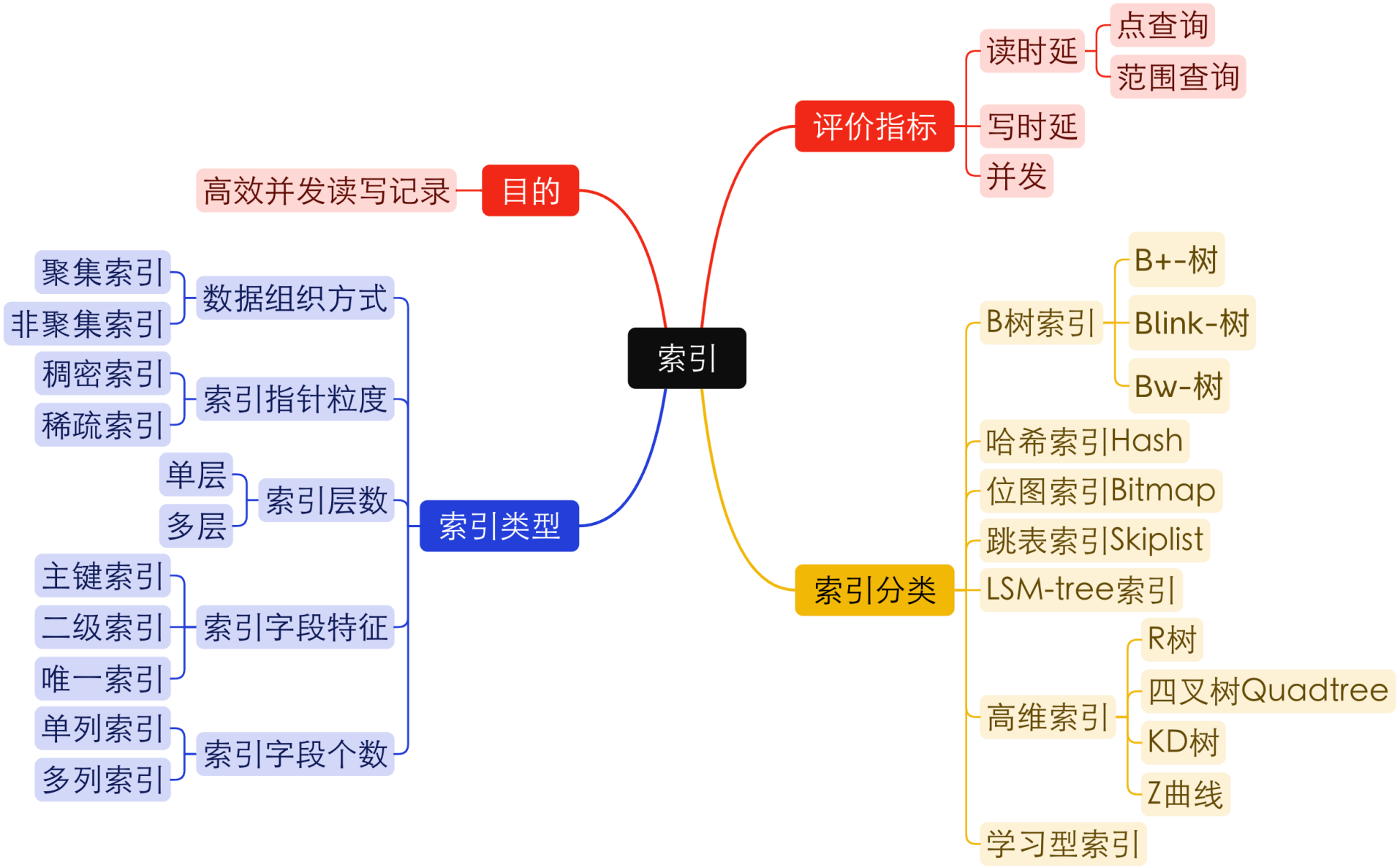
第9章 数据库索引

清华大学数据库组 李国良

liguoliang@tsinghua.edu.cn
http://dbgroup.cs.tsinghua.edu.cn/ligl

索引


索引
目的
高效并发读写记录
评价指标
读时延
点查询
范围查询
写时延
并发
索引类型
数据组织方式
聚集索引
非聚集索引
索引指针粒度
稠密索引
稀疏索引
索引层数
单层
多层
索引字段特征
主键索引
二级索引
唯一索引
索引字段个数
单列索引
多列索引
索引分类
B树索引
B+-树
Blink-树
Bw-树
哈希索引Hash
位图索引Bitmap
跳表索引Skiplist
LSM-tree索引
高维索引
R树
四叉树Quadtree
KD树
Z曲线
学习型索引

目录

1. **索引概述**
2. B+树索引
3. B+树并发控制
4. 哈希索引
5. SkipList跳表索引
6. LSM树索引
7. 位图索引
8. 多维索引

——R树、KD树、Quadtree、Z曲线

目录

1. 索引概述

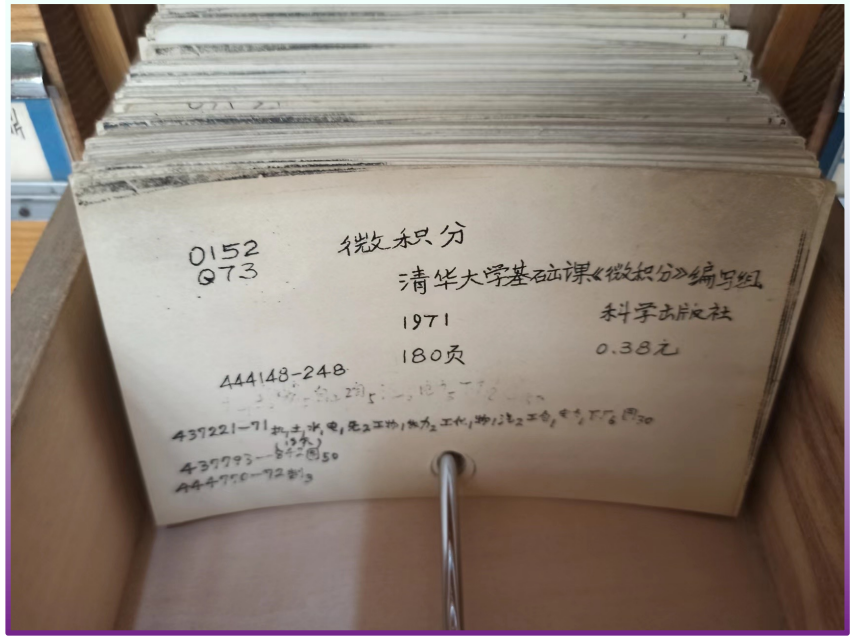
- **索引的目的：加速数据获取速度**
- 聚集索引与非聚集索引：数据是否按照索引列排序存储
- 稠密索引与稀疏索引：是否包含所有索引项
- 单列索引与多列索引：索引列包含列的数目
- 单级索引与多级索引：索引的层数

索引概述

➢ 索引的目的

- 用于从数据库中高效获取满足条件的记录
- SELECT * FROM library WHERE title = ‘Database System’ ;

文献卡片记录了书的基本信息与所在书架位置



文献卡片在卡片柜中以书名笔画数排序



索引概述

➢ 索引的目的

- 索引通过存储（查找键, 数据位置）来快速定位记录
  - 稠密索引：数据位置→记录的位置（页面，槽位）
  - 稀疏索引：数据位置→页面的位置（页面）
- 索引一般存储以页面为单位组织（内存索引除外）
- 索引也需要支持并发控制
- 插入记录时，先插入数据页面，然后插入索引，而且两者要保持事务一致性

索引按照物理存储类型分类

➢ 按照物理存储分为：聚集索引 vs 非聚集索引

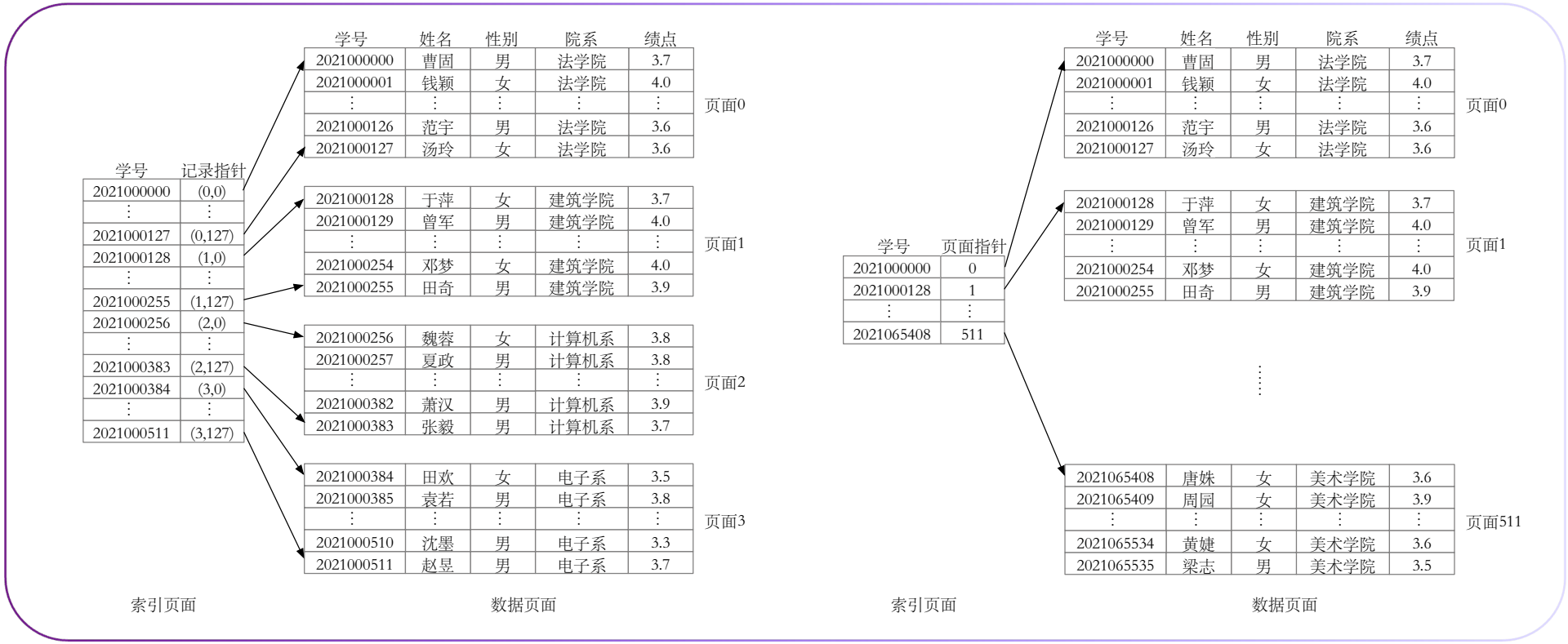
- 聚集索引：数据按照查找键排序
  - 按照查找键连续存储，聚集索引上范围查询效率更高，一张表只能有一个
  - 维护增删时需要移动数据文件中的记录，开销稍大
- 非聚集索引：辅助索引、二级索引
  - **一张表可以有多个**，数据不一定按照索引列顺序存储



索引按照指针记录的粒度分类

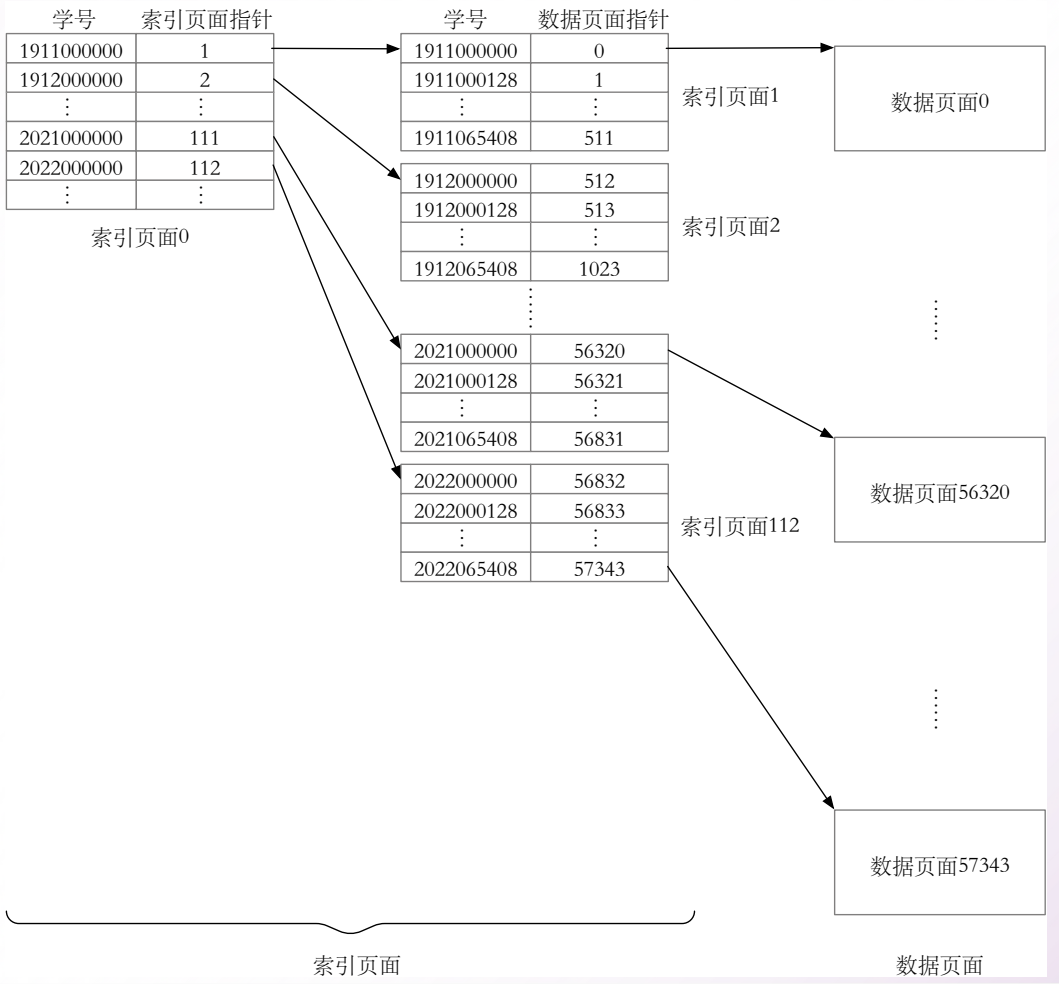
➢ 按照索引粒度分为：稠密索引 vs 稀疏索引

- 稠密索引：索引到数据记录
- 稀疏索引：索引到文件页面



索引按照层数分类

- **单级索引**
  - 一层索引：如哈希
- **多级索引**
  - 索引文件上再建索引
  - 如：B+tree
- **多级索引目的**
  - 多层过滤
  - 进一步降低I/O次数
  - 提升查找速度



索引按照字段特性分类

- **按照字段特性分为：主键索引、唯一索引和普通索引**
- **主键索引**
  - 建立在主键字段上的索引
  - 一张表最多只有一个主键索引
  - 索引列的值不允许有空值
- **唯一索引**
  - 建立在 UNIQUE 字段上的索引
  - 一张表可以有多个唯一索引
  - 索引列的值必须唯一，但是允许有空值



主键索引

索引按照字段个数分类

- **按照字段个数分为：单列索引和多列索引**
- **单列索引**
  - 建立在单个字段上的索引
- **多列索引**
  - 建立在多个字段上的索引



索引组织表

- **用B+树索引组织数据页面，数据页面使用顺序组织方式**
- **叶子节点为数据页面**
- **主键上的B+树聚集索引**
  - 主键上范围查询时更多连续I/O
  - 插入/删除数据时要维护主索引，开销增大
- **索引是否回表：回表指的是通过索引找到页面记录才返回结果**
  - Index only scan不回表：仅查找索引列，根据索引就可以返回结果，如学号是否存在
  - Index scan回表：根据索引列查找其他列的内容，如根据学号查找姓名

索引组织表中，按照主键组织数据，数据记录会经常移动，物理位置改变

数据文件	数据记录指针	代表
索引组织表	主键	MySQL、SQLite
堆表	(页面号, 槽号)	PostgreSQL、DB2

索引类型

➢ 不同类型索引支持不同的数据类型、条件类型

查询类型	索引类型
点查询（score=90）	哈希索引、B+树索引
范围查询（score>60）	B+树索引
多列条件查询(gender=male & rating=5)	位图索引
空间范围查询(115.7°<lag<117.4°& 39.4°<long<41.6°) 最近邻查询(KNN)	多维索引(R树、Quadtree、KD树)

➢ 是否支持持久化、可以否快速恢复

- 内存索引、磁盘索引

目录

1. 索引概述
2. **B+树索引**
3. B+树并发控制
4. 哈希索引
5. SkipList跳表索引
6. LSM树索引
7. 位图索引
8. 多维索引

——R树、KD树、Quadtree、Z曲线

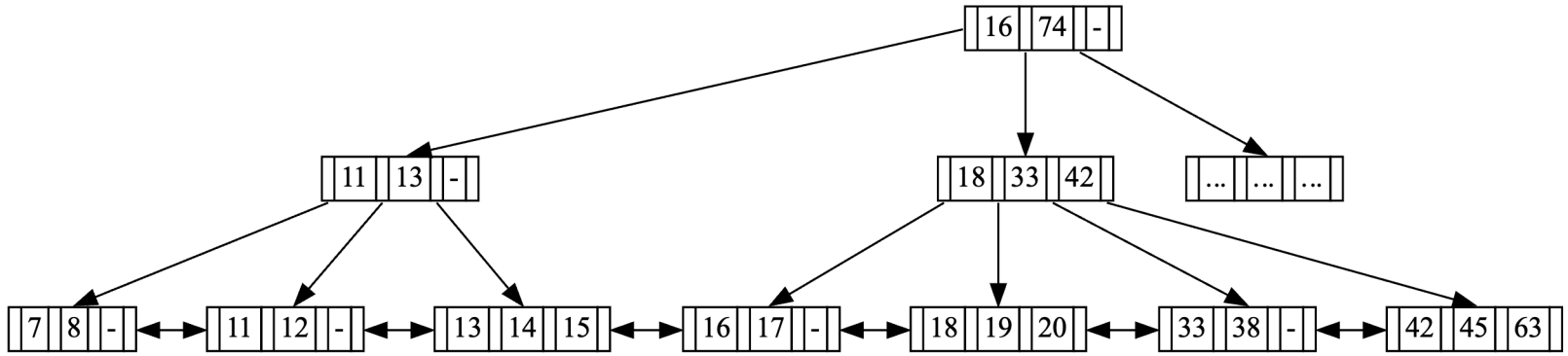
目录

2. B+树索引

- 索引结构
- 查找算法
- 插入算法
- 删除算法

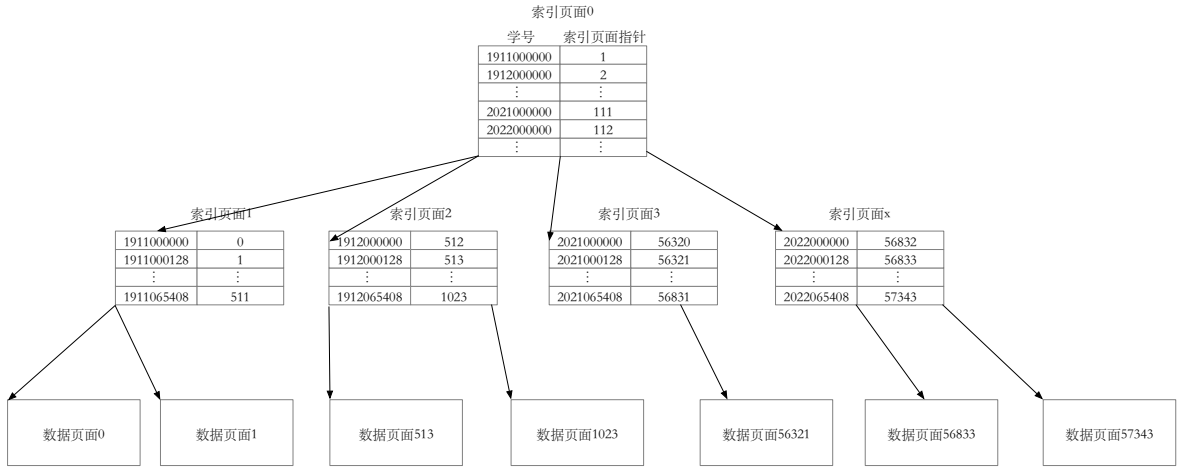
B+树索引结构

- **结构：平衡多叉查找树**
  - 根到所有叶子节点路径长度相同
- **扇出m：每个节点最多分叉数**
  - 子节点数目范围 $\left[\left\lceil\frac{m}{2}\right\rceil, m\right]$
  - 查找键数目范围 $\left[\left\lceil\frac{m}{2}\right\rceil - 1, m-1\right]$



B+树索引结构

- **节点：页面（磁盘索引）**
  - 页面缓冲管理器
- **节点内指针**
  - 内部节点：索引页面号
  - 叶子节点：(数据记录页面号，数据记录槽号)
- $\mathbf{Tree}(P_i) < K_i \leq \mathbf{Tree}(P_{i+1})$
- **扇出m由页面大小和键值大小决定**
  - m通常200～300
  - 10亿条记录仅需要4～5层
  - I/O次数少





目录

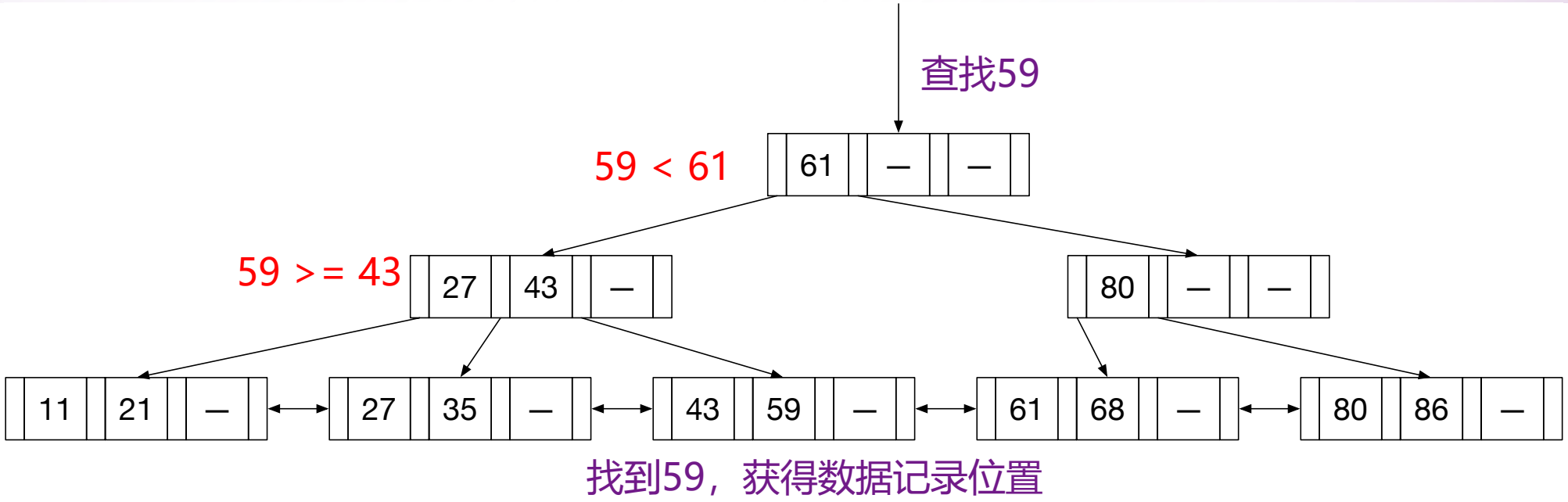
2. B+树索引

- 索引结构
- **查找算法**
- 插入算法
- 删除算法

查找算法

点查询

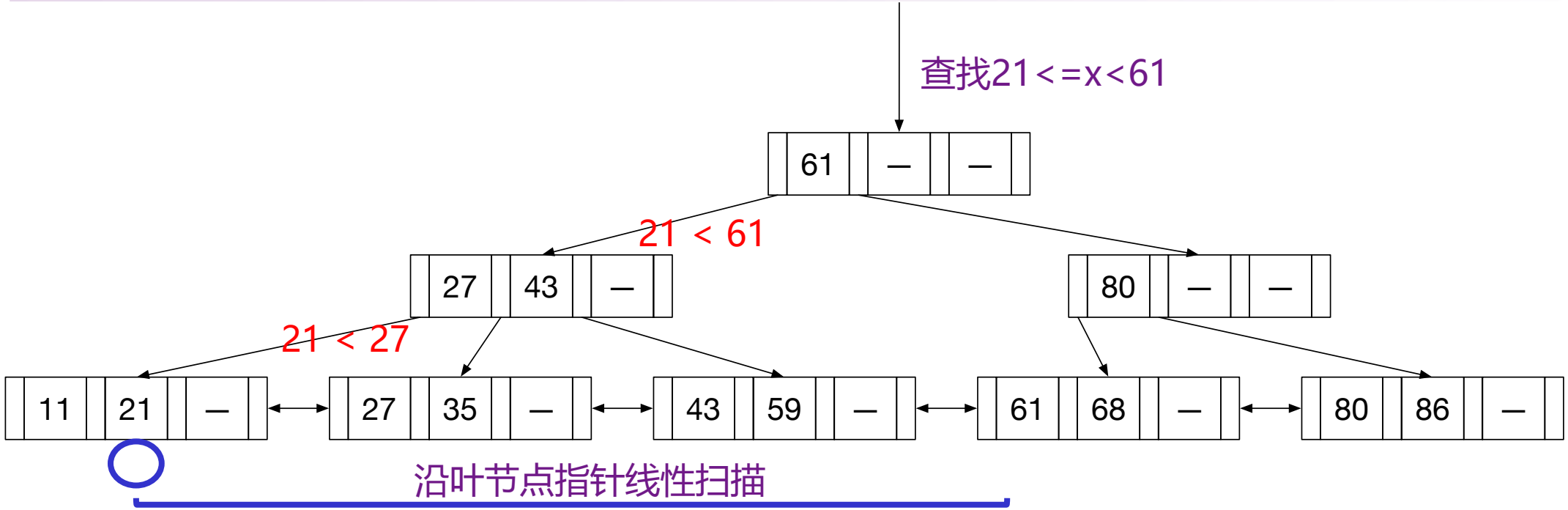
- 从根节点逐层加载索引页面到缓冲区，直到叶子节点
- 注意只判断存在性时，也要访问到叶子节点
- 内部节点存在的键，叶子节点中不一定存在



查找算法

➢ 区间查询（范围查询）

- 叶节点兄弟之间有指针
- 首先根据查找键查找左端点
- 然后按照节点指针向右线性扫描



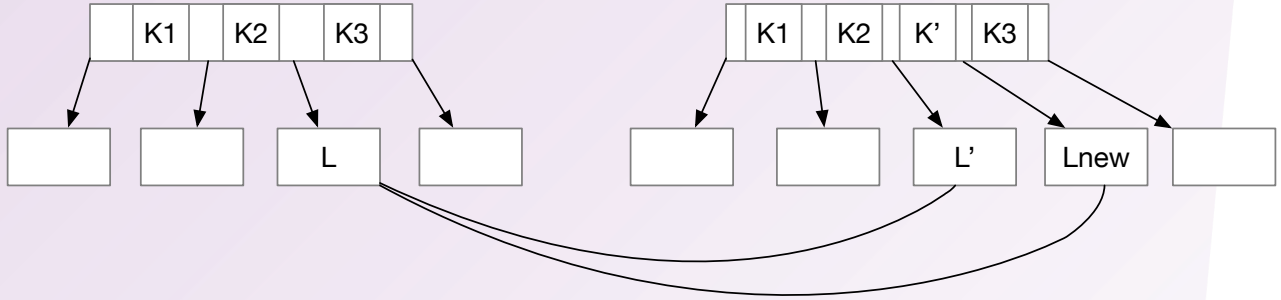
目录

2. B+树索引

- 索引结构
- 查找算法
- **插入算法**
- 删除算法

插入算法

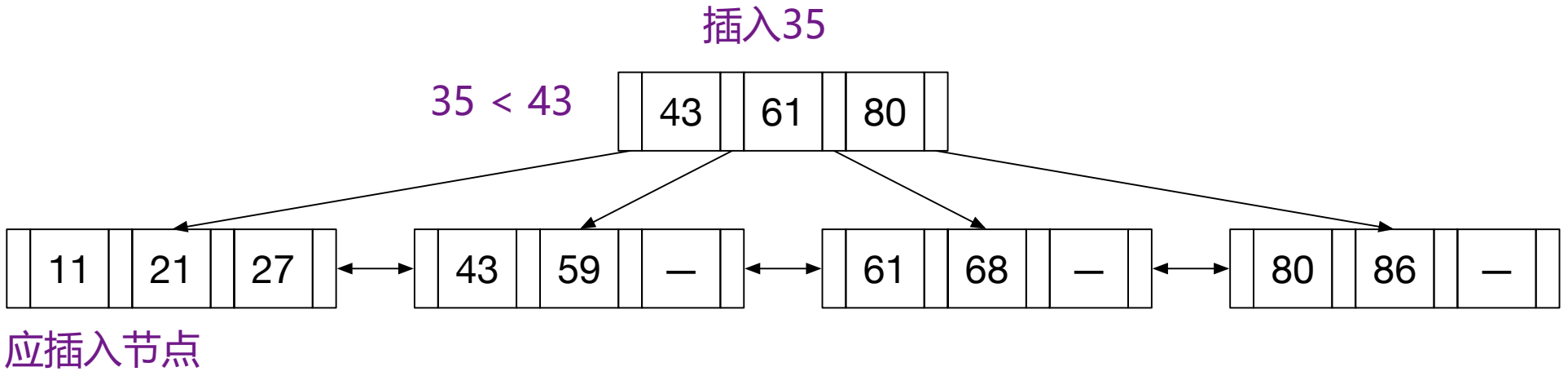
- **找到应插入的叶子节点L**
- **将索引记录插入到L中**
- **如果L有多余空间，结束**
- **否则，需要分裂L**
  - 将节点内容均分为两部分，假设**L中间键为**K′
  - L中小于K′的保存在L'；大于等于K′的移动到L_{new}，
  - 将K′插入L父节点，并更新指向L'和 L_{new} 的指针
  - 如果L父节点溢出，继续分裂
- **内部节点（非叶结点）分裂：**
  - 中间键K′不放在新节点中(L'和L_{new})，仅插入父节点



插入算法

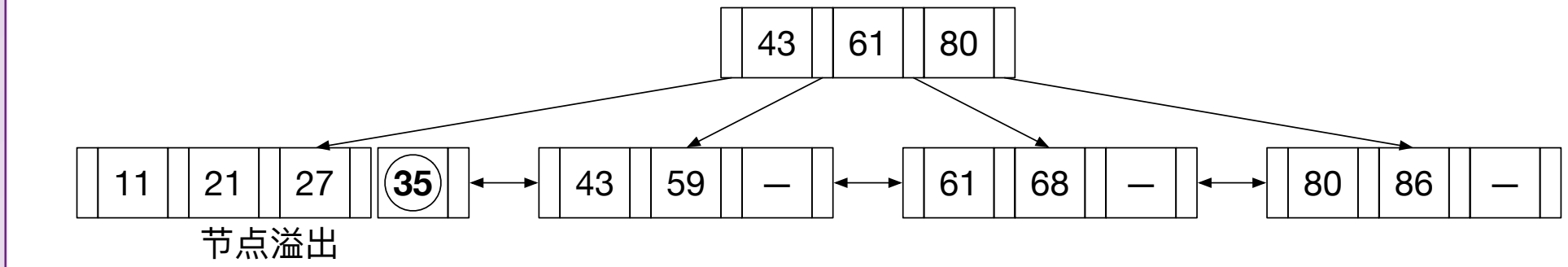
➢ 寻找应插入的节点

- 若叶节点有空余空间，则直接插入



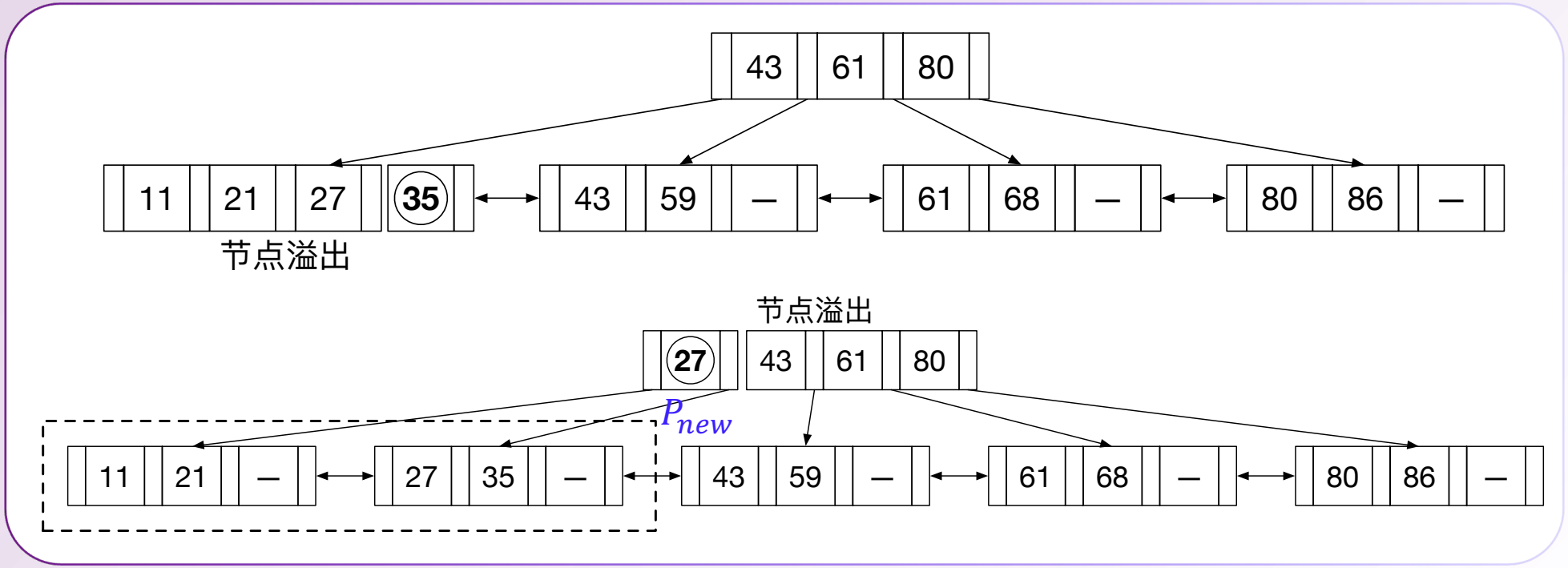
插入算法

- 叶节点溢出，则分裂节点



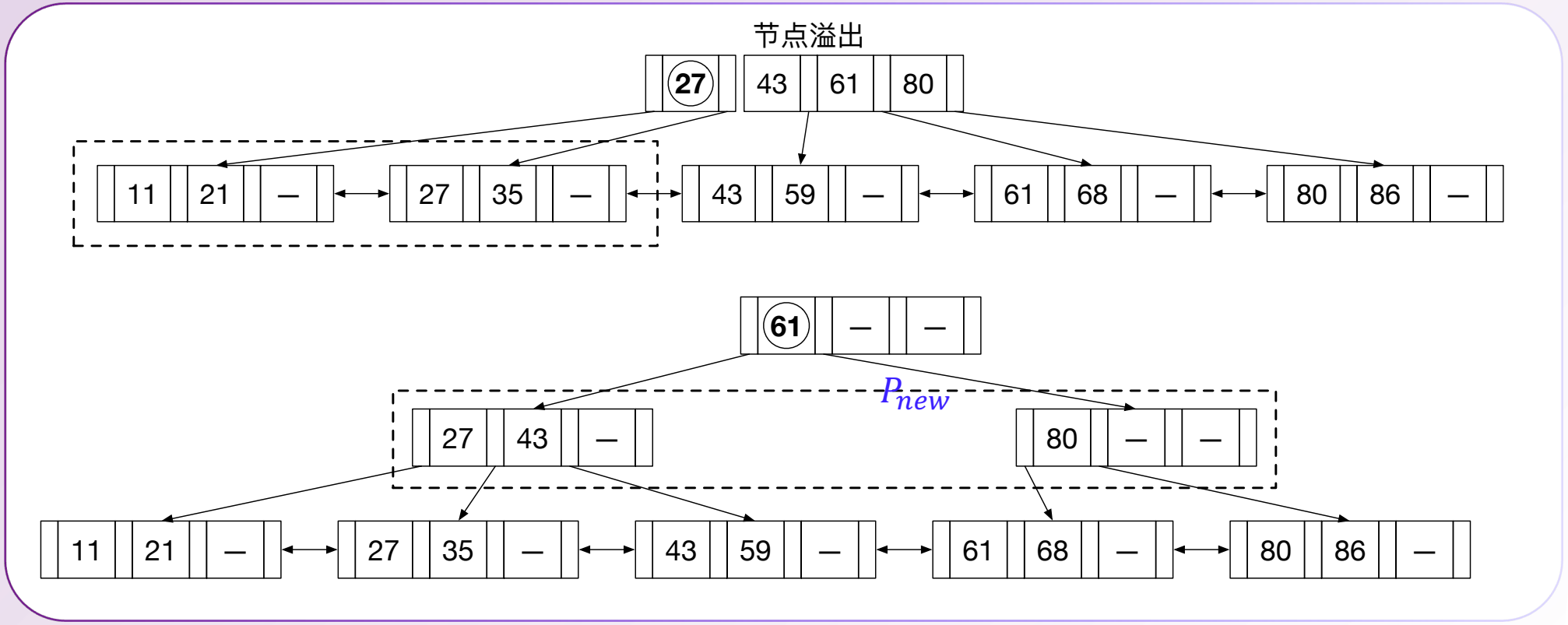
插入算法

- 叶节点从中间分裂
- K′ =27 向上插入父节点
- 若父节点不溢出，则结束



插入算法

- 内部节点溢出，从中间分裂， K′不保留在分裂后的非页节点中
- K′ =61 向上插入父节点（无父节点则分配新根节点）



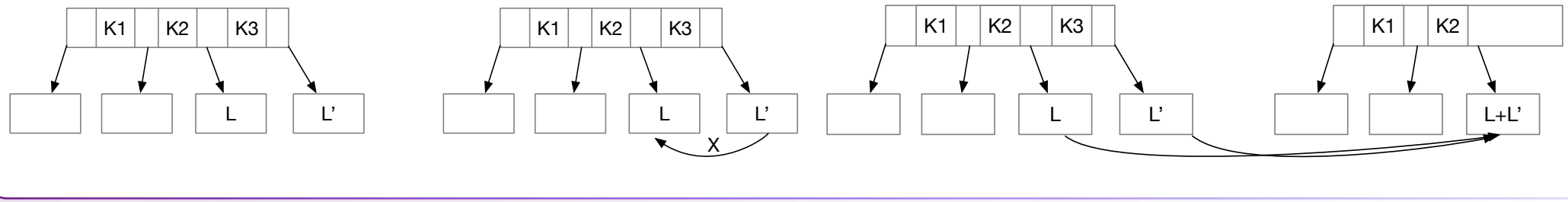
目录

2. B+树索引

- 索引结构
- 查找算法
- 插入算法
- **删除算法**

删除算法

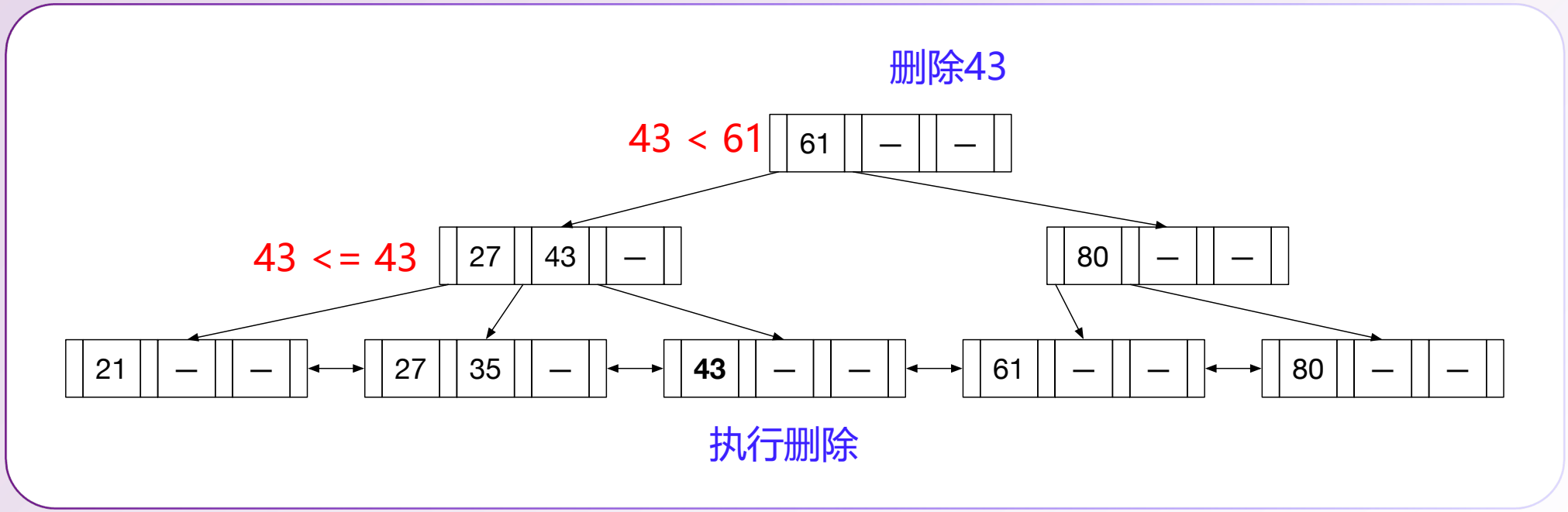
- 找到待删除索引记录所在叶子节点
- 从叶子节点中删除索引记录
- 如果节点中键数目大于等于$\left\lceil \frac{m}{2} \right\rceil - 1$,则结束
- 如果节点中键数目小于等于$\left\lceil \frac{m}{2} \right\rceil - 2$，发生下溢
  - 尝试从兄弟节点借入索引记录,并更新父节点的键值
  - 若兄弟节点无法借入节点，则与兄弟节点合并，并从父节点删除兄弟节点的键值和指针



删除算法

➢ 寻找删除操作发生的叶节点

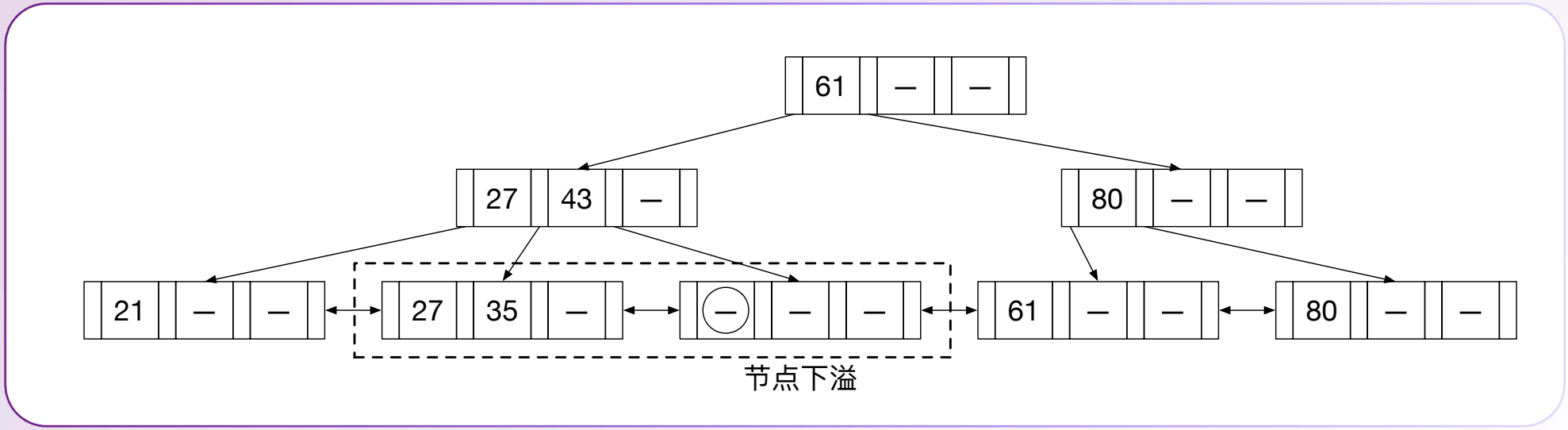
- 如果不存在该查找键，直接结束
- 如果删除操作不会发生下溢，删除索引记录后结束



删除算法

➢ 如果发生下溢

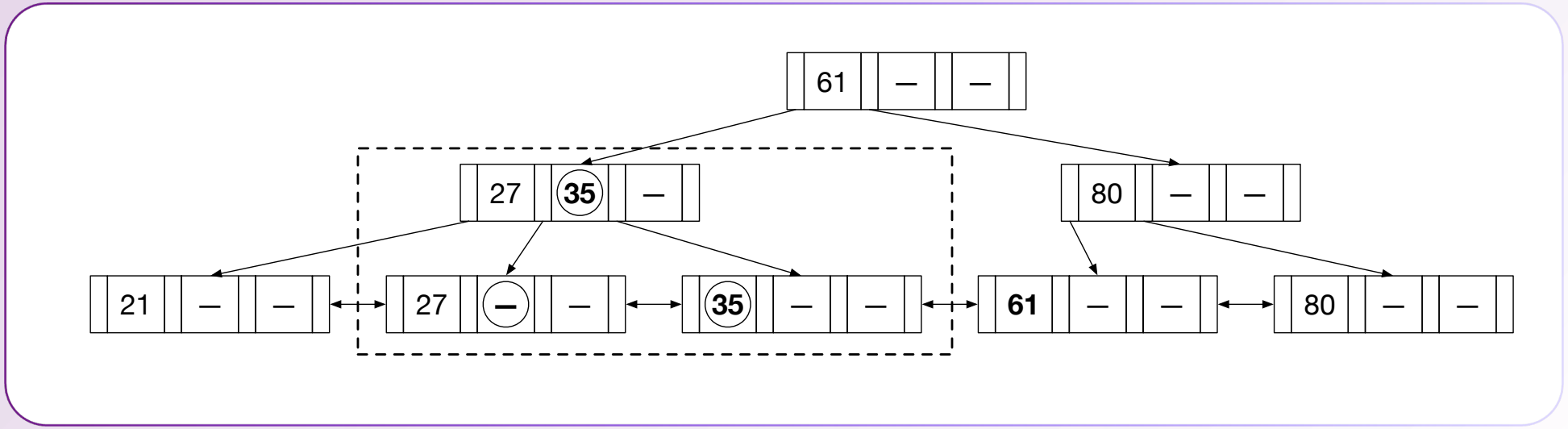
- 尝试从兄弟节点借入索引记录



删除算法

➢ 如果发生下溢

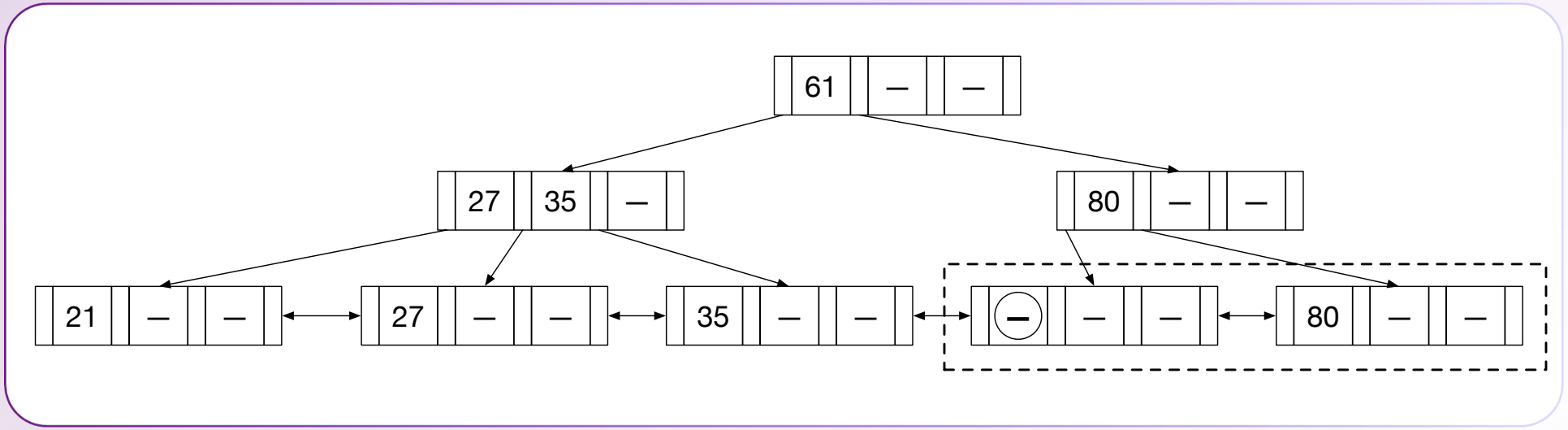
- 尝试从兄弟节点借入索引记录



删除算法

➢ 如果发生下溢

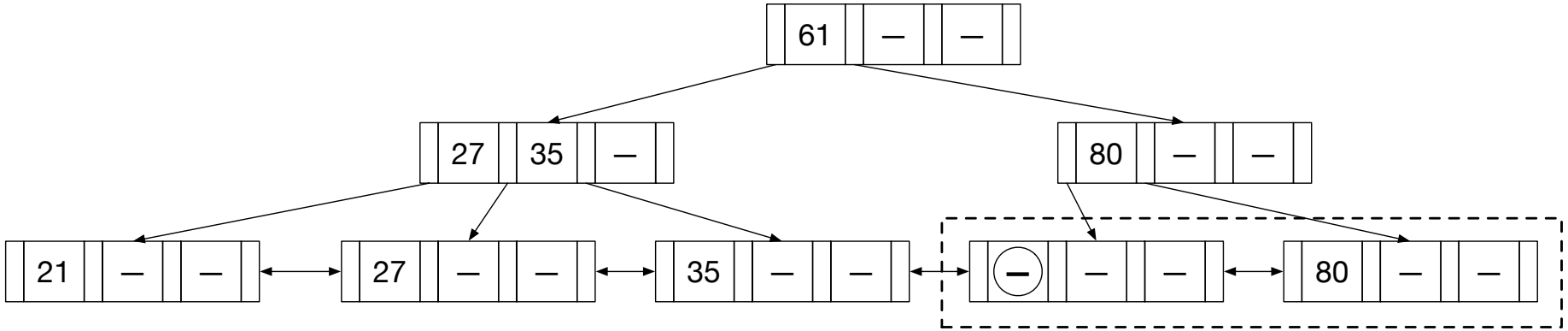
- 尝试从兄弟节点借入索引记录
- 如果兄弟节点无法借入（删除61后），则与兄弟节点合并



删除算法

如果发生下溢

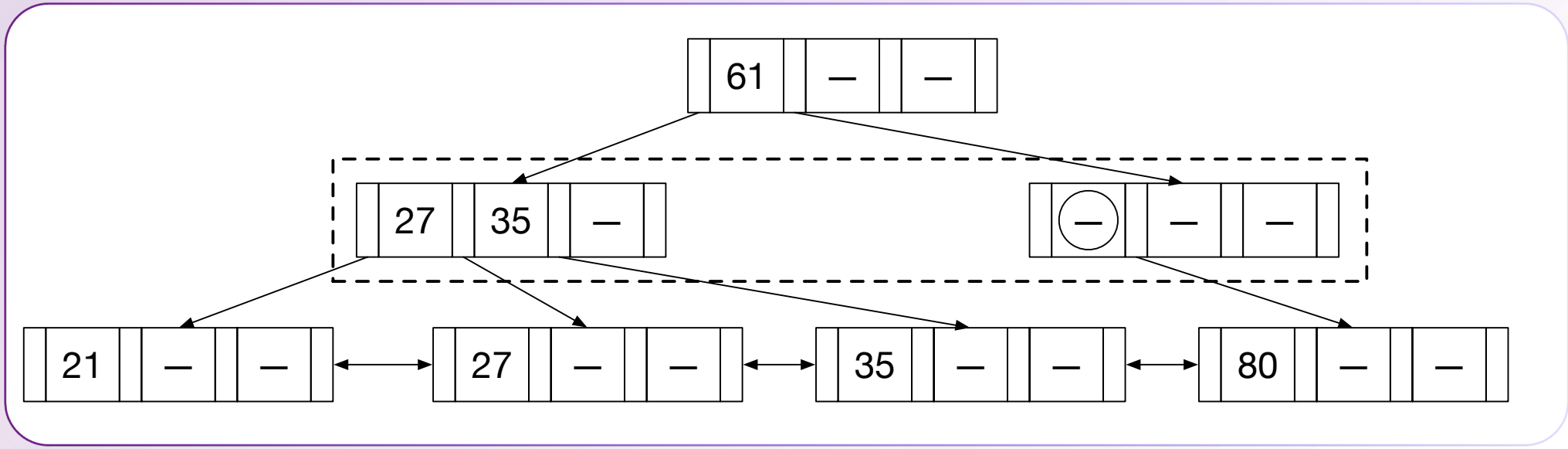
- 尝试从兄弟节点借入索引记录
- 如果兄弟节点无法借入（删除61后），则与兄弟节点合并
  - 从父节点拉取下溢节点与兄弟节点之间的查找键



删除算法

➢ 如果发生下溢

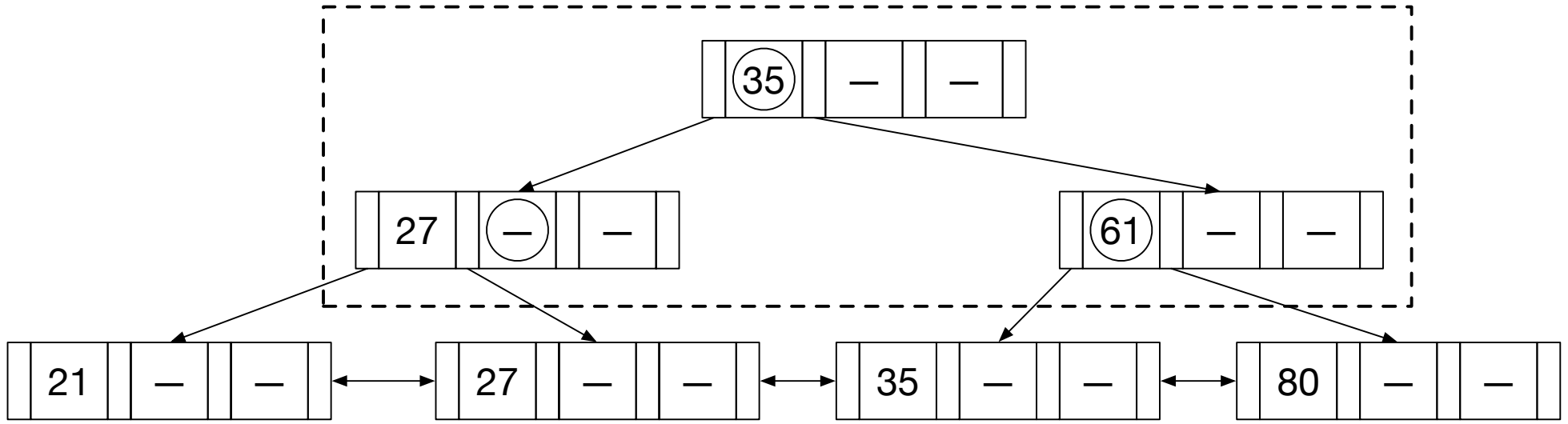
- 尝试从兄弟节点借入索引记录
- 如果兄弟节点无法借入（删除61后），则与兄弟节点合并
  - 从父节点拉取下溢节点与兄弟节点之间的查找键



删除算法

➢ 父节点发生下溢出，同样方法进行解决

- 此处为从兄弟节点借入节点



目录

1. 索引概述
2. B+树索引
3. **B+树并发控制**
4. 哈希索引
5. SkipList跳表索引
6. LSM树索引
7. 位图索引
8. 多维索引

——R树、KD树、Quadtree、Z曲线

目录

3. B+树并发控制

- 闩锁耦合
  - 自上而下：孩子安全后释放祖先锁
  - 乐观自下而上
- B-link树
  - 最多锁三个节点
- Bw树
  - 无锁结构
  - 原位更新→异步合并更新

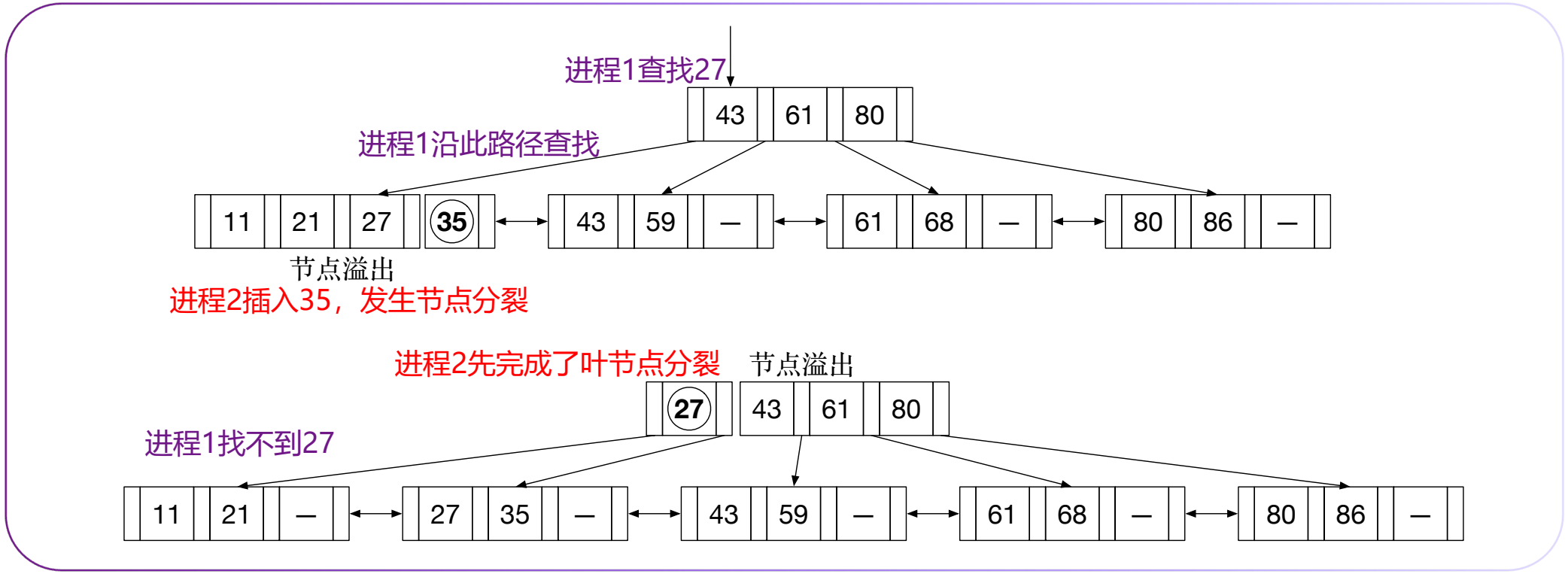
B+树并发操作

➢ 若B+树结构不发生变化

- 查找操作（读操作）可以并行进行

➢ 若B+树结构发生变化

- 读（查找）写（插入）同时进行，则可能发生错误：读写不一致、节点分裂找不到数据



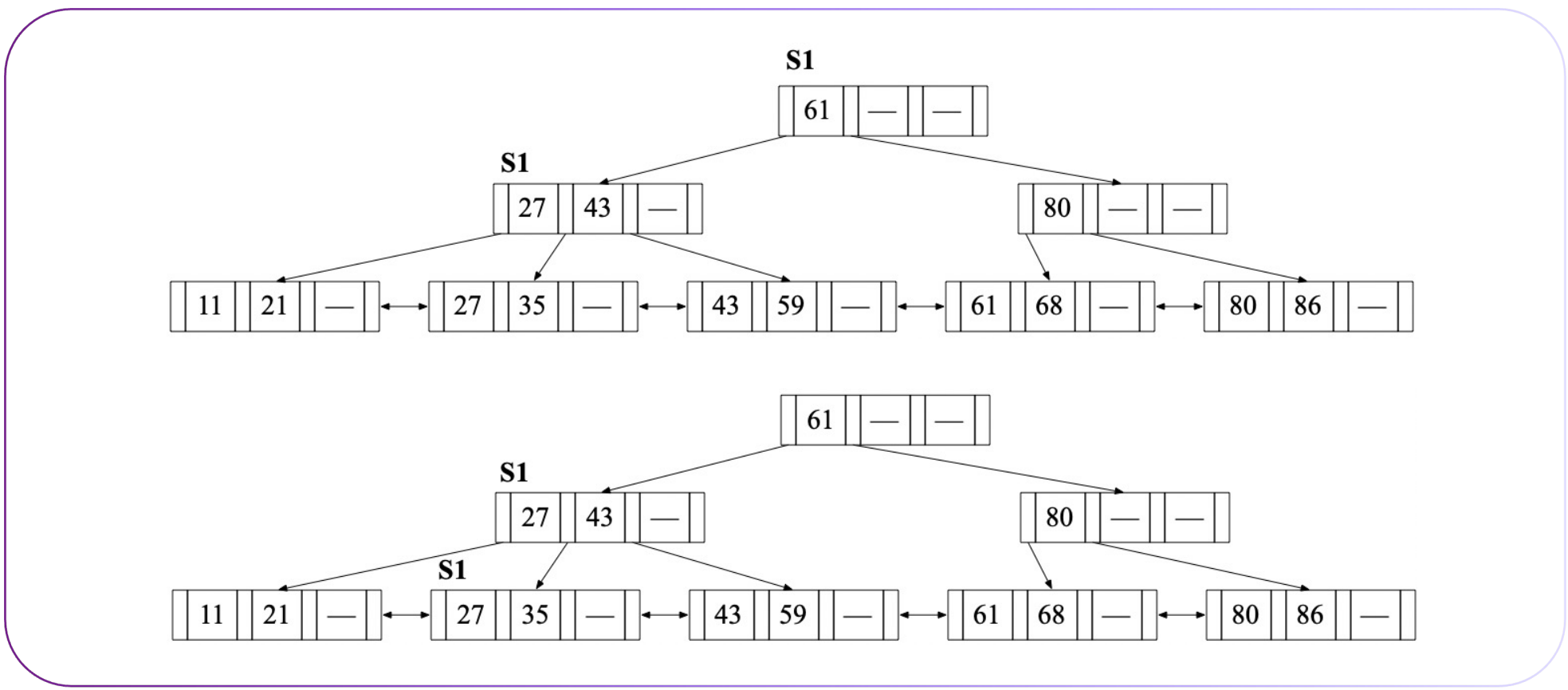
B+树并发处理：闩锁耦合

- **自顶向下加锁（自根向叶子节点）**
- **如果子节点安全，则可以释放其祖先节点的锁**
  - 节点安全：该节点及以上结构不会因此操作发生变化
    - 插入操作时：未满的节点是安全的；溢出不安全
    - 删除操作时：多于半满的节点是安全的；下溢（少于半满）不安全
    - 查找操作时：不会改变结构，一定是安全节点
- **读操作：加共享S(闩)锁；写操作：加互斥X(闩)锁**
  - 读者-写者模型
  - 读者之间可以并发访问；在没有读者时写者才能访问；写者之间互斥

闩锁耦合

➢ 查找操作加锁过程

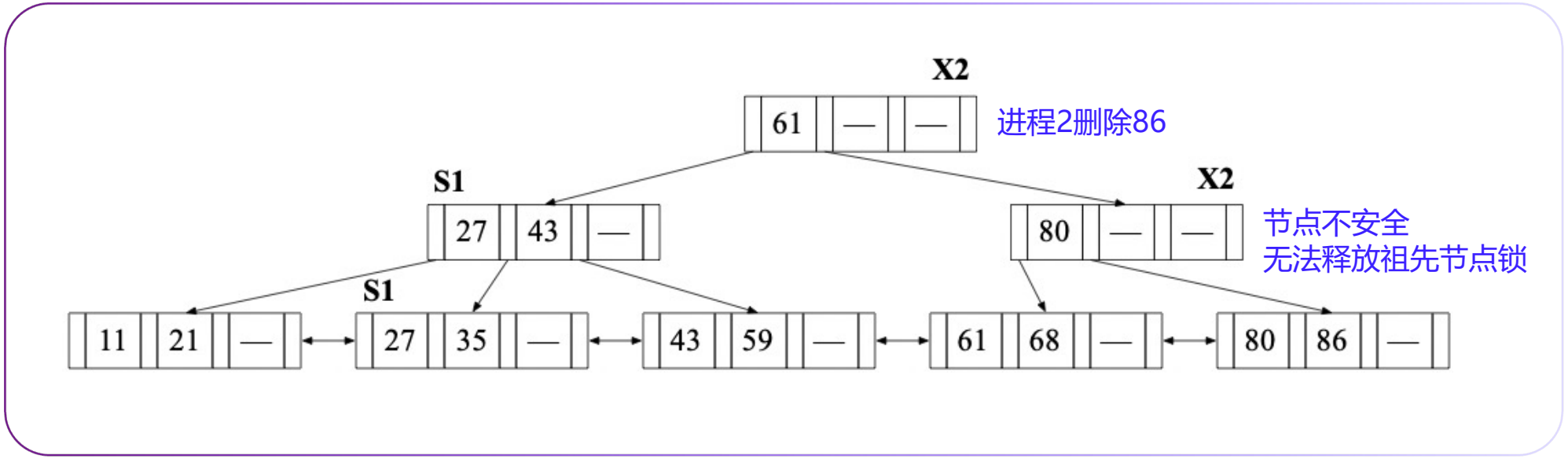
- 进程1查找27，沿路径释放父节点锁，对子节点加锁



闩锁耦合

➢ 修改（插入/删除）操作加锁过程

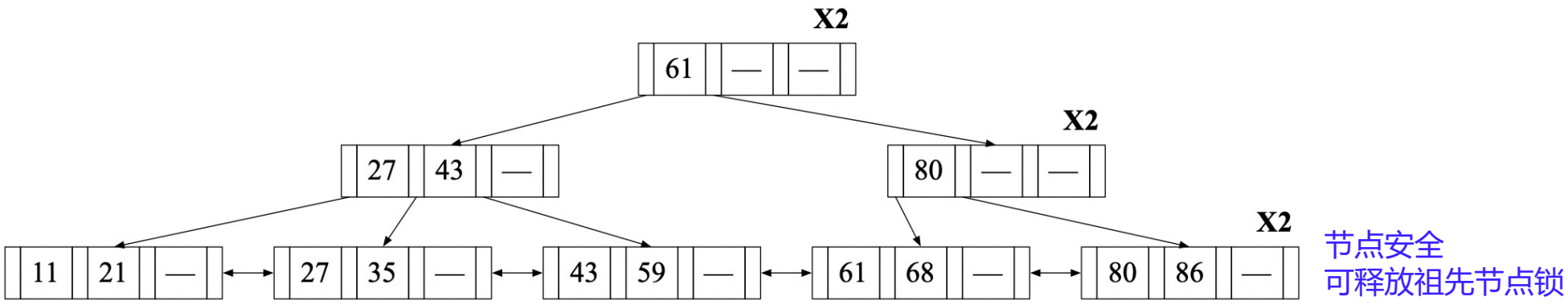
- 进程2删除86
- 对根节点加X锁，对子节点加X锁
- 子节点不安全，继续向下加锁



闩锁耦合

➢ 修改（插入/删除）操作加锁过程

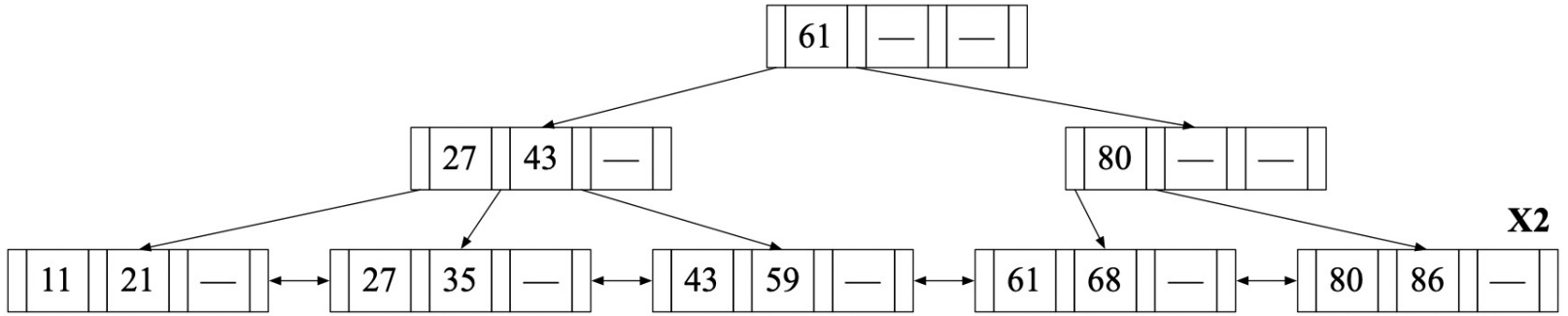
- 进程2删除86
- 对根节点加X锁，对子节点加X锁
- 子节点不安全，继续向下加锁
- 叶子节点安全，可以释放祖先节点的锁



闩锁耦合

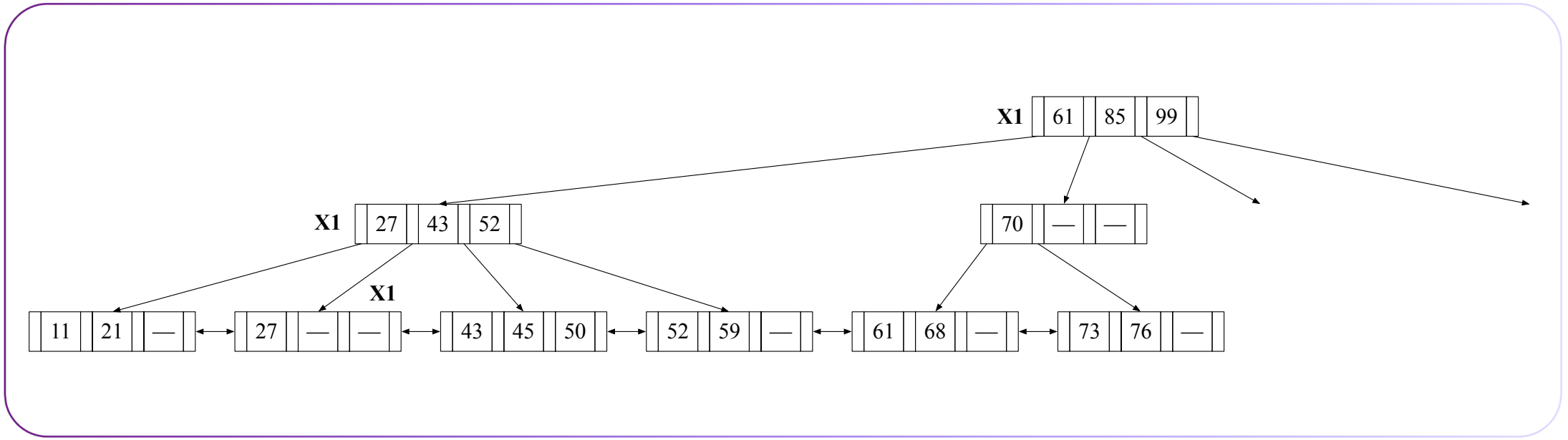
➢ 修改（插入/删除）操作加锁过程

- 进程2删除86
- 对根节点加X锁，对子节点加X锁
- 子节点不安全，继续向下加锁
- 叶子节点安全，可以释放祖先节点的锁



闩锁耦合存在的问题

- **闩锁耦合自上向下加锁，锁竞争严重，特别上层节点访问频繁**
  - 进程1插入28，到达叶节点前都要对树根加锁
  - 阻塞整棵树读/写

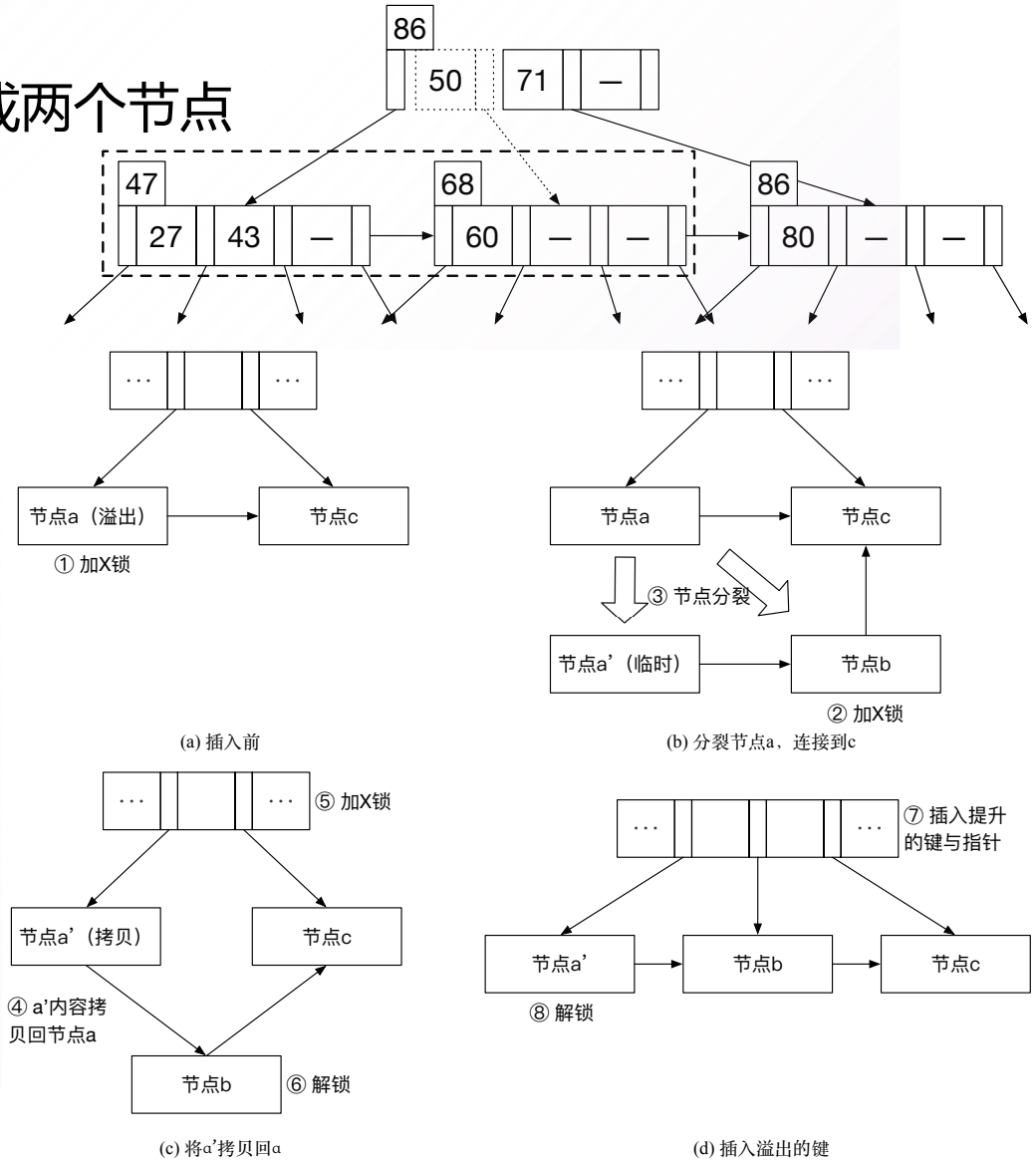


自下而上的乐观锁

- **读操作：与闩锁耦合一致（自上而下加读锁）**
- **写操作：插入或者删除操作：自上而下加读锁，找到叶节点**
  - 插入：尝试对叶子结点升级锁（加写锁）
    - 如果叶子节点不满，直接插入
    - 如果叶子节点满（需要分裂），尝试对父节点加写锁（是否存在冲突）
      - 如果可加写锁，将分裂的key插入该节点；
      - 如不可加写锁：使用闩锁耦合方案从上而下重试。
  - 删除：尝试对叶子结点升级锁（加写锁）
    - 如果叶子节点大于m/2孩子，直接删除
    - 如果叶子节点下溢（需要合并），尝试对父节点加写锁（是否存在冲突）
      - 如果可加写锁，从该父节点删除key；
      - 如不可加写锁：使用闩锁耦合方案从上而下重试。
- **适合读密集型业务，不合适写密集型业务**

B-link 树设计理念

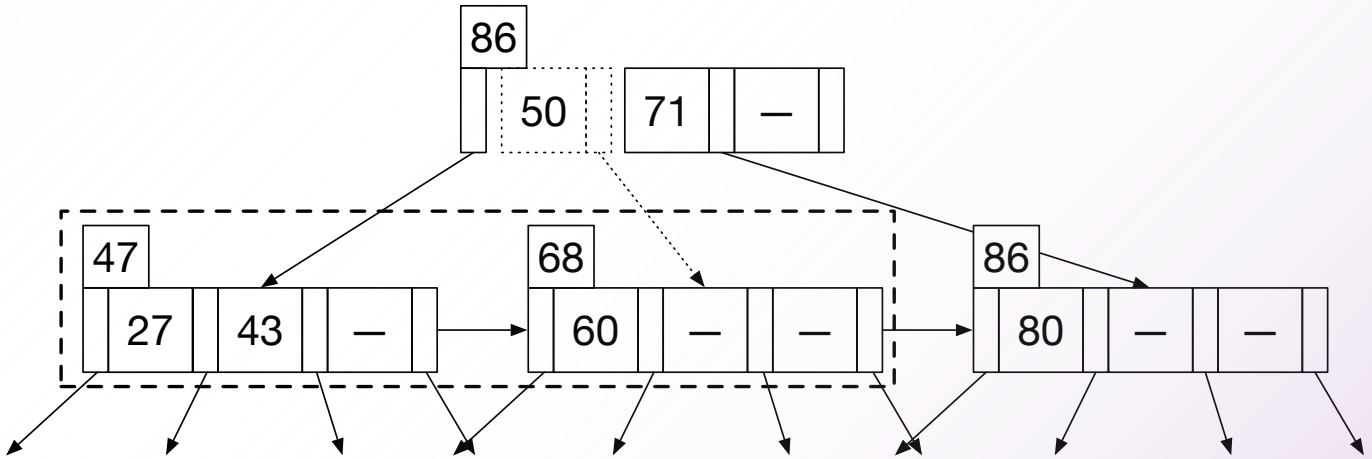
- 节点a溢出[min,max)：锁定该节点
- 将a节点的数据（包括新插入的数据），根据中间键k拆分成两个节点
  - a’ ：[min,k)；b：[k,max)
- 将a’复制给a；
  - 如果此时查找[min,k) 之间的值？ok
  - 如果此时查找[k，max) 之间的值？
    a. 加一个从a’到b的link，通过指针到b找该查找键
  - 如何判断查找键不存在？还是存在于右兄弟节点？
    a. 每个节点加一个high-key，即本节点的最大键值
    b. 如果查找键大于high-key，则到右兄弟节点查找
    c. 如果查找键小于等于high-key，则在本节点查找



B-link 树设计理念

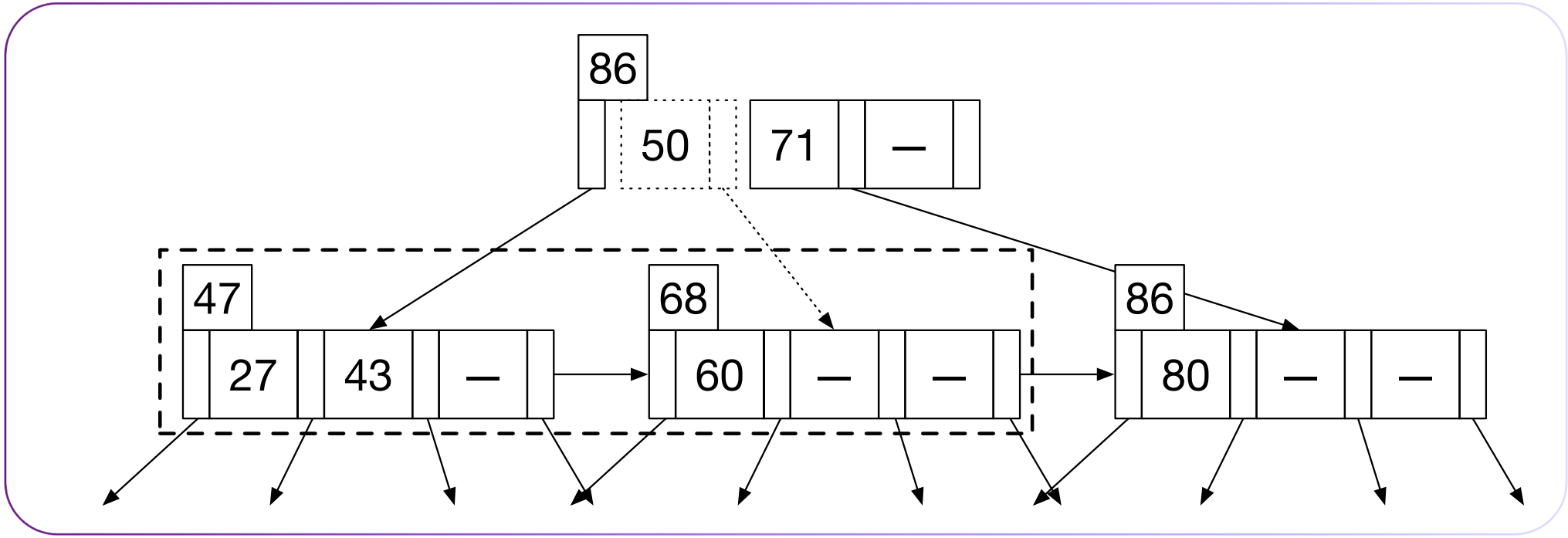
B-link树思想

- 将k移动到a’ 和b的父亲节点
- k的左指针指向a’
- k的右指针指向b
- 插入结束，解锁
- 最多锁三个节点
  - a’
  - b
  - 父亲节点



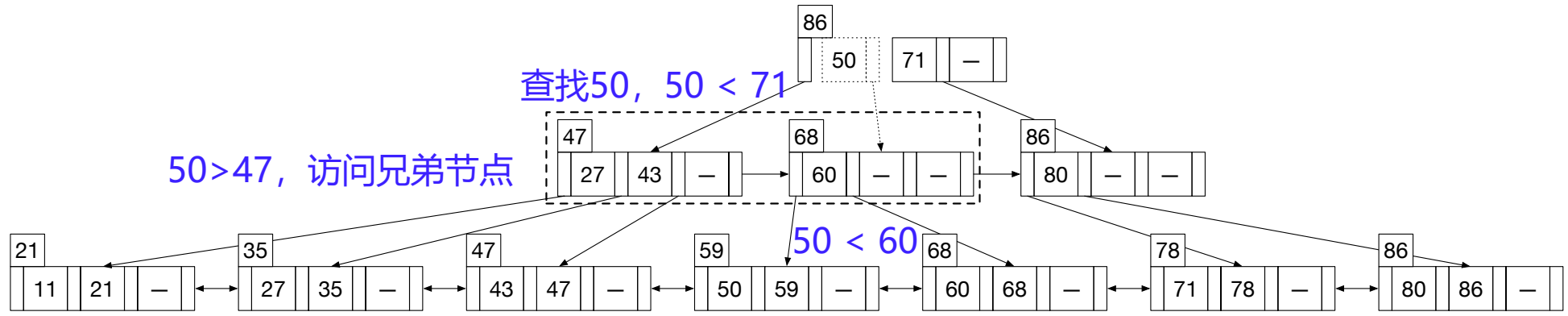
B-link树结构

- 内部节点也有兄弟指针
- 节点分裂的中间状态也为合法状态
- 每个节点多存储一个high key，代表子树中的键的上界



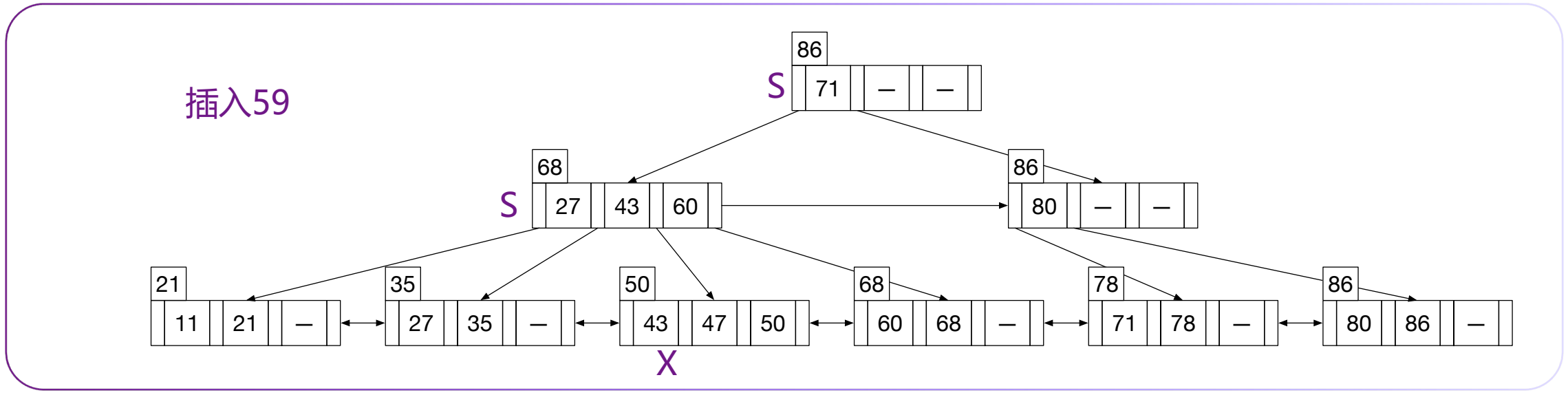
B-link树查找操作

- 下一个访问的节点是子节点或右兄弟节点
- 如果key > high key，则访问右兄弟节点
- 对下一个节点加锁前，释放当前节点的锁

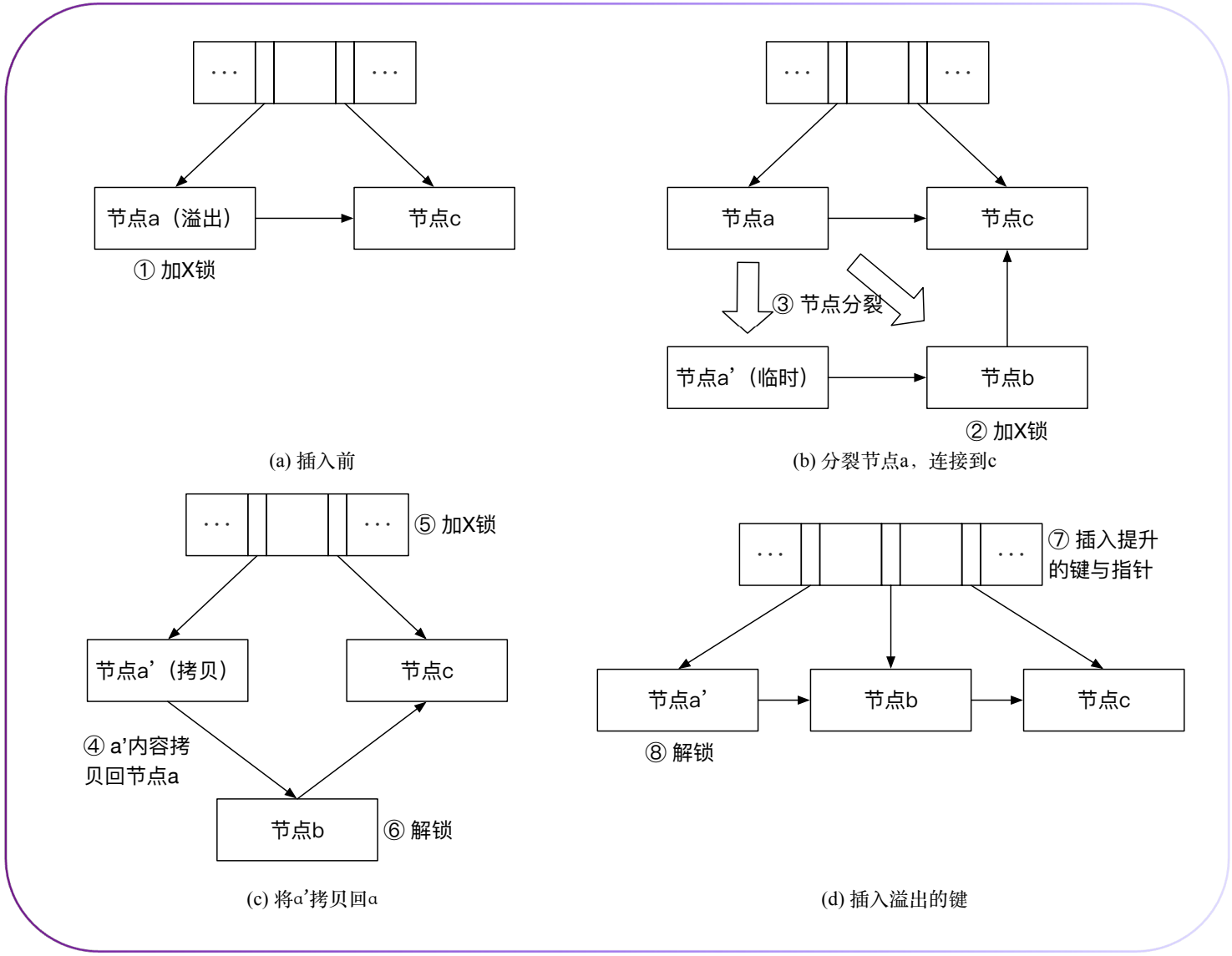


B-link树插入操作：查找待插入节点

- 通过查找算法找到待插入的叶子节点
- 对内部节点加S锁
- 对叶子节点加X锁



B-link树插入操作：溢出分裂节点

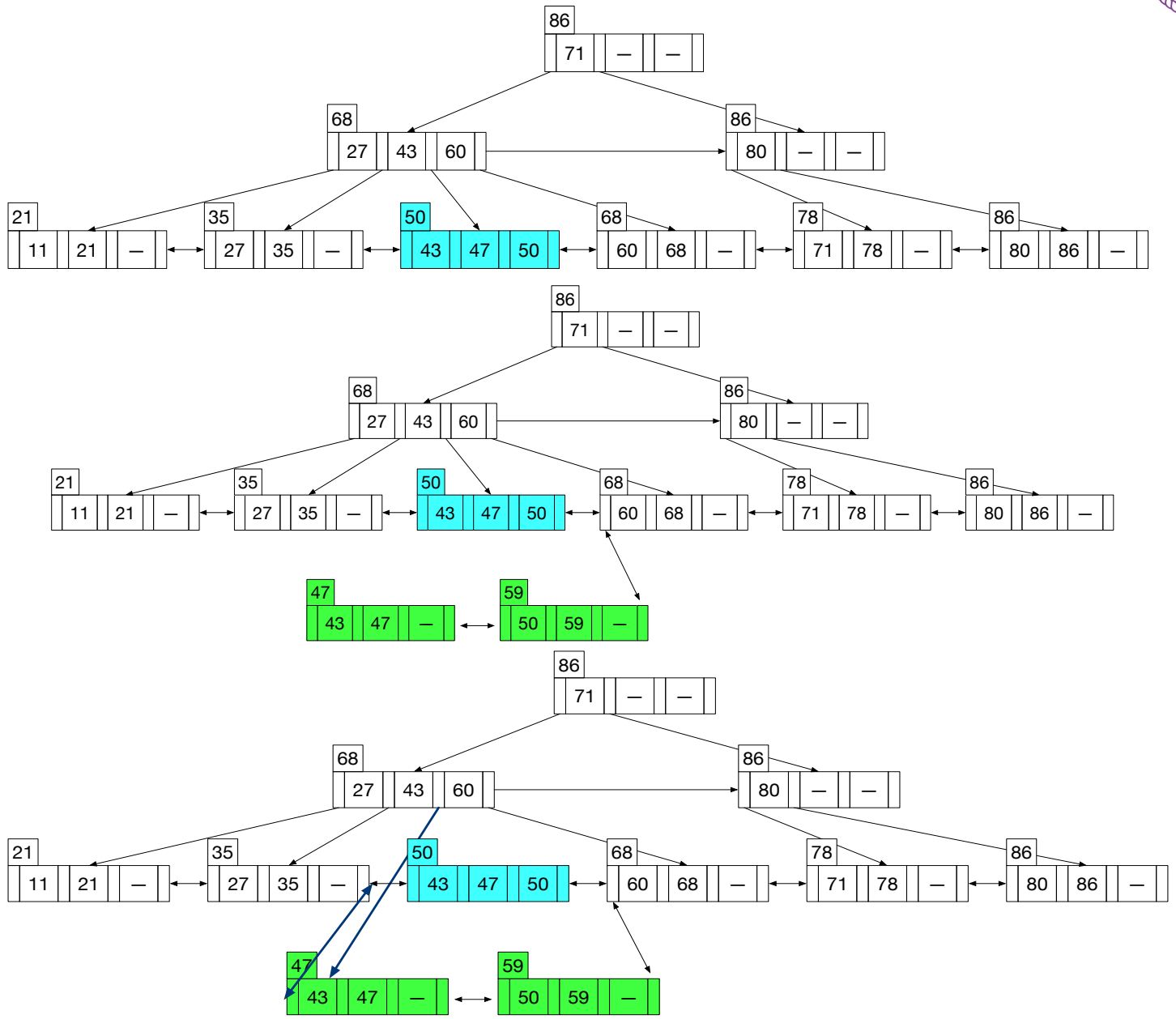


B-link树插入操作：案例

(1) 原始树

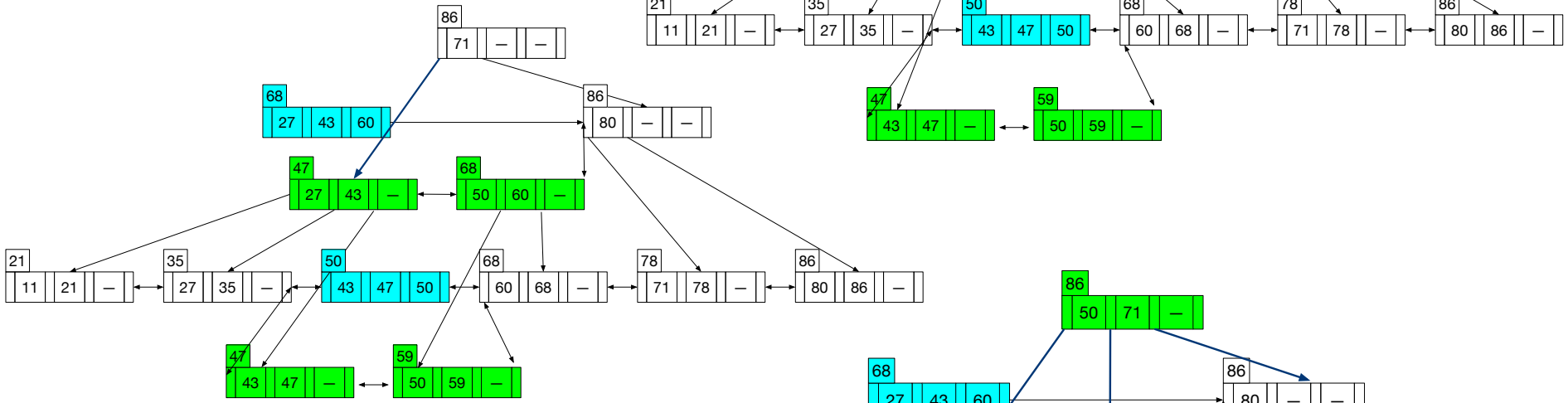
(2) 插入键值59：分裂2个节点

(3)更新父节点指针

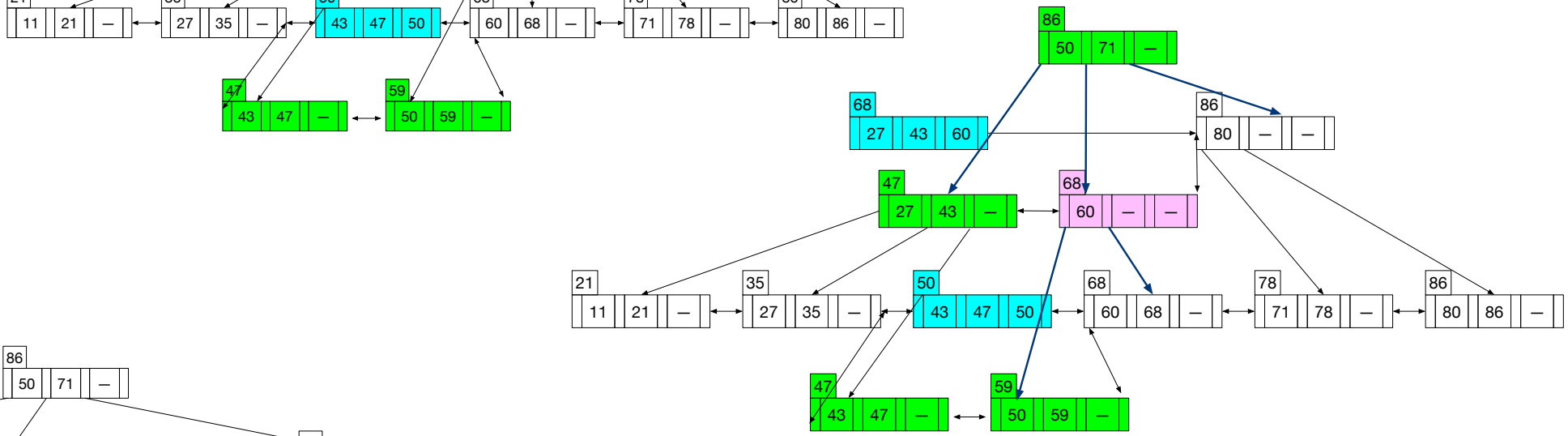


B-link树插入操作：案例

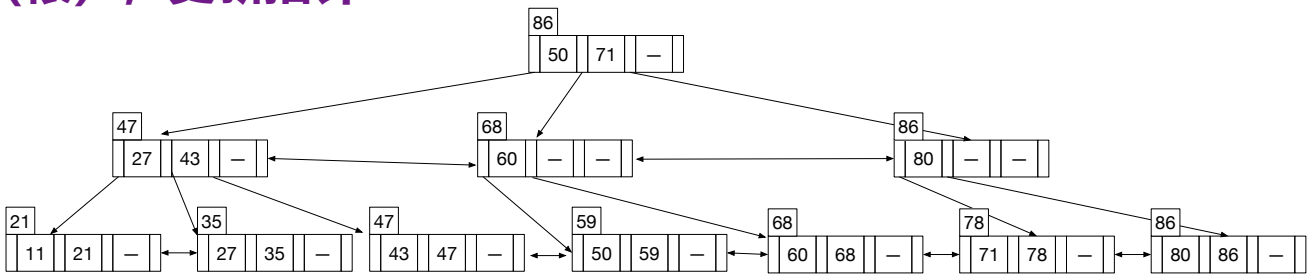
(4) 复制键值50到父节点，父节点分裂



(5) 更新父节点指针

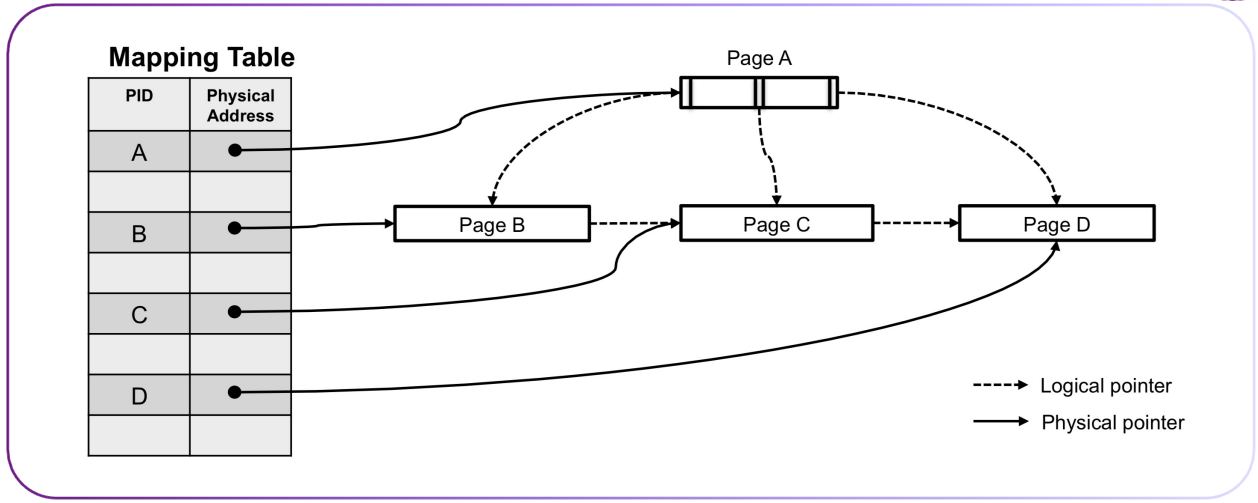


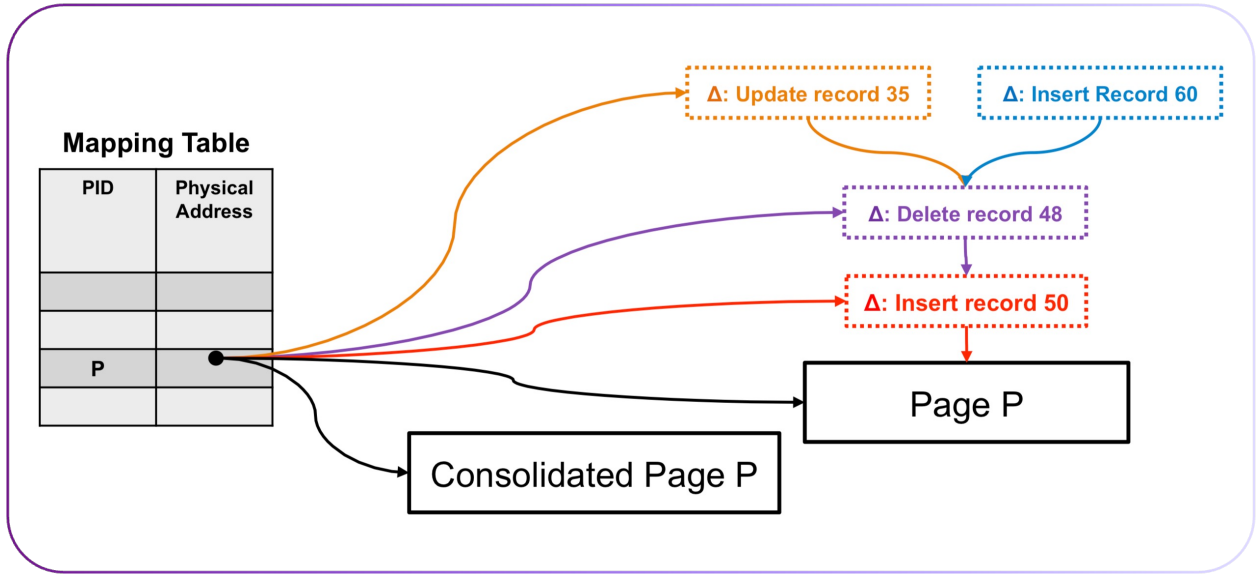
(6)移动键值50到父节点（根），更新指针



Bw树

- **B-link树仍然存在锁竞争**
- **Bw树为无锁结构，避免锁竞争**
- **页面不可修改**
  - 页面原位更新→增量更新页面(适合SSD/Flash)
  - 避免多核下进行原地更新造成的缓存失效
  - 通过LSM-tree思想合并更新
- **使用CAS原子操作使更新生效**
  - 原子操作避免出现中间不合法状态

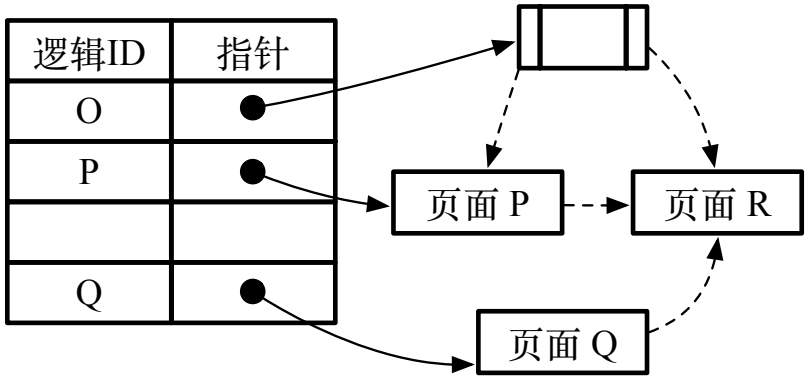


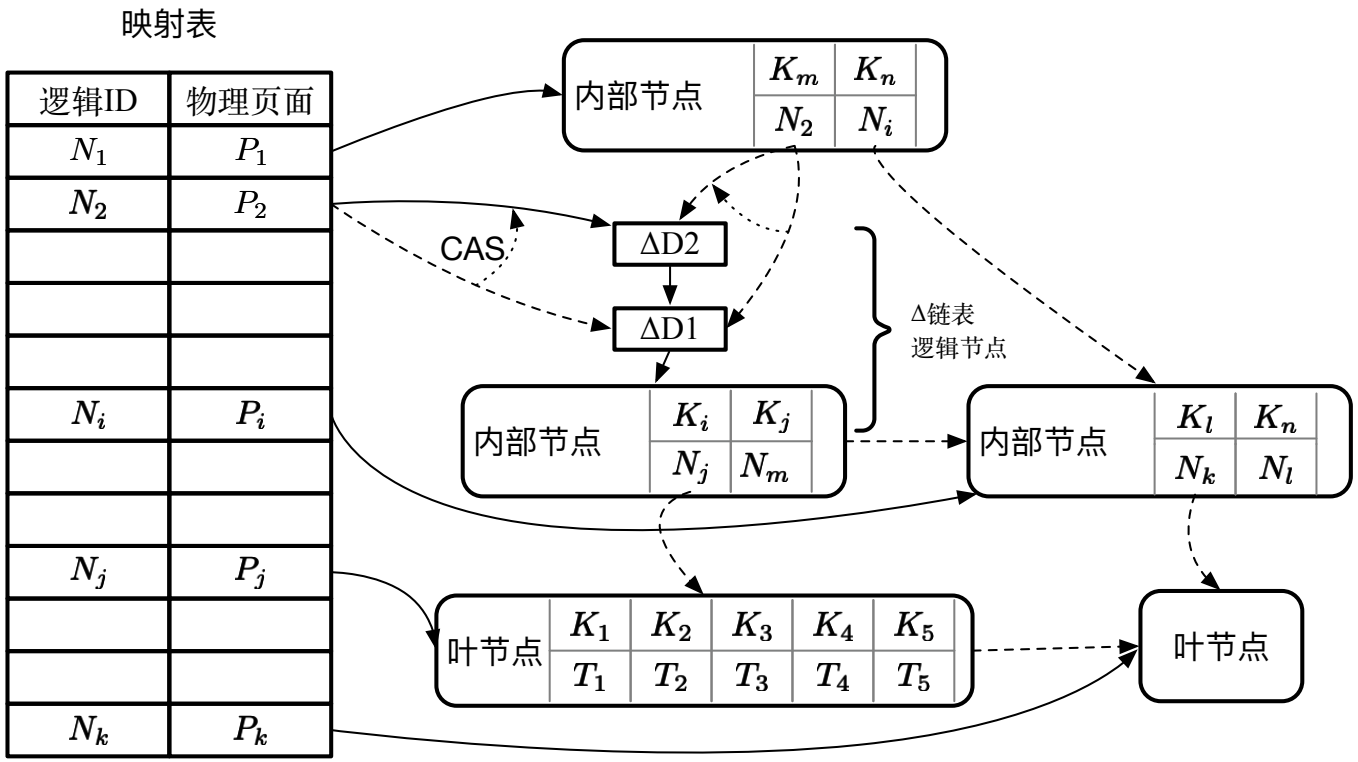


Bw树

➢映射表：将页面逻辑ID映射到内存或硬盘的物理页面地址

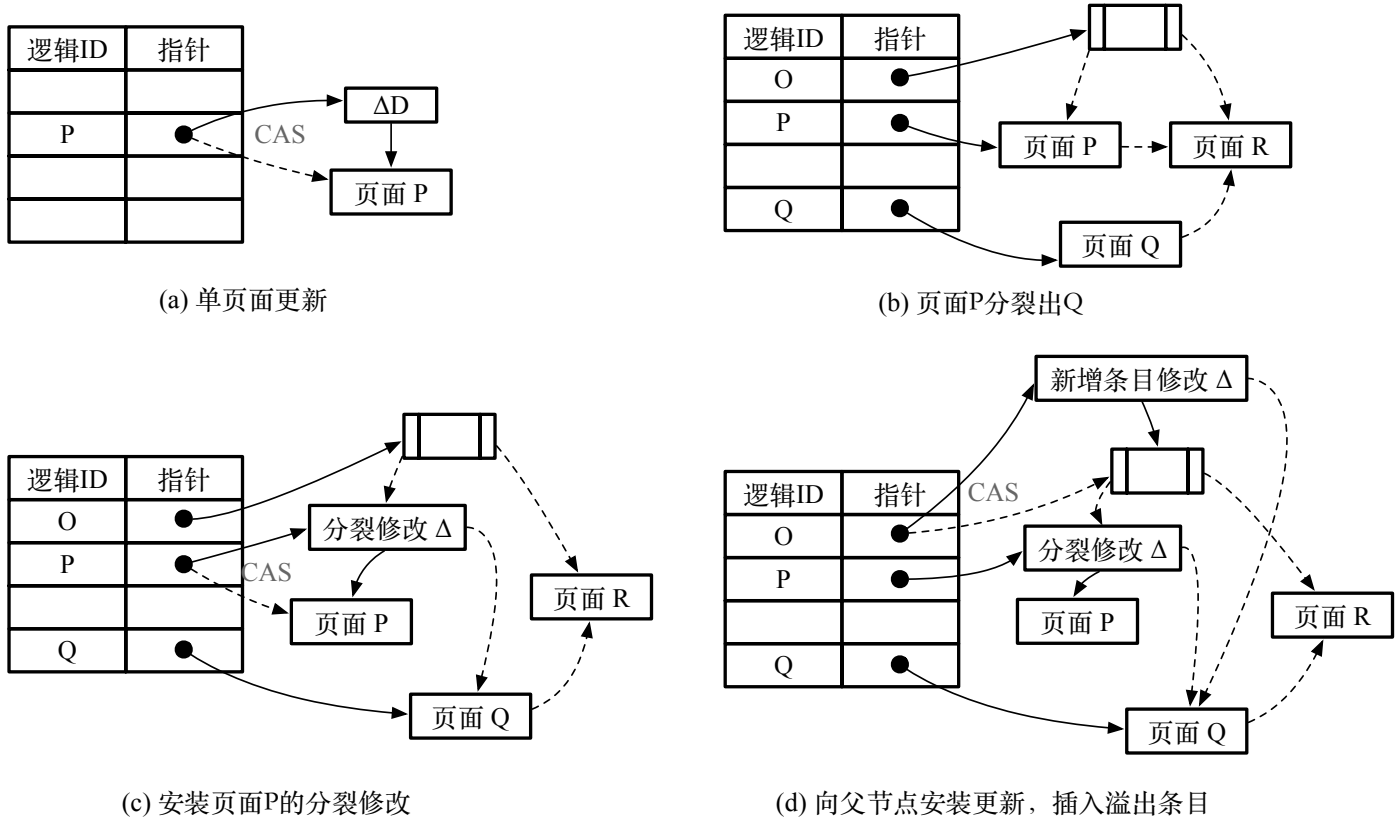
- 原页面P的物理地址
- 页面逻辑ID当前映射的物理地址
- 以及增量记录ΔD的物理地址
- CAS指令或者保留逻辑页面ID当前指向的物理地址，或者更新为ΔD。





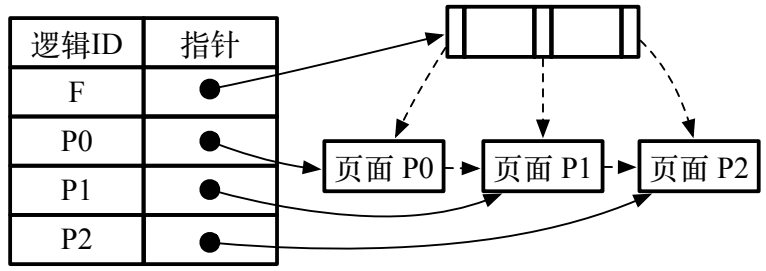
Bw树

- **页面映射表将逻辑页面ID映射到链表第一个节点**
- **单页面更新**
  - 链接更新页面
- **页面分裂**
  - 借鉴B-link树
  - 对插入节点与父节点页面更新

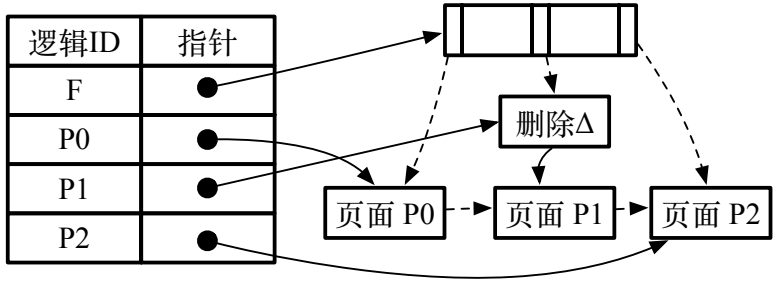


Bw树

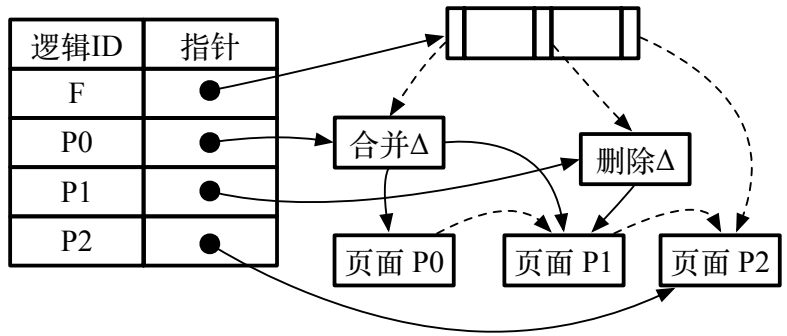
➢ 删除导致节点合并



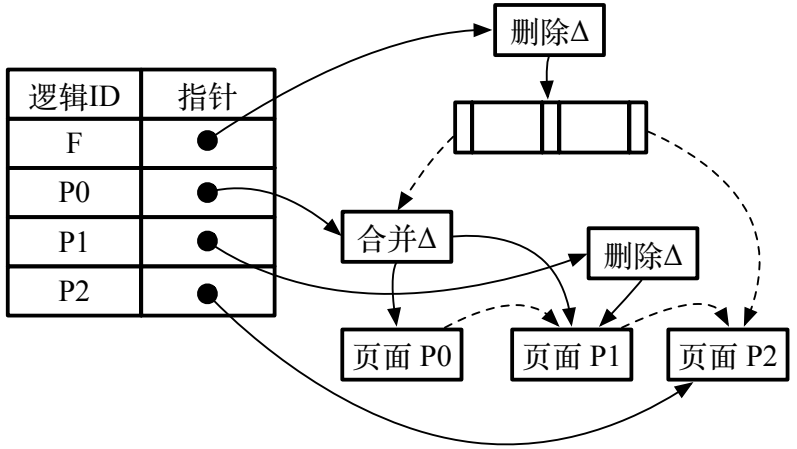
(e) 删除前



(f) 标记删除P1



(g) P0更新为合并节点



(h) 父节点中删除P1

Bw树

- 空间整理与回收
- 合适的时机将ΔD的链与索引节点合并。
- 合并时(Consolidation)会创建一个新的节点，并通过CAS将新节点添加到Bw树上。
- Bw树采用懒惰整理的策略，当查找时发现节点上记录链长度超过设定阈值时，会将记录转移到一整块新申请的物理页上，通过CAS原子操作对页面映射表进行更新。
- 原页面通过垃圾回收机制完成释放，确保在没有其他线程引用后才进行回收。

目录

1. 索引概述
2. B+树索引
3. B+树并发控制
4. **哈希索引**
5. SkipList跳表索引
6. LSM树索引
7. 位图索引
8. 多维索引

——R树、KD树、Quadtree、Z曲线

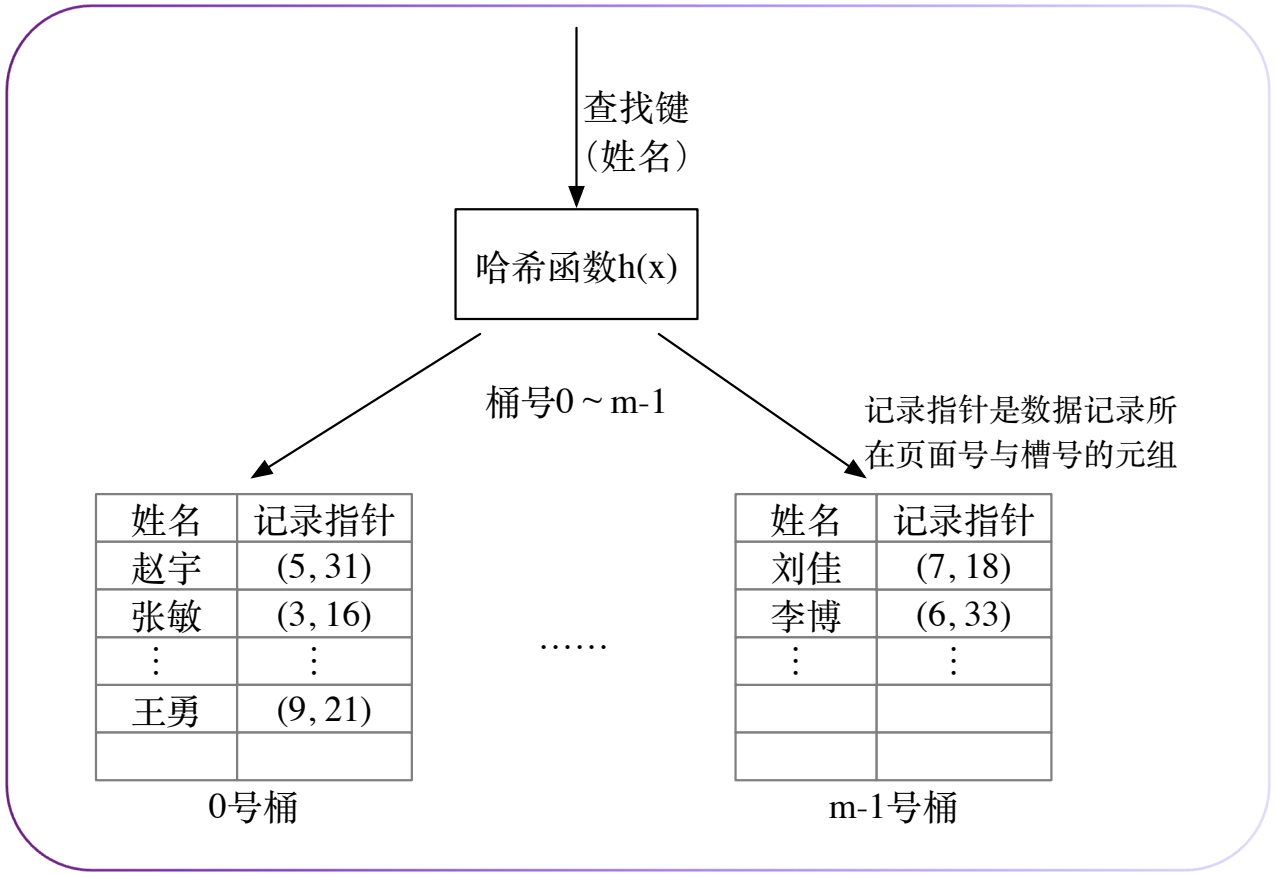
目录

4. 哈希索引

- **哈希函数**
- **静态哈希：哈希桶数不变**
  - 闭哈希：寻找空位置
    - 线性探测新位置（容易堆积）
    - 平方探测新位置
  - 开哈希：哈希桶链表
- **动态哈希：哈希桶数动态变化**
  - 可扩展哈希表
  - 线性哈希表

哈希函数

- 给定查找键，返回一个整数表示（哈希值）
- 哈希值用于确定查找键在哈希表中的位置



哈希函数

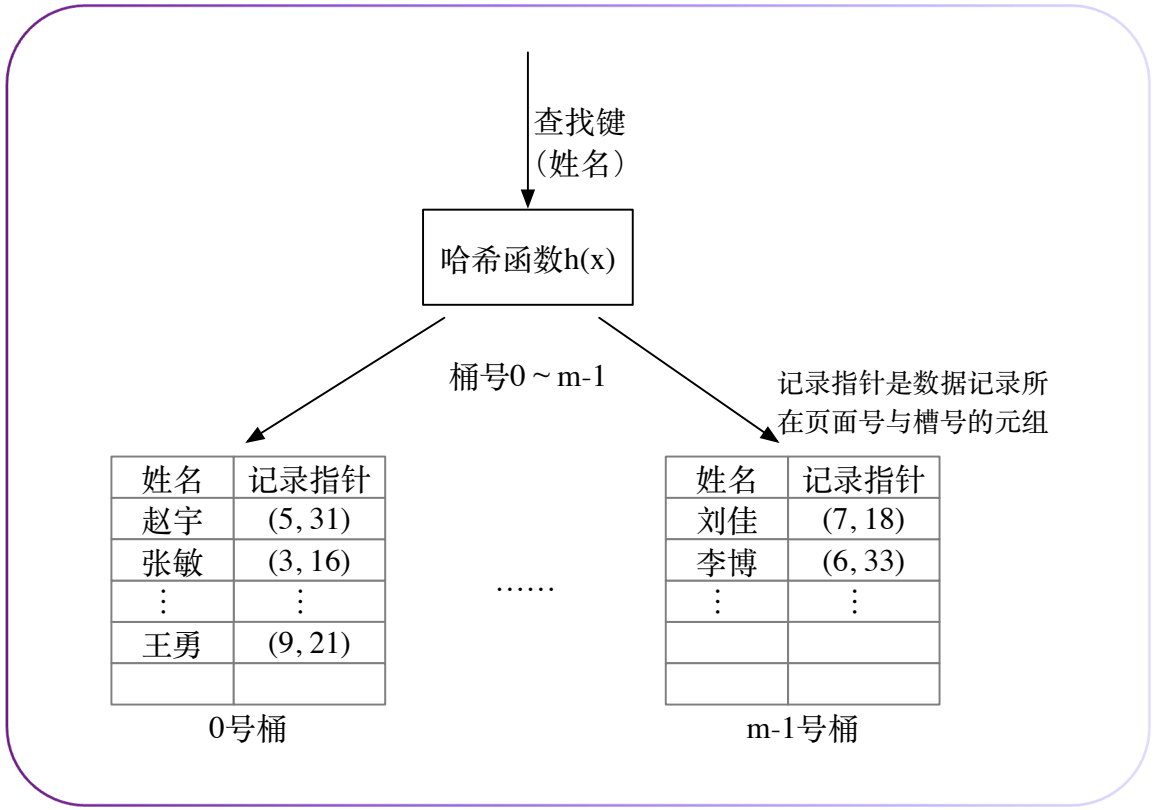
- **MD5**
- **SHA256**

} 加密哈希，防止构造哈希碰撞，用于认证、防篡改等情况，速度慢

- **CRC-64(1975)** 用于错误校验
- **PostgreSQL Lookup3Hash(2006)**
- **MurmurHash(2008)** 快速、通用的哈希函数
- **Google CityHash(2011)** 更适用于长度较短的查找键
- **Facebook RocksDB XXHash(2012)** 速度最快

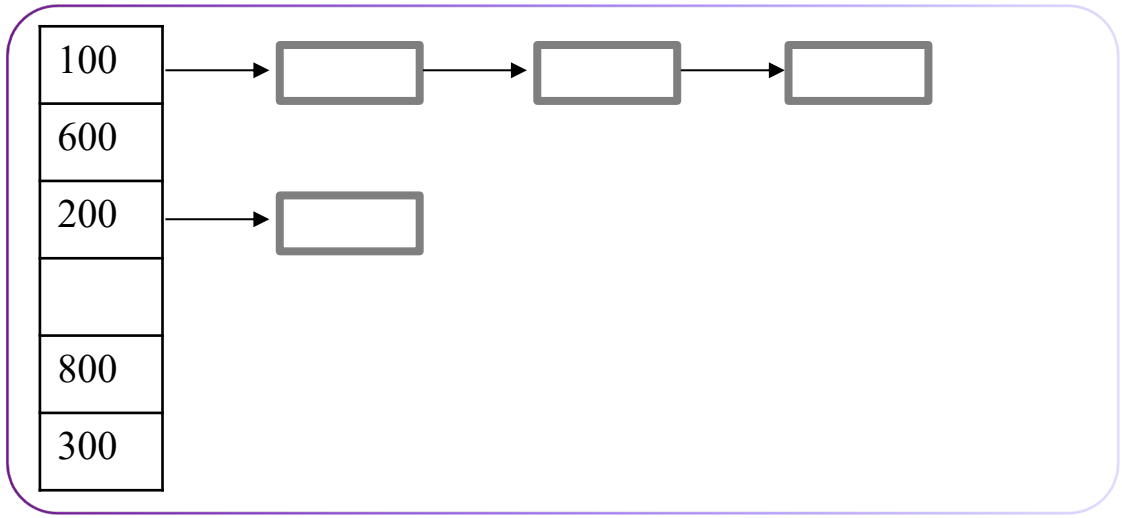
哈希表

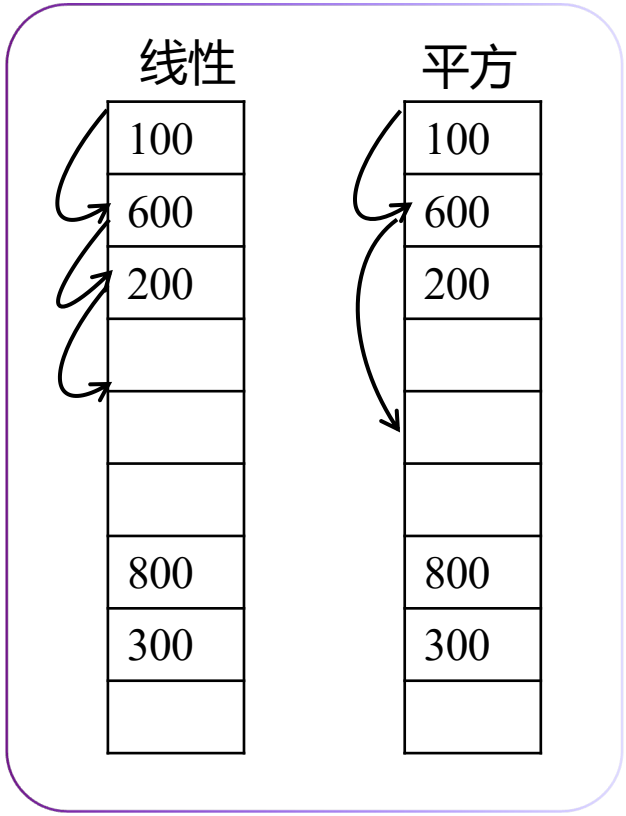
- **持续向哈希表插入数据，桶溢出不可避免**
- **连接空页面形成链表**
  - 降低查找性能
- **静态哈希表**
  - 桶数固定
  - 溢出时需要分配更多桶，重新哈希
- **动态哈希表**
  - 动态调整桶数的哈希表
  - 可扩展哈希表
  - 线性哈希表



静态哈希表

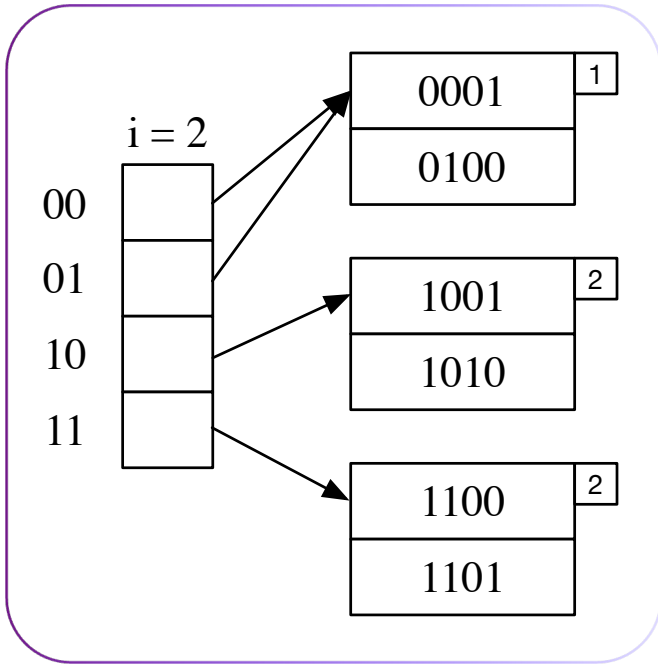
- **闭哈希：当插入哈希值满时，寻找空位置**
  - 线性探测新位置（容易堆积）
    - $+i$
  - 平方探测新位置
    - $+i^2$
- **开哈希：当插入哈希值满时，新建哈希桶，通过哈希桶链表链接**





可扩展哈希表

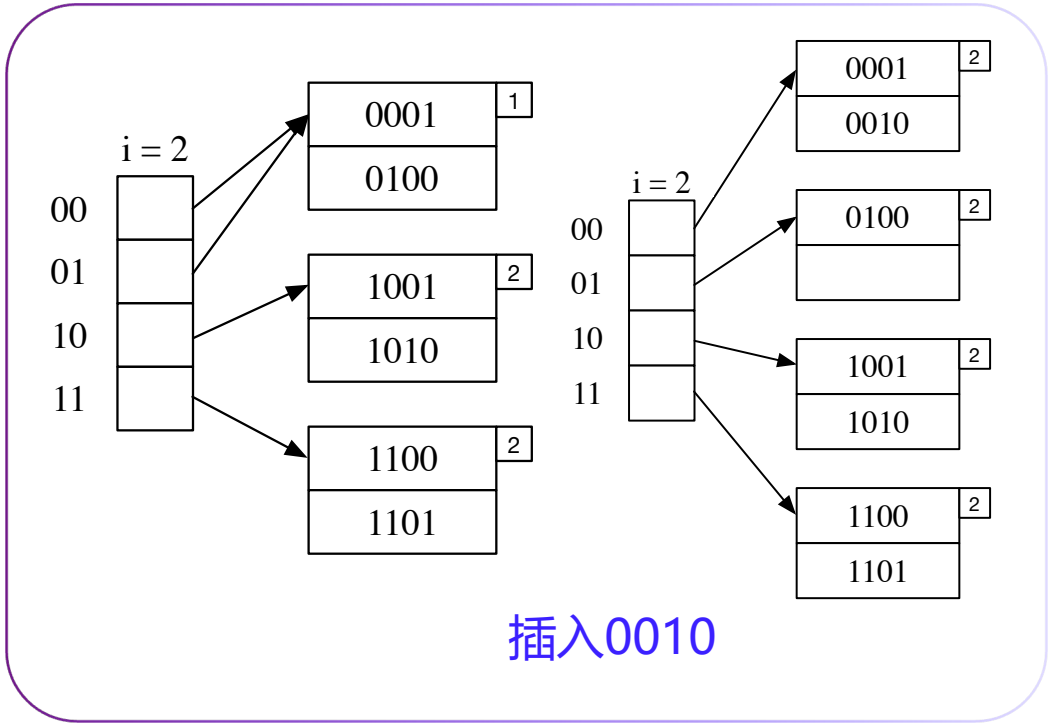
- **带有二进制位数的桶指针**
- **带有二进制位数的桶（每个桶通过桶索引页面形式存储对应的数据）**
- **查找键值K时**
  - 计算K哈希值的前i位（全局深度i）
    - 用于桶的扩容
  - 根据前i位找到对应的哈希桶（局部深度j）
    - 为了节省空间，（稀疏）局部哈希桶合并到一起管理
    - 当局部哈希桶满时，用于分裂



可扩展哈希表

➢ 插入键值K时

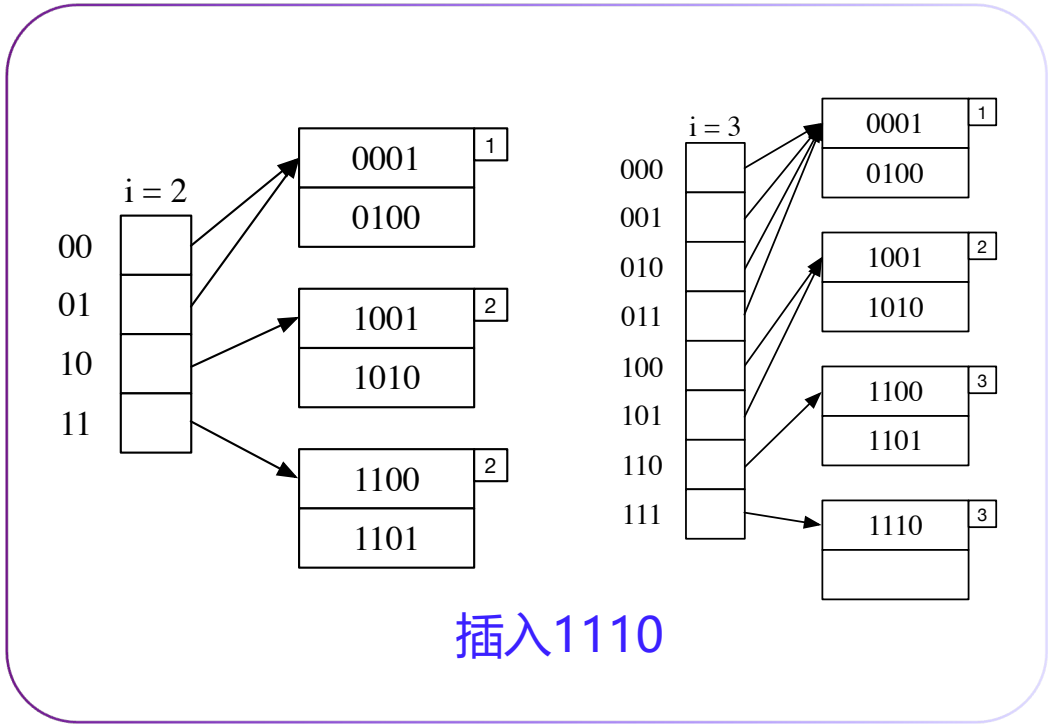
- 计算K哈希值的前i位
- 根据前i位找到对应的哈希桶
- 如果哈希桶有空间，则插入
- 否则根据哈希桶的位数j来决定如何分裂
- 如果j<i: 分裂为两个桶：$b_1 b_2 \cdots b_j 0$与$b_1 b_2 \cdots b_j 1$
- 分裂后的索引页面中如果有空余位置则进行插入，否则再次执行分裂过程($j<i$)或扩容过程($j=i$)。



可扩展哈希表

➢ 插入键值K时

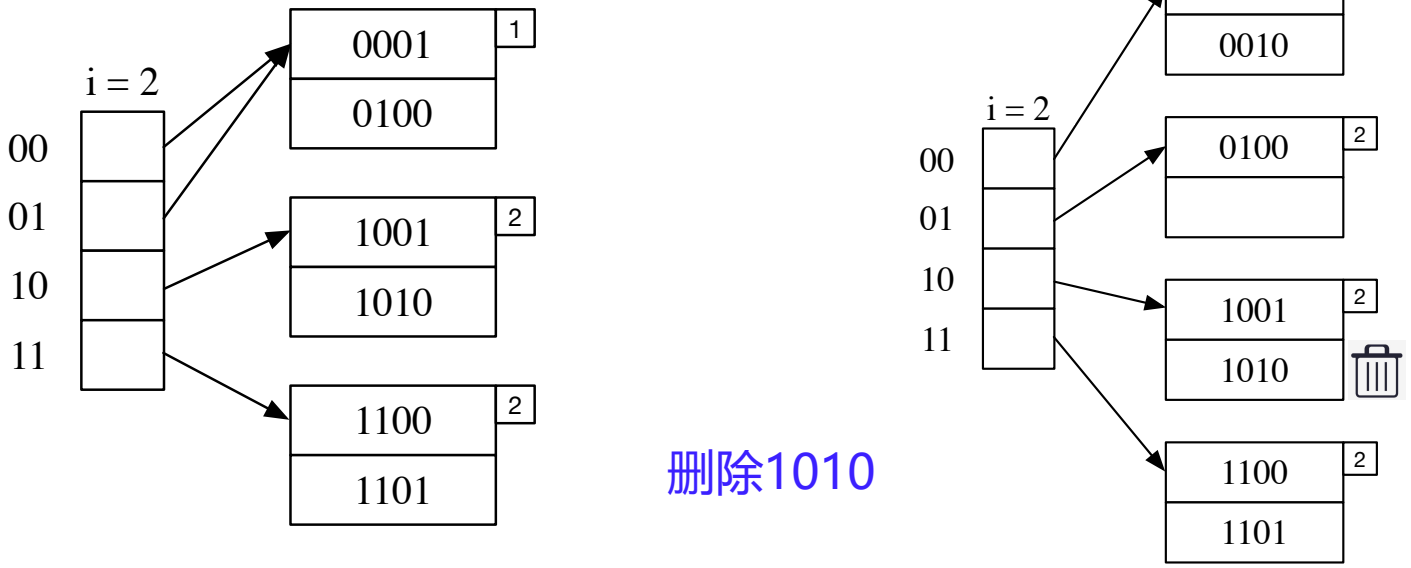
- 计算K哈希值的前i位
- 根据前i位找到对应的哈希桶
- 如果哈希桶有空间，则插入
- 否则根据哈希桶的位数j来决定如何分裂
- 如果j=i: 对哈希表进行扩容
  - 申请容量为原哈希表两倍的桶数组
  - 原桶数组中编号k的桶中数据复制到新桶数组中编号分别为$2k$与$2k+1$的桶中
  - 将哈希表使用哈希值位数i增加1
  - 完成扩容后，再次执行插入过程便能够进行索引页分裂



可扩展哈希表

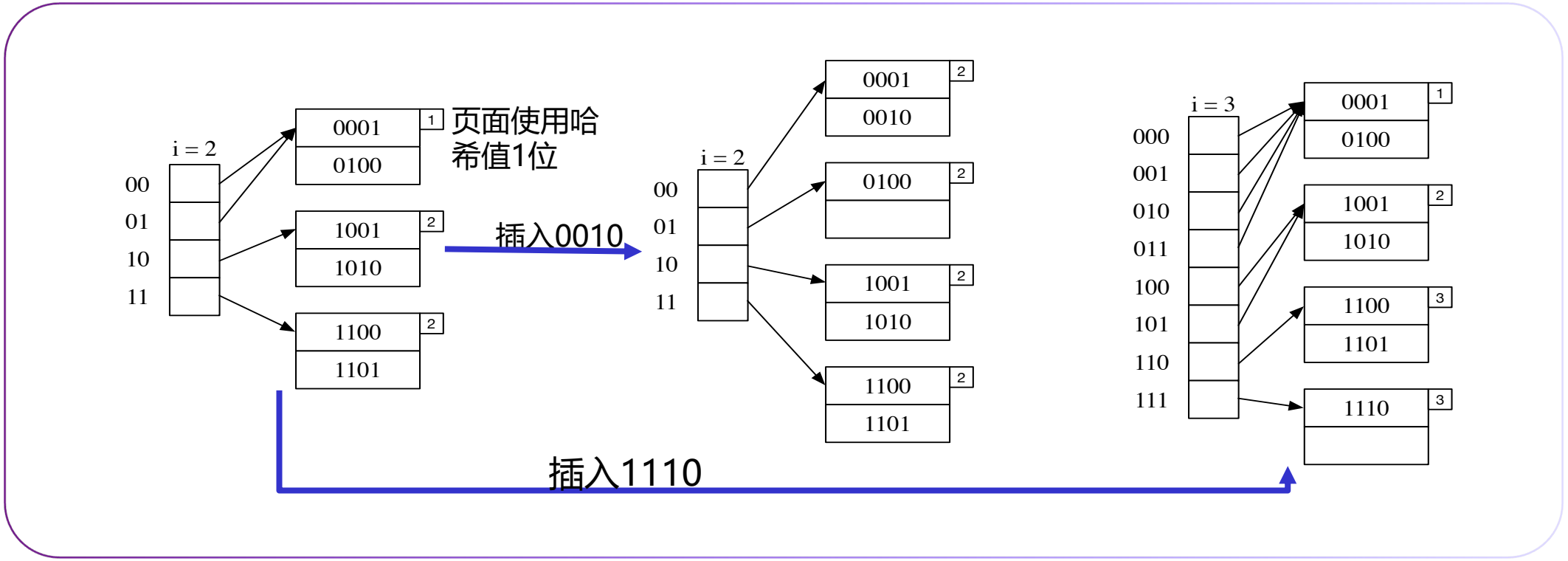
➢ 删除键值K时

- 标记删除（墓碑）



可扩展哈希表

- 使用哈希值二进制表示的前i位
- 每次扩容容量加倍，使用哈希值位数加1
- 索引记录移动采用懒惰策略

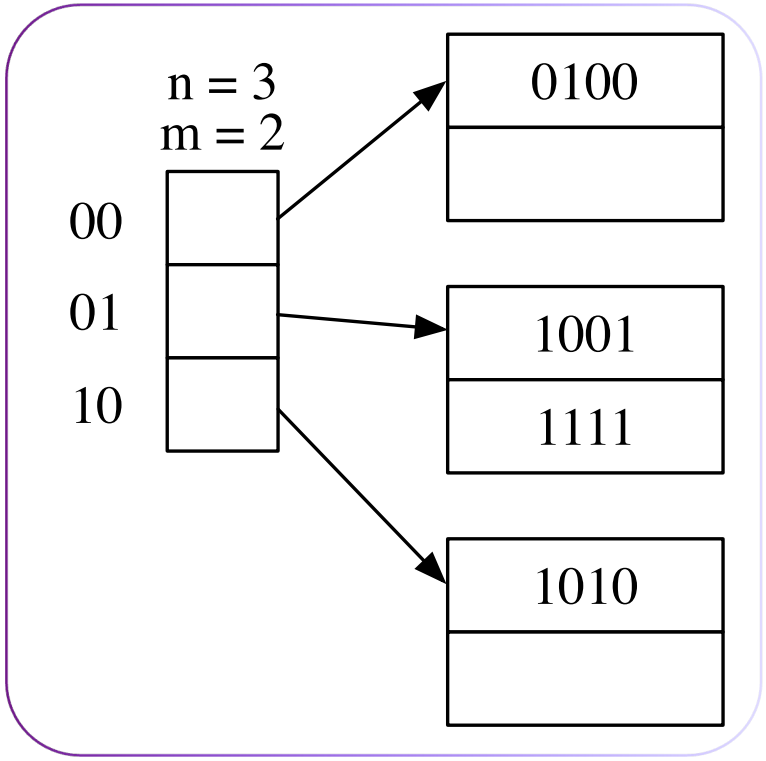


可扩展哈希表的问题

- **哈希表每次容量加倍时需要阻塞对整个哈希表的访问，降低了系统性能；**
- **如果少量桶多次溢出，则需要频繁对哈希表进行扩容，造成一部分桶空，导致空间浪费（例如所有元组哈希值前n位都相同，则需要频繁扩充）。**
- **线性哈希表**
  - 允许在页面后链接存放溢出数据的页面，更加节省空间
  - 能够实现在哈希表扩容时仅需要对个别桶进行分裂，不会大规模阻塞对哈希表的访问。

线性哈希表

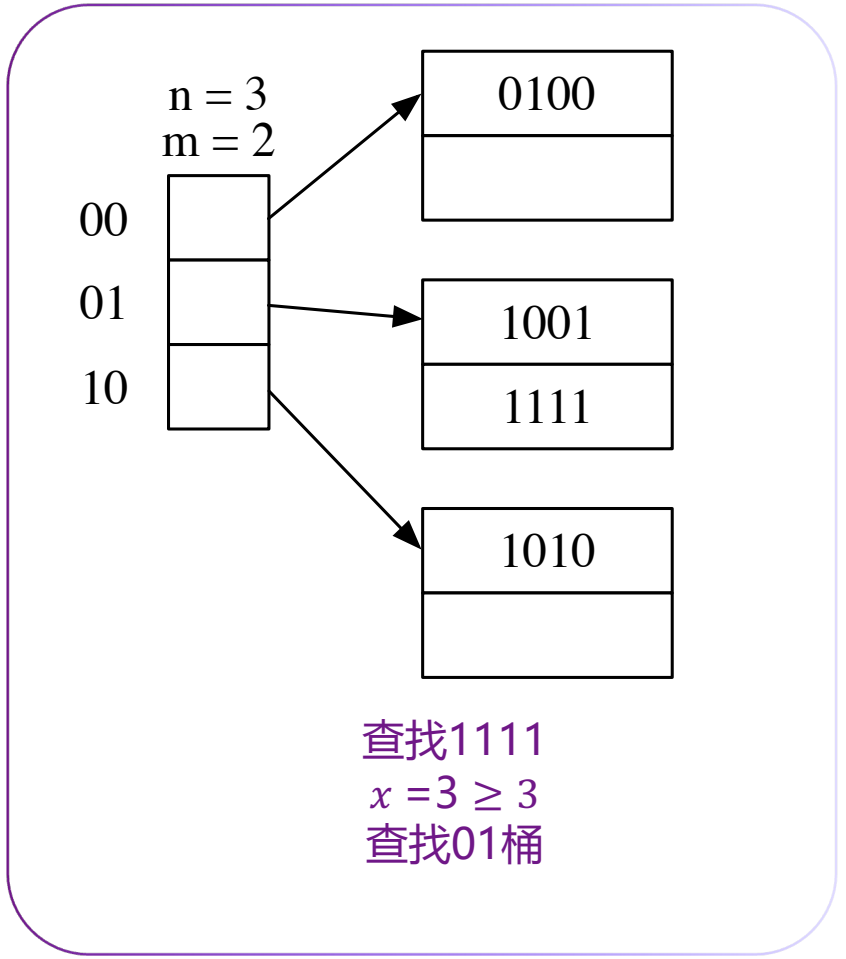
- 线性哈希表使用哈希值的后m位。
- 线性哈希表不再限制桶数组容量为2的整数幂，对容量n的桶数组，二进制位数$m = \lceil log_2 n \rceil$。
- 插入时根据桶数组填充率来进行扩容。
- 线性哈希表也分为两部分：
  - 带有二进制位数（m）的桶指针（当前的桶容量n）
  - 桶（每个桶通过桶索引页面形式存在对应的数据）



线性哈希表

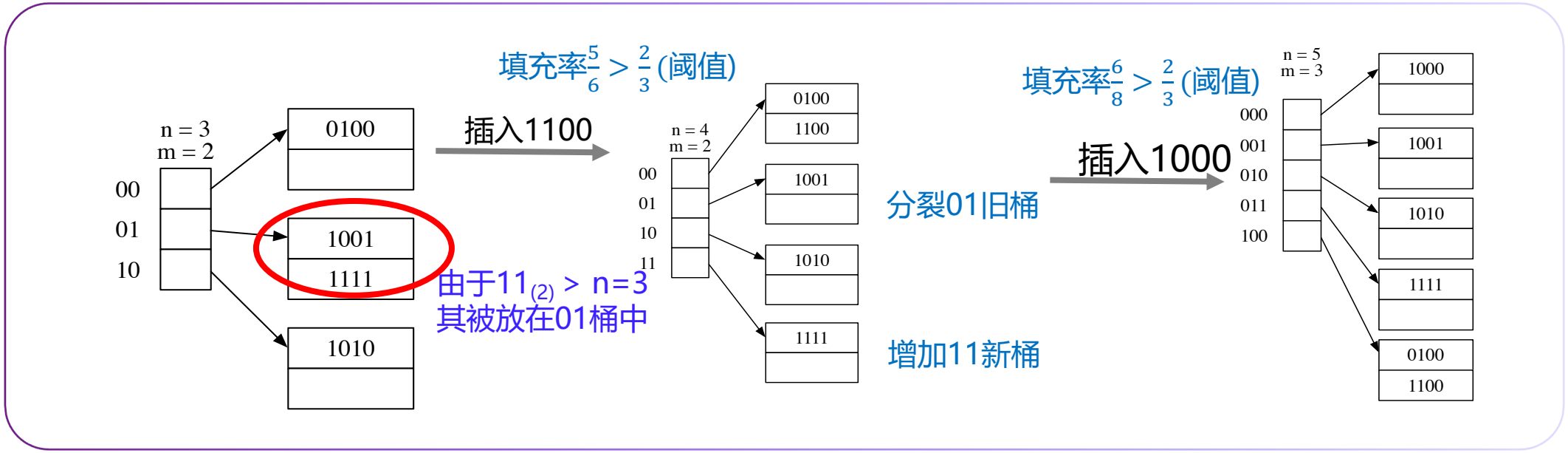
➢ 查找键值K算法

- 由于桶数组并非2的整数幂，而哈希值后m位取值范围为$0 \sim 2^m - 1$，故哈希值对应的桶可能不存在
- 设键哈希值后m位为$b_1 b_2 \cdots b_m$，对应值为$x(x \leq 2^m - 1)$
- 如果$x < n$，可以找到对应的桶。
- 如果$x \geq n$，则无法找到对应的桶，由于使用位数为m，故容量已超过$m-1$位哈希值表示的范围，即有$x \geq n > 2^{m-1}$，此时线性哈希表查找$x - 2^{m-1}$号桶。由于此时$2^{m-1} < x \leq 2^m - 1$，故有$b_1 = 1$，即查找$0 b_2 \cdots b_m$号桶。



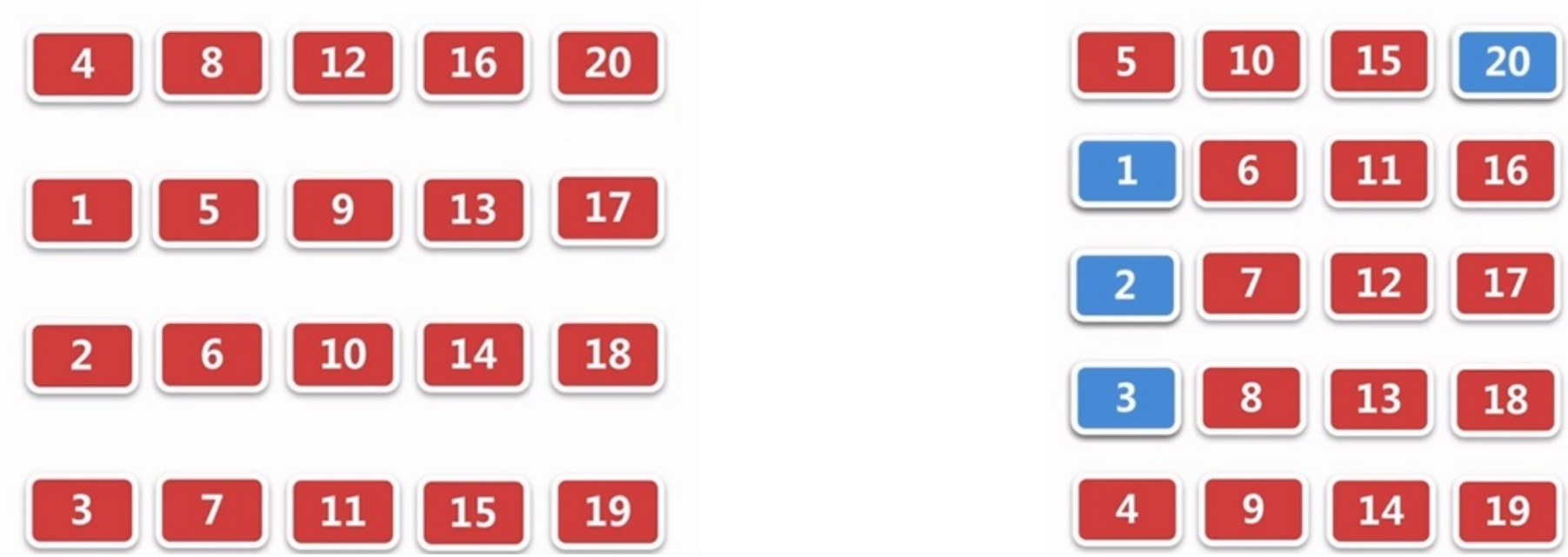
线性哈希表

- 插入键值K，根据填充率决定是否分裂桶页面
- 当前桶数为n，每个桶存放k条数据，哈希表中存放索引条目一共r条，填充率为$\frac{r}{nk}$
- 如果填充率小于等于阈值τ，则新增溢出桶以链表方式链接在应插入的桶页面之后
- 填充率高于设定阈值τ，线性哈希表加入一个新桶并将一个旧桶进行分裂。
- 增加后，$n=n+1$。新桶对应哈希值为n，并设置$n=n+1$。



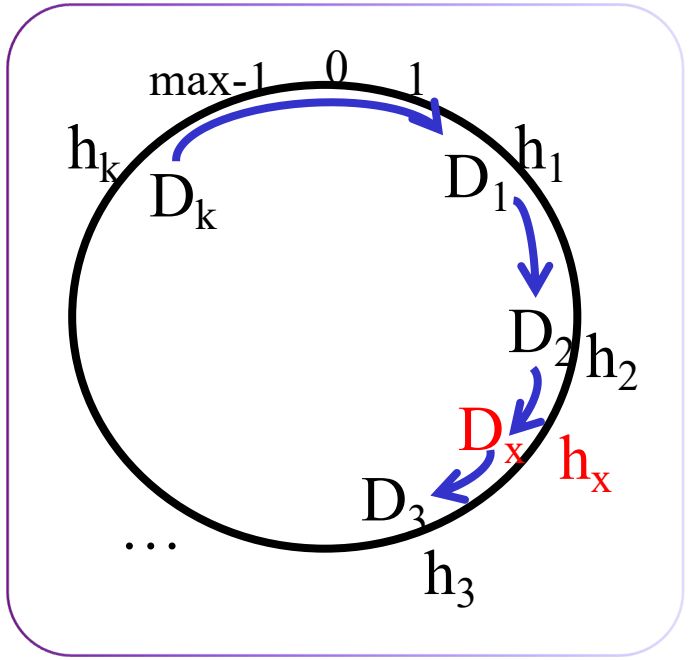
一致性哈希

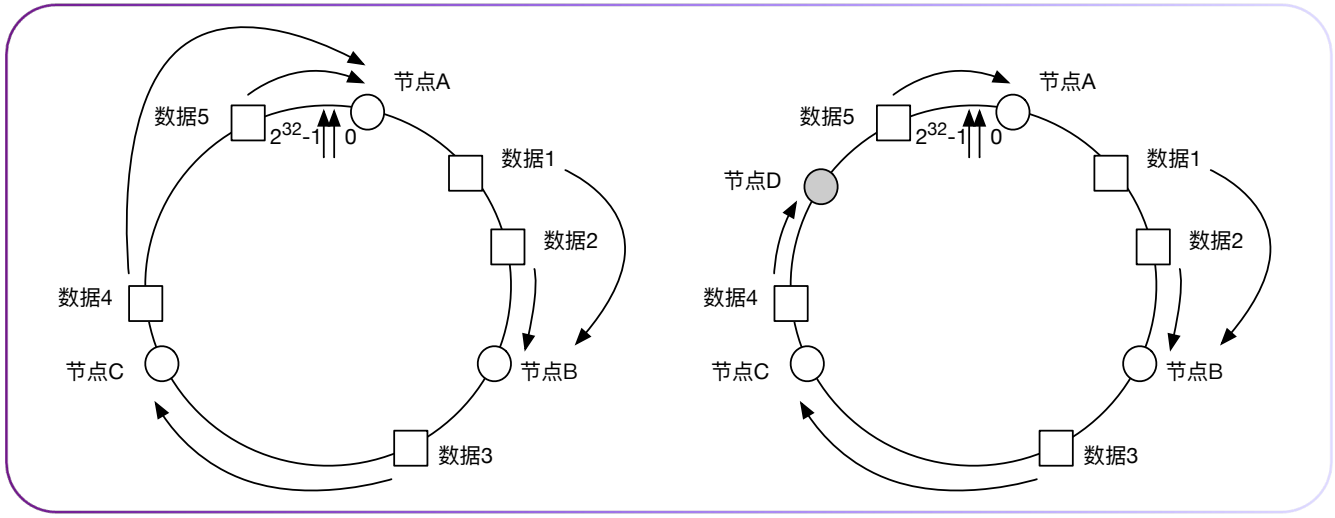
- 如何从4个哈希分片扩展到5个哈希分片？重新哈希?
- 大量数据搬迁



一致性哈希

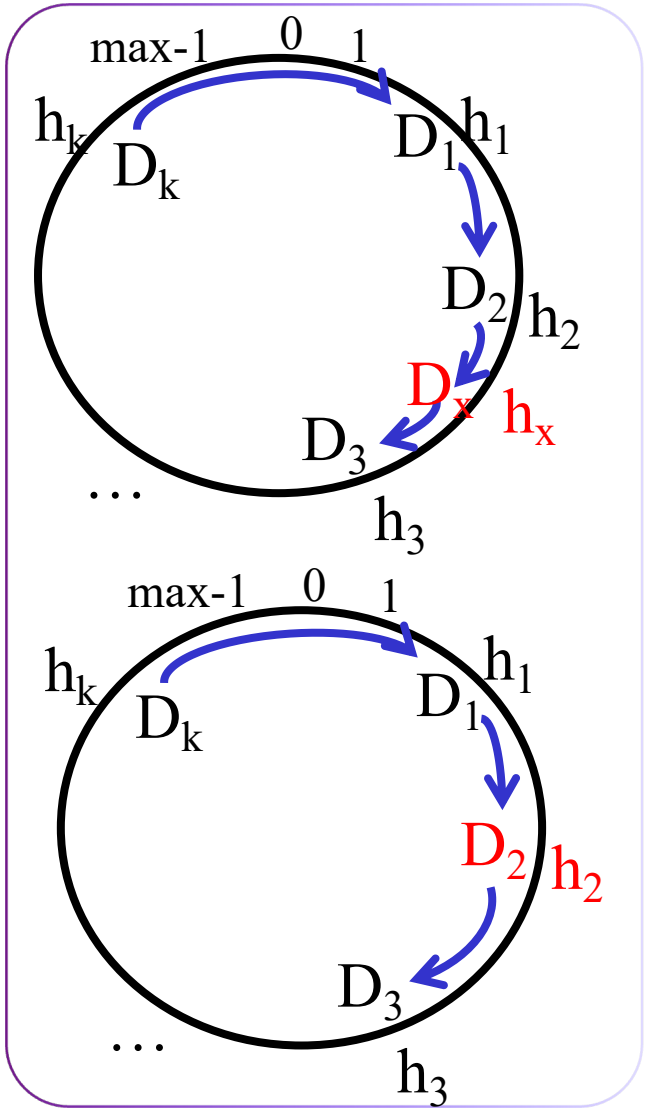
- 将哈希值组织成为一个环[0,max)
- 在哈希环上选择k个值作为数据分片
  - k个值从小到大记为$h_1,h_2, ..., h_k$。
  - k个哈希分片记为$D_1, D_2, ..., D_k$。
- 哈希值在$[h_{i-1},h_i)$之间的数据放在D_i分片，哈希值在$[h_k, max), [0, h_1)$之间的数据放在D1分片





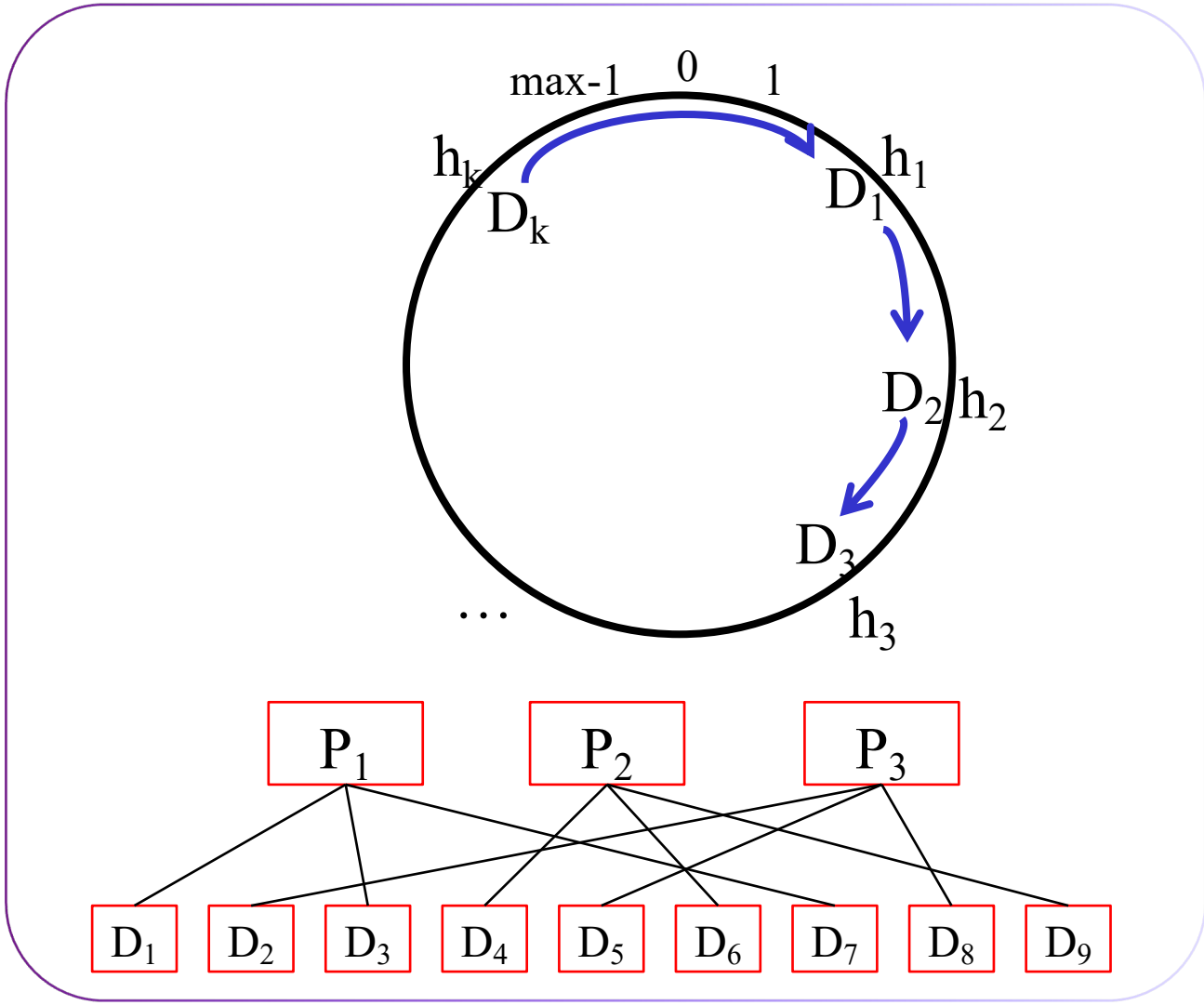
一致性哈希

- **添加一个数据分片**
  - 添加一个哈希值h_x，假设$h_2<h_x<h_3$
  - 添加一个数据分片D_x
  - 将D_3中的一部分数据,满足$[h_2,h_x)$移到D_x
- **删除一个数据分片**
  - 假设删除数据分片D_i
  - 将D_i中的数据移到D_{i+1}
- **添加和删除分片只影响一个数据分片**



一致性哈希

- **负载均衡问题**
  - 某些数据分片数据较多
  - 某些数据分片数据较少
- **增加虚拟节点**
  - 虚拟节点对应哈希环上的值
  - 物理节点包含多个虚拟节点



目录

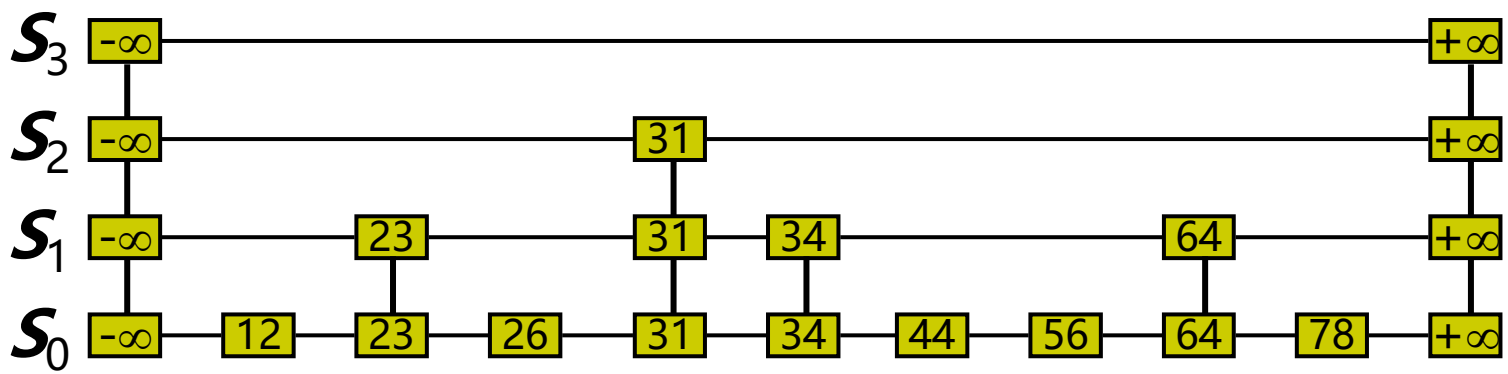
1. 索引概述
2. B+树索引
3. B+树并发控制
4. 哈希索引
5. **SkipList跳表索引**
6. LSM树索引
7. 位图索引
8. 多维索引

   ——R树、KD树、Quadtree、Z曲线

目录

5. SkipList跳表索引

- 跳表结构：内存索引
- 跳表插入
- 跳表查询
- 跳表删除



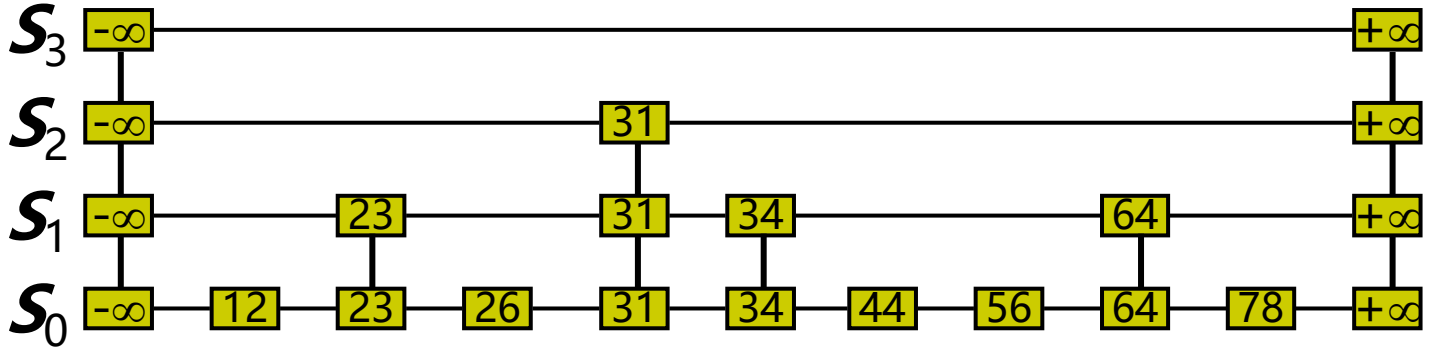
SkipList 跳表

➢ SkipList S 包含多个链表 $S_0, S_1, \ldots, S_h$

- 每个链表S_i 包含 $+\infty$ 和 $-\infty$ 以及S中的键值，键值单调不递减
- S_0 包含S中的全部键值
- 每层的链表S_i包含上层链表的键值S_{i+1}，

$$S_0 \supseteq S_1 \supseteq \ldots \supseteq S_h$$

- S_h 只包含$+\infty$ 和 $-\infty$

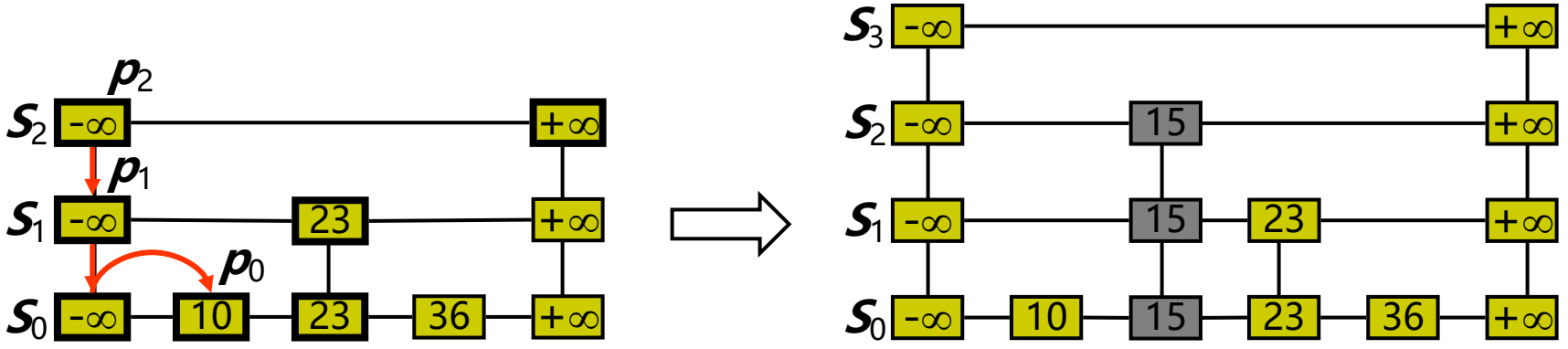


SkipList 跳表

➢ 插入索引项 (x, o)，使用随机算法：

- 通过概率（抛硬币）方法得到i： 抛硬币直到出现正面，抛硬币的次数即为i
- 如果 $i \geq h$, 添加新链表 $S_{h+1}, \ldots, S_{i+1}$
- 在链表$S_0, S_1, \ldots, S_i$中查找大于等于x 的位置 $p_0, p_1, \ldots, p_i$
- 对于 $j \leftarrow 0, \ldots, i$, 将(x, o) 插入到 S_j 的 位置 p_j

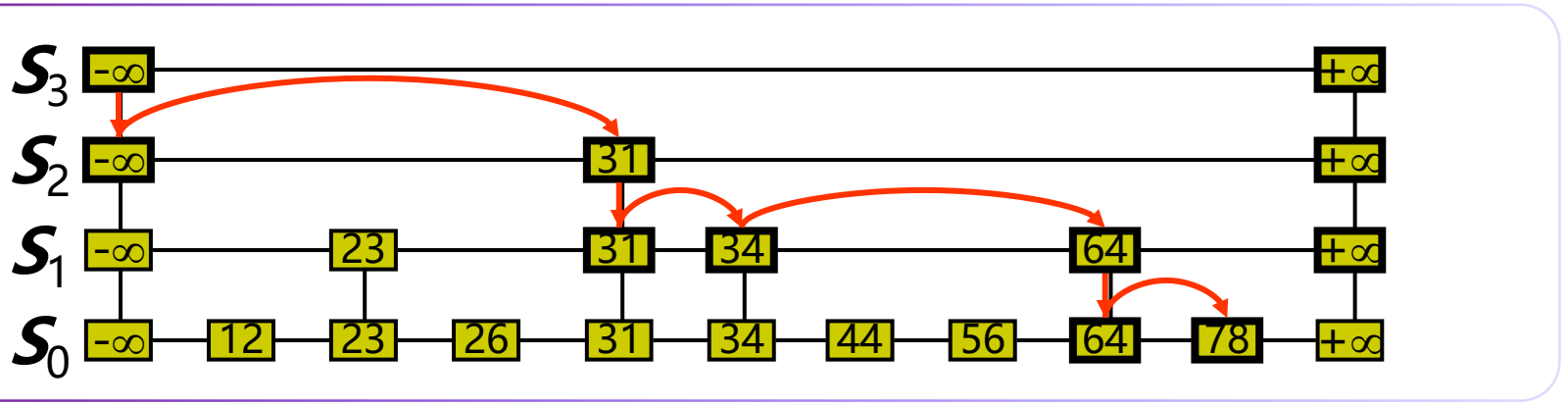
➢ 例子: 插入键值 15, i = 2



SkipList 跳表

➢ 查找键值*x*

- 从最上层开始查找，对于每个位置p, 比较x 和 y ← key(next(p))的大小：
  - ***x*** = ***y***：返回***element***(***next***(***p***))
  - ***x*** > ***y***:本层向前扫描
  - ***x*** < ***y***：跳转到下一层（该节点指针指向的下层节点）
- 如果到最底层还没找到，则返回null

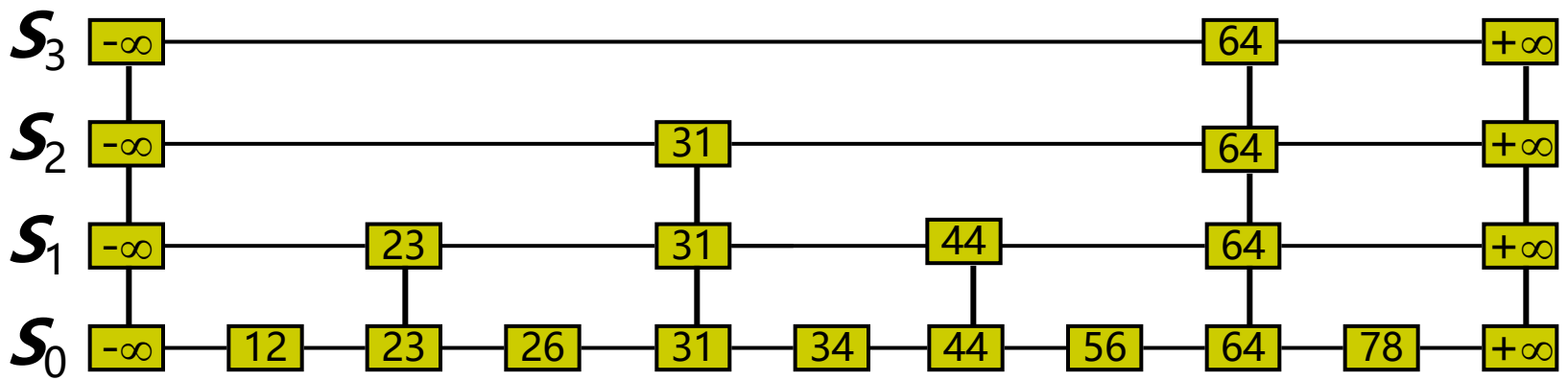


SkipList 跳表

➢ 搜索的复杂度

- 时间： O(log n)
- 空间：O(n)

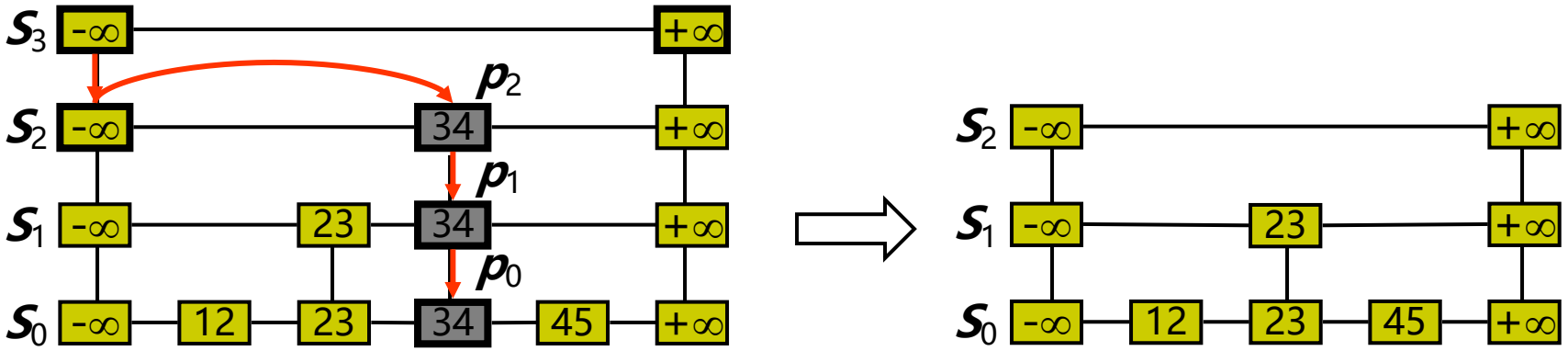
$$n+\frac{n}{2}+\frac{n}{4}+\frac{n}{8}+\cdots=\sum_{i=0}^{\infty}\frac{n}{2^i}=2n$$



SkipList 跳表

➢ 删除索引项(*x, o*)

- 在S 中找到包含 x 的链表$S_0, S_1, \ldots, S_i$，获得大于等于x 位置 $p_0, p_1, \ldots, p_i$
- 将位置 $p_0, p_1, \ldots, p_i$的键值从链表$S_0, S_1, \ldots, S_i$中删除
- 如果有多个只包含 +∞ 和 -∞ 的链表，仅保留最低层的一个



目录

1. 索引概述
2. B+树索引
3. B+树并发控制
4. 哈希索引
5. SkipList跳表索引
6. **LSM树索引**
7. 位图索引
8. 多维索引

——R树、KD树、Quadtree、Z曲线

目录

6. LSM树索引

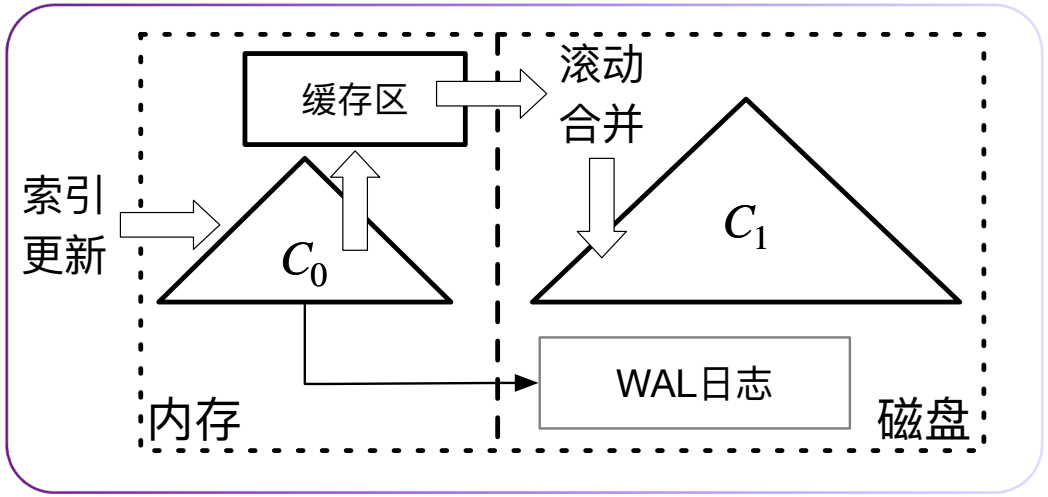
- 思想：顺序写+非原地更新+定期归并+冷热数据分层
- 2阶LSM树
  - 2阶LSM树滚动合并
  - 2阶LSM树索引删除与更新
- 多阶LSM树
  - 布隆过滤器
  - 分区优化
  - LSM树实例：LevelDB
  - LSM树与B+树对比

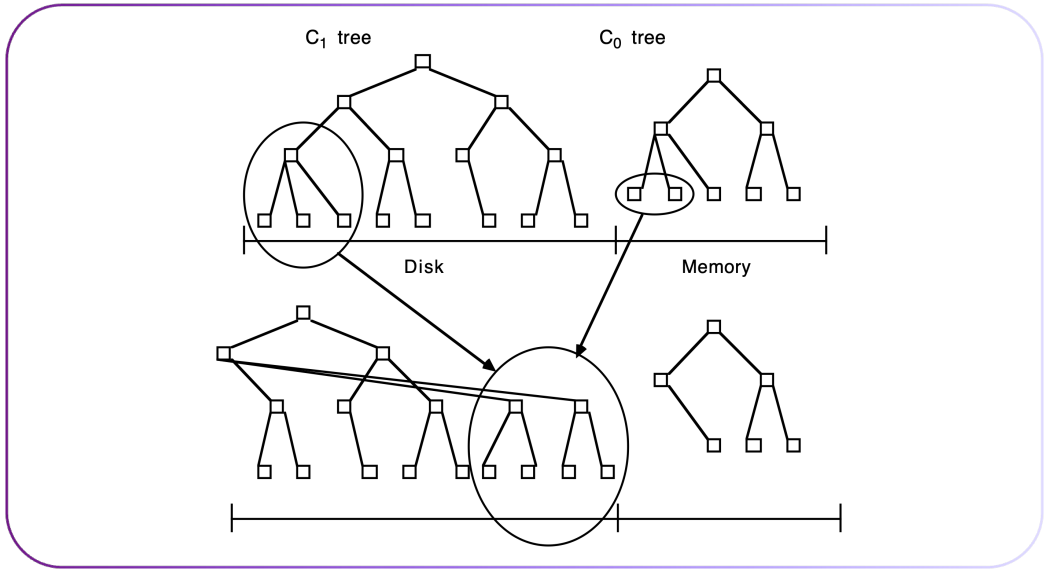
LSM树索引

- **动机：更新密集场景**
  - 微博、微信：数据追加写入，不允许修改
- **B+树在更新密集的场景产生大量随机I/O**
- **LSM树将更新操作转化为批量顺序写入**
- **思路：多级索引积累更新后进行索引合并**
  - 索引合并可以通过顺序写入实现

2阶LSM树

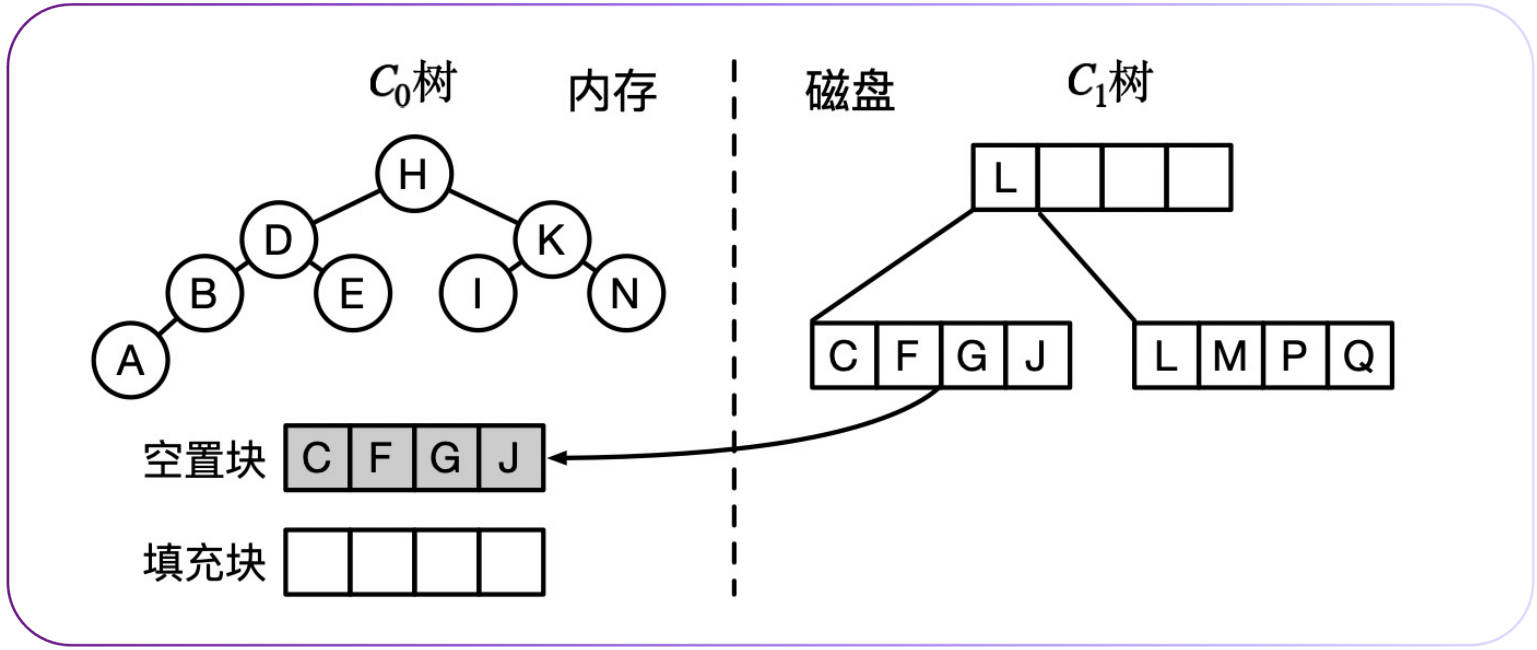
- **内存索引C_0(红黑树、AVL树等)**
  - 缓存区中缓存C1、进行滚动合并
  - WAL持久化C0更新
- **磁盘索引C1(B+树)**
- **查找操作**
  - 先在C0中进行查找，存在则返回
  - 否则在C1中进行查找





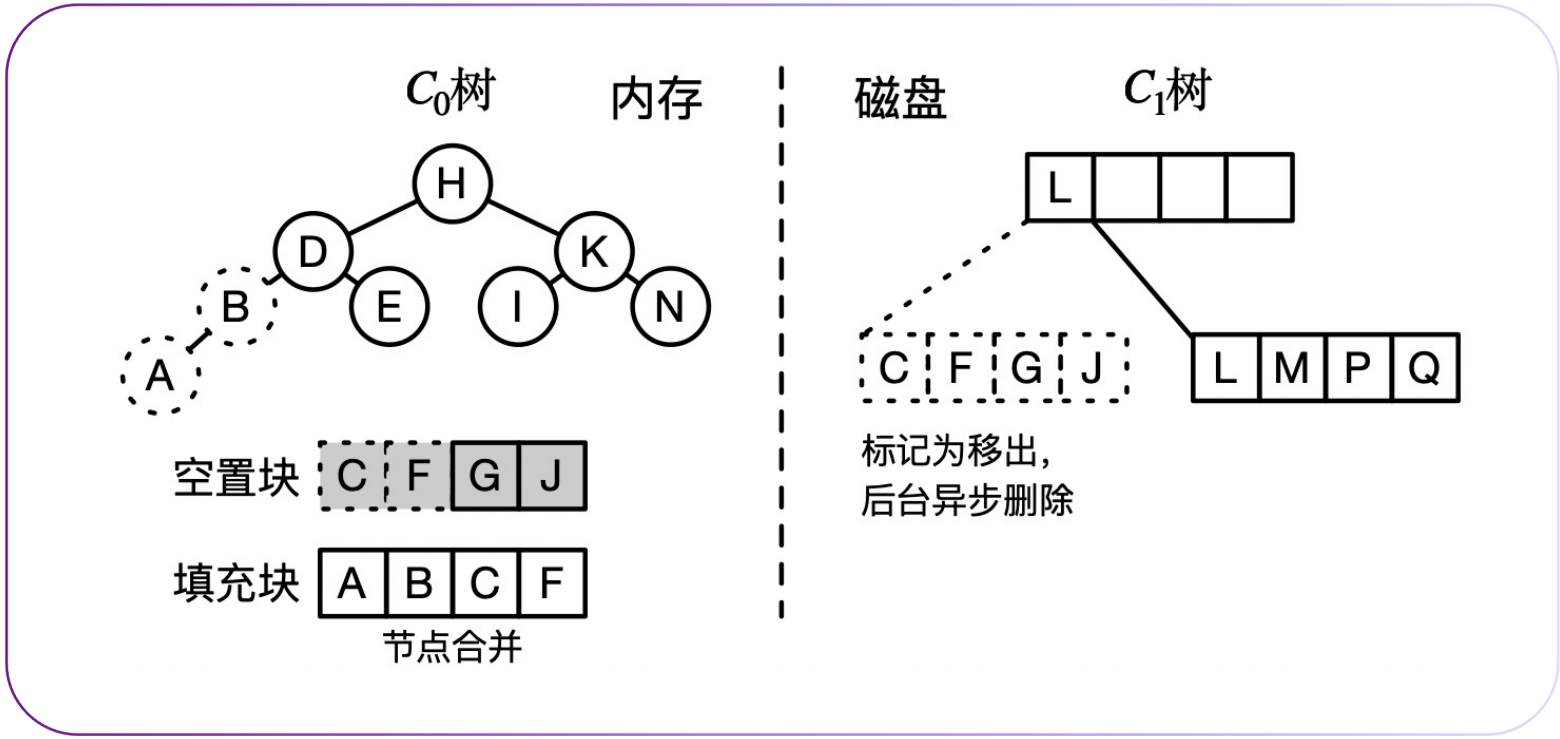
2阶LSM树滚动合并

- **合并条件：C_0树容量达到设定阈值**
- **滚动合并：**
  - 空置块中加载C1树叶子节点的连续页面
  - 填充块中从C0树中取出节点，与空置块进行合并



2阶LSM树滚动合并

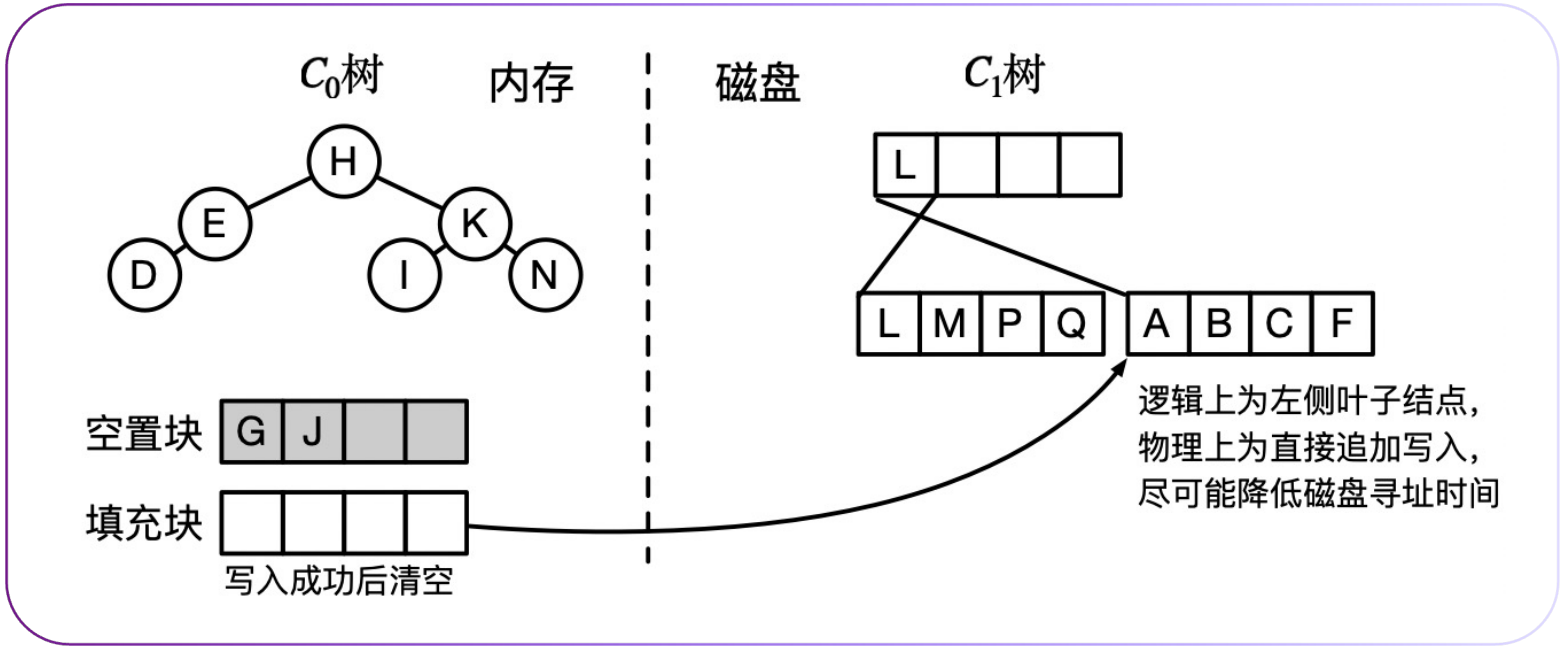
- C1树参与合并的节点被标记移除
- 空置块中来自C1树的剩余的数据随着C0树继续合并被消耗



2阶LSM树滚动合并

➢ 填充块满后追加到C1树文件末尾

- C1树结构变化也通过延迟删除、追加更新的方式实现



2阶LSM树索引删除与更新

- **在C0中查找删除的条目**
- **若待删除条目存在，则直接从C0中删除**
- **否则在C0中插入删除条目**
  - 合并入C_1时进行删除
- **索引更新**
  - 视作删除旧条目，插入新条目

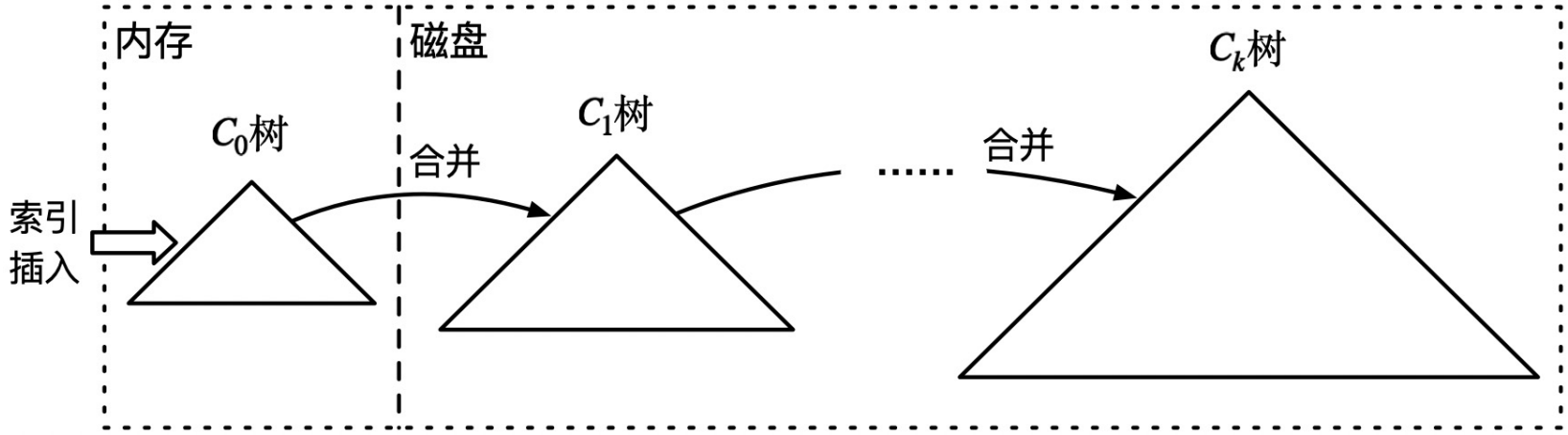
2阶LSM树分析

- 合并时，C1的每个叶子节点平均插入多少个C0索引条目？
- 用滚动合并参数M表示
- $\frac{S_p}{S_e}$为每个页面中索引项数目
- $\frac{S_0}{S_0+S_1}$为C0索引占总索引的比例
- $M = \frac{S_p}{S_e}\frac{S_0}{S_0+S_1}$
- M越小，合并相同数量C0条目需要扫描的C1页面越多（性能降低）
- 随着数据插入，S_1增大，2阶LSM树性能将显著下降

符号	含义
S_e	单个索引项的大小
S_p	磁盘页面的大小
S_0	C_0树索引项总容量
S_1	C_1树索引项总容量

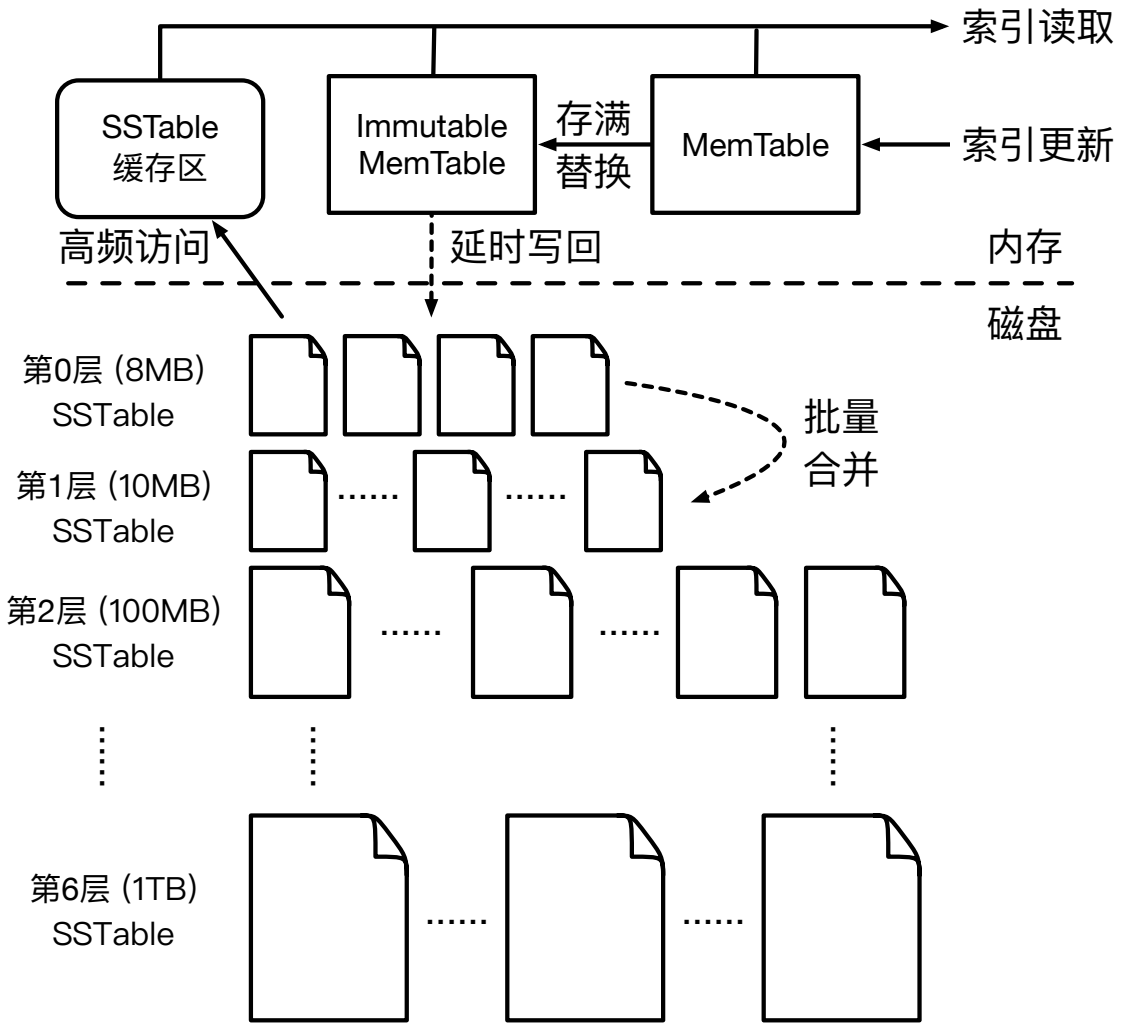
多阶LSM树

- 为降低相邻层级的滚动合并参数，引入磁盘上的多级索引
- 结论：相邻层级容量比例固定时，LSM树I/O成本最低



多阶LSM树

- 插入
  - 先插入内存，如果内存满刷新到第一层
  - 第一层满和第二层合并，以此类推
- 查找
  - 先查找内存，如果找到返回
  - 否则逐层往下找
- 删除
  - 标记删除
- 合并Compaction
  - 异步合并删除



LSM树优化

➢ 滚动合并实现复杂，增加简化条件

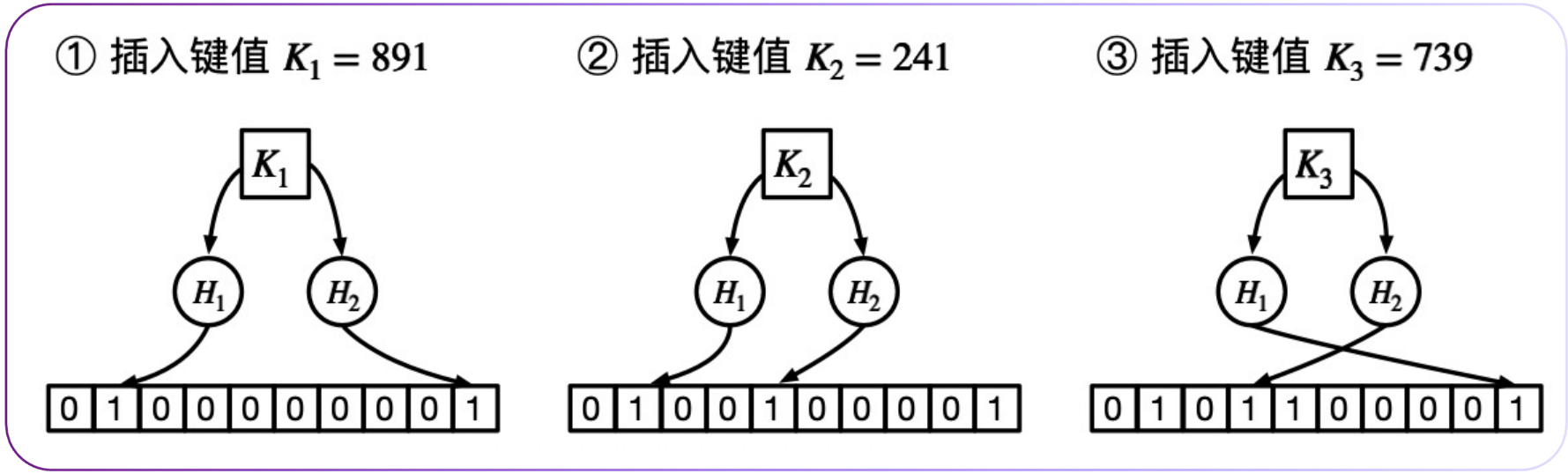
- 内存部分：采用B+树、skiplist等并发安全索引
- 磁盘部分：各层只允许通过合并生成新索引与删除，禁止修改
  - 简化并发控制与故障恢复

➢ LSM树性能问题

- 读放大：查找时若未命中，则需逐层索引查询，增加了I/O
  - 布隆过滤器优化
- 写放大：合并时需要在内存中完成，不断读入、归并、写回
  - 分区优化
- 空间放大：键值可能存在于多层中
  - 异步压缩机制，物理删除被标记删除的数据

布隆过滤器

- **动机：高效检测键值在集合中的存在性**
- **结构：m比特的0/1数组arr，k个哈希函数**
- **插入示例：$m = 10, k = 2$的布隆过滤器**
  - $H_1(K_1)\%m = 1$，将arr第1位置1
  - $H_2(K_1)\%m = 9$，将arr第9位置1



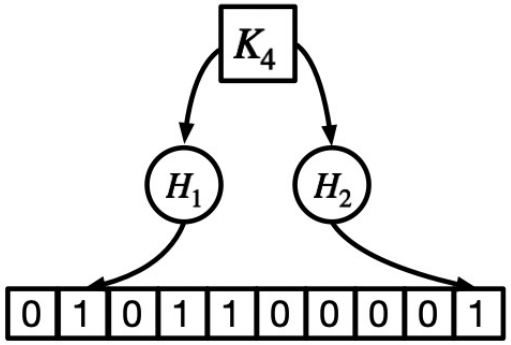
布隆过滤器

➢ 检验示例

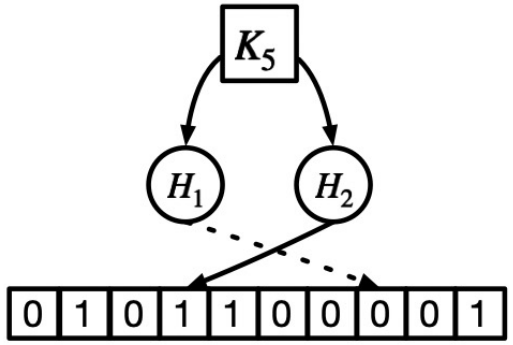
- $H_1(K_4)\%m=1, H_2(K_4)\%m=9$，对应比特位均为1（$K_4$可能存在）
- $H_1(K_5)\%m=3, H_2(K_5)\%m=7$，有一个位置为0（$K_5$一定不存在）

➢ 假阳率分析

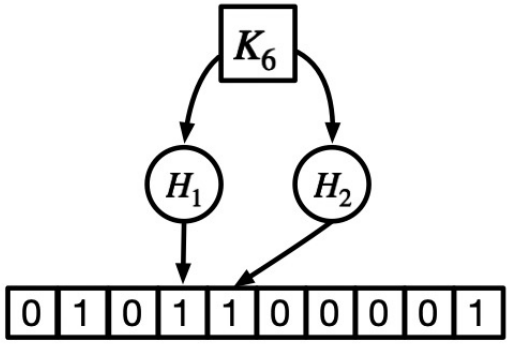
- 键值的k个哈希值位置恰好都为1
- 任一位置在kn次哈希后为1的概率
  - $1-(1-\frac{1}{m})^{kn} \approx 1-e^{-kn/m}$
- 假阳率即任一键值恰好碰撞k次的概率
  - $(1-e^{-\frac{kn}{m}})^k$
- 增大m可以减小假阳率



检验结果: 可能存在



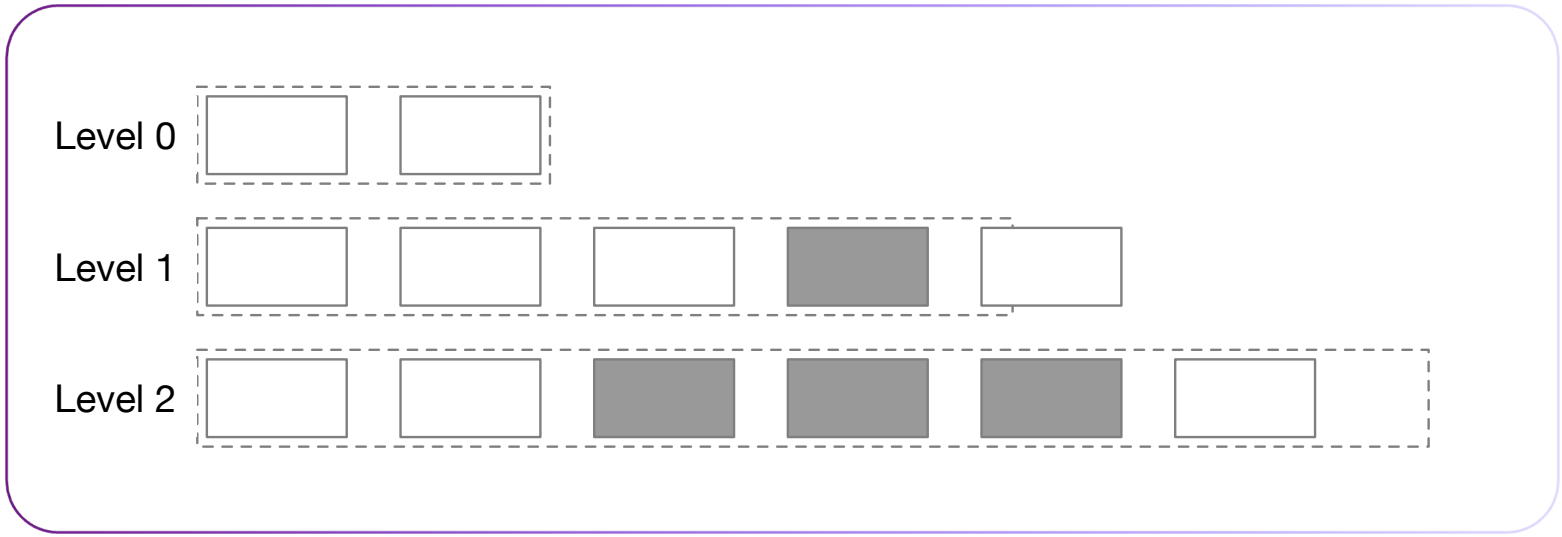
检验结果: 不存在



检验结果: 可能存在

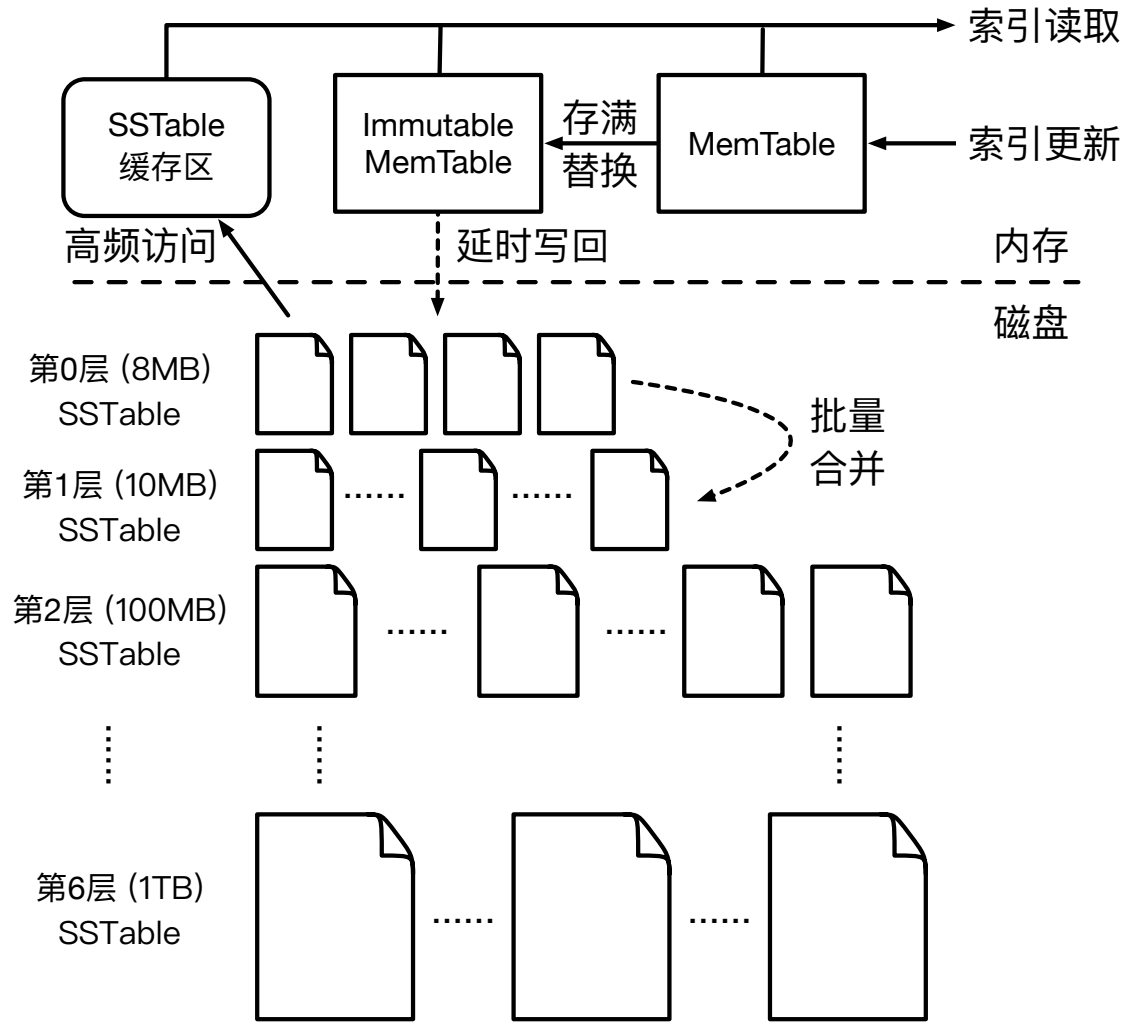
分区优化

- 索引合并的内存与时间开销大
- 将大容量索引拆分成小分区
- 优化针对两种情景
  - 分区无重叠：合并范围不重合时，可以直接追加新分区
  - 偏斜更新：避免无用分区，优化分区的均衡性



LSM实例：LevelDB

- **MemTable**
  - 内存中索引，skiplist
- **SSTable**
  - MemTable存满后转化得到
  - 磁盘上各层索引都由一系列SSTable分区组成
  - 存储键值集合的布隆过滤器
- **异步Compaction**
  - 删除标记过多时，逐层进行合并



LSM与B+树对比

➢ 查找更新适用场景

- B+树：读写均产生树高次随机I/O,适用于读密集型负载
- LSM树：连续写入，读操作需查多层索引，适用于写密集型负载

➢ 整体架构

- B+树：所有页面组成一个树状索引
- LSM树：分为多个层级，各层间相对独立

➢ 内存与磁盘功能

- B+树中内存作为磁盘上索引结构的缓冲区
- LSM树内存索引与磁盘索引相对独立

	B+树	LSM-树
插入	O(log n)	O(1)
删除	O(log n)	O(1)
更新	O(log n)	O(1)
查询	O(log n)	O(L*log n)

L：层数；n：记录数

目录

1. 索引概述
2. B+树索引
3. B+树并发控制
4. 哈希索引
5. SkipList跳表索引
6. LSM树索引
7. **位图索引**
8. 多维索引

   ——R树、KD树、Quadtree、Z曲线

目录

7. 位图索引

- 位图索引结构
- 位图索引上的操作
- 查询执行中生成临时位图索引

Sno	Sname	Sgender	Sage	Sdept
2021310721	赵宇	男	19	CS
2021310722	张敏	女	18	CS
2021310723	王勇	男	18	MA
2021310724	刘佳	女	17	MA
2021310725	李博	男	17	CS

Sgender = 男	10101
Sgender = 女	01010
Sdept = CS	11001
Sdept = MA	00110

位图索引

- 支持多列上有条件限制的查询
- 每条数据占用1bit
  - 1表示满足条件，0表示不满足
- 位图之间进行运算得到满足所有条件的数据位图

Sno	Sname	Sgender	Sage	Sdept
2021310721	赵宇	男	19	CS
2021310722	张敏	女	18	CS
2021310723	王勇	男	18	MA
2021310724	刘佳	女	17	MA
2021310725	李博	男	17	CS

Sgender = 男	10101
Sgender = 女	01010
Sdept = CS	11001
Sdept = MA	00110

位图索引

➢ 查询举例

Select * from Student where Sgender= “男” AND Sdept= “CS”

Sno	Sname	Sgender	Sage	Sdept
2021310721	赵宇	男	19	CS
2021310722	张敏	女	18	CS
2021310723	王勇	男	18	MA
2021310724	刘佳	女	17	MA
2021310725	李博	男	17	CS

Sgender = 男 10101

Sgender = 女 01010

Sdept = CS 11001

Sdept = MA 00110

Sgender=男 AND Sdept = CS

10001

位图索引

ID	Age	Salary
1	25	60
2	45	60
3	50	75
4	50	100
5	50	120
6	70	110
7	85	140
8	30	260
9	25	400
10	45	350
11	50	275
12	60	260

Age

25 100000001000
30 000000010000
45 010000000100
50 001110000010
60 000000000001
70 000001000000
85 000000100000

Salary

60 110000000000
75 001000000000
100 000100000000
110 000001000000
120 000010000000
140 000000100000
260 000000010001
275 000000000010
350 000000000100
400 000000001000

age = 50 & salary = 100

Bitmap index for age = 50 = 001110000010

Bitmap index for salary = 100 = 000100000000

AND: 000100000000

Record #4 !!!

位图索引应用

- **预先构建位图索引**
  - 列的取值种类较少时适用
- **查询执行中进行动态构建**
  - 根据列上的B+树索引/哈希索引得到满足条件数据的位图
  - 位图索引扫描（bitmap index scan)

目录

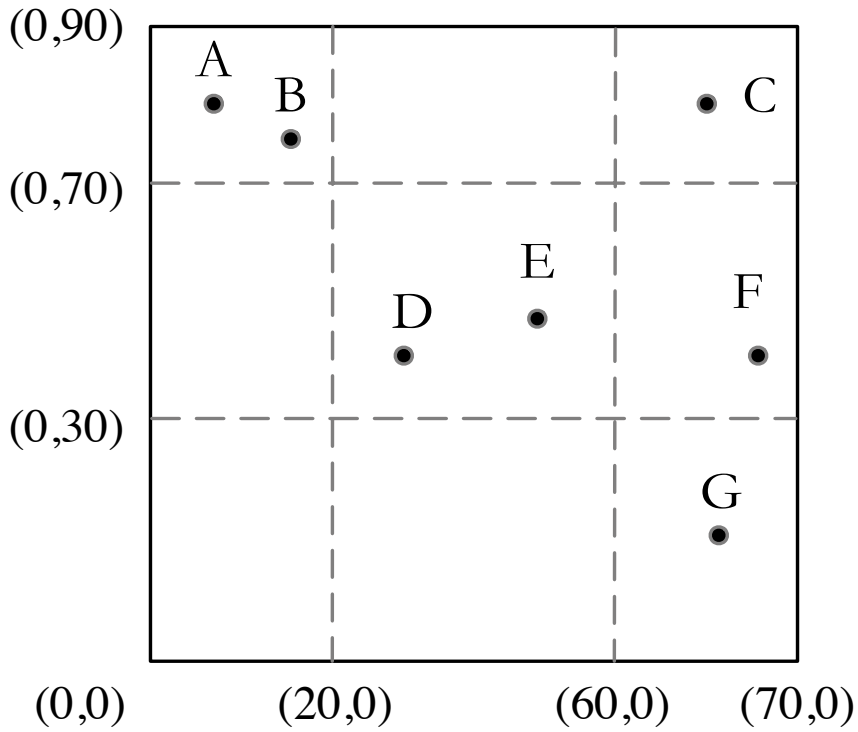
1. 索引概述
2. B+树索引
3. B+树并发控制
4. 哈希索引
5. SkipList跳表索引
6. LSM树索引
7. 位图索引
8. **多维索引**

   ——R树、KD树、Quadtree、Z曲线

目录

8. 多维索引

- 网格文件
- 四叉树
- KD树
- R树
- Z曲线

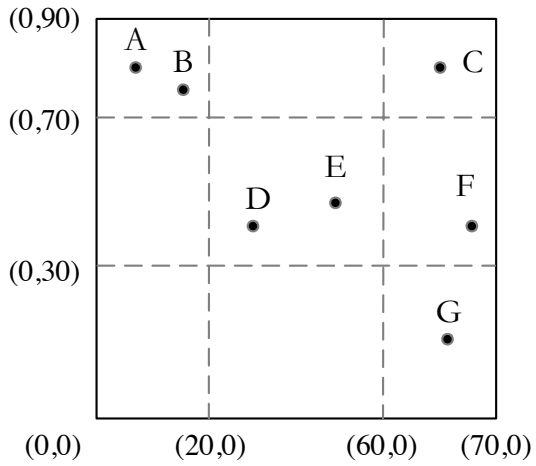


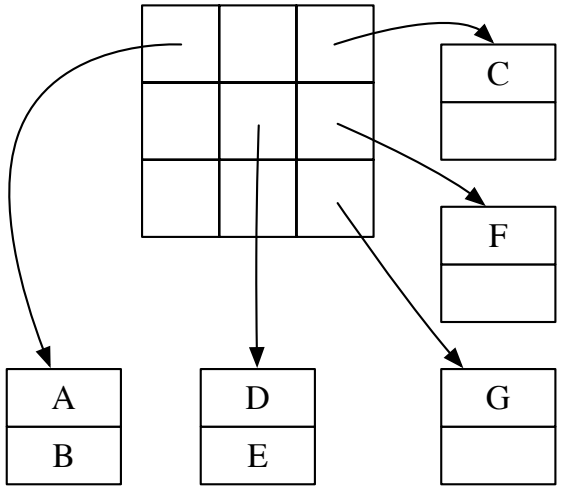
多维索引

- **空间数据与高维数据查询**
- **难以用前述索引解决的查询类型**
- **查询需求**
  - 给定数据点的最近邻点（KNN查）
  - 坐标点处有哪个建筑（点查）
  - 经度/纬度范围内有哪些城市（范围查）

网格文件

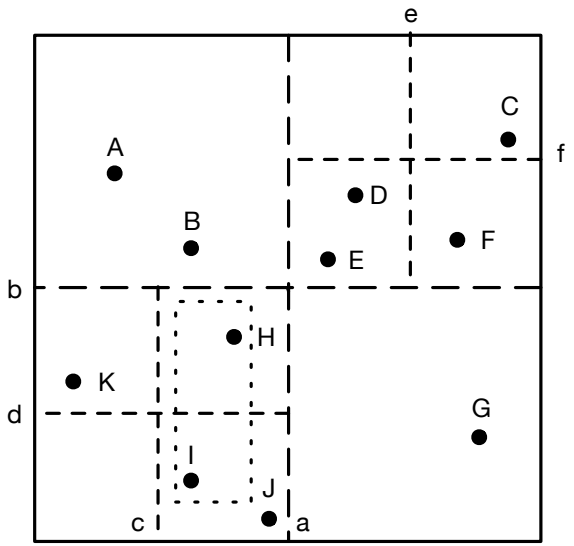
- 将整个空间划分为网格
- 可视作多维的哈希表，每个网格为一个桶
- 最近邻查询
  - 先在给定点落在的网格内查找，再到临近网格查找
- 范围查询
  - 找到与范围有交集的网格

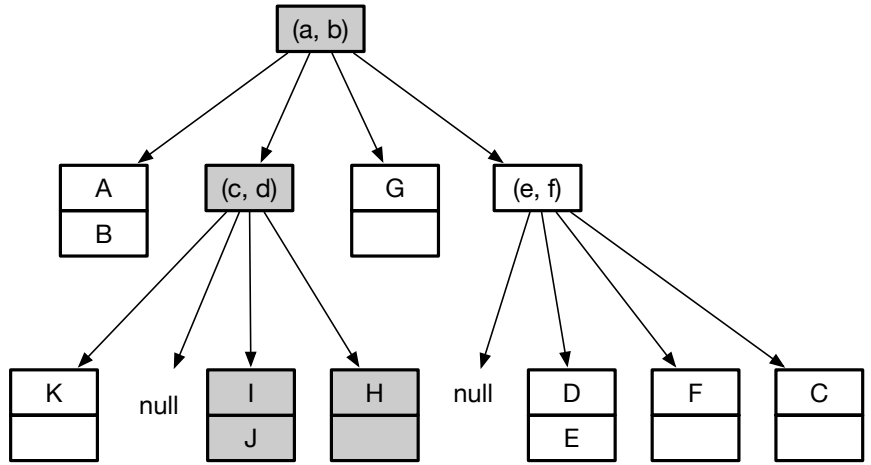




四叉树（Quadtree）

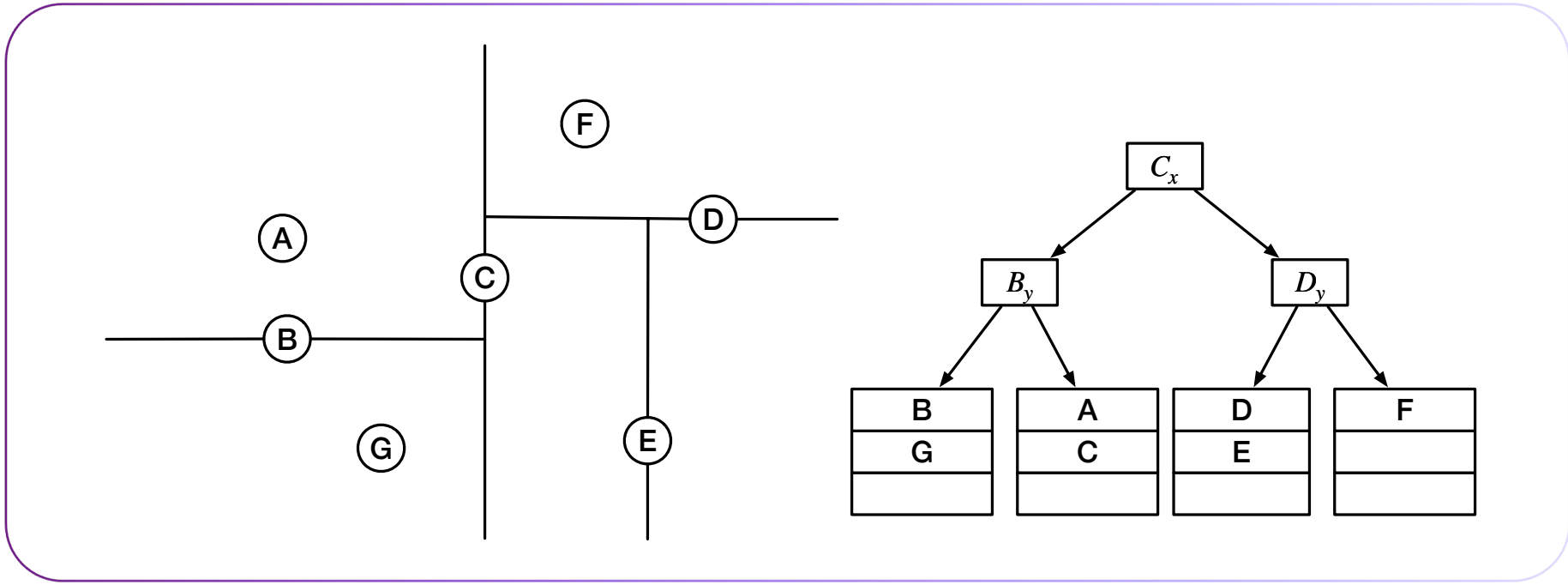
- 网格文件网格静态划分后难以调整
- K维数据的四叉树，每层将子空间等分为2^k份





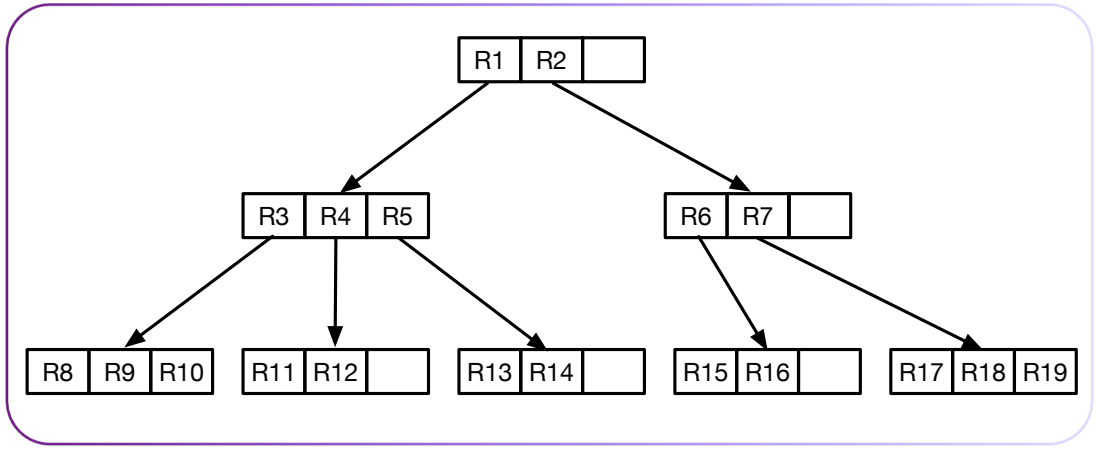
KD树

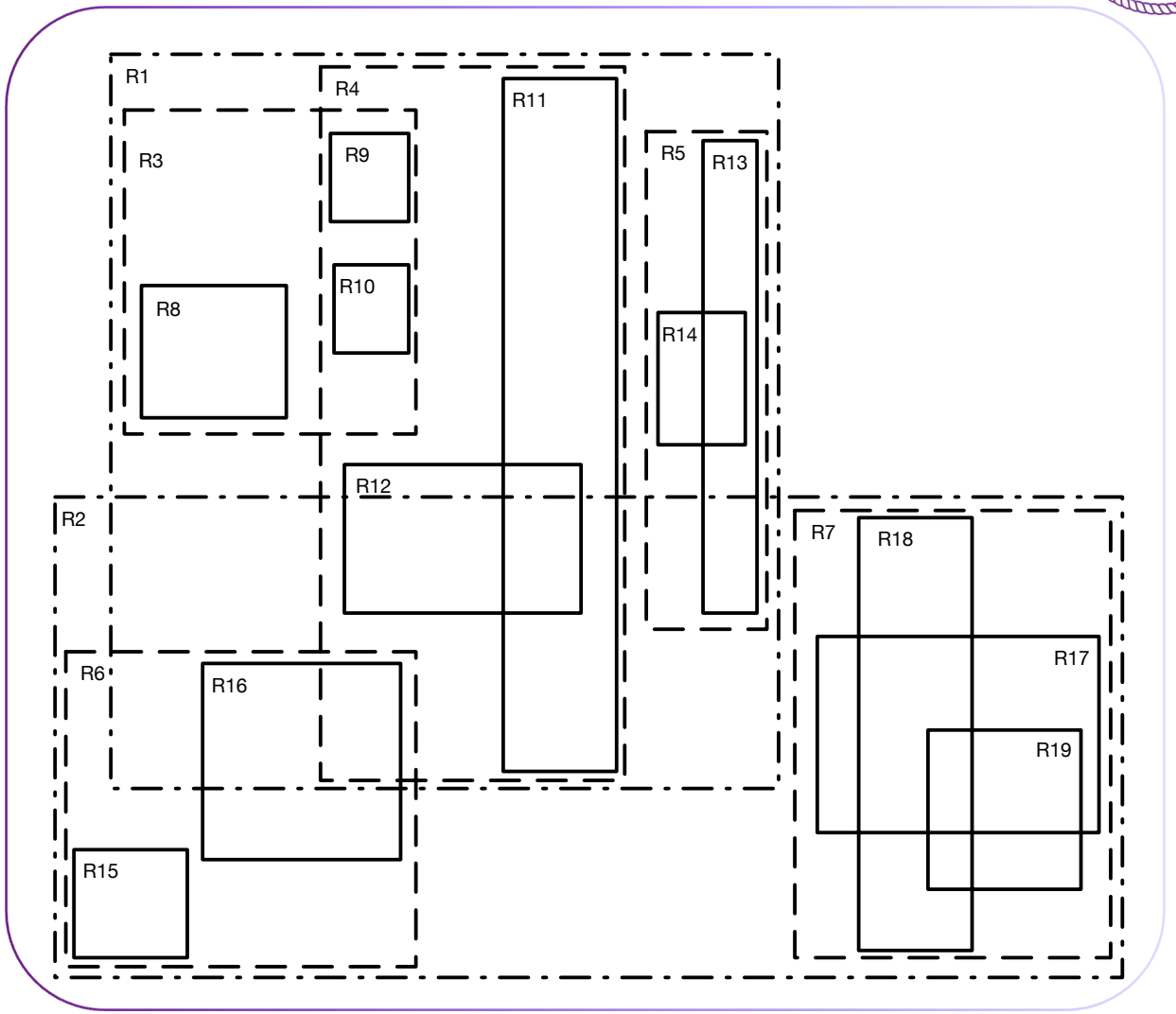
- 四叉树对于分布稀疏的数据会有大量空节点
- KD树每层对一个维度进行分割
  - 针对数据点进行二分，避免了大量空节点



R树

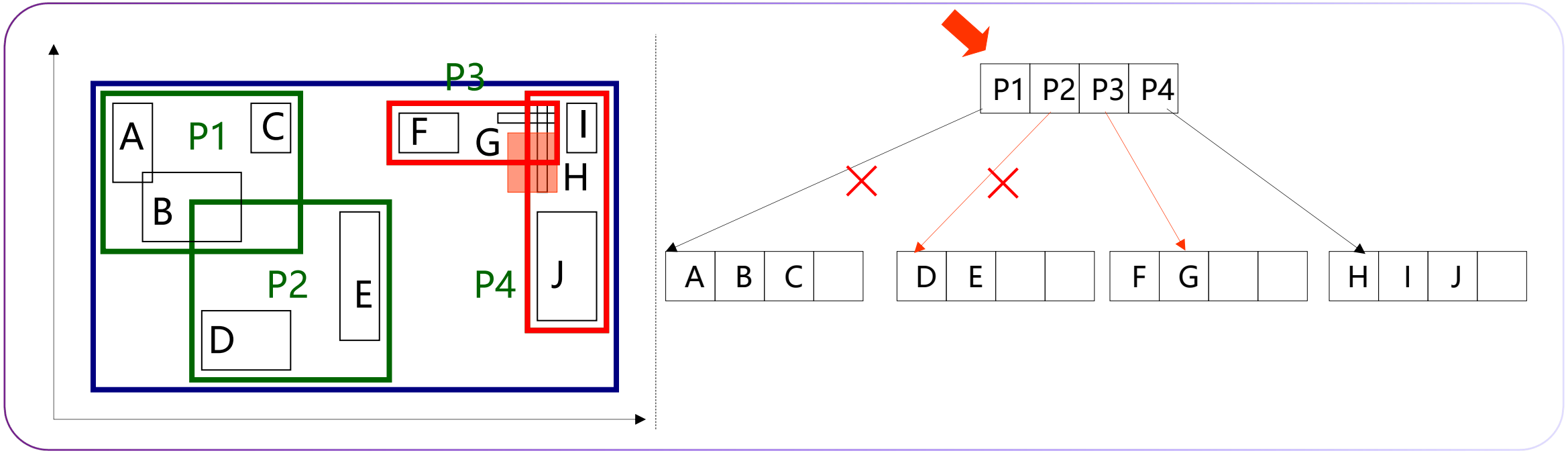
- 空间数据的B+树
- 物体用最小外接矩形表示
- 节点分裂（启发式方法）
  - 贪心，使缩减面积最大





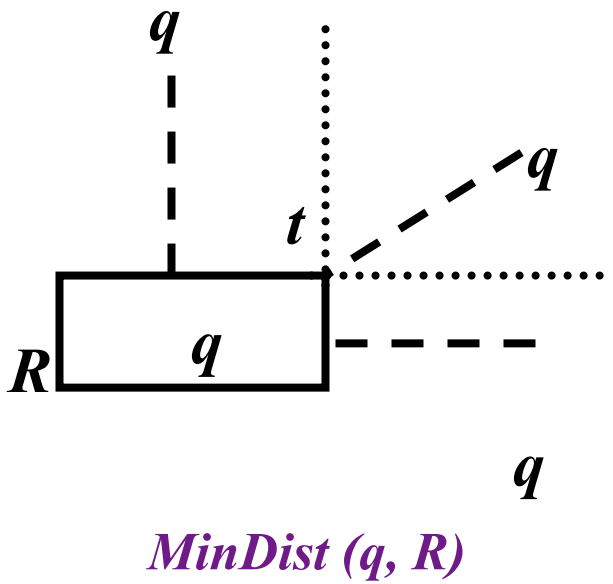
点查、范围查

- 从上往下遍历
- 跳过不相交的节点



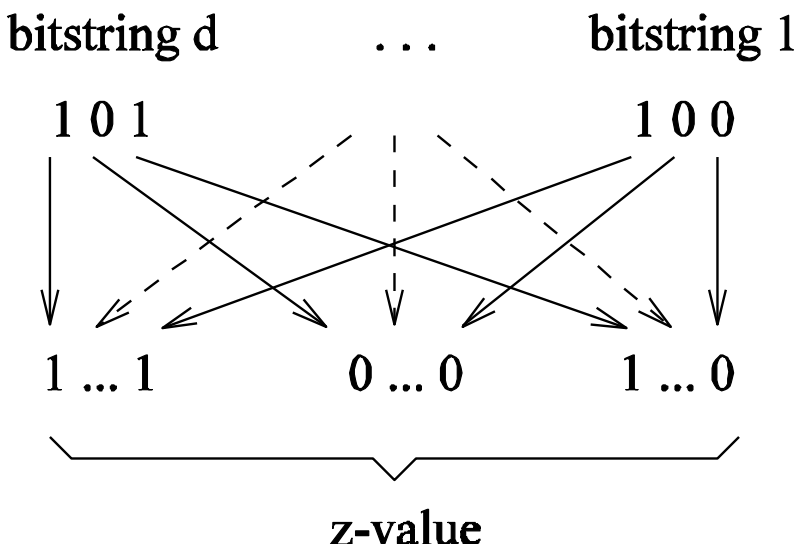
KNN搜索

- **维护一个优先队列Q和结果集A**
- **插入 <root, 0> 到 Q**
- **While Q 不空 并且 |A|<k**
  - 从Q中弹出 （e，MinDist(q, e)）
  - 如果 e 是一个数据对象
    - **将e** 加到结果集 A
  - 如果 e 是一个非叶节点,
    - 对于**e** 的每个孩子C：将<C, MinDist(q, C)> 加入到 Q
  - 如果 e 是一个叶子节点
    - 对于**e** 的每个数据对象, 将 <O, Dist(q, O)> 加入到 Q



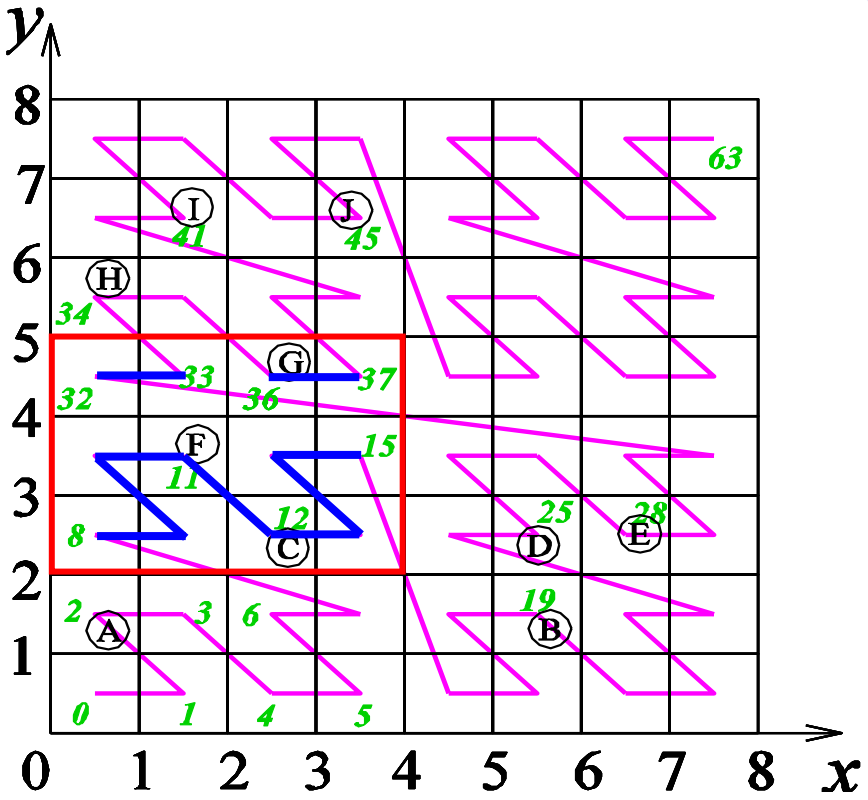
Z曲线

- 高维→低维
- 范围查询→多个范围查询



二维查询: [0,4][2,5]

一维查询: [8,15], [32,33] and [36,37]

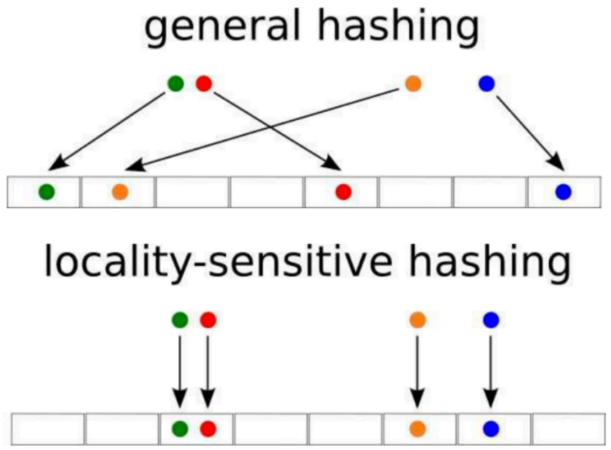


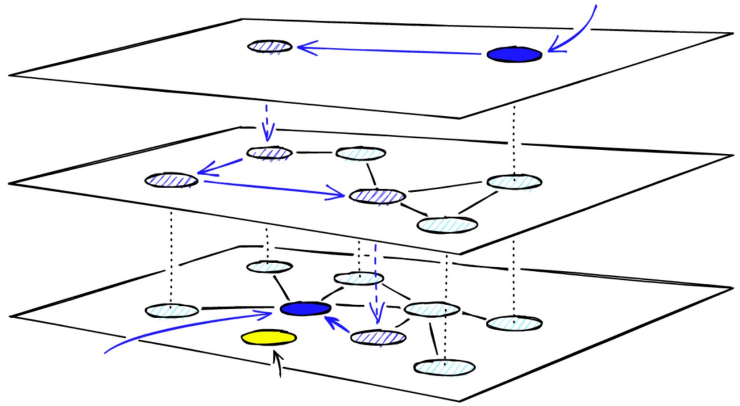
多维索引的对比

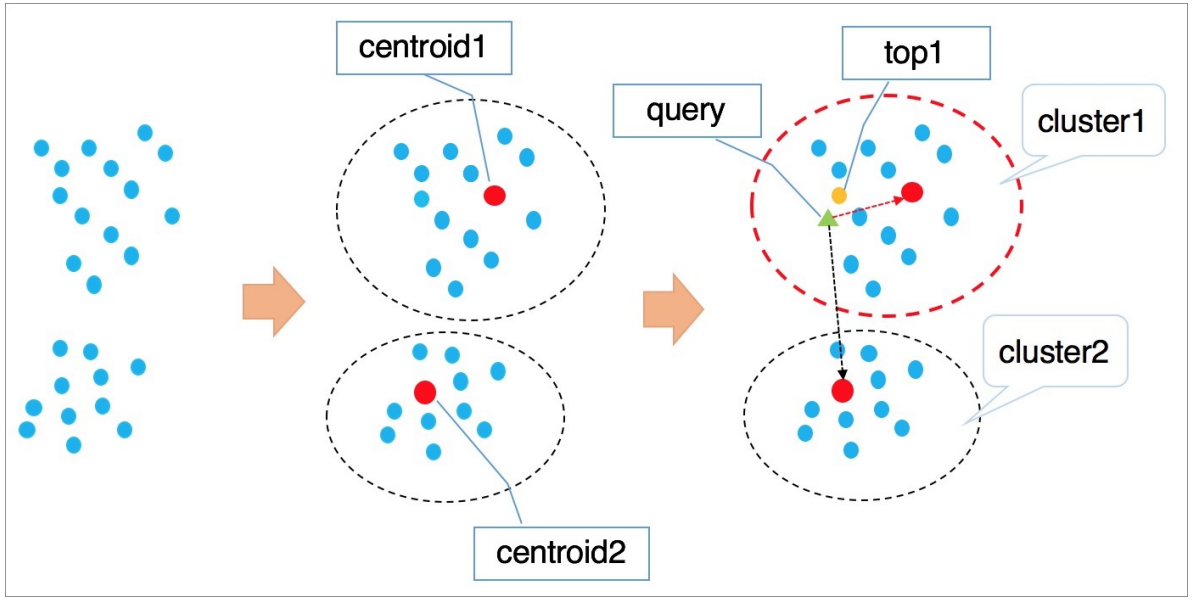
索引	磁盘/内存	平衡	支持的查询类型	维度
R-tree	磁盘	是	点查、范围查, kNN	低
K-d-tree	内存	否	点查、范围查, kNN	低
Quad-tree	内存	否	点查、范围查, kNN	低
Z-curve + B^+-tree	磁盘	是	点查、范围查	低

高维索引

- 局部敏感哈希 Locality-sensitive hashing(LSH)
- M-tree
  - K-means层次聚类
- HNSW (Hierarchical Navigable Small World graphs)
- NSG(Navigating Spreading-out Graph)

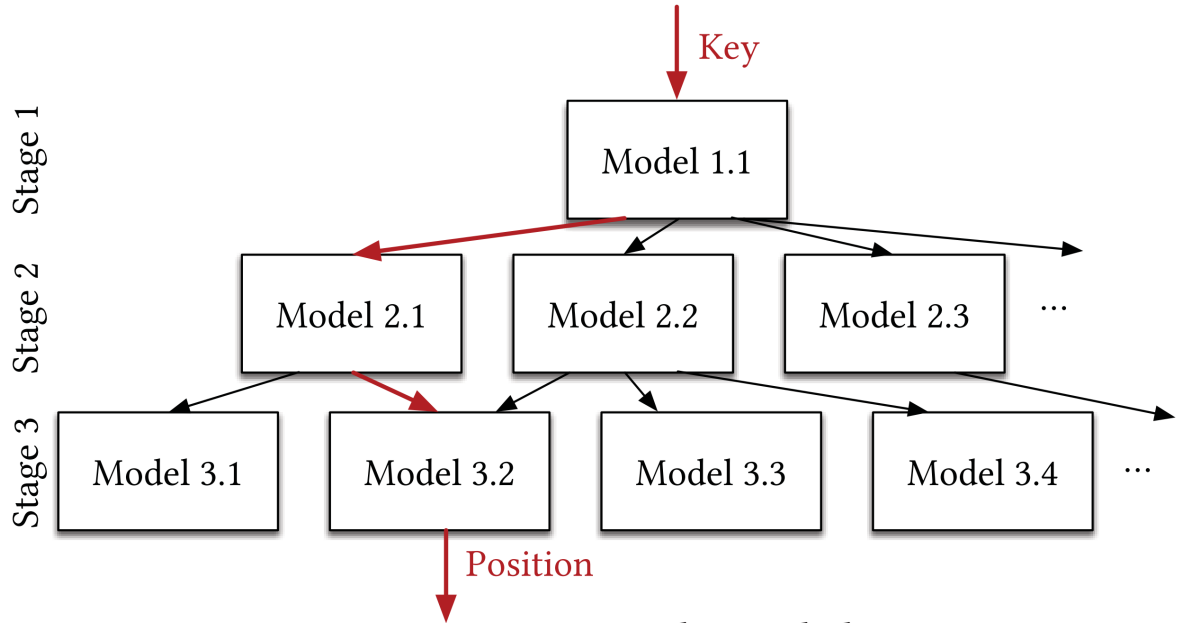


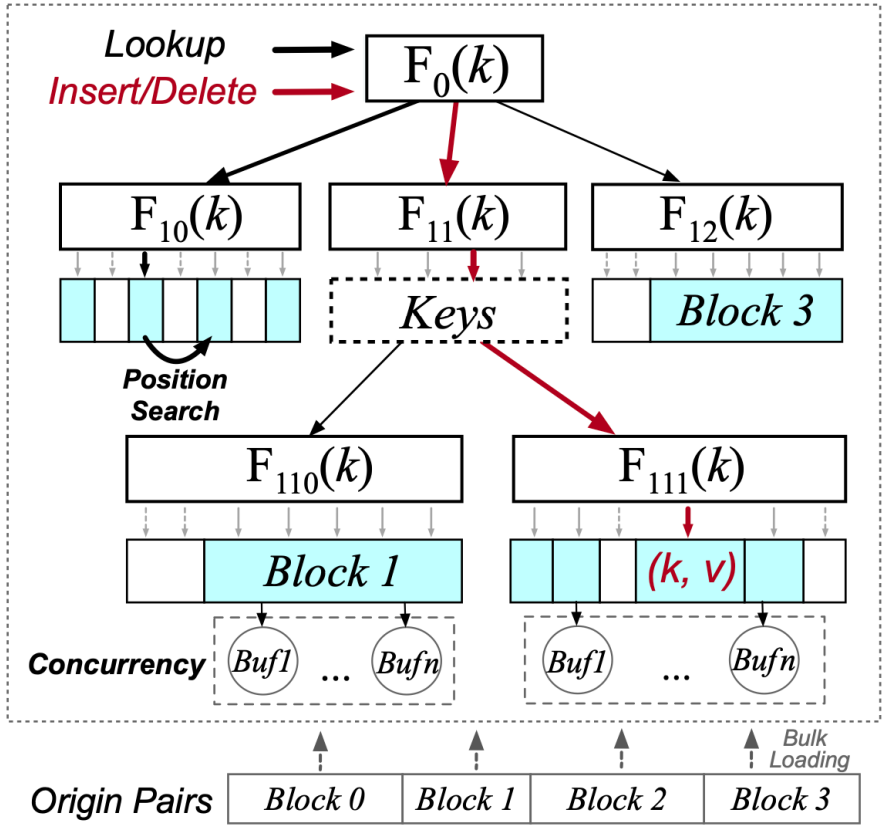




学习型索引

- 预测key所在的页面位置





小结

1. 索引概述
2. B+树索引
3. B+树并发控制
4. 哈希索引
5. SkipList跳表索引
6. LSM树索引
7. 位图索引
8. 多维索引
   - R树、KD树、Quadtree、Z曲线