QCon
全球软件开发大会

InfoQ 极客传媒

破解大模型推理成本难题
YRCache 以存代算加速实践

张文涛
焱融科技CTO

目录

01

KVCache 的技术背景和挑战

推理优化的两个核心方向

模型压缩

1 **量化**

将模型权重和激活值的计算精度从FP32/FP16降低至INT8乃至INT4，用来降低内存占用，减少数据带宽和提升计算速度

2 **剪枝**

剪枝：移除模型中冗余或不重要的权重、神经元或层

3 **蒸馏**

用一个大型“教师模型”来训练一个精简的“学生模型”，以更低的成本达到相近的性能

执行效率

4 **KVCache**

缓存注意力机制中的Key和Value，避免在生成每个新token时重复计算历史上下文，减少计算量，提升系统总吞吐

5 **投机解码**

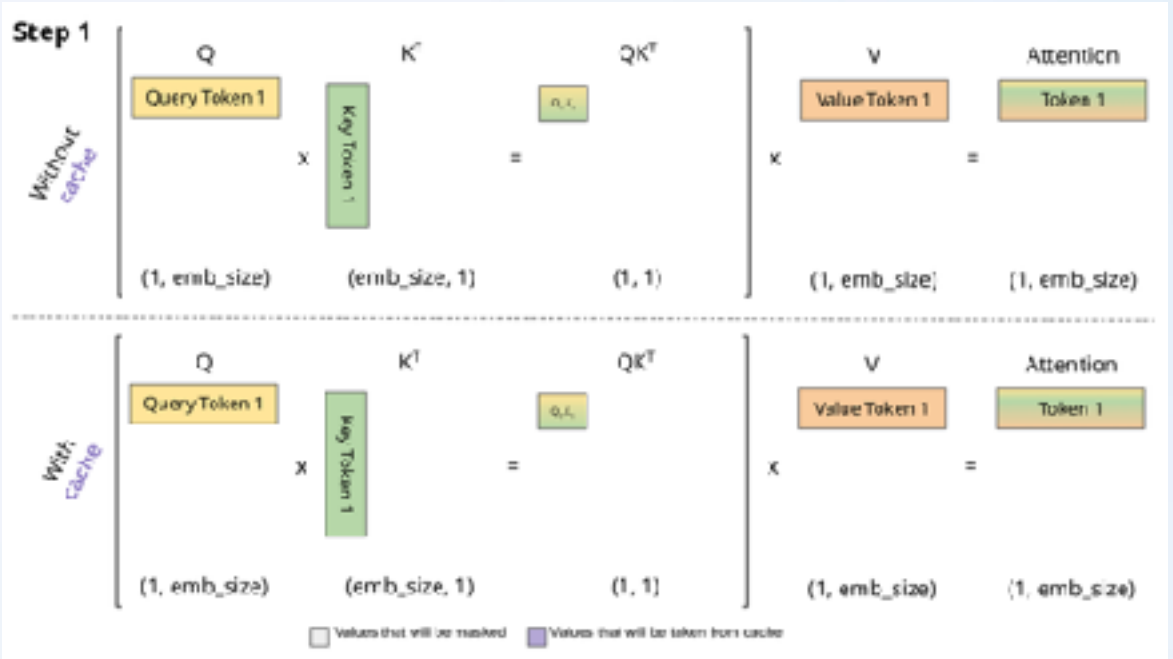
使用一个小型、快速的“草稿模型”预先生成一段token序列，再用大型“目标模型”一次性验证，将大模型零散的、受内存带宽限制的计算，转化为批处理式的计算模式，提升系统总吞吐

6 **连续批处理**

将不同用户的请求组合成批次进行处理，并动态调整批次内的任务，最大化GPU的利用率，提升系统总吞吐

KVCache 的原理和价值｜避免重复计算，提升计算效率

- Token 的注意力仅取决于其前面的 token，因此在每个生成步骤中，我们都需要重新计算相同的先前 token 的注意力，而实际上我们只是想计算新 token 的注意力
- KVCache 的作用就是通过缓存之前的 Key 向量和 Value向量，让我们可以专注于计算新 token 的注意力
- 采用 KVCache 缓存后，需要计算的矩阵就小得多，从而可以加快矩阵乘法的速度
- KVCache 的缺点就是需要更多的 GPU 显存空间



Source： https://medium.com/@joaolages/kv-caching-explained-276520203249

Prefix Cache | 优化 Prefill 阶段的计算效率

- 相同前缀请求的 KVCache 是完全相同的，没必要重新计算
- 对于 Agent/tools，有几 K 到几十 K 长度的共享系统 prompt，没必要重新计算
- 多轮对话，为了让大模型记住上下文信息，需要保留历史对话，随着对话轮次变多，重复计算就越多，KVCache 被重用的就越多， Prefix Cache的效果就越明显
- 通过以存代算，能够节省大量的计算资源，进而提升整体推理能力

Example 1: Shared system prompt

Request A

A chat between a curious user and an artificial intelligence assistant. The assistant gives helpful, detailed, and polite answers to the user's questions. User: Hello

Request B

A chat between a curious user and an artificial intelligence assistant. The assistant gives helpful, detailed, and polite answers to the user's questions. User: How are you?

Example 2: Multi-round conversation

Prompt (round 1)
Human: *What's AI?*

LLM Result (round 1)
LLM: *AI is technology that simulates human intelligence, like Siri or Google Maps.*

共享部分

Prompt (round 2)
Human: *What's AI?*
LLM: *AI is technology that simulates human intelligence, like Siri or Google Maps.*

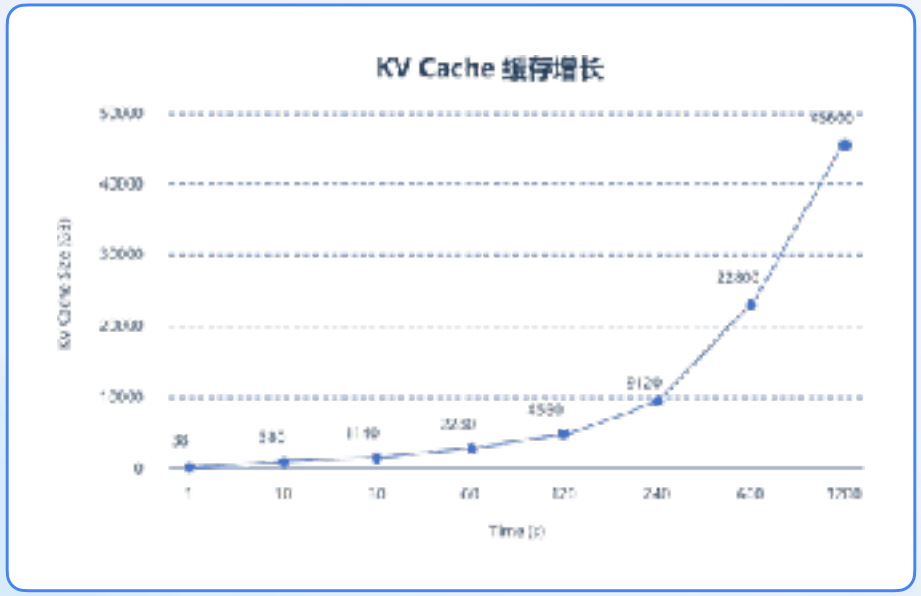
Human: *Cool, thanks!*

LLM Result (round 2)
LLM: *No problem!*

面临的挑战｜有限的显存空间 vs 爆炸式增长的容量

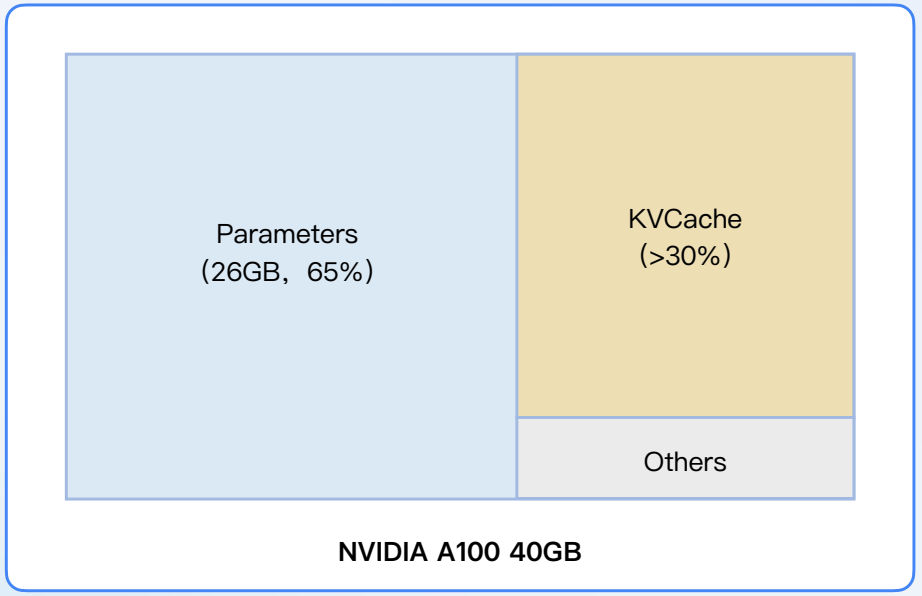
背景

该图展示了 LLaMa-65B 模型下 KVCache 缓存容量增长情况。以 8 张 A100 每秒平均处理 15.2K tokens 为基准，每处理 2K tokens 会生成 5GB 的 KVCache。随着时间推移，token 数量不断增加，进而带动 KVCache 容量同步提升



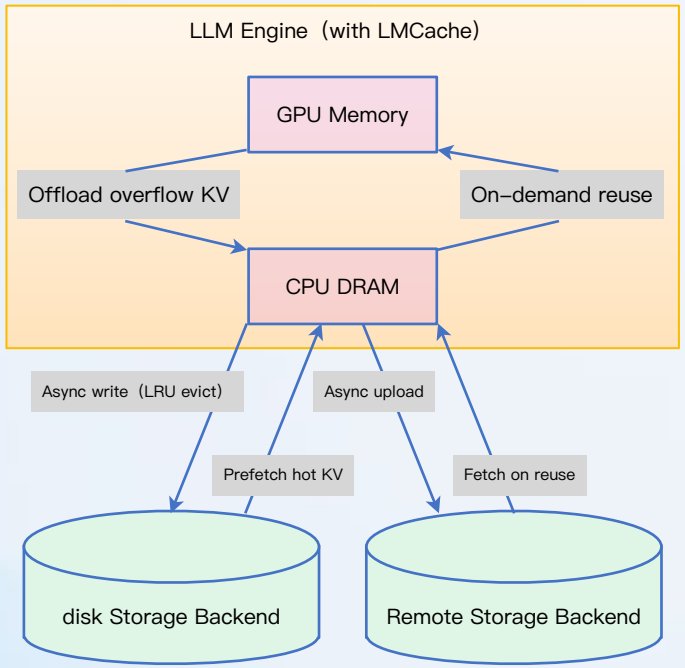
挑战

GPU 显存空间中，只有 **30%** 显存空间是用来存放 KVCache！空间的不足和数据的爆炸式增长形成了鲜明对比！**进而导致 KVCache 没办法被命中！**

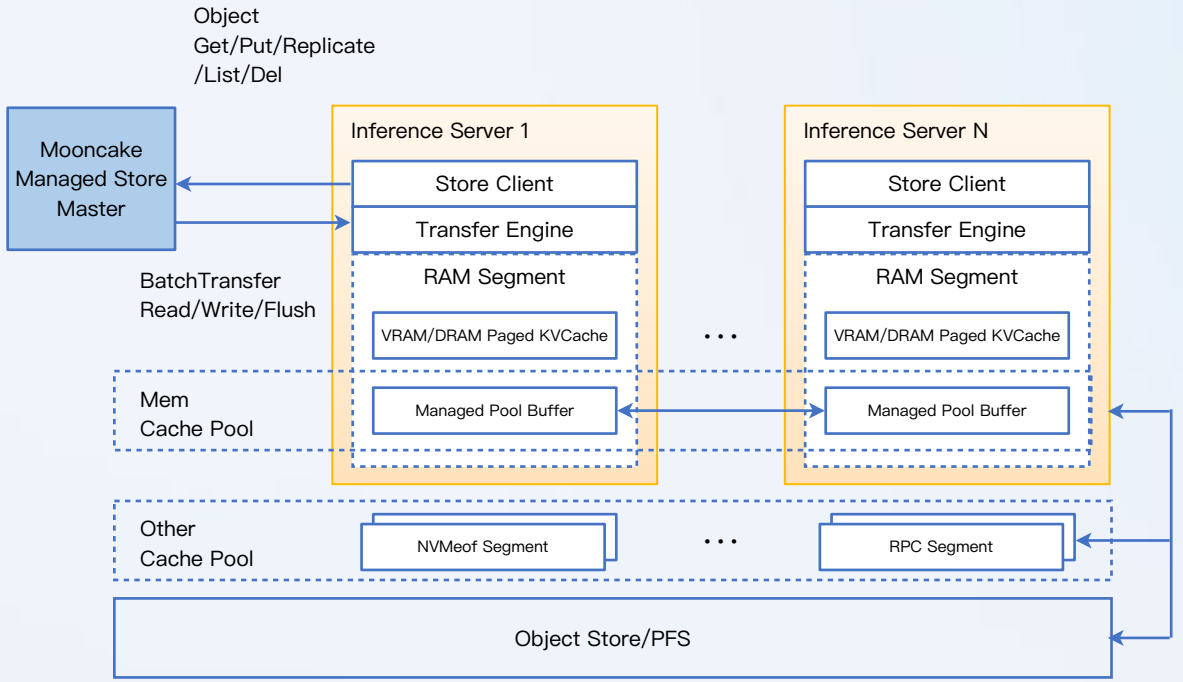


常见的 KVCache Offloading 方案

LMCache

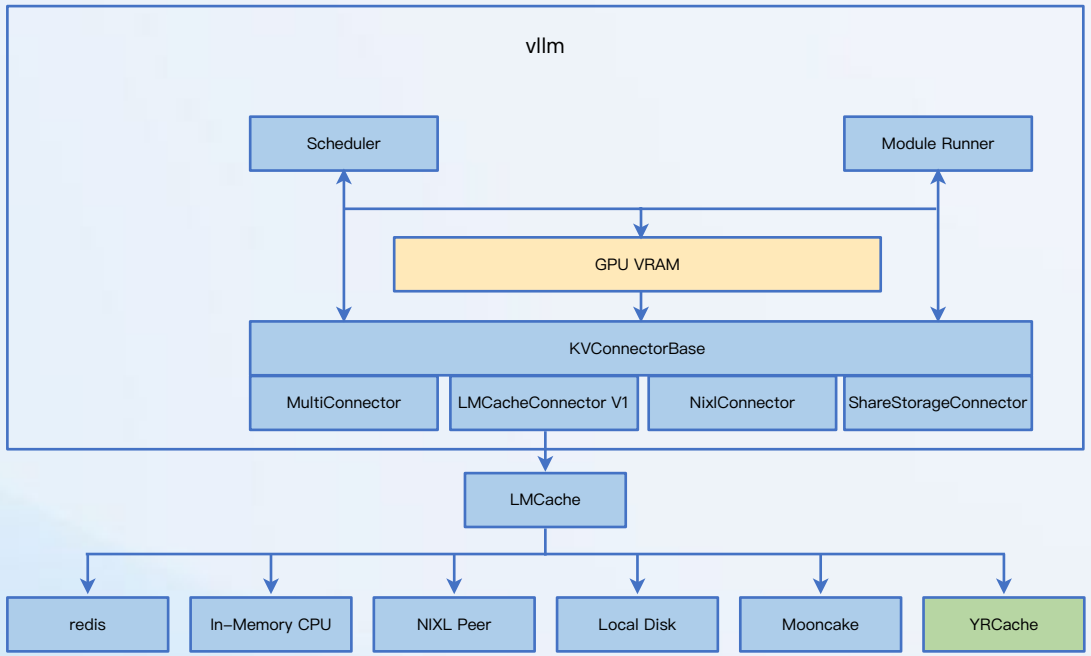


Mooncake Store

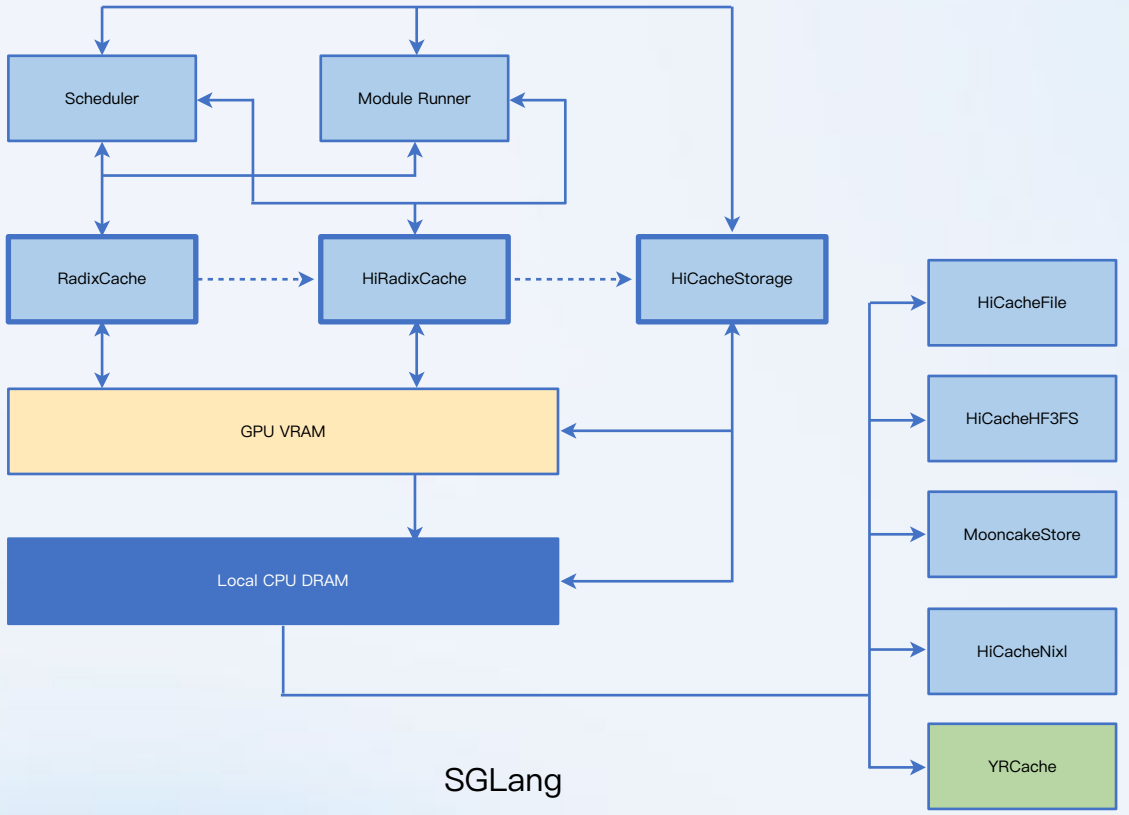


推理平台适配的挑战

vLLM推理平台适配



SGLang推理平台适配



KVCache Offloading 对 PD 分离架构的影响

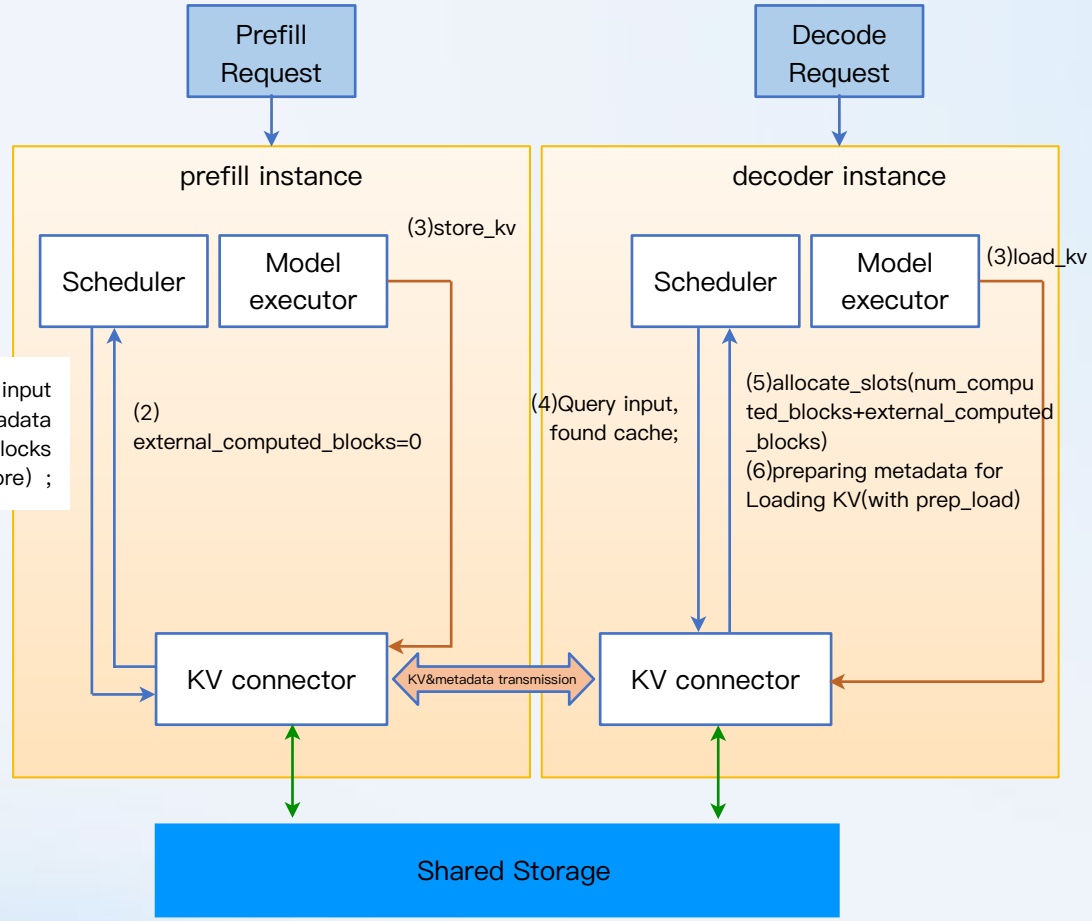
KVCache Offloading方案的优劣势对比

优势

- 减少 P 节点 KVCache 显存驻留时间，将 KVCache Offloading 即可释放显存空间，无需 P 节点等待 D 节点过来获取 KVCache
- 解耦，PD 不需要互相可见和通信，降低了节点间的耦合，便于独立扩展和故障恢复
- 容错性更好，无论是 P 还是 D 故障，KVCache 不会丢失

劣势

- 引入共享存储，额外增加了一次IO，延迟会增加
- 成本高，搭建和维护一个共享存储的成本和复杂度较高



KVCache Offloading 方案面临的技术挑战

缓存空间

常见的推理 GPU 卡只有几十 GB 的显存空间，而留给 KVCache 缓存的空间更小，如何扩展更大的缓存空间，并且不影响访问效率?

更大的缓存空间

缓存命中率

命中率是以存代算的最重要核心指标之一，只有缓存命中率提升才能极大降低重新计算的开销，如何才能有效提升缓存命中率?

更高的缓存命中率

访问效率

访问效率是以存代算的另一个重要的核心指标，只有访问效率足够高，比重新计算要快得多，才能体现出以存代算的价值!

更快的访问速度

平台适配

vLLM、SGLang、TGI 等推理平台众多，如何才能让平台适配更简单，甚至开箱即用?

更简单的平台适配

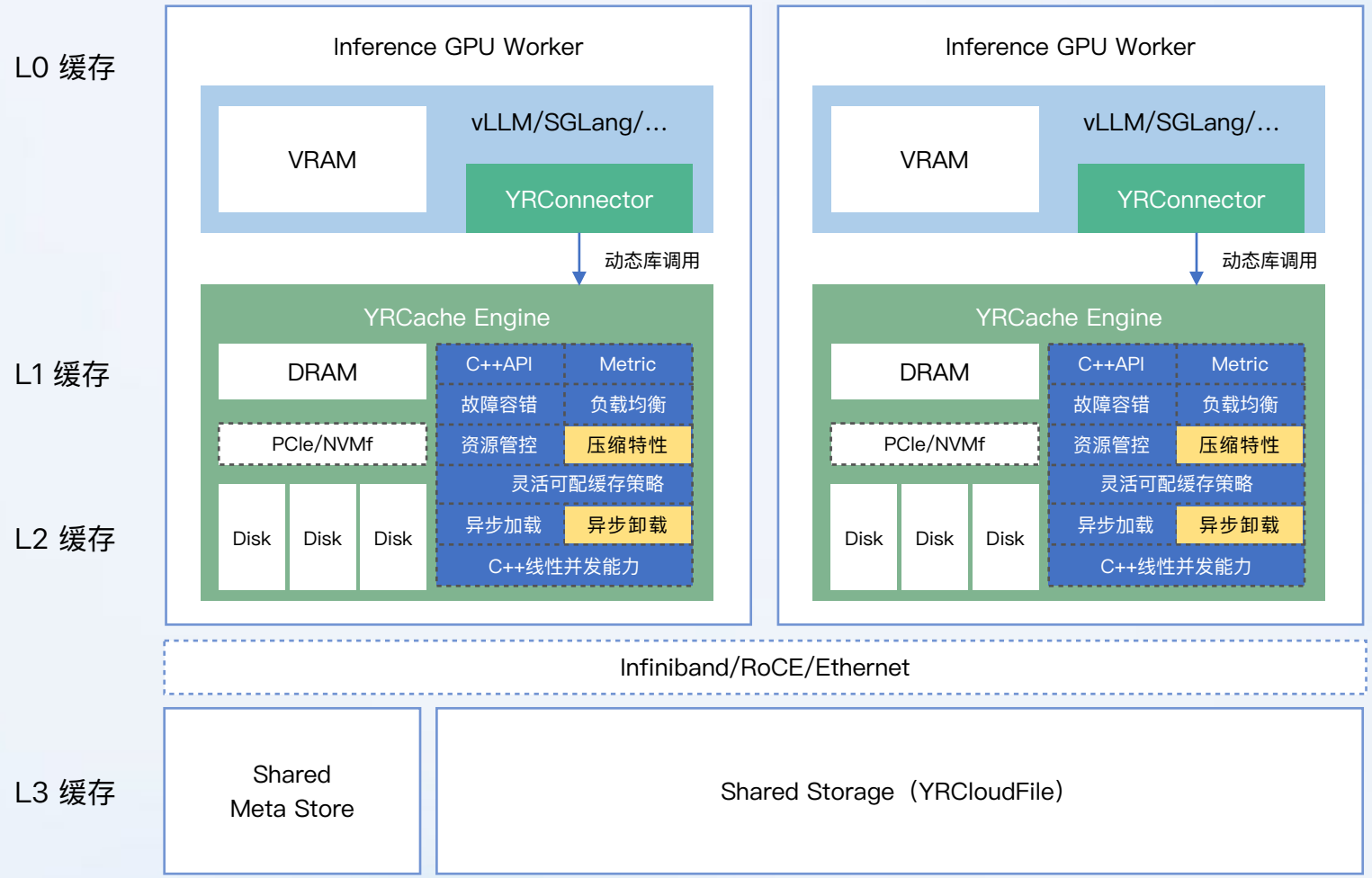
02

YRCache 多级缓存方案

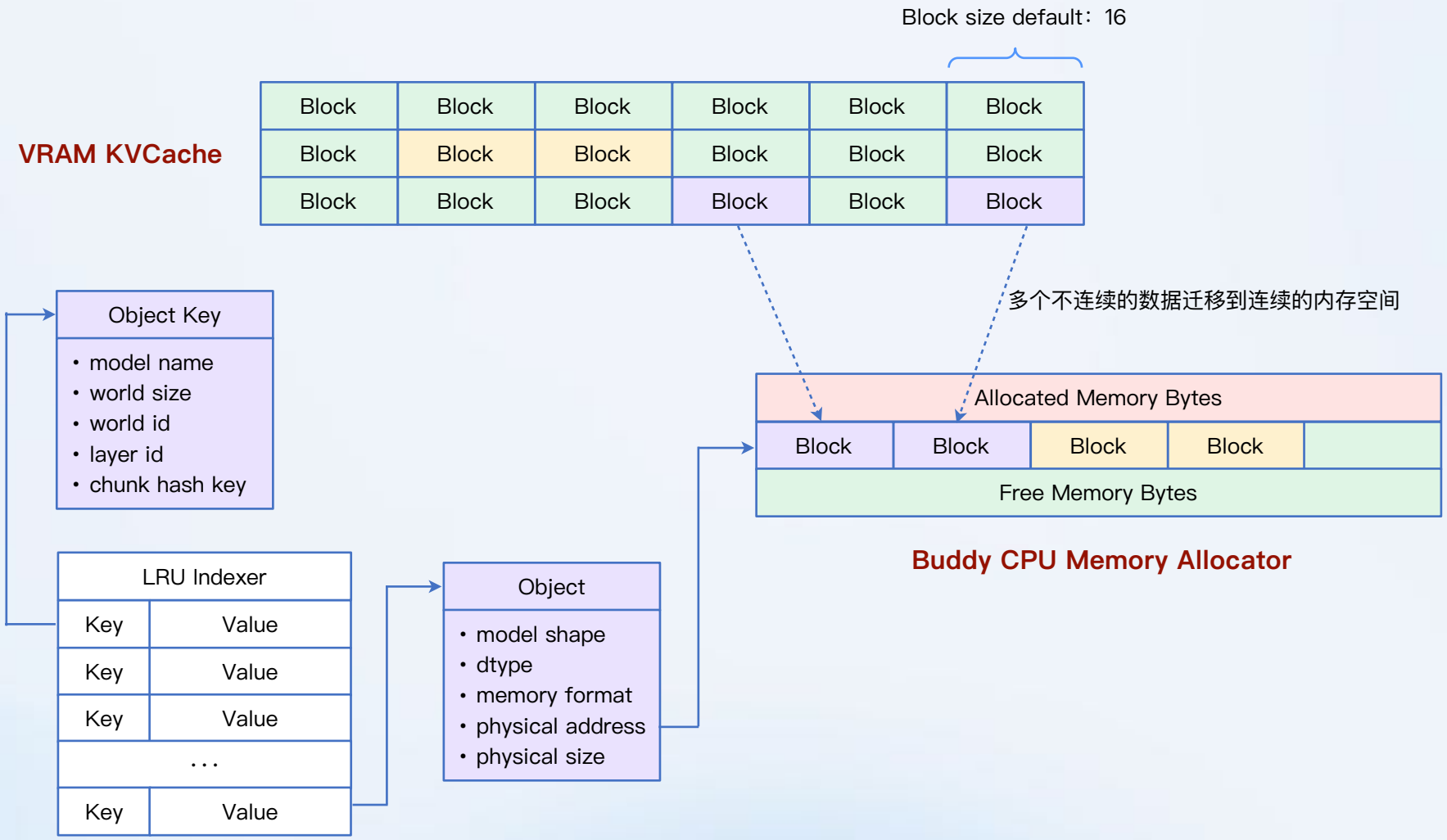
YRCache 的整体架构设计

✦ YRCache 整体设计

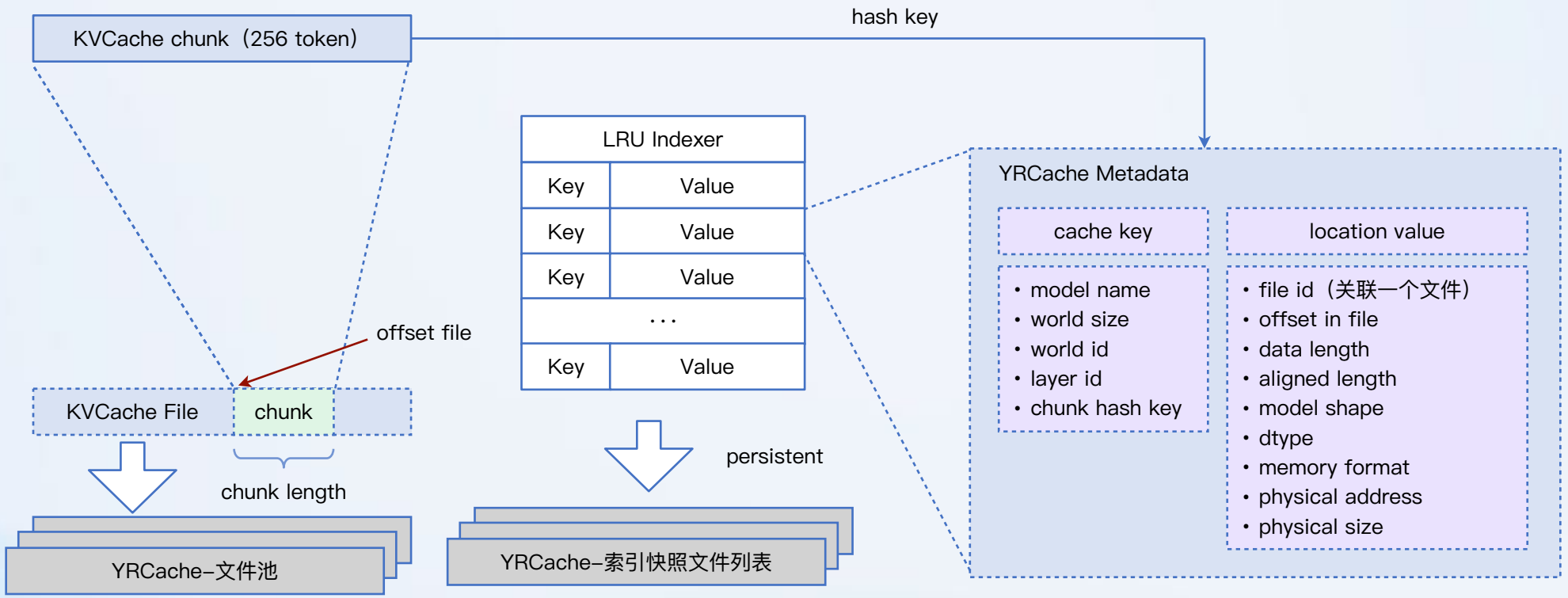
1. YRCache 逻辑上划分为两层，分别是推理平台对接的connector层和缓存管理的Cache Engine层
2. Connector层是以插件的形式嵌入到推理平台，比如说Scheduler会直接调用Connector的查询接口
3. Cache Engine层采用C++实现高性能的数据并发处理能力，并提供C++和Python两种接口
4. YRCache 可以支持多种存储介质做为缓存层，灵活可配，可以任意组合



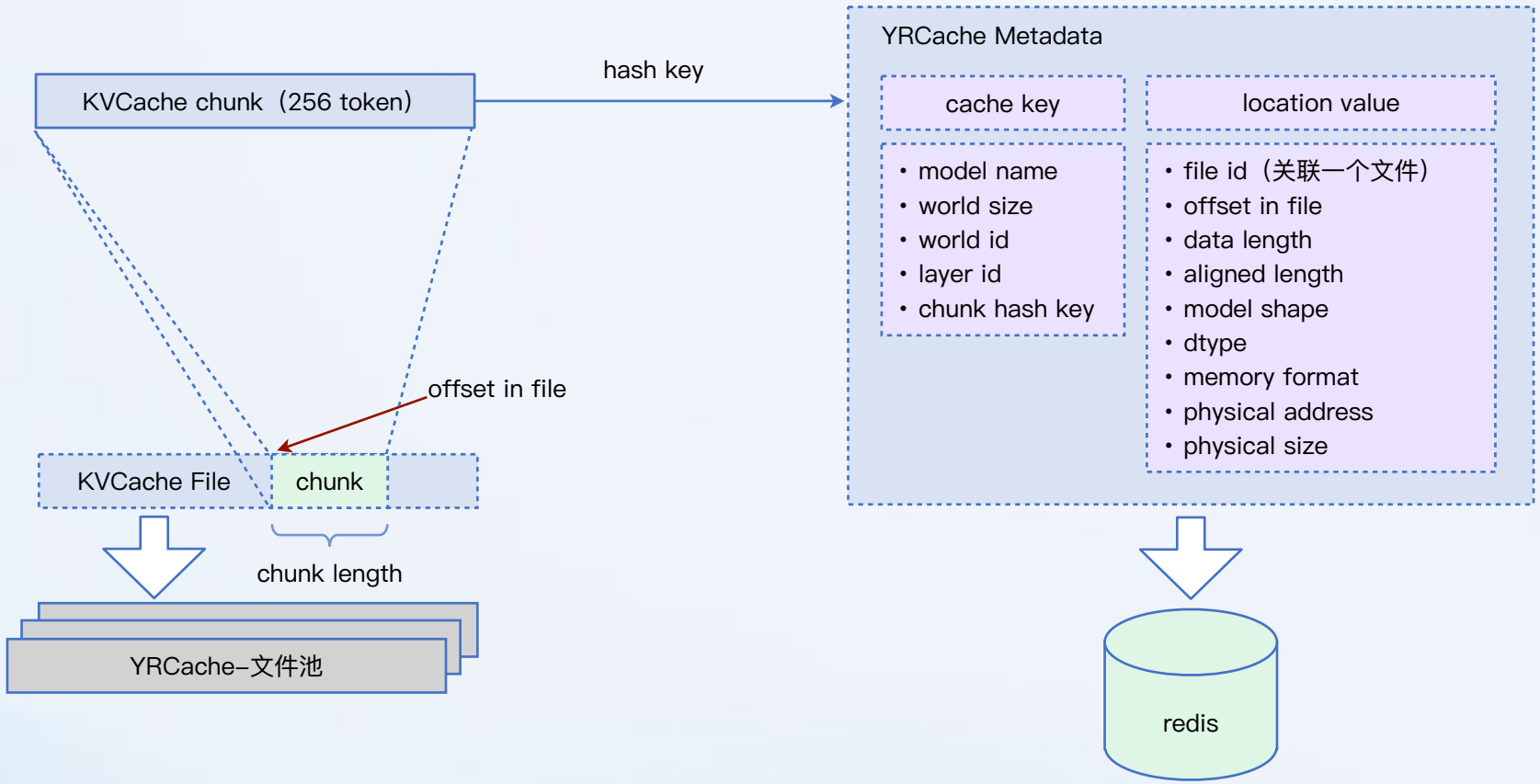
缓存数据管理 | CPU 内存



缓存数据管理 | 本地盘


KVCache chunk（256 token）
hash key
LRU Indexer
Key
Value
Key
Value
Key
Value
...
Key
Value
YRCache Metadata
cache key
location value
• model name
• world size
• world id
• layer id
• chunk hash key
• file id（关联一个文件）
• offset in file
• data length
• aligned length
• model shape
• dtype
• memory format
• physical address
• physical size
offset file
KVCache File
chunk
chunk length
persistent
YRCache-文件池
YRCache-索引快照文件列表

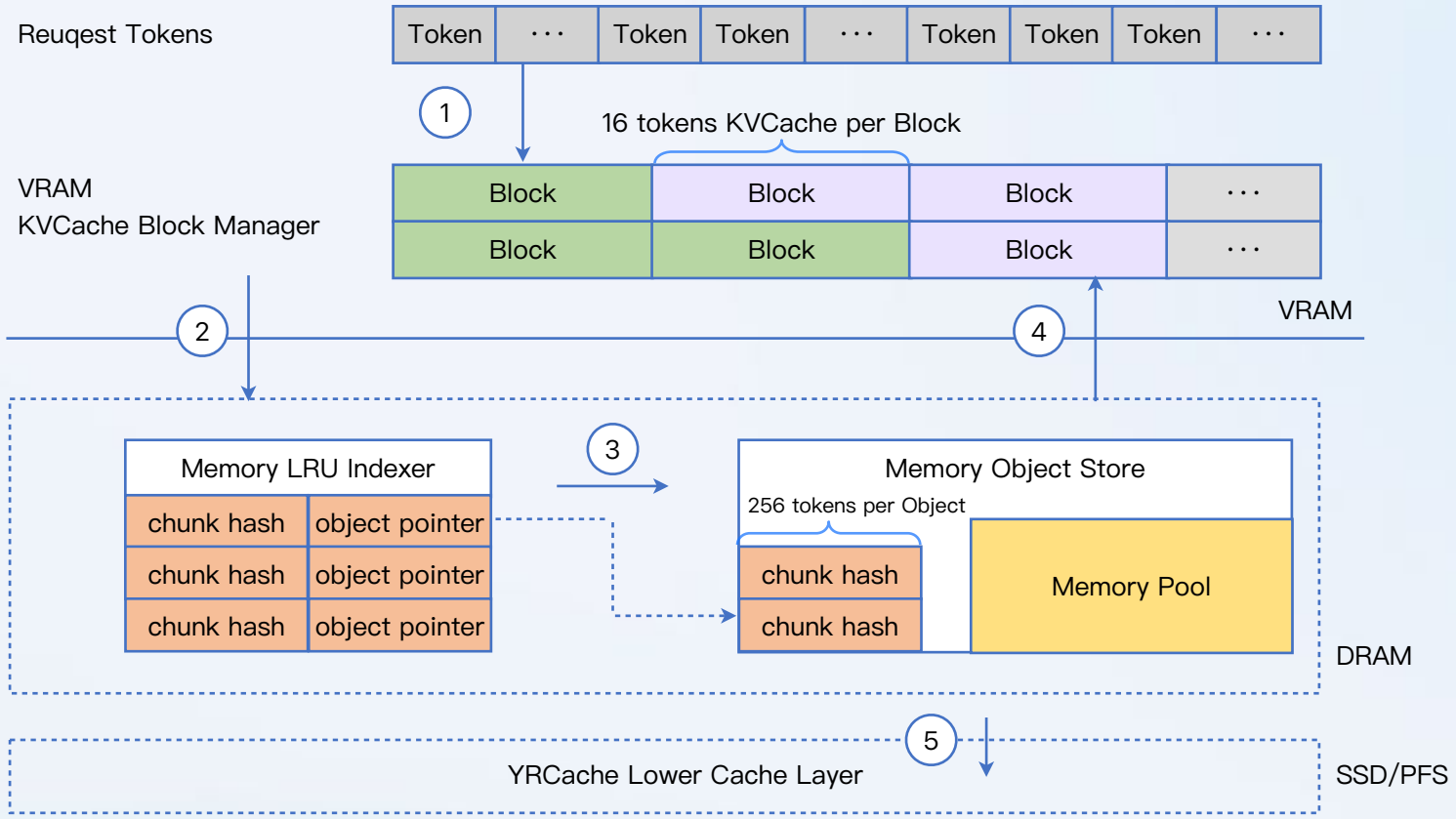
缓存数据管理 | 共享文件存储



KVCache 读写流程 | 写流程

✦ KVCache 缓存写入流程

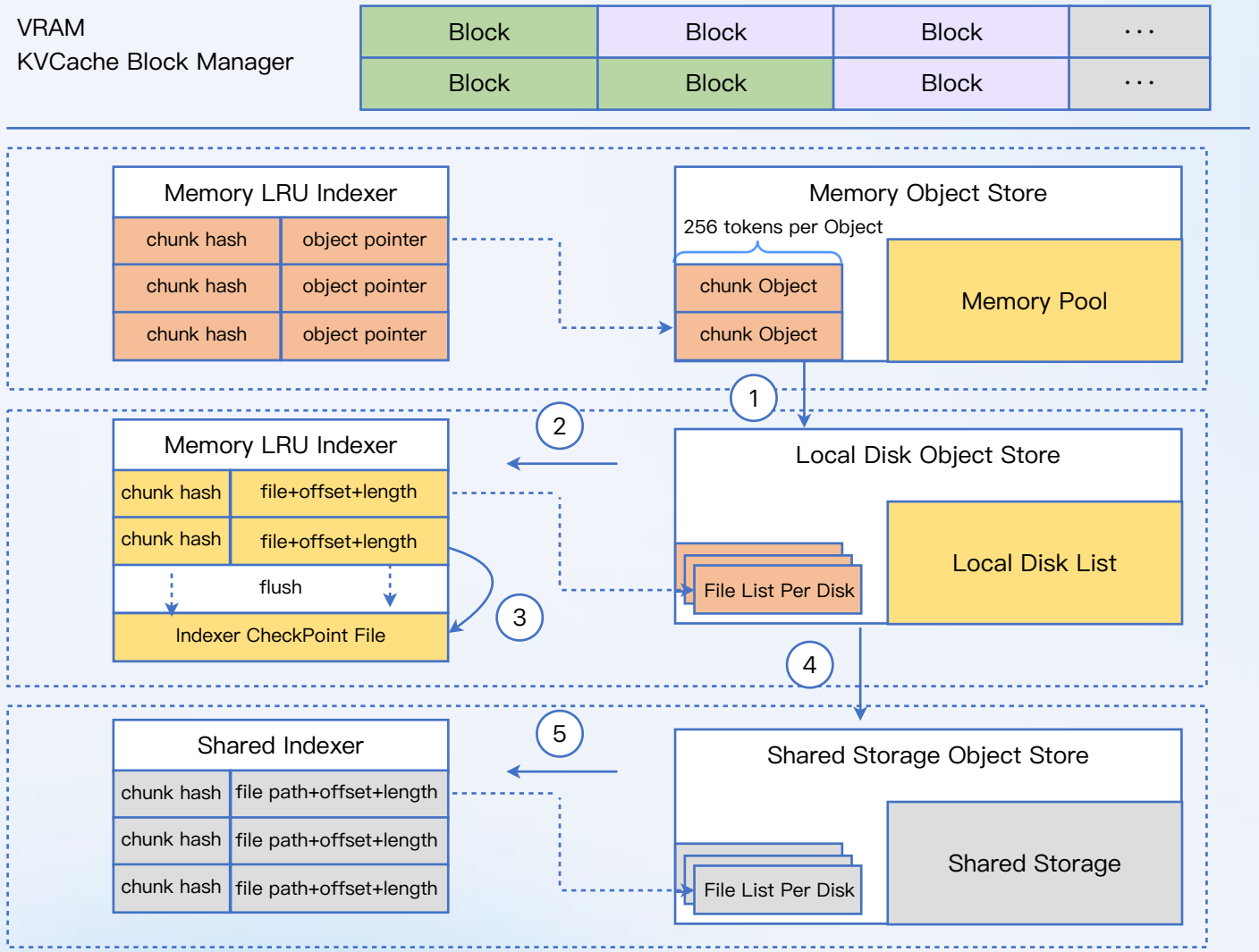
1. 在显存中申请 Block，计算生成 KVCache 并写入 Block
2. 申请 CPU 内存，并把 KVCache 数据拷贝到CPU 内存
3. 把数据写入对应的 Chunk Object
4. 通知上层业务写入完成
5. 异步方式按需写入其他存储层



KVCache 读写流程 | 缓存异步下刷流程

✦ KVCache 异步下刷流程

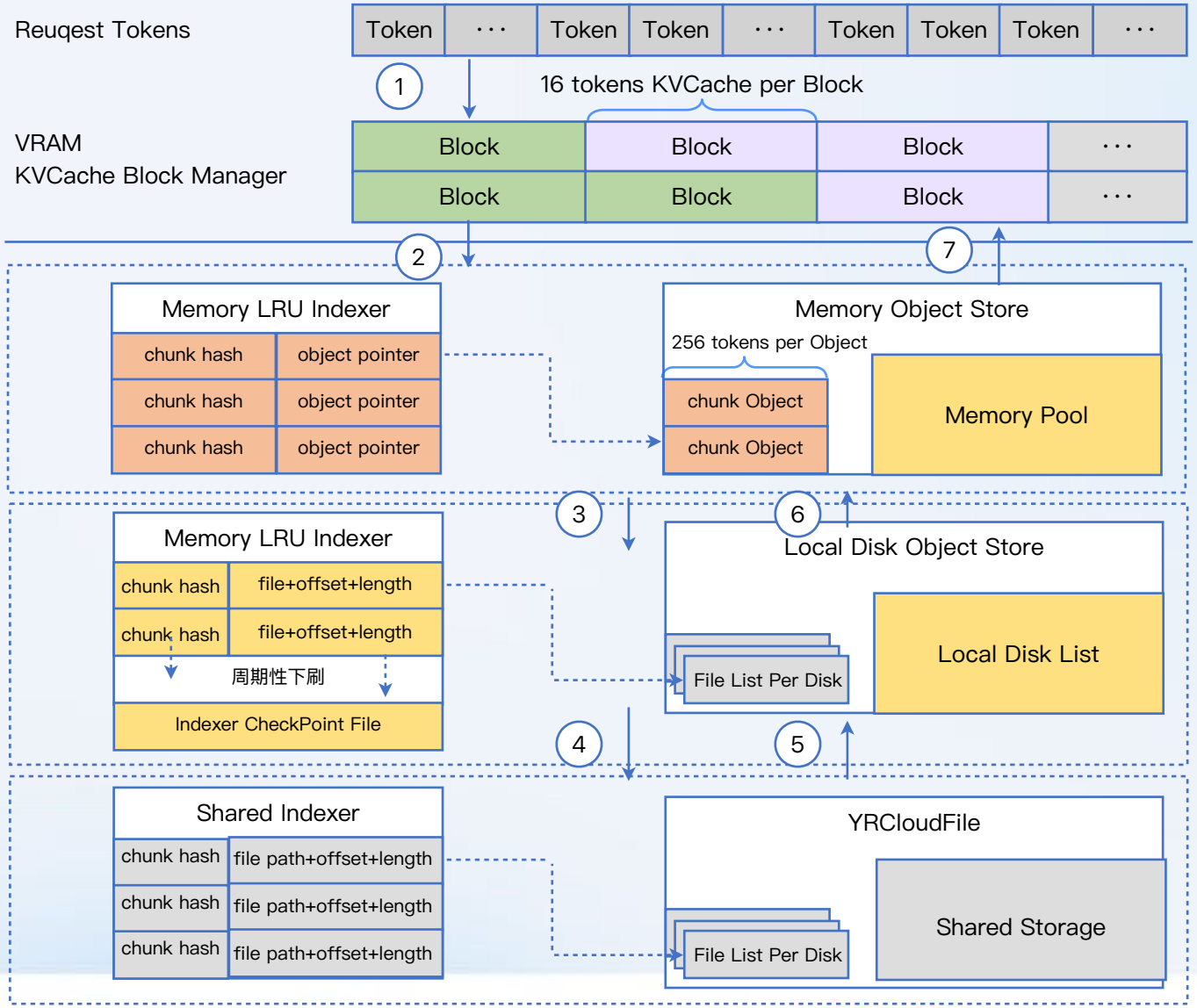
1. 将内存数据块写入本地文件系统
2. 记录文件索引信息到LRU Indexer
3. 周期性写入索引信息到磁盘
4. 将数据透传给共享文件存储，写入共享存储文件
5. 写入完成后更新共享存储的全局索引信息



KVCache 读写流程 | 缓存命中的读流程

✦ KVCache 缓存命中的读流程

1. 分配 VRAM Block 用于存储 KVCache，同时查询 Prefix Cache，没有被 Prefix Cache 命中的 tokens 尝试从 YRCache 读取
2. YRCache 首先从内存缓存中查询，命中后则直接执行第7步
3. 内存缓存中没有命中则申请一块 CPU 内存，然后把请求传递给本地盘缓存，本地盘缓存命中后把数据读入 CPU 内存并执行第6步
4. 本地盘缓存没有命中则把请求传递给共享存储缓存，命中后把数据读入CPU 内存
5. 共享存储命中的数据返回本地盘存储，本地盘存储根据配置需求是否回写本地盘存储，同时把数据透传给上一层
6. 本地盘把命中的数据返回给内存缓存，内存缓存根据配置决定是否回写内存缓存池
7. 把命中的数据从内存拷贝到 GPU 显存



缓存置换策略

✦ 数据生命周期

- **内存层**

  根据配置的内存缓存容量来控制内存缓存层的数据量，超过阈值的数据根据 LRU 算法淘汰，内存缓存层淘汰出去的数据一般都已经被持久化在 SSD、共享文件系统

- **本地盘层**

  本地盘模式下，各个盘之间的数据独立管理，每个盘根据配置容量来控制磁盘写入数据量，超过阈值的数据将根据 LRU 算法淘汰

- **共享存储层**

  周期性的清理历史数据，根据配置定期清理一段时间没有访问的历史数据

✦ 灵活的缓存配置

YRCache 为每一层的每个动作配置特定的数据流转策略

- **Lookup/Get**

  kStopAtCurrentLayer -- 只在当前缓存层查找，如果没命中则直接返回未命中

  kTryFromLowerLayer -- 如果当前层没有命中，需要向更低层的缓存层查找

- **Set**

  kStopAtCurrentLayer -- 数据只缓存在当前层，不向更低层透传数据

  kSetAlsoLowerLayer -- 数据缓存在当前层，同时向更低层透传数据

```
# about cpu memory cache configuration
cpu_cache_config:
  max_cpu_memory_size: 5          # memory size of GiB
  enable_cpu_memory_cache: false # default true
  cache_policy:
    get_policy: 1
    return_policy: 2
    set_policy: 2
    lookup_policy: 1

# about local disk cache configuration
local_disk_cache_config:
  enable_local_disk_cache: true  # default false
  file_count_per_disk: 10        # how many files be created on a dir
  local_disks:                   # local directory list
    - "/mnt/nvme0n1:80"
    - "/mnt/nvme1n1:90"
  cache_policy:
    get_policy: 1
    return_policy: 2
    set_policy: 2
    lookup_policy: 1

# about shared storage cache configuration
shared_storage_cache_config:
  enable_shared_storage_cache: false # default false
  redis_host: "172.16.1.194"           # metadata store redis_host
  redis_port: 6379                   # metadata store redis_port
  redis_pool_size: 8                 # metadata store redis pool size
  redis_wait_timeout_ms: 5000        # metadata store redis wait timeout, unit ms
  redis_connect_timeout_ms: 1000     # metadata store redis connect timeout, unit ms
  redis_socket_timeout_ms: 3000      # metadata store redis socket timeout, unit ms
  file_count: 100                  # how many files be created on shared dir
  mount_path: "/mnt/yrfs"          # shared storage mount pointer
  data_sub_path: "/yrcache_01"     # sub directory of mount pointer
  cache_policy:
    get_policy: 1
    return_policy: 2
    set_policy: 1
    lookup_policy: 1
```

高性能文件系统 YRCloudFile 整体架构

焱融追光一体机 F9000X

3 节点集群性能 480GB/s，750万 IOPS

智能数据分层

冷数据自动分层至对象存储

丰富运维手段

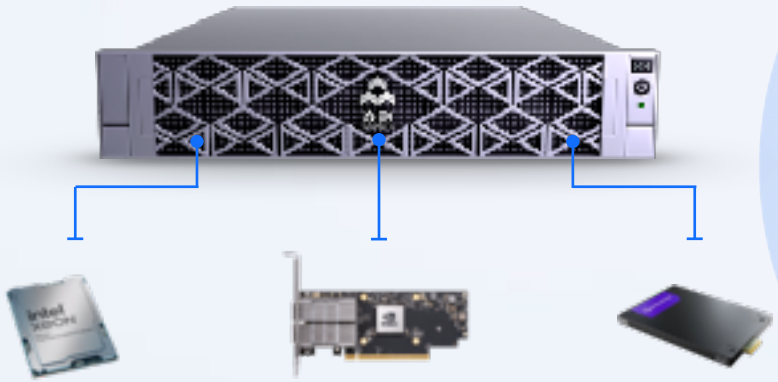
配额管理、日志审计、回收站

弹性数据网络

同一集群可使用IB及以太访问

智能数据加载

打通对象和文件的数据流动



第 5 代英特尔® 至强® 可扩展处理器

NVIDIA NDR400 InfiniBand /400GbE Ethernet RoCE

支持 E3.S/ U.2 PCIe 5.0 TLC 和 QLC NVMe SSD

焱融追光一体机 F9000X

- ✓ 每节点4*400Gb IB/RoCE 网络
- ✓ 支持 SpectrumX
- ✓ 单节点性能 160GB/s 带宽 和 250 万 IOPS
- ✓ 基于多 IB/RoCE 网口 Multi Channel 性能优化
- ✓ 支持 GDS 高级特性

YRCache 一套机制对接多个推理平台

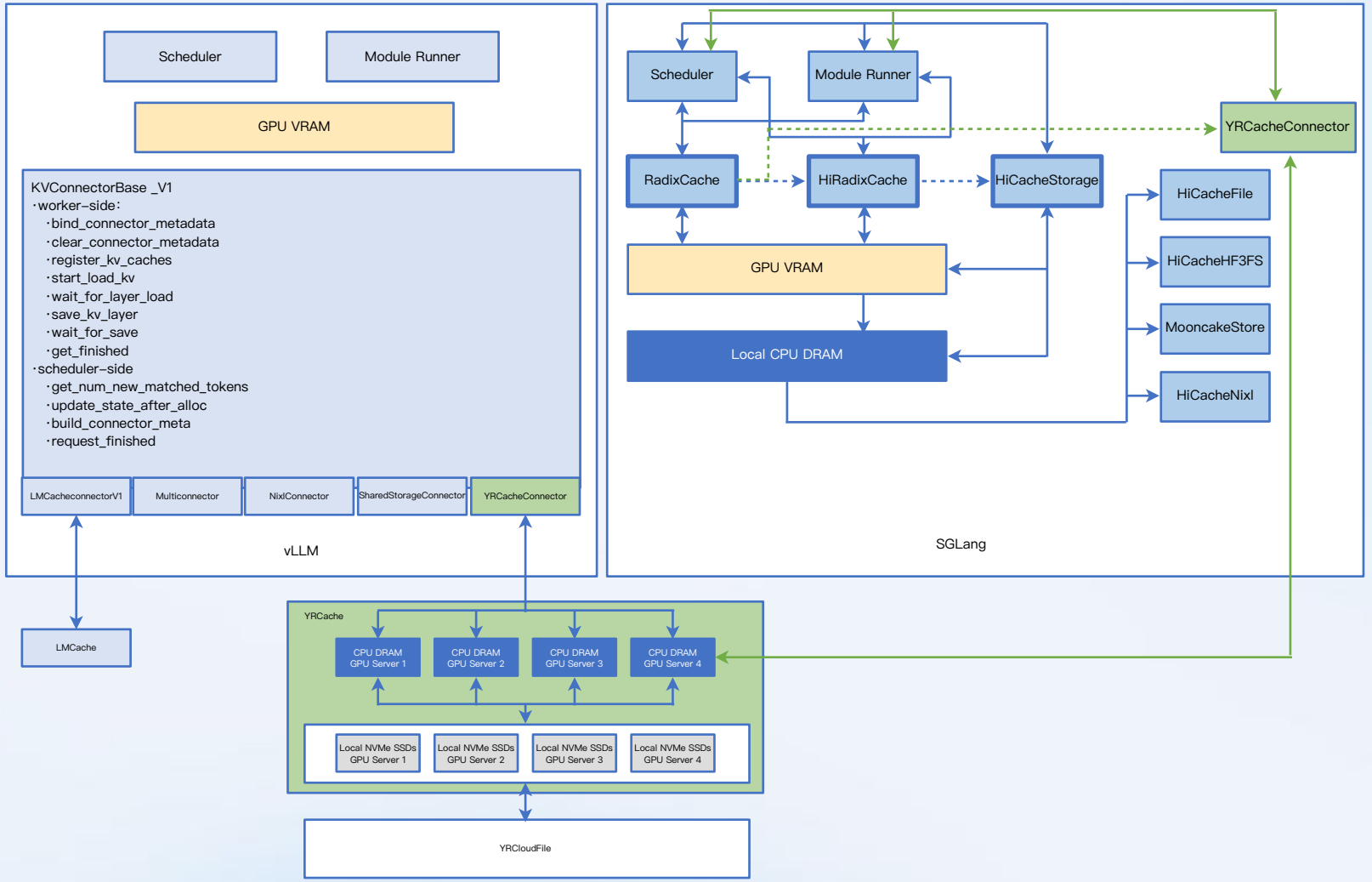
✦ 对接多个推理平台方案

优势

1. 提供统一的多级缓存管理
2. YRCache 代码便于维护

劣势

1. 需要独立维护适配patch
2. 版本兼容性的工作量繁重



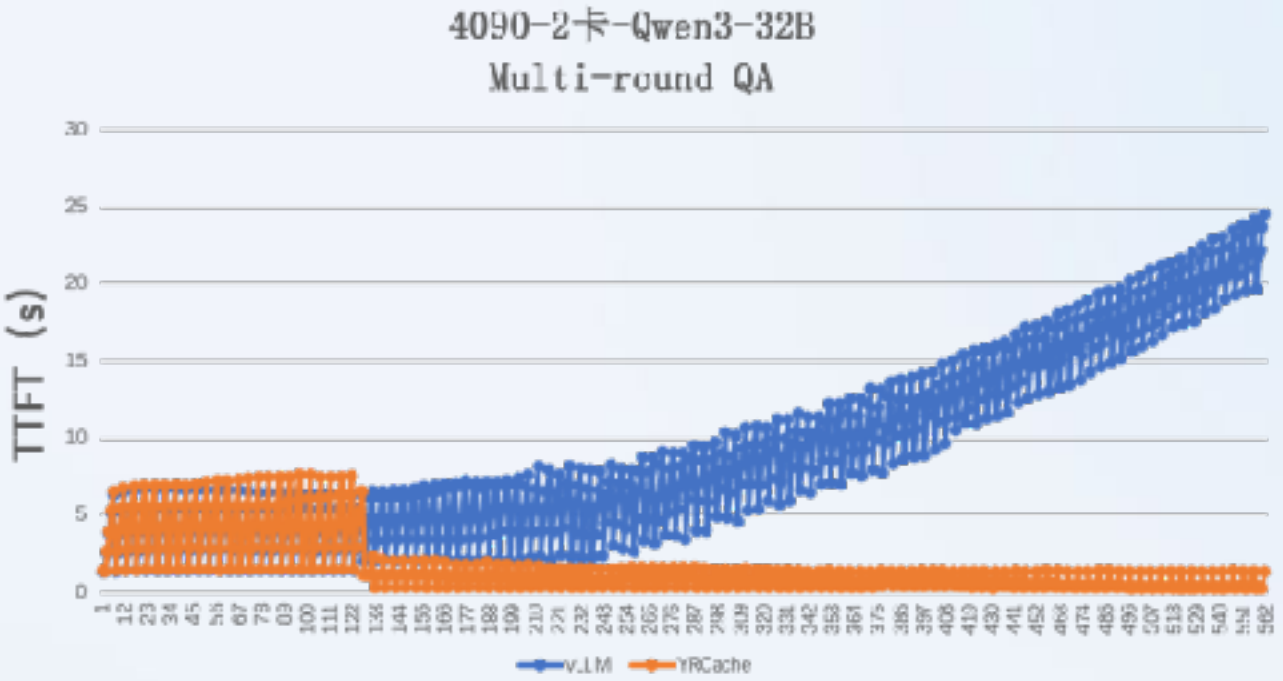
03

针对推理业务的加速实践效果

智能客服/多轮对话场景

测试内容-多轮问答

在相同 NVIDIA 4090*2 显卡配置下，对 Qwen3-32B 模型，通过 multi-round-qa.py 模拟多个用户同时与大模型进行多轮交互，从而分析服务引擎的吞吐量和延迟，在并发数递增时，分别使用原生 vLLM（PrefixCache） 和 vLLM + YRCache 进行测试。



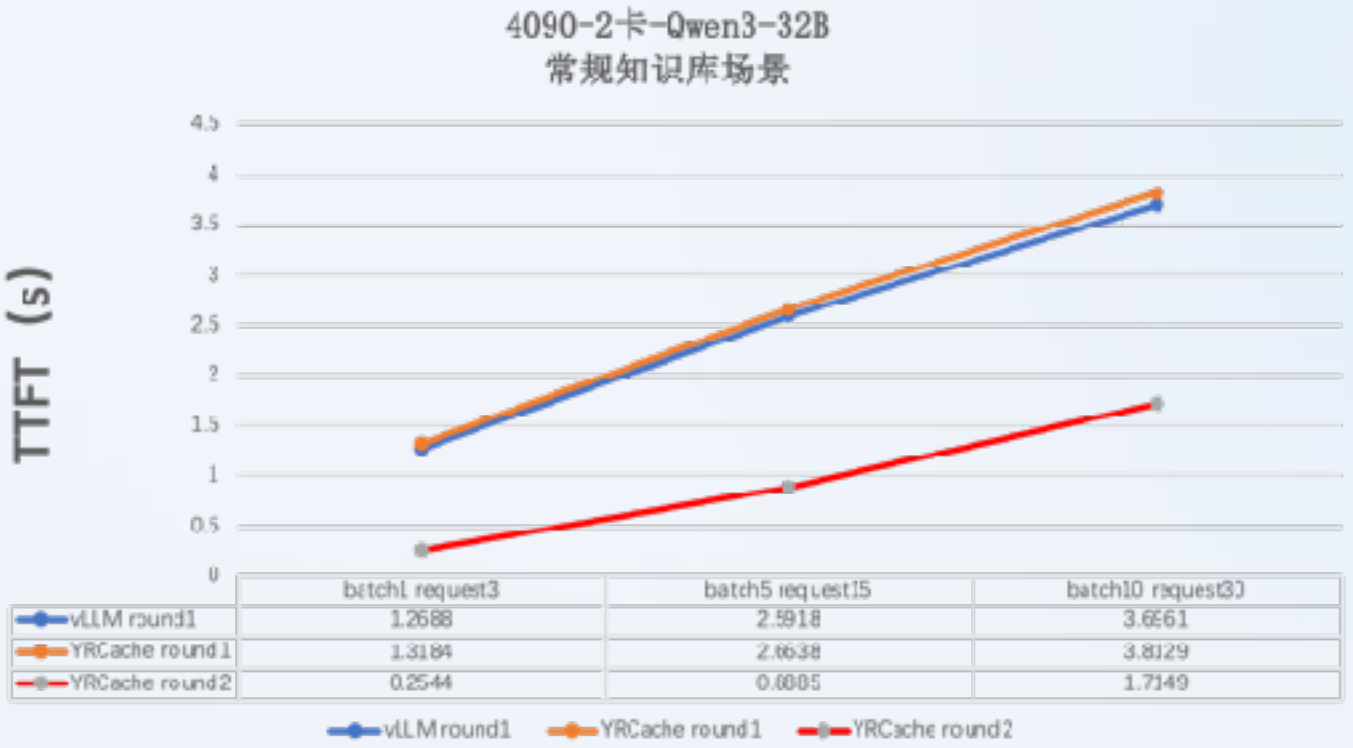
测试结论

- YRCache 在多轮对话场景，YRCache 命中缓存后，TTFT 稳定在2秒内，对比原生 vllm 用时最短为 1/20, YRCache 整体命中率为 56%。

RAG知识库问答场景

测试内容-知识库

在相同 NVIDIA 4090*2 显卡配置下，对 Qwen3-32B 模型，通过 evalscope ，模拟知识库所需要输入4096 toke n和输出1024 token 数量进行并发测试 ，在并发数递增时，分别使用原生 vLLM（PrefixCache） 和 vLLM + YRCache 进行测试。



测试结论

- 在全部并发过程中，在第二轮缓存命中的情况下，YRCache 整体用时降低 55%–80% 左右；

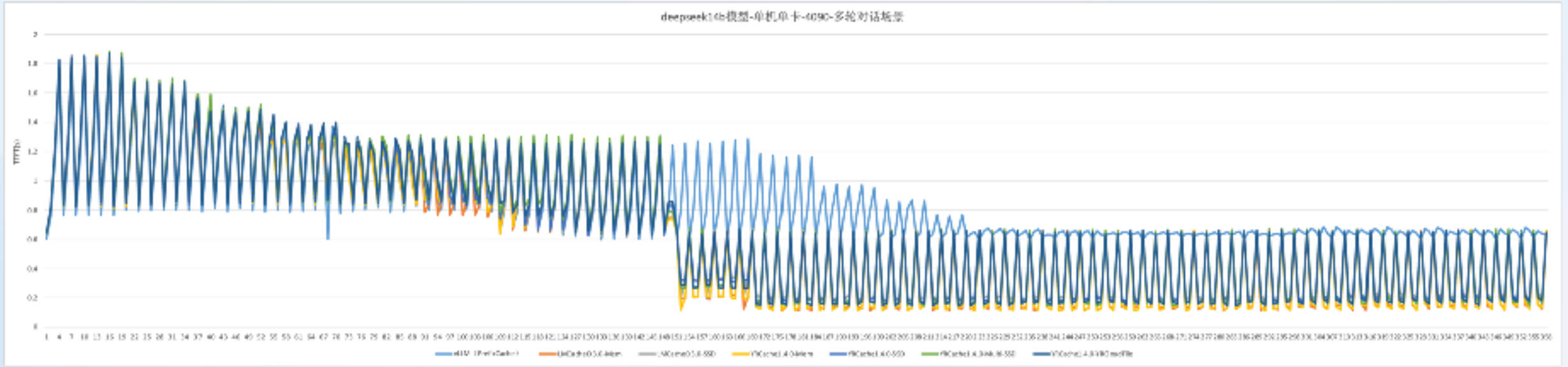
YRCache 与 LMCache 的性能对比 | 多轮对话场景

测试内容

在相同单张 4090 显卡配置下，对 DeepSeek14B 模型，通过 multi-round-qa 工具 进行多轮对话测试，分别使用原生 vLLM（PrefixCache）、vLLM + LMCache 内存模式、vLLM + LMCache 本地盘模式和 vLLM + YRCache 多种缓存模式进行测试

测试结论

- 原生 vLLM 的性能最差，基本上没有 KVCache 命中
- 无论 YRCache 的内存模式、本地盘和远端共享存储模式均和 LMCache 性能上没有差异



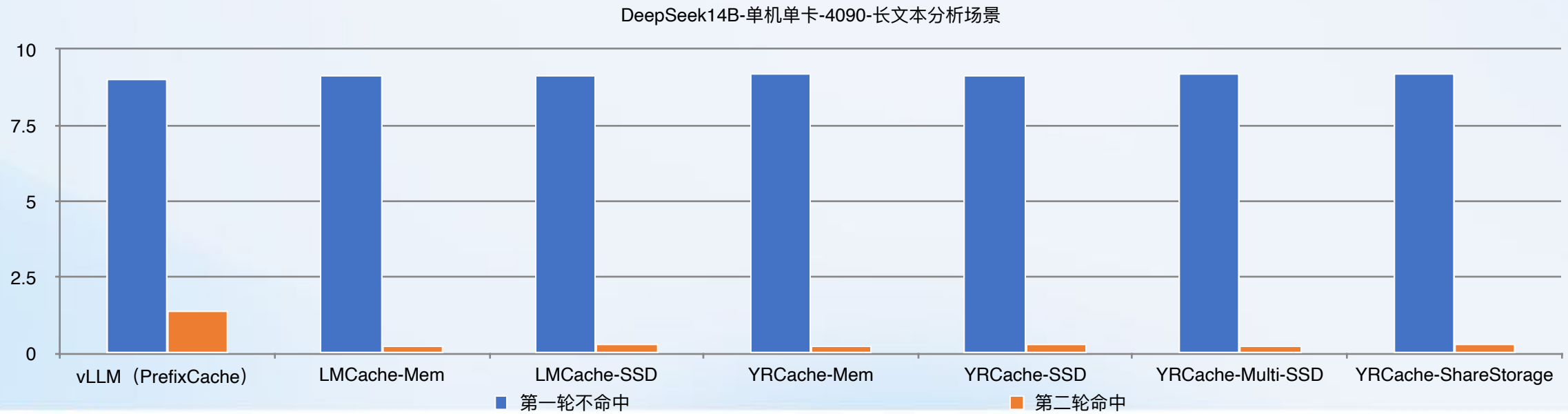
YRCache 与 LMCache 的性能对比 | 长文本分析

测试内容

在相同单张 4090 显卡配置下，对 DeepSeek14B 模型，对多个 128K 长度的文本进行问答，分别使用原生 vLLM（PrefixCache）、vLLM + LMCache 内存模式、vLLM + LMCache 本地盘模式和 vLLM + YRCache 多种缓存模式进行测试景，第一轮是缓存不命中，第二轮是缓存命中

测试结论

- 原生vLLM由于显存空间较小，缓存没办法完全命中，性能最差
- YRCache 和 LMCache 在内存模式下，第二次缓存完全命中的情况下性能相当
- 在 SSD 模式下，YRCache 相比 LMCache 性能有 13%的性能优势



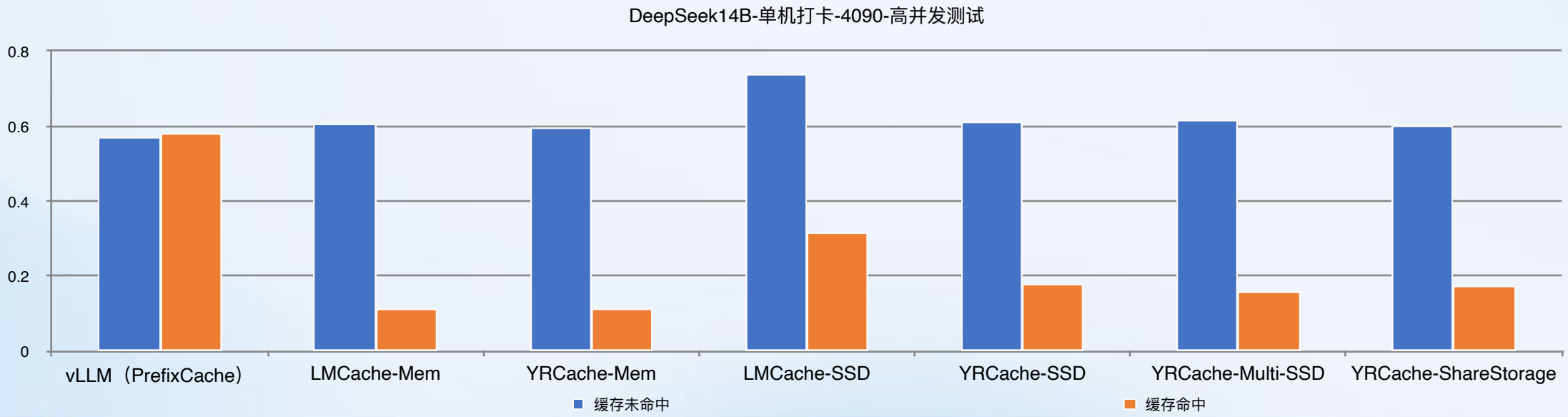
YRCache 与 LMCache 的性能对比 | 高并发测试

测试内容

在相同 NVIDIA 4090*2 显卡配置下，对 Qwen3-32B 模型，通过 evalscope 进行并发测试，在固定并发数量时，分别使用原生 vLLM + LMCache 和 vLLM + YRCache 进行测试，第一轮模拟了KVCache不会命中的场景，第二轮模拟了KVCache缓存命中的场景

测试结论

- 内存模式下，LMCache 和 YRCache 相差不大
- 在本地盘模式下，YRCache 的 TTFT 延迟是 LMCache 的 35%
- 在共享盘模式下，YRCache 的 TTFT 延迟是 LMCache 的 31%



04

总结和未来展望

总结 | 多级智能缓存加速推理效率

更大的缓存空间

PB 级高性能 KVCache 多级缓存

更高的缓存命中率

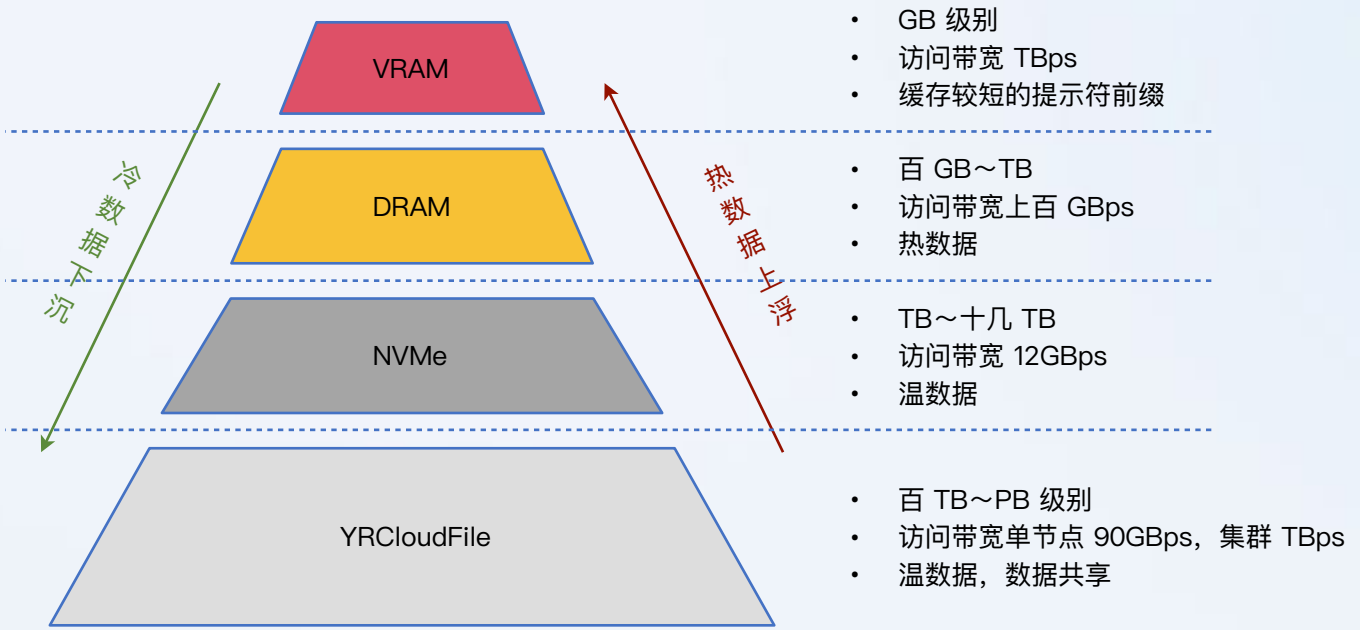
缓存数据越多，缓存命中率就越高

更快的访问速度

智能的缓存管理，访问数据更高效

更灵活的架构选择

支持 KVCache 重用和跨节点传输



未来规划

- 交叉注意力重建与选择性KV重算
  - 现有的KVCache方案中只在复用片段位于输入前缀时有效，如果要复用其他位置片段会忽略跨片段的交叉注意力，容易产生错误，通过度量KV偏差和注意力偏差，识别出导致质量下降的高偏差token，仅对这部分进行重算，以有限代价恢复跨片段依赖
- 对话分块与冗余片段检测
  - 当前的分块方法依赖固定长度，而对话片段在不同上下文中位置常常发生变化，即使内容完全相同，只要在不同会话中落入不同的分块边界，就会导致无法复用，通过采用内容定义分块技术，使得分块边界与文本语义更加吻合，从而减少边界偏移带来的缓存失效
- KVCache 数据压缩
  - 数据量增长迅猛，在从DRAM或者NVMe到共享文件存储之前进行数据压缩，减少网络传输数据量，增加有效数据传输带宽
- 异步加载和缓存置换
  - 优化各个缓存层之间的数据置换策略，将数据的转化流程移除到关键IO路径之外，减少对缓存命中后数据访问的影响

QCon
全球软件开发大会

InfoQ 极客传媒

THANKS

大模型正在重新定义软件

Large Language Model Is Redefining The Software