

# 宝宝床边故事集：存储引擎

Bedtime Stories For Children: Storage Engines

众所周知，故事不总是真的  
而且经常充斥各种 **未经证实** 的 **想象**



# 快速跳转

## 内存结构推演

- 存储接口定义
- 有序的必要性
- 局部性、静态性能与动态性能
- 内存 B+Tree 存储引擎

## B+Tree 存储引擎设计推演

- 磁盘特性
- WAL
- 攒批与 Compaction
- WAL GC
- B+Tree 存储引擎问题分析
- B+Tree 小结

## LSM Tree 存储引擎设计推演

- LSM Tree 基本结构
- LSM Tree 磁盘数据结构
- LSM Tree 的 Compaction 基本分析
- 一个朴素 Compaction 策略例子
- Compaction 策略设计分析
- Universal / Leveled Compaction 设计推演
- LSM Tree 与 B+Tree 对比
- LSM Tree 业界优化方向

## 不同存储引擎的共性与特性:分类方法

- 值域切割引擎
- DeltaMain 存储引擎
- 时域切割引擎
- 先时域再值域的混合切割
- 概念定义:坍塌
- 另一种分类:延迟坍塌与即时坍塌
- 未完待续

(●'▽'●)!

(..●\_●..)

今天我们来讲存储引擎的故事

存储引擎是什么呢？

存储引擎是管理 **数据集** 的系统



# 定义存储引擎的最简接口

存储引擎是管理 **数据集** 的系统

最简单的引擎只需要 Write / Read 两个方法

基于实际需求, Scan 方法在大多数场景也是必须的

存储引擎的接口

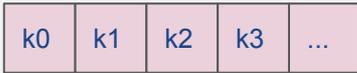
```
Write(key, value)
Read(key) => value
Scan(begin, end) => values
```

我们先看看如何在内存上实现该接口, 分析 **有序性** 和 **局部性** 问题, 再推广至磁盘引擎。

# 暂不考虑 Scan, 来看 Read / Write 的最简实现

## key 连续

key 是连续的, 用 **数组** 紧密存储数据

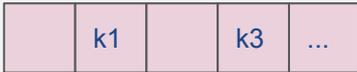


Read(k1) => data[k1] => v1

条件苛刻  
不讨论

## 稀疏数组

key 不连续, 空间充裕, 用 **稀疏数组** 存储



Read(k1) => data[k1] => v1

条件苛刻  
不讨论

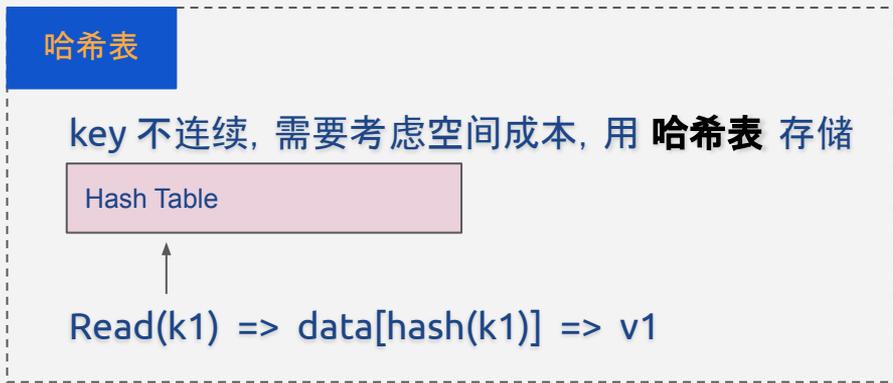
Read / Write 描述的是两集合 keys、values 的单射关系。

最简单的实现, 是使用 **位置 - 内容** 的一一对应关系。

我们只需要一次寻址, 就可以取得 value, 是最快的实现方式: O(1)

但是使用条件太高, 普适性不强, 之后不再讨论

# 用哈希表实现 Read / Write



只考虑 Read / Write, 不需要有序

哈希表是 **位置 - 内容** 方案的延伸。

使用一(位置)对多(内容)的组织方式, 再使用冲突解决手段, 间接实现了单射关系

需要 hash 运算, 然后寻址, 解决 hash 冲突。  
比较快, 基本接近:  $O(1)$

把 key 大小、hash 函数具体实现考虑进来, 综合性能:  $O(\text{key\_size})$

# 用有序数组实现 Read / Write

## 有序数组

key 不连续, 空间宝贵, 用有序数组紧密存储



Read(k) => data[binary\_find(k)] => v

有序数组与刚才的完全不同, 不存在位置与内容的明确对应关系。

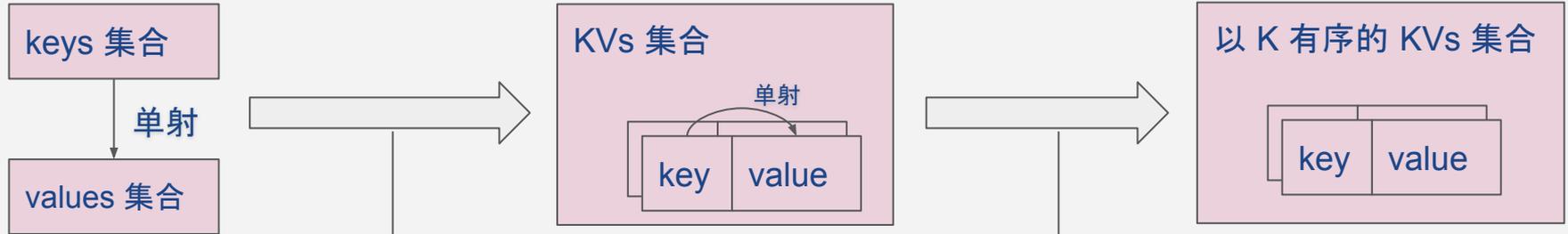
通过二分查找, 确定 key 的存储位置, 就能找到 value, 性能:  $O(\log_2 n)$

考虑 key 大小, 综合性能为:  
 $O(\text{key\_size} * \log_2 n)$

把 **key 前缀相同几率**、**机器字长**、**SIMD** 考虑进来, key\_size 的部分会大大减少

有序数组的 Write 也容易实现, 但成本很高, 等下继续讨论

# 用有序结构实现 Read / Write 的单射关系



两个集合变成一个集合  
集合间的单射关系变成了 KV 合体后的同一个元素  
这使得对元素的操作, 都需要先在集合中的查找到它

将 KVs 集合排序, 用二分查找加速元素查找, 获得较好的  $O(\log_2 n)$  性能  
最简单的有序结构: **有序数组**

这是一个巨大的变化(退化), 从寻址行为, 变成了搜索行为

排序不是唯一的(但是是最简单的)加速查找手段, 例如结合 Bloom Filter + 有序结构

# 来看 Scan 如何实现: 哈希表很难, 有序数组优秀

## 哈希表

key 不连续, 空间很宝贵, 用哈希表存储

Hash Table

Scan(begin, end) => ?

(。\_\_。) 哈希表 Scan 很为难

## 有序数组

key 不连续, 空间宝贵, 用有序数组紧密存储

k=3	k=5	k=7	k=9	...
-----	-----	-----	-----	-----

Scan(begin, end) =>  
data[binary\_find(begin): binary\_find(end)] => values

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:  $O(\log_2 n)$
- 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

(•'•••)!

(..•\_•..)

我们刚分析了有序数组的读操作，有感想吗

这样看来，有序很重要啊

没错，这是我们第一个结论：

**有序性 是实现 Scan 的前提。**

刚才还提到一个局部性，这是什么呢？



## 最简实现的小结

以上讨论了使用 有序稀疏数组、有序紧密数组、哈希表 来实现内存上的存储引擎接口。

**有序性** 是实现 Scan 的前提。

有序紧密数组是较好候选，但 Write 性能会有问题，后面继续讨论。

**局部性** 对性能影响非常大，刚才展示了对 Scan 的影响，接下来继续讨论对 Read 和 Write 的影响。

# 局部性(Locality)

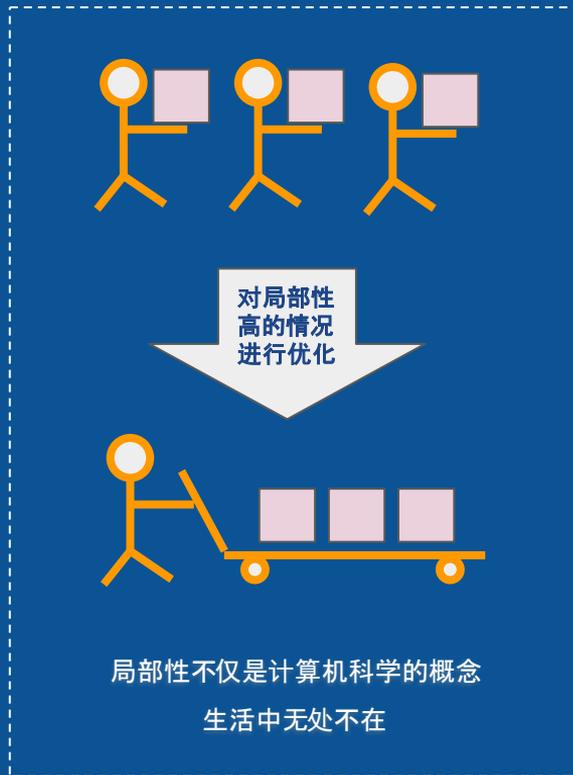
硬件、操作系统等等系统, 绝大部分时候,  
执行 **一次 操作流程** 有额外的开销(**overhead**)。

因此很多部件、模块都设计成:

**连续** 执行 **类似或相同** 的操作、访问 **空间相邻** 的内容时,  
则将多次操作 **合并** 为一次, 或多次之间 **共享上下文** 信息。

这样能极大提升性能。

这种时间、空间上的连续性, 叫做局部性。



# 数据的局部性

我们把 **数据的连续性**及**连续区域大小** 称为 **局部性**，  
把连续存放较多的数据称为 **局部性佳**，  
是追求的目标，能大幅提高性能。

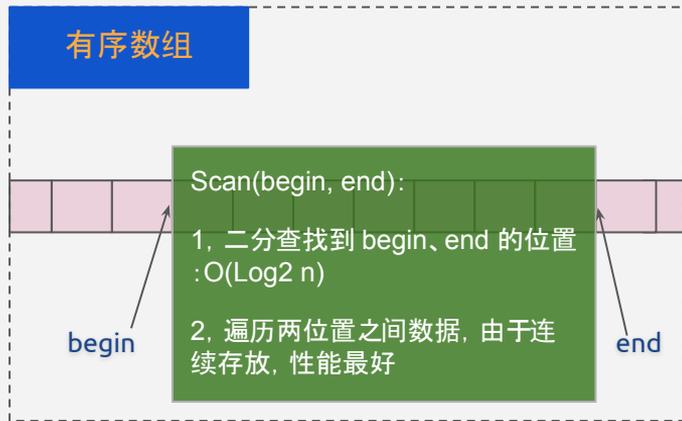
一些受局部性影响的例子：CPU Cache、SIMD、磁盘读写，等等。

TODO: 图

# Scan 的性能与数据 **局部性** 强相关



VS



例如平衡二叉树:

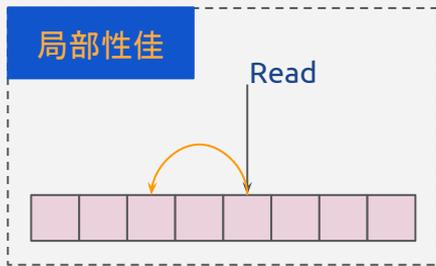
数据局部性差 => Scan 性能较低

# Read 的性能也与 **局部性** 强相关

Read 利用有序性来进行数据二分查找, 将产生多次数据访问。

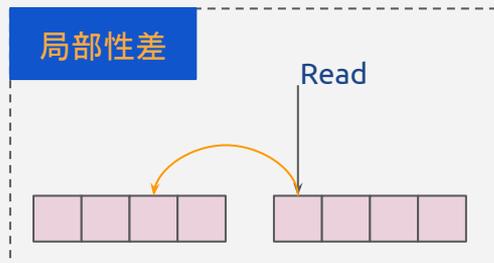
在局部性高的情况下, 数据可以以更少的次数(例如一次性)准备好访问, 从而提高 Read 性能。

具体的性能量化对比, 由 **数据的准备操作的 overhead 消耗占比** 决定。



示例总成本:

- 需要一次数据准备, 可能为:
  - 从磁盘加载数据到内存
  - 从内存加载到 CPU Cache
- 需要两次数据访问



示例总成本:

- 需要两次数据准备, 是左图消耗的两倍
- 需要两次数据访问, 与左图相同

(●'∩'●)∕

(∩∩∩∩∩∩)

这样看局部性应该是越高越好吧

并不是越高越好

我们来分析一下存储引擎的 Write 操作  
就会发现数据布局的局部性过高的问题



# 局部性 过高的性能劣势

从前面 Read 和 Scan 的分析, 可看到局部性高, 性能就更好。

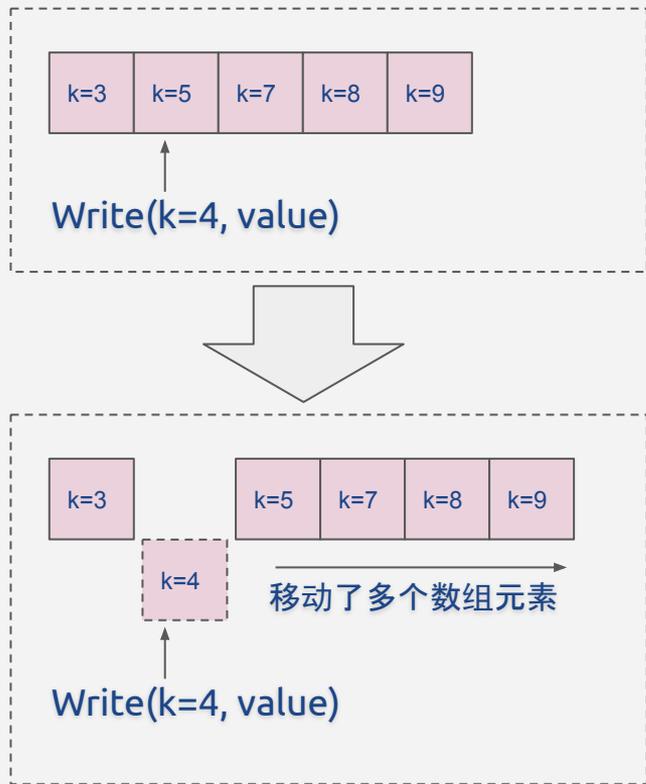
但局部性并不是越高越好,

接下来分析一下存储引擎的 Write 操作,

会发现数据布局的局部性过高的带来的性能问题。



# 有序数组的 Write 实现



(~^~)

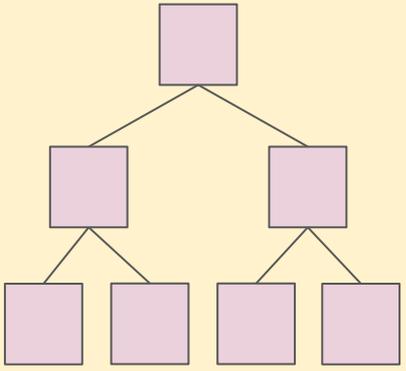
可见 Write 操作移动了大量数据，  
数组越大，Write 可能移动的数据量就越大，  
这存在性能问题。

(•'••)!

刚才分析过，**有序结构** 就可以实现我们的接口，  
不需要是有序数组吧

# 不用有序数组，使用二叉平衡树来实现

Write(key, value)  
Read(key) => value  
Scan(begin, end) => values



Write: 二分查找插入, 保持树平衡,  $O(\log_2 n)$   
Read: 二分查找,  $O(\log_2 n)$   
Scan: 二分查找 begin, 再遍历数据至 end

- 定位性能  $O(\log_2 n)$
- 遍历性能, Emmm ...

Read 性能良好, 二分查找

Write 性能良好, 重平衡过程没有大块数据移动

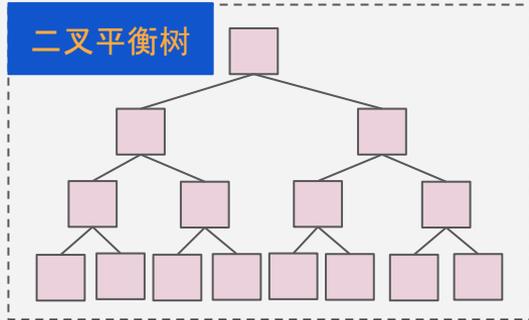
由于每个元素都在不同的存储空间上,  
**局部性非常差**, 读性能 (Read / Scan) 较差

每个元素都要付出额外的指针存储空间,  
**空间 overhead 很大**

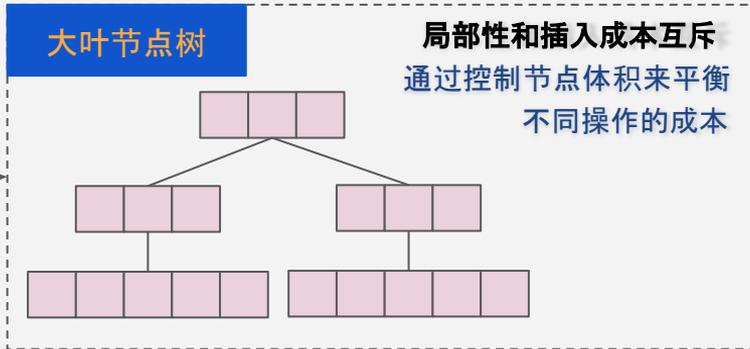
(●'~'●)/ 这和有序数组正好相反呢

# 把数组和二叉树结合(折衷)一下

从有序数组的角度看，  
我们把大数组分割成了一个小的有序数组，  
再用另一种有序结构把小数组组织起来，  
使得 Write 时的插入操作，  
**移动数据量** 减少并且可控



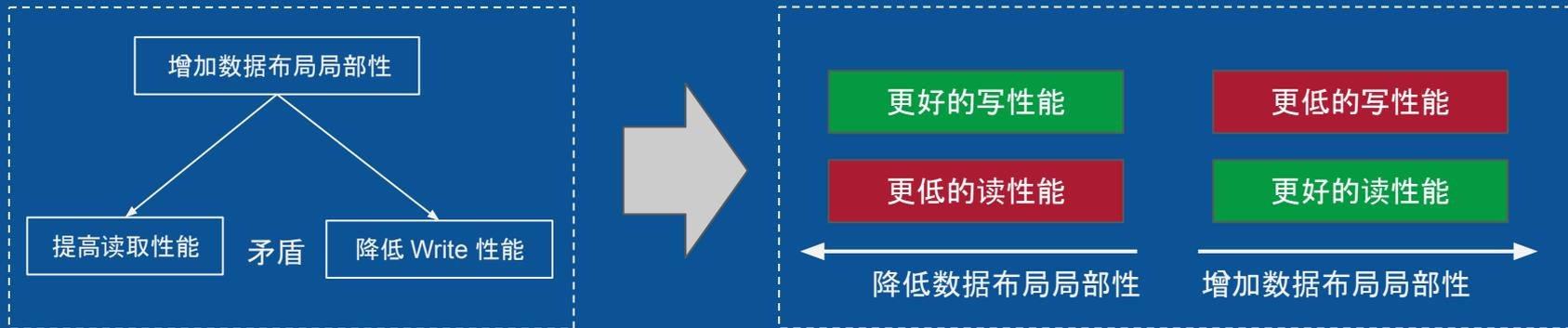
从树的角度看，  
用一个个小的有序数组代替元素作为节点，  
大大 **增加了局部性**，减少了存储 overhead



# 数据布局的局部性与 Write 性能

数据布局的局部性越好，数据的增删操作造成的移动成本就越高。

容易发现：



因此，数据布局的局部性与 Write 性能的矛盾，也就是 **静态性能** (Read / Scan) 与 **动态性能** (Write) 的矛盾。

# 局部性的量化

从刚才的例子可以看出，数据布局的局部性和其他设计目标可能 **冲突**，并不是越高越好。

那什么样的布局局部性最恰当呢？

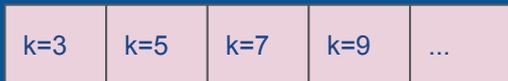
能均摊 **流程** 的 **overhead** 到可接受程度的最小局部性，是比较合适的。

再小会使得流程 overhead 过大，再大可能会削弱其他目标。

除了与 Write 性能冲突，局部性还可能与其他更多的目标冲突，

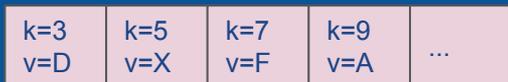
我们来看一个键值分离的例子，观察这些冲突。

# 键值分离存储：布局局部性 **变差**，但写放大 **降低**



这是我们刚才的有序数组的简要示意，省略了 value

补全  
细节

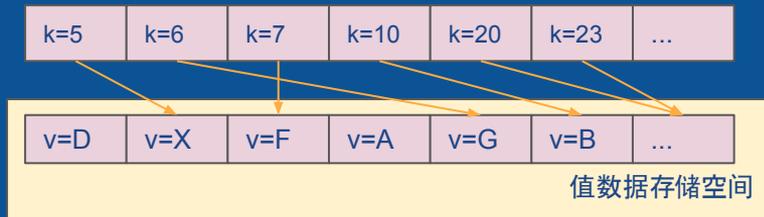


将 key value 都展示出来是这样的

## 取舍

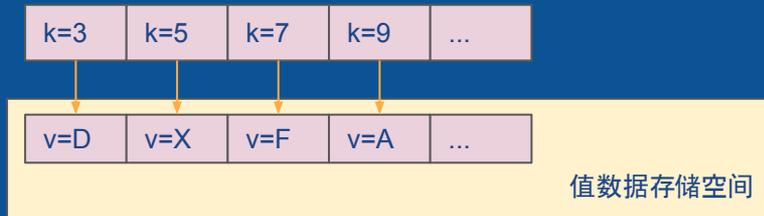
在 value 较大的时候可以考虑键值分离，较小时 Scan 性能下降

键值分离



修改数据不需要移动 value，写放大降低，但 value 变得不连续

数据  
增删



键值分离存储，使用指针实现单射关系

# 复杂流程的局部性的量化

如果数据流经多种或多个流程, 最佳局部性 为:  $\text{Max}(\text{每个流程的最佳局部性})$

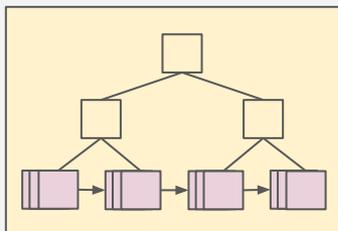
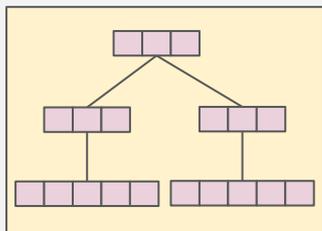
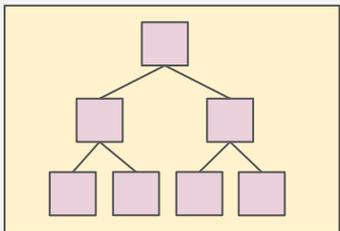
通过分析, 可以获得局部性的量化计算方式,

通过测试, 可以获得每个流程的具体数值,

最终, 获得 **具体的 局部性 要求**。

例如: 使用 fio 工具测试具体的磁盘, 配置不同的 bs 参数测试, 获得合理的 IO min size

# 在多种数据结构中选择



更多其他结构

除了二叉树, 还有红黑树、跳表等很多类似的结构。

它们的设计和实现不同, 但是基本目标都一样: 可以 **低成本维持数据有序**

我们综合多种设计需求来选择数据结构, 其中局部性的高低是重要的指标:

- Scan 多且返回较大的数据集, 选择局部性高的结构
- Read 操作中的数据准备过程的 overhead 占比较高, 选择局部性高的结构
- 其他设计目标

# 选定 B+Tree

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

可以看到, **叶节点在新 key 写入时依旧需要移动数据**, 造成写放大

我们可以通过叶节点大小的配置来进行平衡:

叶节点大:局部性高

- 插入成本高, 慢
- 读取性能高, 快

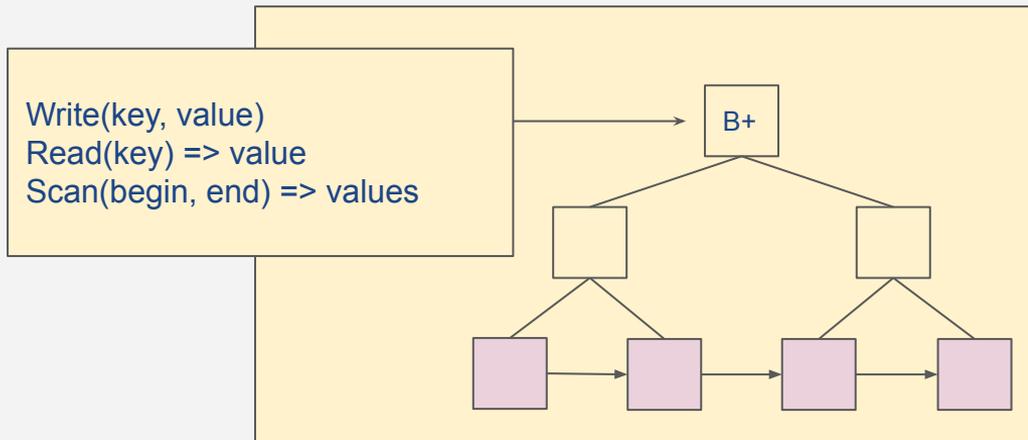
VS

叶节点小:局部性低

- 插入成本低, 快
- 读取性能低, 慢



成果: 选用 B+Tree, 实现内存存储引擎



今天的故事就到这里结束了



# 在内存实现存储引擎接口的小结

- 存储引擎, 是个有序结构
  - 有序性是实现 Scan 的必须
  - 局部性很重要, 但不是越高越好, 和 动态性能矛盾
- 低成本、动态地保持数据有序, 这是设计要点

这些规则很简单, 在内存、磁盘等更多的存储设备都适用

它们都是 **一维线性存储空间**

# 动与静的平衡

设计一个能高速读取的 **静态结构** 是相对容易的，

难的是 **动态** 保持良好的结构，如前分析，这两者相斥。

单纯以 **读取** (Scan / Read) 来衡量引擎性能是不全面的。

例如我们一开始讨论的有序数组，

它是非常快的静态结构，但 **动态性能** 很差。

动与静的平衡，是对数据组织的 **局部性高低** 的取舍的体现。

# 量化一切

对事情进行量化,可以帮助我们更好地理解、设计。  
也可以更准确地对过程进行推算、预估。

一些基本的量化数据可以通过测试、经验得知,  
进一步的量化需要仔细地分析事物之间的关系,找到当中的数学关系。

有条件的话还可以对所思所得进行 **验证**,  
正确的量化意味着对事物的理解脱离了感性认识。

## (第二天)

(●'∩'●)∩

(..∩\_∩..)

昨天我们讲了在内存里如何做一个简单的存储引擎  
今天我们来讨论一下存盘的事情



# 在磁盘实现可持久化的存储引擎

最简单的策略，是以前面讨论的内存存储引擎为基础，在磁盘、内存使用结构完全一样的同构镜像，每个写操作都同时操作两个镜像，读操作仅读取内存镜像。

然而考虑到性能，是完全不可行的，以下展开分析。

以下内容，用颜色区分内存和磁盘数据

浅红：内存里的数据

浅绿：磁盘上的数据

# 磁盘性能特征：IOPS 低，远低于内存

一次磁盘访问，是很高的成本

- 古老的 HDD 只能支持每秒几百次访问
- 普通 SSD 约 5000 次每秒
- 最新的 SSD (典型情况是 PCIE 接口, NVMe 协议) 已经支持每秒上百万次, 但还远远不能和内存比

当数据局部性差时:

- 需要更频繁地访问磁盘
- IOPS 比 IOBW 先达到上限, 性能差

当数据局部性好时:

- IOBW 能达到硬件上限
- IOBW 达到上限是理想的最好性能

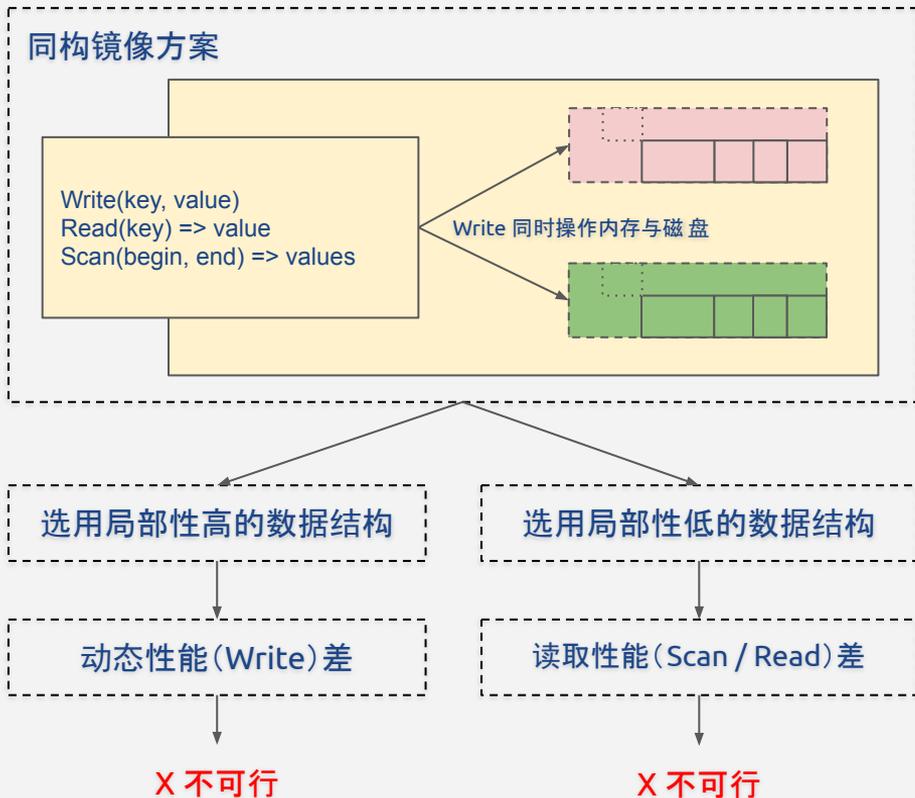
# 磁盘性能特征：连续写入比随机写入快很多

- 磁盘的连续写入，比随机写入快非常多
  - HDD 的机械结构和寻道操作影响
  - 操作系统对连续写入有优化 (IO 调度策略)
  - 在 SSD 上也成立

意味着在写入数据的局部性好的情况下性能更好

TODO: SSD 的随机顺序性能对比图, RocksDB 论文

# 综上，磁盘与内存差异巨大，同构镜像不可行



从分析我们知道，  
磁盘与内存性能差距巨大，  
局部性要求也高很多。

如果与内存做成 **同构镜像**  
会大大拉低整体性能

# 数据结构的设计源于局部性的需求

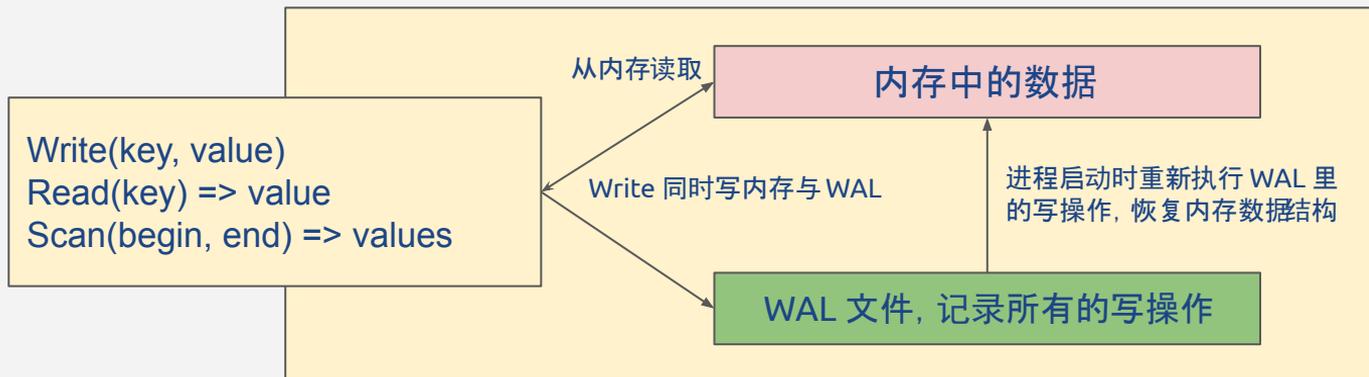
由于内存与磁盘对局部性需求的差异巨大，

我们可以看到内存数据结构和磁盘数据结构的截然不同的构成方式。

# 解决方案:WAL

WAL(Write Ahead Log)是 **异构镜像** 方案

- 异构:磁盘与内存的数据结构不一样
  - 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



# 什么是 WAL

Log 文件，记录所有的写操作

WAL (Write Ahead Log)  
或者叫 Redo Log

写 WAL 都在末尾追加写入，顺序地记录所有修改动作

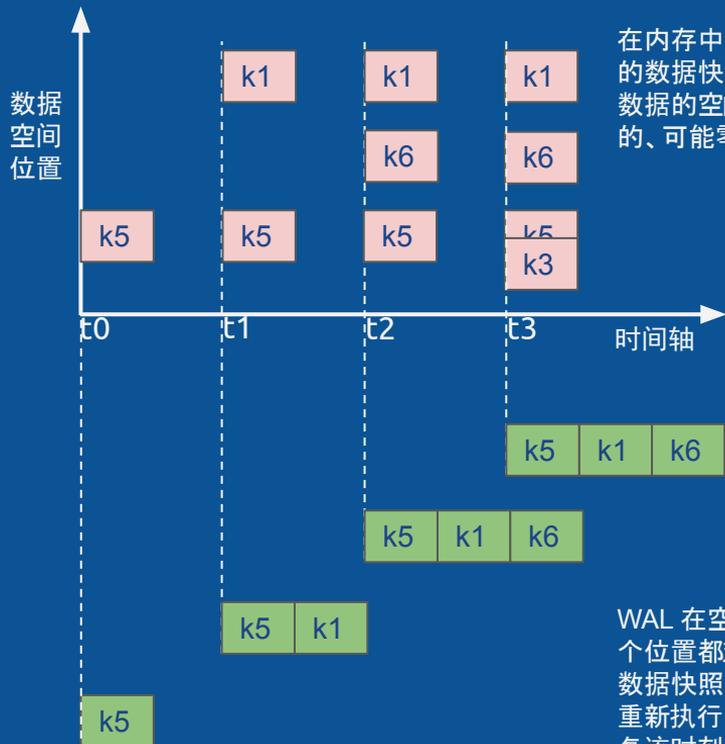
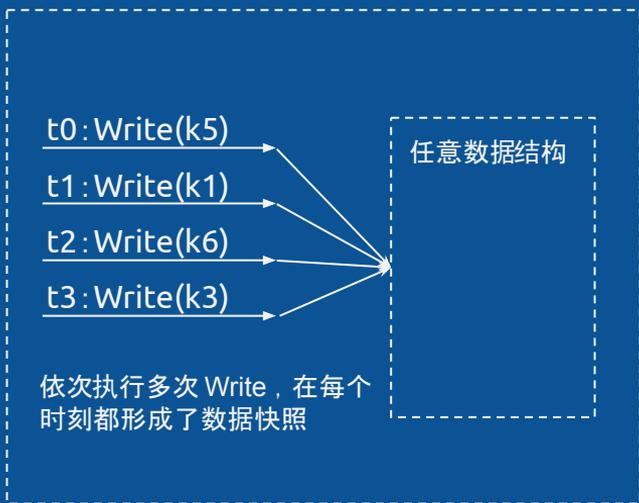
为了存盘数据的安全，避免进程非正常退出丢数据，  
WAL 一般每次写完数据都执行 fsync 操作，  
否则数据可能还留在操作系统的 Page Cache 中没有写到盘上

调用 fsync 消耗大量磁盘 IOPS 资源，

可能成为 **性能瓶颈**

大部分数据库都有选项，允许 WAL 写入时不立即 fsync，而在后台周期性进行。存在数据丢失风险

# 在时间轴上观察 WAL



在内存中每时刻形成的数据快照。数据的空间分布是任意的、可能零散的

WAL 在空间上连续，每个位置都对应某时刻的数据快照。重新执行 WAL 可以恢复该时刻的数据

# WAL 的简单实现



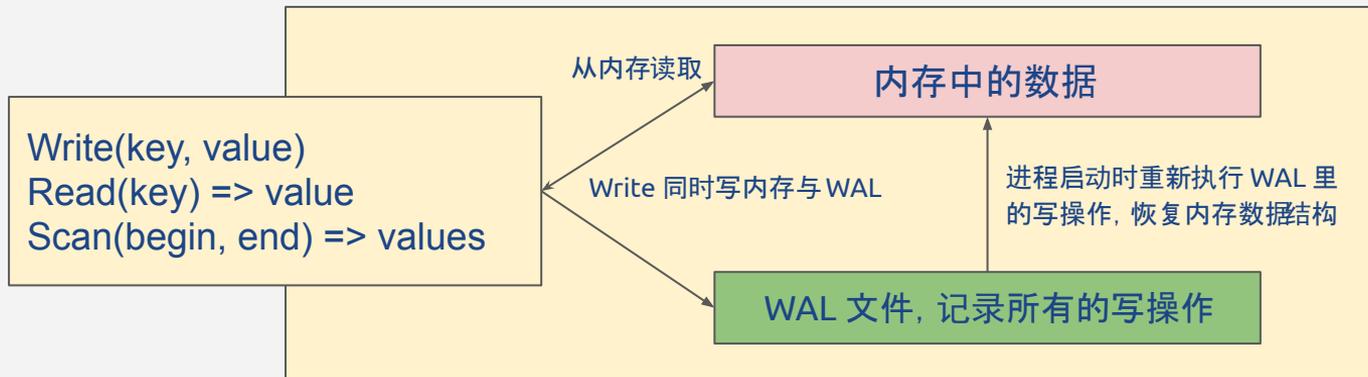
每个 Record(或者叫 Entry)可以包含一到多个数据变更操作

每个 Record 单独记录 Size 和 Checksum

这样, 当读 Log 文件恢复的时候, 如果 Size 异常或者 Checksum 不匹配, 我们就知道这个 Record 损坏了(可能是磁盘损坏)

可知, Record 是 WAL 的原子性操作的单位

# 使用 WAL 实现存盘与恢复

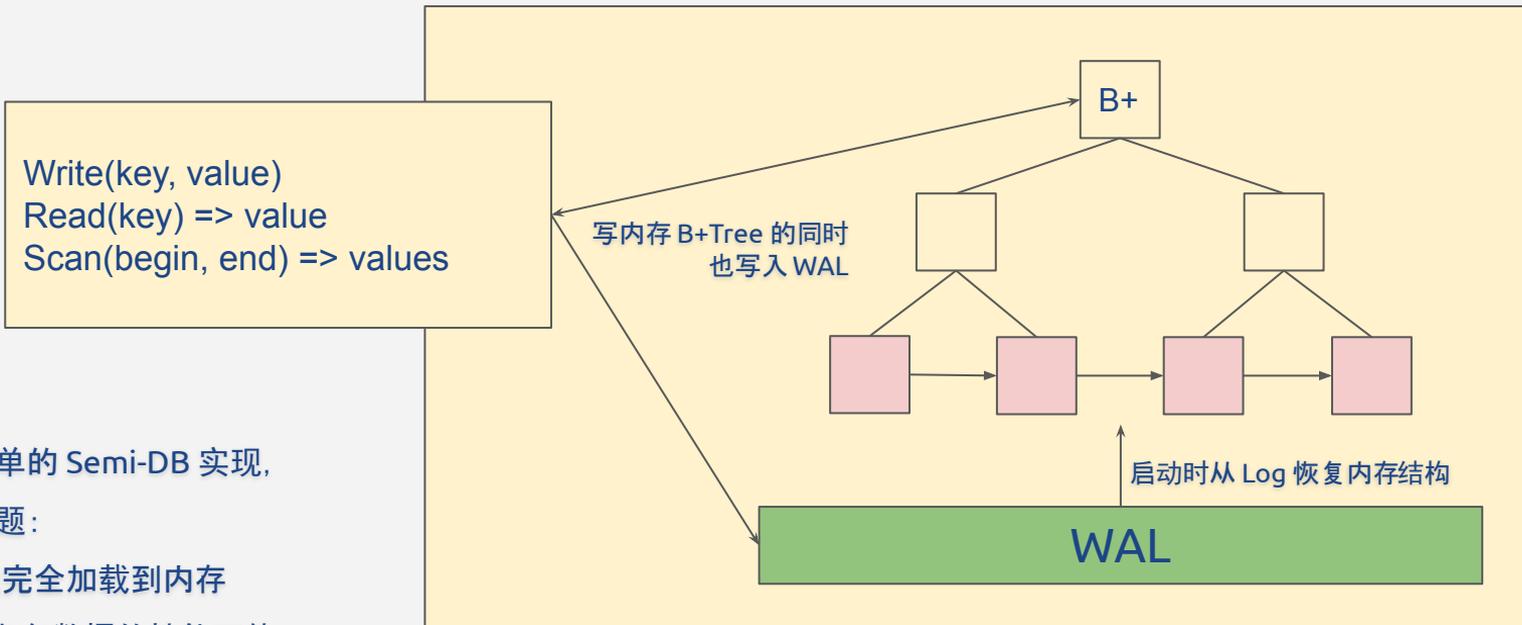


这种 **异构镜像** 的结构, 适用场景是:

- 数据读取延迟要求高, 需要从内存中直接可以获取
- 数据集较小, 可以完全加载到内存
- 举例: 元数据

这个结构很简单, 但应用广泛。为了方便, 我们叫它 **Semi-DB**

# 基于内存 B+Tree 添加 WAL, 实现持久化引擎



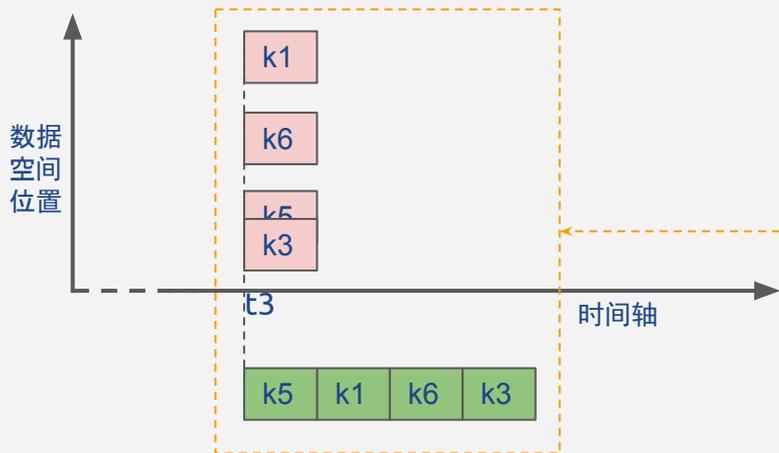
这是一种简单的 Semi-DB 实现,  
存在两个问题:

- 1, 数据只能完全加载到内存
- 2, 从 WAL 恢复数据的性能不佳

# WAL 不是一个高性能的存盘结构

Semi-DB 在运行过程中, WAL 不停增大, 在重启进程从 WAL 重放操作的过程中, 存在一些问题:

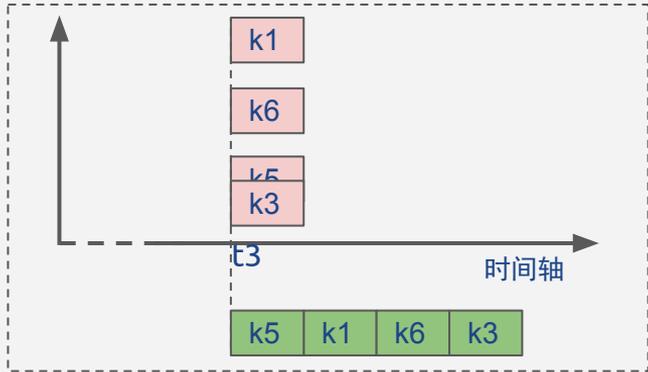
- 重放 WAL 主要是重做 Write 操作, 不一定高效
- WAL 中可能存在相同 key 的多次 Write 的多个版本的数据, 占用了额外空间, 也降低重放性能



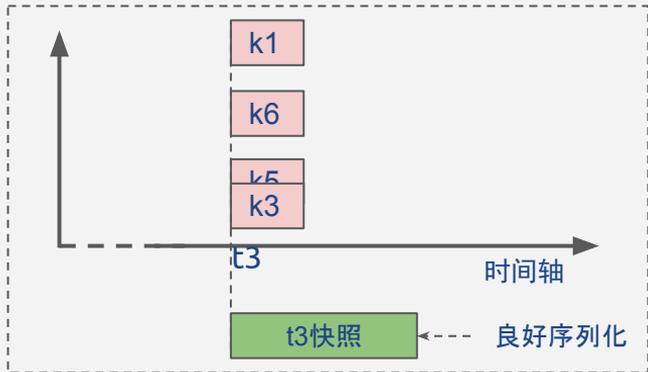
**原因:**

WAL 等价于某时刻数据的快照, 是它的序列化(转化为空间连续的数据)结果, 但不是最优的序列化方式。

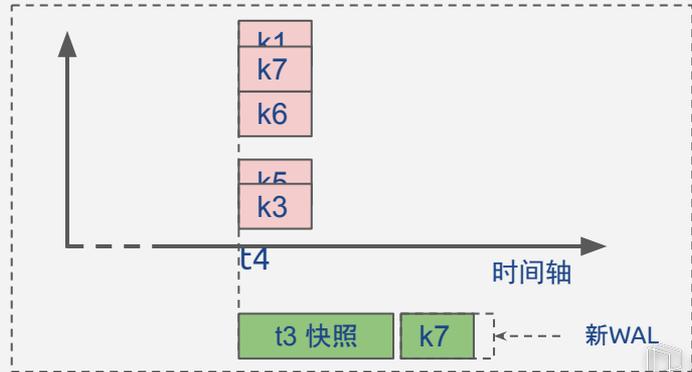
# 整理 WAL 让它更高效



整理



新数据写入  
t4: Write(k7)



整理: 将 t3 时刻的数据良好地序列化, 成为数据文件。

重启时重放数据的过程:

- 反序列化 t3 快照文件
- 重放体积较小的新 WAL

重启时的恢复性能大大提高了



# 攒批 整理，让存盘数据更高效

可见，良好序列化的文件，是正式的存盘数据，

WAL 性能太差，只是过渡。

我们把写往 WAL 和内存、但还没写成正式文件的过程，称为 **攒批**

**攒批：**

将逻辑上局部性低的数据，暂时以局部性高的方式(WAL)存起来，

等攒批足够大、可以整理为局部性高的数据时，整理写盘为正式数据。

从而使得每次**写盘行为** 都有良好的**局部性**。

攒批过程付出了额外的写放大，换取了局部性更高的磁盘 IO、数据布局。



# 攒批 **整理**，让存盘数据更高效

整理带来了额外的写盘操作，也就是写放大，执行时机需要谨慎设计。

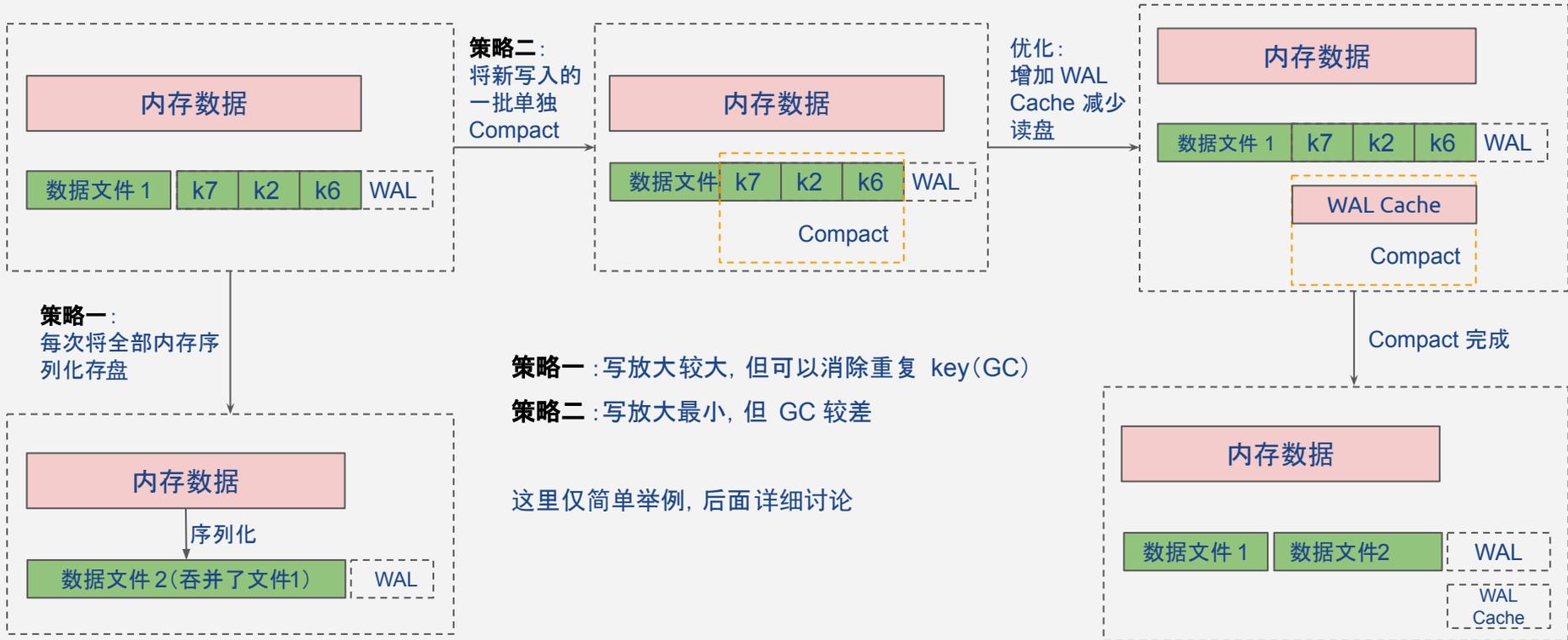
整理的过程在不同的系统里有不同的名字：Compact、Merge、刷脏页，等等

为了方便，我们统一叫做 **Compact / Compaction**

Compaction 策略有较大的设计空间，随需求而变。



# 简单的不同 Compaction 策略的选择举例





# Compaction 积极性 取舍

Compaction 的目标是以写放大换取更高的局部性, 提升静态性能。

积极: 更频繁, 更大范围地进行 Compact



(●'∩'●)!

(..∩\_∩..)

这样我们就实现了存盘啦

但是把所有数据都加载到内存, 大了就放不下了呀

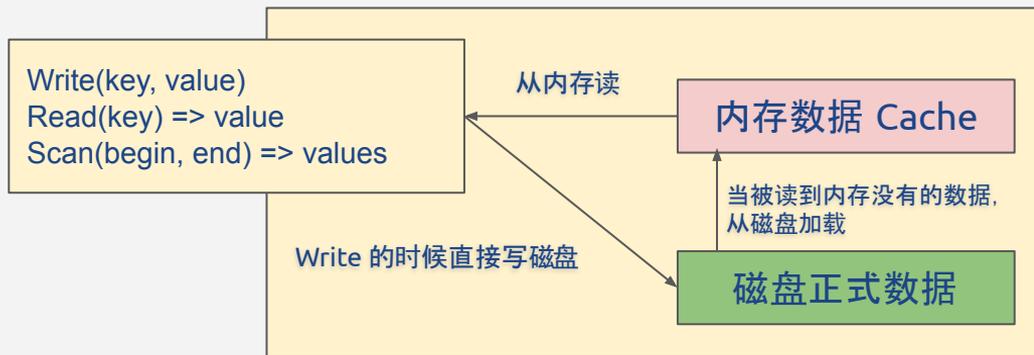


# 解决数据容量大于内存容量的问题

上面对 Compaction 的讨论, 提供了解决 WAL 性能的方案, 同时, 需要在写放大与读性能之间进行平衡取舍。

以下讨论如何解决 Semi-DB 只能完全加载到内存的问题。

方案很简单: 内存只加载部分数据, 作为 Cache。磁盘上存放所有数据。





# 选择数据结构: B+Tree

当每次从磁盘读数据到内存, 也需要较好的局部性, 避免读取慢。

也就是说, 需要磁盘上的数据组织是 **局部性高的有序结构**。

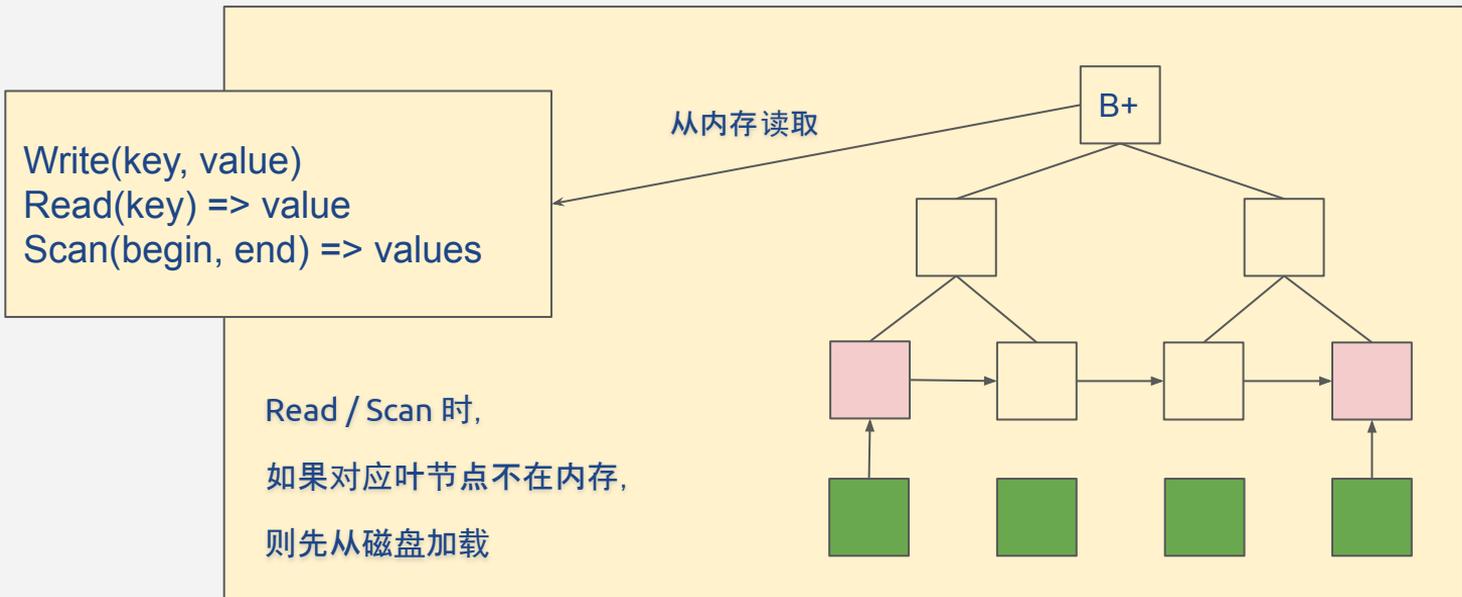
B+Tree 的数据局部性很好, 符合要求。

另外, B+Tree 的叶节点 **大小均匀可控**, 可以把大小设为 4KB 的倍数, 使用 Direct IO 绕开操作系统的 Page Cache, 加速 IO。

方案: 以叶节点为单位, 将被读写访问的叶节点镜像到内存。

# B+Tree 部分镜像至内存

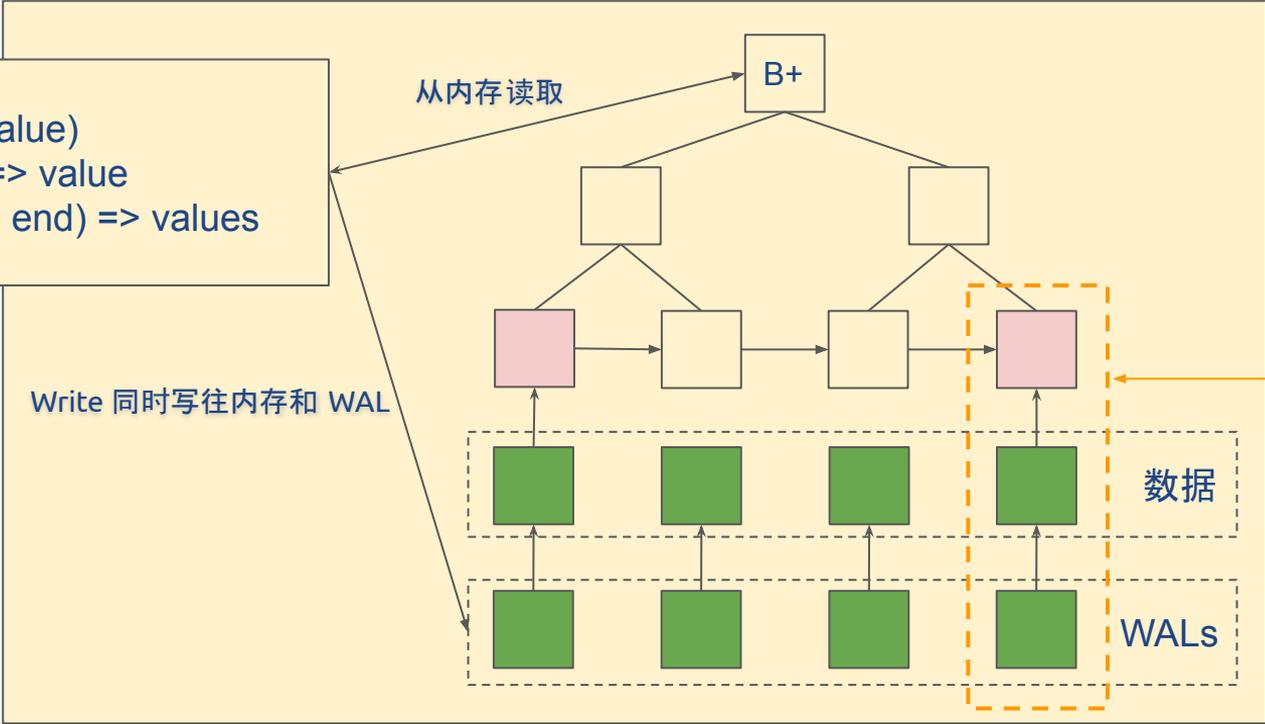
尝试方案



# 为每个叶节点启用 WAL

尝试方案

Write(key, value)  
Read(key) => value  
Scan(begin, end) => values



Write 同时写往内存和 WAL

每个叶节点形成一组 Semi-DB 结构, 使用内存数据全序列化存盘的 Compaction 策略

(●'∪'●)∕

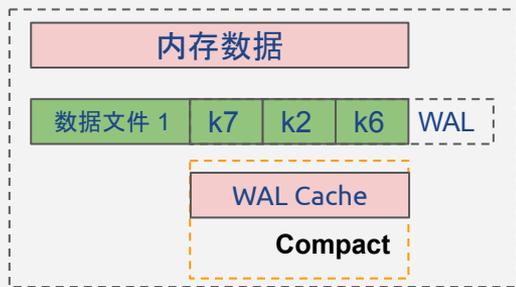
(..∪\_∪..)

每个叶节点一个 WAL？那该有多少 WAL 啊

是的，这会有问题，但也容易解决

# 每个叶节点启用 WAL 将会导致 WAL 过多

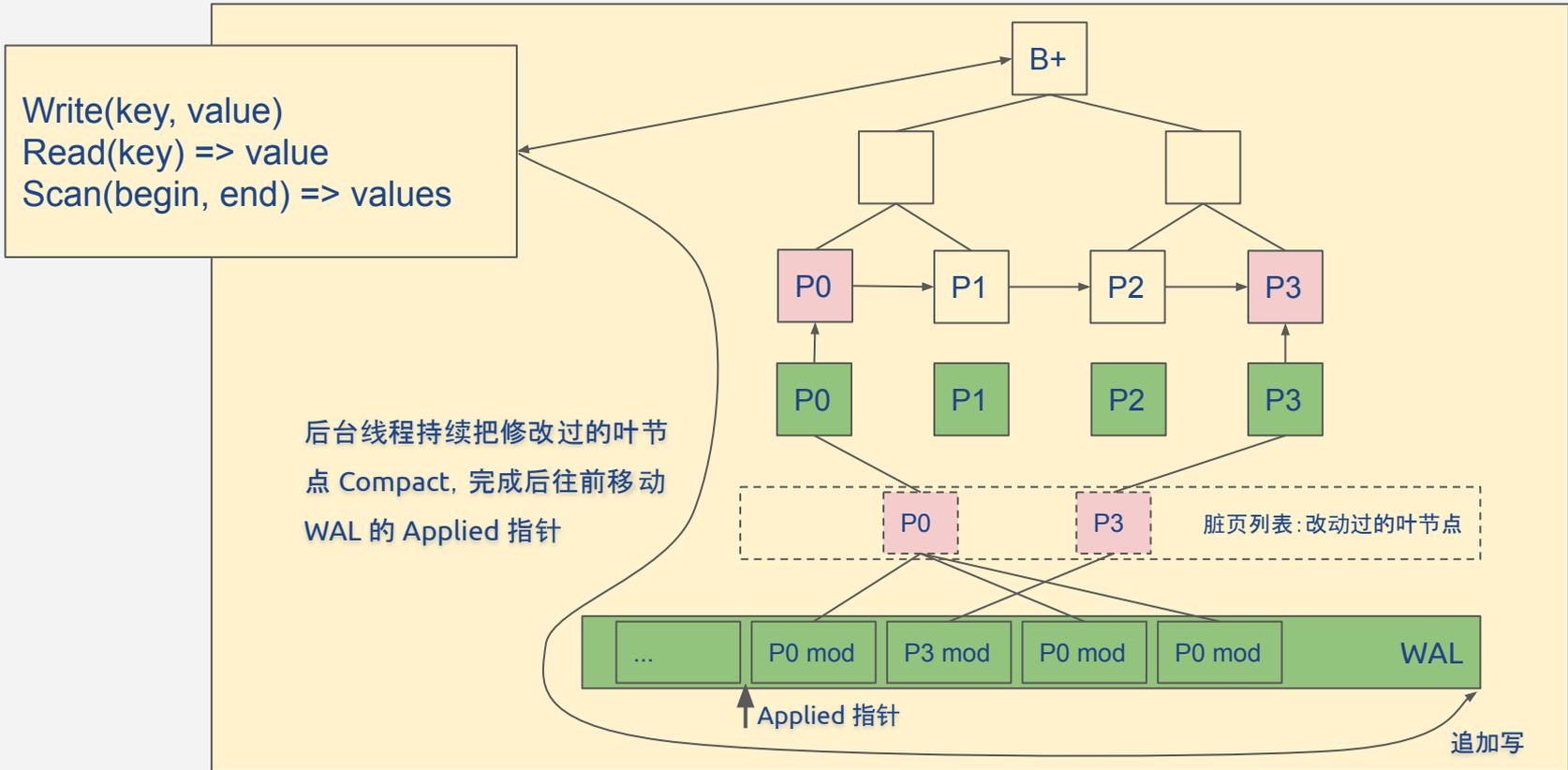
方案问题与解决



如图, Compact 时数据都可以从内存获得  
当 Compact 积极时, 从 WAL 读取数据的机会很少  
那么, 可以把所有叶节点的 WAL 合并起来。

这可以解决 WAL 过多的问题, 但会带来 GC 问题, 下面继续分析。

# 将所有叶节点的 WAL 合并到一个，让局部性更好

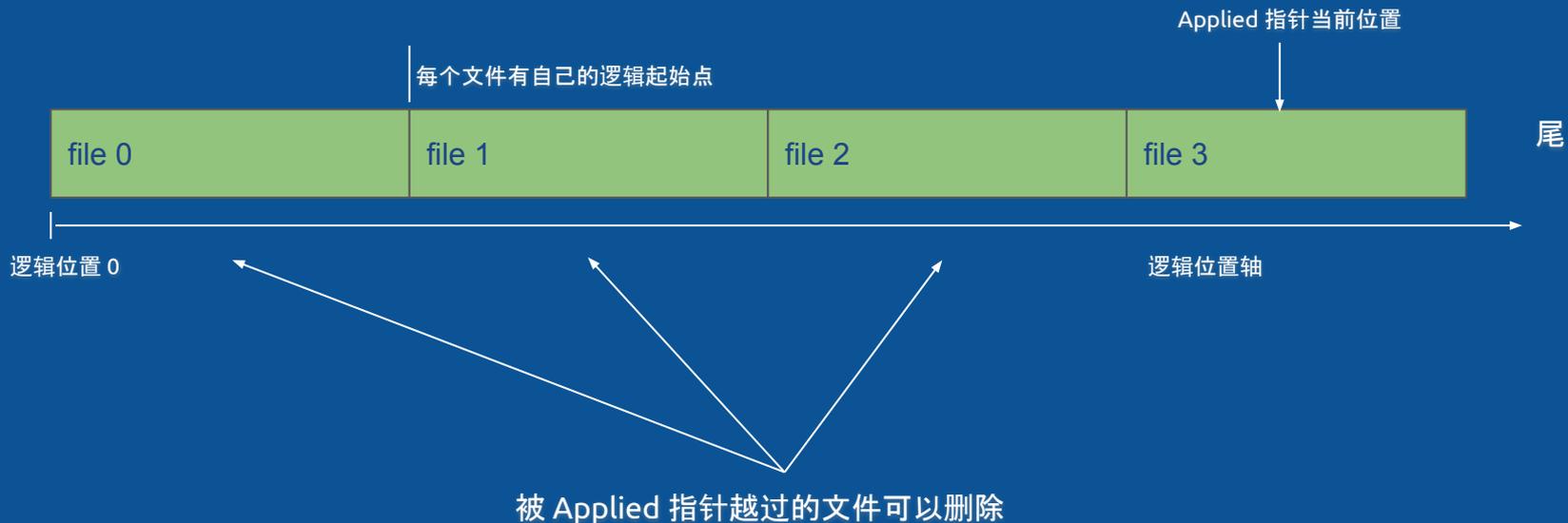


# 使用全局 WAL 的空间回收问题 (GC, Garbage Collection)

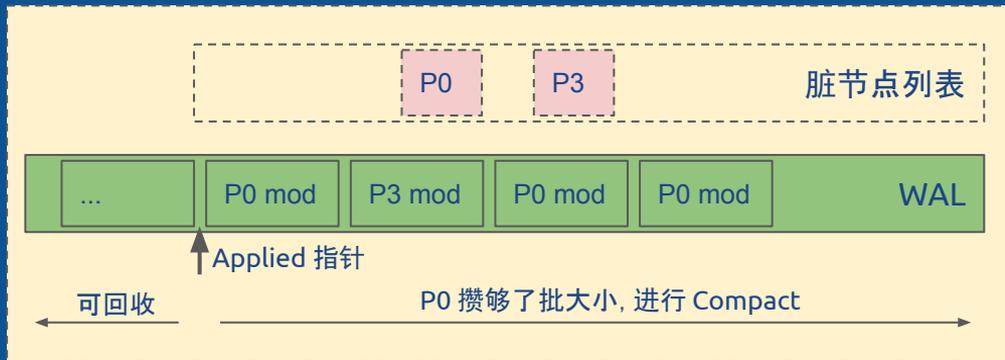


Applied 指针标记了在此位置之前的数据没有意义，  
其占用空间可以回收

# WAL 的 GC 的简单实现



# WAL 的 GC 是个复杂问题

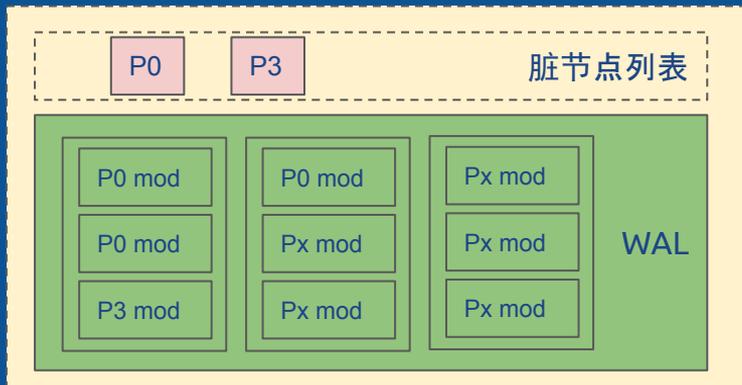


可以看到, P0 攒够数据准备 Compact 时, P3 才只有一点点数据。

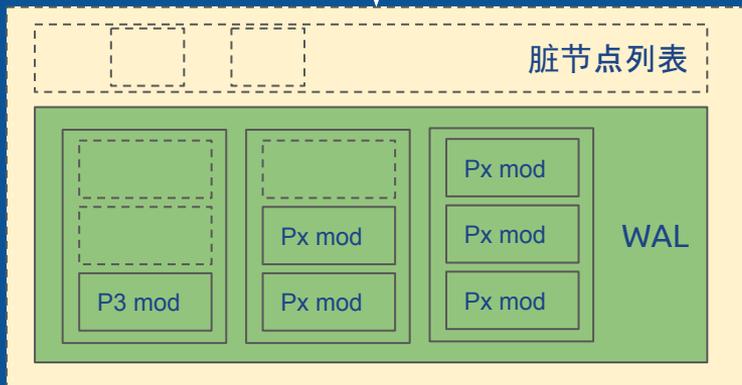
此时有两类选择:

- 尽量不执行攒批不足的 Compact, 容易造成内存、WAL 存储空间资源释放慢, 最终由于资源不足强制 Compact
- P3 执行 Compact, 会带来较大写放大。好处是可以向前移动 Applied 指针, 释放资源

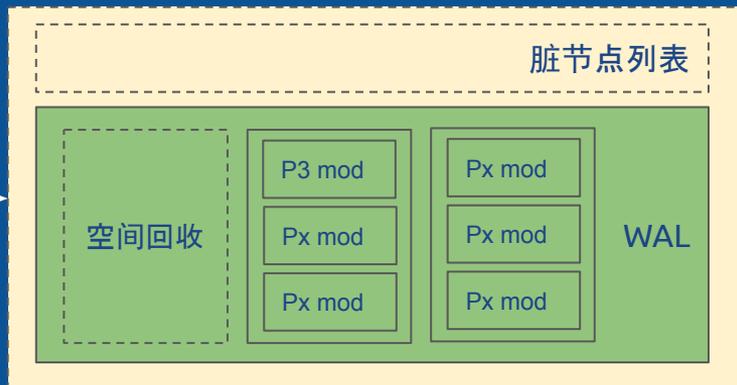
# 解决方案: 不将 WAL 作为一维线性空间



Compact



页面调整



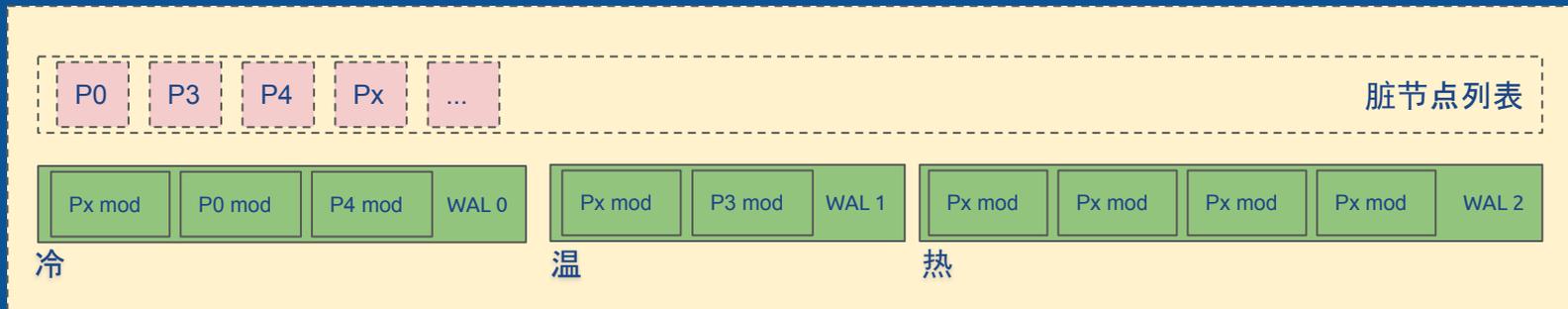
使用页面管理策略:

- 将 WAL 分为多页, 每页存储多个数据项
- 如图示, 进行分配、迁移、回收

在 TiFlash 的新 DeltaMerge 引擎中使用了该策略

在 TiKV 中使用了另一个小型数据库来存储 WAL, 达到同样效果

# 可能的解决方案: 分离冷热节点到不同的 WAL



问题的根源是数据的不同节点接受到的写入量不同, 我们称为冷热不同。

我们可以将不同冷热程度的节点分到不同的 WAL 组。

当一个 WAL 里涉及的叶节点有接近的冷热程度, 那么它们的攒批速度也接近。

当一个叶节点的攒批大小超过阈值的时候, 同 WAL 内其余的叶节点也接近就绪。

从而减少写放大。

# 设计成果：B+Tree 存储引擎

经过以上推演，实现了粗糙但可用的 B+Tree 存储引擎。

和内存版本一样，目标不变：**低代价保持有序结构**

- 将有序结构切割成小段，降低写放大，但保持一定的局部性
- 使用多种手段拉平内存和磁盘之间的性能特性的鸿沟：
  - 用 WAL **攒批**，降低写放大，降低 IOPS
  - 通过 **Compaction** 获得更高效的存盘数据，付出了写放大的代价

# (新的一天)

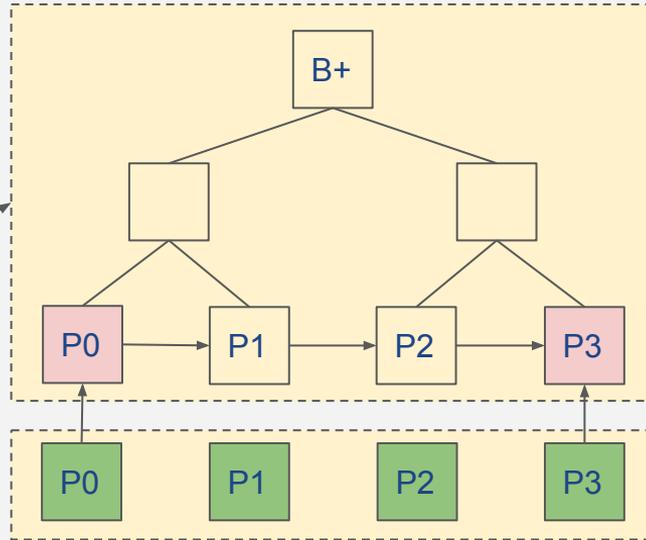


昨天我们实现了一个 B+Tree 存储引擎  
它和业界的结构已经很接近了  
想到什么地方可以改进了吗？



# B+Tree 引擎

内存中的 B+Tree, 可以看作读缓存, 叶节点是磁盘上叶节点的镜像。



所有叶节点存储在磁盘, 我们的持久化数据。

叶节点大小固定, InnoDB 使用表空间来管理。

脏页(节点)列表, 以最早修改时间排序的 FIFO。

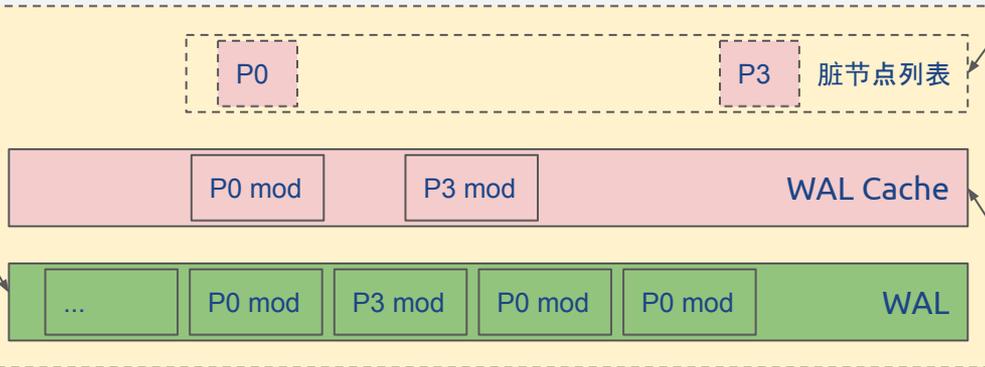
内存资源不足时也可以释放掉, 被访问时从磁盘加载, 再重做 WAL。(影响性能)

Compact 时将脏节点写盘, 不需要读 WAL。Compact 完后移动 WAL 的 Applied 指针(InnoDB 叫 CheckPoint)

WAL, 防止数据丢失。

如果 Applied 指针不能及时前移, 可能会导致空间用尽, 就只能被迫刷脏页。

在 InnoDB 中使用环形空间。



WAL 的缓存。内存资源不足的时候也可以释放掉, 需要时再从 WAL 加载。(影响性能)



# B+Tree 引擎的读性能

Scan / Read:

- 在 B+Tree 中定位叶节点, 性能接近  $O(\log_2 n)$ , 较快
- 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
    - 有磁盘 IO, 及读放大

读缓存是否命中(叶节点在内存)对性能影响巨大

- 所有存储引擎都会碰到这个问题, B+Tree 没有明显劣势
- Scan 在大部分不命中时
  - 小范围 Scan 性能较佳
  - 大范围 Scan, 涉及叶节点较多时, 局部性不佳, 性能较差

# 读缓存

缓存的根本原因是成本：

- 最快的存储硬件单位造价最高，因此容量最低
  - 如果有又快又便宜的，那会淘汰其它产品，直到更快的出现
- 成本导致存储分层：本地 HDD - 本地 SSD - 内存 - CPU Cache

慢速层成本低，容量大。高速层成本高，容量小，高速层可以看作低速层的 **缓存**

可以预计随着硬件发展某些产品成本降低，存储分层依旧会存在，以降低总成本

# 读缓存的命中率

读缓存的命中率首要决定因素是 **低速层 - 高速层** 的 **容量比**：

- 高速层相对容量越低则命中率越低
- 我们的 B+Tree 以内存作为缓存，容量远小于磁盘

假设以下情况，命中率不高，即使改 进缓存算法也较难提升：

- 缓存算法是 LRU
- 对存储引擎进行 key 的全值域的随机读

可以看出，缓存算法就是对输入(读或写)的 **模式预测**：

- 如果预测准确，可以做出 较高的命中率的算法
- 如果没有特定的输入模式(全值域随机)，那么很难获得较高命中率

# 模式匹配 (Pattern Matching)

从之前 WAL 分析可以看到，对数据写入进行 **预测**，可以更好地布局数据、选择策略，对提升性能有很大的帮助。

缓存算法和读模式是否匹配决定命中率，缓存命中率影响读性能对读取行为进行预测，可以调整读缓存策略，提升性能。

从这些事情我们可以看到，对接口行为进行模式匹配，从而预测未来的接口行为，**调整策略**、**数据布局** 来 **适应** 预测，是重要的优化手段。

# 缓存击穿

在随机写入, 或其它模式匹配失败的情况下,  
读操作在读缓存的命中率很低, 击穿了缓存层, **大部分操作落在低速存储层上。**

如果我们的 Write 操作支持 Update / Delete 的语义,  
那么每次写都附带一次读行为, 同样存在缓存击穿的风险。

## Upsert only 语义:

写入 key-value 时, 不需要考虑是  
否存在 key 的旧版本数据

VS

## Update / Insert 语义:

Update key-value 时, 若 key 的旧版本  
数据不存在则报错  
Insert key-value 时, 若 key 的旧版本  
数据存在则报错

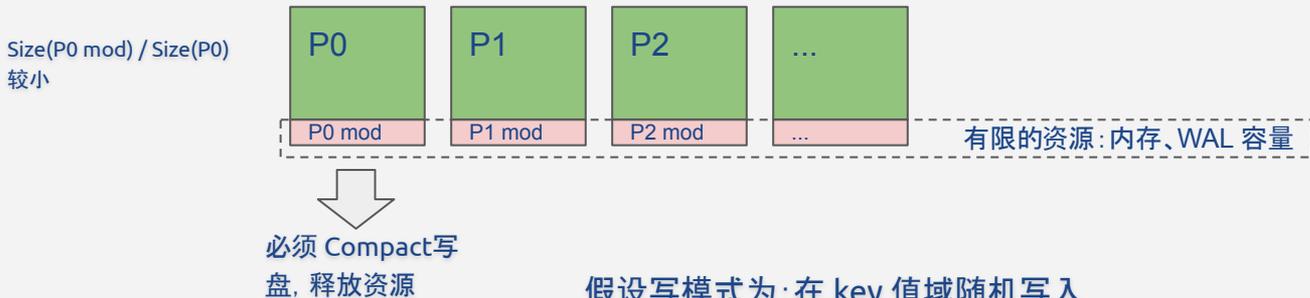


# B+Tree 引擎的 Write 性能

## Write:

- 查找写入叶节点, 内存操作, 性能为  $O(\log_2 n)$ , 较快
- Append 到 WAL, 为顺序 IO, 较快
- 更新到叶节点:
  - 若之前节点不在内存, 需先从磁盘加载
  - InnoDB: double write buffer 解决 partial write, 占用更多 IO 资源
- 需要访问锁, 可能造成性能瓶颈

# B+Tree 引擎的写放大问题



假设写模式为: 在 key 值域随机写入,  
那么:

- 每个叶节点的均摊到的 WriteCache 比较小, 占叶节点的百分比很小
- 所有叶节点的 WriteCache 体积之和, 大小受内存容量限制
- 为了释放资源, 叶节点在修改量占比很小的时候, 就必须 Compact 写盘

这样就导致了巨大的写放大率。

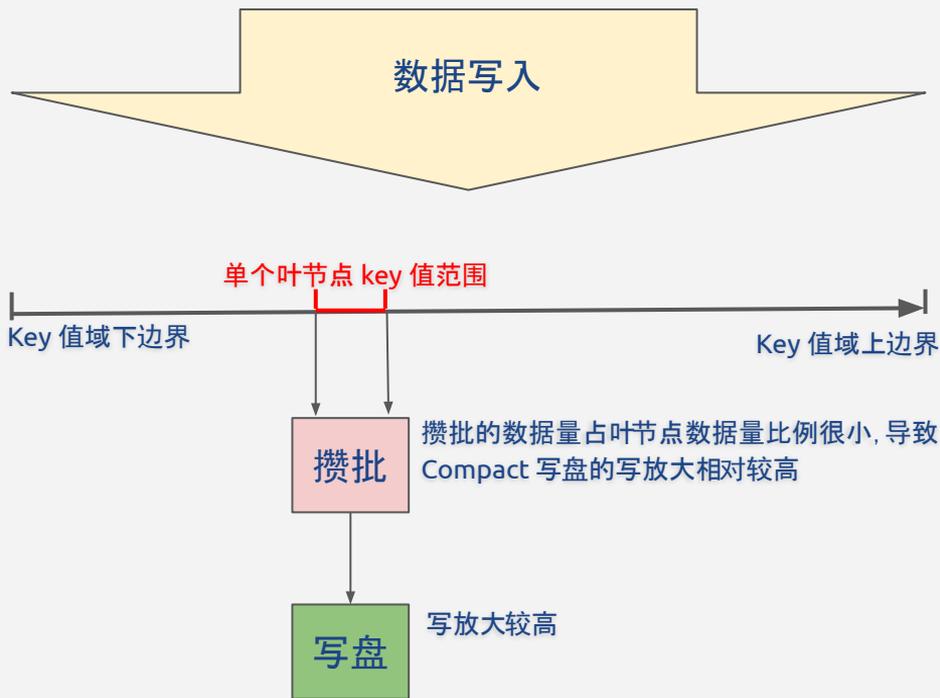
可以看出:

- 写模式越接近随机, 写放大越大
- 原有存量数据越大, 攒批相对大小越小, 写放大越大

# B+Tree 的写放大

TODO: 业界测试报告图

# B+Tree 写放大问题来源



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。



# B+Tree 的写放大问题的量化

对于聚类写入(一段短时间写的值域范围较窄)

, B+Tree 的写放大并不严重。

对于 Append 模式写入能达到很低的写放大。

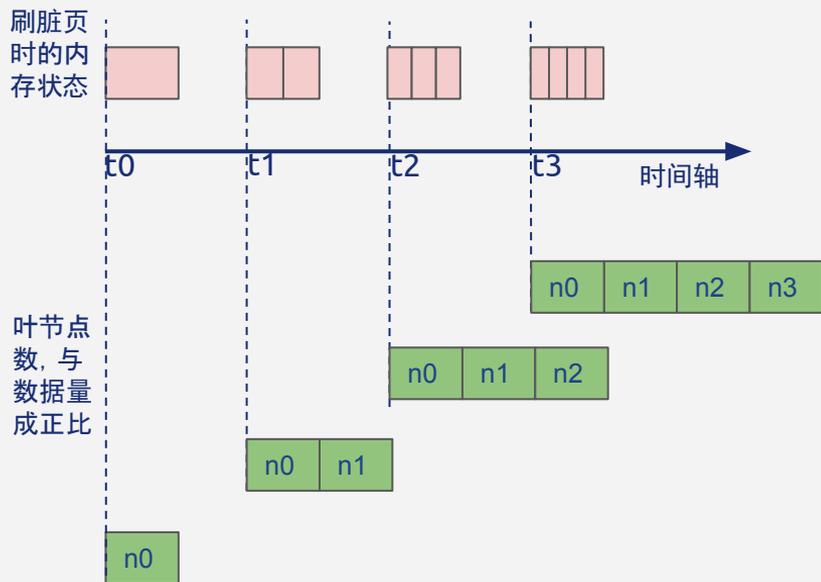
对于离散度很高的写入, 例如 纯随机写入,

随着时间变化, 内存 buff (大小是固定的) 所涉及的节点越多, 导致写放大越大。

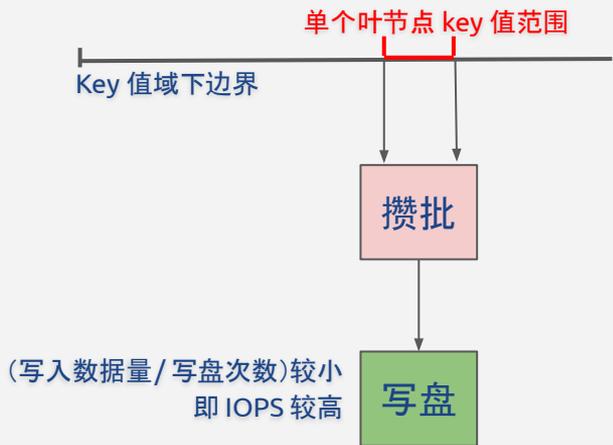
其写放大随着节点数线性变化, 也即是:

**写放大与存量数据成正比, 是 线性关系**

内存总容量是固定的, 随着节点数增多, buff 中涉及的节点数越多, 但 **修改量: 节点数据大小** 越小



# B+Tree 的小 IO、散 IO 问题



与写放大类似, 我们还可以推算出

**写入数据量 / 写盘次数 较小,**

也就是数据局部性差, IOPS 高, 性能较低,

这是 B+Tree 的小 IO 的问题。

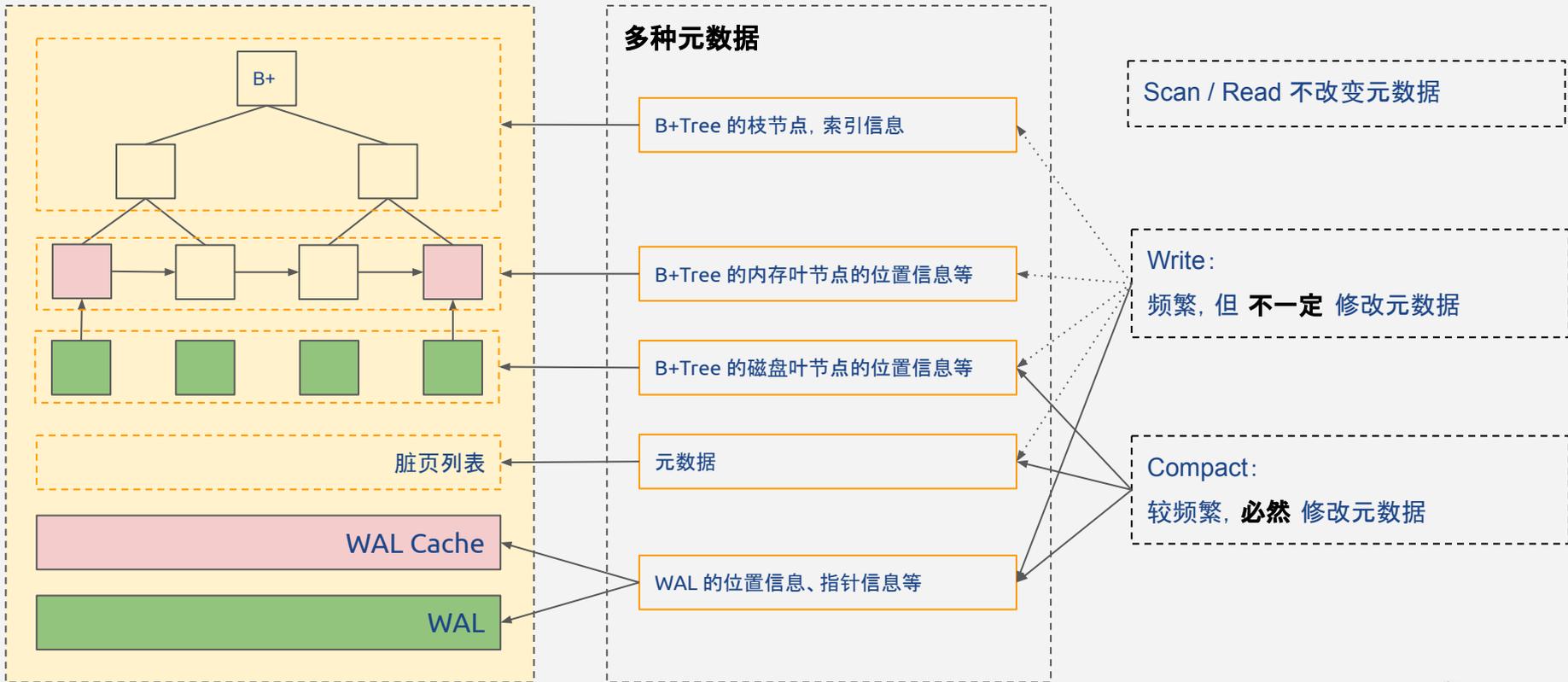
另外, 每次 IO 涉及单个叶节点, 叶节点在磁盘是离散存放的,

这样导致多次 IO 之间不存在连续性,

这是散 IO 的问题。

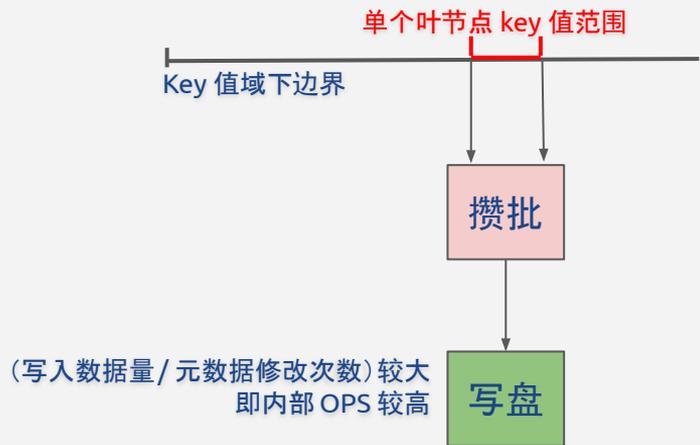


# B+Tree 中的元数据





# B+Tree 的元数据 TPS 问题



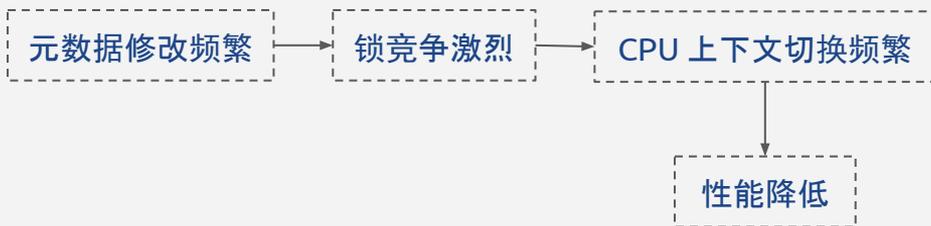
与小 IO 问题类似,

**写入数据量 / 元数据修改次数** 为较大值。

也就是元数据的 OPS 较高。

影响最大的是锁操作,

我们可以看到业界很多在 B+Tree 上的锁优化, 但其仍旧是瓶颈之一。





# B+Tree 引擎问题小结

我们看到了 B+Tree 的几个问题：

- 写放大高
- IO 行为小而散
- 元数据修改频繁

这些问题很大程度是因为 **攒批能力** 不足引起的。

现在我们看看一种新的存储结构 LSM Tree

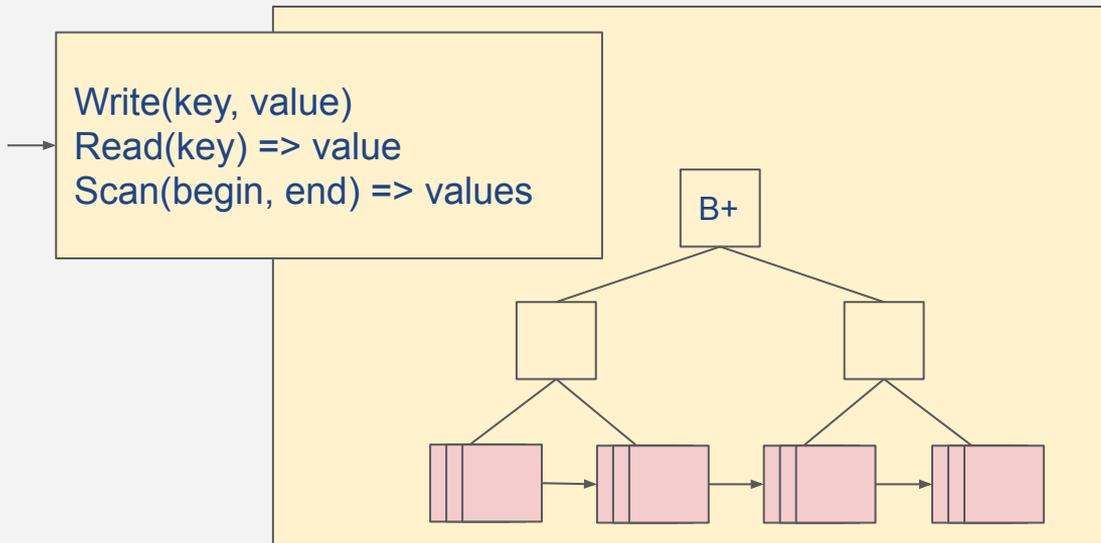
看它怎么解决 B+Tree 的问题

# 回顾一下

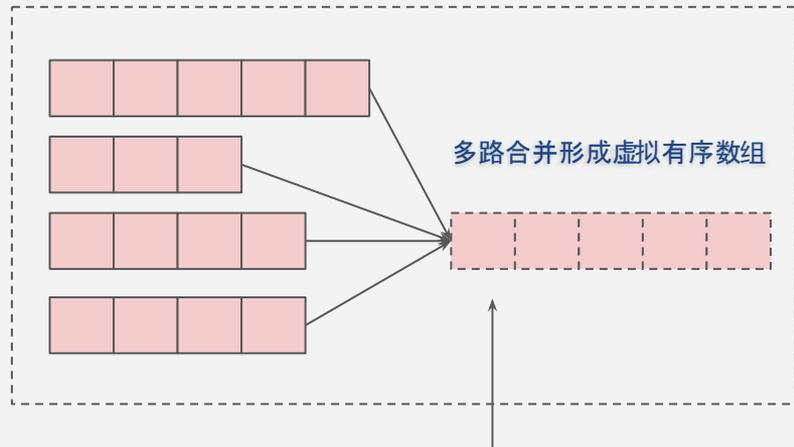
存储引擎是:能低成本动态保持局部性、有序性的结构

B+Tree 引擎把有序数组切成了小段存放, 特征符合

局部性良好  
的有序结构



# 另一种局部性良好的有序结构: LSM Tree



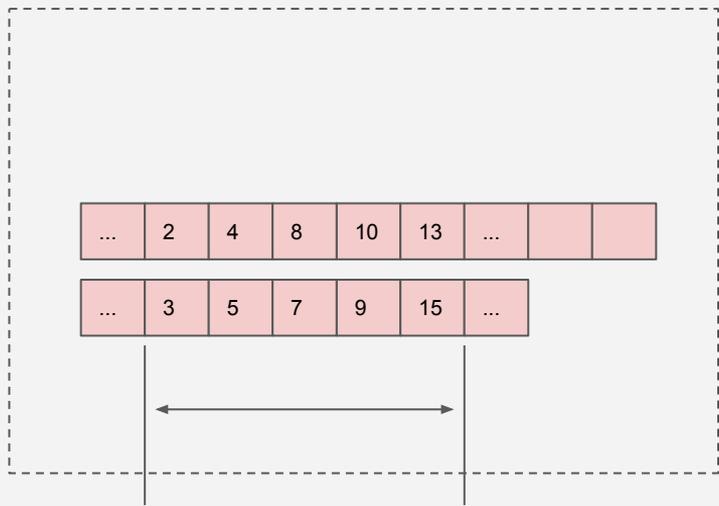
虚拟数组, 局部性较好的有序结构

多个范围重叠的有序数组, 经多路合并后(延迟动作, 需要时才发生),

对外表现为 **一个虚拟有序数组**,

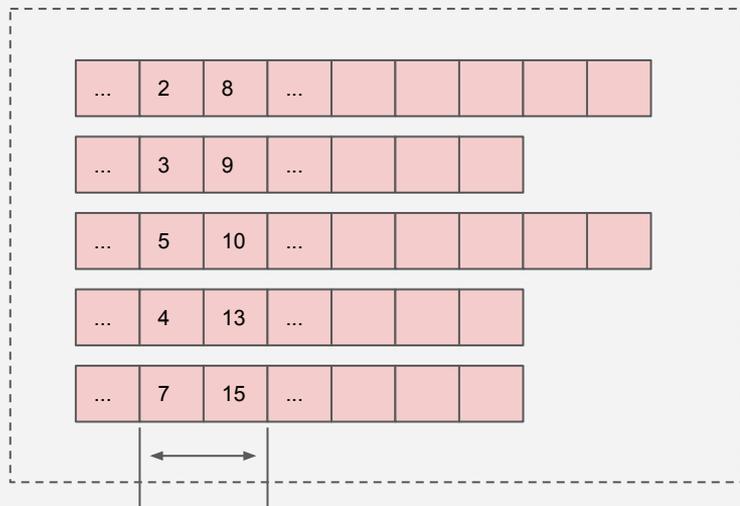
从静态结构观察, 这种结构符合有序性和一定的局部性。

# LSM Tree 的局部性高低, 由数组个数决定



Key 范围为 [2, 15] 的数据, 存放在 2 个物理位置

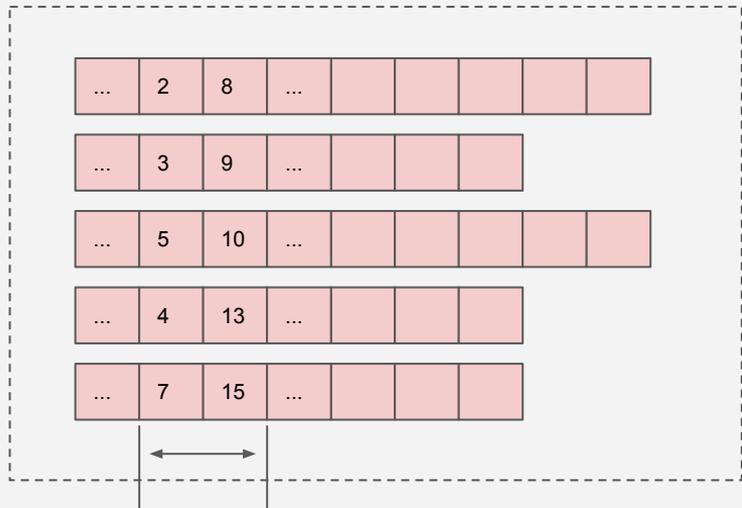
**数组个数少, 局部性高**



Key 范围为 [2, 15] 的数据, 存放在 5 个物理位置

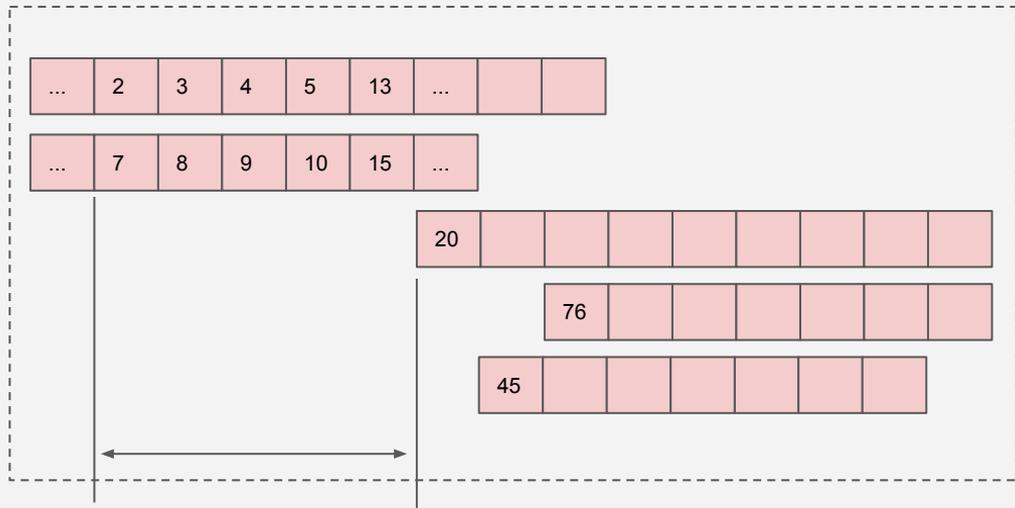
**数组个数多, 局部性低**

# 各个数组内聚性高低，也影响局部性



Key 范围为 [2, 15] 的数据, 存放在 5 个物理位置

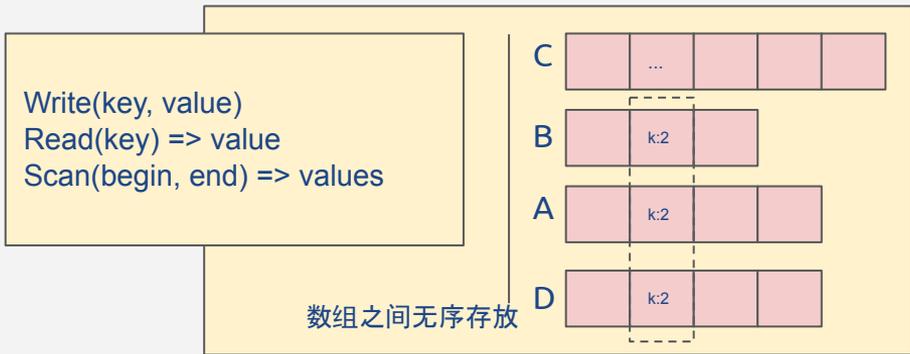
每个数组范围广, 数据间重叠多, 局部性低



Key 范围为 [2, 15] 的数据, 存放在 2 个物理位置

每个数组范围窄, 数组间重叠少, 局部性高

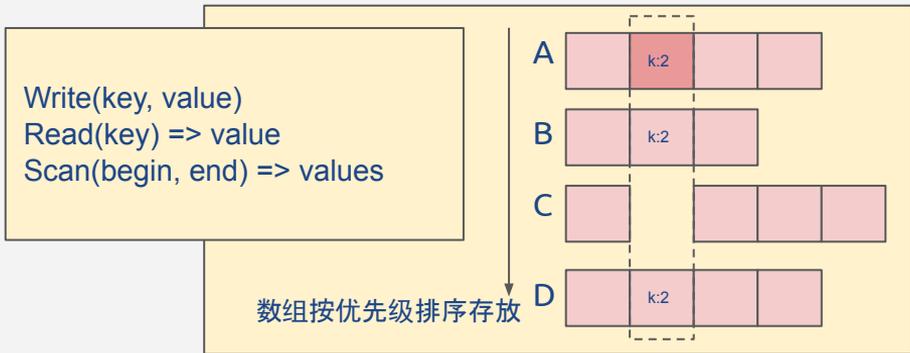
# 多路合并过程中的 key 排重问题



B+Tree 的叶节点的 key 值范围相互不重叠, 不存在 key 重复的问题。

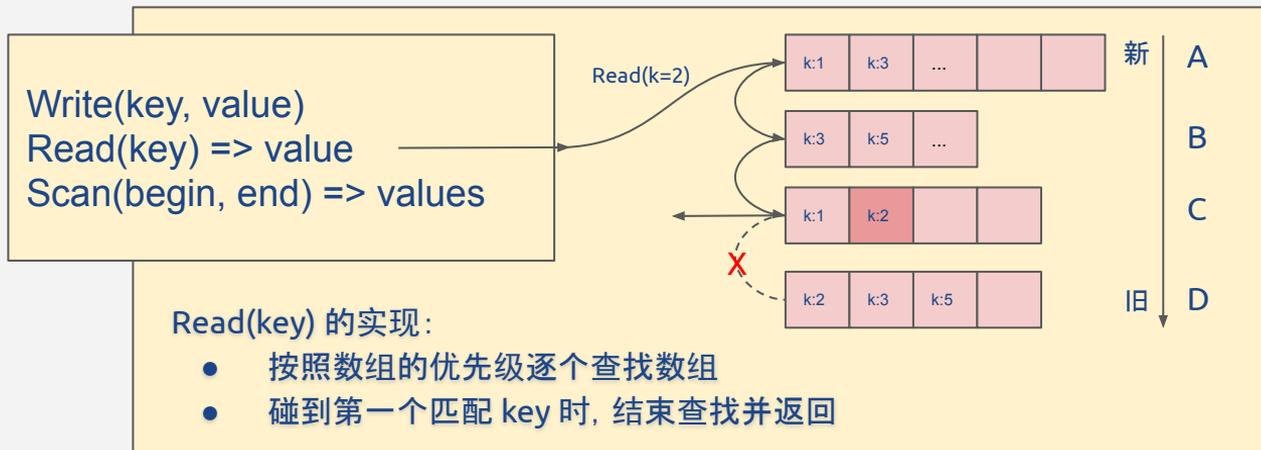
LSM Tree 的数组之间 key 值范围重叠, key 可能存在重复。在多路合并过程中需要选择多个相同 key 的其一。

对多个数组排序



LSM Tree 将数组按优先级排序, 选取优先级最高的 key, 在接下来的 Write 实现分析中, 我们可以看到这个优先级就是写入顺序。

# LSM Tree 的 Read 实现



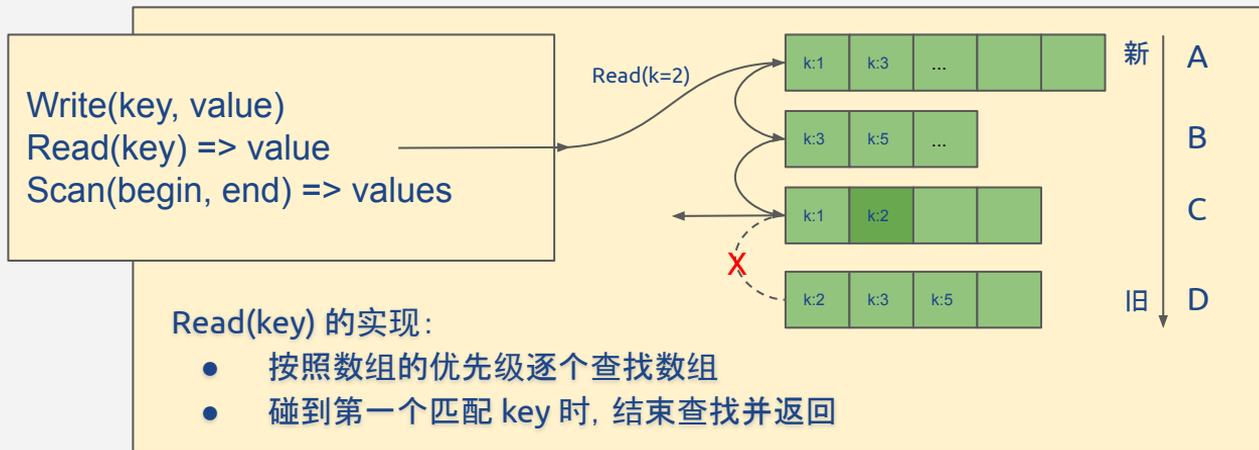
假设总数据量  $N$ , 数组个数  $K$ , 数组大小较为均匀, 则性能:

- 最差为每次都查找到最后一个数组:  $O(K * \log_2(N / K))$
- 最佳为每次都在第一个数组命中:  $O(\log_2(N / K))$

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。

可见性能与  $K$  强相关, 降低数组个数总能提升性能。

# LSM Tree 的 Read 的读放大问题

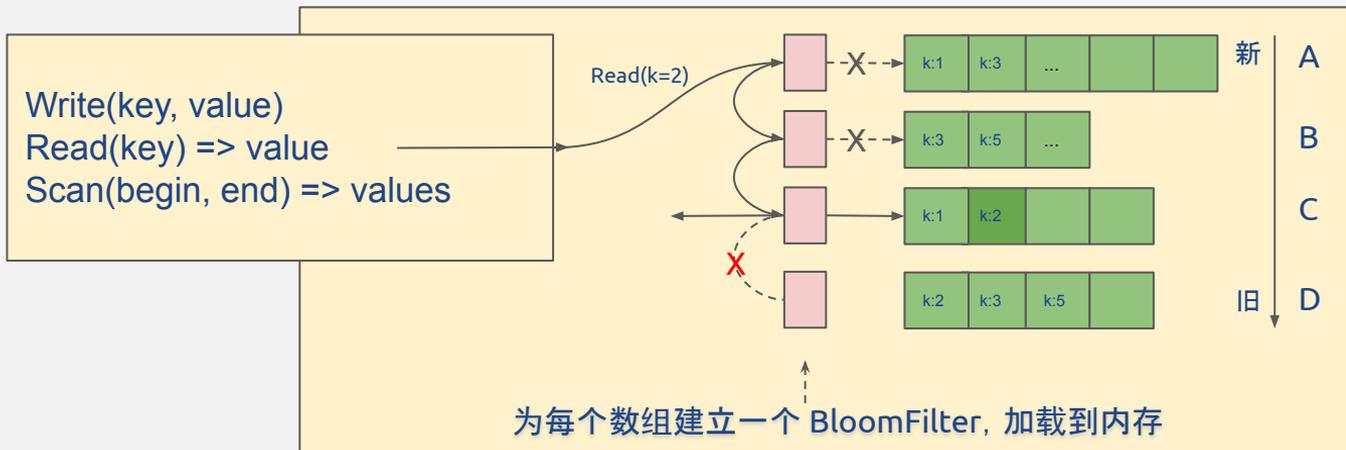


从刚才我们知道, Read 平均需要访问多于一个数组, 才能获得结果。也就是说, 存在读放大问题。

当数组不在内存, 而是磁盘数组时, 读放大问题被大大放大:

- 上述的访问多于一个数组的读放大, 使得缓存命中的几率下降
- 当需要从磁盘读取数据时, 单次点读产生系统层、硬件层的读放大
- 两者结合, Read 的读放大问题就比较明显

# 使用 Bloom Filter 减轻 Read 的读放大问题

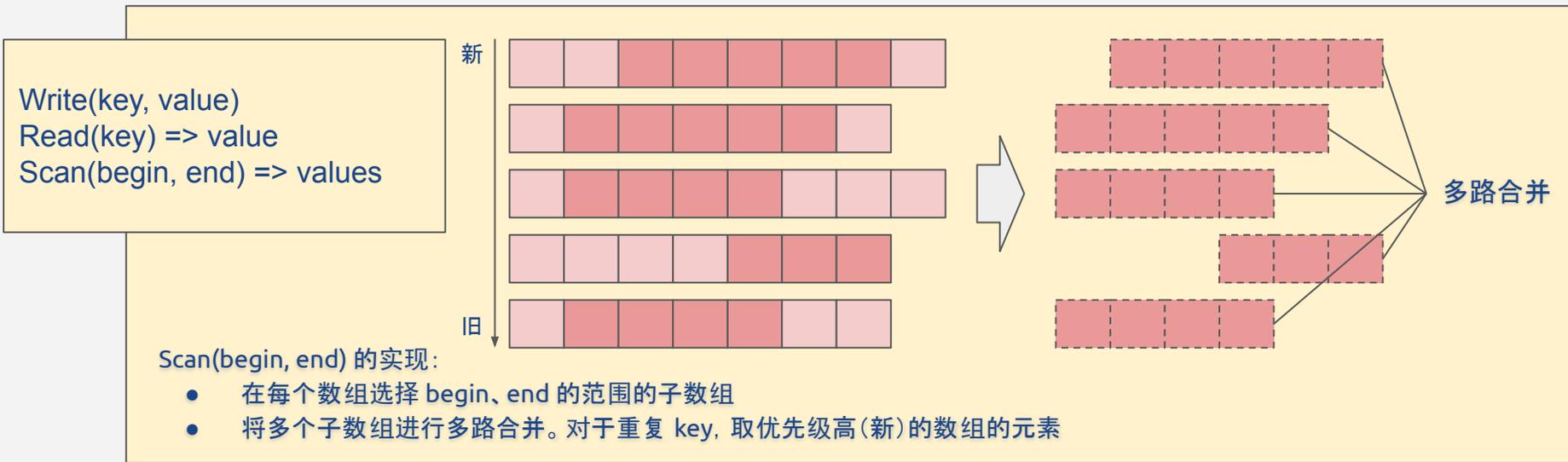


数组文件是一次性写成、之后不再改动，  
因此在数组文件创建的时候，顺带生成对应的 Bloom Filter 是很容易的。

在 LSM Tree 是磁盘数据的情况下，数据由于体积较大，大部分在磁盘，小部分在缓存。  
而 Bloom Filter 的体积较小、可控，可以预先加载到内存。

在 Read 时，先查询数组的 Bloom Filter，为真时再真正查询数组数据，大大减少了读盘次数，减轻读放大。

# LSM Tree 的 Scan

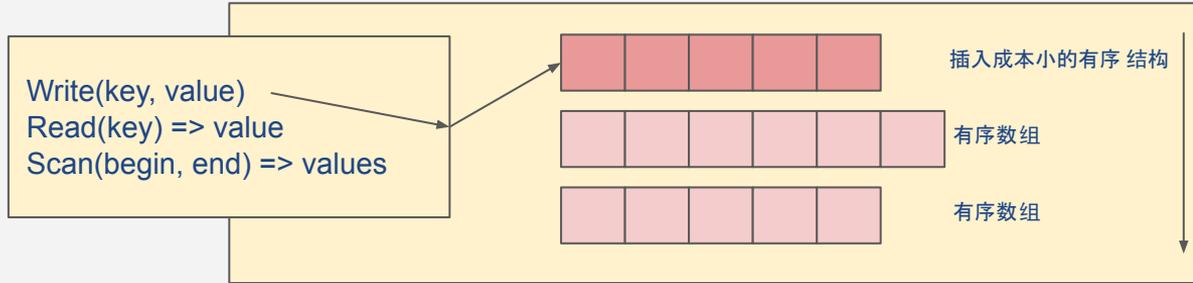


假设总数据量  $N$ , 数组个数  $K$ , 数组大小较为均匀, 则性能为:

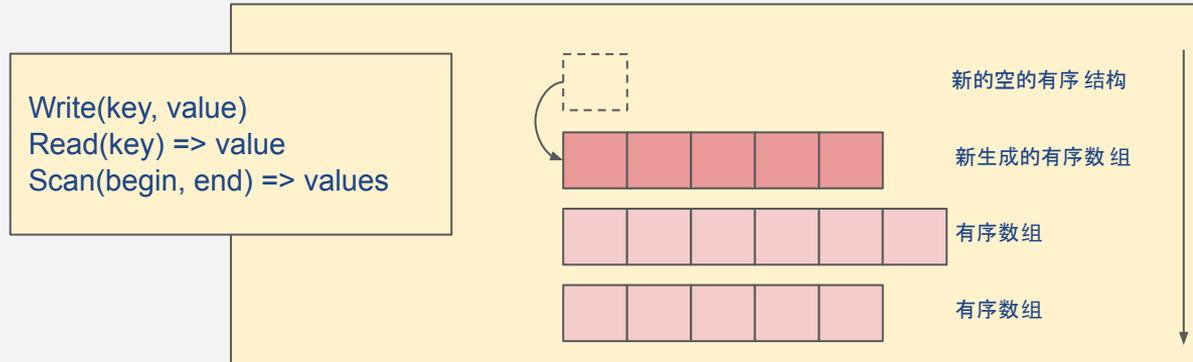
- 定位:  $O(K * \log_2(N / K))$
- 遍历及多路合并, 与局部性相关。之前我们分析过, 局部性与数组个数  $K$  相关

可见, Scan 和 Read 的性能都与  $K$  强相关。

# 纯内存的 LSM Tree 的 Write 实现



Write 只需要插入到最上层数组  
显然插入到数组会造成元素移动,  
因此最上层使用其他有序结构代替。

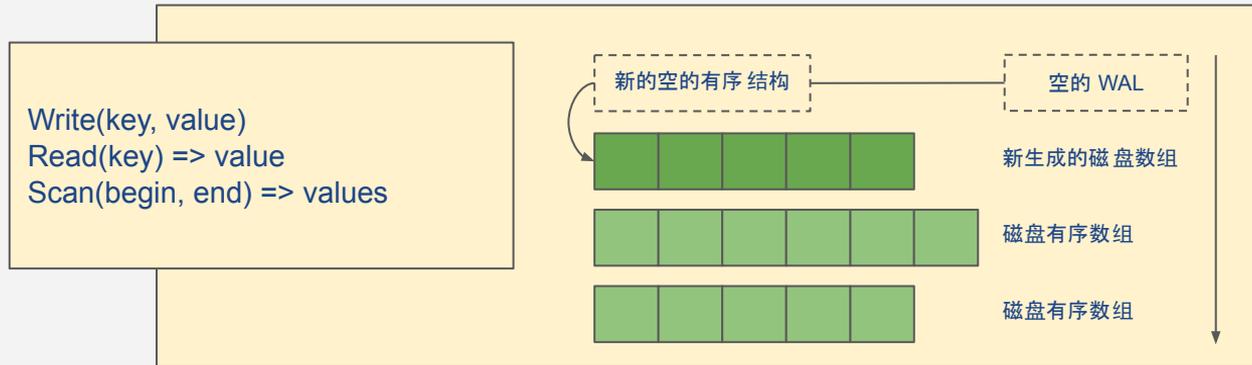
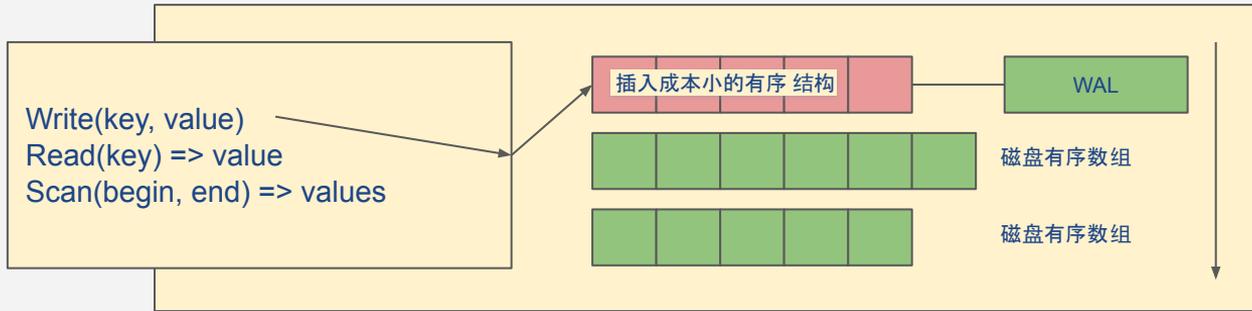


插入时定位性能为  $O(\log_2 n)$   
其中  $n$  是最上层有序结构的大小。

为了减少  $n$ , 并提高冷却数据的局部性,  
可以在  $n$  达到阈值时,  
将最上层下移, 存为有序数组以提高局部性,  
并生成新的空的最上层有序结构。



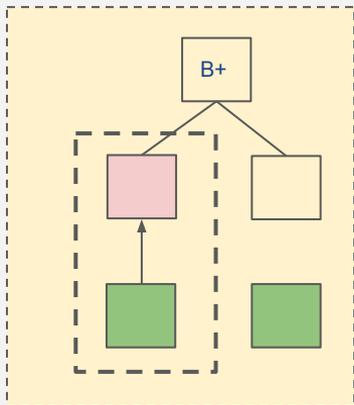
# 磁盘的 LSM Tree 的 Write



与纯内存的 LSM Tree 基本一致，  
增加了 WAL 防止最上层数据的丢失

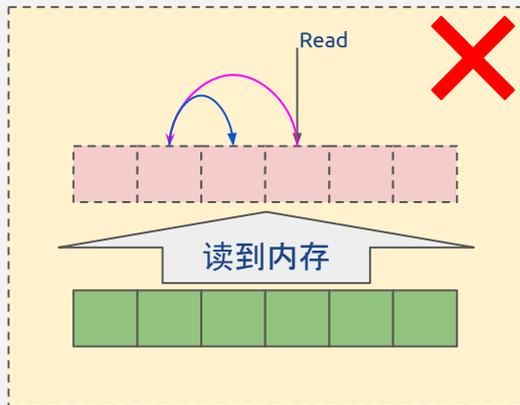
# 磁盘有序数组的读取问题

B+Tree

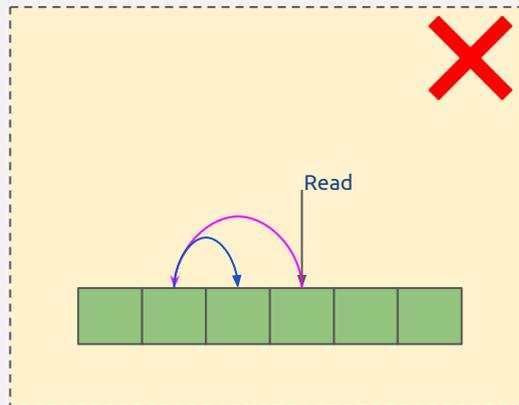


B+Tree 叶节点可以整页读写, 满足磁盘 IO 的局部性要求。再在内存中进行二分查找。

LSM Tree

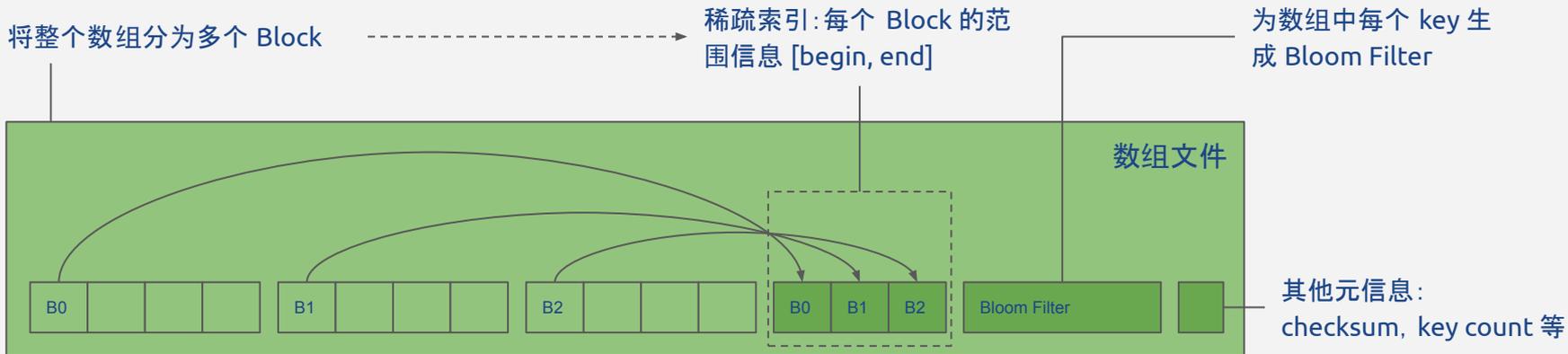


LSM Tree 的磁盘数组不能一次性全加载到内存再进行二分查找, 占用内存会过多, 读放大也过大



不能直接在 LSM Tree 的磁盘数组上二分查找, 会造成大量磁盘点读, 造成不能接受的磁盘 IOPS

# 磁盘有序数组的微观结构

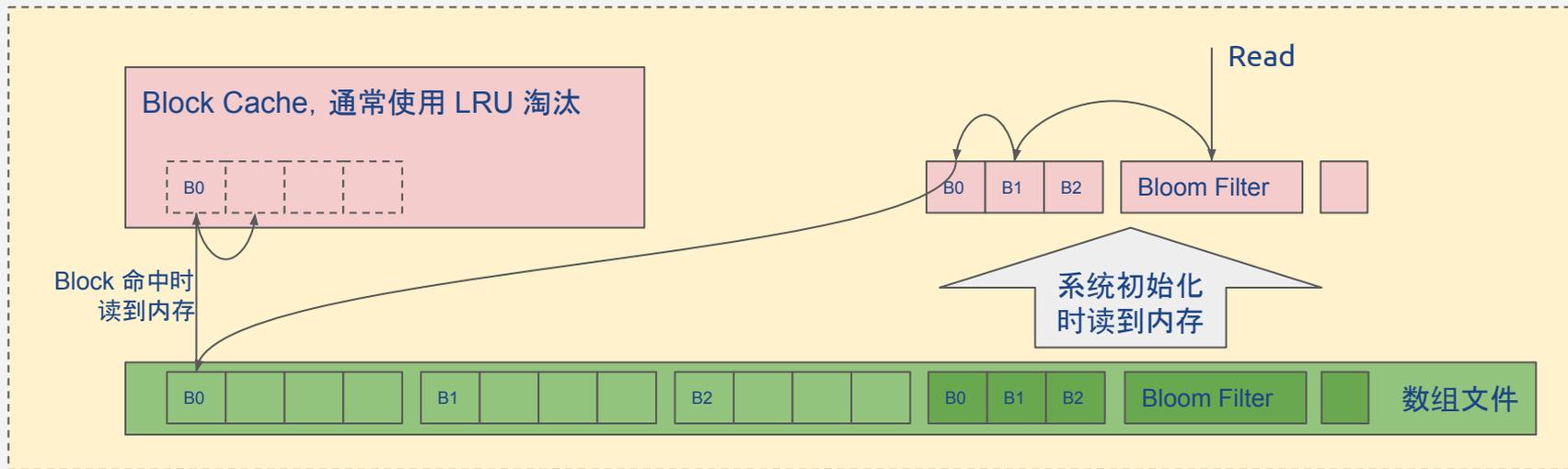


将数组划分为多个 Block, 以 Block 作为磁盘读取单位, 既满足了局部性要求, 又控制了读放大。需要考虑两者平衡。

当需要压缩功能时, 可以将 Block 同时作为 IO 单元和压缩单元。或者增加新的结构层次: 以多个 Block 作为一个压缩组。

压缩算法的局部性要求可能与磁盘的局部性要求有差异, 需要谨慎平衡 Block 的体积,

# 在微观结构帮助下，磁盘有序数组的读取



通过划分 Block、为 Block 建立稀疏索引，实现了二分查找需求。

既消除了高 IOPS 问题，也大大减轻了读放大，也没有大量占用内存。

# LSM Tree 的 Compaction

从前面分析, 我们知道:

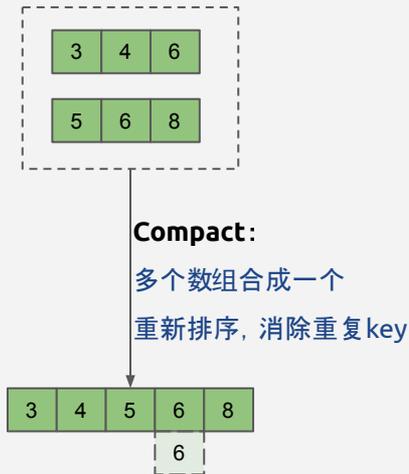
- 数组个数极大影响 Scan / Read 性能
- Write 会不停产生新的数组

因此我们需要一些方法来减少数组数量。

把若干个数组合并成一个新的有序数组是有效的方法, 也就是 **Compact / Compaction**。

Compaction 的目标:

维持数组的个数不超过一定数量, 从而保持读取性能。





# 从小向大进行 Compact

每个新写下来、未经 Compact 的数组文件的大小是固定的, 取决于 WriteBuffer 的大小, 我们把它叫做 L0 文件。

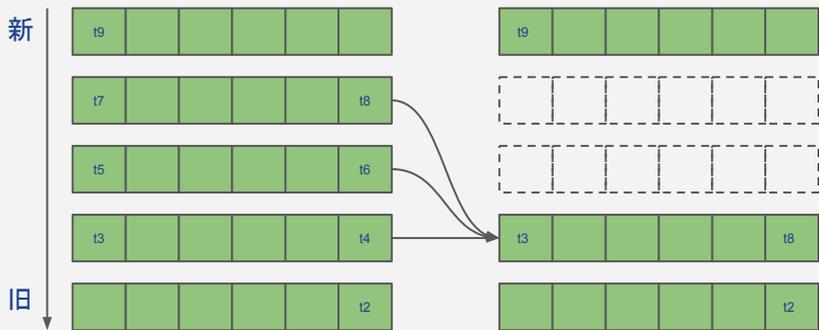
Compact 后的结果是 L0 的倍数。

因此可以指定 L0 文件 Size = 1, 为 L0 的 x 倍大小的文件则 Size = x

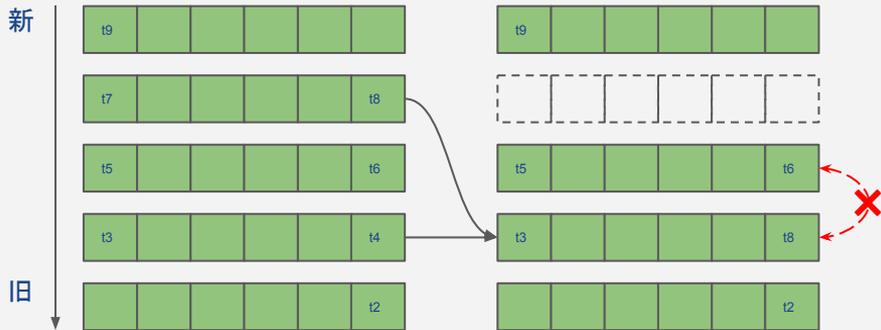
很容易推算出: 优先 Compact 最小体积的数组可以最低成本地减少 总数组个数。

因此 Compation 总是从 L0 层开始, 按文件体积从小到大地进行。

# Compaction 的选取相邻要求

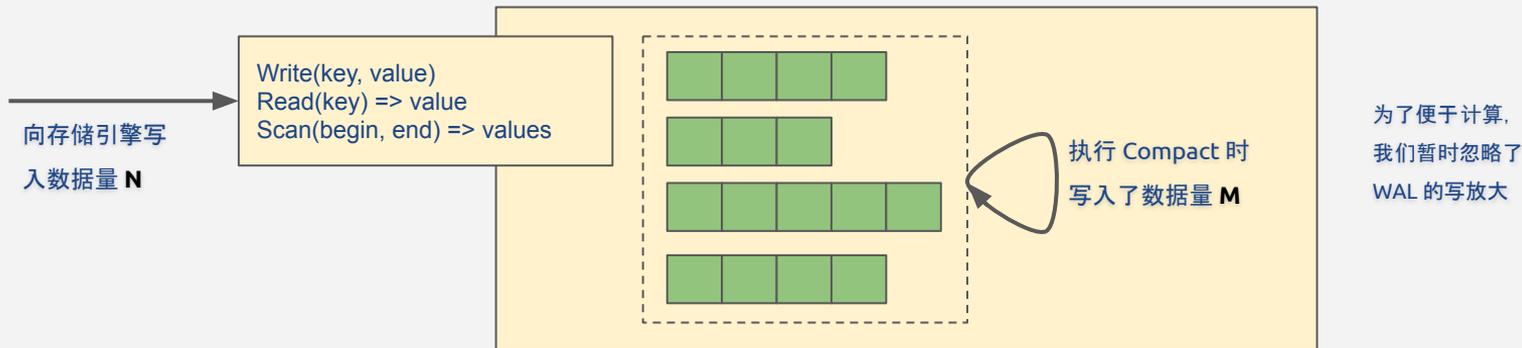


为了保证 Compact 后仍旧有明确的新旧排序，  
要求参与 Compaction 的数组是连续相邻的。



否则数组之间的写入时间会产生重叠：  
导致读取时，无法以优先级进行 key 排重

# Compaction 产生的写放大



写放大的含义很好理解: 磁盘写入的数据量 / 实际的数据量。

如图, 在某时刻:

我们向存储引擎一共写入了数据量  $N$ , 引擎内部执行

Compaction 时又向磁盘写入了数据量  $M$

那么, 该时刻的写放大为:  $(N + M) / N$

当写放大为 3 的时候, 通常意味着数据平均被重复写了三遍。

留意: 写放大为 3 的 **IO 消耗**, 并不是写放大为 1 的三倍, 而是 (1 写 + 2 读 + 2 写):  $1 = 5$  倍。

因为参与 Compaction 的输入数据通常不在内存, 需要从磁盘上读起来。

相比之下 B+Tree 的 Compaction 基本没有读取消耗。此外, LSM Tree 的 Compaction 除了 IO 还伴随着 CPU 消耗。



# 多种多样的 Compaction 策略



有多种 Compaction 策略: 不同的选取方式, 还可对合并后的数组重新切割、组织

使用不同的 Compaction 策略, 会导致不同的数据分布, 从而产生性能的巨大差异。

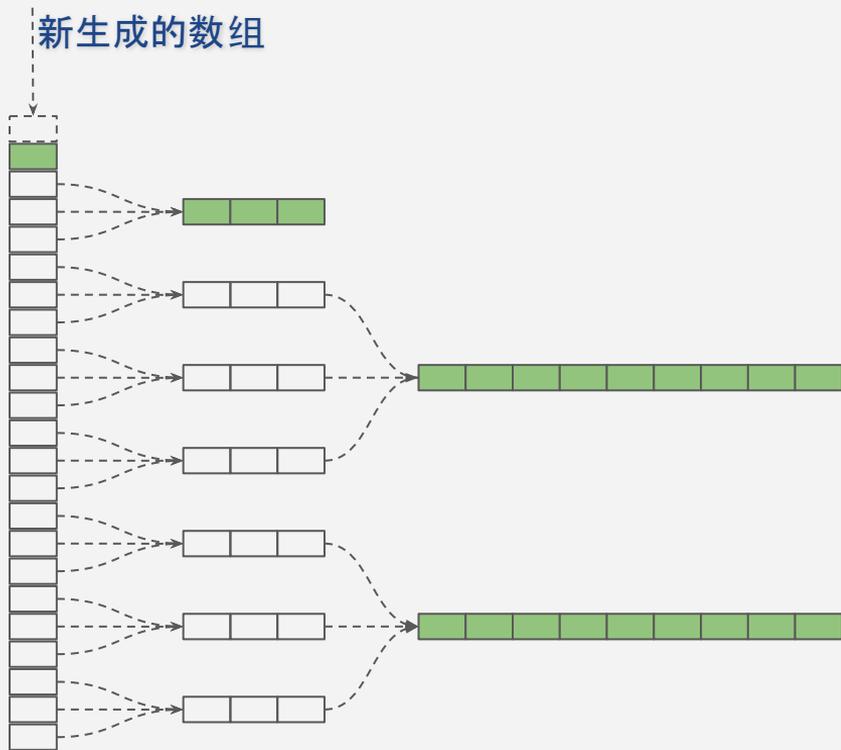
业界现有的策略, 有不同的性能侧重方向, 互有优劣。

# 举例：朴素 Compaction 策略

我们用一个非常简单的 Compaction 策略，来进行讨论分析。

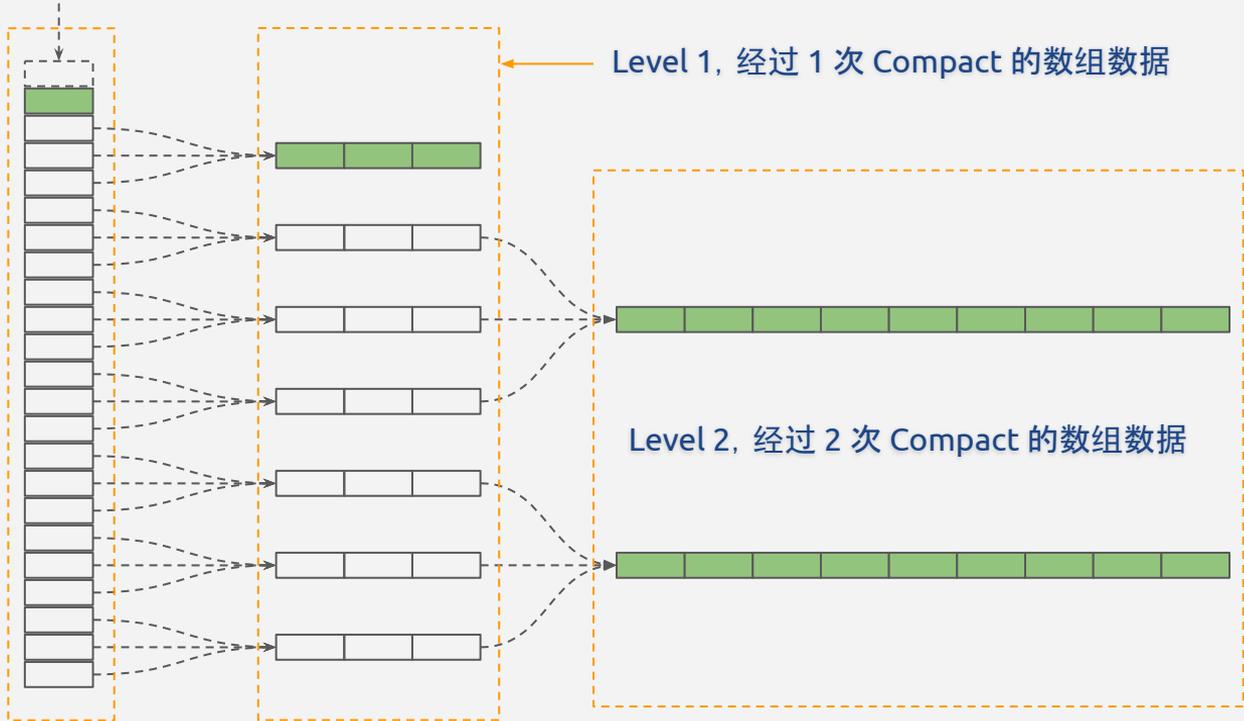
**算法：等大逢 T 进一**

当出现 T 个同等大小的数组时，触发 Compaction，将 T 个有序数组文件，排序后合成一个新的有序数组。



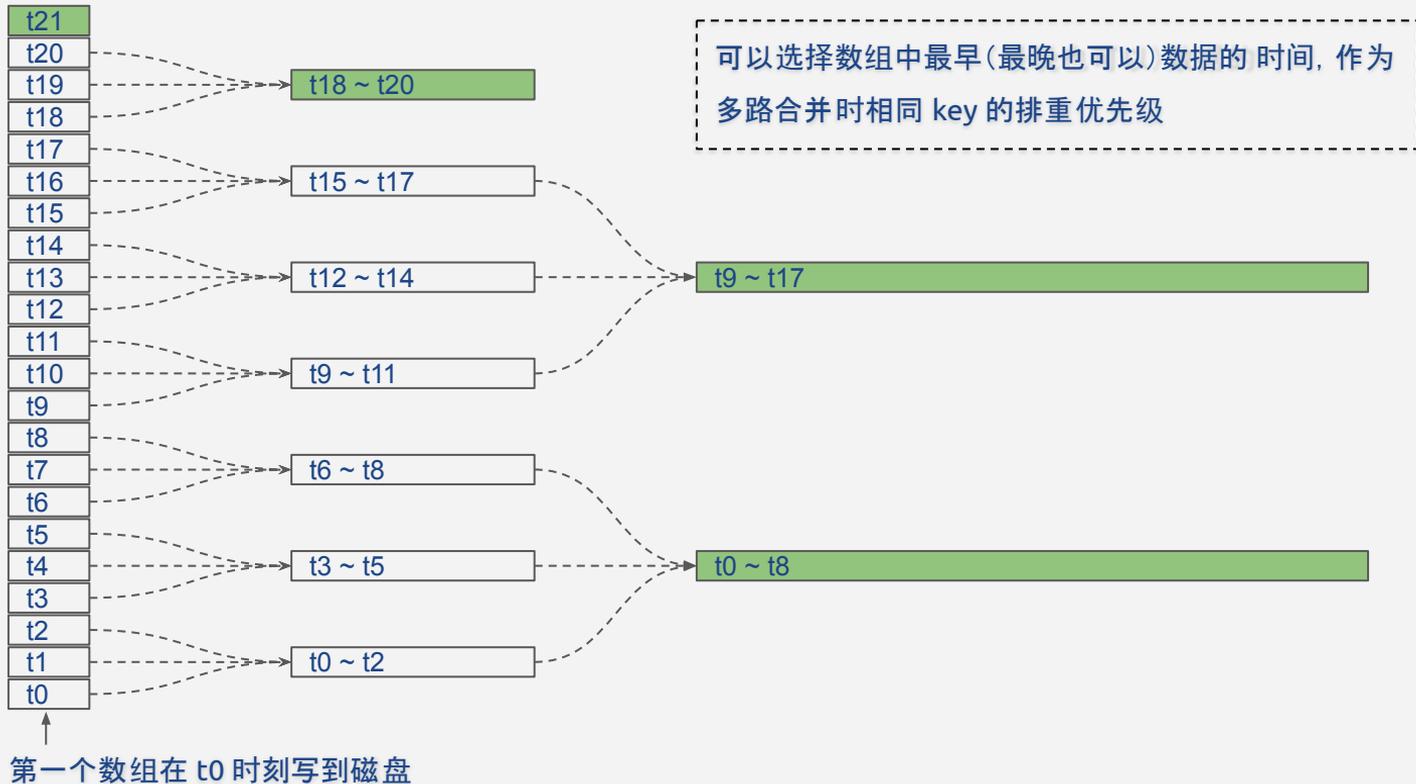
T = 3, 每出现 3 个接近大小的数组，Compact 成更大的

# 由于 Compact 次数不同，自然形成的层 (Level)



在朴素 Compaction 中，  
根据所在层可以直接推算出  
该层的数据写放大

# 遵循了相邻要求，其结果有清晰的优先级





# 朴素 Compaction 策略的写放大

朴素 Compaction 策略的写放大很好量化, 由三个变量决定:

- L0 数组文件的数据量大小  $M$
- 总数据量  $N$  (写入存储引擎的总数据量)
- $T$  (逢  $T$  进一)

当存储的数据足够多、L0 的文件不占主要体积时,

写放大计算:  $\text{Log}_T(N / M) + 1$

当  $T$  是一个适当大的值的时候, 可以有一个相对稳定的写放大。

普遍使用的是  $T = 10$



# 朴素 Compaction 的代价估算

第一个因素是 **数组个数的上限 K**：

- 极端情况一：数组的上限是无限大，那么 Compaction 可以永远不执行，代价为 0，但 Read / Scan 将非常慢
- 极端情况二：数组的上限是 1，那么每次新生成一个数组，我们都应该执行 Compaction，写放大非常大

这是一个 **取舍** 点，K 越大，读操作越慢，但 Compaction 代价越小，写放大越小。

设定 K 值，可以获得符合预期的、较稳定的读性能，  
但写放大不固定，会随着总数据量增大而增大。

与之相对的是设定数组大小上限，可以获得符合预期的、固定的写放大，  
但读性能会随着总数据量增大、最大数组增多而下降。

我们更多使用的是设定 K 值。

# 朴素 Compaction 的优化

在执行 Compact 时, 把更小的也捎带一起执行, 减少小数组文件的数量:

- 捎带所有的执行层之下所有的文件, 它们的大小总和不会超过执行层单个文件大小
- 候选数组个数 T 这个参数就不合适了, 以邻层数组大小比来代替



ClickHouse 的存储引擎 MergeTree 使用了该方案,  
使用  $T = 10$

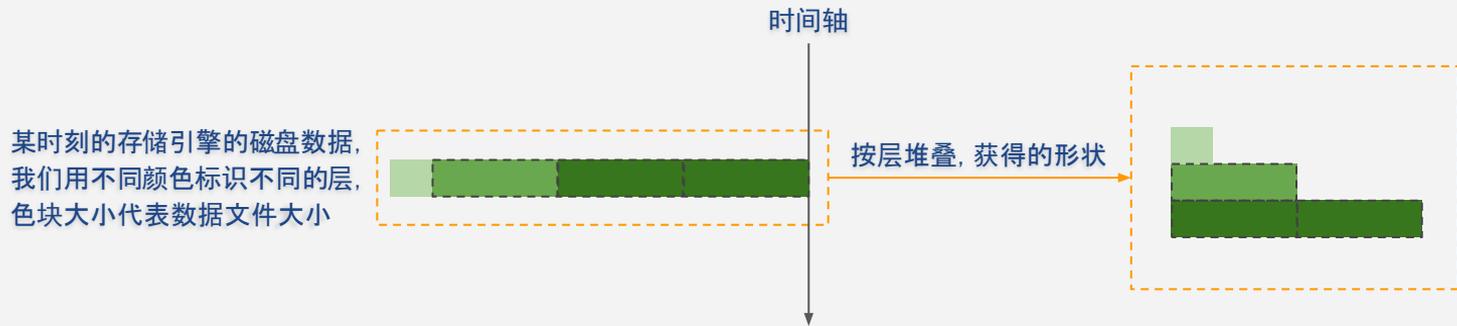


我们分析了 Compaction 的一些基本情况  
再做了一个简单的策略, 运行结果还不错  
现在看能不能找出不同 Compaction 策略之间的共性

# 数据的形状

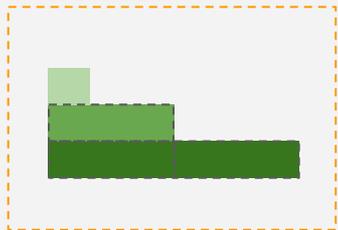
LSM Tree 所存储的数据, 由多个数组组成,  
多个 **数组的大小、新旧关系、值域分布**, 形成特定的 **形状**

由前面分析我们知道, **形状决定了数据集的性能**。  
新数组的加入(数据写入)、Compaction 都会改变数据的形状。

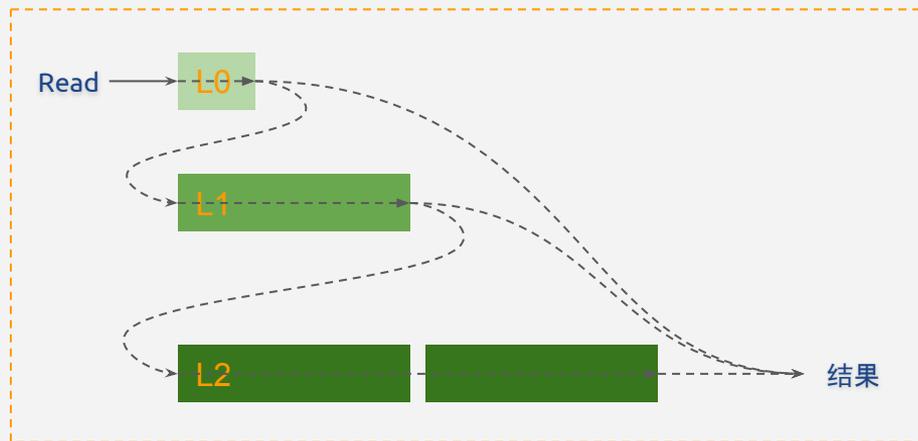


# 数据的形状决定了读取性能

某时刻的数据



收到 Read 请求



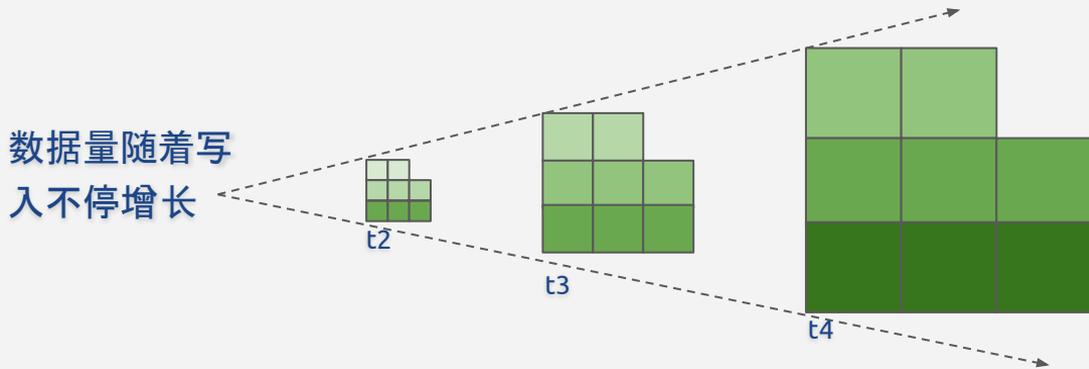
数据的形状决定了Read过程中:操作访问数据的路径,(平均)需要进行多少次操作,每个操作的成本。

因此,形状决定了Read性能。同理可推Scan性能。

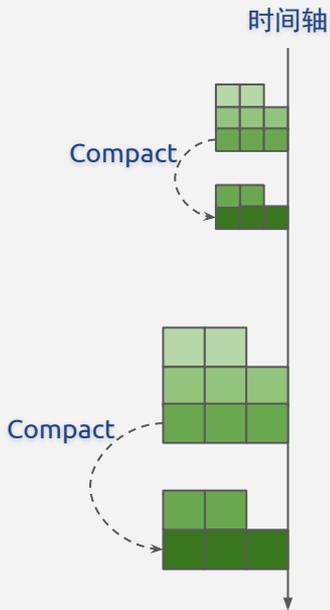
# 形状的稳定性、相似性

我们期望存储引擎在运行过程中能有稳定、可预期的性能。

这意味着, 我们 **期望** 数据在增长的过程中, 形状是 **稳定** 的



# 期望在不同的数据量下 Compaction 的结果相似



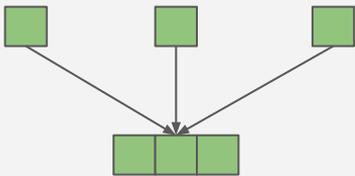
我们希望引擎的性能稳定, 希望数据集的 **整体形状** 是稳定的。

也就是说, 期望在不同的数据量下,

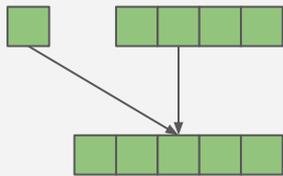
每次执行完 Compaction 之后, 引擎的整体数据的形状与原来相似。

# 合并函数:能循环执行的 Compact 算法

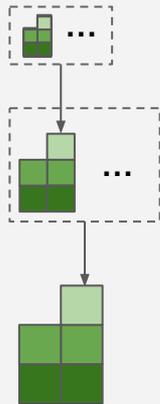
如果某种 Compact 算法符合:  
它的 **结果的形状**,  
(可选:加上 L0 文件一起)可以 **组成它的输入** 形状,  
那么我们称它为合并函数。



一种合并函数:  
逢 3 合一, 3 个 结果可以再组成输入

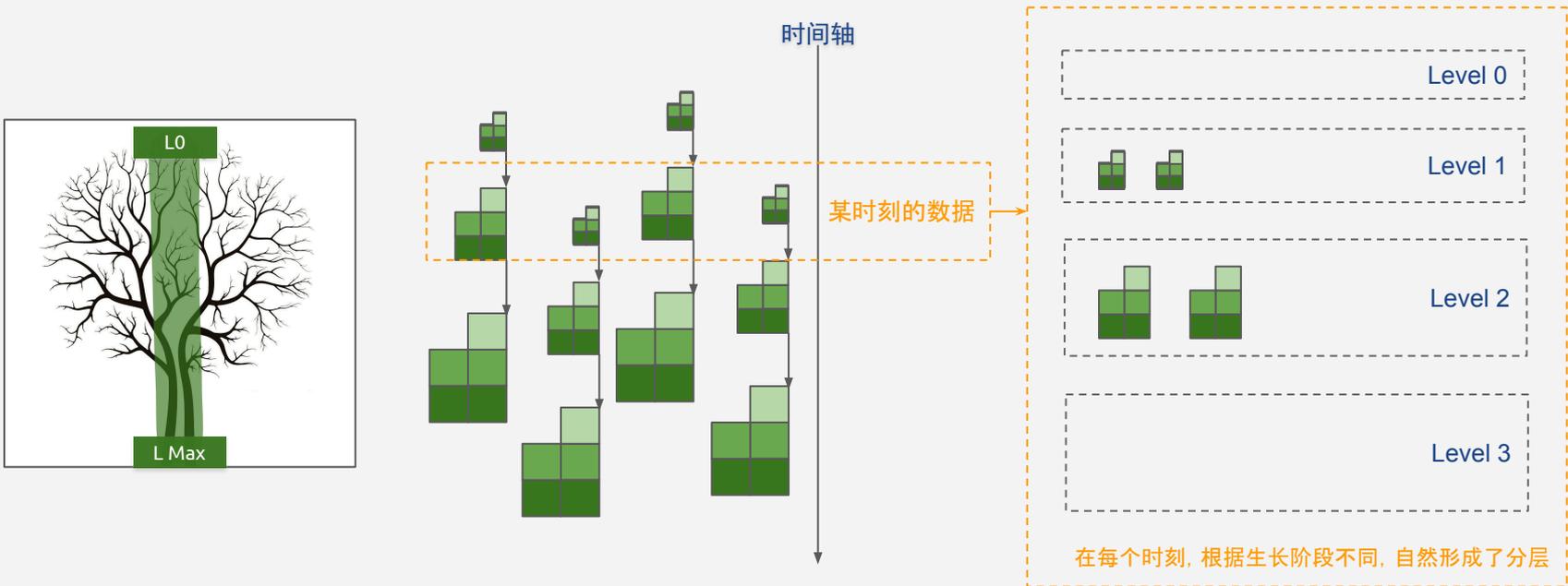


一种合并函数:  
1 加“任意”进 1, 结果可以再作为“任意”



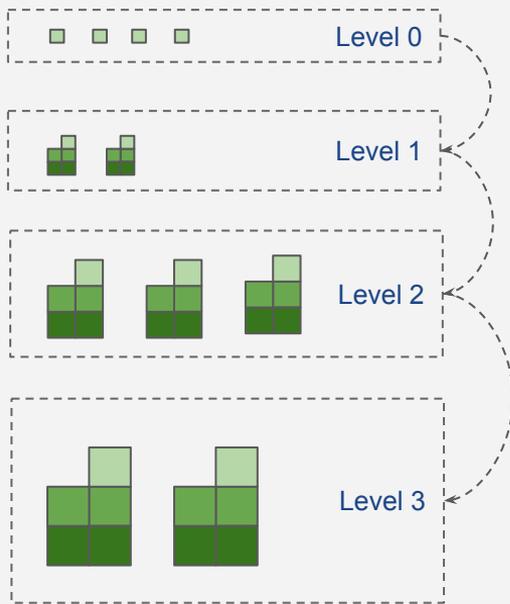
符合这个条件的 Compact 算法,  
就可以循环执行, 让一小簇数据 **生长**, 生长的意思是 Compact 前后 **形状接近**, **体积增大**

# 使用合并函数 Compact, 形成生长图与分层



使用合并函数进行 Compact, 每次执行完的整体数据是一个分形图 / 生长图, 使得整体数据在 Compact 前后的形状相似。

# 使用生长策略后的一些特征

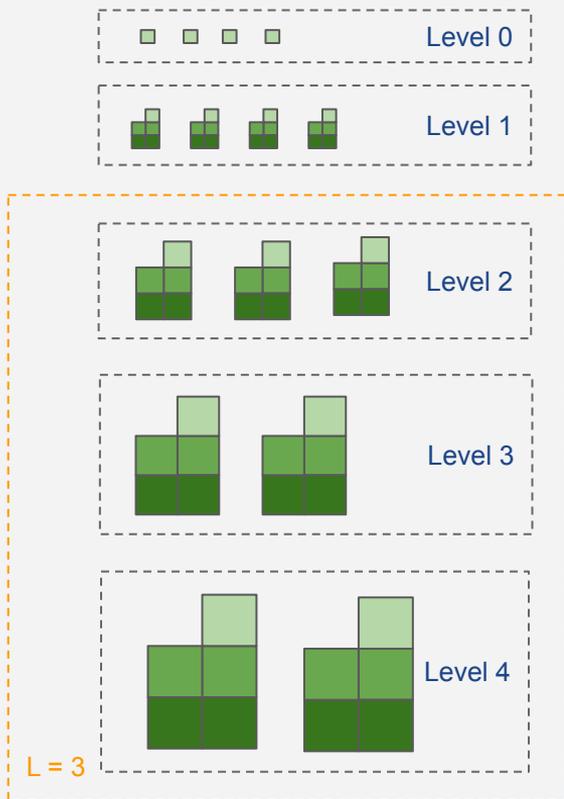


每层之间的数据总量形成等比数列，我们将层间比值称为 size 比，

如果 Compact 算法是仅合不拆的话，那么各层间的数组文件个数接近，每层的层内数组文件的大小也是等比数列。

易推得写放大为  $O(\log_2 n)$ ，2 为 size 比，n 为总数据量

# 动态截取尾部最大的若干层数据



幼体的体积较小，但是数组文件的数量不少。

随着总数据量增加，数组文件的数量也在逐渐增加。

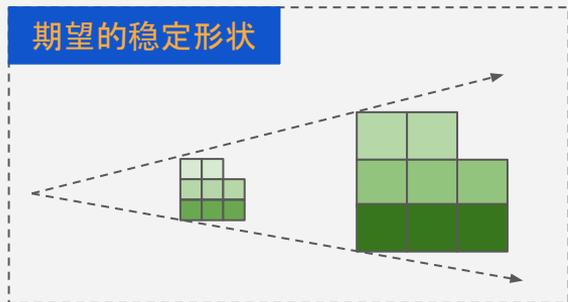
我们的性能与数量相关，因此应该抛弃幼体，只保留最成熟的 L 层，这样就可以保持稳定的数组个数。

在固定了 L 之后，当数据总量为 N 时，

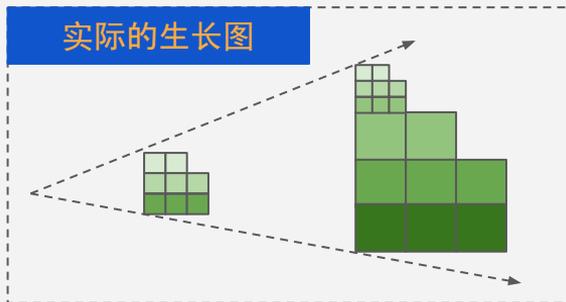
我们可以推算出最大层的期望大小，再反推每层的期望大小。

参见 RocksDB: `level_compaction_dynamic_level_bytes`

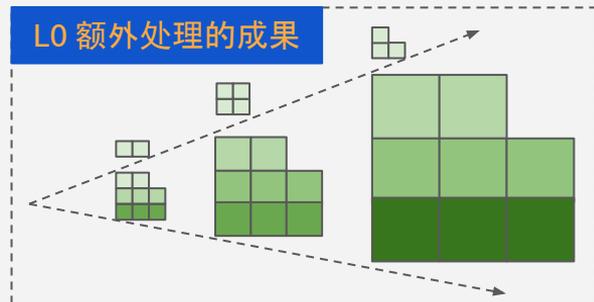
# 对 L0 使用不同的 Compaction 策略



我们**期望**数据在增长的过程中，形状是稳定的。

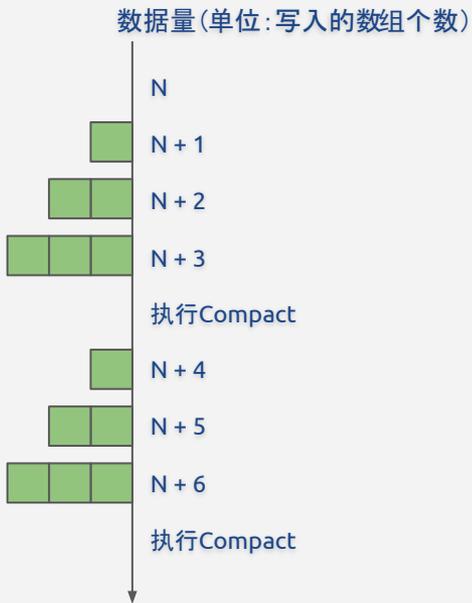


在实施截取尾部的策略之后，由于 L0 的小文件一直新增，无法彻底切除，较多的小体积的数组文件使得性能下降。



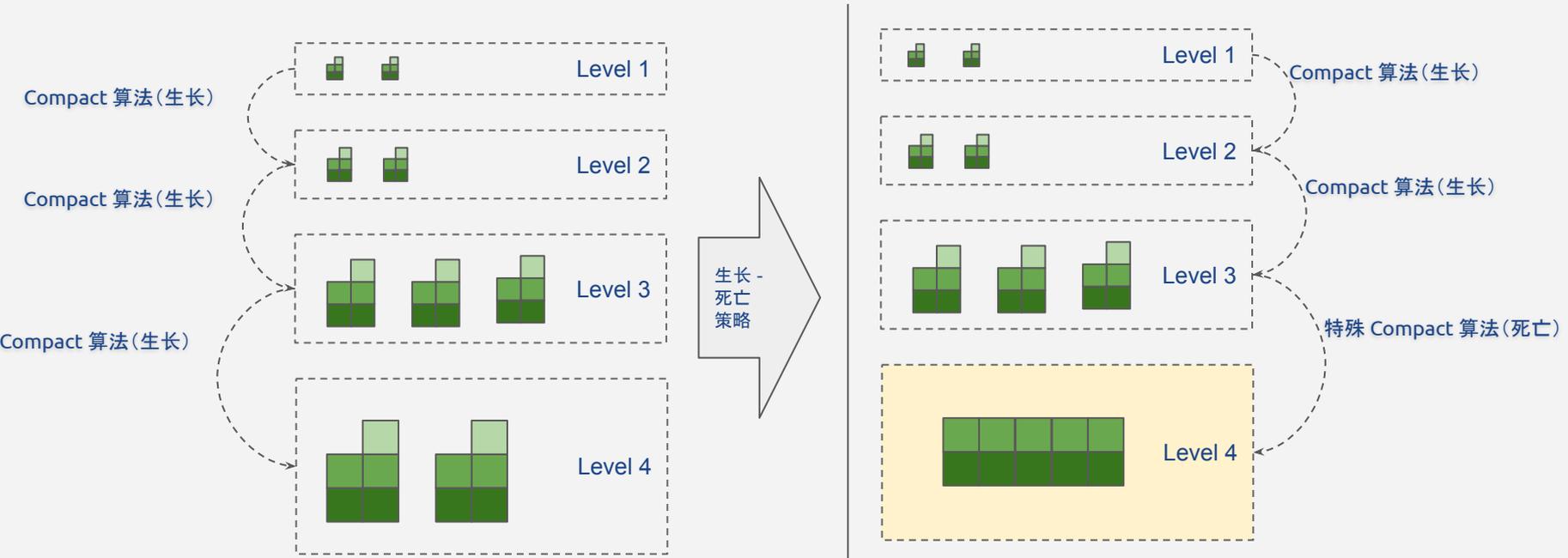
对 L0 文件使用不同的策略，可以把 L0 数量控制在固定数字以下，并且让除了 L0 以外的主体数据的符合期望，保持稳定形状。

# L0 的 Compaction 策略



对 L0 使用(按写入量)周期性触发的 Compaction 策略,  
使得 L0 是相对稳定的、锯齿形的数量

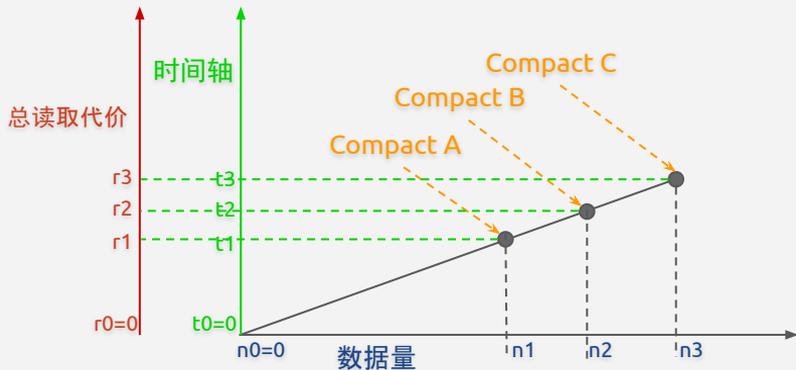
# 生长 - 死亡 策略:对最大层特殊处理



最大层拥有绝大部分数据，  
它的数据形态对性能影响重大。  
默认情况下它由生长算法(Compact 算法)产生

对最大层的生成使用特殊算法，使其有最佳的性能。  
风险：需要谨慎处理，以符合动态截尾，否则整体形状容易随数据量增大而退化

# 寻找最佳的 Compaction 的时机



设想这样的情况：

我们系统的写入、读取在时间分布上是均匀的，  
随着数据的写入和增长，

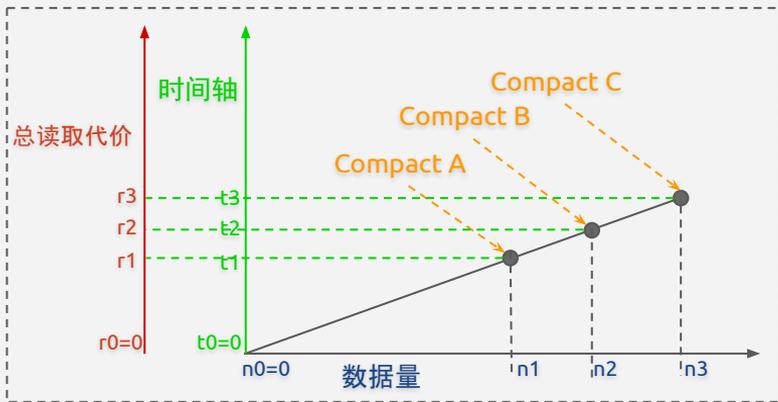
在完美的时间点上执行了多次 Compaction，使得总读取代价最优。

我们期望单次读取代价是稳定的，  
因此总读取代价与时间是线性关系。

由于写入是稳定的，数据量也与时间成正比。

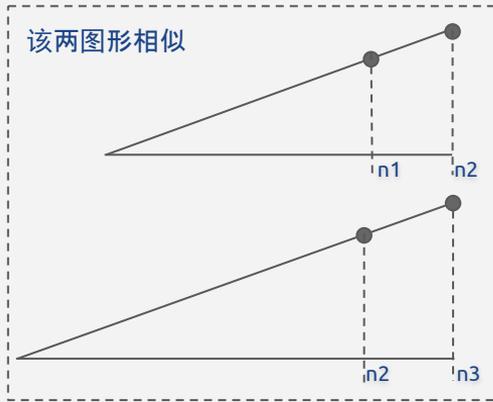
如图，在  $t_1$  时数据量为  $n_1$ ，执行了 Compact 动作 A，以此类推

# 最佳 Compaction 时机：按增长百分比触发



由我们刚才讨论的数据形状的稳定性、相似性，  
我们可以推出该两图形相似。

具体的含义是：我们希望引擎的性能稳定，因此希望存储的数据集的形状是稳定的，而且不管在什么数据量下，Compaction 的结果也是相似的。



我们刚才假设了条件：系统的写入、读取在时间分布上是均匀的。  
这个条件可以化简，  
只需要写入量、读取量是线性关系即可，而这在大部分时候都成立。

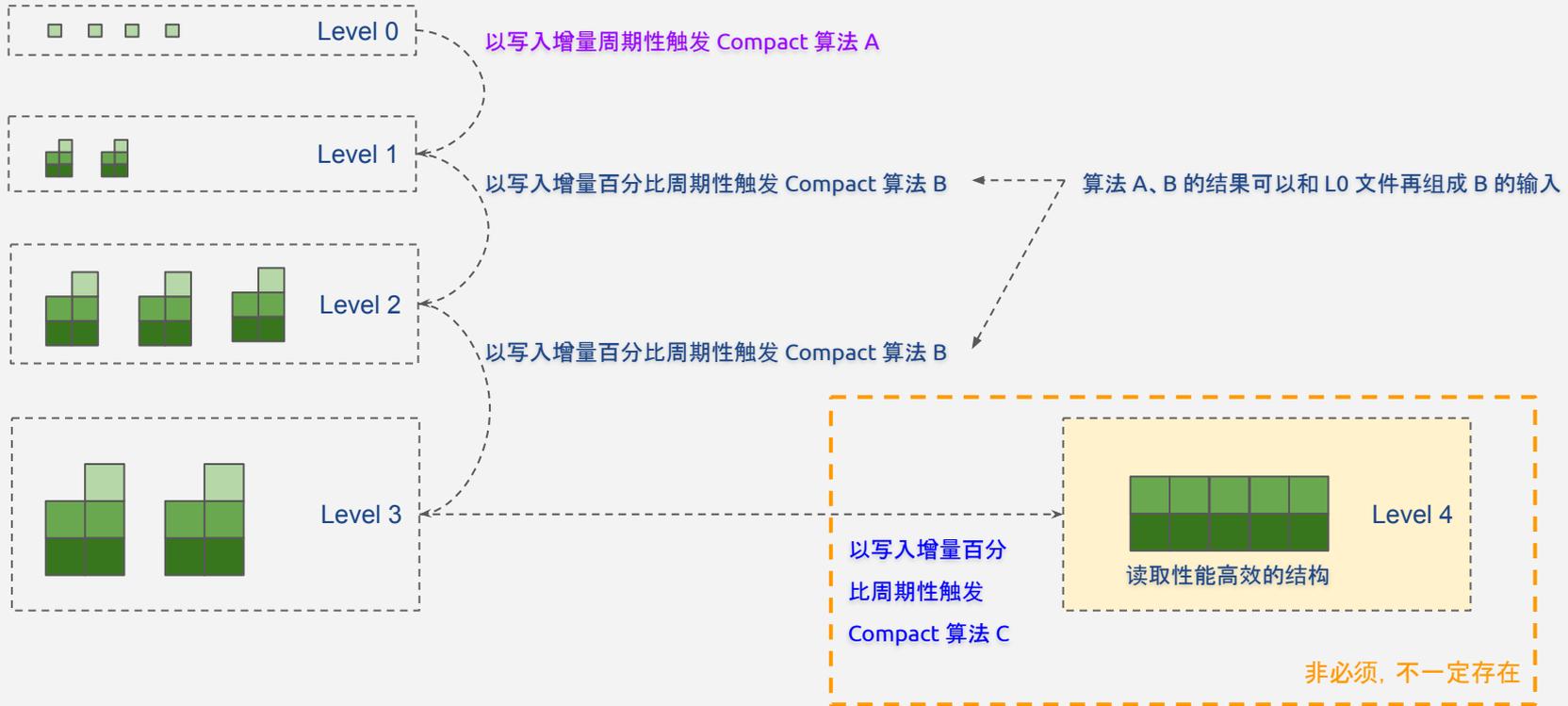
将增长百分比设低，则 Compact 积极执行。设高则消极。

从相似关系和几个变量的线性关系，我们可以推出：  
最佳的 Compaction 时机，是数据量的等比数组。

也就是说，每当数据量增长了固定的百分比时，  
应当执行 Compact。



# 综合起来, 典型 Compaction 策略



(•'•••)

(..•\_•..)

Compaction 是 LSM Tree 最重要的事情

做好 Compaction

我们就有了性能良好的存储引擎了

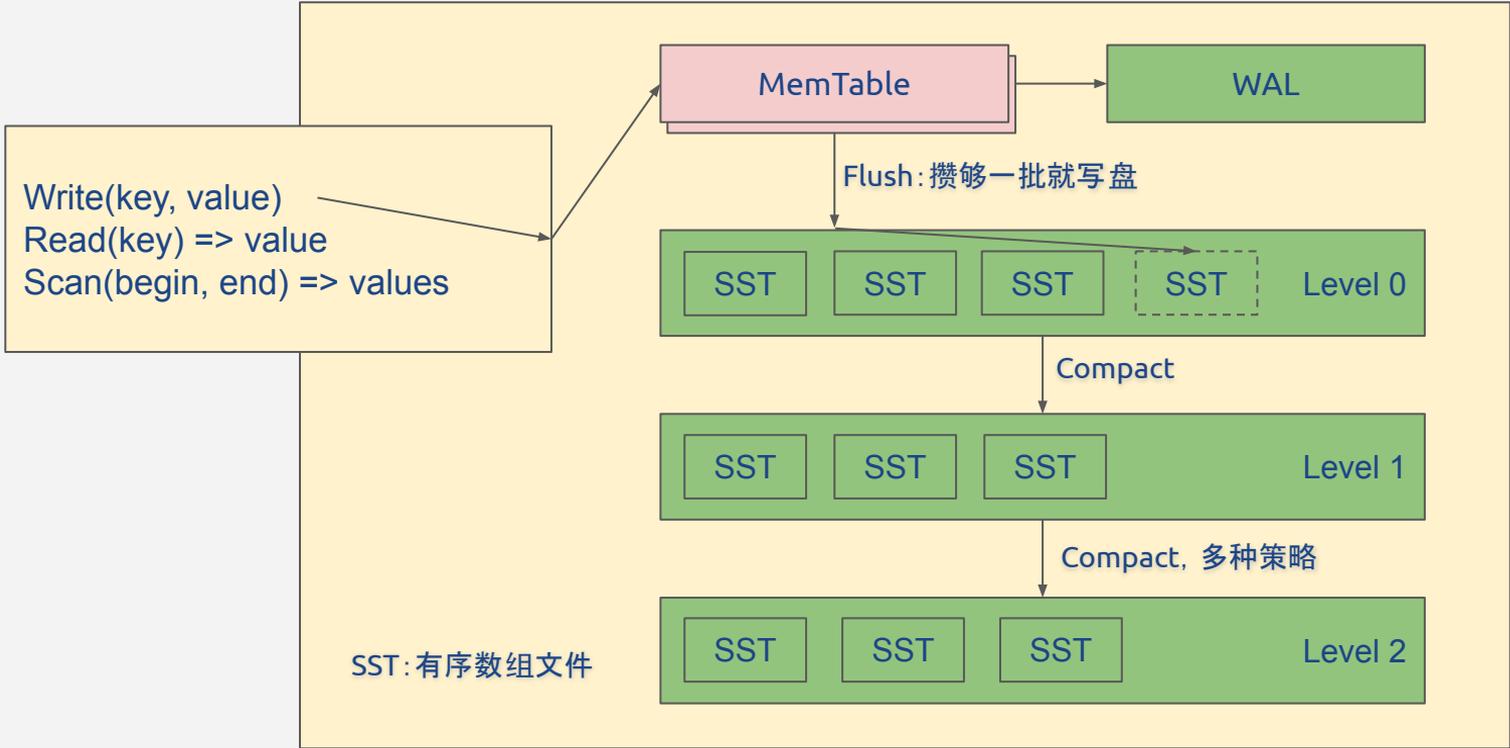
大概明白 LSM Tree 是怎么回事了

业界也是这么做的吗

那我们来看看业界的做法

顺便对比关联一下我们的名词称呼

# LevelDB / RocksDB 的简略示意图





# Universal Compaction

前面我们讨论过 **优化过的朴素 Compaction 策略** (逢 T 进一, 进位的时候把更小的都捎带上), 就是 RocksDB 中的 **Size-Tiered Compaction**, 或者叫做 **Universal Compaction**。

最大的问题: 当写入数据重复 key 较多时, Compaction 过程中消除重复 key 的几率较小, 造成了较大的空间放大。

这篇文章有详细的描述: <https://www.scylladb.com/2018/01/17/compaction-series-space-amplification/>

触发方式等更多细节信息, 可以查看官方的文档: <https://github.com/facebook/rocksdb/wiki/Universal-Compaction>

# Leveled Compaction 的设计推演: Read 性能为目标



每次有 L0 数组落地, 都 Compact 到一个大数组, 这样 **Read 的性能最高**。

同时可以 **解决 Universal Compaction 的空间放大问题**。

但是会造成巨大的写放大, 设为 **W**。



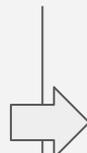
为了缓解写放大, 在 L0 层等待 X 个 L0 文件再执行 Compaction。

频次减轻至  $1/X$ , **写放大相应减轻至  $W/X$** 。

数组个数上限增加到 X, 读性能下降:

由于绝大部分数据都在 Compact 后的大数组, 在 Bloom Filter 的帮助下 Read 的性能下降不多。

Scan 由于不能受益于 Bloom Filter, 性能下降比 Read 快, 但由于大数组占绝大部分数据, 不算太差。



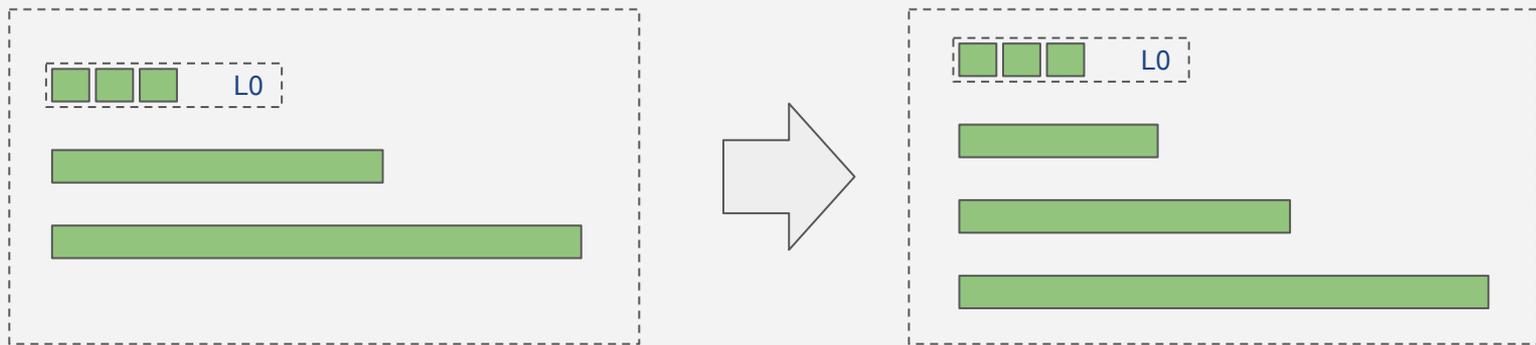
在目标层和 L0 层之间增加一个 T 分之 1 目标层大小的攒批层: 由于攒批层的大小只有目标层  $1/T$ , 则 Compact 到攒批层的代价为  $W/X/T$ 。

攒批层 Compact 至目标层的频次降到  $1/T$ , 当 T 较大时, 该成本约为  $W/X/T$ 。

当 T 较大时, 绝大部分数据都在最大层, 性能下降如左边所述。

综上, 性能下降不多, 总成本降低很多。

# Leveled Compaction: 更多层, 逐层攒批

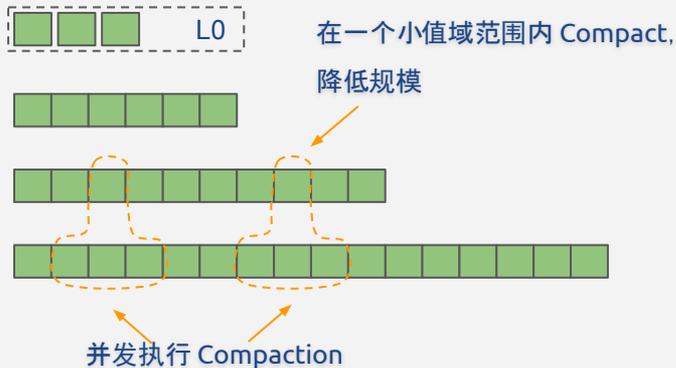


将 1 + 2 层扩展到 1 + L 层。

由之前的计算, 随着层数增加, 我们知道写代价快速下降, Read 的性能下降不多。

但要留意 Scan 性能下降速度比 Read 的更快。

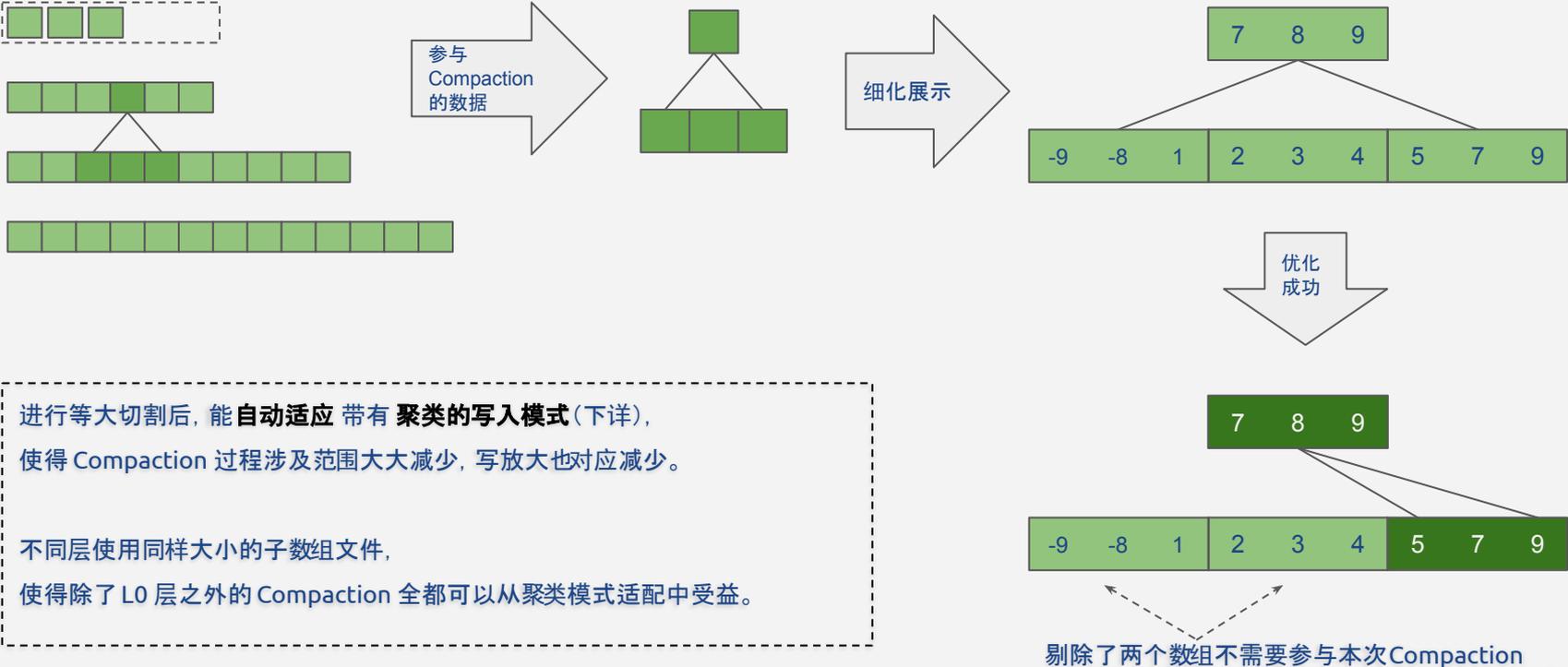
# Leveled Compaction: 等大切割数据



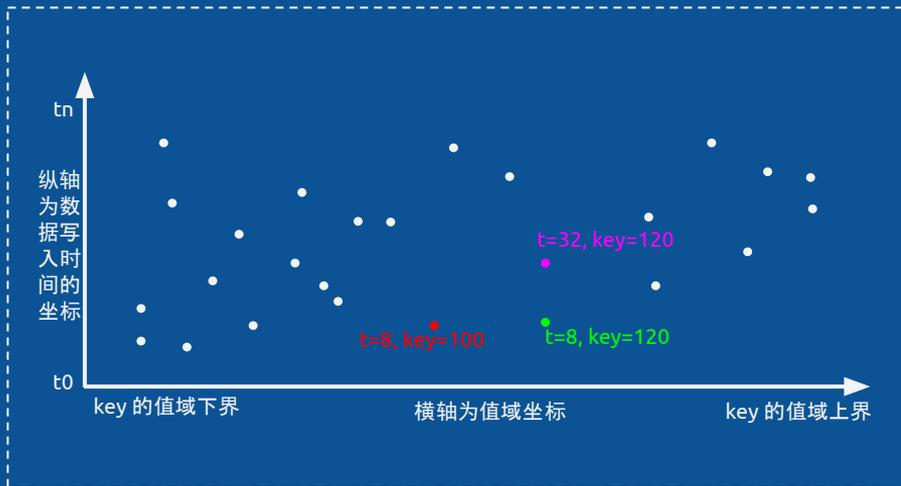
将 L 个层的每个数组切割成等大的子数组文件,  
在非 L0 的 Compaction 过程可以:

- 更小规模、小范围地执行 Compaction
- 并发执行 Compaction

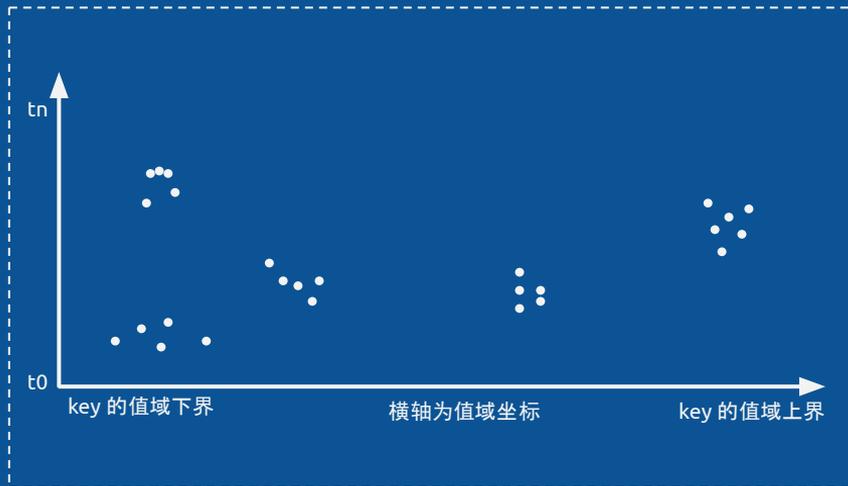
# Leveled Compaction 的等大切割, 能自适应聚类写入



# 聚类的写入模式



随机的写入模式：  
每时刻写入的数据在值域范围内几率均等

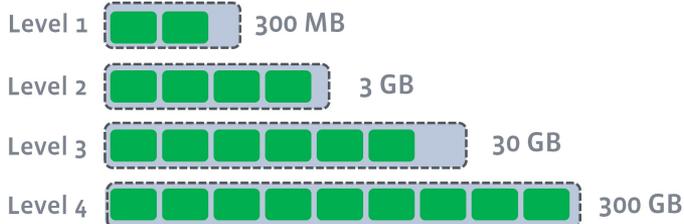


聚类的写入模式：  
一段时间内写入的数据落在值域上比较窄的范围内

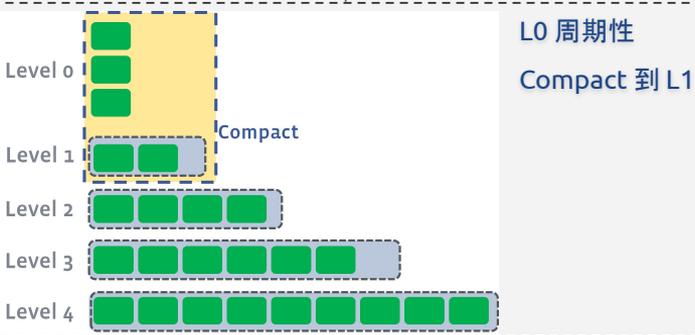
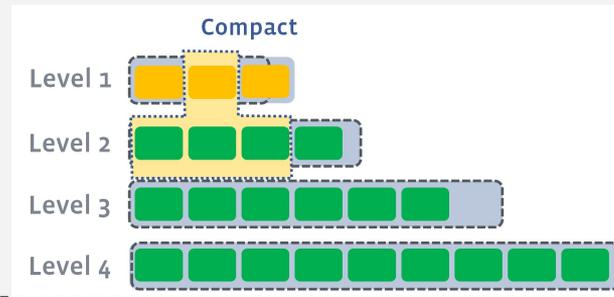


# Levelled Compaction 的触发时机

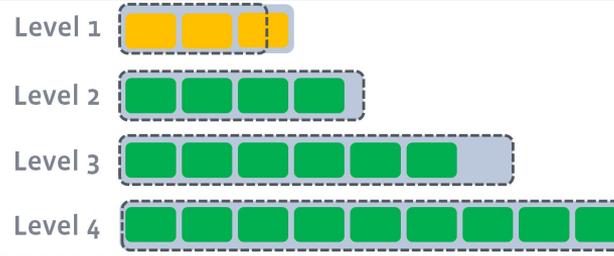
除了 L0, 每层有期望大小(上限)。  
按生长策略, 各层上限为等比数列。  
按截尾策略, 上限根据总容量动态调整。



与下层值域范围重叠的 SST 进行 Compact, 结果放在下层。若 Compact 后下层超出上限, 再次触发



每层容量超出上限时, 触发 Compaction





# Leveled Compaction 小结

Leveled Compaction 目标是尽可能高的 Read 性能。

附带地, Scan 性能也不差(不比 Universal Compaction 差)。

为了这个目标, Leveled Compaction 期望把大部分数据都合并到一个不重叠的数组文件中。

使用多层策略, 逐层攒批, 把写放大降低到可以接受的程度。

写放大详细的描述: <https://www.scylladb.com/2018/01/31/compaction-series-leveled-compaction/>

更细节的成本量化:

Dostoevsky: Better Space-Time Trade-Offs for LSM-Tree Based Key-Value Stores via Adaptive Removal of Superfluous

Merging <https://stratos.seas.harvard.edu/files/stratos/files/dostoyevski.pdf>

对于 Compaction 的触发与执行流程, 可以参考官方文档: <https://github.com/facebook/rocksdb/wiki/Leveled-Compaction>

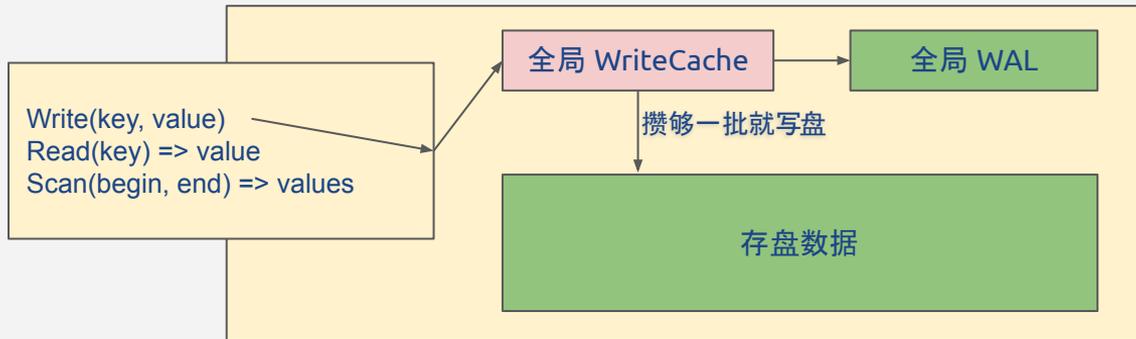
# 回过头来看, LSM Tree 解决了 B+Tree 的什么问题

B+Tree 的几个问题:

- 写放大高
- IO 行为小而散
- 元数据修改频繁

之前分析过, 这些问题很大程度是因为 B+Tree 的攒批能力差引起的,

LSM Tree 使用了全值域攒批的方式, 基本解决了这几个问题。

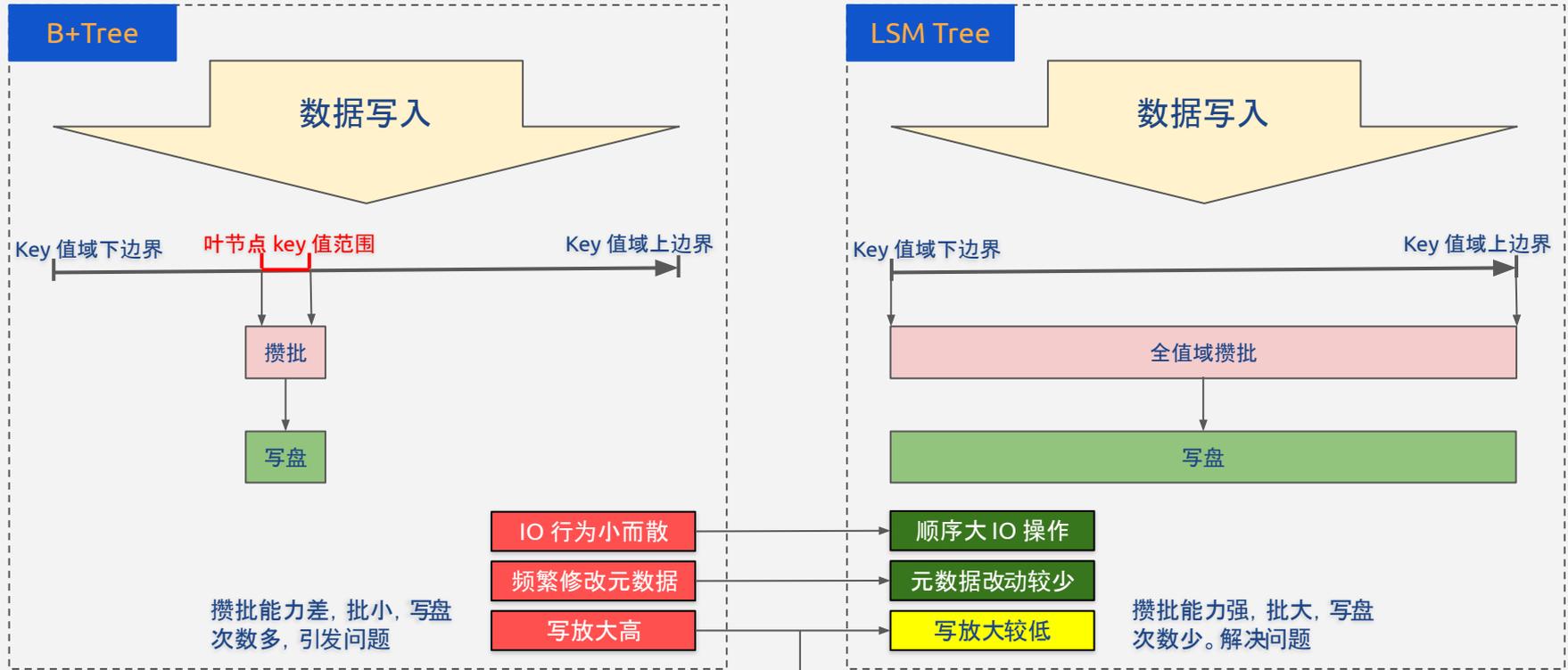


近来新的存储引擎, 大部分都使用全 值域攒批

包括: LSM Tree, Kudu, ClickHouse, 等等



# LSM Tree 解决了什么问题



写放大与数据量  $n$  成线性关系 VS 写放大与  $n$  为  $\text{LogN}(n)$  关系,  $N$  为邻层比



# LSM Tree 带来了什么问题

层层 Compact 带来的写放大:

- Universal compaction 接近最优的写放大, 但还是比较高, 而且存在 GC 问题
- Leveled compaction 有较好的读性能, 但写放大更高(取决于写入模式)

对比 B+Tree, 较低的读取性能:

- 极度依赖于 Compaction 进度, 是否能形成良好的结构
- Read 操作需要读取多个数组文件, 有读放大
- 多路合并 Scan 时有可能降级为 CPU Bound 的性能, 低于 B+Tree 的 IO Bound

以及: 周期性 Compaction 带来的背景 IO 流量峰, 影响前台写入

# CPU Bound 与 IO Bound

CPU Bound: 系统运行时, CPU 是资源瓶颈

IO Bound: IO 是资源瓶颈

那么是哪个更快呢? 都有可能, 取决于硬件配置。

因此, 如果两个系统 A、B 分别为 CPU Bound 和 IO Bound,  
那么它们的性能是不可比的, 得不出 **A 性能是 B 的百分之几** 的判断。

但是, 近年 CPU 提速缓慢, 而 IO 设备的发展则从 HDD、SSD 至 NVM 一路狂奔,  
因此, IO Bound 的系统从发展中的受益更大, 几乎一定优于 CPU Bound。

# LSM Tree 写放大的影响

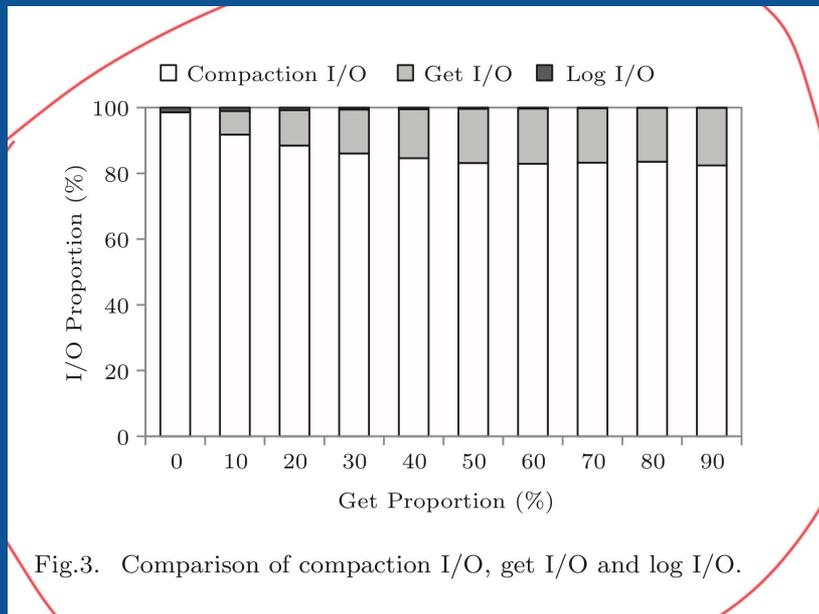


Fig.3. Comparison of compaction I/O, get I/O and log I/O.

对于通常意义的存储引擎来讲, IO 资源是主要消耗。  
而消耗的 IO 资源绝大部分在 Compaction 造成的写放大中。

如图是 LSM Tree 的 Compaction 的 IO 开销。

来源: dCompaction: Speeding up Compaction of the LSM-Tree via Delayed Compaction



# LSM Tree 的优化方向: 工程优化

工程优化指与用户使用模式无关的优化。

例如内存的使用优化, CPU 的优化等等。

以及针对新硬件的优化, 尤其是 IO 硬件的优化。

近年 NVM 开始投入使用, 它对局部性的要求比以前的 SSD 低了非常多, 之前为了局部性而妥协的很多策略、结构都可能重新设计。



# LSM Tree 的优化方向：平衡调优

我们知道写放大与读性能是 LSM Tree 中最尖锐的矛盾，  
如何针对用况进行自使用，采取最合适的 Compaction 策略，就是平衡调优。

以下文章和论文给出了 Sized 和 Leveled Compaction 的各种操作的量化比较，  
也列举了很多自动调优、自适应的 LSM Tree 系统：

- 唐刘@简书: Dostoevsky: 一种更好的平衡 LSM 空间和性能的方式  
<https://www.jianshu.com/p/8fb8f2458253>
- Dostoevsky: Better Space-Time Trade-Offs for LSM-Tree Based Key-Value Stores via Adaptive Removal of Superfluous Merging <https://stratos.seas.harvard.edu/files/stratos/files/dostoyevski.pdf>



# LSM Tree 的优化方向：模式匹配

不少时候用户使用存储引擎有特定的模式，  
例如在值域 Append 写入，例如时序数据库等等。

为特定的模式专门设计数据结构和策略，可以大大提高性能。

这个优化方向的上限应该是最高的。



# LSM Tree 的优化方向小结

粗略的归类：

- 纯工程优化
- 读写平衡与自动适配自动调优
- 为特定模式专门设计数据结构和策略

更多细节参考: **Bokang Zhang: LSM-based Storage Techniques: A survey**

[https://docs.google.com/presentation/d/1pEtCbEjLaHJB2FDRfa0izuQ9xzxTcyG6OPXsJABSpY8/edit#slide=id.g446c4deb4d\\_0\\_353](https://docs.google.com/presentation/d/1pEtCbEjLaHJB2FDRfa0izuQ9xzxTcyG6OPXsJABSpY8/edit#slide=id.g446c4deb4d_0_353)

以及: **LSM-based Storage Techniques: A Survey**

<https://arxiv.org/pdf/1812.07527.pdf>



# B+Tree 和 LSM Tree 的对比和小结

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**



# 今天的小结

我们讨论了 B+Tree 存在的攒批问题,

然后学习了 LSM Tree, 看它是如何解决攒批的问题,

以及它带来的新问题: 读写放大。

我们看到了在一个系统中大量的 **取舍与平衡**。

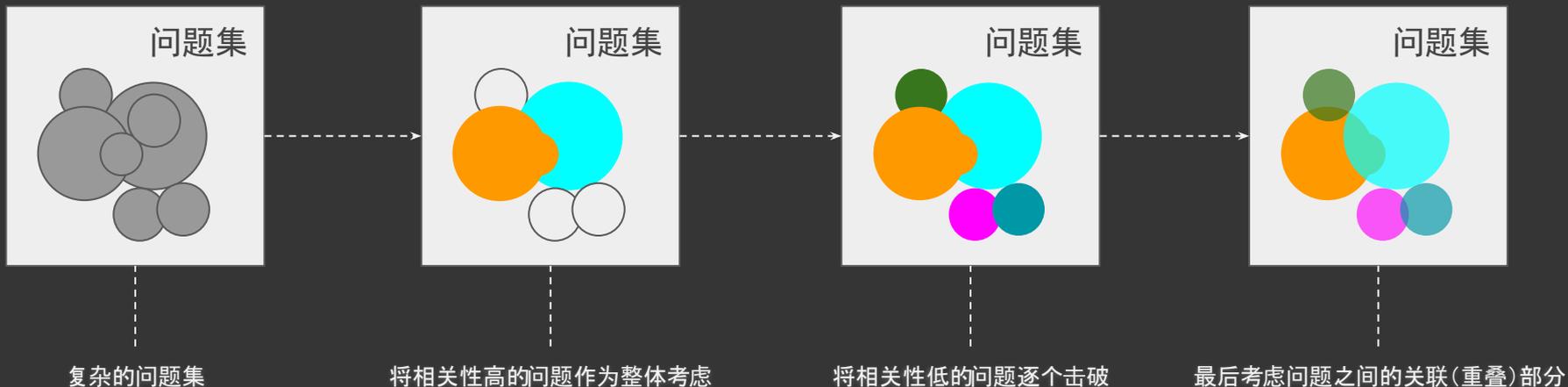
接下来我们还会探讨更多的存储结构, 以及它们和 B+Tree、LSM Tree 的对比。

今天的故事先到这里。

# 设计过程中的解耦

特意地，有些重要的话题我们没有提，例如删除接口、事务实现、列存、压缩，等等。

我们希望能把复杂的问题集分解，逐个击破。



# (新的一天)



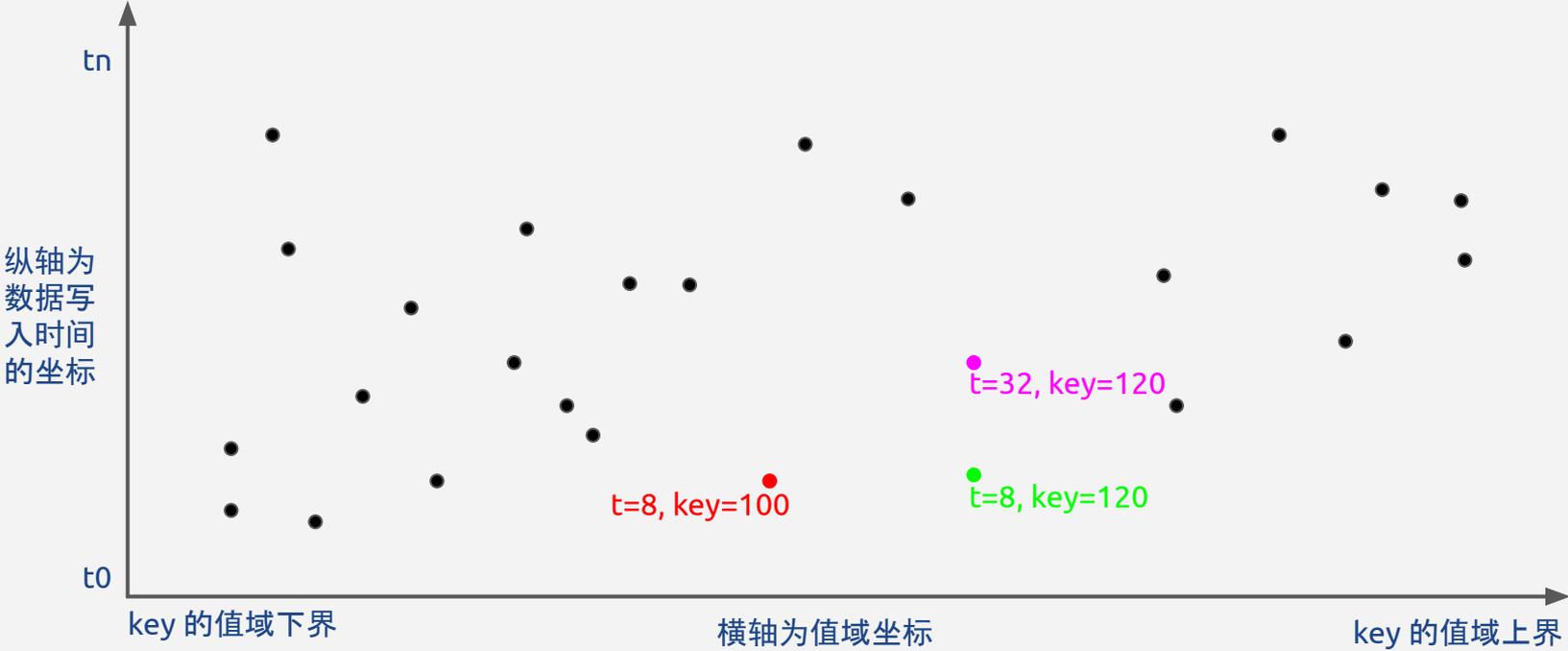
在我们看过两个经典的存储引擎之后  
一个自然而然的问题是：还有别的引擎吗  
这么多种引擎，有分类方式吗

在论文 Fast Scans on Key-Value Stores (<http://www.vldb.org/pvldb/vol10/p1526-bocksrocker.pdf>) 中，  
把存储引擎分成了三类

- Update In Place
- LSM Tree
- Delta Main

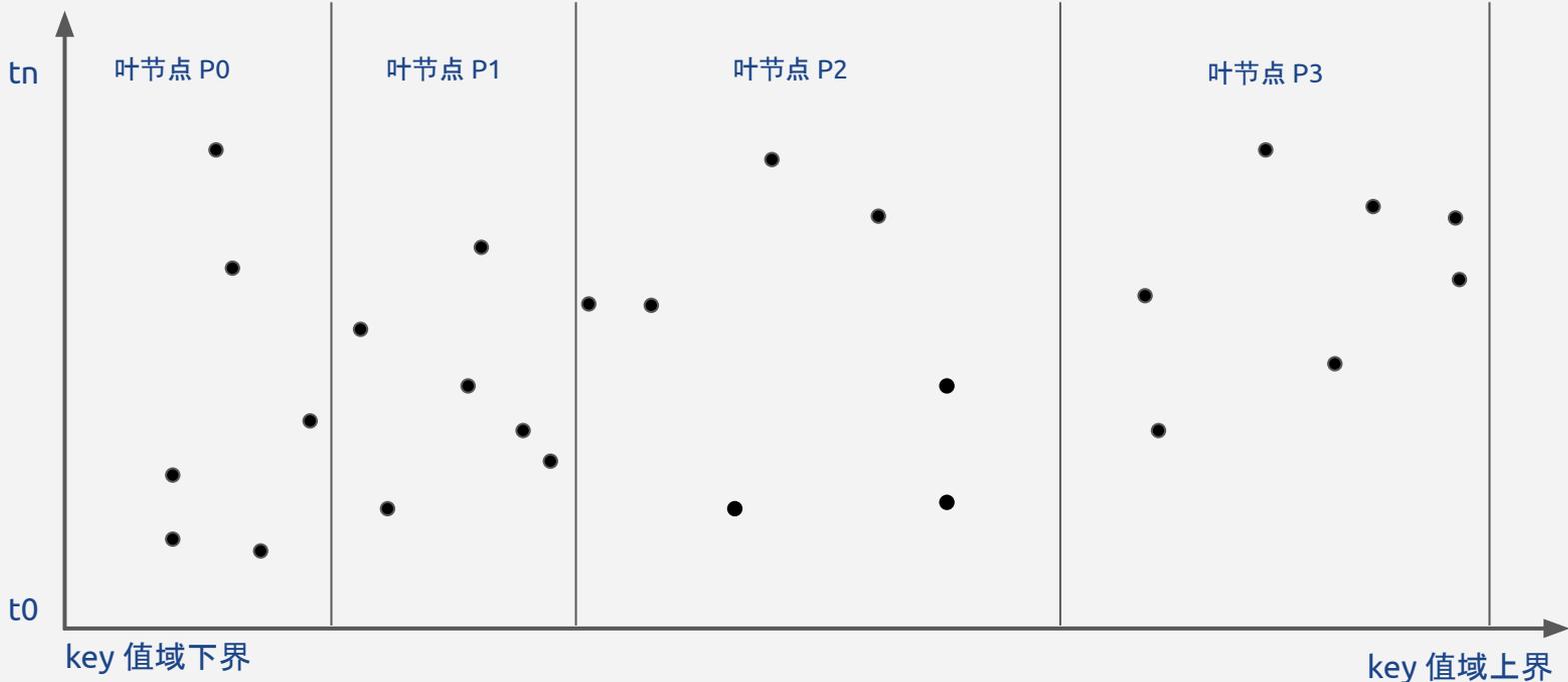
这个分类方式对不对呢？Delta Main 引擎是什么？我们带着问题来一起考察一下

# 我们来观察一下数据的写入



# B+Tree 对数据进行了值域切割, 不进行时域切割

(以数据量为纵轴) 等高切割, 每个叶节点数据量接近



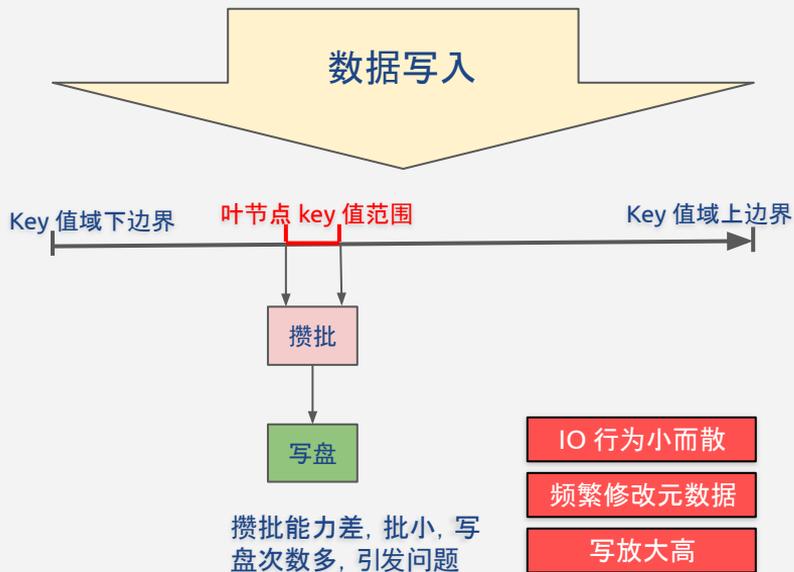
# 值域切割的存储引擎，有共同的特征，归为一类

容易看到，因为进行了值域切割，  
B+Tree 很难利用全局攒批的策略来进行优化。

值域切割都容易碰到攒批能力不足的问题，  
导致：**IO 小、散，元数据 OPS 高，写放大高。**

优点：数据都有 **立即可取** 的值，不需要额外计算，  
因此 **读取性能很高**，Scan 能达到 IO Bound。  
数据可以容易地 **物理删除** (相对于仅打删除标记)。

基于这些特性，我们可以为其分出一个类别：  
值域切割引擎。



(●'∪'●)∪

(∪∪∪∪∪∪)

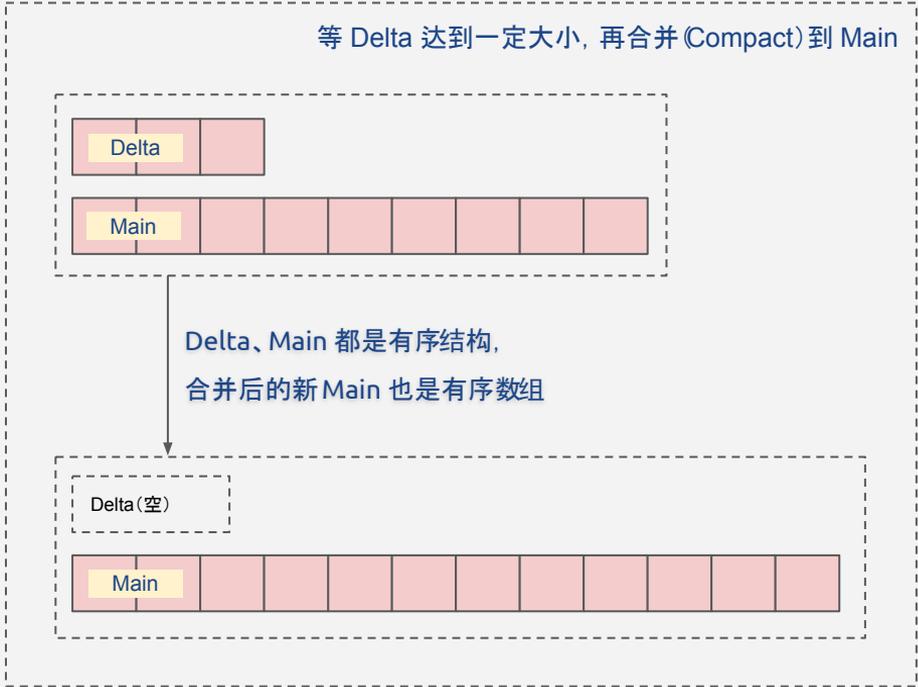
除了 B+Tree 引擎还有别的值域切割引擎吗

当然不止一两种了

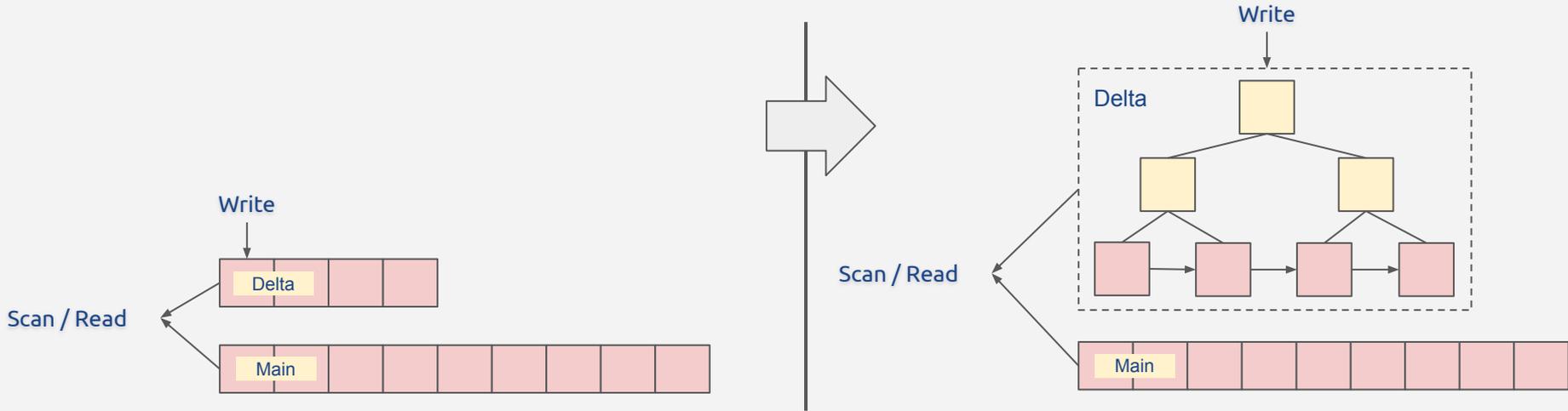
我们来学习一下 DeltaMain

它与 B+Tree 一样, 也是值域切割的存储引擎

# 另一种局部性高的有序结构: DeltaMain



# DeltaMain 中 Delta 的数据结构



Delta 需要响应 Scan、Read, 因此必须有序

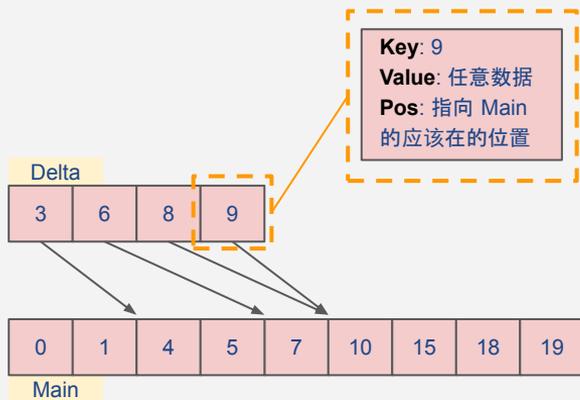
Delta 的作用是写缓存, 需要插入友好

综合考虑, Delta 应该是局部性较低(插入成本低)的有序结构,

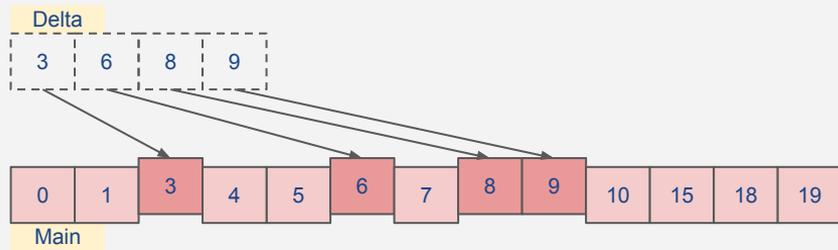
可以是 B+Tree、跳表等任意结构



# Delta 数据的 Position 指针

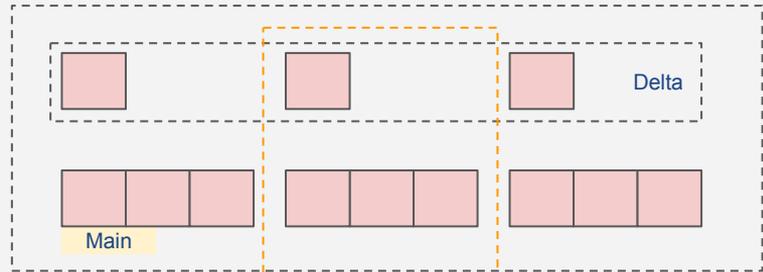
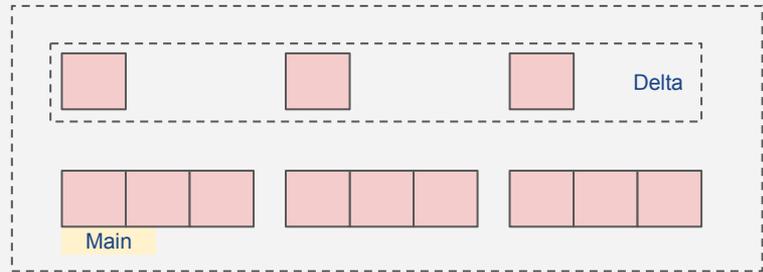
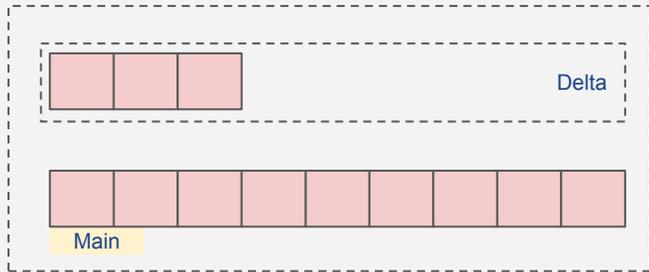


当 Compact 到 Main、或响应 Scan 时进行合并



与 LSM Tree 的多路合并有 **多个有序数组** 参与不同, DeltaMain 只有 **两个** 有序结构, 而且 Delta 的体积远小于 Main, 因此可以在每一个 Delta 数据项上增加 **在 Main 上的插入位置的信息**, 从而加快合并过程, 不需要进行多次 key 比较当然, 不使用 Position 指针, 直接以 key 作二路归并也是简单可行的, 性能也不差

# Main 的数据结构



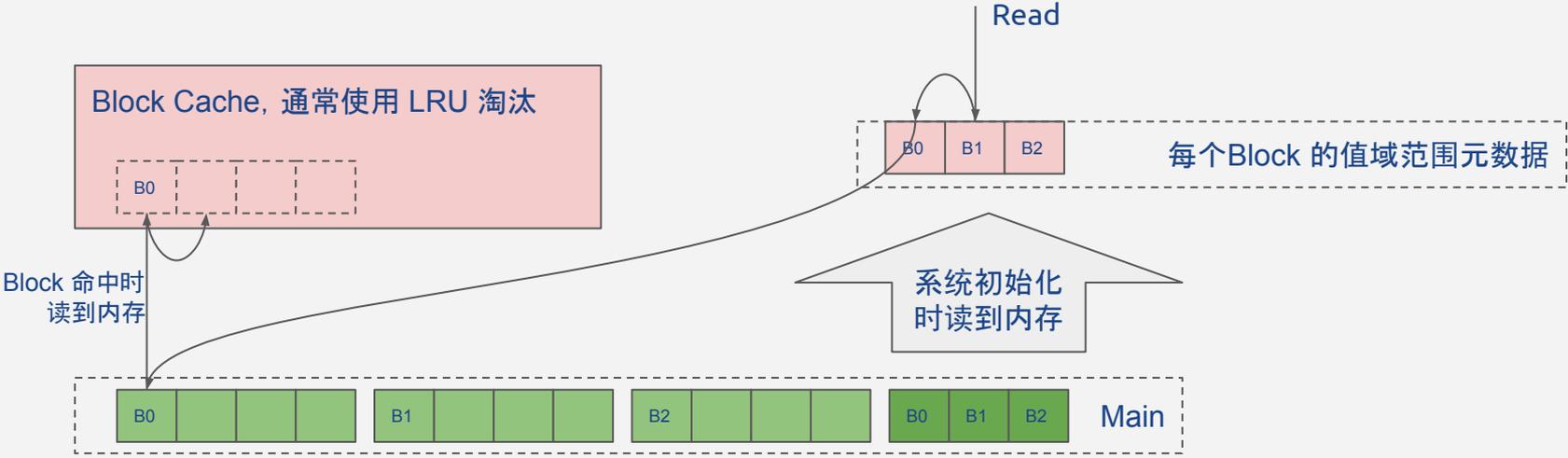
按范围进行Compact

将 Main 有序数组切成多个 Block, 可以:

- 匹配聚类写入
- 降低单次 Compaction 规模
- 减少 Compaction 涉及的范围

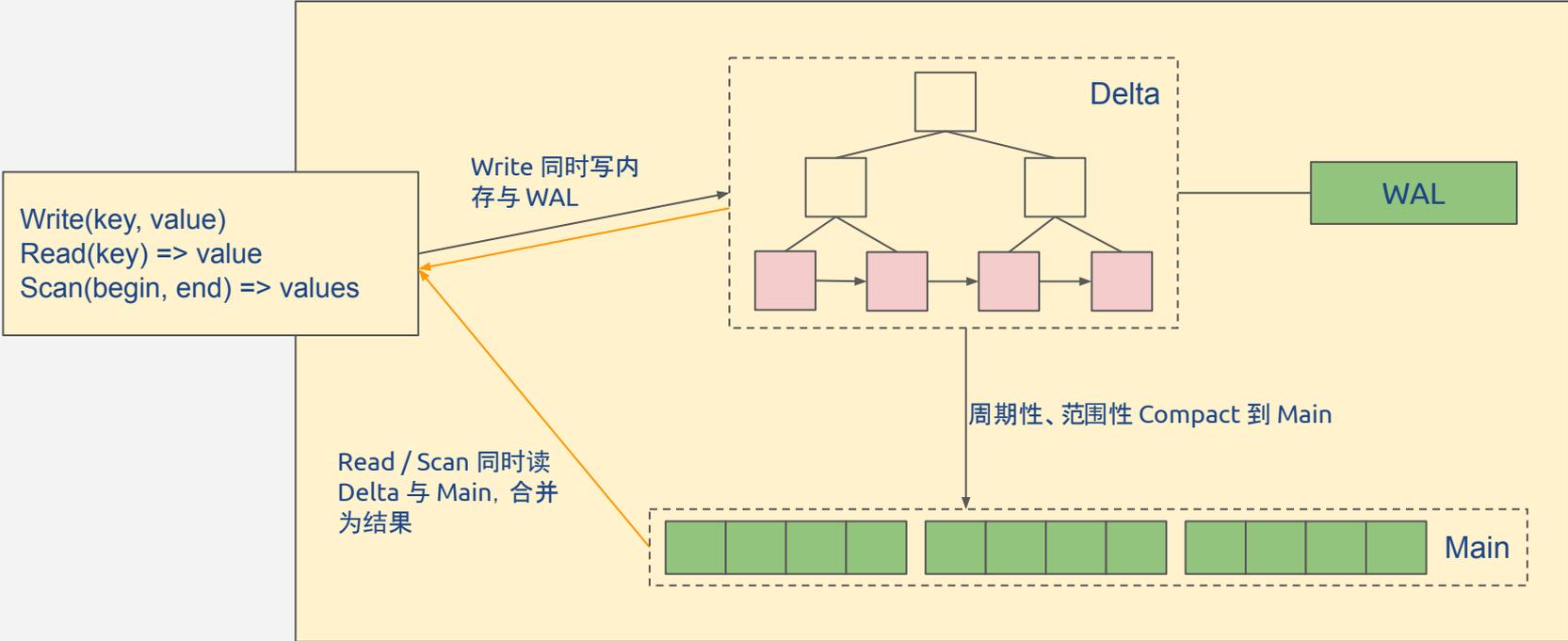
与之前的 LSM Tree 的 Leveled Compaction 的切割类似

# Main 的磁盘数据结构



当 Main 存储在磁盘时, 也需要切割为 Block 才合适加载到内存

# DeltaMain 存储引擎





# DeltaMain 引擎的局部性的量化

对于 Delta 来说, 它主要为 Write 服务, 内部局部性会较低。

对于 DeltaMain 整体来说, 局部性高低取决于 Delta 与 Main 的比例。

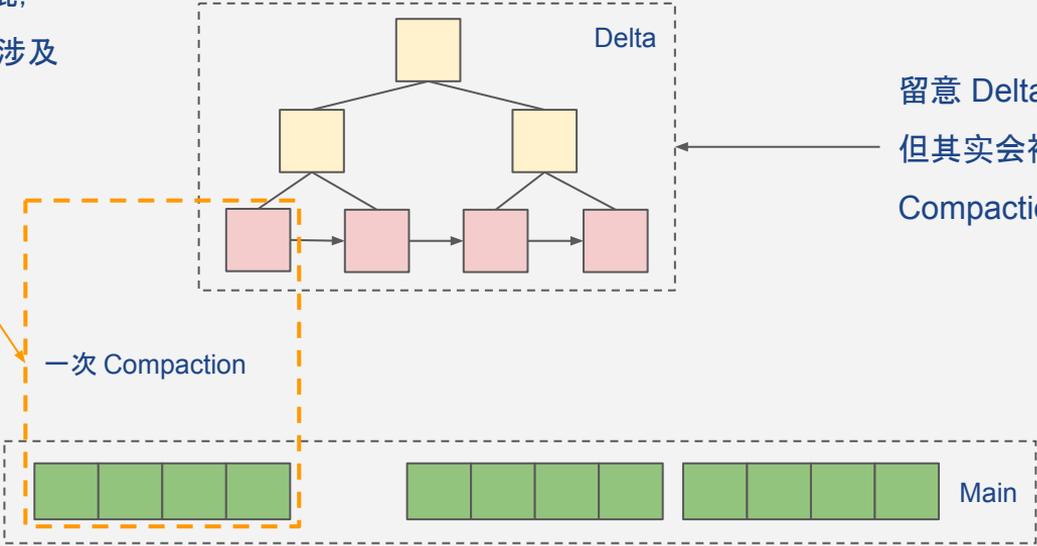
如果 Delta 频繁 Compact, 可以增加整体局部性, 提高 Read / Scan 的性能, 但会造成巨大的写放大。

如果 Delta 尽量延迟 Compact, 除了整体局部性不佳、读取性能下降之外, 还有可能冲击内存资源的上限, 因此不能无限延迟。

我们可以看到, 这些问题与 B+Tree 非常相像。

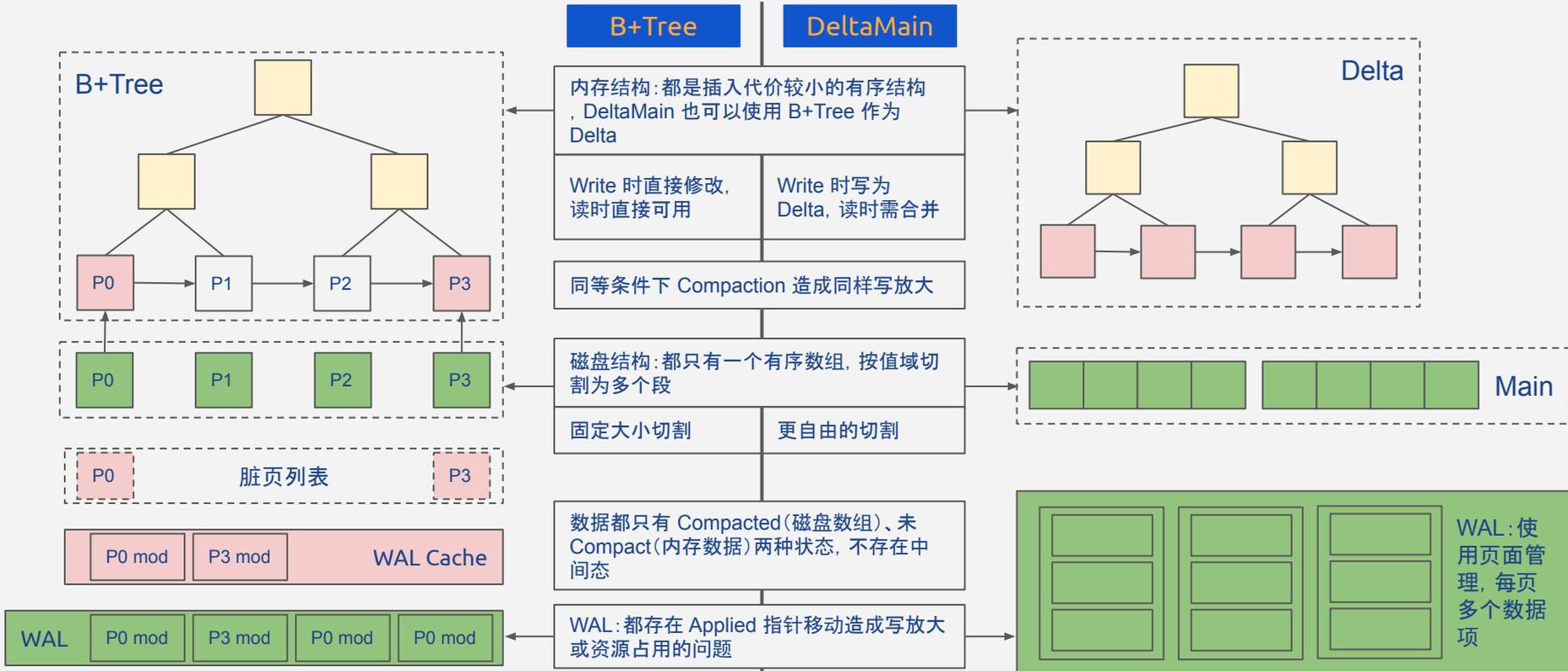
# DeltaMain 是值域切割引擎

Delta 并不是全值域攒批，  
而是根据 Compaction 涉及  
的范围进行攒批



留意 Delta 虽然是连续的，  
但其实会被分拆、归属于其  
Compaction 的值域范围

# B+Tree 引擎与 DeltaMain 引擎对比





# 值域切割引擎的优缺点

DeltaMain 引擎与 B+Tree 引擎一样, 有值域切割的所有优缺点。

缺点: 攒批能力差: IO 小、散; 元数据 OPS 高; 写放大严重。

优点: 读取性能高; 数据容易物理删除。

我们可以看到值域切割的优点非常明显, 主要缺点都是由攒批能力引起的。

如果硬件进一步发展, IO 设备对局部性的要求极大降低, 那么值域切割的价值就大大提升了。

如果运行时内存等资源充足, 有足够的内存空间帮助攒批, 值域切割引擎是很好的选择。

如果能成功预测写模式, 让攒批、Compaction 策略与写模式匹配, 也可以极大降低值域切割的缺点。

举例, 在 OLAP 领域, 写与读通常没有固定的比例, 通常是批量、低频写入,

那么 Compaction 的代价就降低很多, 通常选用 DeltaMain 结构进行存储。

# 写入频率对 Compaction 的影响



我们之前分析过, 同等写入数据量下, Compact 越积极, 写放大(以及对应的 IO 和 CPU 等代价)越大。

在 OLAP 场景下, 写入降频, Compaction 也可以相应降频、减低成本, 从而可以选择值域分割引擎, 获得优秀读取性能。

(•'•••)

(..•\_•..)

明白值域切割的意思了

我们在数据分布图里有两个坐标轴

一个是值域一个是时间轴

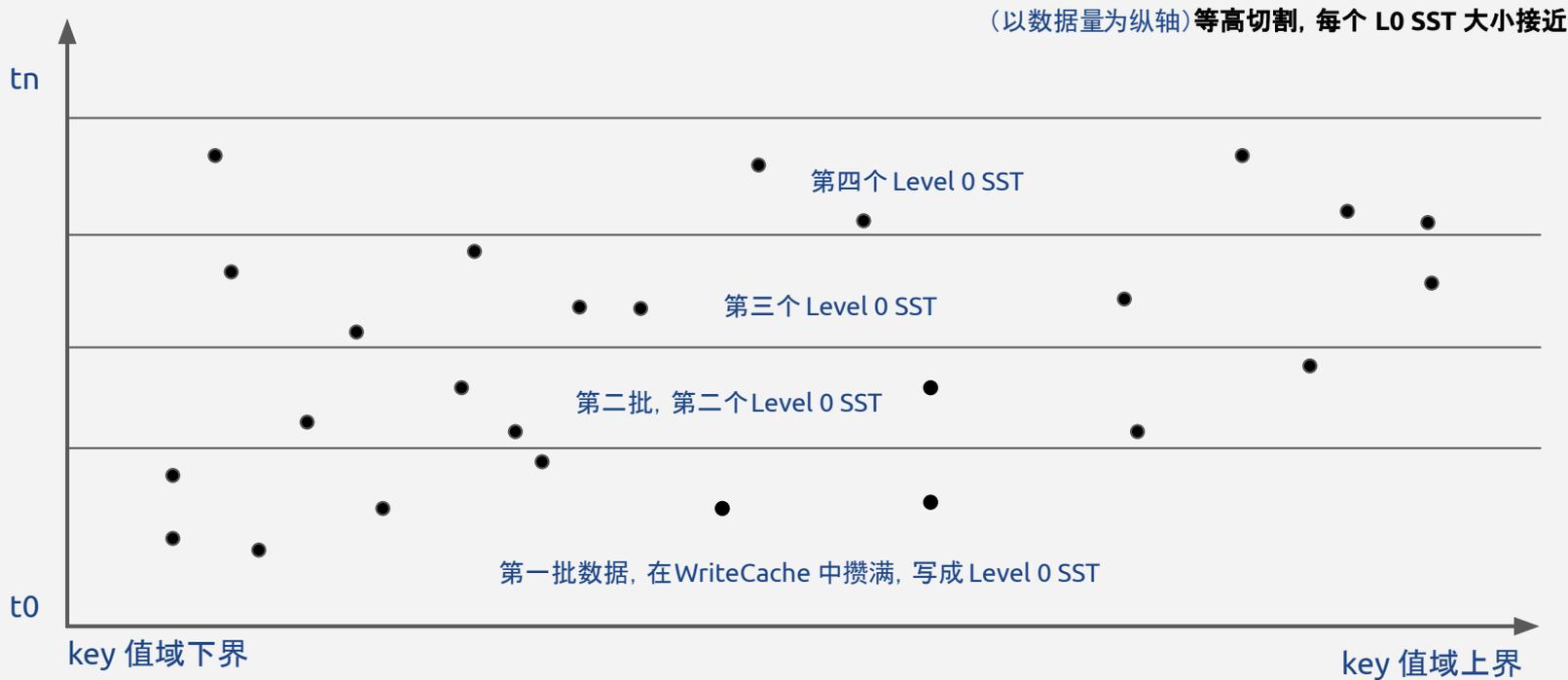
那么是按时间切割数据是一类引擎吗

没错, 这就是时域切割存储引擎

LSM Tree 就是其中一种



# LSM Tree 对数据进行了时域切割



# 存储引擎分类二：时域切割引擎

对于时域切割的数据系统，由于数据在源源不断写入，新数据和之前所有数据都有大几率重叠。

在新数据大几率重叠的情况下，它很不能保证 key 对应的数据只有一个版本，因此 Scan 时必然需要多路合并。

时域切割引擎的 **全局攪批**、**多路合并** (**Read 读放大**、**Scan 时多路合并**)，特征明显，可以为其分出一个类别。





# 时域切割引擎的优缺点

Universal Compaction 策略的 LSM Tree 是时域切割引擎的范本，它不仅使用时域切割，Compact 时也是简单地对不同时域数据进行合并。

因此，它的特性也基本等同于 时域切割引擎的优缺点：

优点：时域切割就是全局攒批，攒批能力最好，不依赖大量内存资源就有良好的写入性能，IO 批次大且连续，对磁盘友好度高。不高的写放大。

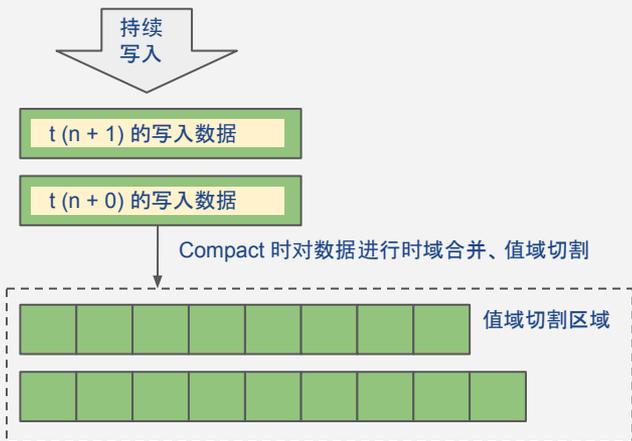
缺点：Read 读放大、Scan 多路合并。

时域切割引擎的优化方向，与之前提及的 LSM Tree 的基本相同。

# Leveled Compaction 的 LSM Tree: 混合切割

对于 Leveled Compaction 的 LSM Tree, 新数据使用时域切割, 旧数据则使用值域切割, 越旧的数据值域划分越细。它用更大的写放大, 换取了更好的读取性能(更少有序数组: 更好的局部性)。

可以发现, 这是先进行了时域切割, 再进行值域切割, 它有时域切割的一切特征, 但也有特有的一些特性。因此, 我们将 **先时域切割再值域切割** 的混合切割, 归入 **时域切割的一个子类别**。

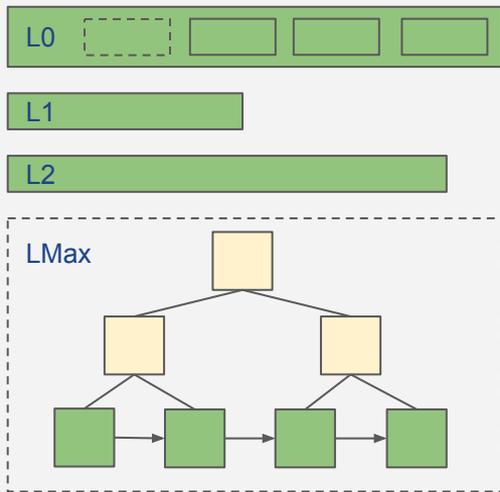


重新再看这篇论文, 它尝试在不同的用况下调整值域切割区域的大小:  
(当然这篇论文所说的远不止这点)

Dostoevsky: Better Space-Time Trade-Offs for LSM-Tree Based Key-Value Stores via Adaptive Removal of Superfluous Merging

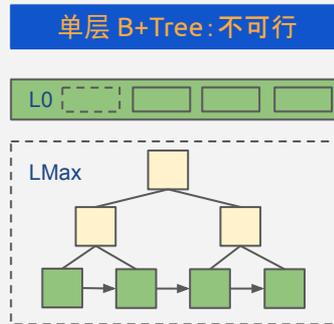
<https://stratos.seas.harvard.edu/files/stratos/files/dostoyevski.pdf>

# 以 B+Tree 代替最大层的 Leveled Compaction



使用一个 B+Tree 来存储 LSM Tree 最大层数据,  
(即 Compaction 策略中的死亡层)

也是一种 **先时域再值域** 的混合切割, 它的性能(可能)在工程上有改进,  
但写放大等基本特性吻合先时域再值域切割的特征, 当然也吻合时域切割大  
分类的特征。



是否可以直接在 L0 下接 B+Tree 呢?

仅使用单层 B+Tree, 在聚类写入的工况下是可行的。

在随机写入的工况下, 每次 Compaction 涉及的范围(也就是叶节点数量)很广。如果不积极 Compact, 那么 L0 会堆积导致性能下降。如果积极 Compact, 那么会造成极大的写放大。

在 L0 与 B+Tree 之间, 增加时域分割的层或者值域切割的层, 都是可行的, 这就回到我们之前讨论的 Dostoevsky 的结构了。



# 先时域再值域切割的特性

首先, 这种混合切割属于时域切割, 继承了时域切割的优缺点:

优点: 全局攒批, 不依赖内存容量即有良好的写性能, 磁盘 IO 友好

缺点: Read 读放大, Scan 多路合并。

作为子类别, 它额外的特性:

- 更高(极高)的写放大。
- 因为数组个数更少, 局部性更好, 因此比纯时域切割的读取性能更好
- 对 GC(回收被删除数据空间)友好, 在有大量修改、删除操作的情况下很有用。
- 能从聚类写入模式中获得较大收益, 减轻 Compaction 成本。



# 从时域、值域关系，分为两大类，一小类

按照写入分布图，有值域和时域两种切割方法。通过刚才的分析，就是**时域切割引擎**和**值域切割引擎**。

为了便于沟通，我们可以也不严谨地叫做**LSM 引擎**和**非 LSM 引擎**。

时域切割引擎 (LSM 引擎)

先时域再值域切割引擎

值域切割引擎 (非 LSM 引擎)

先值域再时域切割引擎  
(是否存在？为了实现 MVCC？)

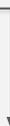


我们讨论了时域、值域切割的分类

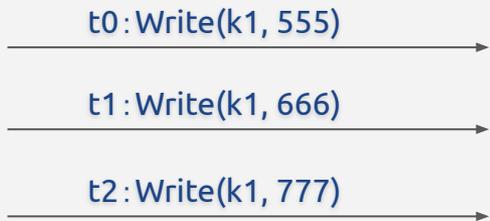
接下来我们引入一个概念：**数据坍塌**

来尝试从另一个角度分析存储引擎的共性和特性

考察新的分类方式

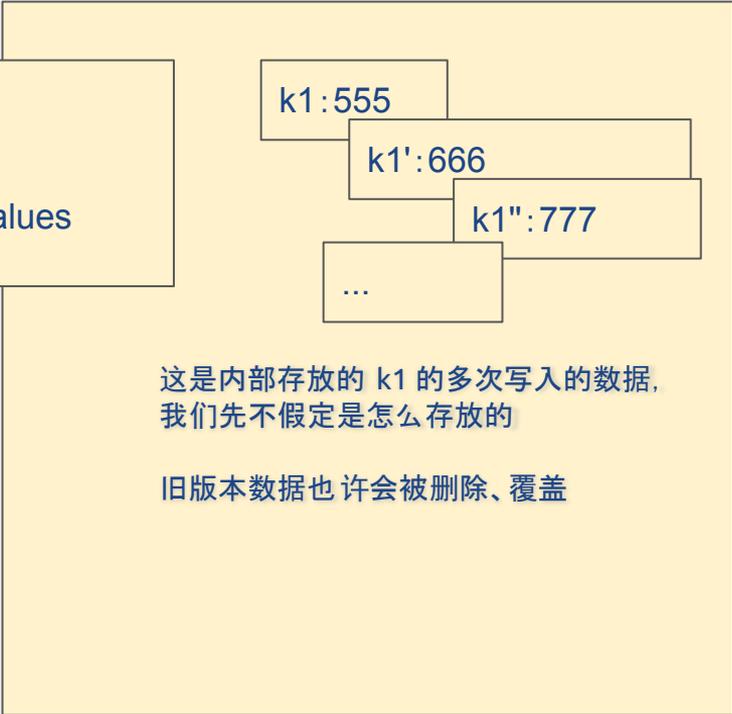


# 我们看一下读写过程，先看写

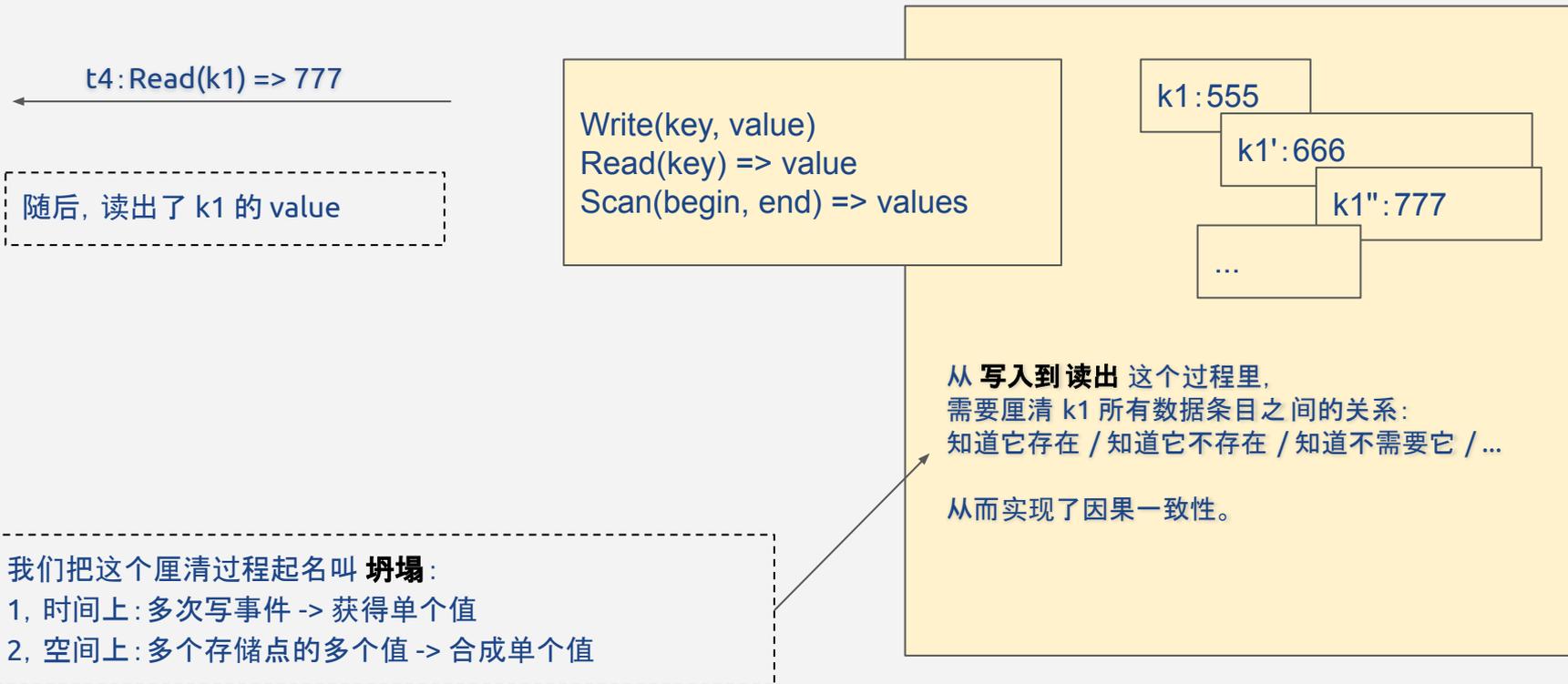


在时间 t0、t1、t2，  
写入了 k1 的不同 value

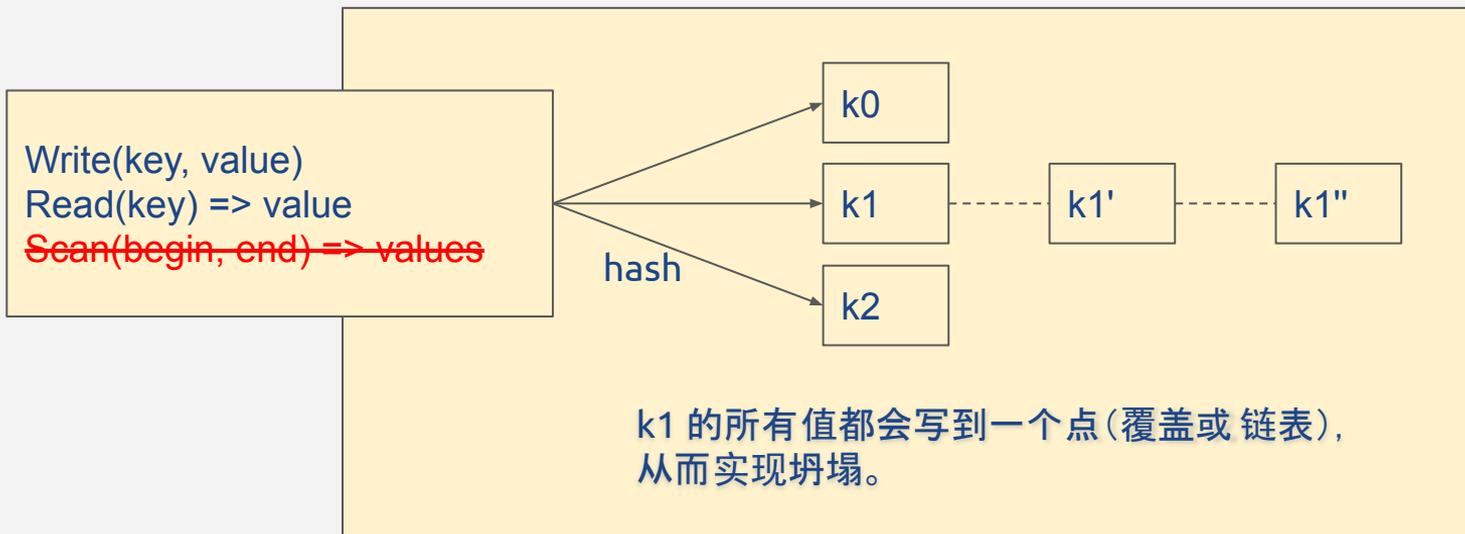
Write(key, value)  
Read(key) => value  
Scan(begin, end) => values



# 读过程



# 举例: Hash Table 中的数据坍塌 (不实现 Scan)





# 回顾几个存储引擎

**B+Tree**: 写入时坍塌

**LSM Tree**: 写入不坍塌, Compact 时实现了部分坍塌、读实现了另一部分坍塌

**DeltaMain**: 写入时坍塌

**Kudu**: 写入时坍塌

**ClickHouse MergeTree**: 写、Compaction、读都不坍塌



# 延迟坍塌

如果引擎在数据写入时不完全坍塌，留给后续的 Compact 和 Read / Scan 过程中共同实现坍塌，那么我们将它分类为 **延迟坍塌** 引擎。

延迟坍塌引擎基本对应于时域切割引擎，有同样的优缺点。

我们从 **延迟** 两字中可以知道到它将成本推迟到了读取，意味着较低的读取性能。



# 即时坍塌

如果引擎在数据写入时实现了完整的坍塌，我们把它叫做 **即时坍塌** 引擎。

如果引擎数据存储在磁盘，由于要满足磁盘的局部性要求，不可能实时修改数据，因此需要把坍塌分 **两步走**：

- 逻辑上的坍塌，在写入时通过旧数据匹配、链接来完成。然后将新数据 攒批写入（与旧数据分离）
- 物理上的坍塌，在 Compact、读取时执行，通过单步骤的简单合并完成

即时坍塌引擎基本对应于值域切割引擎，有同样的优缺点。



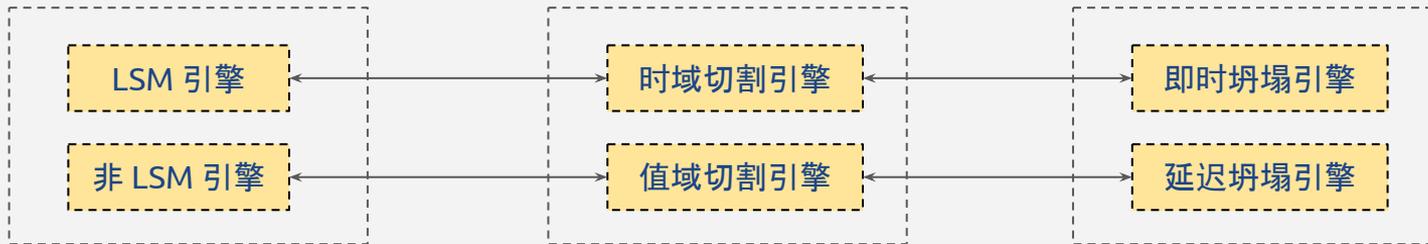
# 即时坍塌引擎的更多特性

对于读取，即时坍塌引擎的每个 key 在每个时刻，都有 **立等可取** 的确定值，因此能支持更多的、依赖于 key 坍塌完成形成的最终值的特性，例如：**预计算**、**预聚合**。

对于写入，即时坍塌引擎需要查找每个写入 key 的旧值，以便于匹配、链接。

这个查找动作，使得即时坍塌引擎天然地支持 Update / Insert 语义（而不仅仅是 Upsert 语义）。但从另一方面来讲，极大概率引发 **读缓存击穿**，导致 **写性能下降**。

# 不同的引擎分类方法





# 坍塌成本

在指定系统中, 平均单个 key 的坍塌成本是固定值

TODO: (推导过程)



# 引擎设计的倾向性

平均单个 key 的坍塌最小成本是固定值

- 在固定的硬件环境下
- 在固定的 workload 下: 写模式(写操作在值域、时域上的分布)固定

基于此, 坍塌的成本在 写、Compact、读 三个操作中如何分配,

是引擎设计的哲学: 为读优化 VS 为写优化

# 未完待续

Coming soon ...

