RAG 技术详解与实战应用

第11讲：性能优化指南：

从冷启动到响应加速你的RAG

目录

上节回顾

Deepseek简介

- DeepSeek相比于业界其他模型的典型特征是具备思维链的能力
- 其采用的是混合专家模型（MoE），采用大规模强化学习与高质量合成数据结合，无需依赖大量标注数据
- 模型具备更强大的思维能力
- 在线厂商提供了Deepseek的推理能力，大家可以基于LazyLLM使用

Deepseek的部署和蒸馏

- 通过LazyLLM一键部署满血Deepseek，并展示使用后的效果优势和劣势
- Deepseek R1蒸馏的意义
- 蒸馏步骤：
  - 数据集准备
  - 模型微调
  - 评测

Deepseek赋能RAG

- 思维链数据结合通用数据混训小模型
- 在RAG中使用带思维连的模型

目录

召回优化增加性能负担

先前课程中，我们使用多种策略优化召回效果,然而多环节的引入也造成了RAG系统的启动及响应速度大大下降，系统性能遭遇瓶颈。

召回优化

- 查询重写、子问题查询、多步骤查询
- 创建多节点组，多路召回，混合检索
- 引入Reranker重排序

……

搜得准，答得好

But

响应慢，体验差

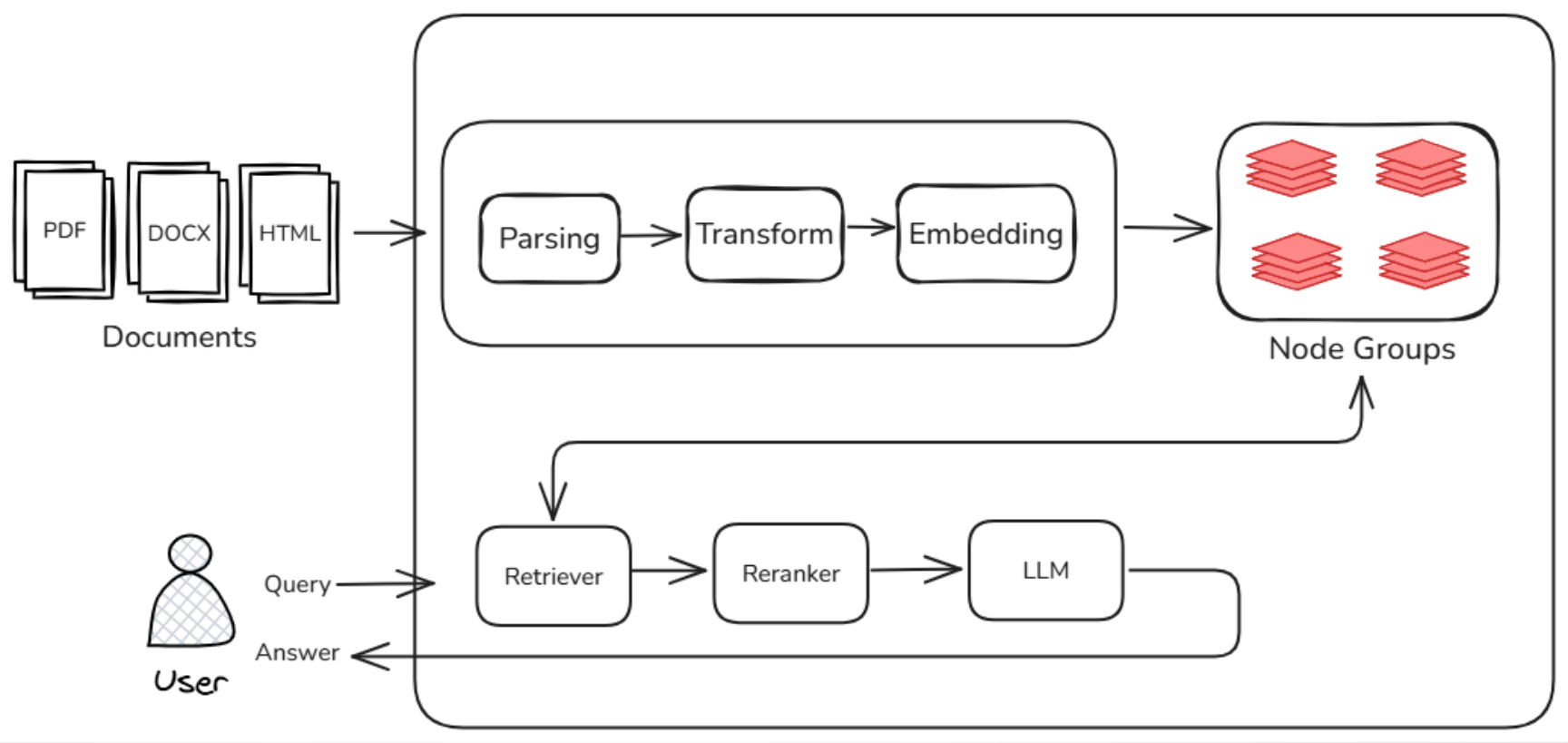
RAG系统的主要阶段

RAG系统的流程主要分为以下三个阶段：

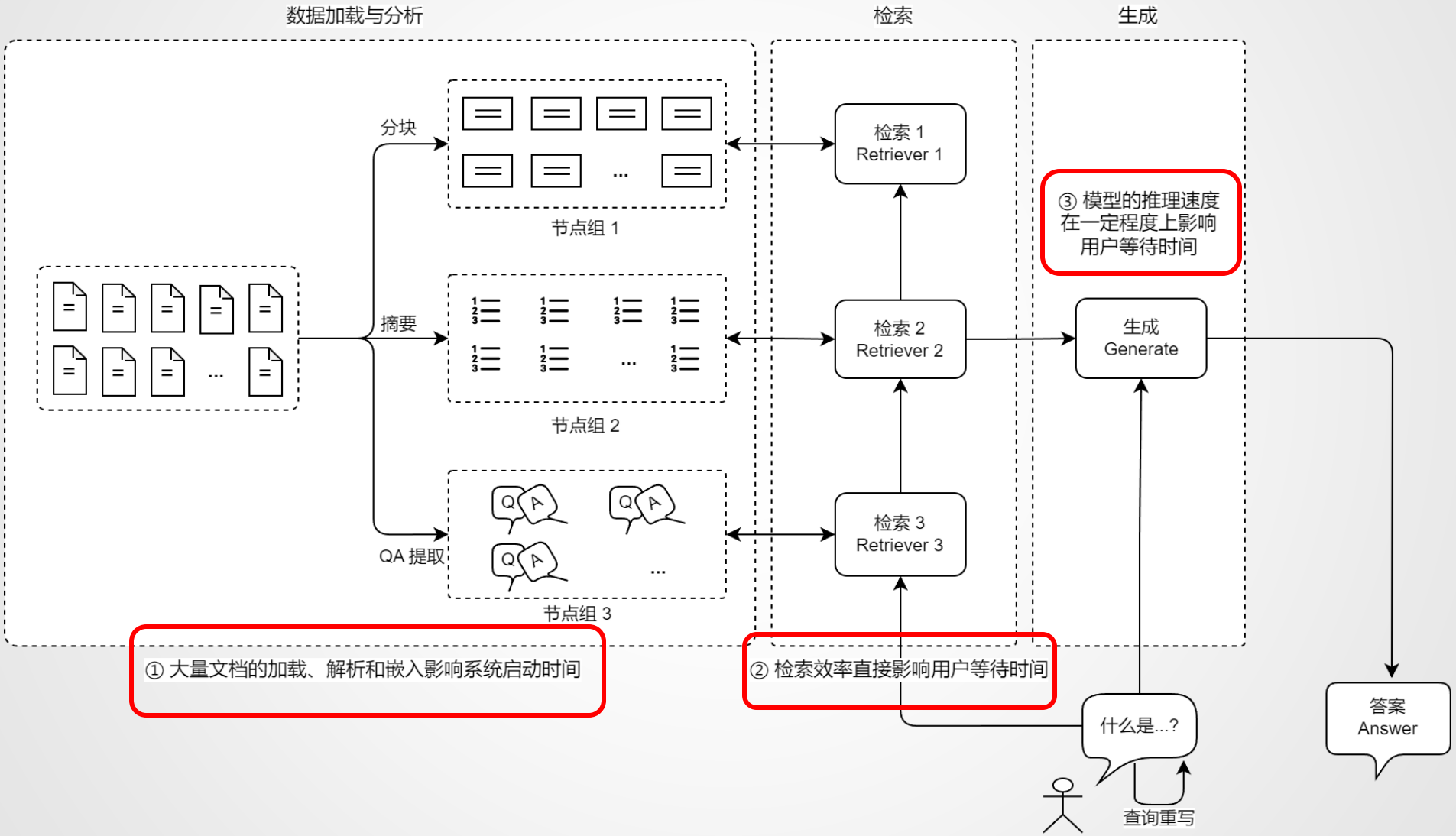
· 文档入库

· 查询检索

· 回答生成



召回优化增加性能负担


数据加载与分析
检索
生成
分块
摘要
QA 提取
节点组 1
节点组 2
节点组 3
检索 1
Retriever 1
检索 2
Retriever 2
检索 3
Retriever 3
③ 模型的推理速度在一定程度上影响用户等待时间
生成
Generate
① 大量文档的加载、解析和嵌入影响系统启动时间
② 检索效率直接影响用户等待时间
什么是...?
查询重写
答案
Answer

RAG系统性能瓶颈

RAG系统在不同阶段遇到的**性能瓶颈**：

· **文档入库阶段：**

文档数量越多、处理逻辑越复杂，文档入库速度越慢，大幅增加**系统启动时间**

· **查询检索阶段：**

检索效率的高低，直接决定了用户等待时间的长短

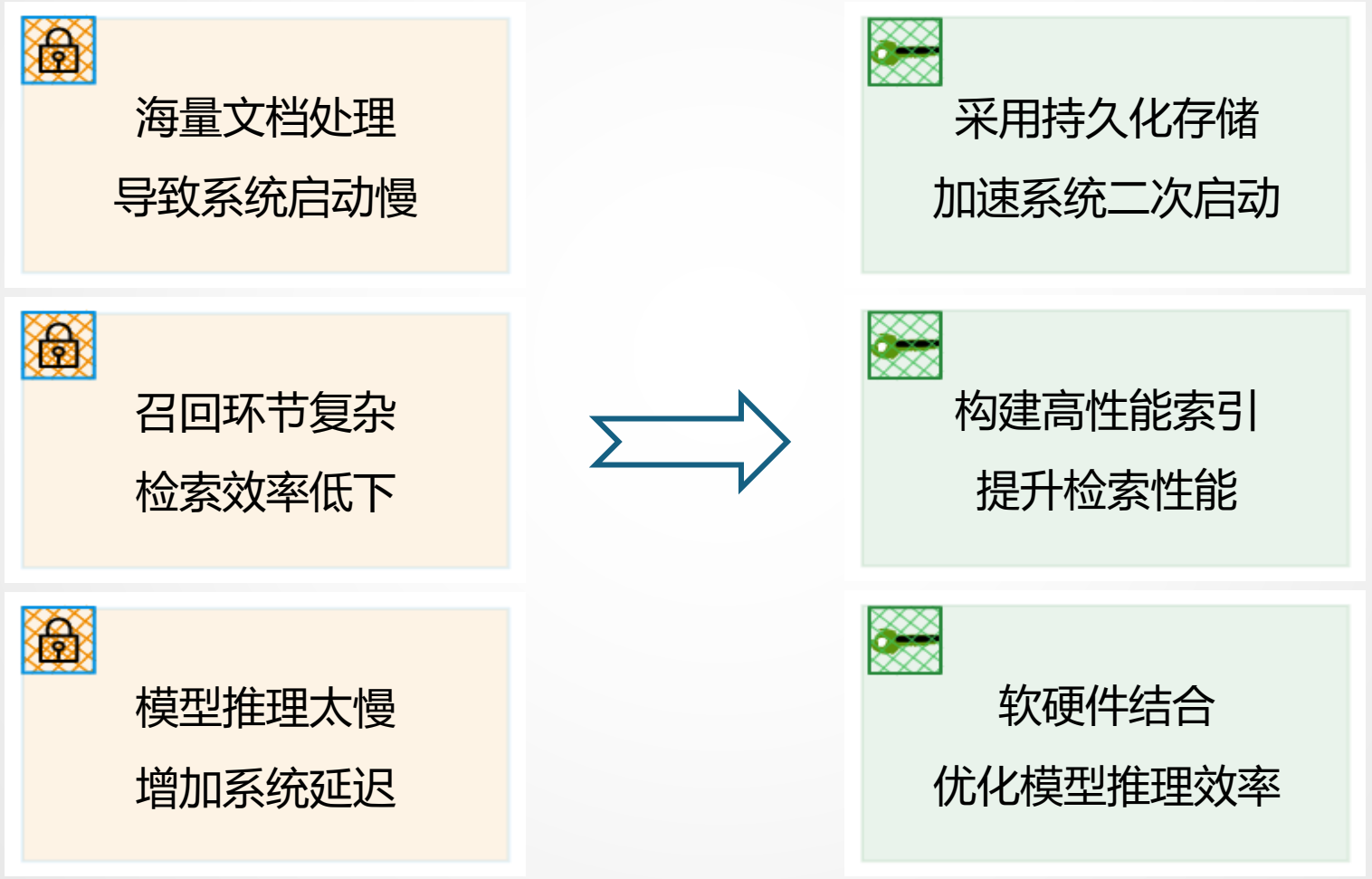
· **生成推理阶段：**

模型推理慢，进一步影响系统响应时间

（p.s. 流式输出只能缩短答案生成的首字延迟，无法缓解中间阶段使用模型带来的延迟）

优化指南概览

本节课程将针对上述的每一个话题，介绍具体的性能优化方案



目录

RAG启动流程

先前搭建的RAG系统，启动时会经历以下环节：

·程序加载

RAG系统启动时，程序首先会从硬盘加载至内存中，并在加载完毕后运行。

·文档加载

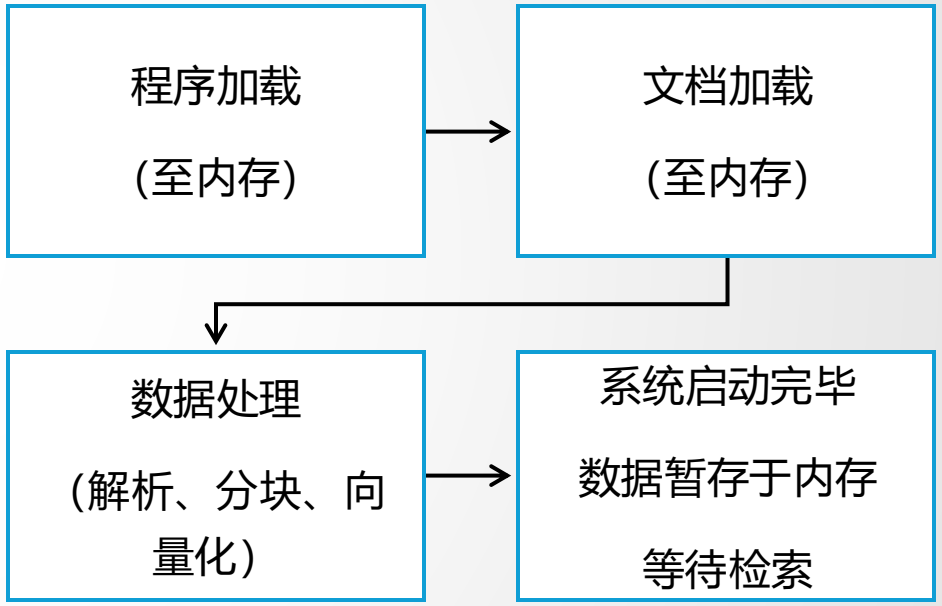
根据路径配置，系统访问硬盘，将期望入库的文件加载至内存当中。

·数据处理

继续执行程序，对入库文件执行解析、分块、数据提取等步骤，并进行向量化。以上环节均发生在内存。

·启动完毕

上述流程完成后，生成的文档切片及向量数据暂存于内存当中，等待后续的检索。



全都在内存？？？

RAG启动流程 – 数据在内存中的问题

RAG启动流程 – 数据在内存中的问题

RAG启动流程

将所有的数据存储在内存中会引发一系列问题：

数据丢失

内存具有“易失性”，即内存中的数据在断电或程序重启时会全部丢失。

系统启动慢

每次系统重启都需要把所有文档重新处理一遍，对于文档数量很大的场景，这将严重影响系统启动性能。

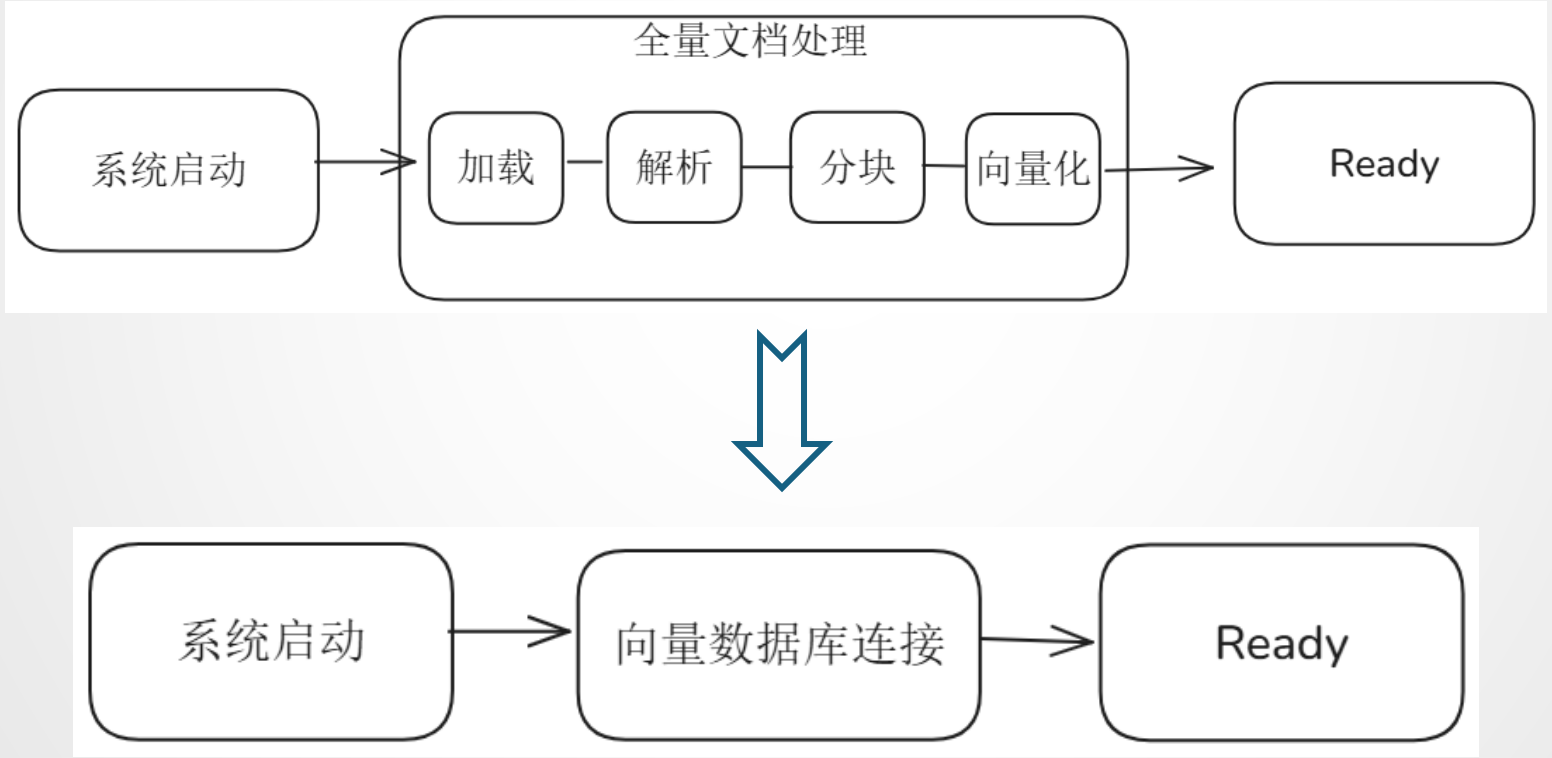
资源占用率高

每次问答仅需极少量的切片和向量数据，所有数据存在内存中会造成大量资源浪费，甚至影响系统性能。

持久化存储提升启动效率

解决方案：

使用**向量数据库，**将向量、切片以及元数据存储在**硬盘**中，以实现数据的**持久化存储**。



将持久化存储接入RAG

优势：

·减少重复计算

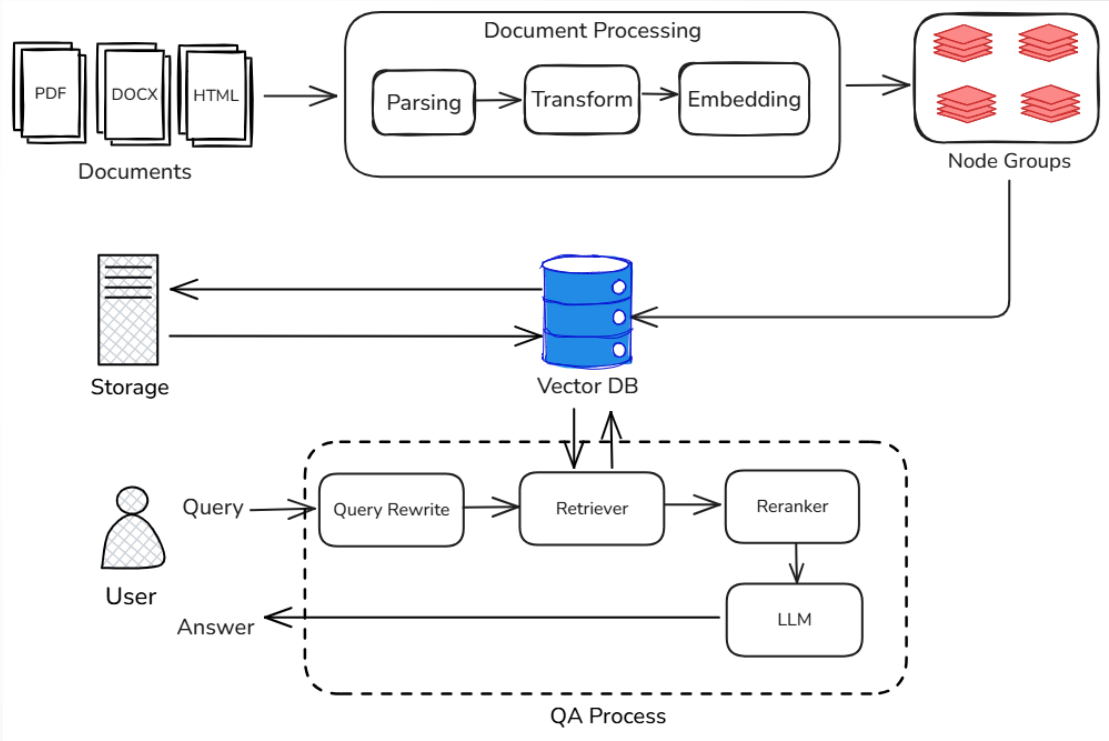
对于已存储数据，系统直接从硬盘中加载，避免系统重启时重新入库；

·提高内存利用率

将多数数据从内存中释放，仅在检索时加载相关数据，提高内存利用率；

·提升查询一致性

避免因模型更新、随机初始化带来的不稳定性，保证数据一致性，提高系统的可靠性。



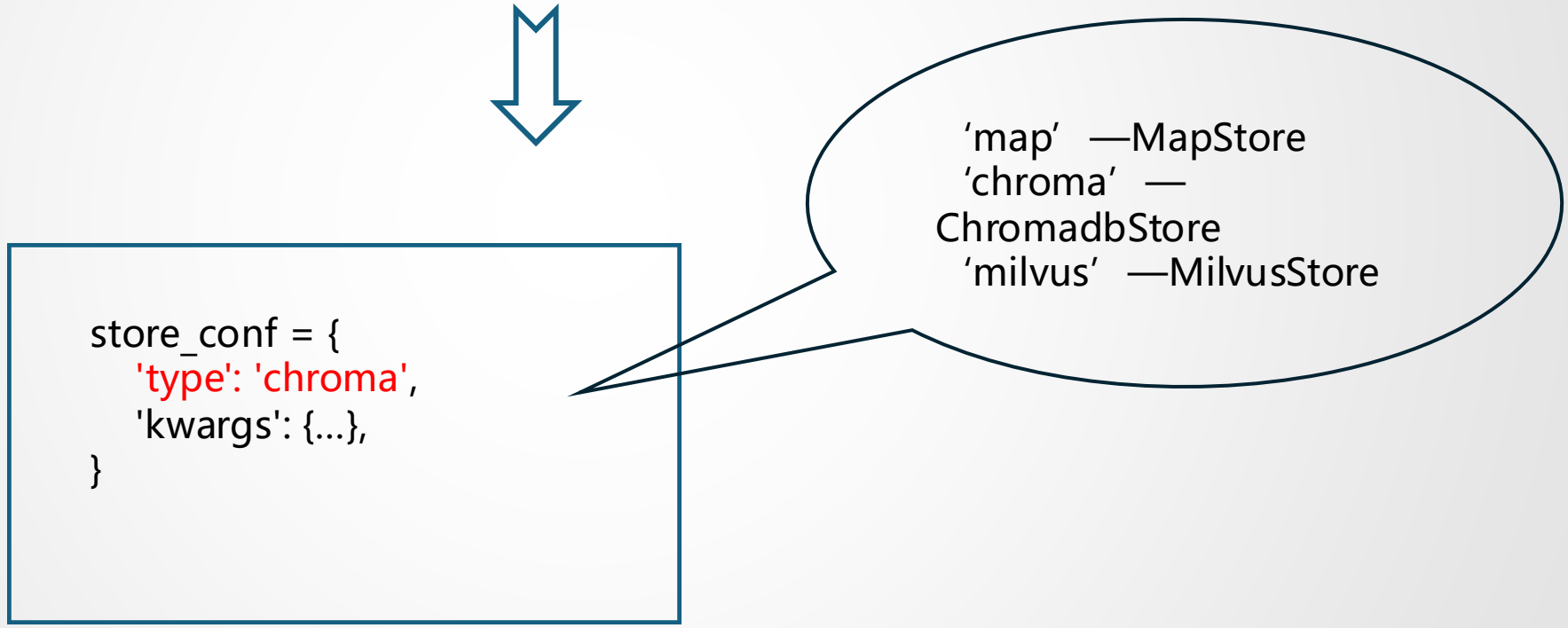
持久化存储提升启动效率 - LazyLLM

LazyLLM提供了多种存储组件同时支持基于**内存**和**向量数据库**两种存储方式，以供开发者灵活选择。

组件名	存储介质	功能描述	适用场景
MapStore	内存	基于内存的存储，提供节点存储的基本能力，能以最快的速度实现数据读写	技术探索阶段，将少量文档处理数据保存在内存中，以技术选型与算法迭代的快速验证
ChromadbStore	硬盘	使用 Chroma向量数据库，实现数据的持久化存储。	轻量级实验，快速POC，缩短从构思到可运行原型的研发周期
MilvusStore	硬盘	使用Milvus向量数据库，实现数据的本地存储/分布式远程存储。	企业级部署，适用于数据体量大，对并发、检索性能都有较高要求的场景

持久化存储提升启动效率 - LazyLLM

documents = lazyllm.Document(dataset_path="xxx", embed=xxx, ..., **store_conf={...}**)



扩展学习（可选）– 程序运行机制

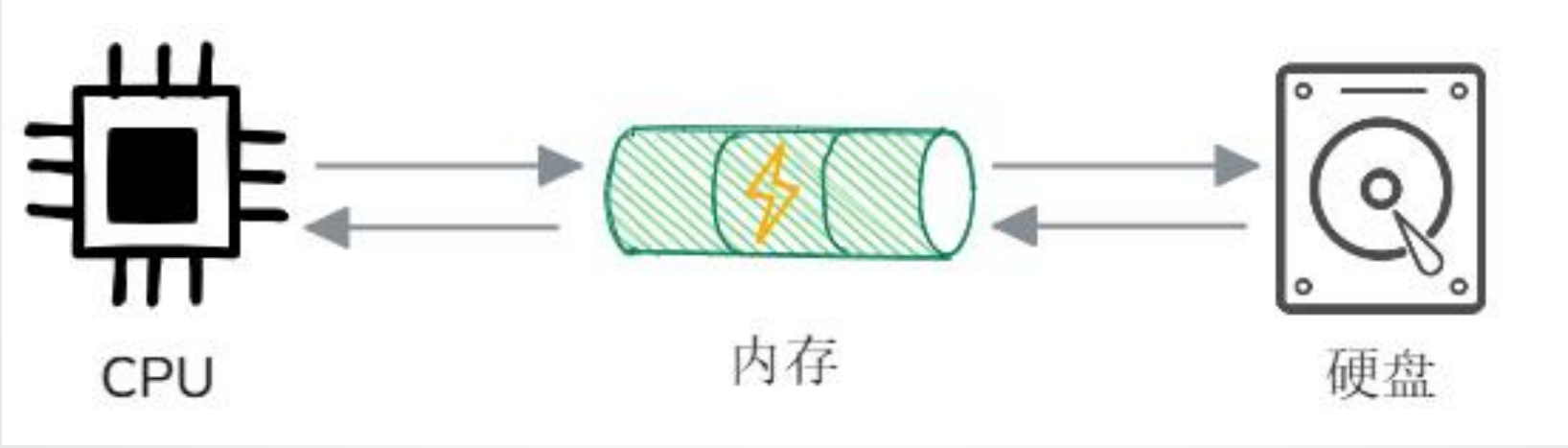
程序运行主要涉及三大核心组件：**CPU、内存和外部存储**（如硬盘）

·程序从硬盘加载至内存并运行

·CPU从内存中读取程序指令并执行

·数据读取：CPU请求操作系统访问硬盘，数据从硬盘加载至内存

·数据写入：CPU先将数据写入内存，最终由操作系统写入硬盘，以实现持久化存储



扩展学习（可选）- 内存与硬盘

内存（Memory，通常指RAM）：临时存储设备，主要用于CPU和硬盘之间的数据交换，是计算机执行程序和处理数据时的临时工作空间。

·高速访问： 读写速度极快（通常以纳秒（ns）计）

·易失性： 数据在断电或服务进程重启时会全部丢失

·容量相对较小：常见内存容量一般在几GB到几百GB之间

·随机访问能力： CPU可以以任意顺序、同等速度访问内存中任意位置数据

硬盘（Storage，通常指HDD/SSD）：持久性存储设备，用于长期保存程序、数据和文件。

·持久存储：数据一旦写入硬盘，即使断电也不会丢失，能够长期保存数据

·容量大：通常从几百GB到数TB甚至数十TB

·访问速度较慢： 机械硬盘（HDD）通常有毫秒级延迟，而固态硬盘（SSD）则快得多，但整体仍远慢于内存

扩展学习（可选）- 硬盘性能演进

·机械硬盘 (HDD)：

依赖旋转碟片+磁头寻道，随机访问需等待转速同步与寻道完成，平均延迟≈4ms（典型7200 RPM HDD）

·SATA 固态硬盘 (SATA SSD)：

无机械部件，闪存 + SATA 6Gb/s接口，随机延迟可低至≈0.04ms，且随机IOPS飙升到数万级

·NVMe 固态硬盘 (NVMe SSD)：

借助PCIe高速总线+并行队列，随机延迟进一步压缩到约5μs量级；顶级Optane/P5800X随机IOPS超1.5M，顺序吞吐可上7 GB/s以上

存储介质	平均随机延迟*	随机 IOPS*	顺序读取吞吐*
HDD（7200 RPM）	≈ 4.2 ms	≈ 400	≈ 150 MB/s
SATA SSD	≈ 0.036 ms	≈ 97 K	≈ 600 MB/s
NVMe SSD	≈ 0.005 ms	≈ 1 500 K	≈ 7200 MB/s

（用NVMe SSD能够把I/O瓶颈压缩到微秒级，但别忘了容量与预算的平衡 —— “性能+容量+成本”三角永远需要权衡。）

扩展学习（可选）- 内存与硬盘

“技术选型（如Embedding模型、Reader）及系统设计（如分块策略、节点组设计）暂未确定，相比海量数据的稳定存储，更倾向于选择少量文档，快速验证系统表现。”

——内存存储

“系统设计及算法优化逐步走向稳定，海量正式文档入库检索，不希望每次运行程序都要花费大量时间等待文档重新入库，期望缩短系统启动时间，数据持久存储。”

——硬盘存储

特性	内存（RAM）	硬盘（HDD/SSD）
存储时效性	易失性，断电数据丢失	持久性，断电数据不丢失
存储容量	较小（数GB至数百GB）	较大（数百GB至数十TB）
数据访问速度	极快（纳秒级）	较慢（毫秒级HDD，微秒级SSD）
价格	单位容量价格昂贵	单位容量价格相对便宜
用途	临时存储，程序运行时数据暂存	长期存储，数据、程序和系统持久保存

扩展学习（可选）- 内存缓存机制

内存缓存（Memory Cache）机制是一种通过**预存储高频访问数据**来减少重复计算优化策略，合理使用缓存机制可显著提升系统响应速度并降低资源消耗。内存缓存机制通常用高性能内存缓存系统实现，常用的内存缓存系统有**Redis和Memcached**。

Redis：

·定位：

功能丰富的缓存 + 轻量持久化

·核心特性：

多数据结构（List/Set/Zset/Hash）

TTL（设置过期时间） & Lua 脚本

RDB / AOF 持久化

·典型用法：

热向量缓存、会话上下文、排行榜

Memcached：

·定位：

极致高并发 KV 缓存

·核心特性：

纯内存、无持久化

简单协议、单数据结构

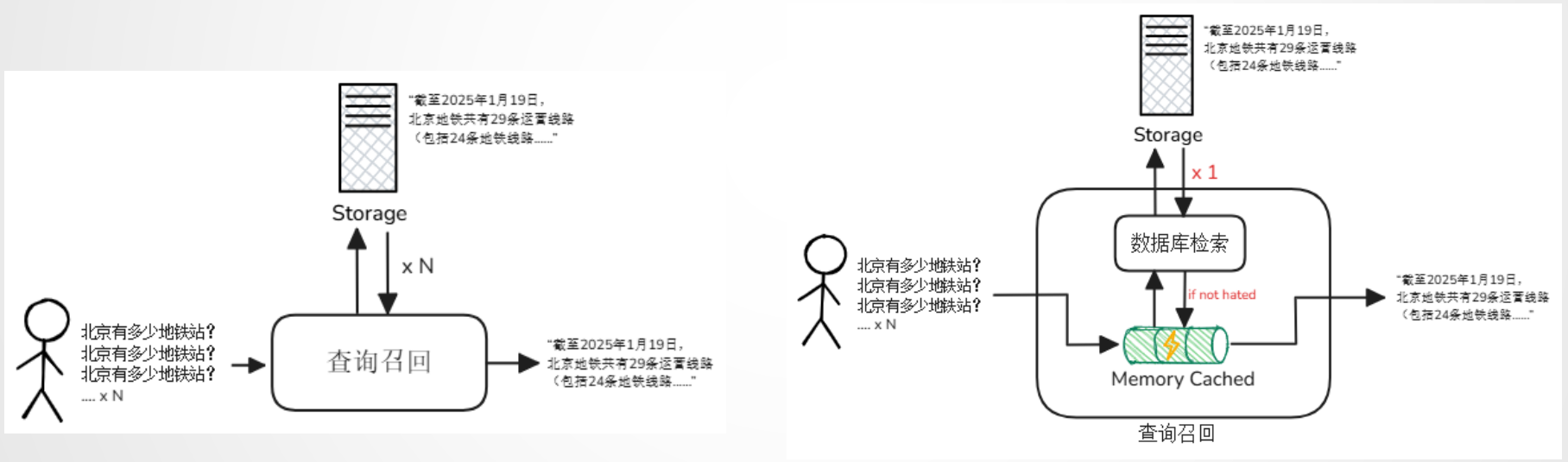
超低延迟（微秒级）

·典型用法：

模型参数缓存、Session、HTML 片段

扩展学习（可选）- 内存缓存机制

在RAG场景下，通常将“热点”数据缓存在内存中，比如经常查询的向量或检索结果，以便后续相似查询可以直接从内存获取这些数据，这种**“以空间换时间”**的方案可以有效减少重复计算和磁盘I/O，最大程度上弥补磁盘存储的性能差距，让RAG检索既**兼顾持久化又不失速度**。



无内存缓存 VS 有内存缓存

扩展学习（可选）- 数据库

数据库（Database）是一种用于有组织地存储、管理、检索和操作数据的软件系统，具有**高效存储、高效访问、支持并发与一致性**的特点。按照存储数据的类型，数据库可以分为**标量数据库**和**向量数据库**。

标量数据库就是**传统关系型数据库**，主要用于处理**结构化数据**（字段清晰的表格状信息如例如ID、姓名、日期，价格、状态等）。标量数据库擅长做**精确查询、聚合分析、事务处理**等任务，广泛应用在金融、审计、医疗、供应链等领域。常用标量数据库如MySQL、PostgreSQL、MongoDB等。

files (7538 rows)

```
SELECT * FROM 'files' LIMIT 0,30
```

执行

id	file_name	file_path	file_type	is_dir	file_size	parent_id	create_time
a98e0911-bc0d-4270-9f1f-b...	_root		FOLDER	1	null	null	2025-03-14 14:47:45.12
8b95ed4f-ba5d-47cd-91f8-e...	_kb_root_dir_	_kb_root_dir_	FOLDER	1	null	a98e0911-bc0d-4270-9f1f-b...	2025-03-14 14:53:38.32
573f0398-b6a2-4eea-b6b3-...	test_1234	_kb_root_dir_/test_1234	FOLDER	1	null	8b95ed4f-ba5d-47cd-91f8-e...	2025-03-14 14:53:38.33
345dce06c59f4d131499c4a...	这是一个pdfocr.pdf	_kb_root_dir_/test_1234/sti...	PDF	0	54348	573f0398-b6a2-4eea-b6b3-...	2025-03-14 14:54:11.68
9261b591949d0bfda6b6df0...	test_doc_trans.pdf	_kb_root_dir_/test_1234/sti...	PDF	0	243405	573f0398-b6a2-4eea-b6b3-...	2025-03-14 14:54:46.83
7649e067-5495-4613-be9d-...	pptx	_kb_root_dir_/pptx	FOLDER	1	null	8b95ed4f-ba5d-47cd-91f8-e...	2025-03-14 15:36:48.26
0ed65282b7dd66b8666a89...	system_info.txt	_kb_root_dir_/pptx/stime_0...	TXT	0	4521	7649e067-5495-4613-be9d-...	2025-03-14 15:38:06.28
dc64cfc4-48ea-44df-a218-7...	上传url	_kb_root_dir_/上传url	FOLDER	1	null	8b95ed4f-ba5d-47cd-91f8-e...	2025-03-14 16:09:12.34
1f9a3948-8328-491c-9013-...	test	_kb_root_dir_/test	FOLDER	1	null	8b95ed4f-ba5d-47cd-91f8-e...	2025-03-14 16:10:25.84
ba01c13618d3aa03a832f6d...	system_info.txt	_kb_root_dir_/test/stime_03...	TXT	0	4521	1f9a3948-8328-491c-9013-...	2025-03-14 16:11:20.07
68d9c895d9a089e7156356...	MDN_Web_Docs.html	_kb_root_dir_/test/stime_03...	HTML	0	null	1f9a3948-8328-491c-9013-...	2025-03-14 16:14:05.86
63b656911913e49fd9aa64c...	Free_eBooks__Project_Gut...	_kb_root_dir_/test/stime_03...	HTML	0	null	1f9a3948-8328-491c-9013-...	2025-03-14 16:14:39.95
72893741-bea0-4a2d-a0f6-f...	test_66	_kb_root_dir_/test_66	FOLDER	1	null	8b95ed4f-ba5d-47cd-91f8-e...	2025-03-14 16:51:12.39
d4f12b0bad7efe1d409d39c...	system_info.txt	_kb_root_dir_/test_66/stime...	TXT	0	4521	72893741-bea0-4a2d-a0f6-f...	2025-03-14 16:51:32.32
088ce49d2ebb7ed355650b...	Free_eBooks__Project_Gut...	_kb_root_dir_/test_66/stime...	HTML	0	null	72893741-bea0-4a2d-a0f6-f...	2025-03-14 16:52:04.91

标量数据库中数据表示意图

扩展学习（可选）- 数据库

向量数据库主要面向非结构化数据（比如文本向量、图像向量等）的存储和相似度检索任务，其专门提供了高效的向量索引结构，可进行大规模数据的高效检索。同时一些向量数据库也支持向量+元数据的混合过滤，方便做基于语义理解的检索、推荐与多模态问答。常用的向量数据库如Chroma、Faiss、Milvus等。

功能 / 特性	Faiss	Chroma	Milvus
服务端 / 数据库形态	✗（库）	✓（本地单机）	✓（分布式集群）
持久化存储	✗（需自行接入）	✓（SQLite 默认）	✓（内置冷热分层）
多索引类型 (IVF / HNSW / PQ / DiskANN…)	✓	✓（follow Faiss）	✓（含 DiskANN 等）
GPU 加速	✓	✗	✓
向量+标量过滤查询	✗	✓（简单）	✓
水平扩展 / 分布式	✗	✗	✓
检索性能	高	较低	高
适用场景	学术 / 算法实验	轻量应用	工业级 / 生态完善

目录

什么是索引（Index）

索引（Index）是一种为了加速数据检索而创建的一种数据结构，其作用是使检索器能够高效定位和检索相关信息。简单来说，索引就像是**书籍中的目录**，它帮助快速定位特定信息，避免逐页查找。

无索引检索

顺序扫描数据集合中的每个元素，逐个寻找与查询相关的内容.这个过程被称为**线性搜索（Linear Search）**，时间复杂度是 **O(n)** 。

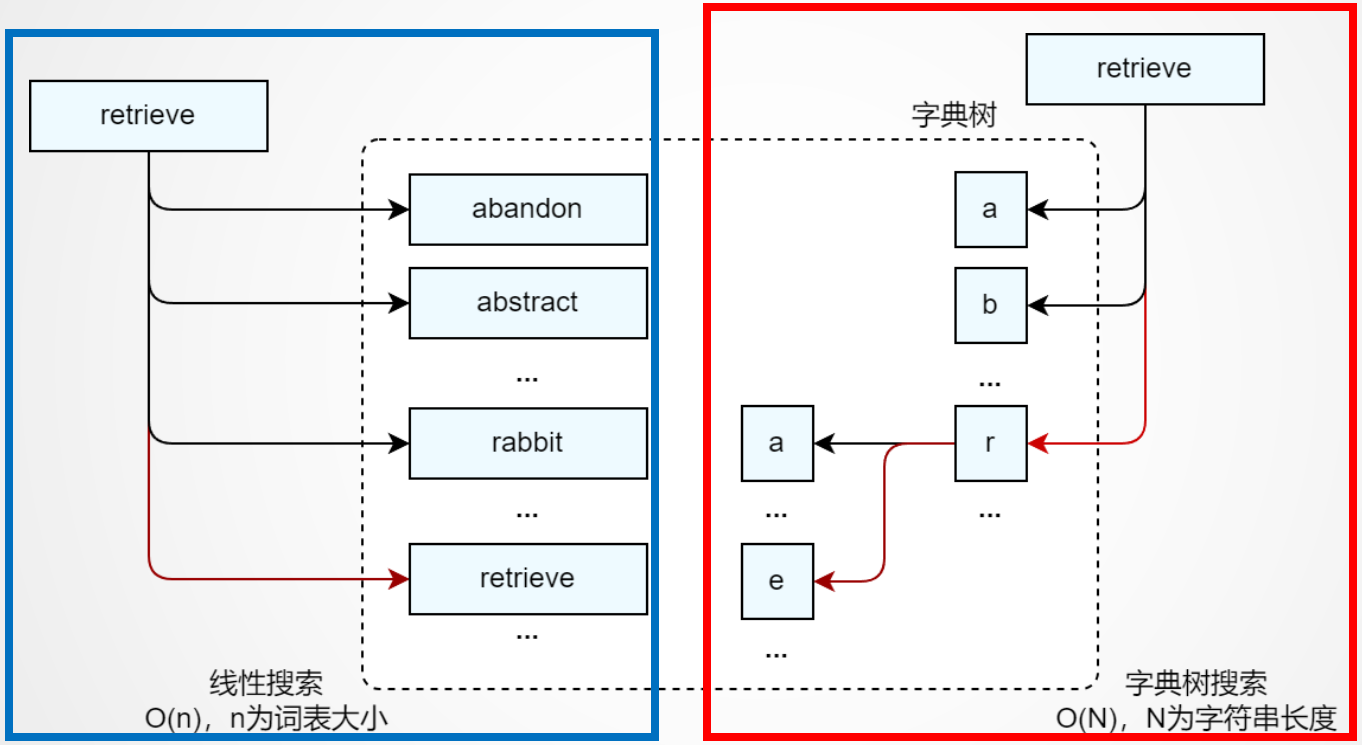
建立索引

将数据集合中的数据组织成高效的数据结构，使得查询时能直接访问相关数据，时间复杂度通常能降到 **O(log n)** 或更低，可以极大提升检索效率。

什么是索引（Index）

举例：在单词表中搜索单词“retrieve”，假设每个字母开头的单词数量为N。

线性搜索：
假设 “retireve”是以 r 开头的单词中第 i 个单词。
计算次数为
（17N+i）



字典树查找：
对每个位置上的字母建立树状索引。
计算次数为
（18+5+20+18+9+5+22+5）=102次

如果N=1000？约17000次 vs 102次

什么是索引（Index）- 标量索引

根据数据存储的类型，索引可以分为**标量索引**和**向量索引**。标量索引即在**传统关系型数据库**中针对**标量列值**（如数字、字符串、布尔值、日期等）建立的数据结构，以帮助数据库高效查找数据。

标量索引主要有以下四个作用：

·**显著提升查询速度**:帮助数据库直接定位到数据文件中的目标位置，从而有效避免了全表扫描这种耗时的操作。

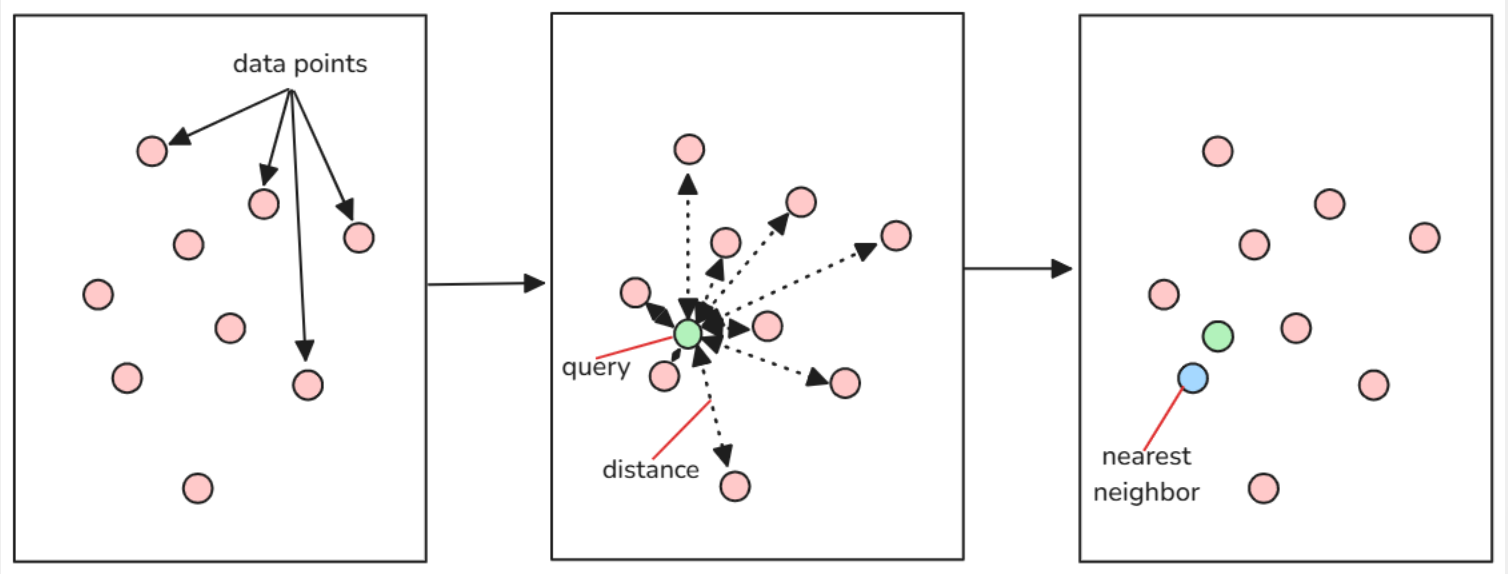
·**降低磁盘I/O次数**：使数据库无需对整个数据集进行读取，这极大地减轻了磁盘 I/O 的负担。

·**优化排序与分组操作**：使服务器避免进行排序和使用临时表的情况，进而提高查询效率。

·**提升复杂查询的性能**：能够对复杂的查询操作提供高效支持，比如在进行**表连接**操作，或者查找最大值、最小值等操作时。

什么是索引（Index）- 向量索引

向量索引是用于**加速高维向量数据相似度检索**的索引结构，它的目标是在大规模数据集中，快速找到与某个查询向量最相似（Top-K）的若干个向量。



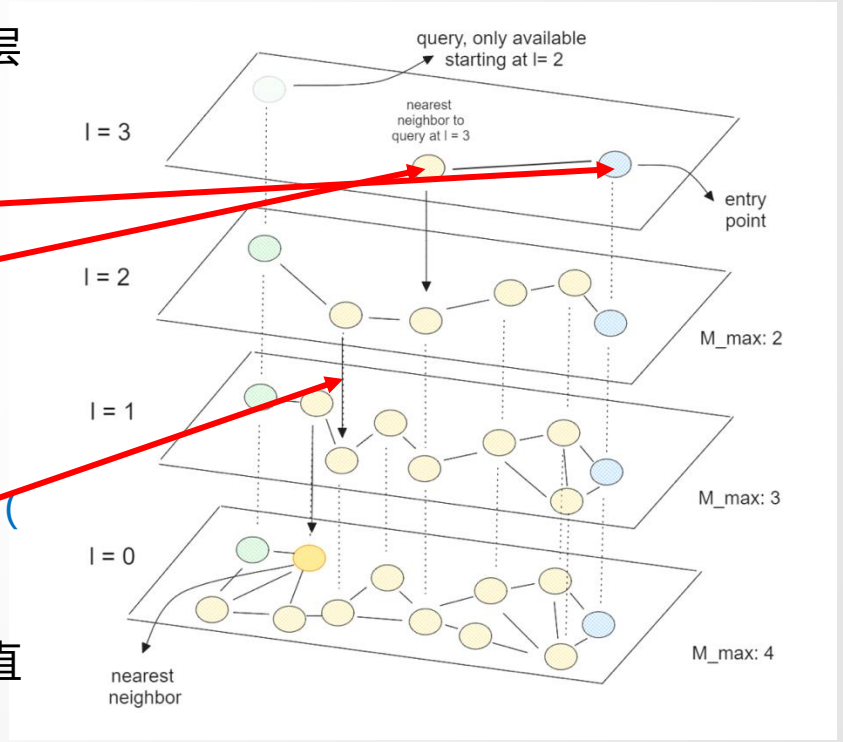
RAG系统中的向量检索可以描述为“给定查询向量 q，在向量集合 V 中找到距离（或相似度）最小（或最大）的若干个向量。”这一过程被称之为**最近邻搜索（Nearest Neighbor，NN）**，当检索参数中存在Top K时，即需要查找与目标数据最相似的前K个数据项时，上述搜搜过程便演化为**K最近邻搜索(K-Nearest Neighbor，K-NN)**

问题暴露：向量规模变大，计算量暴增

什么是索引 – 相似最近邻搜索

近似最近邻搜索（Approximate Nearest Neighbor，ANN）算法通过预先构建向量索引，使查询阶段只访问向量空间中的一个子集，用更少的距离计算换取**近似最优解**。这种**“用精度换速度”**的思想在处理大型高维数据集时尤其有用，可以大幅度减少搜索时间，同时仍然保持较高的结果质量。

- **多层图结构**：构建阶段将向量按距离随机分配到 L ~ 0 多层，小层包含所有向量，大层仅**抽样少量“代表点”**，形成金字塔。
- **顶层随机起点**：查询向量进入 **最高层**，任选一个节点作为起始。
- **贪婪近邻跳转**：在当前层反复执行：
  1.计算当前节点邻居与查询向量的距离；
  2.选择距离最近的邻居；
  3.若该邻居比当前节点更近，则跳转过去并继续，否则停止（达到局部最优）。
- **逐层向下**：把“局部最近节点”作为下一层的入口，层层递进，直到底层 0，找到nearest neighbor。



·HNSW的时间复杂度是O(logN)

HNSW（Hierarchical Navigable Small World，**层次化可导航小世界**）

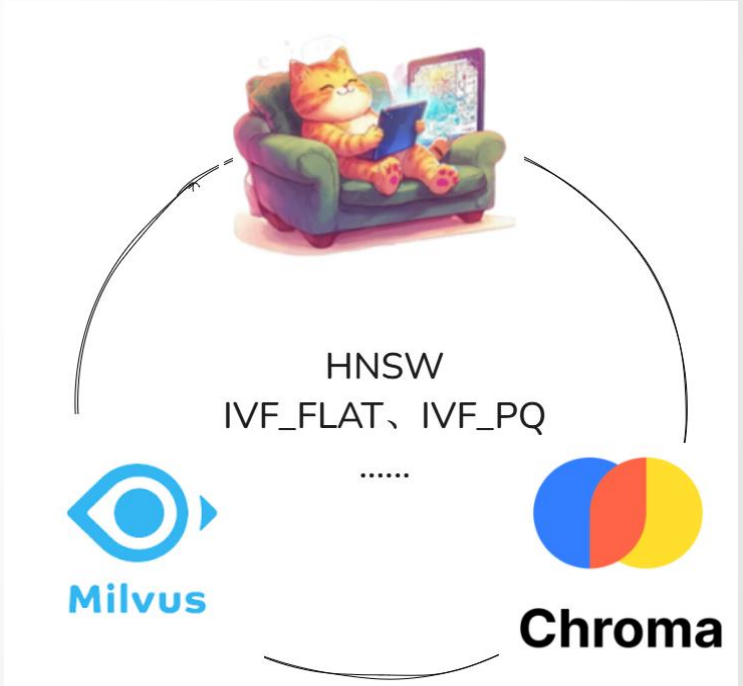
高效的向量数据库

对于开发者来说，**从零开始写一个向量索引**以实现高效检索绝对不是我们想要的。而向量索引恰恰是向量数据库的核心能力之一，我们可以简单认为：

向量数据库 = 向量索引 + 向量存储 + API支持 + 元数据管理

且对于LazyLLM而言，配置向量索引类型同样非常方便。

```
store_conf = {
    'type': 'milvus',
    'kwargs': {
        'uri': store_file,
        'index_kwargs': {
            'index_type': 'HNSW',
            'metric_type':'COSINE'
        }
    },
}
```



Llamaindex中的索引

从框架设计角度，LazyLLM各组件（Document、Store、Index、Similarity、Retriever等）层次清晰，分工明确。相比另一主流的应用开发框架Llamaindex有着更高的可读性和可扩展性。

索引类型	描述	节点组（Node Group）	相似度(Similarity)	索引(Index)
SummaryIndex	仅将节点存入列表	/	不执行相似度计算/LLM判断/cosine	/
VectorStoreIndex	绑定向量数据库的索引	/	cosine	vector index
DocumentSummaryIndex	建立索引过程中用大模型提前总结	summary	llm/cosine	/
KeywordTableIndex	使用大模型提取关键词	/	llm提取	linear

Llamaindex中一些典型的Index并没有单一的清晰概念，而是通过十几个参数实现了很多概念混杂的功能。

另外，这些index需先执行方法“as_retriever()”或“as_query_engine()”才可使用检索功能

工程优化：性能再提升

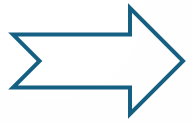
高性能RAG系统不仅依赖更好的技术选型和更优的检索算法，**工程层面的优化**也是决定系统能否稳定、高效落地的关键所在。

实际生产中的问题：

·如何更快海量文档解析、切片处理、向量化?

·如何更快的实现多路召回?

·系统请求量激增，如何优化系统响应?



系统工程优化：

·多任务串行 -> 并行

·IO密集型任务异步调度

...

并行和并发?

并行处理（Parallel Processing）是指在计算机系统中，多个任务或操作同时进行，以提高效率和缩短总耗时。通常情况下，会将一个大任务拆分成多个小任务，由多个处理器（或核心）同时执行。

并发处理(Concurrency Processing)：指一个时间段中有几个程序都处于已启动运行到运行完毕之间，且这几个程序都是在同一个CPU上运行，但任一个时刻点上只有一个程序在CPU上运行。

生活中：饭店中的厨师

串行：一名厨师先洗菜→再炒菜→再煮汤→最后蒸饭

并行：三个厨师分别负责炒菜、煮汤、蒸饭→同时进行，互不影响

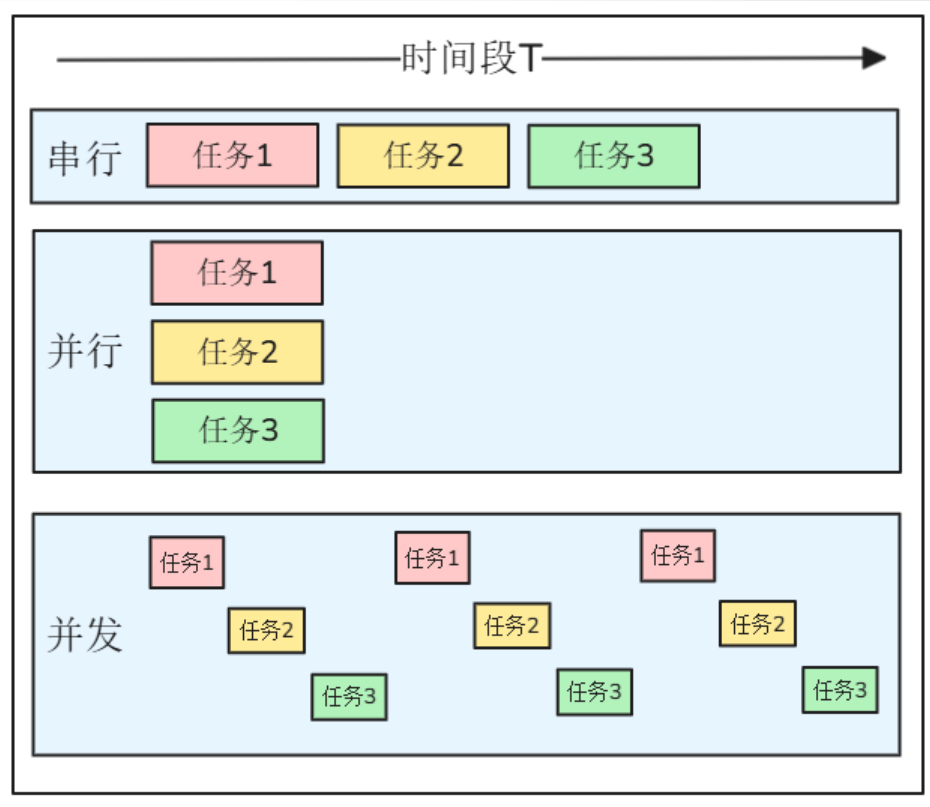
并发：一名厨师多个灶台，把菜丢下锅后等待3min，这3min去炖汤，期间回来翻炒、调味、装盘

程序中：文档解析

串行：海量文档逐个解析

并行：开多个线程，多个文档同时进行解析

并发：同一个线程中，让 文件I/O等待时间“空出来”给别的任务，期间不闲着，去做别的事情



Python中的并行

Python当中的并行主要用多进程和多线程实现。

多线程（Thread）

在同一进程内部并发执行多个线程；共享进程的内存与资源。

核心特点

·**线程切换开销小**，启动快

·受 CPython GIL 约束，仅适合 I/O 密集

·数据共享简单，但**易出现竞态**；需锁、队列等同步手段

Python实现：concurrent.futures中的ThreadPoolExecutor线程池

多进程（Process）：

同一程序在操作系统层面复制出多个独立进程；每个进程拥有独立内存空间和Python解释器。

核心特点：

·真正多核并行，不受 GIL 限制

·进程间通信需IPC（队列、管道、共享内存）

·创建与切换成本高，进程启动慢

Python实现：使用multiprocessing实现多进程

Python中的并行 – GIL(全局解释器锁)

Global Interpreter Lock(GIL，全局解释器锁)是一个防止解释器多线程并发执行机器码的一个全局互斥锁，其核心作用是确保同一时刻只有一个线程执行Python字节码。

GIL存在的必然性是以下情况导致的：

·历史遗留问题：CPython早期设计采用引用计数内存管理机制，未考虑多核时代的并发需求

·线程安全保障：防止多线程同时修改对象引用计数导致内存泄漏(如两个线程同时释放同一对象)

·生态依赖：大量Python扩展库基于GIL的线程安全假设开发，移除成本极高

绕过GIL的技术方案：

·多进程并行（ multiprocessing，使用Queue、Value、Array可灵活实现进程间的内存共享）

·异步编程模型（asyncio实现单线程高并发，适合I/O密集型场景）

·混合编程扩展（将核心算法用C/C++实现，如NumPy、Pandas等科学计算库）

注意：asyncio的功能是通过叫做event loop的python对象，调度每个task的代码的执行。当一个task被执行时，只有等task主动交回控制权，才会轮到其他task执行。（这一点和threading的并发显著不同）。因为要设计代码去实现子任务和主程序之前的控制器的给来给出，**所以“异步”的代码要比 threading 并发和 multiprocessing 更加复杂和麻烦。**

Python中的并行 – 线程的价值

在工程领域，我们通常将任务分为两大类：

计算密集型（CPU密集型）任务：

指在执行过程中主要涉及到大量计算和处理，如执行复杂的计算、算法或逻辑操作，而涉及到的IO操作相对较少的任务。

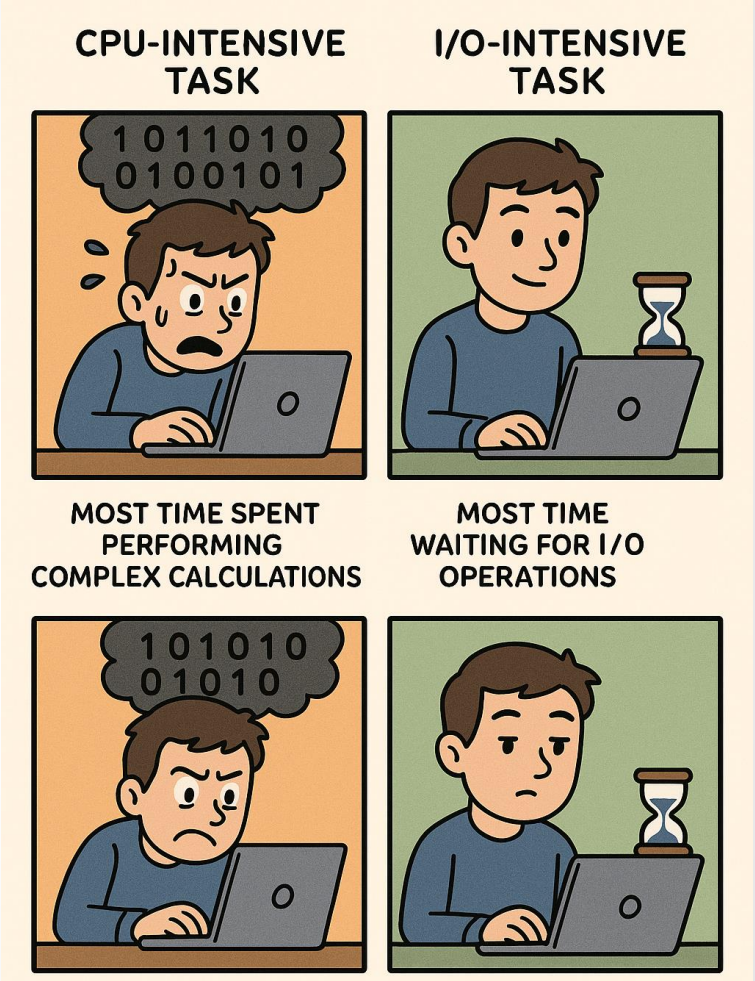
典型场景：**本地模型推理、数据分析**等。

IO密集型任务：

指在执行过程中通常需要与外部资源进行交互，大部分时间都花费在**等待输入输出（IO）操作完成**上，实际的计算量相对较小的任务。

典型场景：**读写文件、网络请求、数据库查询**等。

Python的多线程是为**IO密集型任务**设计的!



计算密集型 VS IO密集型

扩展学习（可选）- 常用标量索引结构

常用的标量索引结构有B树/B+树、哈希表和跳表。

B 树是一种多路平衡查找树，通常用于关系型数据库、文件系统等需要基于磁盘块（page）的高效检索场景。

- 多叉结构：

  m阶表示树中非叶节点的子节点数最多可拥有 m 个。每个非根、非叶节点至少有ceil(m/2)个子节点。

  枝节点的数据（关键词）数量应在ceil(m/2)-1和m – 1之间

- 层次平衡：

  根节点到任意叶节点的路径长度相同，保证查询、插入、删除操作的时间复杂度始终为 O(logN)。

  关键字在节点内按升序排列，子树数量 = 关键字数量 + 1。

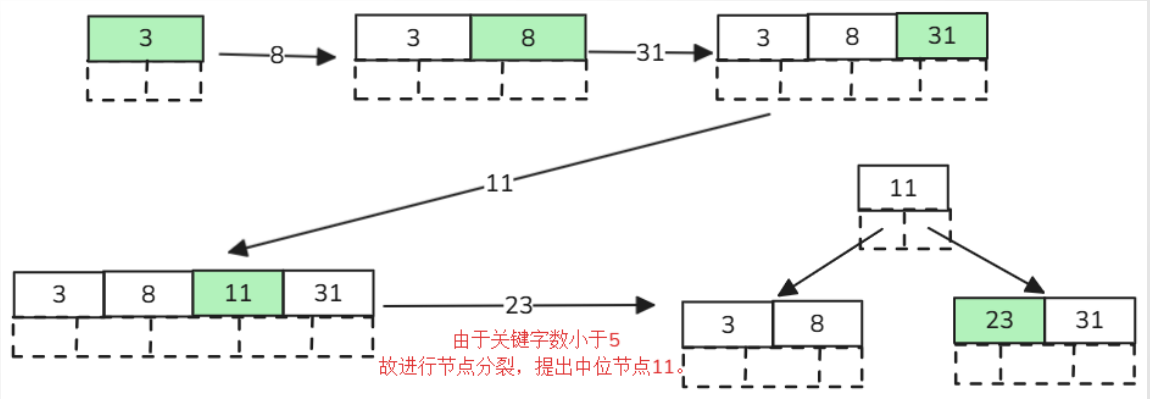
B树相对平衡二叉树，在节点空间利用率进行了改进，每个节点存储更多数据，减少了树的高度，提升了查询性能。

B树构建：

定义5阶树，插入数据3、8、31、11、23...

·节点拆分规则：m=5，关键字数必须小于等于5-1=4

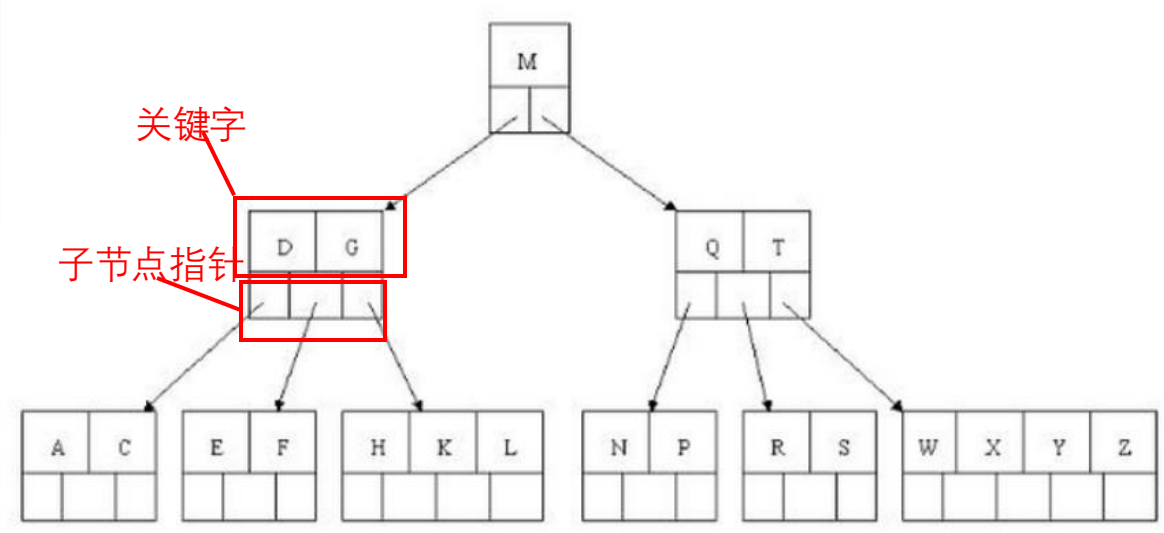
·排序规则：节点比较 - 左小右大



扩展学习（可选） - 常用标量索引结构

寻找字母“E”：

1. 获取根节点比较，根据二分法左小右大，E<M,所以前往左边子节点查询；
2. 拿到关键字D、G，因D<E<G，所以前往中间子节点查询；
3. 拿到E、F，E=E，所以直接返回关键字和指针信息。



B+树在B树的基础上做了以下改进：

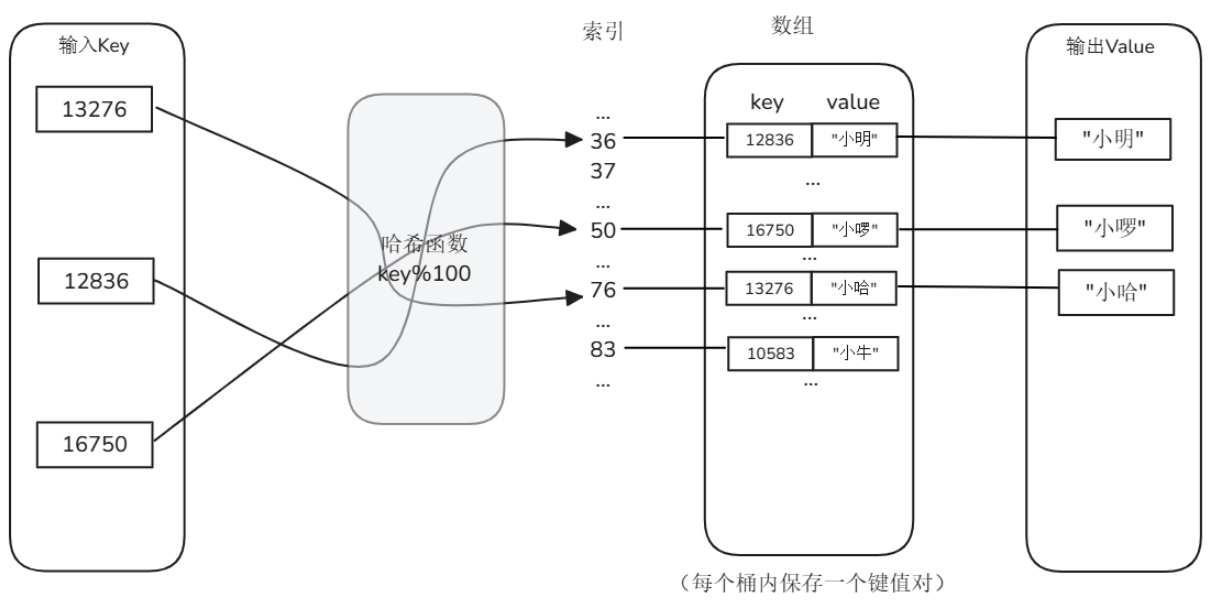
- 非叶子节点不保存具体数据，只保存关键字索引——保证每次数据查询次数一致，提升查询稳定性。
- 叶子节点关键字从小到大有序排列，左边结尾数据会保存右边节点开始数据的指针，对数据排序有着更好的支持。
- 非叶子节点的子节点数=关键字数

扩展学习（可选）- 常用标量索引结构

哈希表通过哈希函数h(key)**把键映射到槽位**，用链表或开放寻址解决碰撞。它放弃有序性，换取**几乎常数时间的精确匹配。**

特性：

- 平均 O(1) 查询/插入：只要负载因子受控，速度与数据量无关。
- 实现简单、并发友好：锁分段或无锁哈希均易于扩展。
- 不支持排序与范围检索：适合主键、ID、 UUID 等精确匹配场景。



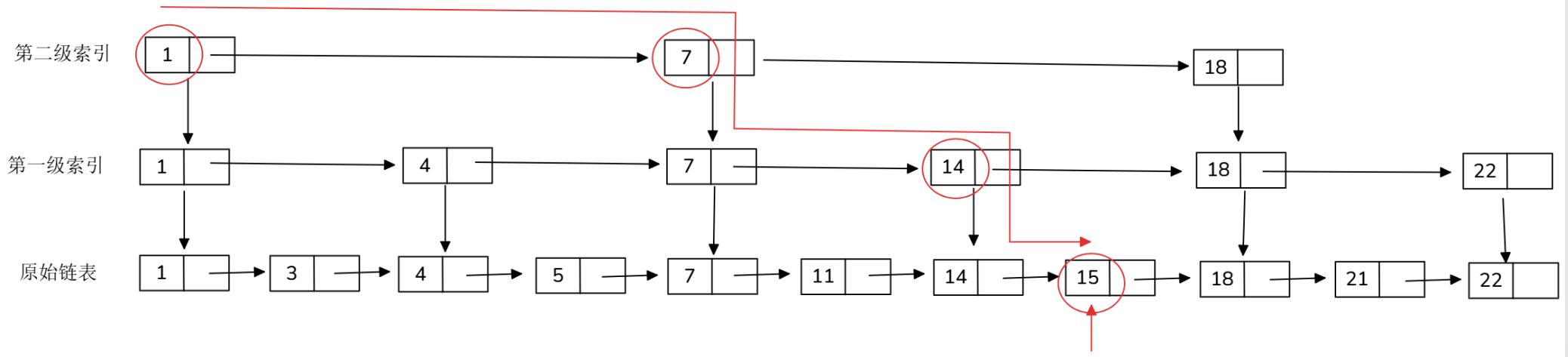
哈希表流程，学号姓名映射

扩展学习（可选）- 常用标量索引结构

跳表由多级有序链表组成，底层包含全部节点；从底层往上，每层按概率 p（常取 0.25-0.5）抽样上一层“跳点”。这种概率平衡让跳表在**无需复杂重平衡**的情况下，也能获得与平衡树相近的性能。

特性：

- 平均O(log N) 查询、插入、删除：路径期望长度与节点数对数相关。
- 实现简单、更新局部：只需调整邻接指针，非常适合高并发写入。
- 天然有序：支持范围扫描、按rank定位。



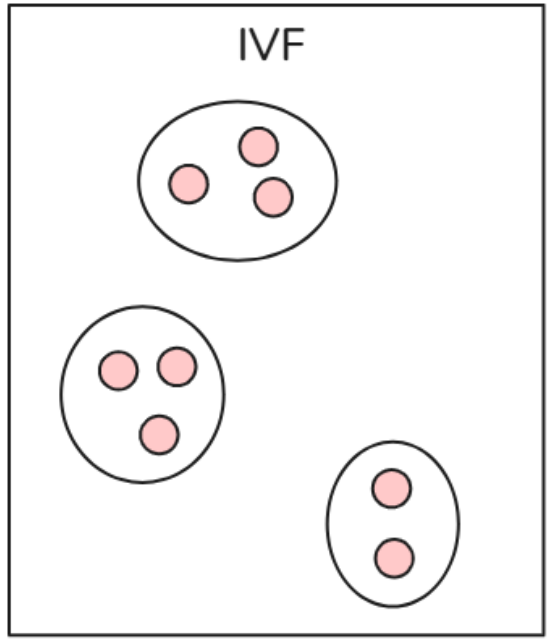
构建二级索引跳表，检索15仅需要遍历4个节点

扩展学习（可选）- 常用向量索引结构

目前向量索引基本都采用 ANN方法，常用的向量索引结构有IVF、PQ、HNSW等。

IVF是向量索引中最基本的索引技术。

- 使用K-means等聚类技术将**整个数据分成多个簇**，数据库中的每个向量都分配给特定的簇。
- IVF**仅在相关簇内部进行暴力搜索**，从而提高搜索速度，减少查询时间。
- 当出现新的查询时，系统不会遍历整个数据集，相反，它会识别最近或最相似的簇，并在这些簇中搜索特定的文档。

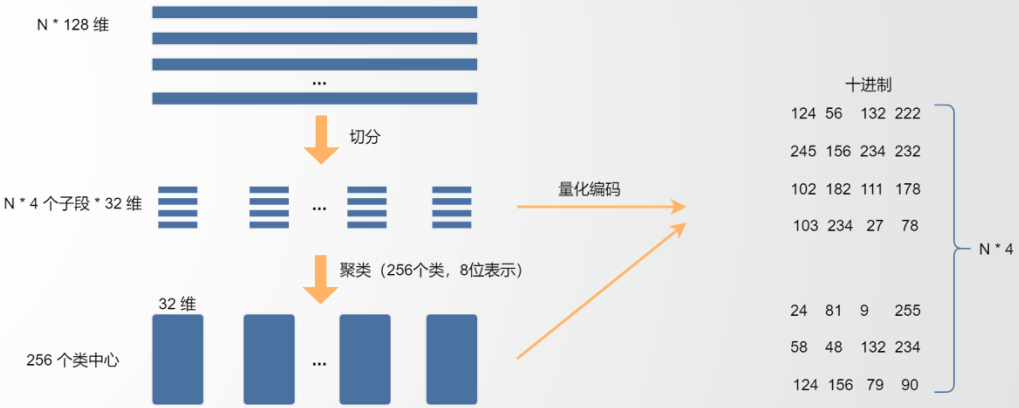


扩展学习（可选）- 常用向量索引结构

乘积量化(Product Quantization，PQ)是一种高效的向量压缩和最近邻搜索（ANN）方法，特别适用于高维数据。它通过**将高维向量分解为多个子空间**，并分别对每个子空间进行量化，从而减少存储开销，并加速最近邻搜索。

PQ的核心阶段分为：训练、量化、查询

- 训练：将高维向量划分为M段，对训练集中的每段进行K-means聚类，假设K=256，则每段都有256个中心向量。
- 量化，有了K个聚类中心向量，便可对他们进行编号，例如ClusterID=0~255。
- 查询：
    1. 将查询向量按照相同规则分段，每段计算与该段K个聚类中心的距离，构成距离表（M*K）。
    2. 遍历候选向量，计算距离表中的距离之和。
    3. 选出最相似topk向量。



训练与量化

扩展学习（可选）- 常用向量索引结构

HNSW（Hierarchical Neighbor Small World）是一种将图分为多个层次的图索引方法，是目前向量数据库中常用的一种索引。层次越高对应的节点越少，从高层次的点开始搜索，逐级搜索最终找到目标节点。

先前内容中已包含对于HNSW的介绍，其每一层中，算法利用“小世界网络”特性（即高聚类和短路径），通过贪心策略快速定位候选节点，**避免了全局遍历**。每层搜索的复杂度为常数时间，而层数为 O(log n)，因此整体复杂度为 O(log n)。因此 HNSW 适合大规模高维数据的快速近似最近邻搜索。

向量索引结构	特点与适用场景
IVF（Inverted File Index）	先用聚类将数据划分成多个“桶”，查询时只在几个相关桶内做局部搜索，适合大规模数据（如百万级）
PQ（Product Quantization）	将高维向量压缩为低维离散编码，提高存储密度，适合内存受限的场景
HNSW（Hierarchical Navigable Small World）	基于图结构，构建多个分层的近邻图，适合中高维、查询速度极快，是目前最主流的 ANN 算法之一

扩展学习（可选）- Why Milvus?

- 检索性能强

  支持 HNSW、IVF_FLAT、IVF_PQ 等多种高效向量索引，且内置分层缓存机制，检索效率高。

- 优秀的架构设计
  - Milvus 的云原生和高度解耦的系统架构确保了系统可以随着数据的增长而不断扩展。
  - 灵活支持Lite、Standalone、Distributed等多种部署模式，兼容各种场景下的部署需求。
  - 分布式的支持允许Milvus支持亿级别的向量存储，极高的提升了系统的稳定性和可扩展性。
- 功能丰富

  Milvus 引入了标量字段索引，可以有效组织标量字段数据，并结合倒排索引、自动索引等技术，极大提升“标量过滤 + 向量检索”这种复杂检索场景的查询效率。

目录

大模型推理速度评测指标

大模型推理服务目标是首token输出尽可能快、吞吐量尽可能高以及每个输出token的时间尽可能短，即模型服务能够尽可能快地为尽可能多地为用户生成文本。

- TTFT（Time to First Token，首字延迟）：

  从向模型输入提示词开始到模型生成第一个token所需时间。

- TPOT（Time Per Output Time）：

  模型在输出阶段（Decode）每个输出token的延时。

- Throughput:

  吞吐量，针对模型服务，模型服务在单位时间内能处理的token数量。

- BS（Batch Size）：

  针对模型服务，即服务合并计算用户请求的数量（提高吞吐量的方法一般是提高BS，但是提高BS会一定程度上影响每个用户的时延）。

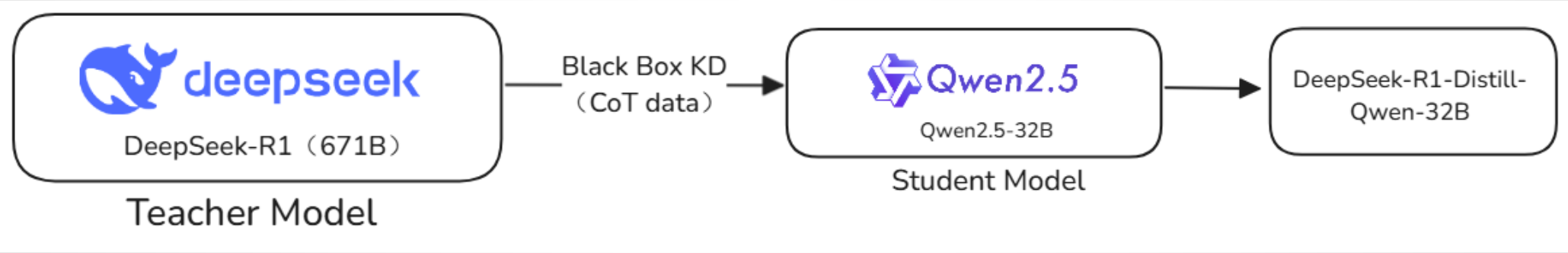
大模型推理速度的影响因素

1. 计算量
    - 模型大小（参数量） -> 蒸馏
    - 使用低精度数据
    - 量化
2. 计算性能
    - 芯片计算能力
    - 算子实现情况
    - 选取合适的并行策略，优化卡间互联
    - 选用一个好的推理框架

模型蒸馏

模型蒸馏（Knowledge Distillation，KD）作为一种模型压缩技术，可以在不显著牺牲模型性能的情况下，将大型复杂模型的知识转移到较小的模型中，使得推理更高效、更易于部署。相比于直接训练一个小模型，学生模型在蒸馏过程中可以学习到教师模型的知识结构，使其泛化能力更强。

核心思想：用一个能力很强、参数量很大的模型作为**教师模型**，指导一个参数量较小的**学生模型**，使其能够在较低计算成本下达到接近教师模型的性能。



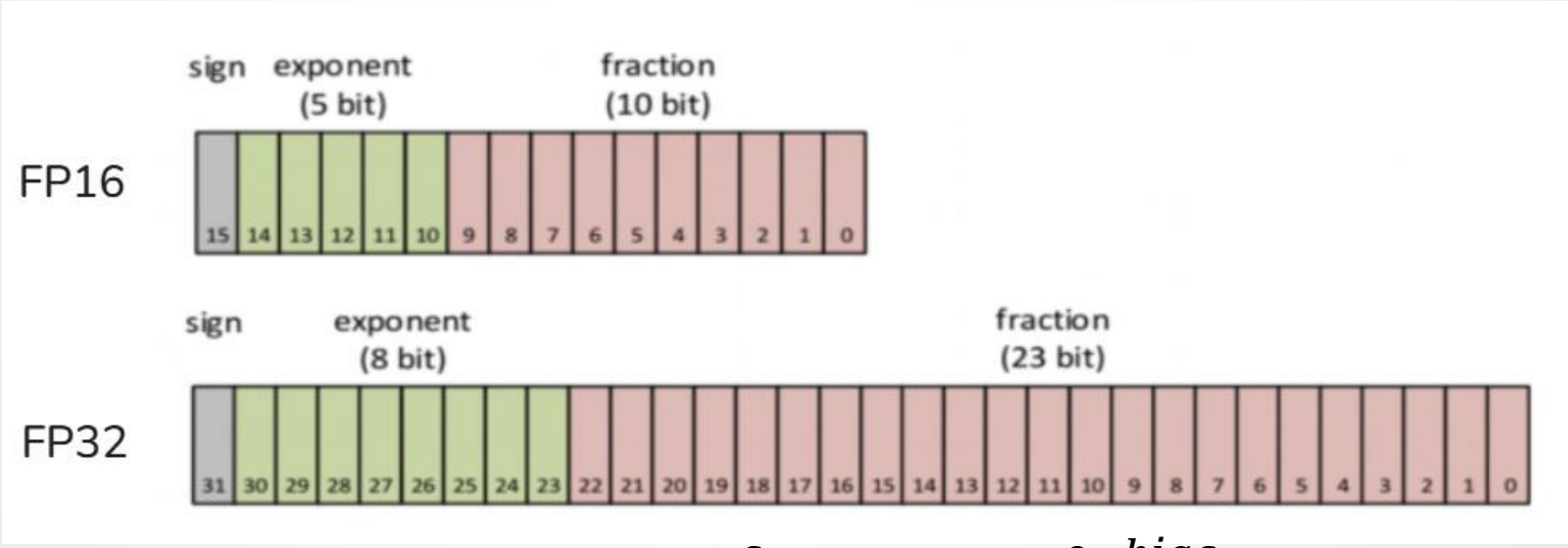
·黑盒知识蒸馏：仅能获取教师模型的输出，无法获取教师模型**内部信息**。

·白盒知识蒸馏：教师模型的**架构和权重**是完全可访问的。这种透明度使学生模型不仅可以学习教师模型的输出，还可以学习其内部表示和决策过程。

关于模型蒸馏的更多细节本节不再赘述，若感兴趣可前往《选修4：Deepseek + RAG》进行回顾学习。

模型参数精度

大模型的训练和推理经常涉及到精度的概念，精度高更准确，但也会带来更高的计算和存储成本。较低的精度会降低计算精度，但可以提高计算效率和性能。所以多种不同精度，可以让你在不同情况下选择最适合的一种。浮点数精度（Floating Point，FP）：最原始的精度，由符号位（sign，正/负）、指数位（exponent，表示整数部分范围）、尾数位（fraction，表示小数部分）组成，常见精度由FP64、FP32、FP16（FP8和FP4不是IEEE的标准格式）



$$value = (-1)^s \times 1.\mathrm{f} \times 2^{\mathrm{e}-bias}$$

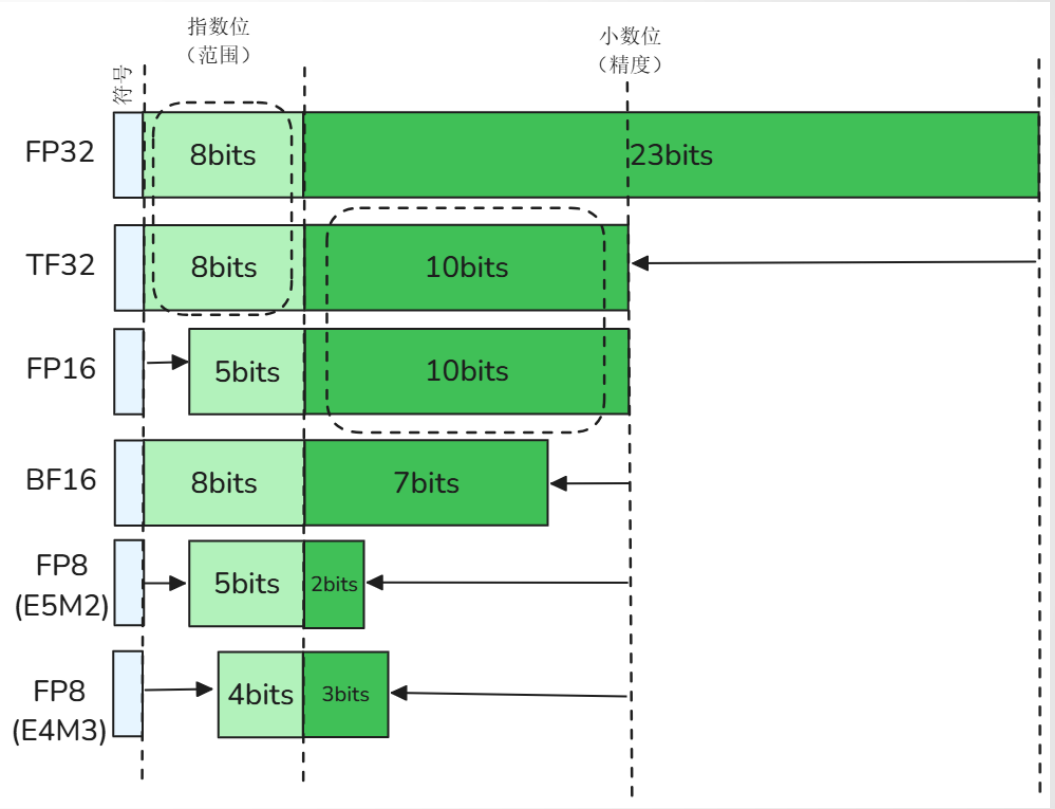
$$bias = 2^{bitlen(e)-1} - 1$$

TF32与BF16

TF32（Tensor Float 32）是英伟达针对机器学习设计的一种特殊的数值类型，用于替代FP32。首次在A100 GPU中支持。其由1个符号位，8位指数位（对齐FP32）和10位小数位（对齐FP16）组成，实际只有19位。在性能、范围和精度上实现了平衡。

BF16（Brain Float 16）由Google Brain提出，也是为了机器学习而设计。由1个符号位，8位指数位（和FP32一致）和**7位小数位（低于FP16）**组成。所以精度低于FP16，但是表示范围和FP32一致，和FP32之间很容易转换。

FP8是H100及之后的显卡特有的一种数据表示方式，目前国产卡几乎都不具备该格式，有E4M3和E5M2两种常见的变种格式。



TF32与BF16

Python中查看是否支持：

TF32：

```
import torch
//是否支持tf32
torch.backends.cuda.matmul.allow_tf32
//是否允许tf32，在PyTorch1.12及更高版本中默认为False
torch.backends.cudnn.allow_tf32
```

BF16：

```
import transformers
transformers.utils.import_utils.is_torch_bf16_gpu_available()
```

模型量化技术（Quantization）

通过对模型参数进行压缩和量化，从而降低模型的存储和计算复杂度，最终达到节省显存、加速计算、降低通讯量的目的。

·核心思想：

降低模型参数的表示精度（如FP32->INT8），在几乎不改模型结构的前提下，大幅减小存储与算力开销。

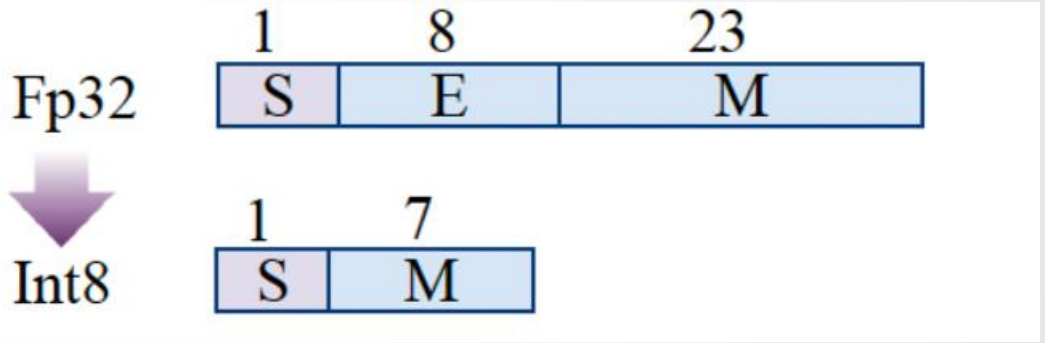
·量化程度：

全量化：所有权重和激活值都量化

部分量化：仅量化权重或激活

·常见量化精度：INT8、INT4

·常用量化技术：GPTQ、AWQ



量化前：FP32-4字节
量化后：INT8-仅1字节

模型量化技术（Quantization）

在模型开源社区中，开发者可以轻松获取大量优秀的量化模型。

Tensor Core

Tensor 核心（Tensor Core）是 NVIDIA GPU在A100及之后才有的硬件加速单元，相比传统 CUDA 核心（CUDA Core），Tensor 核心可以在一次运算中并行处理更多数据，从而大幅提升**矩阵乘法**（Matrix Multiplication, GEMM） 速度，使深度学习训练更高效。

	CUDA Core	Tensor Core
主要用途	通用计算（GPGPU）	AI训练&推理
支持数据类型	FP32、INT32 等	FP16、TF32、INT8、INT4、FP64 等
性能特点	通用性强，支持各种并行计算任务	专用硬件加速，最高提升16倍，吞吐量更高
典型应用	图形渲染、科学计算、高性能计算	深度学习模型训练与推理

CUDA 加速

CUDA（Compute Unified Device Architecture）是 NVIDIA 提供的一种并行计算平台和编程模型，它允许开发者使用类似 C/C++ 的语言在 GPU 上进行通用计算（即 GPGPU：General-Purpose computing on Graphics Processing Units）

·并行计算：GPU 有上千个核心，CUDA 把计算任务分发给大量线程，使大量计算任务瞬间完成

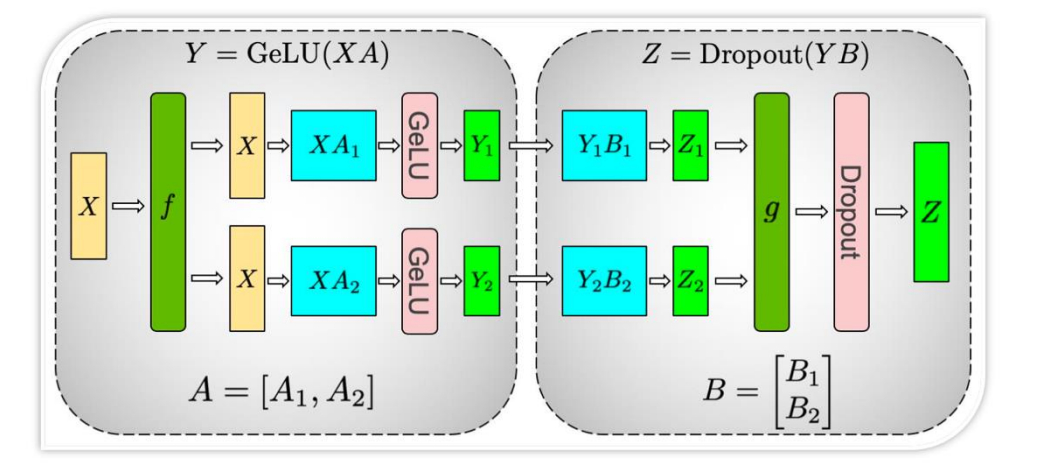
·高效内存管理：CUDA 提供 shared memory、global memory 等机制，让内存访问尽量并行、高速

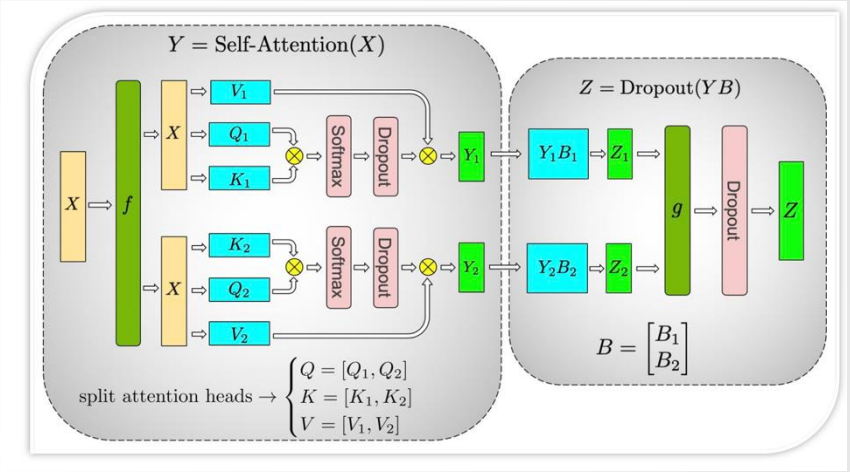
·张量操作优化：很多深度学习库（如 PyTorch）底层用 CUDA 实现了高性能的 matmul、conv、norm 等操作

·自动调用 GPU：框架在支持 CUDA 的情况下，可以自动将数据搬到 GPU，用CUDA核心计算，然后返回结果

·支持混合精度计算（Tensor Core）：在新一代 GPU 上，CUDA 能调用 Tensor Core，用 float16 或 int8 等低精度加速计算，几乎不损失模型精度

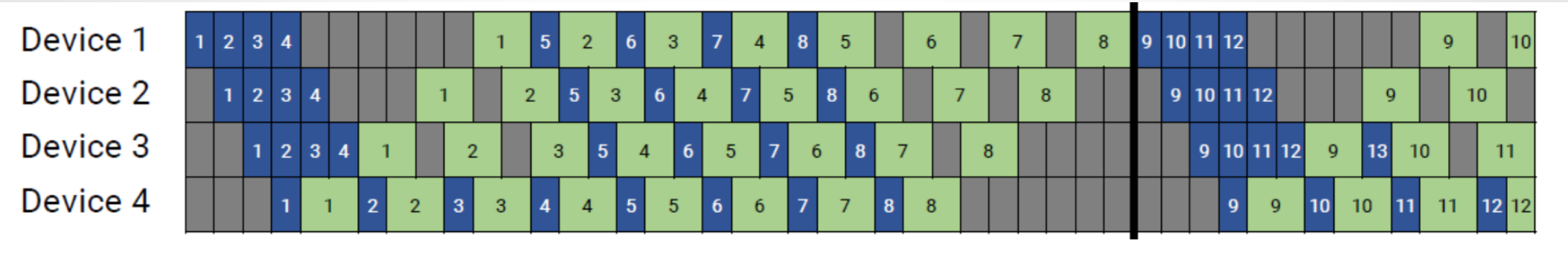
并行策略 – TP & PP





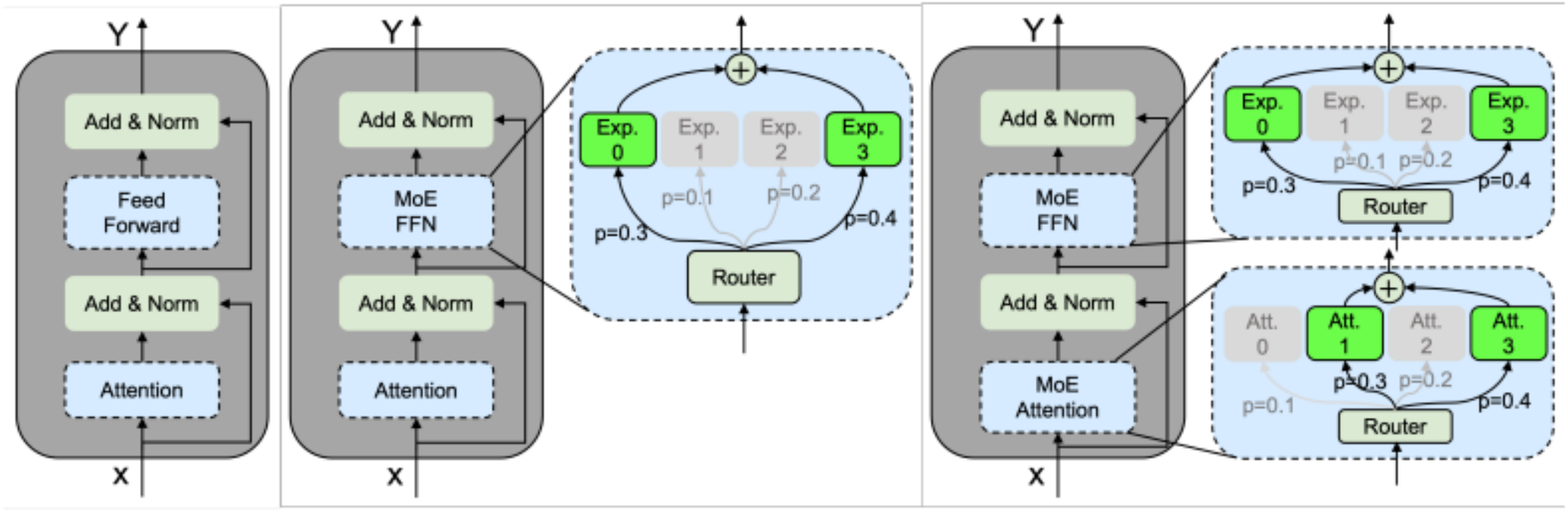
Tensor并行之MLP

Tensor并行之Self Attention



Pipeline Parallel

稀疏模型结构 – MoE



(a) Dense Layer (b) MoE FFN Layer (c) MoE Attention+FFN Layer

Fig. 2. Architectural comparison of dense layer with MoE layers: (a) conventional dense transformer layer, (b) transformer layer with MoE-based feed-forward network, and (c) transformer layer with both MoE-based attention and feed-forward networks.

更好的推理框架

在模型不变的情况下，通过选择更好的推理框架，也可以提升大模型的推理性能。当前主流的模型推理框架有Ollama、LightLLM、vLLM以及LMDeploy等等。

Ollama

·预打包模型：内置了LLaMA、Mistral等多种模型。

·**硬件适配性强**：针对日常使用的硬件进行了CPU和GPU推理优化，无论是**MacBook、PC**还是边缘设备，都能流畅运行AI模型。

·操作便捷：提供简洁的API和CLI，只需简单配置，就能快速启动大语言模型。

LightLLM

·轻量级设计：占用资源少，易于部署和扩展，方便快速上手进行本地部署和定制修改。

·高性能：通过多种优化技术如三进程异步协作、Token Attention等，实现高速推理，获得更高的吞吐量。

·易用性：提供 Docker 容器和详细文档，简化使用流程，降低使用门槛。

vLLM

·**PagedAttention**技术：突破传统KV缓存机制，实现显存分页管理，支持超长序列生成（如10万token对话）。

·吞吐量领先：在 A100 GPU上可达传统框架3倍以上吞吐量，支持动态批处理。

·生态兼容性：原生支持HuggingFace模型格式，兼容PyTorch生态。

LMDeploy

·Turbomind引擎：采用**异步流水线**并行，延迟降低至50ms级别

·量化部署工具链：支持W4A16量化，**模型体积压缩4倍**

·动态批处理：智能合并不同长度请求，GPU利用率达90%+

更好的推理框架

LazyLLM已完成LightLLM、vLLM、LMDeploy三种框架的完美适配，开发者可以根据自己的实际需求，灵活选择推理框架，以实现最适合自己的模型推理体验。

	Ollama	LightLLM	vLLM	LMDeploy
核心优势	个人PC运行LLM	轻量化，极简部署	高并发处理	国产高性能框架
适用场景	个人电脑/较小模型	个人调试/原型开发	企业级服务部署	国产化部署方案
硬件要求	MacBook、PC、边缘设备可以跑	中端显卡也能跑	需NVIDIA高性能显卡	中端显卡也能跑
开源协议	MIT	Apache 2.0	Apache 2.0	Apache 2.0

端侧推理框架

端侧推理框架	主要特点	典型应用芯片
MLC LLM	多后端编译（WebGPU、Metal、CUDA、Vulkan），一套代码多平台跑	手机（iOS/Android）、浏览器、PC
llama.cpp	用纯C/C++写的推理器，极致小巧，适配CPU/Metal/AVX	PC CPU、Mac (M系列)、移动端
GGML/GGUF	格式 + 量化推理库，配合 llama.cpp 使用	CPU优先（包括嵌入式CPU）
Core ML Tools	苹果官方推理框架，专为Apple Silicon（M系列、A系列芯片）优化	iPhone、MacBook、iPad
OpenVINO	Intel出品，优化x86 CPU / VPU / Movidius NPU推理	边缘计算、工控设备
TensorFlow Lite (TFLite)	TensorFlow的端侧轻量版本，支持量化和硬件加速	手机、微型设备（树莓派等）
ONNX Runtime Mobile	ONNX推出的轻量版runtime，支持移动端加速	Android、iOS

感谢聆听

Thanks for Listening