#!/usr/bin/env python3

# -\*- coding: utf-8 -\*-

"""

Module-1: Data Ingestion for VIT Smart FAQ (Vellore MVP)

- Input : data/raw/ (drop all PDFs/DOCX here)

- Output : data/processed/chunks.jsonl (one JSON per chunk)

data/processed/catalog.csv (one row per source file)

Requires: pdfplumber, python-docx, langchain, tqdm

pip install pdfplumber python-docx langchain tqdm

"""

import os, re, json, csv, argparse, hashlib, datetime, sys

from pathlib import Path

from typing import Dict, Any, List, Optional

# PDF & DOCX extractors

import pdfplumber

try:

import docx # python-docx

HAS\_DOCX = True

except Exception:

HAS\_DOCX = False

# Chunking (heading-aware)

from langchain.text\_splitter import RecursiveCharacterTextSplitter

from tqdm import tqdm

# ------------------------------- Cleaning ---------------------------------- #

PAGE\_PATTERNS = [

r"Page\s+\d+\s\*(?:of|/)\s\*\d+",

r"^\s\*Vellore\s+Institute\s+of\s+Technology.\*$", # common letterheads (broad)

r"^\s\*L.?&.?T\s\*EduTech.\*$",

]

PAGE\_REGEXES = [re.compile(p, re.IGNORECASE | re.MULTILINE) for p in PAGE\_PATTERNS]

def clean\_text(raw: str) -> str:

"""Light cleanup: remove common headers/footers, fix hyphenation,

collapse single newlines (keep double as paragraph), normalize spaces."""

if not raw:

return ""

# Normalize newlines

txt = raw.replace("\r\n", "\n").replace("\r", "\n")

# Drop frequent page artifacts

for rgx in PAGE\_REGEXES:

txt = rgx.sub("", txt)

# Remove lines that are just page numbers or whitespace

txt = re.sub(r"^\s\*\d+\s\*$", "", txt, flags=re.MULTILINE)

# Fix hyphenated line breaks: "admis-\nsion" -> "admission"

txt = re.sub(r"(\w)-\n(\w)", r"\1\2", txt)

# Keep paragraph breaks but join within paragraphs:

# replace single \n with space; keep double \n\n

txt = re.sub(r"(?<!\n)\n(?!\n)", " ", txt)

# Collapse 3+ newlines to 2

txt = re.sub(r"\n{3,}", "\n\n", txt)

# Collapse repeated spaces/tabs

txt = re.sub(r"[ \t]{2,}", " ", txt)

# Trim

txt = txt.strip()

return txt

# ------------------------------ Metadata ----------------------------------- #

def \_guess\_ay\_from\_name(name: str) -> Optional[str]:

# Try patterns like 2025-26, 2025–26, 25-26, 2025\_26

name = name.replace("\_", "-")

m = re.search(r"(20\d{2})\D{0,3}(\d{2})", name)

if m:

return f"{m.group(1)}-{m.group(2)}"

return None

def \_bool(val):

return True if val else False

def map\_metadata(filename: str) -> Dict[str, Any]:

"""Infer metadata from filename (lowercased)."""

n = filename.lower()

meta: Dict[str, Any] = {

"campus": "Vellore", # MVP fixed

"domain": None, # UG | PG | Research | Hostel | NRI-UG | Foreign-UG | NRI-Intl

"category": None, # Fee Structure | Eligibility | Documents Required | Courses Offered | Academic Rules | Hostel Norms | Refund Policy | Contacts | FAQ | Admission Process | Exam Pattern

"program": None, # B.Tech | M.Tech | MCA | M.Sc | Ph.D. | Direct Ph.D.

"gender": None, # Male | Female (hostel)

"student\_year": None, # First-Year | Senior (hostel)

"currency": None, # INR | USD

"audience": None, # NRI | Foreign (if needed)

"ay": \_guess\_ay\_from\_name(n),

"stable\_flag": "stable", # fees/refunds/exam windows -> volatile

}

# --- Domain ---

if "vitree" in n or "phd" in n or "ph.d" in n or "research" in n:

meta["domain"] = "Research"

if "direct" in n:

meta["program"] = "Direct Ph.D."

else:

meta["program"] = "Ph.D."

elif n.startswith("mh") or n.startswith("lh") or "hostel" in n:

meta["domain"] = "Hostel"

elif "nri" in n and "ug" in n:

meta["domain"] = "NRI-UG"

meta["audience"] = "NRI"

elif "foreign" in n and "ug" in n:

meta["domain"] = "Foreign-UG"

meta["audience"] = "Foreign"

elif "international" in n and "admissions" in n and "fee" in n:

meta["domain"] = "NRI-Intl"

meta["audience"] = "Foreign"

elif "pg" in n or "mtech" in n or "m.tech" in n or "mca" in n or "msc" in n or "m.sc" in n:

meta["domain"] = "PG"

elif "ug" in n:

meta["domain"] = "UG"

# --- Program (PG/UG specifics) ---

if "mca" in n:

meta["program"] = "MCA"

elif "mtech" in n or "m.tech" in n:

meta["program"] = "M.Tech"

elif "msc" in n or "m.sc" in n:

meta["program"] = "M.Sc"

elif "btech" in n or "b.tech" in n:

meta["program"] = "B.Tech"

# --- Hostel attributes ---

if n.startswith("lh") or "ladies" in n or "girls" in n or "women" in n:

meta["gender"] = "Female"

if n.startswith("mh") or "mens" in n or "men's" in n or "boys" in n:

meta["gender"] = "Male"

if "first-year" in n or "first year" in n or "freshers" in n or "1st-year" in n:

meta["student\_year"] = "First-Year"

if "senior" in n:

meta["student\_year"] = "Senior"

# --- Category ---

if "refund" in n:

meta["category"] = "Refund Policy"

meta["stable\_flag"] = "volatile"

elif "affidavit" in n:

meta["category"] = "Documents Required"

elif "document" in n or "submission" in n or "downloads" in n:

meta["category"] = "Documents Required"

elif "fee" in n and "hostel" in n:

meta["category"] = "Fee Structure"

meta["stable\_flag"] = "volatile"

elif "fee" in n:

meta["category"] = "Fee Structure"

meta["stable\_flag"] = "volatile"

elif "faq" in n:

meta["category"] = "FAQ"

elif "process" in n or "admissions process" in n or "procedure" in n:

meta["category"] = "Admission Process"

elif "programme" in n or "programmes" in n or "courses" in n or "offered" in n:

meta["category"] = "Courses Offered"

elif "eligibility" in n:

meta["category"] = "Eligibility Criteria"

elif "freshers-hostel-admission-information" in n or ("freshers" in n and "hostel" in n):

meta["category"] = "Hostel Norms"

# --- Currency / Audience Hints ---

if any(k in n for k in ["nri", "foreign", "international"]):

meta["currency"] = "USD"

# Default to INR where hostel/tuition but no USD cue

if meta["category"] == "Fee Structure" and not meta.get("currency"):

meta["currency"] = "INR"

# Exam patterns are usually volatile (VITREE/VITEEE/VITMEE)

if meta["domain"] == "Research" and meta.get("category") in (None, "Admission Process"):

# leave category for content-based detection later if needed

pass

return meta

# ------------------------------ Extractors --------------------------------- #

def extract\_pdf\_text(path: Path) -> str:

try:

with pdfplumber.open(str(path)) as pdf:

pages = []

for p in pdf.pages:

t = p.extract\_text() or ""

pages.append(t)

return "\n\n".join(pages)

except Exception as e:

print(f"[WARN] pdfplumber failed on {path.name}: {e}")

return ""

def extract\_docx\_text(path: Path) -> str:

if not HAS\_DOCX:

return ""

try:

d = docx.Document(str(path))

return "\n".join(p.text for p in d.paragraphs)

except Exception as e:

print(f"[WARN] python-docx failed on {path.name}: {e}")

return ""

# ----------------------------- Ingestion Core ------------------------------ #

def chunk\_text(text: str) -> List[str]:

splitter = RecursiveCharacterTextSplitter(

chunk\_size=450,

chunk\_overlap=60,

separators=["\n\n", "\n", ". ", " "],

length\_function=len,

)

return splitter.split\_text(text)

def sha1(s: str) -> str:

return hashlib.sha1(s.encode("utf-8", errors="ignore")).hexdigest()

def ingest\_dir(in\_dir: Path, out\_jsonl: Path, out\_catalog: Path) -> None:

out\_jsonl.parent.mkdir(parents=True, exist\_ok=True)

out\_catalog.parent.mkdir(parents=True, exist\_ok=True)

files: List[Path] = []

for ext in ("\*.pdf", "\*.PDF", "\*.docx", "\*.DOCX"):

files.extend(list(in\_dir.rglob(ext)))

if not files:

print(f"[ERROR] No PDFs/DOCX found in {in\_dir}.")

sys.exit(1)

# Catalog headers

catalog\_rows = [["source\_file", "size\_bytes", "modified\_utc", "pages\_or\_paras",

"domain", "category", "program", "gender", "student\_year",

"audience", "currency", "ay", "stable\_flag", "chunk\_count"]]

total\_chunks = 0

with out\_jsonl.open("w", encoding="utf-8") as fout:

for path in tqdm(files, desc="Ingesting"):

meta = map\_metadata(path.name)

# extract

if path.suffix.lower() == ".pdf":

raw = extract\_pdf\_text(path)

else:

raw = extract\_docx\_text(path)

cleaned = clean\_text(raw)

if not cleaned.strip():

print(f"[WARN] Empty after extract/clean: {path.name}")

continue

chunks = chunk\_text(cleaned)

mtime = datetime.datetime.utcfromtimestamp(path.stat().st\_mtime).strftime("%Y-%m-%d")

# write chunks

for i, ch in enumerate(chunks):

rec = {

"id": f"{path.stem}\_\_{i}",

"text": ch,

"metadata": {

\*\*meta,

"source\_file": path.name,

"source\_path": str(path),

"source\_title": path.stem,

"chunk\_id": i,

"last\_updated": mtime,

"content\_hash": sha1(ch),

},

}

fout.write(json.dumps(rec, ensure\_ascii=False) + "\n")

# catalog row

pages\_or\_paras = cleaned.count("\n\n") + 1

catalog\_rows.append([

path.name,

path.stat().st\_size,

mtime,

pages\_or\_paras,

meta.get("domain"),

meta.get("category"),

meta.get("program"),

meta.get("gender"),

meta.get("student\_year"),

meta.get("audience"),

meta.get("currency"),

meta.get("ay"),

meta.get("stable\_flag"),

len(chunks),

])

total\_chunks += len(chunks)

with out\_catalog.open("w", newline="", encoding="utf-8") as cf:

csv.writer(cf).writerows(catalog\_rows)

print(f"[DONE] Wrote {total\_chunks} chunks → {out\_jsonl}")

print(f"[DONE] Wrote catalog → {out\_catalog}")

# --------------------------------- CLI ------------------------------------- #

def main():

p = argparse.ArgumentParser(description="Ingest VIT PDFs/DOCX → cleaned chunks JSONL")

p.add\_argument("--in", dest="in\_dir", default="data/raw", help="Input folder with PDFs/DOCX")

p.add\_argument("--out", dest="out\_jsonl", default="data/processed/chunks.jsonl", help="Output JSONL path")

p.add\_argument("--catalog", dest="catalog", default="data/processed/catalog.csv", help="Catalog CSV path")

args = p.parse\_args()

in\_dir = Path(args.in\_dir)

out\_jsonl = Path(args.out\_jsonl)

out\_catalog = Path(args.catalog)

ingest\_dir(in\_dir, out\_jsonl, out\_catalog)

if \_\_name\_\_ == "\_\_main\_\_":

main()

ABOVE IS DATA INGESTION

#!/usr/bin/env python3

# -\*- coding: utf-8 -\*-

"""

One-pass ingest → clean → sanitize → chunk → dedup → FINAL JSONL

for the VIT Smart FAQ (Vellore MVP).

USAGE:

pip install "langchain<0.3" pdfplumber python-docx scikit-learn tqdm

python ingest\_clean\_finalize.py --in data/raw \

--out data/processed/chunks\_clean\_final.jsonl \

--catalog data/processed/clean\_catalog.csv \

--stats data/processed/clean\_stats.txt

"""

import os, re, csv, json, argparse, hashlib, datetime, sys, unicodedata

from pathlib import Path

from typing import Dict, Any, List, Optional, Tuple

from collections import defaultdict

import numpy as np

from tqdm import tqdm

# PDF & DOCX

import pdfplumber

try:

import docx # python-docx

HAS\_DOCX = True

except Exception:

HAS\_DOCX = False

# Chunking

from langchain.text\_splitter import RecursiveCharacterTextSplitter

# Dedup

from sklearn.feature\_extraction.text import TfidfVectorizer

from sklearn.metrics.pairwise import cosine\_similarity

# ------------------------------ Heuristics ---------------------------------- #

SECTION\_HINTS = {

"Fee Structure": [

"fee structure", "tuition", "fees", "charges", "payment", "category 1", "category 2",

"admission fee", "caution deposit", "mess", "special mess", "non-veg", "non veg"

],

"Eligibility Criteria": [

"eligibility", "minimum marks", "pcm", "pcb", "aggregate", "criteria", "age limit"

],

"Documents Required": [

"documents required", "affidavit", "undertaking", "certificate", "photograph",

"marksheet", "reporting", "upload", "submission"

],

"Courses Offered": [

"programmes offered", "programs offered", "branches", "specialization", "schools", "department"

],

"Academic Rules": [

"attendance", "grading", "grade point", "ffcs", "regulations", "cgpa"

],

"Hostel Norms": [

"hostel rules", "norms", "biometric", "in/out", "gate", "appliances", "cooking", "timings"

],

"Refund Policy": [

"refund", "withdrawal", "vacate", "deduction", "refund policy"

],

"Contacts": [

"contact", "email", "helpline", "phone", "reach out", "support"

],

"Admission Process": [

"process", "procedure", "application", "counselling", "slot booking", "steps"

],

"Exam Pattern": [

"exam pattern", "duration", "sections", "marks", "syllabus", "mcq"

],

"FAQ": [

"frequently asked questions", "faqs", "question", "answer"

],

}

AY\_PAT = re.compile(r'(20\d{2})\D{0,3}(\d{2})') # 2025-26, 2025–26, 2025\_26

EMAIL\_PAT = re.compile(r'[A-Za-z0-9.\_%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,}')

PHONE\_PAT = re.compile(r'(?:(?:\+?\d{1,3}[\s-]?)?\d{10,})')

PAGE\_PATTERNS = [

r"Page\s+\d+\s\*(?:of|/)\s\*\d+",

r"^\s\*Vellore\s+Institute\s+of\s+Technology.\*$",

]

PAGE\_REGEXES = [re.compile(p, re.IGNORECASE | re.MULTILINE) for p in PAGE\_PATTERNS]

# Replacement map for common Unicode punctuation/whitespace weirdness

REPLACEMENTS = {

"\u2018": "'", "\u2019": "'", "\u201A": "'", "\u201B": "'",

"\u201C": '"', "\u201D": '"', "\u201E": '"',

"\u2013": "-", "\u2014": "-", "\u2212": "-", # en/em dash, minus

"\u00A0": " ", # NBSP

"\uFFFD": "", # replacement char

}

# Bullets, arrows, dingbats (often from Office/PUA fonts)

ARROW\_BULLET\_CHARS = [

"•","●","◦","∙","·","▪","▫","◾","◽","■","□",

"➤","➔","➜","➝","➞","➟","➡","➠","→","⇒",

"►","▸","▹","▶","▷","❯","❱","❭",

"","","","","","","","","","","",

]

ARROW\_BULLET\_SET = set(ARROW\_BULLET\_CHARS)

# ------------------------------ Utilities ----------------------------------- #

def sha1(s: str) -> str:

return hashlib.sha1(s.encode("utf-8", errors="ignore")).hexdigest()

def guess\_ay\_from\_name(name: str) -> Optional[str]:

name = name.replace("\_", "-")

m = AY\_PAT.search(name)

if m:

return f"{m.group(1)}-{m.group(2)}"

return None

def sanitize\_text(s: str) -> str:

"""Unicode normalize + remove PUA/controls + normalize punctuation + map bullets/arrows."""

if not s:

return s

s = unicodedata.normalize("NFKC", s)

for k, v in REPLACEMENTS.items():

s = s.replace(k, v)

# remove (cid:###) artifacts

s = re.sub(r"\(cid:\d+\)", "", s)

# map bullets/arrows/checkmarks to "- "

s = "".join(("- " if ch in ARROW\_BULLET\_SET else ch) for ch in s)

# strip other control/private-use chars (keep \n and \t)

def \_ok(ch: str) -> bool:

cat = unicodedata.category(ch)

return not (cat and cat[0] == "C") or ch in ("\n", "\t")

s = "".join(ch for ch in s if \_ok(ch))

# collapse sequences like "-- - -" -> " - "

s = re.sub(r"(?:\s\*-\s\*){2,}", " - ", s)

# tidy whitespace/newlines

s = re.sub(r"[ \t]{2,}", " ", s)

s = re.sub(r"\s+\n", "\n", s)

s = re.sub(r"\n\s+", "\n", s)

s = re.sub(r"\n{3,}", "\n\n", s)

# limit repeated punctuation

s = re.sub(r'([.!?])\1{2,}', r'\1\1', s) # "!!!!!" -> "!!"

s = re.sub(r'(,){2,}', r'\1', s) # ",,,," -> ","

s = re.sub(r'(-){3,}', r'--', s) # "-----" -> "--"

# trim stray punctuation at edges

s = re.sub(r'^[\s\-\.,;:]+', '', s)

s = re.sub(r'[\s\-\.,;:]+$', '', s)

return s.strip()

def clean\_text\_basic(raw: str) -> str:

"""Remove obvious page artifacts, keep paragraphs but join single line-breaks."""

if not raw:

return ""

txt = raw.replace("\r\n", "\n").replace("\r", "\n")

for rgx in PAGE\_REGEXES:

txt = rgx.sub("", txt)

txt = re.sub(r"^\s\*\d+\s\*$", "", txt, flags=re.MULTILINE) # standalone page numbers

txt = re.sub(r"(\w)-\n(\w)", r"\1\2", txt) # hyphens across lines

txt = re.sub(r"(?<!\n)\n(?!\n)", " ", txt) # join single newlines

txt = re.sub(r"\n{3,}", "\n\n", txt) # keep paragraph breaks

txt = re.sub(r"[ \t]{2,}", " ", txt)

return txt.strip()

def extract\_pdf\_text(path: Path) -> str:

try:

with pdfplumber.open(str(path)) as pdf:

pages = [(p.extract\_text() or "") for p in pdf.pages]

return "\n\n".join(pages)

except Exception as e:

print(f"[WARN] pdfplumber failed on {path.name}: {e}")

return ""

def extract\_docx\_text(path: Path) -> str:

if not HAS\_DOCX:

return ""

try:

d = docx.Document(str(path))

return "\n".join(p.text for p in d.paragraphs)

except Exception as e:

print(f"[WARN] python-docx failed on {path.name}: {e}")

return ""

def map\_metadata\_from\_path(path: Path) -> Dict[str, Any]:

n = path.name.lower()

parts = [p.lower() for p in path.parts]

meta: Dict[str, Any] = {

"campus": "Vellore",

"domain": None, # UG | PG | Hostel | Research | ...

"category": None, # Fee Structure | Documents Required | ...

"program": None, # MCA | M.Tech | M.Sc | B.Tech | Ph.D.

"gender": None, # Male | Female (hostel)

"student\_year": None, # First-Year | Senior (hostel)

"currency": None, # INR | USD

"audience": None, # NRI | Foreign

"ay": guess\_ay\_from\_name(n),

"stable\_flag": "stable",

}

# domain

if any("hostel" in p for p in parts) or n.startswith(("mh", "lh")):

meta["domain"] = "Hostel"

elif any("research" in p for p in parts) or "vitree" in n or "phd" in n or "ph.d" in n:

meta["domain"] = "Research"

elif any("pg" == p or "postgrad" in p for p in parts) or any(k in n for k in ["mtech","m.tech","mca","msc","m.sc"]):

meta["domain"] = "PG"

elif any("ug" == p for p in parts) or "ug" in n:

meta["domain"] = "UG"

# audience

if "nri" in n and "ug" in n:

meta["audience"] = "NRI"

if "foreign" in n and "ug" in n:

meta["audience"] = "Foreign"

if "international" in n and "admissions" in n and "fee" in n:

meta["domain"] = meta["domain"] or "NRI-Intl"

meta["audience"] = "Foreign"

# program

if "mca" in n:

meta["program"] = "MCA"

elif "mtech" in n or "m.tech" in n:

meta["program"] = "M.Tech"

elif "msc" in n or "m.sc" in n:

meta["program"] = "M.Sc"

elif "btech" in n or "b.tech" in n:

meta["program"] = "B.Tech"

elif meta["domain"] == "Research":

meta["program"] = "Ph.D."

# hostel attrs

if n.startswith("lh") or "ladies" in n or "girls" in n or "women" in n:

meta["gender"] = "Female"

if n.startswith("mh") or "mens" in n or "men's" in n or "boys" in n:

meta["gender"] = "Male"

if any(k in n for k in ["first-year","first year","freshers","1st-year"]):

meta["student\_year"] = "First-Year"

if "senior" in n:

meta["student\_year"] = "Senior"

# category

if "refund" in n:

meta["category"] = "Refund Policy"; meta["stable\_flag"] = "volatile"

elif "affidavit" in n:

meta["category"] = "Documents Required"

elif any(k in n for k in ["document","submission","download"]):

meta["category"] = "Documents Required"

elif "fee" in n and "hostel" in n:

meta["category"] = "Fee Structure"; meta["stable\_flag"] = "volatile"

elif "fee" in n:

meta["category"] = "Fee Structure"; meta["stable\_flag"] = "volatile"

elif "faq" in n:

meta["category"] = "FAQ"

elif any(k in n for k in ["process","procedure","admissions process"]):

meta["category"] = "Admission Process"

elif any(k in n for k in ["programme","programmes","courses","offered"]):

meta["category"] = "Courses Offered"

elif "eligibility" in n:

meta["category"] = "Eligibility Criteria"

elif "freshers-hostel-admission-information" in n or ("freshers" in n and "hostel" in n):

meta["category"] = "Hostel Norms"

# currency

if any(k in n for k in ["nri","foreign","international"]):

meta["currency"] = "USD"

if meta["category"] == "Fee Structure" and not meta.get("currency"):

meta["currency"] = "INR"

return meta

def refine\_category\_from\_text(text: str, current: Optional[str]) -> Optional[str]:

t = text.lower()

scores = {cat: sum(kw in t for kw in keys) for cat, keys in SECTION\_HINTS.items()}

best = max(scores, key=scores.get)

return best if scores[best] >= 2 else current

def infer\_ay\_from\_text(text: str, current: Optional[str]) -> Optional[str]:

if current: return current

m = AY\_PAT.search(text)

return f"{m.group(1)}-{m.group(2)}" if m else current

def extract\_contacts(text: str) -> Tuple[List[str], List[str]]:

emails = sorted(set(EMAIL\_PAT.findall(text)))

phones = sorted(set(PHONE\_PAT.findall(text)))

return emails, phones

def chunk\_text(text: str) -> List[str]:

splitter = RecursiveCharacterTextSplitter(

chunk\_size=450, chunk\_overlap=60,

separators=["\n\n", "\n", ". ", " "],

length\_function=len,

)

return splitter.split\_text(text)

def merge\_small\_chunks(records: List[Dict[str, Any]], min\_len=280, max\_len=520) -> List[Dict[str, Any]]:

out = []; buf = None

for rec in records:

key = (rec["metadata"].get("source\_file"),

rec["metadata"].get("domain"),

rec["metadata"].get("category"),

rec["metadata"].get("program"))

if buf is None:

buf = rec; continue

buf\_key = (buf["metadata"].get("source\_file"),

buf["metadata"].get("domain"),

buf["metadata"].get("category"),

buf["metadata"].get("program"))

if len(buf["text"]) < min\_len and key == buf\_key:

merged\_text = (buf["text"].rstrip() + "\n\n" + rec["text"].lstrip()).strip()

if len(merged\_text) <= max\_len:

buf["text"] = merged\_text

continue

out.append(buf); buf = rec

if buf is not None:

out.append(buf)

return out

def near\_dedup(records: List[Dict[str, Any]], threshold=0.97) -> Tuple[List[Dict[str, Any]], int]:

nrecs = len(records)

if nrecs < 2:

return records, 0

texts = [r["text"] for r in records]

vec = TfidfVectorizer(max\_features=30000, ngram\_range=(1,2))

X = vec.fit\_transform(texts)

if X.shape[1] == 0:

return records, 0

keep = []; killed = set(); n = X.shape[0]

for i in range(n):

if i in killed: continue

keep.append(i)

if i + 1 >= n: continue

tail = X[i+1:]

if tail.shape[0] == 0: continue

sims = cosine\_similarity(X[i], tail).ravel()

for j\_off, sim in enumerate(sims, start=1):

if sim >= threshold:

killed.add(i + j\_off)

result = [records[idx] for idx in range(n) if idx in keep and idx not in killed]

return result, len(killed)

# ------------------------------ Main ---------------------------------------- #

def main():

ap = argparse.ArgumentParser(description="Ingest + sanitize + chunk + dedup to FINAL JSONL")

ap.add\_argument("--in", dest="in\_dir", default="data/raw", help="Input folder (recursive)")

ap.add\_argument("--out", dest="out\_jsonl", default="data/processed/chunks\_clean\_final.jsonl",

help="Output FINAL cleaned JSONL")

ap.add\_argument("--catalog", dest="catalog\_csv", default="data/processed/clean\_catalog.csv",

help="Output catalog CSV per-file")

ap.add\_argument("--stats", dest="stats\_path", default="", help="Optional stats .txt (glyphs before/after)")

args = ap.parse\_args()

in\_dir = Path(args.in\_dir).expanduser().resolve()

out\_jsonl = Path(args.out\_jsonl).expanduser().resolve()

catalog\_csv = Path(args.catalog\_csv).expanduser().resolve()

out\_jsonl.parent.mkdir(parents=True, exist\_ok=True)

catalog\_csv.parent.mkdir(parents=True, exist\_ok=True)

print(f"[INFO] Scanning input folder: {in\_dir}")

print(f"[INFO] Writing FINAL chunks : {out\_jsonl}")

print(f"[INFO] Writing catalog : {catalog\_csv}")

files: List[Path] = []

for ext in ("\*.pdf", "\*.PDF", "\*.docx", "\*.DOCX"):

files.extend(in\_dir.rglob(ext))

if not files:

print(f"[ERROR] No PDFs/DOCX found under: {in\_dir}"); sys.exit(1)

file\_reports = []

all\_cleaned: List[Dict[str, Any]] = []

# stats for glyphs before/after

def has\_weird(s: str) -> bool:

if not s: return False

return any(ch in s for ch in ARROW\_BULLET\_SET) or "(cid:" in s or "\uFFFD" in s

glyphs\_before = 0

glyphs\_after = 0

total\_chunks = 0

for path in tqdm(files, desc="Processing files"):

meta\_base = map\_metadata\_from\_path(path)

# Extract → basic clean → sanitize once globally

raw = extract\_pdf\_text(path) if path.suffix.lower() == ".pdf" else extract\_docx\_text(path)

cleaned\_basic = clean\_text\_basic(raw)

cleaned = sanitize\_text(cleaned\_basic)

if not cleaned.strip():

print(f"[WARN] Empty after extract/clean: {path.name}")

file\_reports.append([path.name, 0, 0, 0, "EMPTY"])

continue

# Chunk

chunks = chunk\_text(cleaned)

total\_chunks += len(chunks)

# Build per-chunk records

recs = []

mtime = datetime.datetime.utcfromtimestamp(path.stat().st\_mtime).strftime("%Y-%m-%d")

for i, ch in enumerate(chunks):

# extra per-chunk sanitize (idempotent)

if has\_weird(ch): glyphs\_before += 1

ch2 = sanitize\_text(ch)

if has\_weird(ch2): glyphs\_after += 1

recs.append({

"id": f"{path.stem}\_\_{i}",

"text": ch2,

"metadata": {

\*\*meta\_base,

"source\_file": path.name,

"source\_path": str(path),

"source\_title": path.stem,

"chunk\_id": i,

"last\_updated": mtime,

"content\_hash": sha1(ch2),

}

})

# Per-chunk enrich (category/AY/contacts)

refined\_cats = 0; inferred\_ays = 0; contacts\_enriched = 0

for r in recs:

new\_cat = refine\_category\_from\_text(r["text"], r["metadata"].get("category"))

if new\_cat != r["metadata"].get("category"):

r["metadata"]["category"] = new\_cat; refined\_cats += 1

new\_ay = infer\_ay\_from\_text(r["text"], r["metadata"].get("ay"))

if new\_ay and new\_ay != r["metadata"].get("ay"):

r["metadata"]["ay"] = new\_ay; inferred\_ays += 1

emails, phones = extract\_contacts(r["text"])

if emails or phones:

r["metadata"]["emails"] = emails

r["metadata"]["phones"] = phones

contacts\_enriched += 1

# Merge tiny chunks & dedup

recs.sort(key=lambda x: x["metadata"].get("chunk\_id", 0))

merged = merge\_small\_chunks(recs, min\_len=280, max\_len=520)

if len(merged) >= 2:

deduped, killed = near\_dedup(merged, threshold=0.97)

else:

deduped, killed = merged, 0

# Exact dedup inside file

seen = set(); unique = []

for r in deduped:

h2 = sha1(r["text"])

if h2 in seen: continue

seen.add(h2); r["metadata"]["content\_hash2"] = h2; unique.append(r)

all\_cleaned.extend(unique)

file\_reports.append([

path.name, len(chunks), len(merged), len(unique),

f"refined\_cat:{refined\_cats}|inferred\_ay:{inferred\_ays}|contacts:{contacts\_enriched}|near\_killed:{killed}"

])

# Global exact dedup

global\_seen = set(); final\_records: List[Dict[str, Any]] = []

for r in all\_cleaned:

h = sha1(r["text"])

if h in global\_seen: continue

global\_seen.add(h); final\_records.append(r)

# Write outputs

with out\_jsonl.open("w", encoding="utf-8") as f:

for r in final\_records:

f.write(json.dumps(r, ensure\_ascii=False) + "\n")

with catalog\_csv.open("w", newline="", encoding="utf-8") as cf:

w = csv.writer(cf)

w.writerow(["source\_file","chunks\_raw","chunks\_after\_merge","chunks\_final","notes"])

for row in file\_reports:

w.writerow(row)

# Optional stats

if args.stats\_path:

sp = Path(args.stats\_path)

sp.parent.mkdir(parents=True, exist\_ok=True)

with sp.open("w", encoding="utf-8") as sf:

sf.write(f"total\_input\_chunks: {total\_chunks}\n")

sf.write(f"glyphs\_before: {glyphs\_before}\n")

sf.write(f"glyphs\_after: {glyphs\_after}\n")

sf.write(f"final\_chunks: {len(final\_records)}\n")

print(f"[INFO] wrote stats → {sp}")

print(f"[DONE] Files processed : {len(files)}")

print(f"[DONE] Final chunks : {len(final\_records)} → {out\_jsonl}")

print(f"[DONE] Clean catalog : {catalog\_csv}")

print("[TIP ] Next: use this FINAL JSONL for embeddings & indexing.")

# --------- Optional: simple keyword→filter helper for your router ---------- #

DOMAIN\_MAP = {

"ug":"UG", "undergrad":"UG",

"pg":"PG", "postgrad":"PG", "post graduate":"PG", "post-graduate":"PG",

"hostel":"Hostel", "accommodation":"Hostel",

"research":"Research", "phd":"Research", "vitree":"Research",

}

PROGRAM\_MAP = {

"mca":"MCA", "m.tech":"M.Tech", "mtech":"M.Tech", "msc":"M.Sc", "m.sc":"M.Sc",

"btech":"B.Tech","b.tech":"B.Tech","phd":"Ph.D"

}

CATEGORY\_MAP = {

"refund policy":"Refund Policy", "refund":"Refund Policy",

"fee structure":"Fee Structure", "fees":"Fee Structure", "tuition":"Fee Structure",

"courses offered":"Courses Offered", "programs offered":"Courses Offered", "programmes offered":"Courses Offered",

"eligibility criteria":"Eligibility Criteria", "eligibility":"Eligibility Criteria",

"documents required":"Documents Required", "documents":"Documents Required", "document submission":"Documents Required",

"academic rules":"Academic Rules", "attendance":"Academic Rules", "grading":"Academic Rules", "ffcs":"Academic Rules",

"hostel norms":"Hostel Norms", "norms":"Hostel Norms",

"admission process":"Admission Process", "process":"Admission Process", "procedure":"Admission Process",

"exam pattern":"Exam Pattern", "faq":"FAQ", "faqs":"FAQ",

"contacts":"Contacts", "contact":"Contacts", "helpline":"Contacts"

}

AUDIENCE\_MAP = {

"nri":"NRI", "foreign":"Foreign", "international":"Foreign"

}

def build\_filters\_from\_query(q: str) -> Dict[str, Any]:

ql = q.lower()

where: Dict[str, Any] = {}

for k,v in DOMAIN\_MAP.items():

if k in ql: where["domain"] = v; break

for k,v in PROGRAM\_MAP.items():

if k in ql: where["program"] = v; break

for k,v in CATEGORY\_MAP.items():

if k in ql: where["category"] = v; break

for k,v in AUDIENCE\_MAP.items():

if k in ql: where["audience"] = v; break

where.setdefault("campus", "Vellore") # MVP default

return where

if \_\_name\_\_ == "\_\_main\_\_":

main()

ABOVE IS PREPROCESSED ONE

#!/usr/bin/env python3

# -\*- coding: utf-8 -\*-

"""

Build a FAISS index from chunks\_clean\_final.jsonl

Embeddings: Gemini (default) or Sentence-Transformers (local)

Usage:

python embeddings\_faiss\_gemini.py --chunks Data/processed/chunks\_clean\_final.jsonl ^

--persist Data/index/faiss ^

--collection vit\_faq\_vellore ^

--emb gemini

# Or local embeddings (no API):

python embeddings\_faiss\_gemini.py --chunks Data/processed/chunks\_clean\_final.jsonl ^

--persist Data/index/faiss ^

--collection vit\_faq\_vellore ^

--emb st

"""

import os, sys, json, argparse

from pathlib import Path

from tqdm import tqdm

# Only use the community import (no deprecation warning)

from langchain\_community.vectorstores import FAISS

try:

from dotenv import load\_dotenv

load\_dotenv()

except Exception:

pass

# ---------------- Embedding backends ---------------- #

class GeminiEmbeddings:

"""

Minimal adapter for Gemini embeddings.

Uses the CURRENT model: models/text-embedding-004

"""

def \_\_init\_\_(self, model: str = "models/text-embedding-004", api\_key\_env: str = "GEMINI\_API\_KEY"):

import google.generativeai as genai

api\_key = os.getenv(api\_key\_env)

if not api\_key:

print("[ERROR] GEMINI\_API\_KEY not set. Set it via environment or .env file.")

sys.exit(1)

genai.configure(api\_key=api\_key)

self.genai = genai

self.model = model

def embed\_documents(self, texts):

out = []

for t in texts:

# embed\_content returns {'embedding': [...]} for this model

r = self.genai.embed\_content(model=self.model, content=t)

vec = r.get("embedding")

if vec is None:

raise RuntimeError("Gemini embed\_content returned no 'embedding'. Check model & quota.")

out.append(vec)

return out

def embed\_query(self, q):

r = self.genai.embed\_content(model=self.model, content=q)

vec = r.get("embedding")

if vec is None:

raise RuntimeError("Gemini embed\_content returned no 'embedding' for query.")

return vec

class STEmbeddings:

"""Sentence-Transformers local embeddings (no API)."""

def \_\_init\_\_(self, model\_name: str = "all-MiniLM-L6-v2"):

try:

from sentence\_transformers import SentenceTransformer

except Exception as e:

print("[ERROR] sentence-transformers not installed. pip install sentence-transformers")

raise

self.model = SentenceTransformer(model\_name)

def embed\_documents(self, texts):

return self.model.encode(texts, normalize\_embeddings=True).tolist()

def embed\_query(self, q):

return self.model.encode([q], normalize\_embeddings=True)[0].tolist()

# ---------------- Utilities ---------------- #

def load\_chunks(jsonl\_path: Path):

texts, metas, ids = [], [], []

with jsonl\_path.open("r", encoding="utf-8") as f:

for line in f:

if not line.strip():

continue

obj = json.loads(line)

txt = (obj.get("text") or "").strip()

if not txt:

continue

texts.append(txt)

metas.append(obj.get("metadata", {}) or {})

ids.append(obj.get("id") or f"id\_{len(ids)}")

return texts, metas, ids

# ---------------- Build FAISS ---------------- #

def build\_faiss(chunks, persist\_dir: Path, collection: str, emb):

texts, metas, ids = chunks

if not texts:

print("[ERROR] No texts to index. Is your chunks file empty?")

sys.exit(1)

print(f"[INFO] Building FAISS index with {len(texts)} docs (this may take a minute)...")

# langchain\_community FAISS expects .embed\_documents/.embed\_query on embedding object

vectors = emb.embed\_documents(texts)

# Create store from precomputed embeddings to avoid re-embedding inside FAISS

# (this path avoids issues if your embedding object isn't a LangChain Embeddings subclass)

from langchain\_community.docstore.in\_memory import InMemoryDocstore

from langchain\_core.documents import Document

import faiss

import numpy as np

import uuid

dim = len(vectors[0])

index = faiss.IndexFlatIP(dim) # cosine similarity via normalized vectors (Gemini returns L2; we keep IP; normalize if needed)

# normalize embeddings for cosine similarity

vec\_np = np.array(vectors, dtype="float32")

norms = np.linalg.norm(vec\_np, axis=1, keepdims=True) + 1e-12

vec\_np = vec\_np / norms

index.add(vec\_np)

# Build documents & ID map

docs = {}

id\_map = {}

for i, (t, m) in enumerate(zip(texts, metas)):

did = ids[i] if i < len(ids) else str(uuid.uuid4())

docs[did] = Document(page\_content=t, metadata=m)

id\_map[i] = did

store = FAISS(

embedding\_function=emb, # kept for API compatibility; not used since we precomputed

index=index,

docstore=InMemoryDocstore(docs),

index\_to\_docstore\_id=id\_map,

)

persist\_dir.mkdir(parents=True, exist\_ok=True)

store.save\_local(str(persist\_dir), index\_name=collection)

print(f"[DONE] FAISS saved at: {persist\_dir}/{collection}.faiss (+ .pkl)")

# ---------------- Main ---------------- #

if \_\_name\_\_ == "\_\_main\_\_":

ap = argparse.ArgumentParser()

ap.add\_argument("--chunks", required=True, help="Path to Data/processed/chunks\_clean\_final.jsonl")

ap.add\_argument("--persist", required=True, help="Folder to save FAISS (e.g., Data/index/faiss)")

ap.add\_argument("--collection", default="vit\_faq\_vellore", help="Index name")

ap.add\_argument("--emb", choices=["gemini","st"], default="gemini", help="Embedding backend")

args = ap.parse\_args()

chunks\_path = Path(args.chunks).resolve()

if not chunks\_path.exists():

print(f"[ERROR] chunks not found: {chunks\_path}")

sys.exit(1)

if args.emb == "gemini":

emb = GeminiEmbeddings() # uses models/text-embedding-004

else:

emb = STEmbeddings()

texts, metas, ids = load\_chunks(chunks\_path)

build\_faiss((texts, metas, ids), Path(args.persist).resolve(), args.collection, emb)

ABOVE ONE IS EMBEDDINGS

# ETL/load\_sqlite.py

# Build SQLite DB from hostel staging CSVs + curated academics CSVs (NO VITREE/RULES)

# Output: Data/sql/vit\_vellore.db

import sqlite3, pathlib, csv, re, os, glob

from typing import Optional, Tuple, List

import pandas as pd

# -------------------- Paths --------------------

BASE = pathlib.Path("Data")

STAGING = BASE / "staging" # auto-staged CSVs from PDFs (hostel + misc)

PROCESSED= BASE / "processed" # curated CSVs (source of truth for academics)

SQLDIR = BASE / "sql"

SQLDIR.mkdir(parents=True, exist\_ok=True)

DB\_PATH = SQLDIR / "vit\_vellore.db"

MONEY = r"(?:₹|INR|USD|\$)\s\*[\d,]+(?:\.\d+)?"

# -------------------- DB utils --------------------

def \_mk\_conn():

con = sqlite3.connect(DB\_PATH)

con.execute("PRAGMA journal\_mode=WAL;")

con.execute("PRAGMA synchronous=NORMAL;")

con.row\_factory = sqlite3.Row

return con

def \_exec\_schema(con: sqlite3.Connection, ddl: str):

con.executescript(ddl)

def \_schema(con: sqlite3.Connection):

"""Create HOSTEL + ACADEMICS schema (no VITREE/RULES)."""

\_exec\_schema(con, """

-- ---------------- HOSTEL ----------------

CREATE TABLE IF NOT EXISTS blocks (

id INTEGER PRIMARY KEY AUTOINCREMENT,

block\_name TEXT,

display\_name TEXT,

gender TEXT, -- Male | Female

level TEXT, -- First-Year | Senior | NULL

block\_type TEXT

);

CREATE TABLE IF NOT EXISTS hostel\_fees (

id INTEGER PRIMARY KEY AUTOINCREMENT,

block\_id INTEGER NOT NULL,

ay TEXT, -- 2025-26

category TEXT, -- Indian | NRI | Foreign | CAT1 | CAT2 (if any)

occupancy TEXT, -- "2 Sharing" / "6 Sharing" / "2 Bed" etc

ac INTEGER, -- 1=AC, 0=Non-AC, NULL=unknown

mess\_type TEXT, -- Special / Non-Veg / Veg / ...

room\_mess\_fee TEXT,

admission\_fee TEXT,

caution\_deposit TEXT,

other\_fee TEXT,

total\_fee TEXT,

currency TEXT, -- INR | USD

source\_file TEXT

);

CREATE TABLE IF NOT EXISTS amenities (

id INTEGER PRIMARY KEY AUTOINCREMENT,

block\_id INTEGER NOT NULL,

key TEXT,

value TEXT

);

CREATE TABLE IF NOT EXISTS contacts (

id INTEGER PRIMARY KEY AUTOINCREMENT,

block\_id INTEGER NOT NULL,

name TEXT,

role TEXT,

phone TEXT,

email TEXT

);

-- lightweight extracted helper tables from Hostel\_info

CREATE TABLE IF NOT EXISTS mh\_blocks (

block\_code TEXT,

block\_name TEXT,

email TEXT,

landline TEXT

);

CREATE TABLE IF NOT EXISTS lh\_blocks (

block\_code TEXT,

landline TEXT

);

CREATE TABLE IF NOT EXISTS hostel\_contacts (

role TEXT,

name TEXT,

email TEXT,

phone TEXT

);

-- helpful hostel indexes

CREATE INDEX IF NOT EXISTS idx\_blocks\_name ON blocks(block\_name);

CREATE INDEX IF NOT EXISTS idx\_hostel\_fees ON hostel\_fees(block\_id, ay, category);

-- ---------------- ACADEMICS ----------------

CREATE TABLE IF NOT EXISTS programs (

id INTEGER PRIMARY KEY AUTOINCREMENT,

level TEXT, -- UG | PG | MCA | MSc

program TEXT, -- e.g., "B.Tech CSE"

school TEXT, -- optional

duration TEXT, -- e.g., "4 years"

campus TEXT, -- optional

source\_file TEXT

);

CREATE TABLE IF NOT EXISTS eligibility (

id INTEGER PRIMARY KEY AUTOINCREMENT,

level TEXT,

program TEXT, -- "ALL" for common/overall level criteria

criteria TEXT,

source\_file TEXT

);

CREATE TABLE IF NOT EXISTS documents\_required (

id INTEGER PRIMARY KEY AUTOINCREMENT,

level TEXT,

program TEXT, -- "ALL" for level-wide docs

item TEXT,

details TEXT,

source\_file TEXT

);

CREATE TABLE IF NOT EXISTS academic\_fees (

id INTEGER PRIMARY KEY AUTOINCREMENT,

level TEXT,

program TEXT, -- "ALL" when fees are not per-program

category TEXT, -- Indian | NRI | Foreign | CAT1 | CAT2 etc.

ay TEXT, -- 2025-26

tuition TEXT,

one\_time TEXT,

caution TEXT,

total TEXT,

currency TEXT, -- INR | USD

source\_file TEXT

);

CREATE TABLE IF NOT EXISTS scholarships (

id INTEGER PRIMARY KEY AUTOINCREMENT,

level TEXT, -- UG | PG | Research | etc.

name TEXT,

criteria TEXT,

amount TEXT,

currency TEXT,

source\_file TEXT

);

-- helpful academics indexes

CREATE INDEX IF NOT EXISTS idx\_prog ON programs(level, program);

CREATE INDEX IF NOT EXISTS idx\_elig ON eligibility(level, program);

CREATE INDEX IF NOT EXISTS idx\_docs ON documents\_required(level, program);

CREATE INDEX IF NOT EXISTS idx\_fees ON academic\_fees(level, program, category, ay);

CREATE INDEX IF NOT EXISTS idx\_sch ON scholarships(level, name);

""")

# -------------------- HOSTEL staging → SQL --------------------

def \_get\_or\_create\_block(con: sqlite3.Connection, block\_name: str, display: Optional[str],

gender: str, level: Optional[str], btype: Optional[str]) -> int:

row = con.execute(

"SELECT id FROM blocks WHERE block\_name=? AND IFNULL(level,'')=IFNULL(?, '') AND IFNULL(block\_type,'')=IFNULL(?, '') AND IFNULL(gender,'')=IFNULL(?, '')",

(block\_name, level, btype, gender)

).fetchone()

if row: return row[0]

con.execute("INSERT INTO blocks(block\_name, display\_name, gender, level, block\_type) VALUES(?,?,?,?,?)",

(block\_name, display or block\_name, gender, level, btype))

return con.execute("SELECT last\_insert\_rowid()").fetchone()[0]

def \_guess\_meta\_from\_filename(name: str) -> Tuple[str, Optional[str], str, str]:

"""Returns: gender, level, ay, category\_hint (hostel-only helper)"""

n = name.lower()

gender = "Male" if n.startswith("mh") else ("Female" if n.startswith("lh") else "")

level = "Senior" if "senior" in n else ("First-Year" if ("first-year" in n or "first year" in n) else None)

ay = ""

m = re.search(r"(20\d{2})\D{0,3}(\d{2})", n)

if m: ay = f"{m.group(1)}-{m.group(2)}"

if "nri" in n or "foreign" in n:

cat = "NRI"

elif "indian" in n:

cat = "Indian"

else:

cat = ""

return gender, level, ay, cat

HEADER\_ALIASES = {

"room\_type": ["roomtype", "room", "acnonac", "ac/nonac", "ac", "nonac", "category", "roomcategory"],

"mess\_type": ["messtype", "diet", "veg", "nonveg", "specialmess", "special"],

"room\_mess\_fee": ["roomandmessfee", "roommessfee", "hostelfee", "room&messfee", "roommess", "roomandmess", "hostelandmessfee"],

"admission\_fee": ["admissionfee", "admission"],

"caution\_deposit":["cautiondeposit", "refundabledeposit", "caution", "deposit"],

"other\_fee": ["other", "utility", "electricity", "maintenance", "service"],

"total\_fee": ["total", "grandtotal", "overalltotal", "nettotal", "totalamount"],

}

def \_norm(s: str) -> str:

return re.sub(r"[^a-z0-9]+","", (s or "").lower())

def \_score\_header\_row(cells: List[str]) -> int:

toks = [\_norm(c) for c in cells]

score = 0

for k in "room mess total admission caution deposit fee ac non".split():

if any(k in t for t in toks): score += 1

return score

def \_find\_header\_idx(rows: List[List[str]]) -> int:

best\_i, best\_s = 0, -1

for i in range(min(6, len(rows))):

s = \_score\_header\_row(rows[i])

if s > best\_s:

best\_s, best\_i = s, i

return best\_i

def \_map\_columns(header\_cells: List[str]) -> dict:

hnorm = [\_norm(c) for c in header\_cells]

mapping = {k: None for k in HEADER\_ALIASES.keys()}

for k, aliases in HEADER\_ALIASES.items():

for idx, h in enumerate(hnorm):

if any(a in h for a in aliases):

if k == "total\_fee":

mapping[k] = idx

elif mapping[k] is None:

mapping[k] = idx

return mapping

def \_detect\_occupancy(text: str) -> Optional[str]:

m = re.search(r"\b(\d+)\s\*(?:/|\s)?\s\*(\d+)?\s\*(sharing|seater|bed|occupancy)\b", text, flags=re.I) \

or re.search(r"\b(\d+)\s\*(sharing|seater|bed|occupancy)\b", text, flags=re.I)

if not m: return None

g = [x for x in m.groups() if x and x.isdigit()]

word = (m.groups()[-1] or "").title()

return f"{g[0]}/{g[1]} {word}" if len(g) == 2 else f"{g[0]} {word}"

def \_detect\_ac(text: str) -> Optional[int]:

if re.search(r"\bNon[- ]?AC\b", text, flags=re.I): return 0

if re.search(r"\bAC\b|\bA/C\b", text, flags=re.I): return 1

return None

def \_clean\_amt(s: str) -> str:

s = (s or "").strip()

return re.sub(r"[^\d₹$,\.INRUSD ]", "", s) if s else ""

def \_row\_has\_any\_value(\*vals) -> bool:

return any((v or "").strip() for v in vals)

def \_currency\_from(filename: str, row\_text: str, header\_text: str) -> str:

if "usd" in row\_text.lower() or "$" in row\_text or "usd" in header\_text.lower():

return "USD"

if "inr" in row\_text.lower() or "₹" in row\_text or "inr" in header\_text.lower():

return "INR"

fn = filename.lower()

if ("nri" in fn) or ("foreign" in fn): return "USD"

if ("indian" in fn): return "INR"

return ""

def load\_staging\_csvs\_hostel(con: sqlite3.Connection):

files = sorted(STAGING.glob("\*.csv"))

for path in files:

stem = path.stem.lower()

# only fee-structure CSVs (leave other CSVs to the academics loader)

if not (stem.startswith("mh-") or stem.startswith("lh-") or "hostel" in stem):

continue

gender, level, ay, cat\_hint = \_guess\_meta\_from\_filename(path.stem)

block\_title = ("Men Hostel" if gender=="Male" else "Ladies Hostel") + (f" {level}" if level else "")

block\_id = \_get\_or\_create\_block(con, block\_title, block\_title, gender or "", level, None)

rows = list(csv.reader(path.open("r", encoding="utf-8")))

if not rows:

continue

hi = \_find\_header\_idx(rows)

header = rows[hi]

colmap = \_map\_columns(header)

header\_text = " ".join(header)

for r in rows[hi+1:]:

if not any((c or "").strip() for c in r):

continue

room\_col = colmap.get("room\_type"); mess\_col = colmap.get("mess\_type")

room\_txt = (r[room\_col] if (room\_col is not None and room\_col < len(r)) else "") or ""

mess\_txt = (r[mess\_col] if (mess\_col is not None and mess\_col < len(r)) else "") or ""

joined = " ".join([c for c in r if c])

occ = \_detect\_occupancy(room\_txt) or \_detect\_occupancy(joined) or ""

ac = \_detect\_ac(room\_txt) or \_detect\_ac(joined)

mess = mess\_txt.strip()

if re.fullmatch(MONEY, mess): mess = ""

if re.search(r"\bspecial\b", mess, re.I): mess = "Special Mess"

elif re.search(r"non\s\*veg", mess, re.I): mess = "Non Veg"

elif re.search(r"\bveg\b", mess, re.I): mess = "Veg"

def pick\_amount(key: str) -> str:

idx = colmap.get(key); return \_clean\_amt(r[idx]) if (idx is not None and idx < len(r)) else ""

room\_mess = pick\_amount("room\_mess\_fee")

admission = pick\_amount("admission\_fee")

caution = pick\_amount("caution\_deposit")

other = pick\_amount("other\_fee")

total = pick\_amount("total\_fee")

row\_text = " ".join(r)

currency = \_currency\_from(path.name, row\_text, header\_text)

category = cat\_hint or ("NRI" if currency=="USD" else ("Indian" if currency=="INR" else ""))

if not \_row\_has\_any\_value(room\_mess, admission, caution, other, total):

m\_all = re.findall(MONEY, row\_text)

if m\_all:

room\_mess = m\_all[0]

if len(m\_all) > 1:

total = m\_all[-1]

con.execute("""

INSERT INTO hostel\_fees(block\_id, ay, category, occupancy, ac, mess\_type,

room\_mess\_fee, admission\_fee, caution\_deposit, other\_fee, total\_fee, currency, source\_file)

VALUES(?,?,?,?,?,?,?,?,?,?,?,?,?)

""", (block\_id, ay, category, occ, ac, mess, room\_mess, admission, caution, other, total, currency, path.name))

print("[OK] Loaded hostel fee CSVs into SQLITE.")

# ---------- parse Hostel\_info into mh\_blocks/lh\_blocks/hostel\_contacts ----------

def \_flatten\_hostel\_info\_frames() -> str:

paths = sorted(glob.glob(os.path.join(str(STAGING), "Hostel\_info\_\_camelot\_\*.csv")))

if not paths:

return ""

frames = []

for p in paths:

df = pd.read\_csv(p, header=None, dtype=str).fillna("")

frames.append(df)

raw = pd.concat(frames, ignore\_index=True)

raw["joined"] = raw.apply(lambda r: " ".join([c for c in r if isinstance(c, str)]), axis=1)

text = "\n".join(raw["joined"].tolist())

text = re.sub(r"\s+", " ", text)

text = text.replace("vit.ac.i n", "vit.ac.in")

return text

def load\_hostel\_info(con: sqlite3.Connection):

text = \_flatten\_hostel\_info\_frames()

if not text:

print("[WARN] Hostel\_info\_\_camelot\_\*.csv not found in staging; skipping mh/lh blocks + contacts.")

return

pat\_mh = re.compile(

r"(MH [A-Z](?: ANNEX)?)\s+([A-Z0-9 .’'--]+?(?:ANNEX)?)(?:\s+-\s\*[A-Z ]+)?\s+0416\s\*220\s\*(\d{4})\s+([A-Za-z.]+@vit\.ac\.in)",

flags=re.I,

)

mh = pd.DataFrame(pat\_mh.findall(text), columns=["block\_code","block\_name","last4","email"])

if not mh.empty:

mh["landline"] = "0416 220 " + mh["last4"]

mh.drop(columns=["last4"], inplace=True)

mh["block\_code"] = mh["block\_code"].str.upper().str.strip()

mh["block\_name"] = (mh["block\_name"].str.replace("–", "-", regex=False).str.title().str.strip())

pat\_lh = re.compile(r"(LH [A-Z]|RGT H|LH GH \(Annex\))\s+0416\s\*220\s\*(\d{4})", flags=re.I)

lh = pd.DataFrame(pat\_lh.findall(text), columns=["block\_code","last4"])

if not lh.empty:

lh["landline"] = "0416 220 " + lh["last4"]

lh.drop(columns=["last4"], inplace=True)

lh["block\_code"] = lh["block\_code"].str.upper().str.strip()

rows = [

("Section Supervisor (MH)", "Mr. Arasu R", "rarasu@vit.ac.in", "0416-220-2523"),

("Section Supervisor (LH)", "Ms. G. Subbulakshmi", "gsubbulakshmi@vit.ac.in", "0416-220-2711"),

("Residential Block Supervisor (LH, Transport)", "Ms. Mythily A", "mythily.a@vit.ac.in", "9488839864, 9791297375"),

]

contacts = pd.DataFrame(rows, columns=["role","name","email","phone"])

cur = con.cursor()

cur.execute("DELETE FROM mh\_blocks")

cur.execute("DELETE FROM lh\_blocks")

cur.execute("DELETE FROM hostel\_contacts")

if not mh.empty:

mh.to\_sql("mh\_blocks", con, index=False, if\_exists="append")

if not lh.empty:

lh.to\_sql("lh\_blocks", con, index=False, if\_exists="append")

contacts.to\_sql("hostel\_contacts", con, index=False, if\_exists="append")

print("[OK] Loaded Hostel Info (blocks + contacts).")

# -------------------- ACADEMICS: load from Data/processed/\*\* --------------------

def \_append\_df(con, table, df):

df = df.where(pd.notna(df), "")

df.to\_sql(table, con, if\_exists="append", index=False)

def \_clear\_academics(con):

for t in ["programs","eligibility","documents\_required","academic\_fees","scholarships"]:

con.execute(f"DELETE FROM {t}")

def \_load\_programs\_generic(con, path: pathlib.Path, level\_hint=None):

df = pd.read\_csv(path, dtype=str).fillna("")

df.columns = [c.strip().lower() for c in df.columns]

for c in ["level","program","school","duration","campus","source\_file"]:

if c not in df.columns: df[c] = ""

if level\_hint and (df["level"] == "").all():

df["level"] = level\_hint

\_append\_df(con, "programs", df[["level","program","school","duration","campus","source\_file"]])

def \_load\_documents\_generic(con, path: pathlib.Path, level\_hint=None):

df = pd.read\_csv(path, dtype=str).fillna("")

df.columns = [c.strip().lower() for c in df.columns]

rename = {"document":"item","doc":"item","name":"item","description":"details","detail":"details"}

df = df.rename(columns={k:v for k,v in rename.items() if k in df.columns})

for c in ["level","program","item","details","source\_file"]:

if c not in df.columns: df[c] = ""

if level\_hint and (df["level"] == "").all(): df["level"] = level\_hint

\_append\_df(con, "documents\_required", df[["level","program","item","details","source\_file"]])

def \_load\_eligibility\_generic(con, path: pathlib.Path, level\_hint=None):

df = pd.read\_csv(path, dtype=str).fillna("")

df.columns = [c.strip().lower() for c in df.columns]

for c in ["level","program","criteria","source\_file"]:

if c not in df.columns: df[c] = ""

if level\_hint and (df["level"] == "").all(): df["level"] = level\_hint

\_append\_df(con, "eligibility", df[["level","program","criteria","source\_file"]])

def \_load\_scholarships\_generic(con, path: pathlib.Path, level\_hint=None):

df = pd.read\_csv(path, dtype=str).fillna("")

df.columns = [c.strip().lower() for c in df.columns]

for c in ["level","name","criteria","amount","currency","source\_file"]:

if c not in df.columns: df[c] = ""

if level\_hint and (df["level"] == "").all(): df["level"] = level\_hint

\_append\_df(con, "scholarships", df[["level","name","criteria","amount","currency","source\_file"]])

def \_load\_ug\_fees(con, path: pathlib.Path):

df = pd.read\_csv(path, dtype=str).fillna("")

df.columns = [c.strip().lower().replace(" ","\_") for c in df.columns]

rename = {

"tuition\_fee":"tuition","tuition\_fees":"tuition",

"one\_time\_fee":"one\_time","admission\_fee":"one\_time","enrolment\_fee":"one\_time","registration\_fee":"one\_time",

"caution\_deposit":"caution","caution\_deposit\_inr":"caution",

"total\_fee":"total","total\_inr":"total"

}

for k,v in rename.items():

if k in df.columns and v not in df.columns:

df = df.rename(columns={k:v})

for c in ["level","program","category","ay","tuition","one\_time","caution","total","currency","source\_file"]:

if c not in df.columns: df[c] = ""

# infer AY if empty

if (df["ay"] == "").all():

df["ay"] = "2025-26"

\_append\_df(con, "academic\_fees",

df[["level","program","category","ay","tuition","one\_time","caution","total","currency","source\_file"]])

def \_load\_pg\_fees(con, path: pathlib.Path):

"""Reshape pg\_fees.csv (cat1/cat2 columns) into academic\_fees rows."""

df = pd.read\_csv(path, dtype=str).fillna("")

df.columns = [c.strip().lower() for c in df.columns]

rows = []

for \_, r in df.iterrows():

rows.append(dict(level="PG", program=r.get("program",""), category="CAT1", ay="2025-26",

tuition=r.get("cat1\_tuition\_inr",""), one\_time="",

caution=r.get("caution\_deposit\_inr",""), total=r.get("total\_cat1\_inr",""),

currency="INR", source\_file="pg\_fees.csv"))

rows.append(dict(level="PG", program=r.get("program",""), category="CAT2", ay="2025-26",

tuition=r.get("cat2\_tuition\_inr",""), one\_time="",

caution=r.get("caution\_deposit\_inr",""), total=r.get("total\_cat2\_inr",""),

currency="INR", source\_file="pg\_fees.csv"))

if rows:

out = pd.DataFrame(rows)

\_append\_df(con, "academic\_fees",

out[["level","program","category","ay","tuition","one\_time","caution","total","currency","source\_file"]])

def load\_academics\_from\_processed(con: sqlite3.Connection):

# UG

ug = PROCESSED / "UG"

if ug.exists():

if (ug/"programs\_ug (1).csv").exists(): \_load\_programs\_generic(con, ug/"programs\_ug (1).csv", "UG")

if (ug/"fee\_structure\_2025\_2026.csv").exists(): \_load\_ug\_fees(con, ug/"fee\_structure\_2025\_2026.csv")

if (ug/"eligibility\_2025\_2026.csv").exists(): \_load\_eligibility\_generic(con, ug/"eligibility\_2025\_2026.csv", "UG")

if (ug/"ug\_documents.csv").exists(): \_load\_documents\_generic(con, ug/"ug\_documents.csv", "UG")

if (ug/"scholarships\_ug.csv").exists(): \_load\_scholarships\_generic(con, ug/"scholarships\_ug.csv", "UG")

# PG

pg = PROCESSED / "PG"

if pg.exists():

if (pg/"pg\_programs.csv").exists(): \_load\_programs\_generic(con, pg/"pg\_programs.csv", "PG")

if (pg/"pg\_fees.csv").exists(): \_load\_pg\_fees(con, pg/"pg\_fees.csv")

if (pg/"pg\_eligibility.csv").exists(): \_load\_eligibility\_generic(con, pg/"pg\_eligibility.csv", "PG")

if (pg/"pg\_documents.csv").exists(): \_load\_documents\_generic(con, pg/"pg\_documents.csv", "PG")

# MCA

mca = PROCESSED / "MCA"

if mca.exists():

if (mca/"mca\_programs.csv").exists(): \_load\_programs\_generic(con, mca/"mca\_programs.csv", "MCA")

if (mca/"mca\_fees.csv").exists(): \_load\_ug\_fees(con, mca/"mca\_fees.csv") # same schema keys

if (mca/"mca\_eligibility (1).csv").exists(): \_load\_eligibility\_generic(con, mca/"mca\_eligibility (1).csv", "MCA")

if (mca/"mac\_documents.csv").exists(): \_load\_documents\_generic(con, mca/"mac\_documents.csv", "MCA")

# MSc

msc = PROCESSED / "MSC"

if msc.exists():

if (msc/"msc\_programs (1).csv").exists(): \_load\_programs\_generic(con, msc/"msc\_programs (1).csv", "MSc")

if (msc/"msc\_fees (1).csv").exists(): \_load\_ug\_fees(con, msc/"msc\_fees (1).csv")

if (msc/"msc\_eligibility (1).csv").exists(): \_load\_eligibility\_generic(con, msc/"msc\_eligibility (1).csv", "MSc")

if (msc/"msc\_documents.csv").exists(): \_load\_documents\_generic(con, msc/"msc\_documents.csv", "MSc")

# -------------------- MAIN --------------------

def main():

con = \_mk\_conn()

\_schema(con) # 1) create ALL tables (hostel + academics)

load\_staging\_csvs\_hostel(con) # 2) hostel fee CSVs → hostel\_fees

load\_hostel\_info(con) # 3) mh\_blocks / lh\_blocks / hostel\_contacts

# 4) academics from curated CSVs

\_clear\_academics(con)

load\_academics\_from\_processed(con)

con.commit()

con.close()

print(f"[DONE] SQLite DB ready at: {DB\_PATH}")

if \_\_name\_\_ == "\_\_main\_\_":

main()

ABOVE IS LOAD SQLITE

# Stage HOSTEL PDFs → CSVs (Data/staging) for eyeballing before SQLite load.

import pathlib, csv

IN = pathlib.Path("Data/Raw/HOSTEL")

OUT = pathlib.Path("Data/staging"); OUT.mkdir(parents=True, exist\_ok=True)

PDFS = [

"MH-Senior-FEE-structure-Indian-NRI-FOREIGN-Category-2025-26.pdf",

"MH-First-Year-FEE-structure-Indian-NRI-FOREIGN-Category-2025-26.pdf",

"LH-FEE-structure-Indian-Category-2025-26.pdf",

"LH-FEE-structure-NRI-Foreign-Category-2025-26.pdf",

"Hostel\_info.pdf",

"Hostel-joint-Affidavit-2025.pdf",

]

def try\_camelot(pdf\_path: pathlib.Path):

try:

import camelot

except Exception:

return []

tables = []

try:

t = camelot.read\_pdf(str(pdf\_path), pages='all', flavor='lattice')

for i, table in enumerate(t):

tables.append(("camelot", i, table.df))

except Exception:

pass

return tables

def try\_pdfplumber(pdf\_path: pathlib.Path):

import pdfplumber

tables = []

try:

with pdfplumber.open(str(pdf\_path)) as pdf:

for pi, page in enumerate(pdf.pages):

try:

for ti, rows in enumerate(page.extract\_tables() or []):

tables.append(("pdfplumber", f"{pi}\_{ti}", rows))

except Exception:

pass

except Exception:

pass

return tables

def normalize\_df(df):

rows = [list(df.columns)]

for \_, r in df.iterrows(): rows.append([str(x) for x in r.tolist()])

return rows

def write\_csv(rows, out\_csv: pathlib.Path):

with out\_csv.open("w", newline="", encoding="utf-8") as f:

w = csv.writer(f); [w.writerow([c.strip() if isinstance(c, str) else c for c in r]) for r in rows]

def main():

for pdf\_name in PDFS:

pdf\_path = IN / pdf\_name

if not pdf\_path.exists():

print(f"[WARN] Missing {pdf\_path}"); continue

staged = False

for src, tag, obj in (try\_camelot(pdf\_path) or []):

out\_csv = OUT / f"{pdf\_path.stem}\_\_{src}\_{tag}.csv"

write\_csv(normalize\_df(obj), out\_csv); print(f"[STAGED] {out\_csv}"); staged = True

if not staged:

for src, tag, rows in (try\_pdfplumber(pdf\_path) or []):

out\_csv = OUT / f"{pdf\_path.stem}\_\_{src}\_{tag}.csv"

write\_csv(rows, out\_csv); print(f"[STAGED] {out\_csv}"); staged = True

if not staged: print(f"[FAIL] No tables detected in {pdf\_path}")

if \_\_name\_\_ == "\_\_main\_\_":

main()

ABOVE IS STAGE\_HOSTEL\_TABLES

# app/handlers/academics.py — SQL-first answers for programs/eligibility/documents/academic\_fees/scholarships

import re, sqlite3

from typing import Dict, Any, List, Optional, Tuple

from app.utils.aliases import norm, level\_alias, category\_alias

from app.utils.render import md\_table, bullets, join\_sources

ACA\_COLUMNS\_PROGRAMS = ["Program", "School", "Duration", "Campus", "Source"]

ACA\_COLUMNS\_ELIG = ["Level/Program", "Criteria", "Source"]

ACA\_COLUMNS\_DOCS = ["Level/Program", "Item", "Details", "Source"]

ACA\_COLUMNS\_FEES = ["Level/Program", "AY", "Category", "Tuition", "One-time", "Caution", "Total", "Curr", "Source"]

ACA\_COLUMNS\_SCH = ["Level", "Name", "Criteria", "Amount", "Curr", "Source"]

def is\_academic\_like(q: str) -> bool:

ql = q.lower()

keys = [

"ug","undergrad","pg","mca","msc","programme","program","course",

"eligibility","criteria","documents","doc to submit","submit docs","admission",

"academic fee","tuition","semester fee","scholarship","stipend"

]

return any(k in ql for k in keys)

def detect\_academics\_intent(q: str) -> Dict[str, Any]:

ql = q.lower()

level = None

if any(w in ql for w in ["ug","undergrad","b.tech","btech","bsc","bca","bba"]):

level = "UG"

elif any(w in ql for w in ["pg","m.tech","mtech","msc","m.sc","mca","mba"]):

level = "PG"

# Program name extraction is open-ended; rely on SQL LIKE later

want\_programs = any(w in ql for w in ["program","programme","courses","offer","available"])

want\_elig = "eligib" in ql or "criteria" in ql

want\_docs = "document" in ql or "submit" in ql or "submission" in ql

want\_fees = "fee" in ql or "tuition" in ql

want\_schol = "scholar" in ql

ay = None

m = re.search(r"\b(20\d{2})\s\*[-/–]\s\*(\d{2})\b", ql)

if m:

ay = f"{m.group(1)}-{m.group(2)}"

cat = None

if "nri" in ql: cat = "NRI"

elif "foreign" in ql: cat = "Foreign"

elif "indian" in ql: cat = "Indian"

return {

"level": level,

"program\_like": None, # set later if we capture any quoted string

"want\_programs": want\_programs,

"want\_elig": want\_elig,

"want\_docs": want\_docs,

"want\_fees": want\_fees,

"want\_schol": want\_schol,

"ay": ay,

"category": cat

}

def \_maybe\_extract\_program\_like(q: str) -> Optional[str]:

# Try picking quoted fragments as program-like hints

m = re.search(r"['\"]([^\"']{2,80})['\"]", q)

if m:

return m.group(1).strip()

# small heuristic: words after 'for ' or 'in '

m = re.search(r"\b(for|in)\s+([A-Za-z0-9 &/\-]{2,80})$", q, re.I)

if m:

return m.group(2).strip()

return None

def \_sql\_like\_prog(cond: List[str], args: List[Any], prog\_like: Optional[str]):

if prog\_like:

cond.append("(LOWER(program) LIKE ?)")

args.append(f"%{prog\_like.lower()}%")

def \_src(row):

return row["source\_file"] or ""

def answer\_academics\_sql(con: sqlite3.Connection, q: str) -> Optional[str]:

if not con: return None

intent = detect\_academics\_intent(q)

if not any([intent["want\_programs"], intent["want\_elig"], intent["want\_docs"], intent["want\_fees"], intent["want\_schol"]]):

return None

level = level\_alias(intent["level"]) if intent["level"] else None

prog\_like = \_maybe\_extract\_program\_like(q)

# PROGRAMS

if intent["want\_programs"]:

sql = "SELECT level, program, school, duration, campus, source\_file FROM programs WHERE 1=1"

cond, args = [], []

if level:

cond.append("LOWER(level)=LOWER(?)"); args.append(level)

\_sql\_like\_prog(cond, args, prog\_like)

rows = con.execute(sql + (" AND " + " AND ".join(cond) if cond else "") + " ORDER BY program", args).fetchall()

if rows:

tbl = {

"columns": ACA\_COLUMNS\_PROGRAMS,

"rows": [[r["program"] or "", r["school"] or "", r["duration"] or "", r["campus"] or "", \_src(r)] for r in rows]

}

return md\_table(f"Programs Offered ({level or 'All'})", tbl["columns"], tbl["rows"])

# fall-through to other branches if nothing

# ELIGIBILITY

if intent["want\_elig"]:

sql = "SELECT level, program, criteria, source\_file FROM eligibility WHERE 1=1"

cond, args = [], []

if level: cond.append("LOWER(level)=LOWER(?)"); args.append(level)

\_sql\_like\_prog(cond, args, prog\_like)

rows = con.execute(sql + (" AND " + " AND ".join(cond) if cond else ""), args).fetchall()

if rows:

tbl = {

"columns": ACA\_COLUMNS\_ELIG,

"rows": [[(r["level"] or "") + (" / " + (r["program"] or "")), r["criteria"] or "", \_src(r)] for r in rows]

}

return md\_table("Eligibility", tbl["columns"], tbl["rows"])

# DOCUMENTS

if intent["want\_docs"]:

sql = "SELECT level, program, item, details, source\_file FROM documents\_required WHERE 1=1"

cond, args = [], []

if level: cond.append("LOWER(level)=LOWER(?)"); args.append(level)

\_sql\_like\_prog(cond, args, prog\_like)

rows = con.execute(sql + (" AND " + " AND ".join(cond) if cond else "") + " ORDER BY level, program, item", args).fetchall()

if rows:

tbl = {"columns": ACA\_COLUMNS\_DOCS,

"rows": [[(r["level"] or "") + (" / " + (r["program"] or "")), r["item"] or "", r["details"] or "", \_src(r)]

for r in rows]}

return md\_table("Documents to Submit", tbl["columns"], tbl["rows"])

# ACADEMIC FEES

if intent["want\_fees"]:

sql = """SELECT level, program, ay, category, tuition, one\_time, caution, total, currency, source\_file

FROM academic\_fees WHERE 1=1"""

cond, args = [], []

if level: cond.append("LOWER(level)=LOWER(?)"); args.append(level)

if intent["ay"]: cond.append("ay=?"); args.append(intent["ay"])

if intent["category"]:

cond.append("LOWER(category)=LOWER(?)"); args.append(category\_alias(intent["category"]))

\_sql\_like\_prog(cond, args, prog\_like)

order = " ORDER BY level, program, ay, category"

rows = con.execute(sql + (" AND " + " AND ".join(cond) if cond else "") + order, args).fetchall()

if rows:

tbl = {"columns": ACA\_COLUMNS\_FEES,

"rows": [[(r["level"] or "") + (" / " + (r["program"] or "")), r["ay"] or "", r["category"] or "",

r["tuition"] or "", r["one\_time"] or "", r["caution"] or "", r["total"] or "",

r["currency"] or "", \_src(r)] for r in rows]}

return md\_table("Academic Fee Details", tbl["columns"], tbl["rows"])

# SCHOLARSHIPS

if intent["want\_schol"]:

sql = "SELECT level, name, criteria, amount, currency, source\_file FROM scholarships WHERE 1=1"

cond, args = [], []

if level: cond.append("LOWER(level)=LOWER(?)"); args.append(level)

rows = con.execute(sql + (" AND " + " AND ".join(cond) if cond else "") + " ORDER BY level, name", args).fetchall()

if rows:

tbl = {"columns": ACA\_COLUMNS\_SCH,

"rows": [[r["level"] or "", r["name"] or "", r["criteria"] or "", r["amount"] or "", r["currency"] or "", \_src(r)]

for r in rows]}

return md\_table("Scholarships", tbl["columns"], tbl["rows"])

return None

ABOVE ONE IS ACADEMICS app/handlers - section

# app/handlers/hostels.py — hostel routing (contacts/blocks/landlines/fees) with your existing vibe

import re, sqlite3

from typing import Dict, Any, List, Optional, Tuple

from app.utils.aliases import norm

from app.utils.render import md\_table, bullets

def detect\_hostel\_intent(q: str) -> Dict[str, Any]:

qq = q.lower()

hostelish = any(w in qq for w in [

"hostel","block","mh","lh","boys","girls","mens","ladies",

"warden","supervisor","director","manager","landline","mess","laundry","fee","fees"

])

g = None

if any(w in qq for w in ["men", "boys", "mh", "mens"]): g = "MH"

if any(w in qq for w in ["ladies", "girls", "women", "lh"]): g = "LH" if g is None else g

wants\_landlines = hostelish and any(w in qq for w in ["landline", "phone", "contact number", "call"])

role\_hit = any(w in qq for w in ["director","chief warden","associate chief","warden","manager","assistant manager","supervisor"])

wants\_contacts = hostelish and role\_hit or role\_hit

wants\_blocks = hostelish and "block" in qq and any(w in qq for w in ["name","names","code","codes","list","all"])

wants\_fees = hostelish and ("fee" in qq or "mess" in qq or "room" in qq)

specific\_block = None

m = re.search(r"\b(MH|LH)\s\*-?\s\*([A-Z]{1,2})(?:\s\*ANNEX)?\b", q, re.I)

if m:

specific\_block = (m.group(1).upper() + " " + m.group(2).upper()).strip()

if "annex" in qq: specific\_block += " ANNEX"

reverse\_phone = None

m = re.search(r"(\+?\d[\d\s\-]{7,})", q)

if m: reverse\_phone = m.group(1)

reverse\_mail = None

m = re.search(r"([A-Za-z0-9.\_%+\-]+@[A-Za-z0-9.\-]+\.[A-Za-z]{2,})", q)

if m: reverse\_mail = m.group(1)

return {

"hostelish": hostelish, "gender": g,

"landlines": wants\_landlines, "contacts": wants\_contacts,

"blocks": wants\_blocks, "fees": wants\_fees,

"specific\_block": specific\_block,

"reverse\_phone": reverse\_phone, "reverse\_mail": reverse\_mail

}

def reverse\_lookup\_bundle(con: sqlite3.Connection, intent: Dict[str,Any]) -> str:

# reuse the mh\_blocks/lh\_blocks + hostel\_contacts for lookups

q\_phone = intent["reverse\_phone"]; q\_mail = intent["reverse\_mail"]

def \_get\_all\_contacts():

try:

return con.execute("SELECT role,name,phone,email FROM hostel\_contacts").fetchall()

except Exception:

return []

def \_get\_mh\_blocks():

try:

return con.execute("SELECT block\_code, block\_name, landline, IFNULL(email,'') as email FROM mh\_blocks").fetchall()

except Exception:

return []

def \_get\_lh\_blocks():

try:

return con.execute("SELECT block\_code, block\_code AS block\_name, landline, '' as email FROM lh\_blocks").fetchall()

except Exception:

return []

hits = []

if q\_phone:

n = re.sub(r"\D", "", q\_phone)

for r in \_get\_all\_contacts():

ph = re.sub(r"\D", "", r["phone"] or "")

if n and ph.find(n) != -1:

hits.append(["Contact","-", r["role"] or "", r["name"] or "", r["phone"] or "", r["email"] or ""])

for r in \_get\_mh\_blocks():

ph = re.sub(r"\D", "", r["landline"] or "")

if n and ph.find(n) != -1:

hits.append(["Block","MH", r["block\_code"] or "", r["block\_name"] or "", r["landline"] or "", r["email"] or ""])

for r in \_get\_lh\_blocks():

ph = re.sub(r"\D", "", r["landline"] or "")

if n and ph.find(n) != -1:

hits.append(["Block","LH", r["block\_code"] or "", r["block\_name"] or "", r["landline"] or "", ""])

if hits:

return md\_table("Reverse lookup — phone", ["Type","Hostel","Role/Code","Name","Phone","Email"], hits)

return "\_No matches for that phone number.\_"

if q\_mail:

mail = q\_mail.strip().lower()

for r in \_get\_all\_contacts():

if (r["email"] or "").lower() == mail:

hits.append(["Contact","-", r["role"] or "", r["name"] or "", r["phone"] or "", r["email"] or ""])

for r in \_get\_mh\_blocks():

if (r["email"] or "").lower() == mail:

hits.append(["Block","MH", r["block\_code"] or "", r["block\_name"] or "", r["landline"] or "", r["email"] or ""])

if hits:

return md\_table("Reverse lookup — email", ["Type","Hostel","Role/Code","Name","Phone","Email"], hits)

return "\_No matches for that email.\_"

return "\_No reverse lookup target found.\_"

def answer\_hostel\_sqlfirst(con: sqlite3.Connection, q: str, intent: Dict[str,Any]) -> Optional[str]:

if not con: return None

ql = q.lower()

# Landlines / blocks lists from mh\_blocks/lh\_blocks

if intent["landlines"] or intent["blocks"] or intent["specific\_block"]:

if intent["specific\_block"]:

code = intent["specific\_block"]

rows = []

mhr = con.execute("SELECT 'MH' as hostel, block\_code, block\_name, landline, IFNULL(email,'') as email FROM mh\_blocks WHERE UPPER(block\_code)=UPPER(?)", (code,)).fetchall()

lhr = con.execute("SELECT 'LH' as hostel, block\_code, block\_code as block\_name, landline, '' as email FROM lh\_blocks WHERE UPPER(block\_code)=UPPER(?)", (code,)).fetchall()

for r in list(mhr) + list(lhr):

rows.append([r["hostel"], r["block\_code"], r["block\_name"], r["landline"], r["email"]])

return md\_table("Block Details", ["Hostel","Block Code","Block Name","Landline","Email"], rows) if rows else "\_No matching block found.\_"

# list or landlines

if intent["gender"] in (None, "MH"):

mh = con.execute("SELECT block\_code, block\_name, landline, IFNULL(email,'') as email FROM mh\_blocks").fetchall()

if mh:

out = md\_table("Men’s Hostel — Block-wise Landlines", ["Block Code","Block Name","Landline","Email"],

[[r["block\_code"], r["block\_name"], r["landline"], r["email"]] for r in mh])

else:

out = ""

else:

out = ""

if intent["gender"] in (None, "LH"):

lh = con.execute("SELECT block\_code, block\_code as block\_name, landline, '' as email FROM lh\_blocks").fetchall()

if lh:

t = md\_table("Ladies’ Hostel — Block-wise Landlines", ["Block Code","Block Name","Landline","Email"],

[[r["block\_code"], r["block\_name"], r["landline"], r["email"]] for r in lh])

out = (out + "\n\n" + t) if out else t

return out or "\_No hostel blocks/landlines found in DB.\_"

# Contacts (hostel\_contacts)

if intent["contacts"]:

rows = con.execute("SELECT role, name, phone, email FROM hostel\_contacts ORDER BY role").fetchall()

if not rows: return "\_No contacts available.\_"

lines = []

for r in rows:

who = f"{r['role']}: {r['name']}" if r["name"] else r["role"]

bits = []

if r["phone"]: bits.append(r["phone"])

if r["email"]: bits.append(r["email"])

lines.append(f"{who} — " + " | ".join(bits) if bits else who)

return bullets(lines, "Hostel — Key Contacts")

# Fees: readable table from hostel\_fees + blocks

if intent["fees"]:

cond, args = [], []

if "2025-26" in ql:

cond.append("hf.ay=?"); args.append("2025-26")

if "nri" in ql: cond.append("LOWER(hf.category)='nri'")

if "indian" in ql: cond.append("LOWER(hf.category)='indian'")

if "foreign" in ql: cond.append("LOWER(hf.category)='foreign'")

sql = f"""

SELECT b.display\_name as block, b.gender, b.level, hf.ay, hf.category, hf.occupancy, hf.ac, hf.mess\_type,

hf.room\_mess\_fee, hf.admission\_fee, hf.caution\_deposit, hf.other\_fee, hf.total\_fee, hf.currency, hf.source\_file

FROM hostel\_fees hf

JOIN blocks b ON b.id=hf.block\_id

{"WHERE " + " AND ".join(cond) if cond else ""}

ORDER BY b.gender, b.level, b.block\_name, hf.occupancy, hf.ac DESC, hf.mess\_type

"""

rows = con.execute(sql, args).fetchall()

if rows:

return md\_table("Hostel Fee Details",

["Block","Gender","Level","AY","Category","Occ","AC","Mess","Room+Mess","Admission","Caution","Other","Total","Curr","Source"],

[[r["block"], r["gender"] or "", r["level"] or "", r["ay"] or "", r["category"] or "", r["occupancy"] or "",

"AC" if (r["ac"]==1) else ("Non-AC" if (r["ac"]==0) else ""), r["mess\_type"] or "", r["room\_mess\_fee"] or "",

r["admission\_fee"] or "", r["caution\_deposit"] or "", r["other\_fee"] or "", r["total\_fee"] or "", r["currency"] or "", r["source\_file"] or ""]

for r in rows

])

return "\_No hostel fee rows matched your filters.\_"

return None

ABOVE IS HOSTELS app/handlers

# app/handlers/rules.py — refund/attendance/discipline/exam etc.

import re, sqlite3

from typing import Optional

from app.utils.render import md\_table, bullets

RULE\_KEYS = ["refund","attendance","discipline","anti-ragging","anti ragging","exam","examination"]

def detect\_rules\_intent(q: str) -> bool:

ql = q.lower()

return any(k in ql for k in RULE\_KEYS)

def answer\_rules\_sql(con: sqlite3.Connection, q: str) -> Optional[str]:

if not con: return None

ql = q.lower()

cat = None

for k in RULE\_KEYS:

if k in ql:

cat = "anti-ragging" if k in ("anti ragging","anti-ragging") else k

break

sql = "SELECT category, title, text, IFNULL(ay,'') as ay, source\_file FROM rules"

args = []

if cat:

sql += " WHERE LOWER(category)=LOWER(?)"

args.append(cat)

sql += " ORDER BY ay DESC, title"

rows = con.execute(sql, args).fetchall()

if not rows:

return None

lines = [f"\*\*{r['title']}\*\* — {r['text']} \*(AY: {r['ay']} | src: {r['source\_file']})\*" for r in rows]

return bullets(lines, f"Rules — {cat.title() if cat else 'All'}")

ABOVE IS RULES app/handlers

import re

def norm(s: str) -> str:

return re.sub(r"[^a-z0-9]+","", (s or "").lower())

def level\_alias(txt: str) -> str:

if not txt: return ""

t = txt.lower()

if t in ("ug","undergrad","undergraduate","btech","b.tech","bsc","bca","bba"): return "UG"

if t in ("pg","postgrad","postgraduate","mtech","m.tech","msc","m.sc","mca","mba"): return "PG"

return txt

def category\_alias(txt: str) -> str:

if not txt: return ""

t = txt.lower()

if "nri" in t: return "NRI"

if "foreign" in t: return "Foreign"

if "indian" in t: return "Indian"

return txt

ABOVE IS ALIASES app/utils

# app/utils/fallback\_rag.py — FAISS retrieval + Gemini embeddings + simple summaries

import os, sys, re

from pathlib import Path

from typing import List, Tuple

from langchain\_community.vectorstores import FAISS

from langchain\_core.embeddings import Embeddings as LCEmbeddings

class GeminiLCEmbeddings(LCEmbeddings):

def \_\_init\_\_(self, model: str = "models/text-embedding-004", api\_key\_env: str = "GEMINI\_API\_KEY"):

import google.generativeai as genai

api\_key = os.getenv(api\_key\_env)

if not api\_key:

print("[ERROR] GEMINI\_API\_KEY not set.")

sys.exit(1)

genai.configure(api\_key=api\_key)

self.genai = genai

self.model = model

def embed\_documents(self, texts: List[str]) -> List[List[float]]:

return [self.genai.embed\_content(model=self.model, content=t)["embedding"] for t in texts]

def embed\_query(self, text: str) -> List[float]:

return self.genai.embed\_content(model=self.model, content=text)["embedding"]

def \_keyword\_score(text: str, q: str) -> int:

qwords = [w for w in re.findall(r"[a-z0-9]+", q.lower()) if len(w) > 2]

t = text.lower()

return sum(t.count(w) for w in qwords)

def \_postfilter\_and\_rerank(docs, query: str, topn: int = 12):

scored = []

for d in docs:

s = \_keyword\_score(d.page\_content or "", query)

scored.append((s, d))

scored.sort(key=lambda x: x[0], reverse=True)

return [d for s, d in scored][:topn]

def \_summarize(text: str) -> str:

blob = re.sub(r"\s+", " ", text or "").strip()

if not blob: return ""

parts = re.split(r"(?<=[.!?])\s+", blob)

return " ".join(parts[:8])

def faiss\_answer\_or\_summary(index\_dir: Path, collection: str, q: str) -> str:

emb = GeminiLCEmbeddings()

store = FAISS.load\_local(str(index\_dir), embeddings=emb, index\_name=collection, allow\_dangerous\_deserialization=True)

docs = store.similarity\_search(q, k=40)

docs = \_postfilter\_and\_rerank(docs, q, topn=12)

if not docs:

return "\_I couldn’t find enough context to answer that from your PDFs.\_"

ctx, sources = [], []

for d in docs:

md = d.metadata or {}

src = md.get("source\_file") or md.get("source\_title") or "unknown"

sources.append(src); ctx.append(d.page\_content or "")

text = "\n\n".join(ctx)

para = \_summarize(text)

srcs = []

seen = set()

for s in sources:

if s and s not in seen:

seen.add(s); srcs.append(s)

if len(srcs) >= 5: break

return f"\*\*Answer (from your PDFs):\*\* {para}\n\n\*Sources:\* {', '.join(srcs)}"

ABOVE IS FALLBACK.PY app/utils

from typing import List, Optional

def md\_table(title: str, cols: List[str], rows: List[List[str]]) -> str:

seen=set(); uniq=[]

for r in rows:

tup=tuple("" if c is None else str(c) for c in r)

if tup in seen: continue

seen.add(tup); uniq.append(list(tup))

out = []

if title: out.append(f"\*\*{title}\*\*")

out.append("| " + " | ".join(cols) + " |")

out.append("|" + "|".join(["---"]\*len(cols)) + "|")

for r in uniq:

out.append("| " + " | ".join("" if c is None else str(c) for c in r) + " |")

return "\n".join(out)

def bullets(lines: List[str], title: Optional[str]=None) -> str:

if not lines: return ""

seen=set(); out\_lines=[]

for ln in lines:

if ln in seen: continue

seen.add(ln); out\_lines.append(ln)

out = [f"\*\*{title}\*\*" if title else ""]

out += [f"- {ln}" for ln in out\_lines]

return "\n".join([x for x in out if x])

def join\_sources(srcs: List[str], maxn: int = 5) -> str:

if not srcs: return ""

uniq = []

seen = set()

for s in srcs:

if not s or s in seen: continue

seen.add(s); uniq.append(s)

if len(uniq) >= maxn: break

return ", ".join(uniq)

ABOVE IS RENDER.PY app/utils

# fix\_router\_and\_test.py — put this in project root and run with python

import os, pathlib, sqlite3, textwrap, sys

ROOT = pathlib.Path(\_\_file\_\_).resolve().parent

APP = ROOT / "app"

HAND = APP / "handlers"

UTIL = APP / "utils"

DB = ROOT / "Data" / "sql" / "vit\_vellore.db"

for p in [APP, HAND, UTIL]:

p.mkdir(parents=True, exist\_ok=True)

(p / "\_\_init\_\_.py").write\_text("", encoding="utf-8")

sql\_router\_code = r"""

import sqlite3, pathlib, re

from typing import Dict, Any, List

DB = pathlib.Path('Data/sql/vit\_vellore.db')

ACA\_KEYWORDS = {

'programs': ['program','course','degree','b.tech','m.tech','mca','msc','programme'],

'eligibility': ['eligibility','qualify','criteria','requirements','min marks'],

'documents': ['document','documents','docs','upload','certificate','proof','bonafide'],

'fees': ['tuition','fee','fees','academic fee','semester fee'],

'scholarships': ['scholarship','waiver','merit','financial aid']

}

def detect\_structured\_intent(q: str) -> str:

ql = q.lower()

if any(k in ql for k in ['contact','phone','email','supervisor','warden','director','manager']):

return 'contacts'

if 'block' in ql and any(k in ql for k in ['name','names','list','all','codes','code']):

return 'blocks'

if any(k in ql for k in ['hostel','room','mess','laundry','mh','lh']):

return 'hostel'

for tag, words in ACA\_KEYWORDS.items():

if any(w in ql for w in words):

return tag

if any(k in ql for k in ['fee','fees','tuition']):

return 'fees'

return 'text'

def parse\_filters(q: str) -> Dict[str, Any]:

ql = q.lower()

f = {'ay': None, 'gender': None, 'category': None, 'level': None, 'program\_like': None, 'level\_like': None}

m = re.search(r'\\b(20\\d{2})\\b', ql)

if m:

yr = m.group(1); f['ay'] = f'{yr}-{str(int(yr[-2:])+1).zfill(2)}'

if any(k in ql for k in ['boy','boys','men','mh','mens']): f['gender'] = 'Male'

if any(k in ql for k in ['girl','girls','ladies','lh','women']): f['gender'] = 'Female'

if 'nri' in ql: f['category'] = 'NRI'

elif 'foreign' in ql: f['category'] = 'Foreign'

elif 'indian' in ql: f['category'] = 'Indian'

if 'senior' in ql: f['level'] = 'Senior'

if 'first year' in ql or 'fresh' in ql: f['level'] = 'First-Year'

m = re.search(r'\\b(b\\.?tech|m\\.?tech|mca|msc|cse|ece|mechanical|biotech|ai|data)\\b', ql)

if m: f['program\_like'] = m.group(1).replace('.','')

if 'ug' in ql: f['level\_like'] = 'UG'

if 'pg' in ql: f['level\_like'] = 'PG'

return f

def \_mk():

con = sqlite3.connect(DB)

con.row\_factory = sqlite3.Row

return con

def \_safe\_fetch(con, sql, args) -> List[sqlite3.Row]:

try: return con.execute(sql, args).fetchall()

except sqlite3.OperationalError: return []

# -------- HOSTEL --------

def sql\_hostel\_overview(f: Dict[str,Any], limit\_rows: int = 800) -> Dict[str, Any]:

con = \_mk()

where = ['1=1']; args=[]

if f['gender']: where.append('IFNULL(b.gender,\"\")=IFNULL(?, \"\")'); args.append(f['gender'])

if f['level']: where.append('IFNULL(b.level,\"\")=IFNULL(?, \"\")'); args.append(f['level'])

if f['ay']: where.append('IFNULL(hf.ay,\"\")=IFNULL(?, \"\")'); args.append(f['ay'])

if f['category']: where.append('IFNULL(hf.category,\"\")=IFNULL(?, \"\")');args.append(f['category'])

where.append(\"\"\"(

COALESCE(hf.total\_fee,'')<>'' OR COALESCE(hf.room\_mess\_fee,'')<>'' OR

COALESCE(hf.admission\_fee,'')<>'' OR COALESCE(hf.caution\_deposit,'')<>'' OR

COALESCE(hf.occupancy,'')<>'' OR COALESCE(hf.mess\_type,'')<>''

)\"\"\")

sql = f\"\"\"

SELECT b.display\_name AS block, b.gender, b.level, b.block\_type,

hf.ay, hf.category, hf.occupancy, hf.ac, hf.mess\_type,

hf.room\_mess\_fee, hf.admission\_fee, hf.caution\_deposit, hf.other\_fee, hf.total\_fee, hf.currency,

hf.source\_file

FROM hostel\_fees hf

JOIN blocks b ON b.id = hf.block\_id

WHERE {' AND '.join(where)}

ORDER BY b.block\_type, block, hf.occupancy, hf.ac DESC, hf.mess\_type

\"\"\"

rows = \_safe\_fetch(con, sql, args); con.close()

columns = ['Block','Gender','Level','Type','AY','Category','Occ','AC','Mess',

'Room+Mess','Admission','Caution','Other','Total','Curr','Source']

tbl = {'title':'Hostel Fee Details (Vellore)', 'columns':columns, 'rows':[]}

for r in rows[:limit\_rows]:

tbl['rows'].append([

r['block'] or '', r['gender'] or '', r['level'] or '', r['block\_type'] or '',

r['ay'] or '', r['category'] or '', r['occupancy'] or '',

'AC' if (r['ac']==1) else ('Non-AC' if (r['ac']==0) else ''),

r['mess\_type'] or '', r['room\_mess\_fee'] or '', r['admission\_fee'] or '',

r['caution\_deposit'] or '', r['other\_fee'] or '', r['total\_fee'] or '',

r['currency'] or '', r['source\_file'] or ''

])

return {'table': tbl, 'bullets': []}

def sql\_block\_contacts(\_f: Dict[str,Any]) -> Dict[str,Any]:

con = \_mk()

rows = \_safe\_fetch(con, \"

SELECT '' AS block, name, role, phone, email

FROM hostel\_contacts

ORDER BY role, name

\", [])

con.close()

cols = ['Block','Name','Role','Phone','Email']

tbl = {'title':'Hostel Contacts', 'columns':cols, 'rows':[

[r['block'], r['name'] or '', r['role'] or '', r['phone'] or '', r['email'] or '']

for r in rows

]}

return {'table': tbl, 'bullets': []}

def sql\_list\_blocks(f: Dict[str,Any]) -> Dict[str,Any]:

con = \_mk()

if f['gender'] == 'Female':

sql = \"SELECT block\_code AS display\_name, 'Female' AS gender FROM lh\_blocks ORDER BY block\_code\"

elif f['gender'] == 'Male':

sql = \"SELECT COALESCE(block\_name, block\_code) AS display\_name, 'Male' AS gender FROM mh\_blocks ORDER BY block\_code\"

else:

sql = \"

SELECT COALESCE(block\_name, block\_code) AS display\_name, 'Male' AS gender FROM mh\_blocks

UNION ALL

SELECT block\_code AS display\_name, 'Female' AS gender FROM lh\_blocks

ORDER BY gender, display\_name

\"

rows = \_safe\_fetch(con, sql, []); con.close()

cols = ['Block','Gender']

tbl = {'title':'Hostel Blocks', 'columns':cols, 'rows':[

[r['display\_name'] or '', r['gender'] or ''] for r in rows

]}

return {'table': tbl, 'bullets': []}

# -------- ACADEMICS --------

def \_like\_clause(col: str, val: str, wh: list, args: list):

if val:

wh.append(f'LOWER({col}) LIKE ?')

args.append(f'%{val.lower()}%')

def sql\_programs(f: Dict[str,Any], q: str) -> Dict[str,Any]:

con = \_mk()

where, args = ['1=1'], []

\_like\_clause('level', f.get('level\_like'), where, args)

\_like\_clause('program', f.get('program\_like'), where, args)

\_like\_clause('program', q, where, args)

rows = \_safe\_fetch(con, f\"

SELECT level, program, school, duration, campus, source\_file

FROM programs

WHERE {' AND '.join(where)}

ORDER BY level, program

\", args); con.close()

cols = ['Level','Program','School','Duration','Campus','Source']

return {'table':{'title':'Programs','columns':cols,'rows':[

[r['level'] or '', r['program'] or '', r['school'] or '', r['duration'] or '', r['campus'] or '', r['source\_file'] or '']

for r in rows

]}, 'bullets':[]}

def sql\_eligibility(f: Dict[str,Any], q: str) -> Dict[str,Any]:

con = \_mk()

where, args = ['1=1'], []

\_like\_clause('level', f.get('level\_like'), where, args)

\_like\_clause('program', f.get('program\_like'), where, args)

\_like\_clause('criteria', q, where, args)

rows = \_safe\_fetch(con, f\"

SELECT level, program, criteria, source\_file

FROM eligibility

WHERE {' AND '.join(where)}

ORDER BY level, program

\", args); con.close()

cols = ['Level','Program','Criteria','Source']

return {'table':{'title':'Eligibility','columns':cols,'rows':[

[r['level'] or '', r['program'] or '', r['criteria'] or '', r['source\_file'] or '']

for r in rows

]}, 'bullets':[]}

def sql\_documents(f: Dict[str,Any], q: str) -> Dict[str,Any]:

con = \_mk()

where, args = ['1=1'], []

\_like\_clause('level', f.get('level\_like'), where, args)

\_like\_clause('program', f.get('program\_like'), where, args)

\_like\_clause('item', q, where, args)

rows = \_safe\_fetch(con, f\"

SELECT level, program, item, details, source\_file

FROM documents\_required

WHERE {' AND '.join(where)}

ORDER BY level, program, item

\", args); con.close()

cols = ['Level','Program','Document','Details','Source']

return {'table':{'title':'Documents Required','columns':cols,'rows':[

[r['level'] or '', r['program'] or '', r['item'] or '', r['details'] or '', r['source\_file'] or '']

for r in rows

]}, 'bullets':[]}

def sql\_academic\_fees(f: Dict[str,Any], q: str) -> Dict[str,Any]:

con = \_mk()

where, args = ['1=1'], []

\_like\_clause('program', f.get('program\_like'), where, args)

if f.get('level\_like'): where.append('LOWER(level)=?'); args.append(f['level\_like'].lower())

if f.get('category'): where.append('IFNULL(category,\"\")=IFNULL(?, \"\")'); args.append(f['category'])

if f.get('ay'): where.append('IFNULL(ay,\"\")=IFNULL(?, \"\")'); args.append(f['ay'])

\_like\_clause('program', q, where, args)

rows = \_safe\_fetch(con, f\"

SELECT level, program, category, ay, tuition, one\_time, caution, total, currency, source\_file

FROM academic\_fees

WHERE {' AND '.join(where)}

ORDER BY level, program, category, ay

\", args); con.close()

cols = ['Level','Program','Category','AY','Tuition','One-time','Caution','Total','Curr','Source']

return {'table':{'title':'Academic Fees','columns':cols,'rows':[

[r['level'] or '', r['program'] or '', r['category'] or '', r['ay'] or '',

r['tuition'] or '', r['one\_time'] or '', r['caution'] or '', r['total'] or '',

r['currency'] or '', r['source\_file'] or '']

for r in rows

]}, 'bullets':[]}

def sql\_scholarships(f: Dict[str,Any], q: str) -> Dict[str,Any]:

con = \_mk()

where, args = ['1=1'], []

\_like\_clause('level', f.get('level\_like'), where, args)

\_like\_clause('name', q, where, args)

rows = \_safe\_fetch(con, f\"

SELECT level, name, criteria, amount, currency, source\_file

FROM scholarships

WHERE {' AND '.join(where)}

ORDER BY level, name

\", args); con.close()

cols = ['Level','Name','Criteria','Amount','Curr','Source']

return {'table':{'title':'Scholarships','columns':cols,'rows':[

[r['level'] or '', r['name'] or '', r['criteria'] or '', r['amount'] or '', r['currency'] or '', r['source\_file'] or '']

for r in rows

]}, 'bullets':[]}

""".lstrip()

rag\_answer\_code = r"""

\"\"\"SQL-first router; FAISS fallback.\"\"\"

from app.sql\_router import (

detect\_structured\_intent, parse\_filters,

sql\_hostel\_overview, sql\_block\_contacts, sql\_list\_blocks,

sql\_programs, sql\_eligibility, sql\_documents, sql\_academic\_fees, sql\_scholarships

)

def \_fmt\_table(tbl: dict) -> str:

if not tbl or not tbl.get('rows'):

return '\_No matching rows.\_'

title = f\"\*\*{tbl.get('title','Results')}\*\*\"

cols = tbl['columns']

header = ' | '.join(cols)

sep = ' | '.join(['---']\*len(cols))

lines = [title, '', header, sep]

for r in tbl['rows']:

lines.append(' | '.join(str(x) if x is not None else '' for x in r))

return '\\n'.join(lines)

def \_pack(table\_dict=None, bullets=None):

parts = []

if table\_dict: parts.append(\_fmt\_table(table\_dict))

if bullets: parts.append('\\n'.join(f'- {b}' for b in bullets))

return '\\n\\n'.join(parts) if parts else '\_No results.\_'

def \_sql\_route(q: str) -> str | None:

intent = detect\_structured\_intent(q)

f = parse\_filters(q)

if intent == 'contacts':

return \_pack(sql\_block\_contacts(f).get('table'))

if intent == 'blocks':

return \_pack(sql\_list\_blocks(f).get('table'))

if intent in ('hostel','tabular'):

return \_pack(sql\_hostel\_overview(f).get('table'))

if intent == 'programs':

return \_pack(sql\_programs(f, q).get('table'))

if intent == 'eligibility':

return \_pack(sql\_eligibility(f, q).get('table'))

if intent == 'documents':

return \_pack(sql\_documents(f, q).get('table'))

if intent == 'fees':

return \_pack(sql\_academic\_fees(f, q).get('table'))

if intent == 'scholarships':

return \_pack(sql\_scholarships(f, q).get('table'))

return None

def answer(query: str) -> str:

sql\_text = \_sql\_route(query)

if sql\_text is not None:

return sql\_text

try:

from app.utils.fallback\_rag import answer as rag\_fallback

rag = rag\_fallback(query, max\_chunks=6, max\_tokens=450)

return rag or '\_I couldn't find that in my sources.\_'

except Exception:

return '\_I couldn't find that in my sources.\_'

if \_\_name\_\_ == '\_\_main\_\_':

import sys

q = ' '.join(sys.argv[1:]) or 'Show MH Senior NRI hostel fees 2025'

print(answer(q))

""".lstrip()

# Write files

(APP / "sql\_router.py").write\_text(sql\_router\_code, encoding="utf-8")

(APP / "rag\_answer.py").write\_text(rag\_answer\_code, encoding="utf-8")

print("[OK] Wrote app/sql\_router.py and app/rag\_answer.py")

# Basic DB sanity

if not DB.exists():

print(f"[WARN] DB not found at {DB}. Run ETL/load\_sqlite.py first.")

else:

con = sqlite3.connect(DB)

counts = {}

for t in ["programs","eligibility","documents\_required","academic\_fees","scholarships","hostel\_fees"]:

try:

counts[t] = con.execute(f"select count(\*) from {t}").fetchone()[0]

except Exception as e:

counts[t] = f"ERR: {e}"

con.close()

print("[DB] counts:", counts)

# Quick functional smoke tests

try:

from app.sql\_router import detect\_structured\_intent, parse\_filters, sql\_documents, sql\_programs, sql\_academic\_fees

print("[TEST] intent(documents):", detect\_structured\_intent("UG documents required"))

f = parse\_filters("UG documents required")

r = sql\_documents(f, "UG documents required")

print("[TEST] documents rows:", len(r["table"]["rows"]))

print("[TEST] intent(programs):", detect\_structured\_intent("UG programs CSE"))

f2 = parse\_filters("UG programs CSE")

r2 = sql\_programs(f2, "UG programs CSE")

print("[TEST] programs rows:", len(r2["table"]["rows"]))

print("[TEST] fees intent:", detect\_structured\_intent("B.Tech tuition fee Indian 2025"))

f3 = parse\_filters("B.Tech tuition fee Indian 2025")

r3 = sql\_academic\_fees(f3, "B.Tech tuition fee Indian 2025")

print("[TEST] fees rows:", len(r3["table"]["rows"]))

except Exception as e:

print("[TEST] error:", e)

# Print final answers preview

try:

from app.rag\_answer import answer

for q in [

"UG documents required",

"UG programs CSE",

"B.Tech tuition fee Indian 2025"

]:

print("\n=== Q:", q)

out = answer(q)

# show first 25 lines to keep console readable

lines = out.splitlines()

print("\n".join(lines[:25] if len(lines)>25 else lines))

except Exception as e:

print("[ANSWER] error:", e)

ABOVE ONE IS FIC\_ROUTER\_AND\_TEST.PY from app folder

"""

Router: SQL-first for fees/programs/eligibility/documents/scholarships/hostel.

Terse tables + tiny bullets. FAISS fallback only for long-form explainers.

"""

from app.sql\_router import (

detect\_structured\_intent, parse\_filters,

sql\_hostel\_overview, sql\_block\_contacts, sql\_list\_blocks,

sql\_programs, sql\_eligibility, sql\_documents, sql\_academic\_fees, sql\_scholarships

)

def \_fmt\_table(tbl: dict) -> str:

if not tbl or not tbl.get("rows"):

return "\_No matching rows.\_"

title = f"\*\*{tbl.get('title','Results')}\*\*"

cols = tbl["columns"]

header = " | ".join(cols)

sep = " | ".join(["---"]\*len(cols))

lines = [title, "", header, sep]

for r in tbl["rows"]:

lines.append(" | ".join(str(x) if x is not None else "" for x in r))

return "\n".join(lines)

def \_fmt\_bullets(bullets: list[str]) -> str:

return "\n".join(f"- {b}" for b in bullets) if bullets else ""

def \_pack(table\_dict=None, bullets=None):

parts = []

if table\_dict: parts.append(\_fmt\_table(table\_dict))

if bullets: parts.append(\_fmt\_bullets(bullets))

return "\n\n".join(parts) if parts else "\_No results.\_"

def \_sql\_route(q: str) -> str | None:

intent = detect\_structured\_intent(q)

f = parse\_filters(q)

# HOSTEL

if intent == "contacts":

return \_pack(sql\_block\_contacts(f).get("table"))

if intent == "blocks":

return \_pack(sql\_list\_blocks(f).get("table"))

if intent in ("hostel", "tabular"):

return \_pack(sql\_hostel\_overview(f).get("table"))

# ACADEMICS

if intent == "programs":

return \_pack(sql\_programs(f, q).get("table"))

if intent == "eligibility":

return \_pack(sql\_eligibility(f, q).get("table"))

if intent == "documents":

return \_pack(sql\_documents(f, q).get("table"))

if intent == "fees":

return \_pack(sql\_academic\_fees(f, q).get("table"))

if intent == "scholarships":

return \_pack(sql\_scholarships(f, q).get("table"))

return None

def answer(query: str) -> str:

sql\_text = \_sql\_route(query)

if sql\_text is not None:

return sql\_text

# FAISS fallback (tight + sourcey)

try:

from app.utils.fallback\_rag import answer as rag\_fallback

rag = rag\_fallback(query, max\_chunks=6, max\_tokens=450)

return rag or "\_I couldn't find that in my sources.\_"

except Exception:

return "\_I couldn't find that in my sources.\_"

if \_\_name\_\_ == "\_\_main\_\_":

import sys

q = " ".join(sys.argv[1:]) or "Show MH Senior NRI hostel fees 2025"

print(answer(q))

ABOVE IS RAG ANSWER.PY app folder

import sqlite3, pathlib, re

from typing import Dict, Any, List

DB = pathlib.Path("Data/sql/vit\_vellore.db")

ACA\_KEYWORDS = {

"programs": ["program","course","degree","b.tech","m.tech","mca","msc"],

"eligibility": ["eligibility","qualify","criteria","requirements","min marks"],

"documents": ["document","docs","upload","certificate","proof","bonafide"],

"fees": ["tuition","fee","fees","academic fee","semester fee"],

"scholarships": ["scholarship","waiver","merit","financial aid"]

}

def detect\_structured\_intent(q: str) -> str:

ql = q.lower()

# hostel

if any(k in ql for k in ["contact","phone","email","supervisor","warden","director","manager"]):

return "contacts"

if "block" in ql and any(k in ql for k in ["name","names","list","all","codes","code"]):

return "blocks"

if any(k in ql for k in ["hostel","room","mess","laundry","mh","lh"]):

return "hostel"

# academics

for tag, words in ACA\_KEYWORDS.items():

if any(w in ql for w in words): return tag

if any(k in ql for k in ["fee","fees","tuition"]):

return "tabular"

return "text"

def parse\_filters(q: str) -> Dict[str, Any]:

ql = q.lower()

f = {"ay": None, "gender": None, "category": None, "level": None, "program\_like": None, "level\_like": None}

m = re.search(r"\b(20\d{2})\b", ql)

if m:

yr = m.group(1); f["ay"] = f"{yr}-{str(int(yr[-2:])+1).zfill(2)}"

if any(k in ql for k in ["boy","boys","men","mh","mens"]): f["gender"] = "Male"

if any(k in ql for k in ["girl","girls","ladies","lh","women"]): f["gender"] = "Female"

if "nri" in ql: f["category"] = "NRI"

elif "foreign" in ql: f["category"] = "Foreign"

elif "indian" in ql: f["category"] = "Indian"

if "senior" in ql: f["level"] = "Senior"

if "first year" in ql or "fresh" in ql: f["level"] = "First-Year"

m = re.search(r"\b(b\.?tech|m\.?tech|mca|msc|cse|ece|mechanical|biotech|ai|data)\b", ql)

if m: f["program\_like"] = m.group(1).replace(".", "")

if "ug" in ql: f["level\_like"] = "UG"

if "pg" in ql: f["level\_like"] = "PG"

return f

def \_mk():

con = sqlite3.connect(DB)

con.row\_factory = sqlite3.Row

return con

def \_safe\_fetch(con, sql, args) -> List[sqlite3.Row]:

try: return con.execute(sql, args).fetchall()

except sqlite3.OperationalError: return []

# -------- HOSTEL --------

def sql\_hostel\_overview(f: Dict[str,Any], limit\_rows: int = 800) -> Dict[str, Any]:

con = \_mk()

where = ["1=1"]; args=[]

if f["gender"]: where.append("IFNULL(b.gender,'')=IFNULL(?, '')"); args.append(f["gender"])

if f["level"]: where.append("IFNULL(b.level,'')=IFNULL(?, '')"); args.append(f["level"])

if f["ay"]: where.append("IFNULL(hf.ay,'')=IFNULL(?, '')"); args.append(f["ay"])

if f["category"]: where.append("IFNULL(hf.category,'')=IFNULL(?, '')");args.append(f["category"])

where.append("""(

COALESCE(hf.total\_fee,'')<>'' OR COALESCE(hf.room\_mess\_fee,'')<>'' OR

COALESCE(hf.admission\_fee,'')<>'' OR COALESCE(hf.caution\_deposit,'')<>'' OR

COALESCE(hf.occupancy,'')<>'' OR COALESCE(hf.mess\_type,'')<>''

)""")

sql = f"""

SELECT b.display\_name AS block, b.gender, b.level, b.block\_type,

hf.ay, hf.category, hf.occupancy, hf.ac, hf.mess\_type,

hf.room\_mess\_fee, hf.admission\_fee, hf.caution\_deposit, hf.other\_fee, hf.total\_fee, hf.currency,

hf.source\_file

FROM hostel\_fees hf

JOIN blocks b ON b.id = hf.block\_id

WHERE {' AND '.join(where)}

ORDER BY b.block\_type, block, hf.occupancy, hf.ac DESC, hf.mess\_type

"""

rows = \_safe\_fetch(con, sql, args); con.close()

columns = ["Block","Gender","Level","Type","AY","Category","Occ","AC","Mess",

"Room+Mess","Admission","Caution","Other","Total","Curr","Source"]

tbl = {"title":"Hostel Fee Details (Vellore)", "columns":columns, "rows":[]}

for r in rows[:limit\_rows]:

tbl["rows"].append([

r["block"] or "", r["gender"] or "", r["level"] or "", r["block\_type"] or "",

r["ay"] or "", r["category"] or "", r["occupancy"] or "",

"AC" if (r["ac"]==1) else ("Non-AC" if (r["ac"]==0) else ""),

r["mess\_type"] or "", r["room\_mess\_fee"] or "", r["admission\_fee"] or "",

r["caution\_deposit"] or "", r["other\_fee"] or "", r["total\_fee"] or "",

r["currency"] or "", r["source\_file"] or ""

])

return {"table": tbl, "bullets": []}

def sql\_block\_contacts(\_f: Dict[str,Any]) -> Dict[str,Any]:

con = \_mk()

rows = \_safe\_fetch(con, """

SELECT '' AS block, name, role, phone, email

FROM hostel\_contacts

ORDER BY role, name

""", [])

con.close()

cols = ["Block","Name","Role","Phone","Email"]

tbl = {"title":"Hostel Contacts", "columns":cols, "rows":[

[r["block"], r["name"] or "", r["role"] or "", r["phone"] or "", r["email"] or ""]

for r in rows

]}

return {"table": tbl, "bullets": []}

def sql\_list\_blocks(f: Dict[str,Any]) -> Dict[str,Any]:

con = \_mk()

if f["gender"] == "Female":

sql = "SELECT block\_code AS display\_name, 'Female' AS gender FROM lh\_blocks ORDER BY block\_code"

elif f["gender"] == "Male":

sql = "SELECT COALESCE(block\_name, block\_code) AS display\_name, 'Male' AS gender FROM mh\_blocks ORDER BY block\_code"

else:

sql = """

SELECT COALESCE(block\_name, block\_code) AS display\_name, 'Male' AS gender FROM mh\_blocks

UNION ALL

SELECT block\_code AS display\_name, 'Female' AS gender FROM lh\_blocks

ORDER BY gender, display\_name

"""

rows = \_safe\_fetch(con, sql, []); con.close()

cols = ["Block","Gender"]

tbl = {"title":"Hostel Blocks", "columns":cols, "rows":[

[r["display\_name"] or "", r["gender"] or ""]

for r in rows

]}

return {"table": tbl, "bullets": []}

# -------- ACADEMICS --------

def \_like\_clause(col: str, val: str, wh: list, args: list):

if val:

wh.append(f"LOWER({col}) LIKE ?")

args.append(f"%{val.lower()}%")

def sql\_programs(f: Dict[str,Any], q: str) -> Dict[str,Any]:

con = \_mk()

where, args = ["1=1"], []

\_like\_clause("level", f["level\_like"], where, args)

\_like\_clause("program", f["program\_like"], where, args)

\_like\_clause("program", q, where, args)

rows = \_safe\_fetch(con, f"""

SELECT level, program, school, duration, campus, source\_file

FROM programs

WHERE {' AND '.join(where)}

ORDER BY level, program

""", args); con.close()

cols = ["Level","Program","School","Duration","Campus","Source"]

return {"table":{"title":"Programs","columns":cols,"rows":[

[r["level"] or "", r["program"] or "", r["school"] or "", r["duration"] or "", r["campus"] or "", r["source\_file"] or ""]

for r in rows

]}, "bullets":[]}

def sql\_eligibility(f: Dict[str,Any], q: str) -> Dict[str,Any]:

con = \_mk()

where, args = ["1=1"], []

\_like\_clause("level", f["level\_like"], where, args)

\_like\_clause("program", f["program\_like"], where, args)

\_like\_clause("criteria", q, where, args)

rows = \_safe\_fetch(con, f"""

SELECT level, program, criteria, source\_file

FROM eligibility

WHERE {' AND '.join(where)}

ORDER BY level, program

""", args); con.close()

cols = ["Level","Program","Criteria","Source"]

return {"table":{"title":"Eligibility","columns":cols,"rows":[

[r["level"] or "", r["program"] or "", r["criteria"] or "", r["source\_file"] or ""]

for r in rows

]}, "bullets":[]}

def sql\_documents(f: Dict[str,Any], q: str) -> Dict[str,Any]:

con = \_mk()

where, args = ["1=1"], []

\_like\_clause("level", f["level\_like"], where, args)

\_like\_clause("program", f["program\_like"], where, args)

\_like\_clause("item", q, where, args)

rows = \_safe\_fetch(con, f"""

SELECT level, program, item, details, source\_file

FROM documents\_required

WHERE {' AND '.join(where)}

ORDER BY level, program, item

""", args); con.close()

cols = ["Level","Program","Document","Details","Source"]

return {"table":{"title":"Documents Required","columns":cols,"rows":[

[r["level"] or "", r["program"] or "", r["item"] or "", r["details"] or "", r["source\_file"] or ""]

for r in rows

]}, "bullets":[]}

def sql\_academic\_fees(f: Dict[str,Any], q: str) -> Dict[str,Any]:

con = \_mk()

where, args = ["1=1"], []

\_like\_clause("program", f["program\_like"], where, args)

if f["level\_like"]: where.append("LOWER(level)=?"); args.append(f["level\_like"].lower())

if f["category"]: where.append("IFNULL(category,'')=IFNULL(?, '')"); args.append(f["category"])

if f["ay"]: where.append("IFNULL(ay,'')=IFNULL(?, '')"); args.append(f["ay"])

\_like\_clause("program", q, where, args)

rows = \_safe\_fetch(con, f"""

SELECT level, program, category, ay, tuition, one\_time, caution, total, currency, source\_file

FROM academic\_fees

WHERE {' AND '.join(where)}

ORDER BY level, program, category, ay

""", args); con.close()

cols = ["Level","Program","Category","AY","Tuition","One-time","Caution","Total","Curr","Source"]

return {"table":{"title":"Academic Fees","columns":cols,"rows":[

[r["level"] or "", r["program"] or "", r["category"] or "", r["ay"] or "",

r["tuition"] or "", r["one\_time"] or "", r["caution"] or "", r["total"] or "",

r["currency"] or "", r["source\_file"] or ""]

for r in rows

]}, "bullets":[]}

def sql\_scholarships(f: Dict[str,Any], q: str) -> Dict[str,Any]:

con = \_mk()

where, args = ["1=1"], []

\_like\_clause("level", f["level\_like"], where, args)

\_like\_clause("name", q, where, args)

rows = \_safe\_fetch(con, f"""

SELECT level, name, criteria, amount, currency, source\_file

FROM scholarships

WHERE {' AND '.join(where)}

ORDER BY level, name

""", args); con.close()

cols = ["Level","Name","Criteria","Amount","Curr","Source"]

return {"table":{"title":"Scholarships","columns":cols,"rows":[

[r["level"] or "", r["name"] or "", r["criteria"] or "", r["amount"] or "", r["currency"] or "", r["source\_file"] or ""]

for r in rows

]}, "bullets":[]}

ABOVE IS SQL\_ROUTER.PY app folder