数据库系统及存储引擎简述

2022年08月29号

电子科技大学

提　纲

一、数据库及相关技术

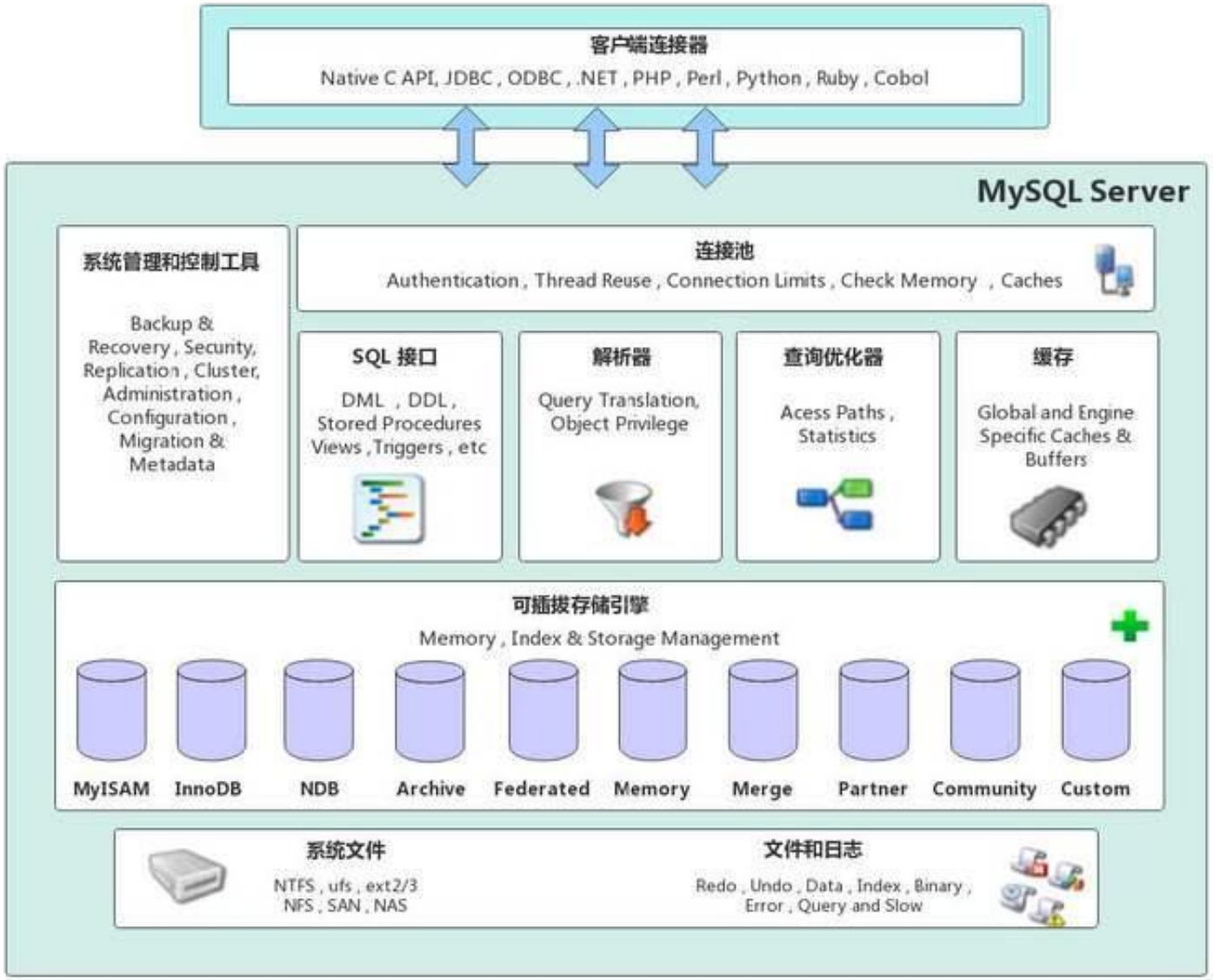
二、内存存储

三、持久型存储引擎

PART 01

数据库及相关技术

一、数据库及相关技术



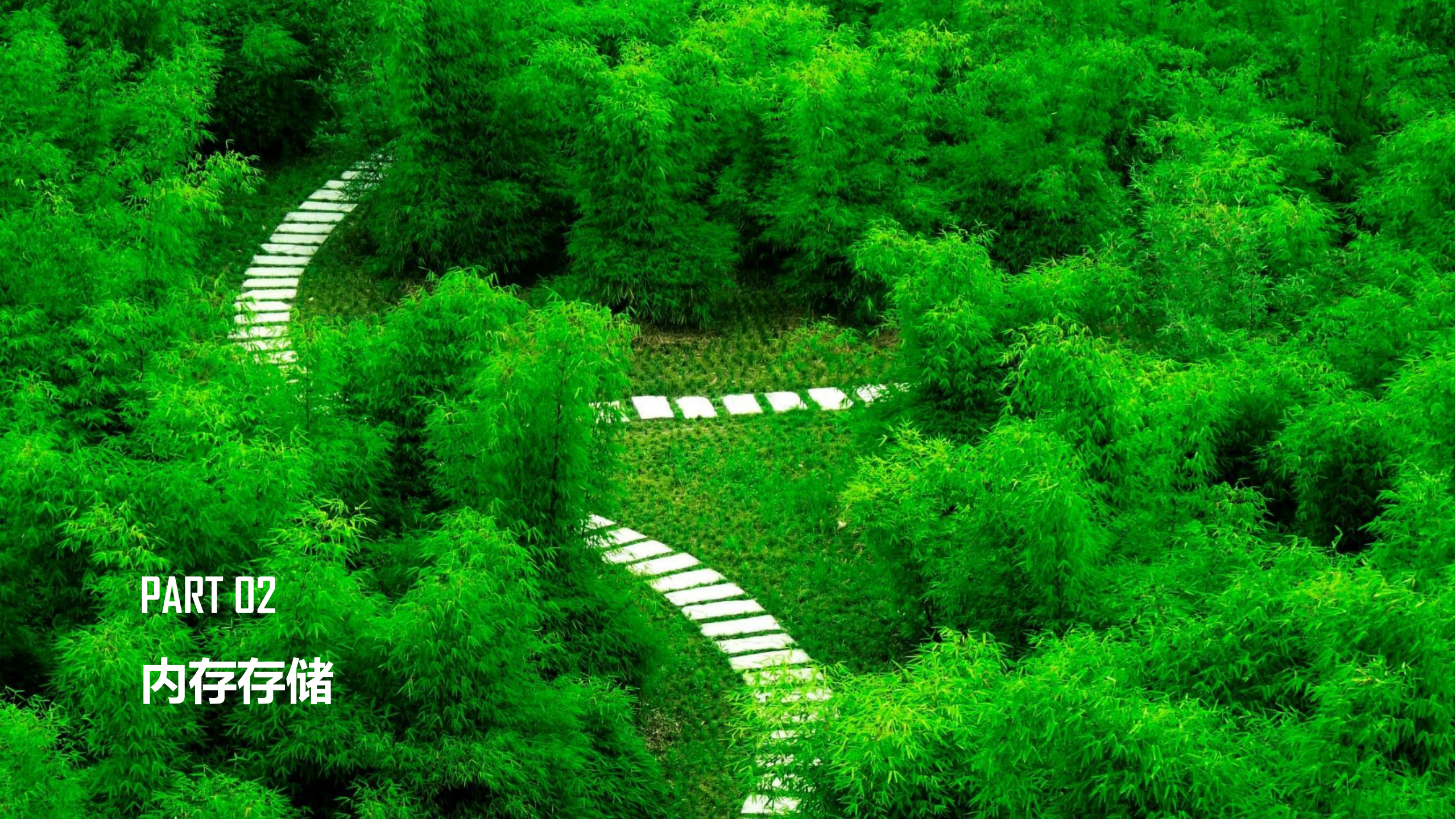
组成架构（mysql）：

- SQL接口
- 计算层
  - 解析器
  - 查询优化器
- 存储层
  - 存储引擎
  - 文件

一、数据库及相关技术

相关技术：

- 计算层
  - 并发控制（事务）
  - 查询优化技术
- 存储层
  - LRU
  - 索引技术
    - B+tree
  - 故障恢复机制



PART 02

内存存储

二、 内存存储

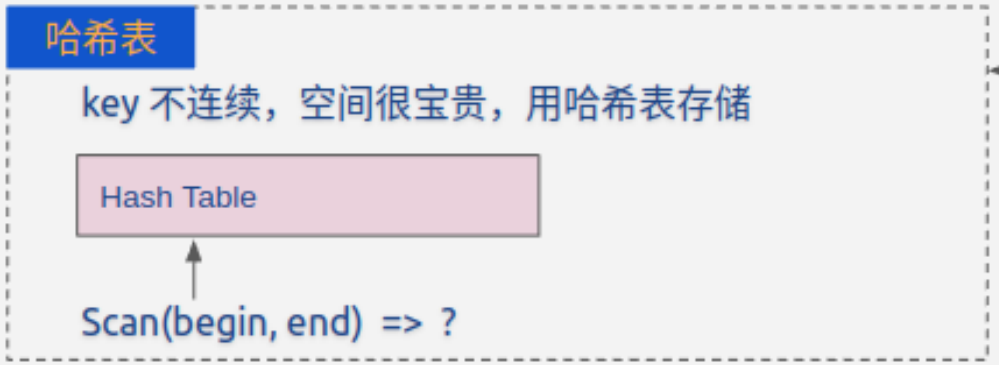
存储应该实现什么样的功能?

- Write/Read接口

数据组织方式

- 有序数组
  - 查找性能：O(logn)
  - 插入性能：查找+挪数据
- 哈希表（数组实现）
  - 查找性能O(1)
  - 插入性能O(1)

数据库read除了单点查询还有scan操作



(。 ﹏ 。) 哈希表 Scan 很为难

有序数组的 Scan：两次二分查找，确定 begin / end 的位置，两位置之间的数据就是结果集 values

结果集 values 是连续存储的，**局部性** 很好。

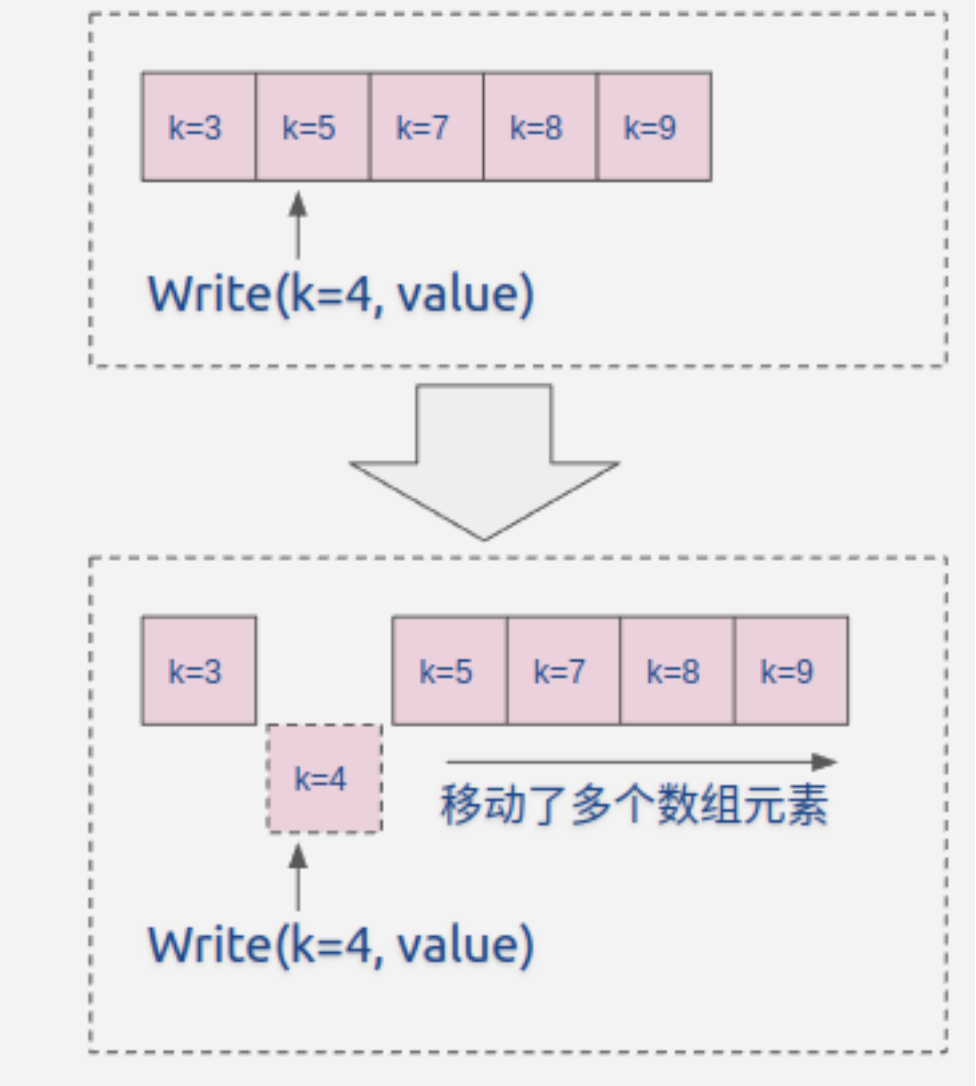
Scan 的性能是两部分的和：

- 定位过程，很不错：O(Log2 n)
- Values 的获取和返回，由于局部性好所以很好

综合起来性能优秀

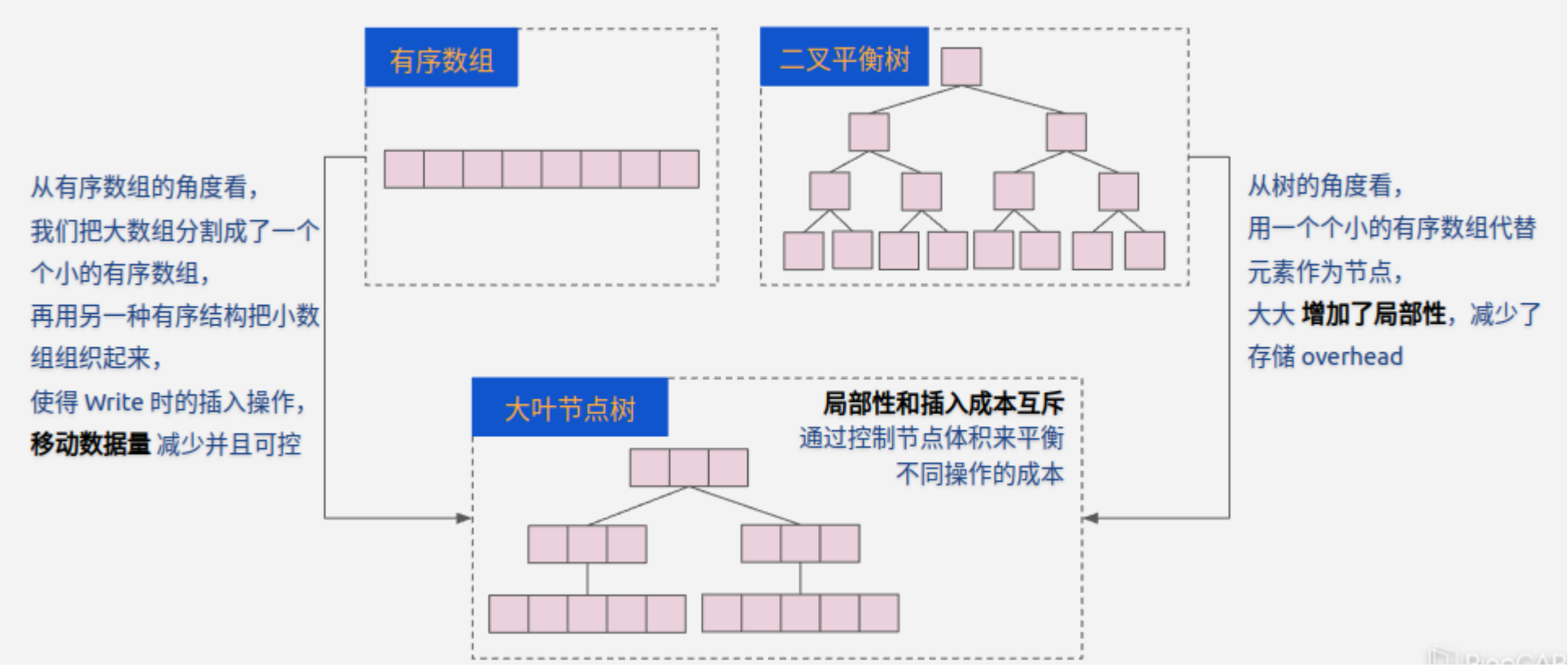
局部性原理

- 局部性
  - Scan性能与数据局部性强相关
    - 平衡二叉树数据局部性差，scan性能低
  - Read性能也与数据局部性强相关
- 局部性过高的缺点
  - 写入时候需要大量的数据移动
  - 对于局部性差的写入相对占优



B+树

有序数组和平衡二叉树的折中。



B+树

有序数组和平衡二叉树的折中。

我们假定 Scan 性能很重要，选择 B+Tree：

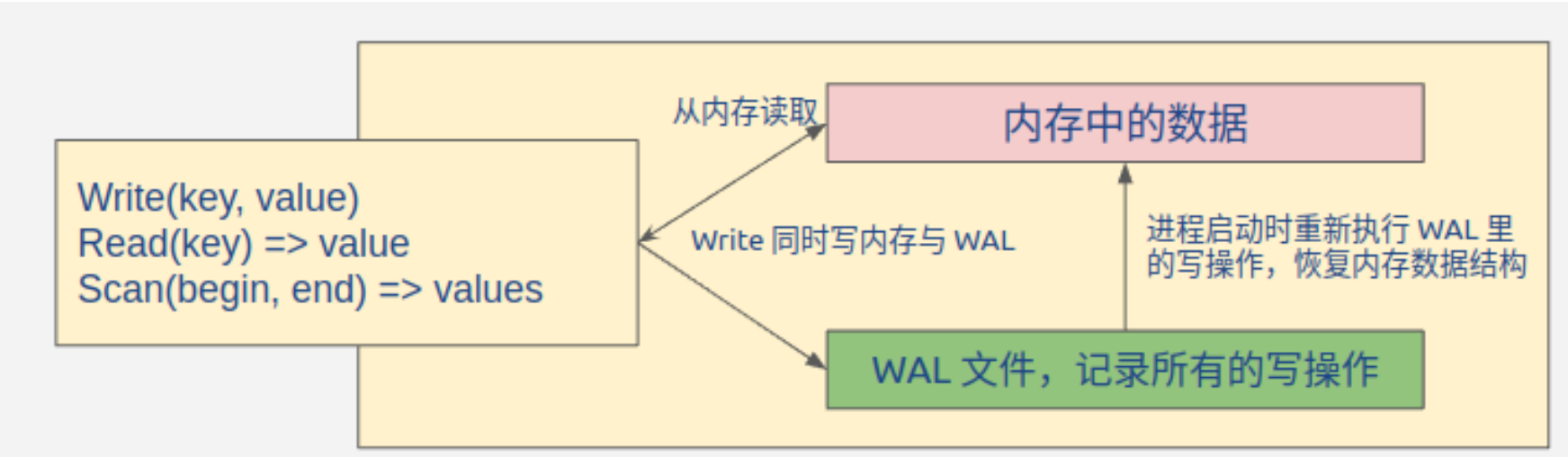
- 在插入过程中动态保持有序
- 把数组拆成多个小段，把小段作为叶节点用 B+Tree 组织起来，让插入过程代价尽量小
- 每小段（也就是叶节点）是一个有序数组，插入数据时只需要移动插入点之后的数据，大大减少移动量

PART 03

持久型存储引擎

三、持久型存储引擎

持久化存储实现方案一：异构镜像



异构：磁盘与内存数据结构不一致

- 磁盘使用局部性高的结构
- 内存可以是任意结构

镜像：逻辑上两边的数据等价

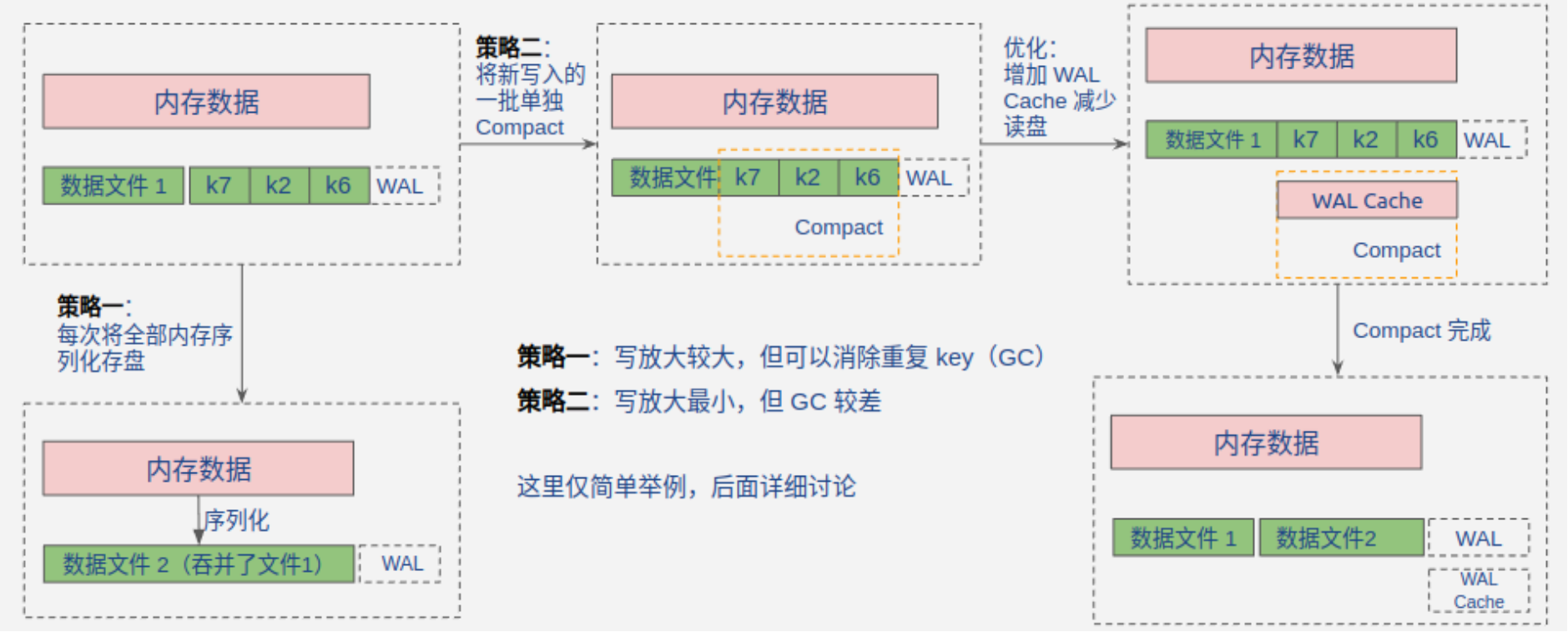
三、持久型存储引擎

WAL 等价于某时刻数据的快照，是数据序列化结果，但不是最优的序列化方式。

•重放 WAL 主要是重做 Write 操作，不一定高效

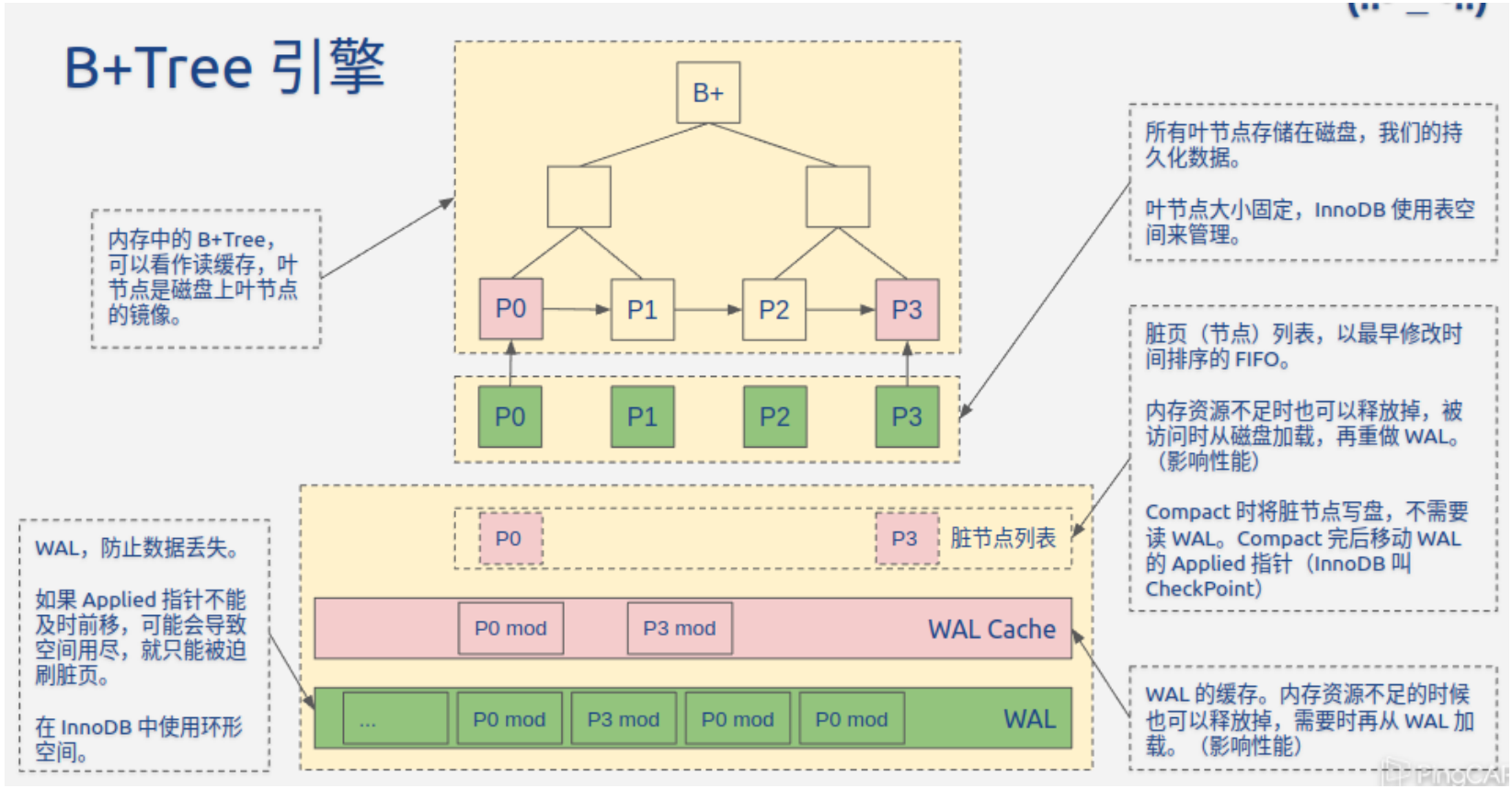
•WAL 中可能存在相同 key 的多次 Write 的多个版本的数据，占用了额外空间，也降低重放性能

解决方式：compaction/刷脏页等



三、持久型存储引擎

以叶节点为单位，将被读写访问的叶节点镜像到内存。



读性能

如果数据所在的叶节点：

在内存，完成读取，较快

不在内存，加载相关叶节点，再从中查找

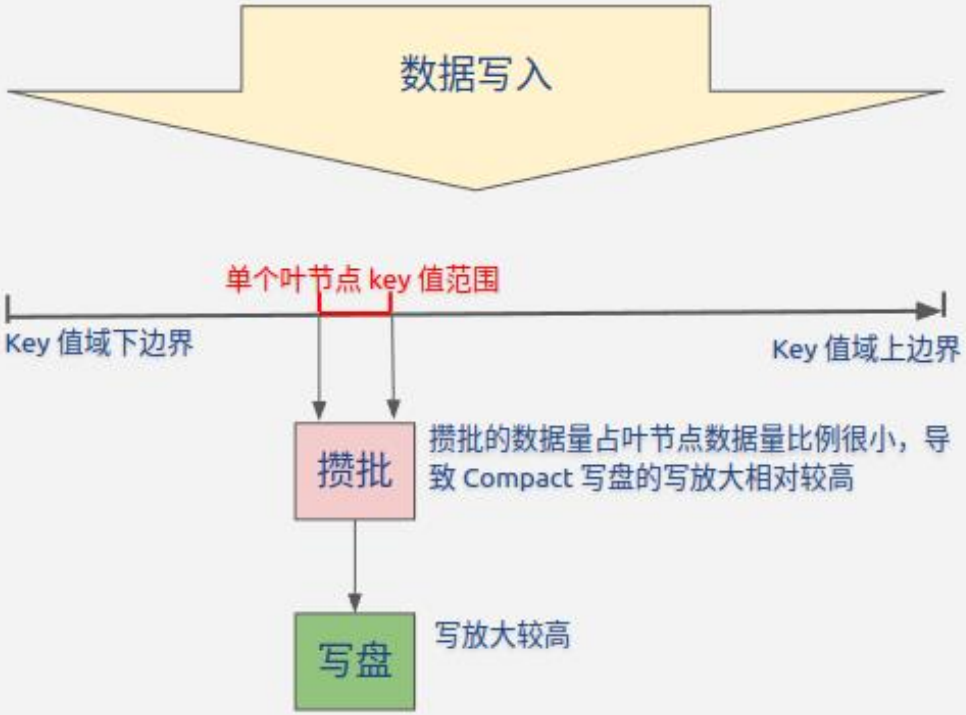
写性能

Append 到 WAL，为顺序 IO，较快

更新到叶节点，若之前节点不在内存，需要先从磁盘加载

三、持久型存储引擎

B+tree写放大问题



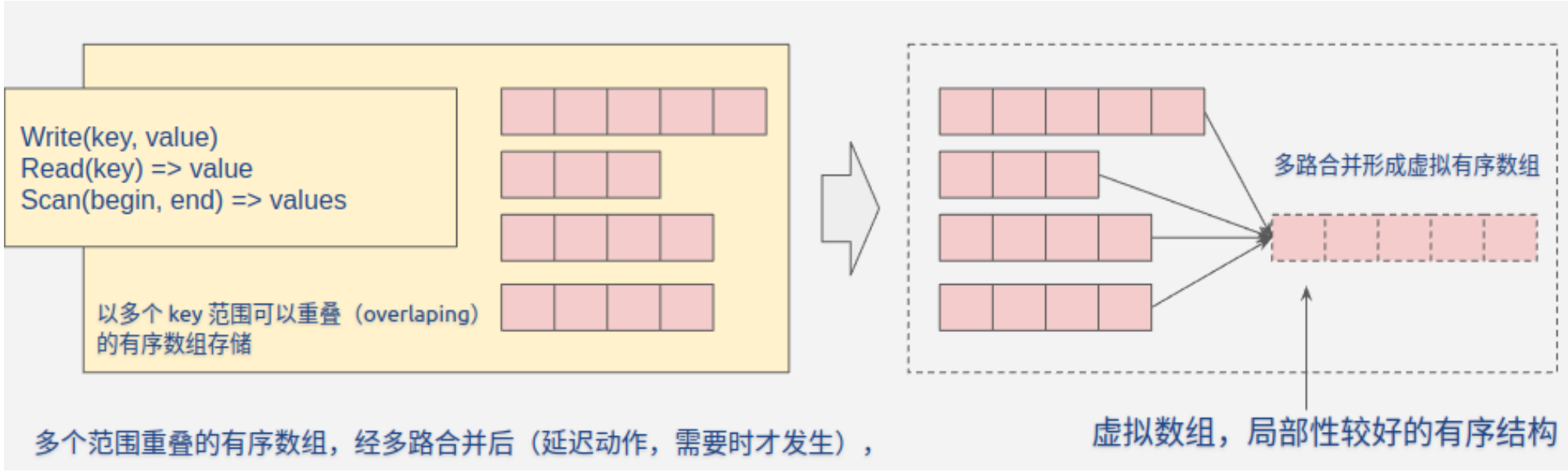
叶节点的 key 值范围，
只覆盖了非常窄的 key 值域，
存量数据越大，叶节点范围越窄。

也就是攒批的 key 值范围很窄。
只有在单热点写入的模式下，
才能达到良好的攒批效果。

这是 B+Tree 写放大的问题来源。

三、持久型存储引擎

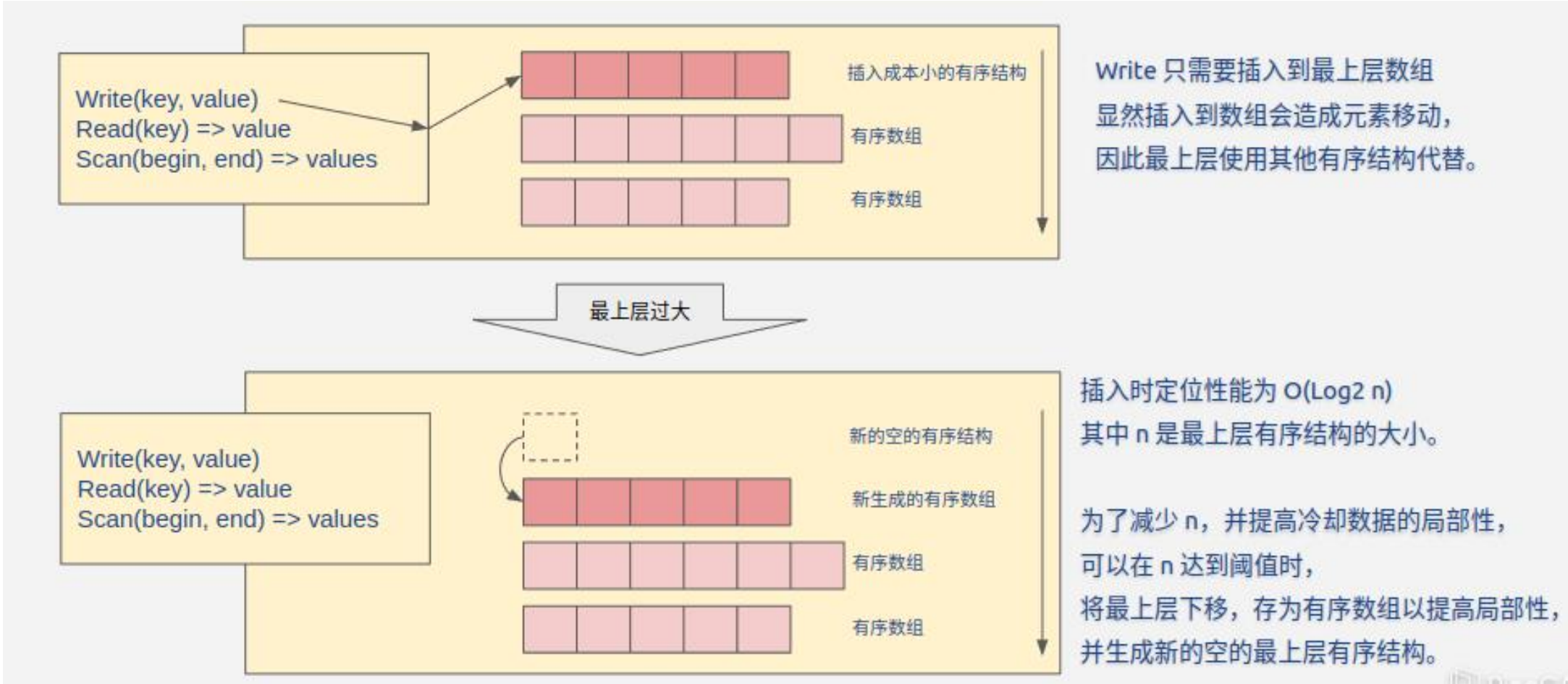
内存版LSM-tree



三、持久型存储引擎

内存版LSM-tree

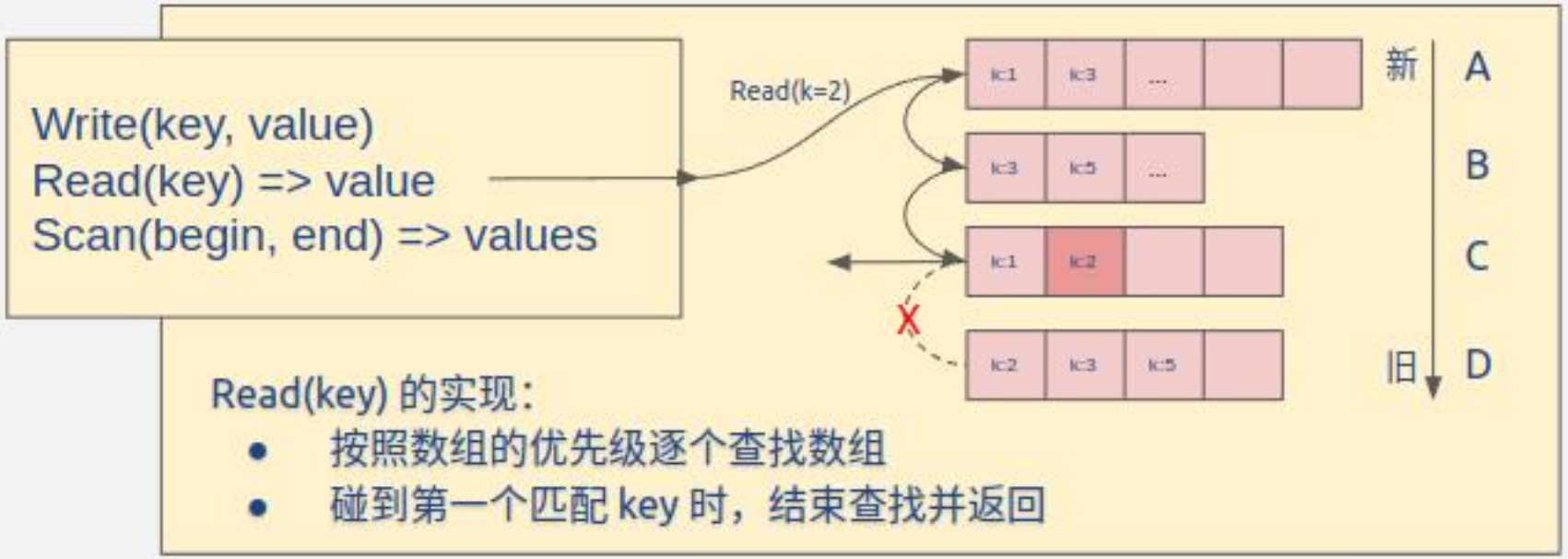
写实现



三、持久型存储引擎

内存版LSM-tree

读实现



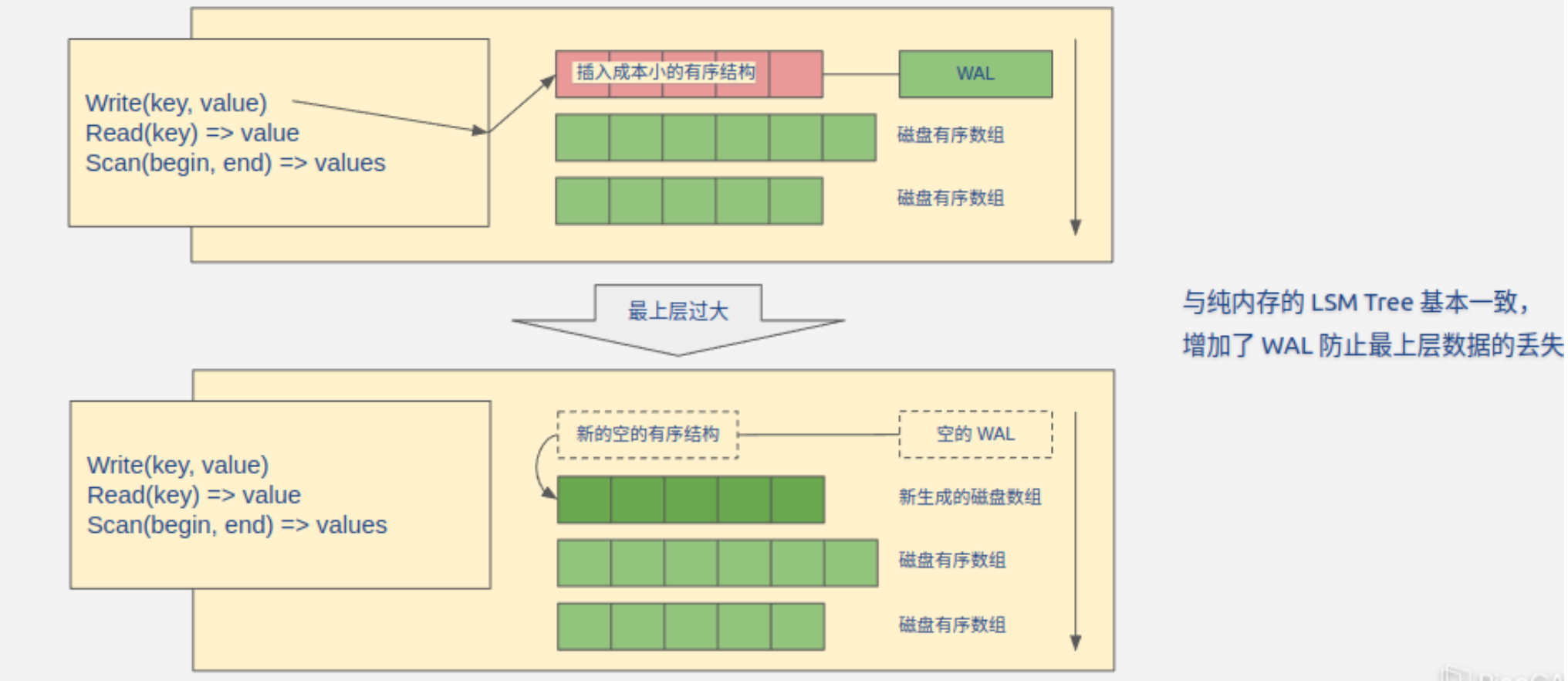
假设总数据量 N，数组个数 K，数组大小较为均匀，则性能：

- 最差为每次都查找到最后一个数组： O(K * Log2(N / K))
- 最佳为每次都在第一个数组命中： O(Log2(N / K))

对数据的读写模式（如果存在）进行匹配，性能可以向最佳情况靠近。

三、持久型存储引擎

磁盘版LSM-tree



三、持久型存储引擎

对比

B+Tree 的由于每段小值域内分别攒批，攒批能力不足，导致的问题：

- 刷脏页写放大
- 小散 IO
- 元数据 OPS 高

LSM Tree 通过全局攒批解决了 B+Tree 的这些问题，带来了新的问题

- Compact 写放大
- 读放大（读时需多路合并）

从中可以看到系统设计中大量的 **取舍与平衡**

三、持久型存储引擎

研究方向

B+tree:
1、元数据管理
2、与非易失性内存的结合

LSM-tree
1、读放大、空间放大、读放大之间的平衡
2、与非易失性内存的结合
3、基于SSD的特性进行优化