.environ['HF_HOME']}/tok")
transformers.AutoModelForSeq2SeqLM.from_pretrained(
    "google/flan-t5-small", cache_dir=f"{os.environ['HF_HOME']}/gen")
print("Cached: corpus + tokenizer + generator.")
PY
```

Listing 1: Caches into \$HOME/.cache/hf$_{r}ag$

2 Get an Interactive Slot (10–30 min)

```
# 2 CPUs, 2G RAM, 20 minutes
salloc -N1 -n1 -c2 --mem=2G -p parallel --time=00:20:00 --pty bash

# After it starts:
source ~/llamaenv_local/bin/activate
cd ~/project
export HF_HOME=$HOME/.cache/hf_rag
```

What these flags assign:

- `-N1`: one node. `-n1`: one *task* (process group).

- `-c2`: two CPUs for that task (room for two Python workers).
- `--mem=2G`: memory reservation per node.

3 RAG Script (with Code-Block Explanations)

Save as `labs/ragging/rag_example.py`:

```
#!/usr/bin/env python3
"""
Ultra-light CPU RAG:
- "Training" = fitting a TF-IDF vectorizer on a small corpus (fast on CPU).
- Retrieval: top-k passages by cosine similarity; fallback to token-overlap.
- Generation: FLAN-T5-Small conditioned on the retrieved context.
- Modes: single query (--query "...") or batched (--queries_file).
"""

import os, sys, argparse, math, time, re
from typing import List, Tuple

import torch
from datasets import load_dataset
from transformers import AutoTokenizer, AutoModelForSeq2SeqLM

# ---------------- 1) Normalization utilities ----------------
_ws = re.compile(r"\s+")
def norm(txt: str) -> str:
    """Lowercase + collapse whitespace -> helps both retrievers."""
    return _ws.sub(" ", txt.lower()).strip()

def simple_overlap_score(q: str, doc: str) -> float:
    """Fallback similarity: token Jaccard-like score."""
    qs = set(norm(q).split()); ds = set(norm(doc).split())
    if not qs or not ds: return 0.0
    inter = len(qs & ds); return inter / math.sqrt(len(qs) * len(ds))

# ---------------- 2) Retriever (TF-IDF or fallback) ----------
class Retriever:
    """
    On init: either fits a TF-IDF vectorizer (training) or
    falls back to a token-overlap heuristic if sklearn isn't present.
    """
    def __init__(self, docs: List[str]):
        self.docs = docs
        self.kind = "fallback"
        try:
            from sklearn.feature_extraction.text import TfidfVectorizer
            self.vec = TfidfVectorizer(max_features=20000, ngram_range=(1,2))
            self.mat = self.vec.fit_transform(docs) # <-- training step
            self.kind = "tfidf"
        except Exception as e:
            self.vec = None; self.mat = None
            print(f"[WARN] sklearn unavailable, using overlap fallback: {e}",
                  file=sys.stderr)
```

```
    def search(self, query: str, k: int = 3) -> List[Tuple[int, float]]:
        """Return top-k (doc_index, score) for a query."""
        if self.kind == "tfidf":
            import numpy as np
            qv = self.vec.transform([query])
            sims = (self.mat @ qv.T).toarray().ravel() # cosine on L2 rows
            topk = np.argsort(-sims)[:k]
            return [(int(i), float(sims[i])) for i in topk]
        else:
            scored = [(i, simple_overlap_score(query, d))
                      for i, d in enumerate(self.docs)]
            scored.sort(key=lambda x: -x[1])
            return scored[:k]

# ---------------- 3) Prompt builder -------------------------
def build_prompt(query: str, passages: List[str]) -> str:
    """
    Small, deterministic instruction. Keeps demo stable on CPU.
    """
    ctx = "\n\n".join(f"- {p}" for p in passages)
    return (f"Answer the question concisely using the context.\n"
            f"Context:\n{ctx}\n\nQuestion: {query}\nAnswer:")

# ---------------- 4) Main ----------------------------------
def main():
    ap = argparse.ArgumentParser()
    ap.add_argument("--subset", type=int, default=2000, help="Corpus size")
    ap.add_argument("--k", type=int, default=3, help="#passages retrieved")
    ap.add_argument("--query", type=str, default=None, help="Single query")
    ap.add_argument("--queries_file", type=str, default=None, help="File of queries")
    ap.add_argument("--batch", type=int, default=4, help="Batch size for file mode")
    ap.add_argument("--max_new_tokens", type=int, default=64)
    ap.add_argument("--dry_run", action="store_true", help="Use tiny slice")
    args = ap.parse_args()

    # ---- Assign cache directory (shared across steps) ----
    cache = os.environ.get("HF_HOME", os.path.join(os.getcwd(), ".hf_rag_cache"))

    # ---- 4.1 Load corpus (fast) ----
    split = f"train[:{min(args.subset, 2000)}]" if args.dry_run else f"train[:{args.
        subset}]"
    ds = load_dataset("ag_news", split=split, cache_dir=os.path.join(cache, "ds"))
    corpus = [f"{r.get('title','')} - {r['text']}".strip(" -") for r in ds]

    # ---- 4.2 Train retriever (fit TF-IDF) ----
    t0 = time.time()
    retr = Retriever(corpus)
    t_train = time.time() - t0
    print(f"[retriever] kind={retr.kind} trained_on={len(corpus)} docs in {t_train:.2f}s
        ")

    # ---- 4.3 Load generator (small T5) ----
    tok = AutoTokenizer.from_pretrained("google/flan-t5-small",
```

```
                                    cache_dir=os.path.join(cache, "tok"))
    gen = AutoModelForSeq2SeqLM.from_pretrained("google/flan-t5-small",
                                           cache_dir=os.path.join(cache, "gen"))
    gen.eval() # inference mode

    # ---- 4.4 One question -> one answer ----
    def answer_one(q: str) -> str:
        hits = retr.search(q, k=args.k)
        ctxs = [corpus[i] for i,_ in hits]
        prompt = build_prompt(q, ctxs)
        inputs = tok(prompt, return_tensors="pt")
        with torch.no_grad():
            out = gen.generate(**inputs, max_new_tokens=args.max_new_tokens)
        return tok.decode(out[0], skip_special_tokens=True)

    # ---- 4.5 Single-query path ----
    if args.query:
        a = answer_one(args.query)
        print("\nQ:", args.query)
        print("A:", a)
        sys.exit(0)

    # ---- 4.6 Batched path (supports torchrun sharding) ----
    if args.queries_file and os.path.exists(args.queries_file):
        with open(args.queries_file) as f:
            queries = [line.strip() for line in f if line.strip()]
        # Assignment via env: if torchrun is used, these are set.
        rank = int(os.environ.get("LOCAL_RANK", 0)) # this worker's index
        world = int(os.environ.get("WORLD_SIZE", 1)) # total workers
        shard = queries[rank::world] # strided partitioning
        print(f"[rank {rank}/{world}] processing {len(shard)} queries")

        for i in range(0, len(shard), args.batch):
            for q in shard[i:i+args.batch]:
                ans = answer_one(q)
                print(f"\nQ: {q}\nA: {ans}")
        sys.exit(0)

    print("Nothing to do: provide --query '...' or --queries_file path.")
    return

if __name__ == "__main__":
    main()
```

Listing 2: labs/ragging/rag$_{e}xample.py$

Syntax & block explanations (annotated):

- `#!/usr/bin/env python3`: portable shebang so the OS runs the file with your default Python 3.
- `docstring` under the shebang: human-readable description; tools can parse it.
- `import os, sys, argparse, ...`: standard-library modules first (I/O, CLI parsing, timers, regex), then third-party (`torch`, `datasets`, `transformers`).

- `from typing import List, Tuple`: type hints; helpful for editors and code clarity.
- **Regex & helpers**
  - `re.compile(r"\s+")` builds a compiled regex for "one or more whitespace".
  - `norm()` lowercases and collapses whitespace to stabilize similarity scoring.
  - `simple_overlap_score()` uses set intersections of tokens; the denominator $\sqrt{|Q| \cdot |D|}$ balances length.
- **Class** `Retriever`:
  - `__init__`: tries to import `TfidfVectorizer`. `fit_transform(docs)` is the *training* step (learns vocabulary & IDF); failure triggers a clean fallback.
  - `search()`: for TF–IDF, multiply document matrix by the query vector; for fallback, score via token overlap and sort.
  - `List[Tuple[int, float]]`: returns pairs (document index, score).
- **Prompting**:
  - `"\n\n".join(...)` builds a simple, deterministic context block; deterministic prompts reduce output variance on CPU.
  - f-strings (`f"...{var}..."`) perform in-place string interpolation.
- **Argparse & assignments**:
  - `--subset`: limits corpus; controls build time and memory.
  - `--k`: # of passages per query; quality vs. speed trade-off.
  - `--max_new_tokens`: generation length; the main latency knob.
  - `--dry_run`: `train[:2000]` cap for quick demo.
- **Caching & env**:
  - `HF_HOME` points the HuggingFace cache to a persistent location; we read it via `os.environ.get(...)`.
  - `cache_dir=os.path.join(cache, "tok"/"gen"/"ds")` keeps artifacts tidy.
- **Generation**:
  - `tok(prompt, return_tensors="pt")` tokenizes the prompt into PyTorch tensors.
  - `with torch.no_grad():  gen.generate(...)` disables gradients for faster, smaller inference.
  - `tok.decode(..., skip_special_tokens=True)` cleans special tokens like <s>.
- **Parallel batch path**:
  - `LOCAL_RANK`, `WORLD_SIZE` are exported by `torchrun` automatically.
  - `queries[rank::world]` is *strided sharding*: worker 0 handles lines $0, \text{world}, 2\cdot\text{world}, \ldots$
- **Guard block** `if __name__ == "__main__":`: only runs `main()` when the file is executed as a script (not imported).

4 Tiny Batch of Queries (optional)

```
What happened in the sports world?
Which company announced a new product?
How did the stock market perform?
Describe a political event mentioned.
```

Listing 3: labs/ragging/queries.txt

5 Run & Observe

A. Single query (fastest path)

```
export HF_HOME=$HOME/.cache/hf_rag
python labs/ragging/rag_example.py --dry_run --query "What is deep learning?"
```

B. Parallel batch with `torchrun` (2 processes)

```
torchrun --nproc_per_node=2 labs/ragging/rag_example.py \
  --dry_run --queries_file labs/ragging/queries.txt --k 3 --max_new_tokens 64
```

Assignments & environment variables (what gets set where):

- `HF_HOME` (you set): base directory for tokenizer/model/dataset cache.
- `WORLD_SIZE`, `LOCAL_RANK` (set by `torchrun`): total processes and this process's index.
- `MASTER_ADDR`, `MASTER_PORT` (optional for single node): distributed rendezvous address/port; `torchrun` can infer defaults on one node.

6 Live Monitoring During the Lab

A. Slurm accounting (RAM/CPU while running)

```
watch -n2 "sstat -j $SLURM_JOB_ID --format=JobID,MaxRSS,AveCPU,MaxVMSize"
```

B. Process view inside the allocation

```
watch -n2 "ps -u $USER -o pid,pcpu,pmem,etime,cmd | grep python | grep -v grep"
```

C. Queue + partitions

```
watch -n5 'squeue -u $USER -o "%.9i %.2t %.10M %.18R %.12C %j"'
watch -n10 'sinfo -o "%P %.6t %.6D %.9m"'
```

7 What Counts as "Training" Today?

- **Retriever training** = fit TF–IDF on the corpus (learns vocabulary & IDF weights).
- **Generator fine-tuning** = *not* performed; we use pre-trained FLAN-T5-Small.
- **Parallel inference** = shard queries by `LOCAL_RANK` over `WORLD_SIZE`.

8 Troubleshooting

Symptom	**Fix**
`ModuleNotFoundError: sklearn`	`pip install scikit-learn==1.*`; fallback still works but is weaker.
Slow first run (downloads)	Ensure `HF_HOME` is set; run the pre-cache step.
`torchrun` hangs	Stay single-node; use only `--nproc_per_node=2`.
OOM	Reduce `--subset`, `--k`, or `--max_new_tokens`.

9 Extensions (optional)

- Swap TF–IDF for FAISS + MiniLM embeddings (still CPU, slightly heavier).
- Add retrieval metrics (recall@k) and simple latency/throughput logging per rank.
- Demonstrate 2-node query sharding (advanced).