

第5章：索引结构

Index Structures

邹兆年

哈尔滨工业大学
计算机科学与技术学院
海量数据计算研究中心
电子邮件: znzou@hit.edu.cn

2023年秋

教学内容¹

- ① Indexes
- ② Hash-based Index Structures
 - Extensible Hash Tables
 - Linear Hash Tables
- ③ Tree-based Index Structures
 - B+ Trees
- ④ Log-Structured Merge-Trees (LSM-Trees)
- ⑤ Advanced Topics in Indexing

¹更新于2023年10月29日

Indexes

索引(Index)

索引能够帮助DBMS快速找到关系中满足搜索条件的元组

- 索引对于提高查询处理效率至关重要 [▶ 演示](#)

Example (索引)

索引		Student关系					
Sname	元组地址	地址	Sno	Sname	Ssex	Sage	Sdept
Abby	addr ₃	addr ₁	CS-001	Elsa	F	19	CS
Ed	addr ₂	addr ₂	CS-002	Ed	M	19	CS
Elsa	addr ₁	addr ₃	MA-001	Abby	F	18	Math
Nick	addr ₄	addr ₄	PH-001	Nick	M	20	Physics

查询: `SELECT Sdept FROM Student WHERE Sname = 'Elsa';`

- 如果没有索引, 则只能通过扫描Student关系来完成查询
- 如果有上述索引, 则可以通过该索引来快速找到元组

索引的分类

按照索引的实现方式，可将索引分为两类

- 有序索引(ordered index): 通过按索引键有序排列索引项来实现索引
- 哈希索引(hash index): 通过按索引键哈希值分桶来实现索引

有序索引(Ordered Index)

索引键(index key): 索引根据一组属性(索引键)来定位元组

- 索引记录了元组的索引键值与元组地址的对应关系

索引项(index entry): 索引中的(键值, 地址)对

- 有序索引中的索引项按索引键值排序

Example (有序索引)

有序索引		地址	Student关系				
Sname	元组地址		Sno	Sname	Ssex	Sage	Sdept
Abby	addr ₃	addr ₁	CS-001	Elsa	F	19	CS
Ed	addr ₂	addr ₂	CS-002	Ed	M	19	CS
Elsa	addr ₁	addr ₃	MA-001	Abby	F	18	Math
Nick	addr ₄	addr ₄	PH-001	Nick	M	20	Physics

哈希索引(Hash Index)

哈希索引由若干桶(bucket)构成

- h : 哈希函数
- 键为 K 的索引项属于编号为 $h(K)$ 的桶
- 哈希索引只支持索引键上的等值查找

Example (哈希索引)

哈希索引

桶0

$h(\text{Sdept})$	地址
$h('Math') = 0$	$addr_3$

桶1

$h(\text{Sdept})$	地址
$h('CS') = 1$	$addr_1$
$h('CS') = 1$	$addr_2$

桶2

$h(\text{Sdept})$	地址
$h('Physics') = 2$	$addr_4$

地址

$addr_1$

$addr_2$

$addr_3$

$addr_4$

Student关系

Sno	Sname	Ssex	Sage	Sdept
CS-001	Elsa	F	19	CS
CS-002	Ed	M	19	CS
MA-001	Abby	F	18	Math
PH-001	Nick	M	20	Physics

有序索引的分类(一)

根据数据文件中的元组是否按索引键排序, 可将有序索引可分为两类

- 聚簇索引(clustered index)
- 非聚簇索引(nonclustered index)

聚簇索引(Clustered Index)

如果数据文件中的元组是按索引键排序的，则索引是聚簇索引 ▶ 演示

- 聚簇索引的索引键通常是关系的**主键**
- 一个关系上通常**只有一个**聚簇索引(为什么?)

Example (聚簇索引)

聚簇索引		地址	Student关系				
Sno	元组地址		Sno	Sname	Ssex	Sage	Sdept
CS-001	addr ₁	addr ₁	CS-001	Elsa	F	19	CS
CS-002	addr ₂	addr ₂	CS-002	Ed	M	19	CS
MA-001	addr ₃	addr ₃	MA-001	Abby	F	18	Math
PH-001	addr ₄	addr ₄	PH-001	Nick	M	20	Physics

◀ ◻ ▶ ◀ ◻ ▶ ◀ ≡ ▶ ◀ ≡ ▶ ≡ 🔍 ↺

非聚簇索引(Nonclustered Index)

如果数据文件中的元组不是按索引键排序的，则索引是非聚簇索引

- 一个关系上可以有**多个**非聚簇索引

Example (非聚簇索引)

非聚簇索引		地址	Student关系				
Sname	元组地址		Sno	Sname	Ssex	Sage	Sdept
Abby	addr ₃	addr ₁	CS-001	Elsa	F	19	CS
Ed	addr ₂	addr ₂	CS-002	Ed	M	19	CS
Elsa	addr ₁	addr ₃	MA-001	Abby	F	18	Math
Nick	addr ₄	addr ₄	PH-001	Nick	M	20	Physics

◀ ◻ ▶ ◀ ◻ ▶ ◀ ≡ ▶ ◀ ≡ ▶ ≡ 🔍 ↺

索引组织表(Index-Organized Table)

索引组织表 = 聚簇索引文件 + 数据文件

- 在聚簇索引的索引项中存储**元组本身**，而不是元组地址
- 无需根据元组地址从磁盘读元组，减少1次I/O

Example (索引组织表)

索引组织表

Sno	Sname	Ssex	Sage	Sdept
CS-001	Elsa	F	19	CS
CS-002	Ed	M	19	CS
MA-001	Abby	F	18	Math
PH-001	Nick	M	20	Physics

有序索引的分类(二)

根据关系中每个元组在索引中是否都有一个对应索引项，可将有序索引分为两类

- 稠密索引(dense index)
- 稀疏索引(sparse index)

稠密索引(Dense Index)

如果关系中每个元组在索引中都有一个对应索引项，则索引是稠密索引

- 非聚簇索引一定是稠密索引

Example (稠密索引)

非聚簇索引			Student关系				
Sname	元组地址	地址	Sno	Sname	Ssex	Sage	Sdept
Abby	addr ₃	addr ₁	CS-001	Elsa	F	19	CS
Ed	addr ₂	addr ₂	CS-002	Ed	M	19	CS
Elsa	addr ₁	addr ₃	MA-001	Abby	F	18	Math
Nick	addr ₄	addr ₄	PH-001	Nick	M	20	Physics

稀疏索引(Sparse Index)

如果关系中只有部分元组在索引中有对应索引项，则索引是稀疏索引

- 聚簇索引通常是稀疏索引
- 可以只对数据文件每页中的第一个元组建立索引项
- 可以只对数据文件每个不同的索引键值的第一个元组建立索引项

Example (稀疏索引)

聚簇索引		Page ₁ addr ₁ addr ₂	Student关系				
Sno	元组地址		Sno	Sname	Ssex	Sage	Sdept
CS-001	addr ₁	Page ₂ addr ₃ addr ₄	CS-001	Elsa	F	19	CS
MA-001	addr ₃		CS-002	Ed	M	19	CS
			Sno	Sname	Ssex	Sage	Sdept
			MA-001	Abby	F	18	Math
			PH-001	Nick	M	20	Physics

有序索引的分类(三)

根据索引键是否为关系的主键，可将有序索引可分为两类

- 主索引(primary index)
- 二级索引(secondary index)

主索引(Primary Index)

主索引的索引键是关系的主键

- 一个关系 **只有一个主索引**

Example (主索引)

主索引		地址	Student 关系				
Sno	元组地址		Sno	Sname	Ssex	Sage	Sdept
CS-001	addr ₁	addr ₁	CS-001	Elsa	F	19	CS
CS-002	addr ₂	addr ₂	CS-002	Ed	M	19	CS
MA-001	addr ₃	addr ₃	MA-001	Abby	F	18	Math
PH-001	addr ₄	addr ₄	PH-001	Nick	M	20	Physics

二级索引(Secondary Index)

二级索引的索引键不是关系的主键

- 二级索引通常是**非聚簇索引**
- 一个关系可以有**多个二级索引**

Example (二级索引)

二级索引		地址	Student关系				
Sname	元组地址		Sno	Sname	Ssex	Sage	Sdept
Abby	addr ₃	addr ₁	CS-001	Elsa	F	19	CS
Ed	addr ₂	addr ₂	CS-002	Ed	M	19	CS
Elsa	addr ₁	addr ₃	MA-001	Abby	F	18	Math
Nick	addr ₄	addr ₄	PH-001	Nick	M	20	Physics

创建主索引

- 在CREATE TABLE或ALTER TABLE语句中使用**PRIMARY KEY**声明主键时, **自动建立主索引**
- 只能在CREATE TABLE或ALTER TABLE语句中声明主索引

Example (创建主索引)

```
CREATE TABLE Student (  
    Sno CHAR(6),  
    Sname VARCHAR(10),  
    Ssex CHAR,  
    Sage INT,  
    Sdept VARCHAR(20),  
    PRIMARIY KEY (Sno));
```

创建二级索引

语句: **CREATE INDEX 索引名 ON 关系名(索引键)**

- 用**ASC**或**DESC**声明索引属性的排序方式

Example (创建二级索引)

```
CREATE INDEX idx_sname_sage ON Student (Sname, Sage DESC);
```

MySQL中的索引²

- 主索引是**索引组织表**
- 二级索引的索引项中存储的不是元组地址, 而是元组的**主键值**

Example (MySQL中的主索引和二级索引)

MySQL二级索引

Sname	Sno
Abby	MA-001
Ed	CS-002
Elsa	CS-001
Nick	PH-001

MySQL主索引/索引组织表

Sno	Sname	Ssex	Sage	Sdept
CS-001	Elsa	F	19	CS
CS-002	Ed	M	19	CS
MA-001	Abby	F	18	Math
PH-001	Nick	M	20	Physics

思考题

分析MySQL索引设计的优缺点

²MySQL InnoDB存储引擎

唯一索引 (Unique Index)

唯一索引(unique index)的索引键值不能重复

- 主索引一定是唯一索引
- 二级索引不一定是唯一索引

Example (唯一索引vs 非唯一索引)

唯一索引			Student关系				
Sno	元组地址	地址	Sno	Sname	Ssex	Sage	Sdept
CS-001	addr ₂	addr ₁	CS-001	Elsa	F	19	CS
CS-002	addr ₃	addr ₂	CS-002	Ed	M	19	CS
MA-001	addr ₄	addr ₃	MA-001	Abby	F	18	Math
PH-001	addr ₁	addr ₄	PH-001	Nick	M	20	Physics

创建唯一索引

创建唯一索引有两种方法

- 在CREATE TABLE或ALTER TABLE语句中使用**UNIQUE**声明唯一约束时, 自动创建唯一索引
- 使用语句: **CREATE UNIQUE INDEX 索引名 ON 关系名(索引键)**

Example (创建唯一索引)

```
CREATE TABLE Student (  
  Sno CHAR(6) PRIMARY KEY,  
  Sname VARCHAR(10),  
  Ssex CHAR,  
  Sage INT,  
  Sdept VARCHAR(20),  
  UNIQUE (Sname));
```

```
CREATE UNIQUE INDEX ukey_sname ON Student(Sname);
```

外键索引(Foreign Key Index)

外键索引的索引键是关系的外键

- 当被参照关系的元组被删除时，外键索引可以加快参照完整性检查
- 当被参照关系的元组的主键值被修改时，外键索引可以加快参照完整性检查
- ON DELETE|UPDATE [NO ACTION|RESTRICT|CASCADE|SET NULL|DEFAULT]

Example (外键索引)

外键索引		地址	SC		
Sno	元组地址		Sno	Cno	Grade
CS-001	addr ₄	addr ₁	PH-001	1002	92
CS-001	addr ₅	addr ₂	PH-001	2003	85
CS-002	addr ₆	addr ₃	PH-001	3006	88
MA-001	addr ₇	addr ₄	CS-001	1002	95
PH-001	addr ₁	addr ₅	CS-001	3006	90
PH-001	addr ₂	addr ₆	CS-002	3006	80
PH-001	addr ₃	addr ₇	MA-001	1002	

创建外键索引

- 在CREATE TABLE或ALTER TABLE语句中使用**FOREIGN KEY**声明外键时，会为外键创建索引

Example (创建外键索引)

```
CREATE TABLE SC (  
    Sno CHAR(6),  
    Cno CHAR(4),  
    Grade INT,  
    PRIMARY KEY (Sno, Cno),  
    FOREIGN KEY (Sno) REFERENCES Student);
```

删除索引

删除二级索引

- PostgreSQL语句: **DROP INDEX 索引名;**
- MySQL语句: **DROP INDEX 索引名 ON 关系名;**
- 删除二级索引后不需要重新组织关系中的元组

删除主索引

- PostgreSQL中不能直接删除主索引, 只能删除主键约束
- MySQL语句: **DROP INDEX 'PRIMARY' ON 关系名;**
- 删除主索引后需要重新组织关系中元组

索引结构(Index Structures)

有序索引的数据结构

- 平衡树
- 跳表(skiplist): 多用于内存数据库系统
- 字典树(trie): 多用于内存数据库系统
- 日志结构合并树(log-structured merge-tree, LSM-tree): 多用于NoSQL数据库系统的存储引擎

哈希索引的数据结构

- 哈希表

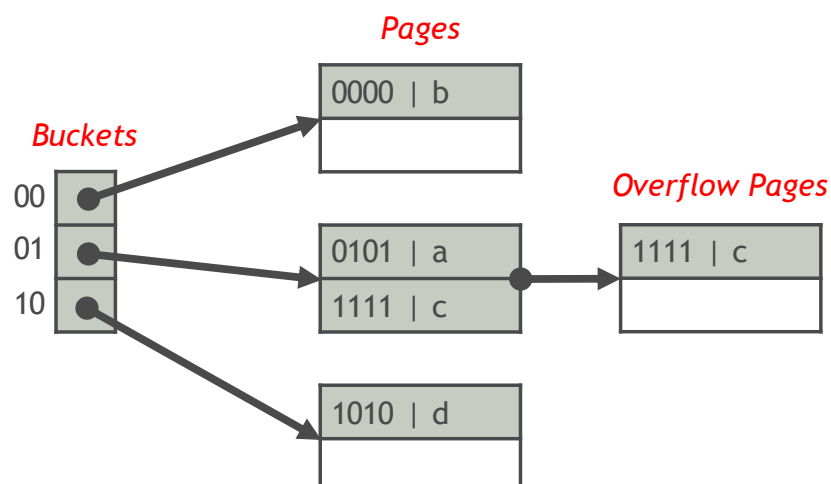
不同的索引结构具有不同的功能和性能

Hash-based Index Structures

外存哈希表(Secondary-Storage Hash Tables)

一个外存哈希表包含多个桶(bucket)

- 设 $hash$ 是一个哈希函数, 键为 K 的索引项(index entry)属于编号为 $hash(K)$ 的桶
- 每个桶中存放一个指针, 指向存储该桶中索引项的页的链表



外存哈希表的分类

静态哈希表(Static Hash Tables)

- 桶的数量固定不变

动态哈希表(Dynamic Hash Tables)

- 桶的数量动态变化，使每个桶中的索引项存储在大约1个页中
- 可扩展哈希表(extensible hash tables)
- 线性哈希表(linear hash tables)

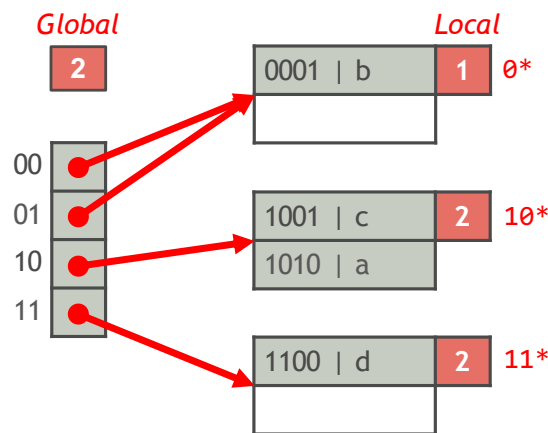
Hash-based Index Structures Extensible Hash Tables

可扩展哈希表(Extensible Hash Tables)

一个可扩展哈希表包含 2^i 个桶

- i : 全局深度(global depth)
- 键值为 K 的索引项属于编号等于 $hash(K)$ 的前 i 位的桶

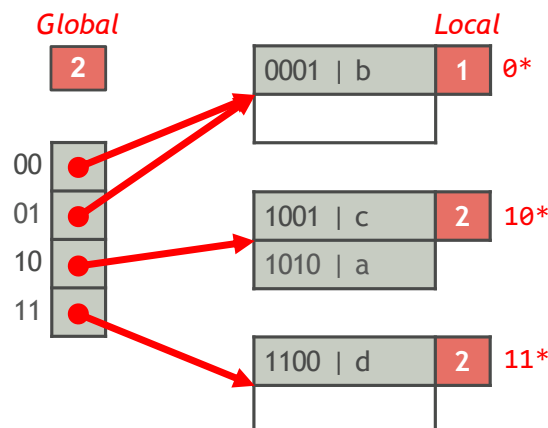
例: $hash(a) = 1010$, $hash(b) = 0001$, $hash(c) = 1001$, $hash(d) = 1100$



可扩展哈希表(续)

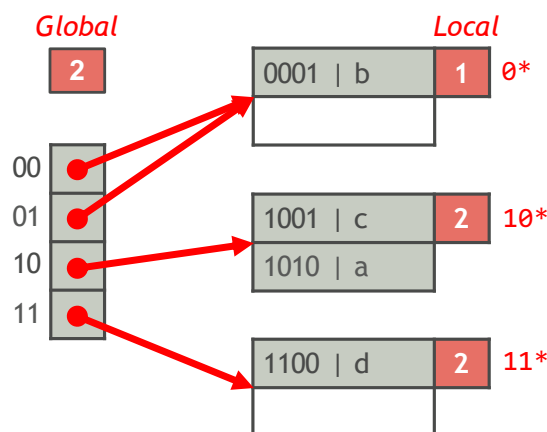
每个桶中存放一个指针，指向存储该桶中索引项的页

- 每个桶均没有溢出页(overflow page)
- 如果容纳得下，多个相邻桶中的全部索引项可以存入同一个页
- 每个页记录一个局部深度(local depth) j ，该页中的全部索引项的 $hash(K)$ 的前 j 位相同，用于标识这些索引项都存于这个页



可扩展哈希表的性质

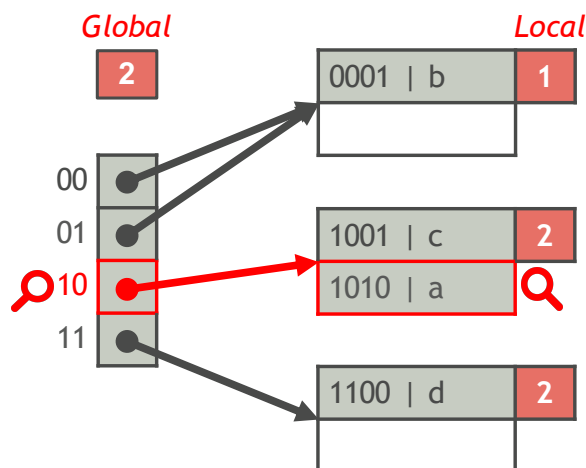
- 桶数 = $2^{\text{global_depth}}$
- 全局深度 \geq 每个页的局部深度
- 一个页被多个桶共享当且仅当这个页的局部深度小于全局深度
- 设一个页的局部深度为 j ，则页中全部索引项的 $\text{hash}(K)$ 的前 j 位相同



查找索引项

- 确定索引项所属的桶
- 在桶指向的页中查找索引项

例: $K = a$, $\text{hash}(a) = 1010$

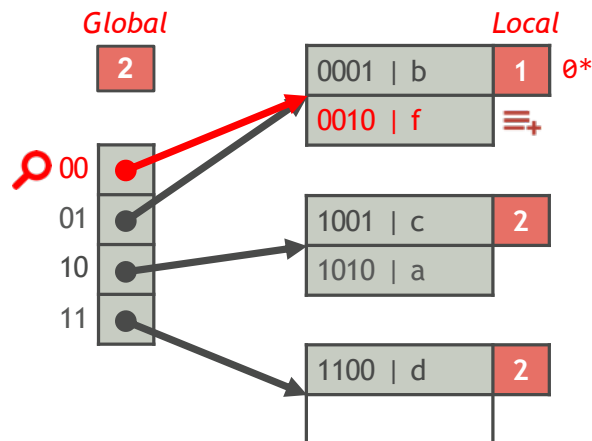


插入索引项

① 找到索引项被插入的页 P

② 如果 P 中有足够的空闲空间，则将索引项插入 P 中；否则，分裂 P

例: $K = f$, $\text{hash}(f) = 0010$



插入索引项(续)

如果 P 溢出且 P 的局部深度小于全局深度

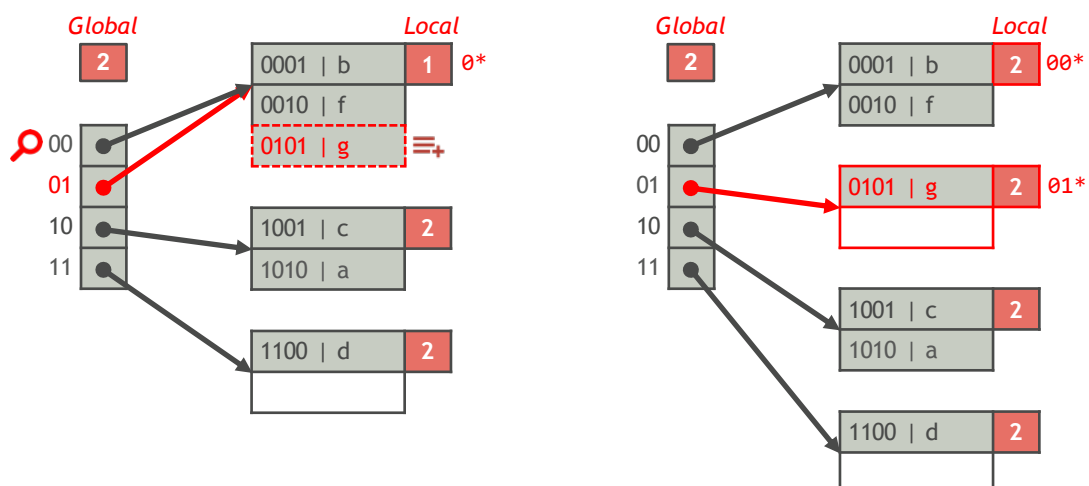
① 将 P 的局部深度 j 加1

② 创建一个新页 P' ，令 P 和 P' 的局部深度相同

③ 根据键的哈希值的前 j 位，将 P 中索引项在 P 和 P' 中重新分配

④ 更新指向 P 的桶中的指针

例: $K = g$, $\text{hash}(g) = 0101$

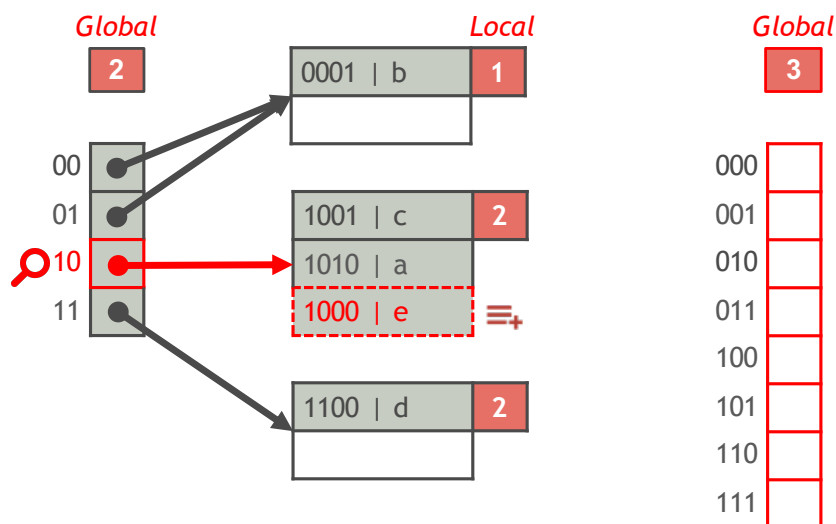


插入索引项(续)

如果 P 溢出且 P 的局部深度等于全局深度

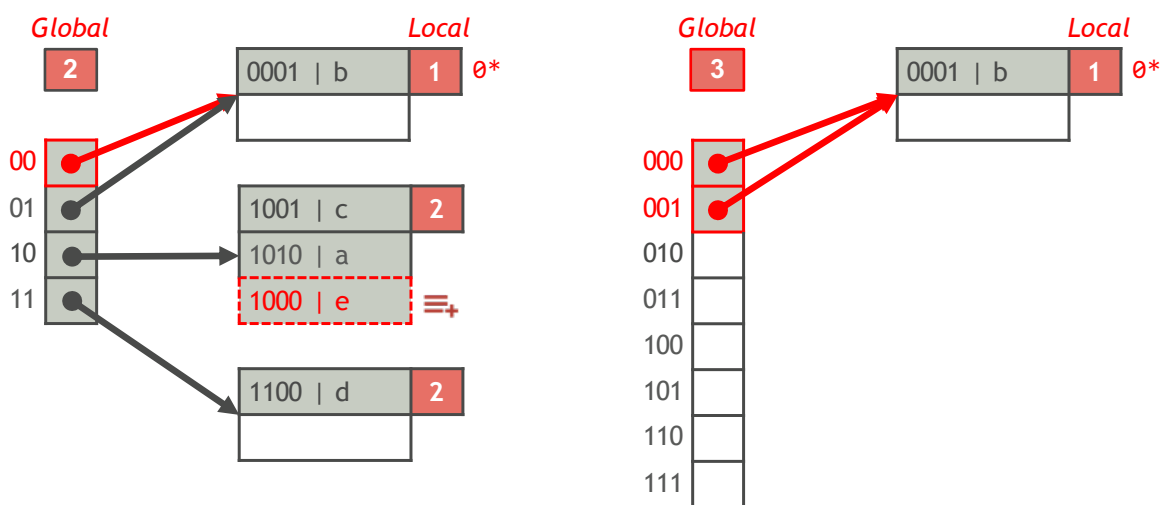
- ① 将全局深度加1，即桶的数量翻倍
- ② 更新每个桶中的页指针
- ③ 对于 P ，使用前面介绍的方法分裂 P

例: $K = e$, $\text{hash}(e) = 1000$



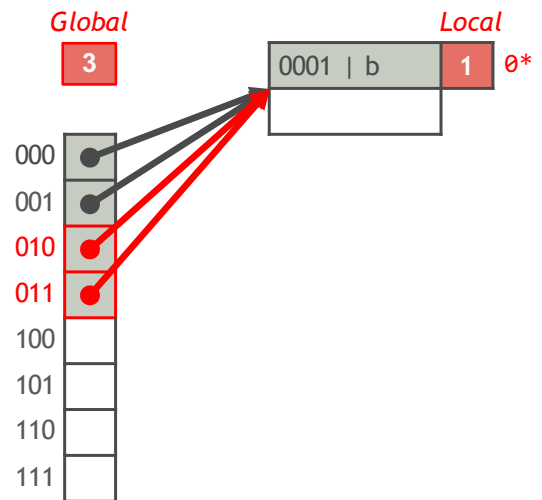
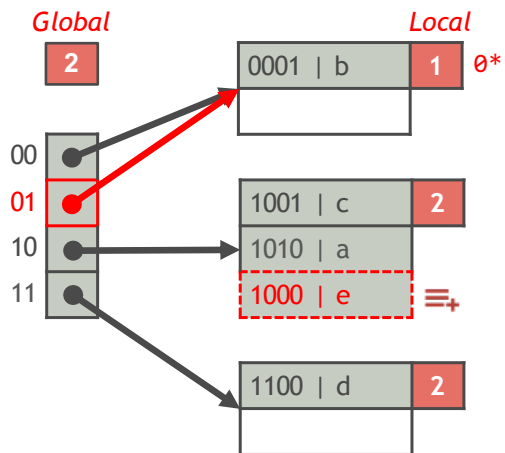
步骤1/5

插入索引项(续)



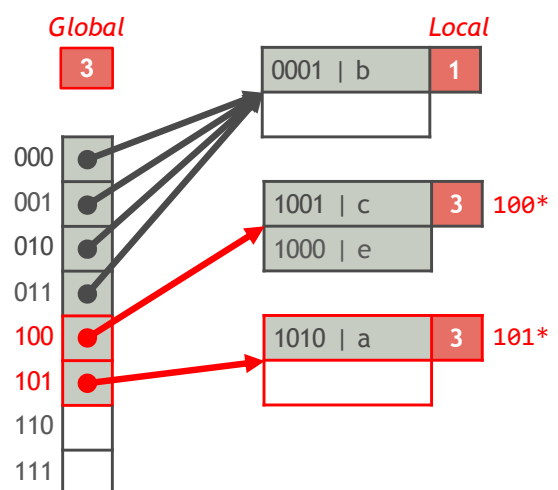
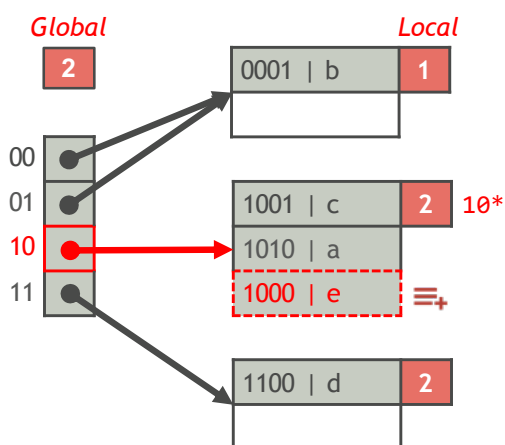
步骤2/5

插入索引项(续)



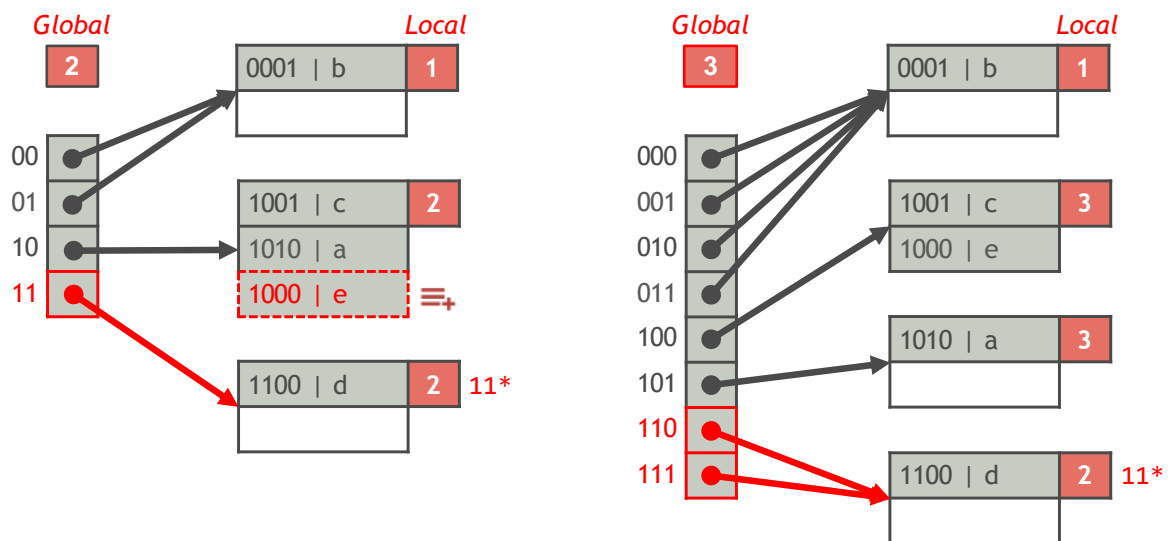
步骤3/5

插入索引项(续)



步骤4/5

插入索引项(续)



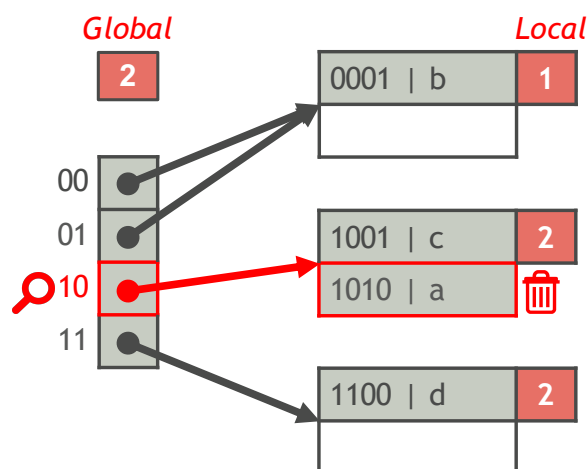
步骤5/5

删除索引项

① 找到索引项所在的页

② 从页中删除索引项

例: $K = a$, $hash(a) = 1010$



思考题

删除索引项后，是否需要合并页？

Hash-based Index Structures

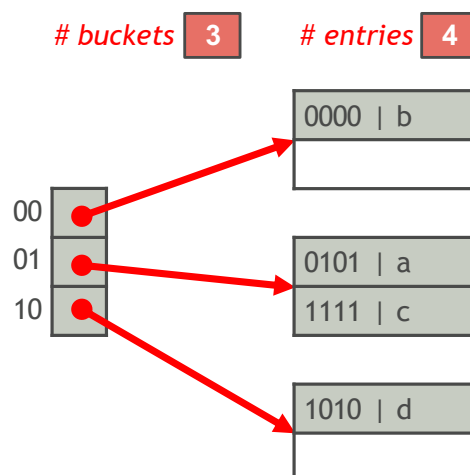
Linear Hash Tables

线性哈希表(Linear Hash Tables)

线性哈希表包含 n 个桶

- 每个桶中保存一个指针，指向存储该桶中索引项的页的链表
- 假设每个页最多存储 b 个索引项，每个桶只有1个页，则线性哈希表中最多存储 θbn 个索引项，其中 $0 < \theta < 1$ 是一个阈值
- 记录线性哈希表中桶的数量(# buckets)和索引项的数量(# entries)

例: $b = 2, \theta = 0.85$



哈希方案(Hashing Scheme)

- 设桶号为 $0, 1, \dots, n-1$
- 令 $m = 2^{\lceil \log_2 n \rceil}$, 因此 $m \leq n < 2m$
- 对于键值为 K 的索引项, 如果 $\text{hash}(K) \bmod 2m < n$, 则该索引项属于编号为 $\text{hash}(K) \bmod 2m$ 的桶; 否则, 该索引项属于编号为 $\text{hash}(K) \bmod m$ 的桶

Example (线性哈希表的哈希方案)

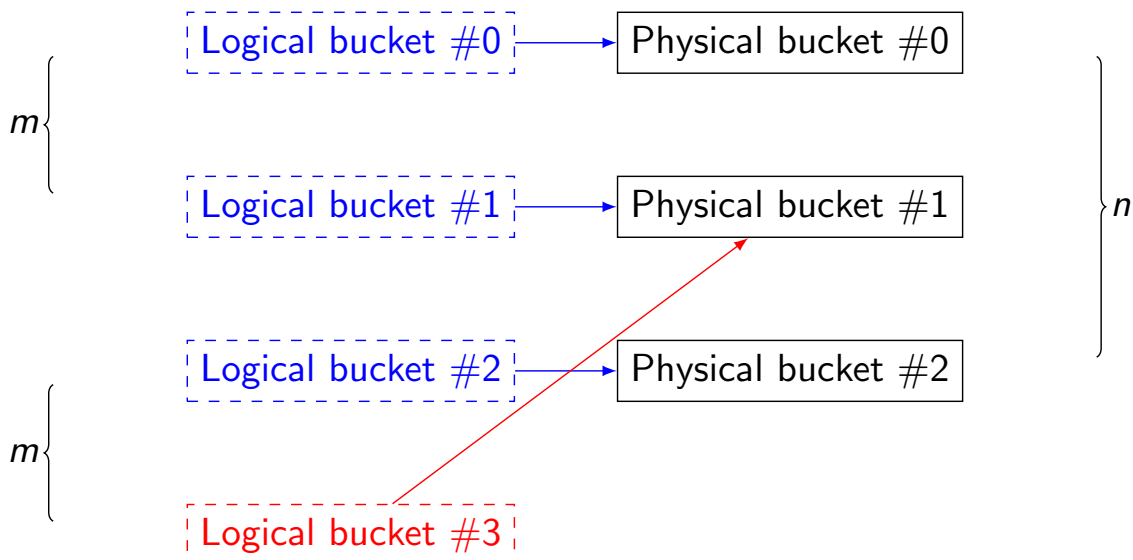
设 $n = 3$, 有 $m = 2$

Bucket #0	$\text{hash}(K) = 0, 4, 8, \dots$
Bucket #1	$\text{hash}(K) = 1, \textcolor{red}{3}, 5, \textcolor{red}{7}, 9, \dots$
Bucket #2	$\text{hash}(K) = 2, 6, 10, \dots$

桶的负载不平衡

哈希方案的解释

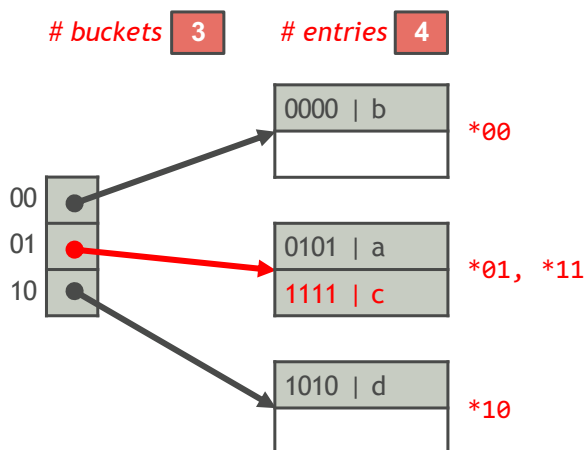
- 逻辑上有 $2m$ 个桶, 物理上有 n 个桶, $n < 2m$
- 键值为 K 的索引项的逻辑桶号 $b(K) = \text{hash}(K) \bmod 2m$
- 如果 $b(K) < n$, 则该索引项属于 $b(K)$ 号物理桶
- 如果 $b(K) \geq n$, $b(K)$ 号物理桶不存在, 则该索引项被放入 $b(K) \bmod m$ 号物理桶



查找索引项

- ① 确定索引项所属的桶
- ② 在桶指向的页链表中查找索引项

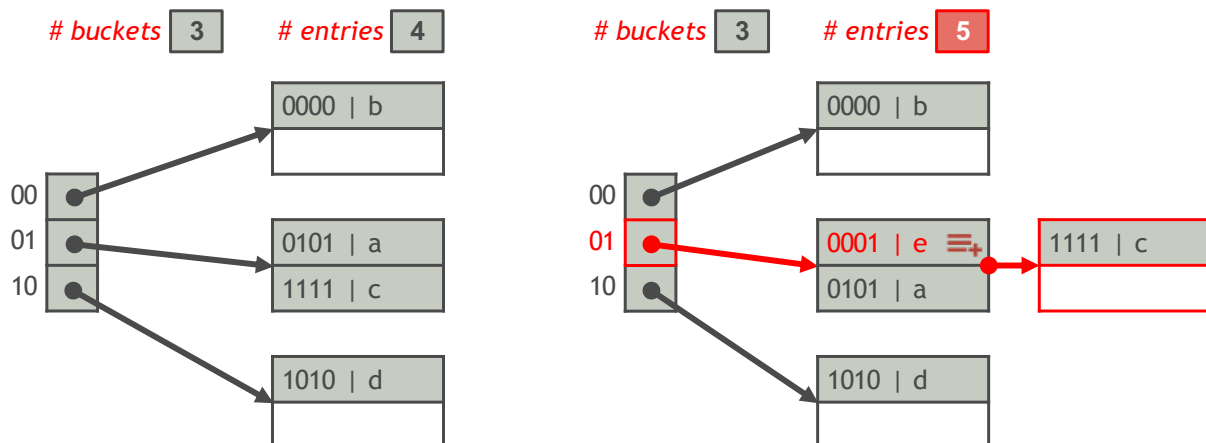
例: $K = c$, $hash(c) = 1111$ 。根据哈希方案，键为 c 的索引项在 01 号桶中



插入索引项

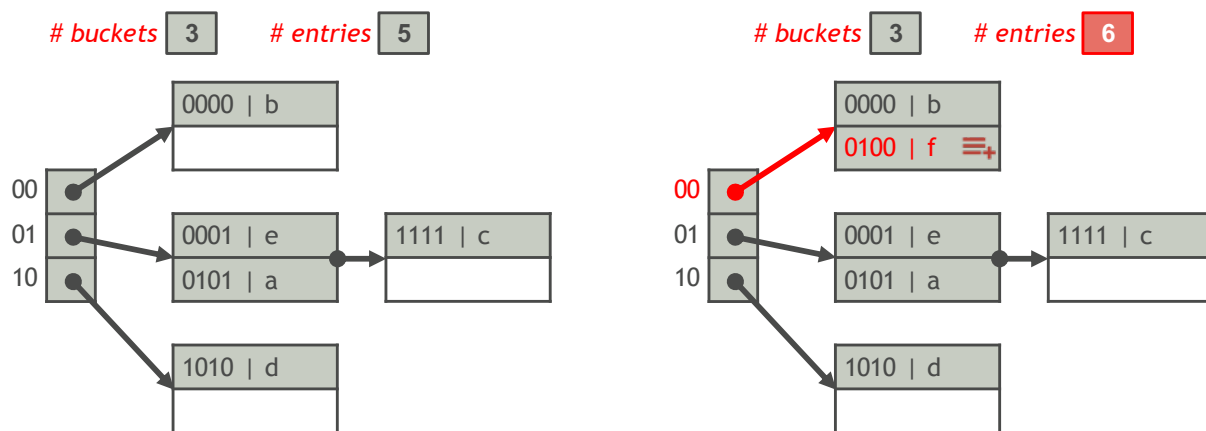
- ① 将索引项插入它所属的桶 B
- ② 将索引项的数量 (# entries) 加 1
- ③ 如果 $\# \text{ entries} \leq \theta bn$ ，则插入完成；否则，将桶的数量 (# buckets) 加 1，按照哈希方案重新分配 B 中的索引项

例1: $hash(e) = 0001$, $\theta = 0.85$



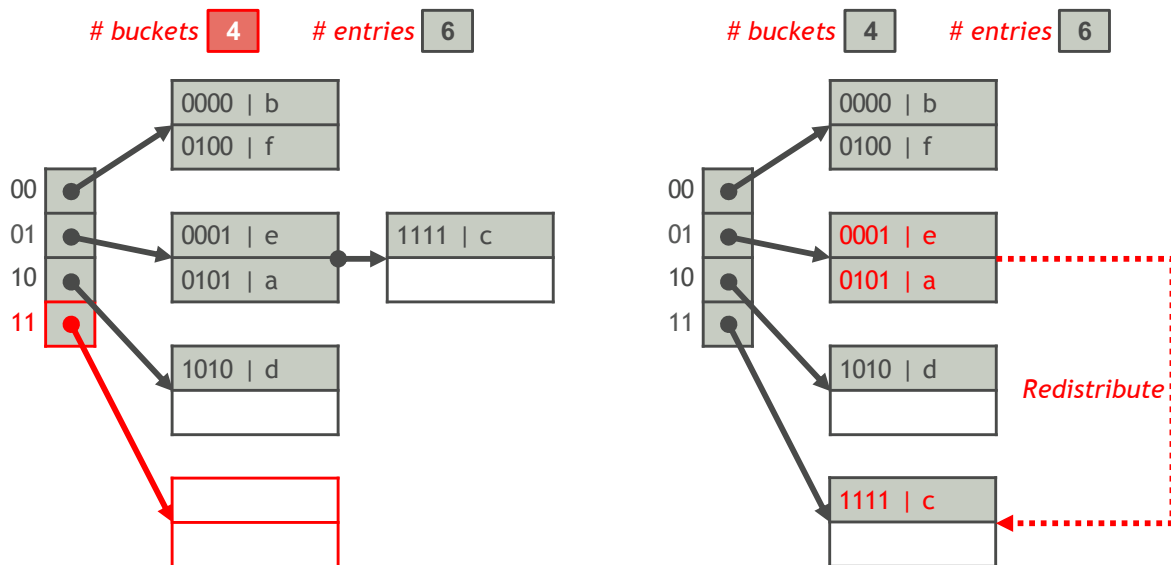
插入索引项(续)

例2: $hash(f) = 0100$, $\theta = 0.85$



步骤1/2

插入索引项(续)

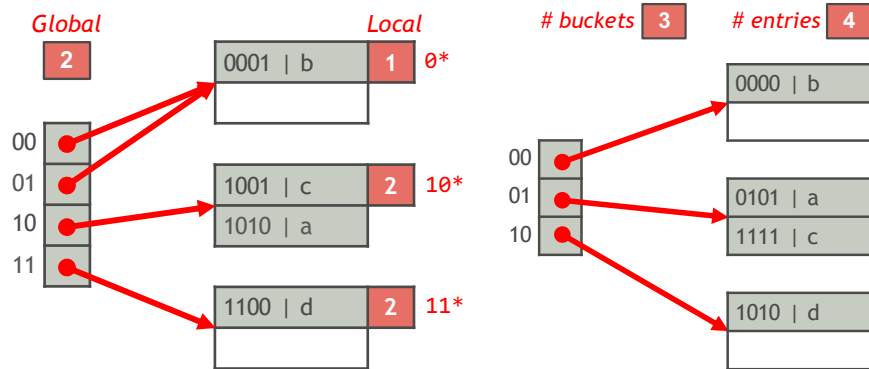


步骤2/2

因为新创建的桶的编号为11，所以只需将原来01号桶中应属于11号桶的索引项重新分配到11号桶中；其他桶中的索引项无需重分配

可扩展哈希表VS 线性哈希表

	可扩展哈希表	线性哈希表
桶的数量	2^{global_depth}	n
是否有溢出页	无	有
哈希方案	$hash(K)$ 的前 $global_depth$ 位	$hash(K) \bmod 2m$ 或 $hash(K) \bmod m$
页分裂条件	页发生溢出	$\#entries > \theta bn$
增加桶的方法	桶数翻倍($global_depth$ 加1)	桶数加1



Tree-based Index Structures

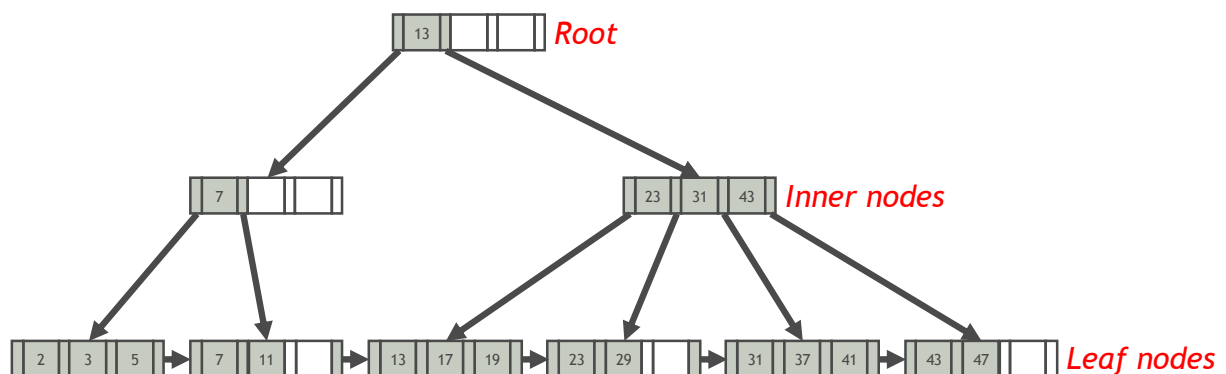
Tree-based Index Structures

B+ Trees

B+树(B+ Trees)

B+树是一棵 M 路平衡搜索树，它具有以下性质：

- B+树是一棵完美的平衡树，所有叶节点都在同一层上
- 除根节点外，每个节点至少“半满”，即 $M/2 - 1 \leq \#keys \leq M - 1$ ³
- 每个节点恰好放入1个页

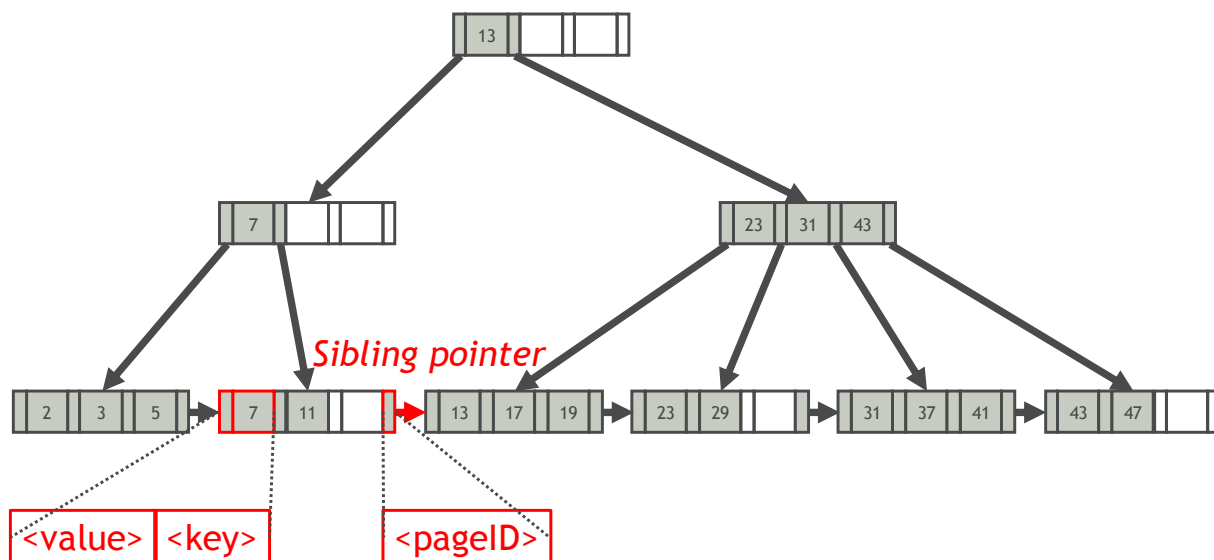


³Raghu Ramakrishnan, Johannes Gehrke. Database Management Systems, 3rd Edition. 2003

B+树的叶节点(Leaf Nodes)

每个叶节点包含一个索引项数组和一个指向右侧兄弟叶节点的指针(右兄弟节点的页号)

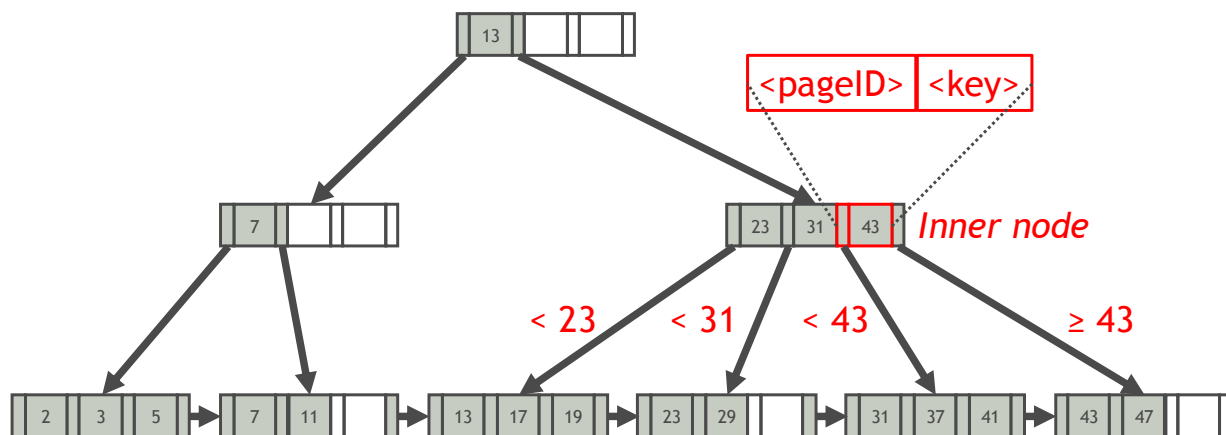
- 索引项数组通常按索引键排序



B+树内节点(Inner Nodes)

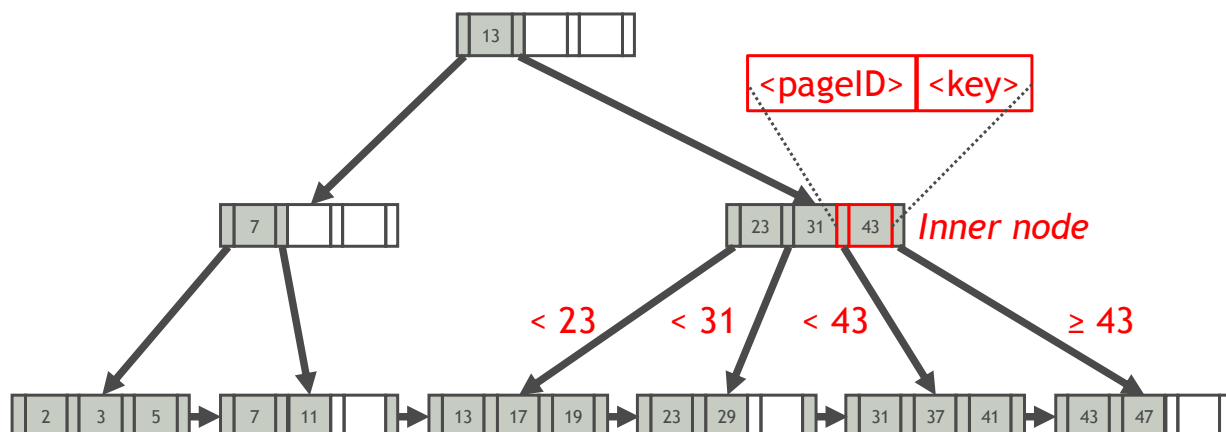
每个内节点包含一个键数组 Key 和一个指向儿子节点的指针的数组 Ptr

- Key 中有 k 的非空键值当且仅当 Ptr 中有 $k + 1$ 个非空指针
- Key 中的键值排序
- $Ptr[0]$ 指向的子树中的键值 $< Key[0]$
- $Ptr[k + 1]$ 指向的子树中的键值 $\geq Key[k]$
- $Key[i - 1] \leq Ptr[i]$ 指向的子树中的键值 $< Key[i]$



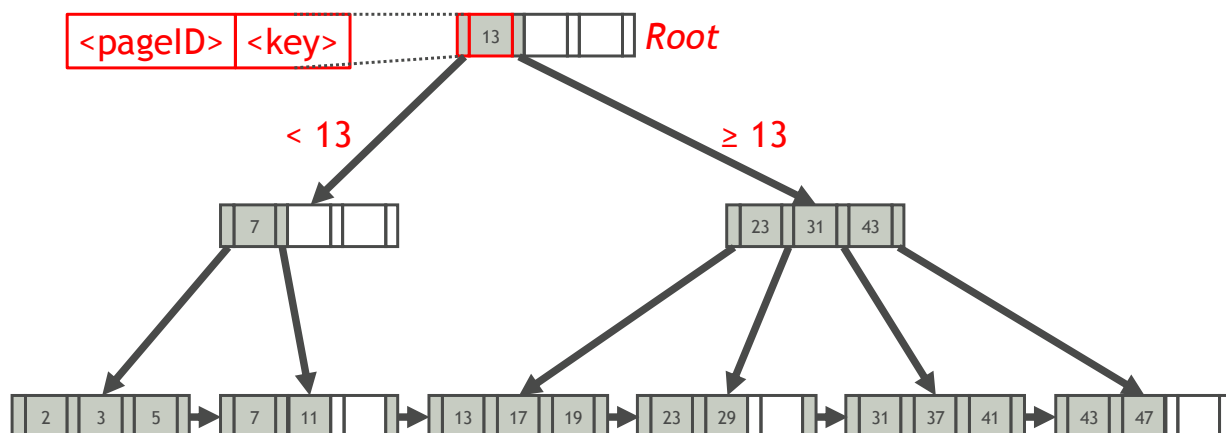
B+树内节点(Inner Nodes)

对于 $i \geq 1$, $Key[i - 1]$ 存放的是 $Ptr[i]$ 指向的子树叶节点中的最小键值



B+树的根节点(Root Node)

根节点和内节点的内部结构相同, 但不要求“半满”(根节点中包含至少1个键即可)



Rucbase中B+树节点的定义ix_node_handle.h

```
class IxNodeHandle {
friend class IxIndexHandle;
friend class IxScan;

private:
const IxFileHdr *file_hdr; // 用到了file_hdr的keys_size, col_
Page *page;

IxPageHdr *page_hdr; // page->data的第一部分, 指针指向首地址,
后续占用长度为sizeof(IxPageHdr)
char *keys; // page->data的第二部分, 指针指向首地址, 后续占用
长度为file_hdr->keys_size, 每个key的长度为file_hdr->col_len
Rid *rids; // page->data的第三部分, 指针指向首地址, 每个rid的
长度为sizeof(Rid)
};
```

▶ 演示

◀ ◻ ▶ ◀ ◻ ▶ ◀ ≡ ▶ ◀ ≡ ▶ ≡ 🔍 ↺

Rucbase中B+树节点的定义ix_node_handle.h

```
struct IxPageHdr {
page_id_t next_free_page_no;
page_id_t parent; // its parent's page_no
int num_key; // # current keys (always equals to #child - 1)
插入的keys数量, key_idx ∈ [0, num_key)
bool is_leaf;
page_id_t prev_leaf; // previous leaf node's page_no, effective if is_leaf
page_id_t next_leaf; // next leaf node's page_no, effective if is_leaf
};
```

▶ 演示

◀ ◻ ▶ ◀ ◻ ▶ ◀ ≡ ▶ ◀ ≡ ▶ ≡ 🔍 ↺

Rucbase中B+树的定义ix_index_handle.h

```
class IxIndexHandle {
friend class IxScan;
friend class IxManager;

private:
DiskManager *disk_manager_;
BufferPoolManager *buffer_pool_manager_;
int fd_;
IxFileHdr file_hdr_; // 存了root_page, 但root_page初始化为2 (
存FILE_HDR_PAGE, 第1页存LEAF_HEADER_PAGE)
std::mutex root_latch_; // 用于索引并发 (请自行选择并发粒度
在 Tree级 或 Page级 )
};
```

► 演示

Rucbase中B+树的定义ix_index_handle.h

```
struct IxFileHdr {
page_id_t first_free_page_no;
int num_pages;           // disk pages
page_id_t root_page;    // root page no
ColType col_type;
int col_len;            // ColMeta->len
int btree_order;        // children per page 每个结点最多可插入的
键值对数量
int keys_size;          // keys_size = (btree_order + 1) * col_len
page_id_t first_leaf;   // 在上层IxManager的open函数进行初始
化, 初始化为root page_no
page_id_t last_leaf;
};
```

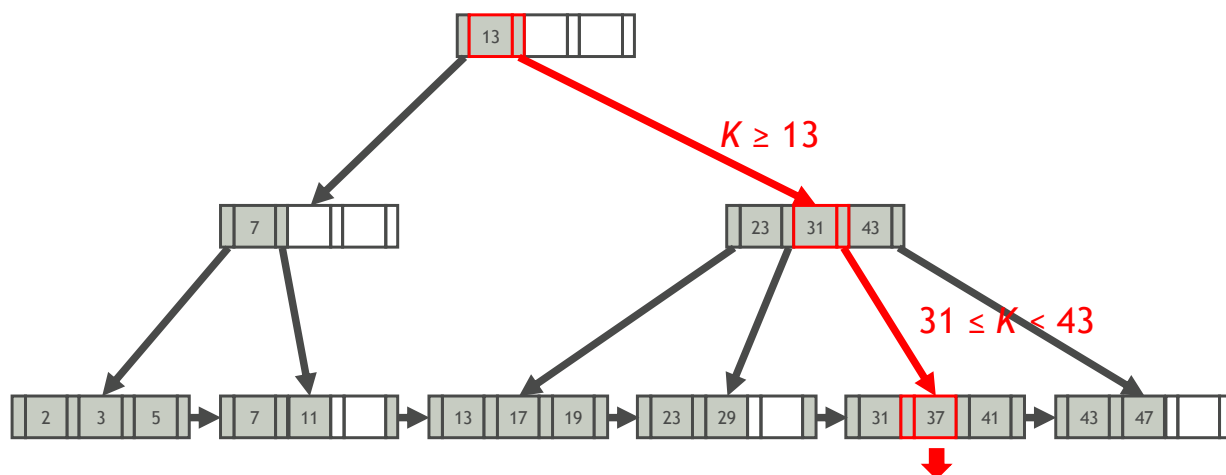
► 演示

查找索引项

查找键为 K

- ① 在内节点的引导下，找到 K 属于哪个叶节点
- ② 在该叶节点中查找键值为 K 的索引项

例: $K = 37$

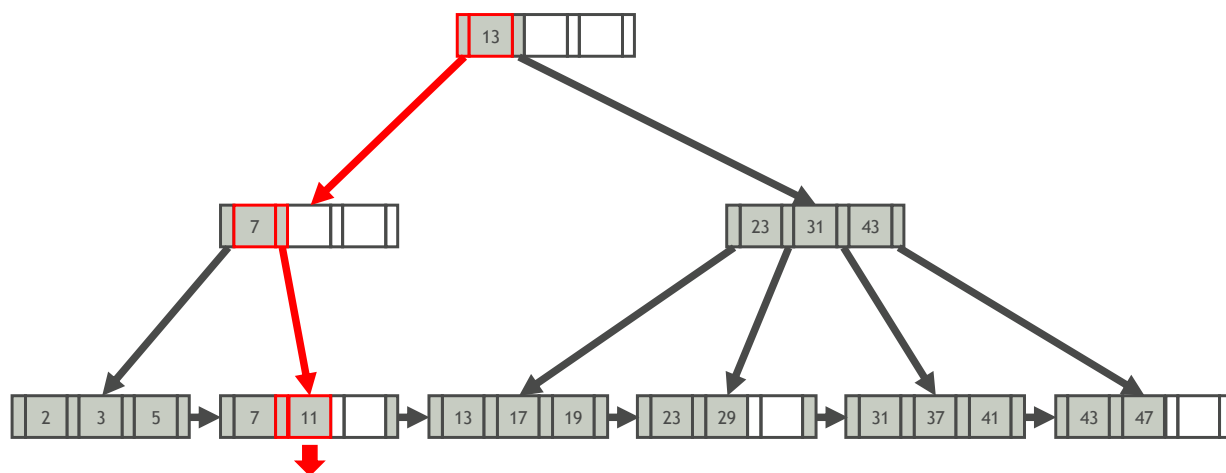


区间查询

查找键在区间 $[L, U]$ 内的全部索引项

- ① 找到具有大于等于 L 的最小键的索引项 E
- ② 扫描 E 右侧的索引项，如果键 $\leq U$ ，则输出；否则，终止

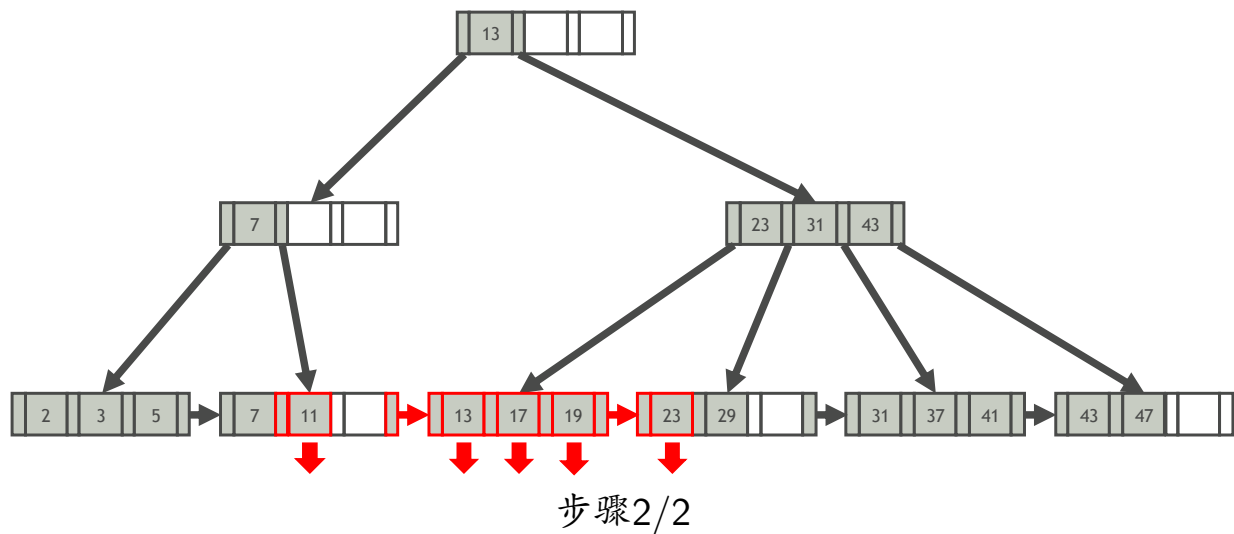
例: $K \in [10, 25]$



步骤1/2

区间查询(续)

例: $K \in [10, 25]$



Rucbase中B+树查找函数

```
IxNodeHandle *IxIndexHandle::FindLeafPage(  
    const char *key, Operation operation,  
    Transaction *transaction);
```

```
bool IxIndexHandle::GetValue(  
    const char *key, std::vector<Rid> *result,  
    Transaction *transaction);
```

// 在当前node中查找第一个 $\geq \text{target}$ 的key_idx

```
int IxNodeHandle::lower_bound(const char *target) const;
```

// 在当前node中查找第一个 $> \text{target}$ 的key_idx

```
int IxNodeHandle::upper_bound(const char *target) const;
```

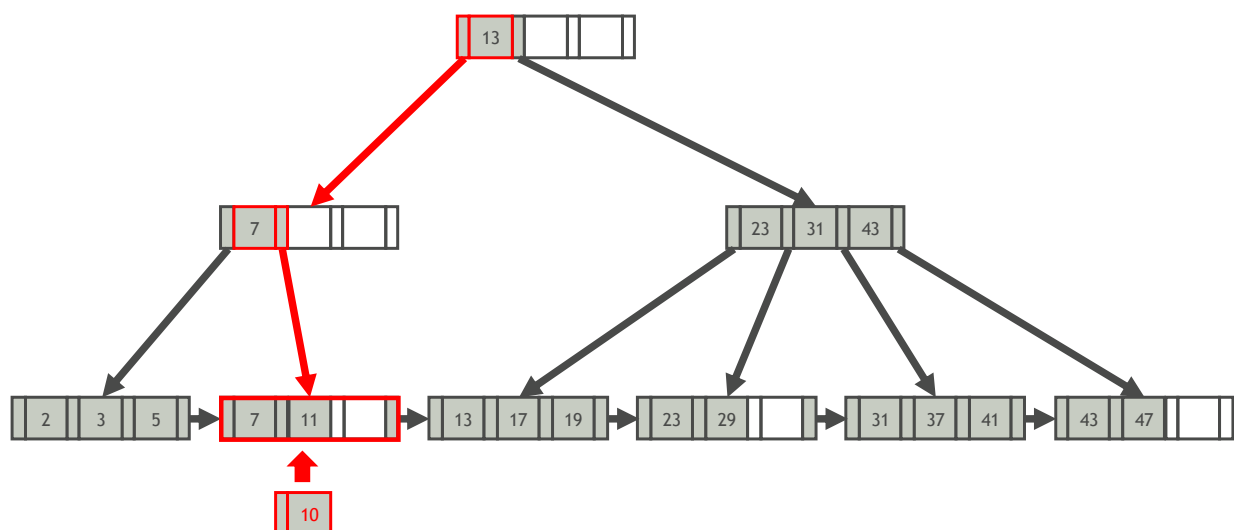
插入索引项

插入键为 K 的索引项

- ① 找到 K 应在的叶节点 L
- ② 将索引项插入 L
- ③ 如果 L 不溢出，则插入完成；否则，**分裂(split)** L

插入索引项(续)

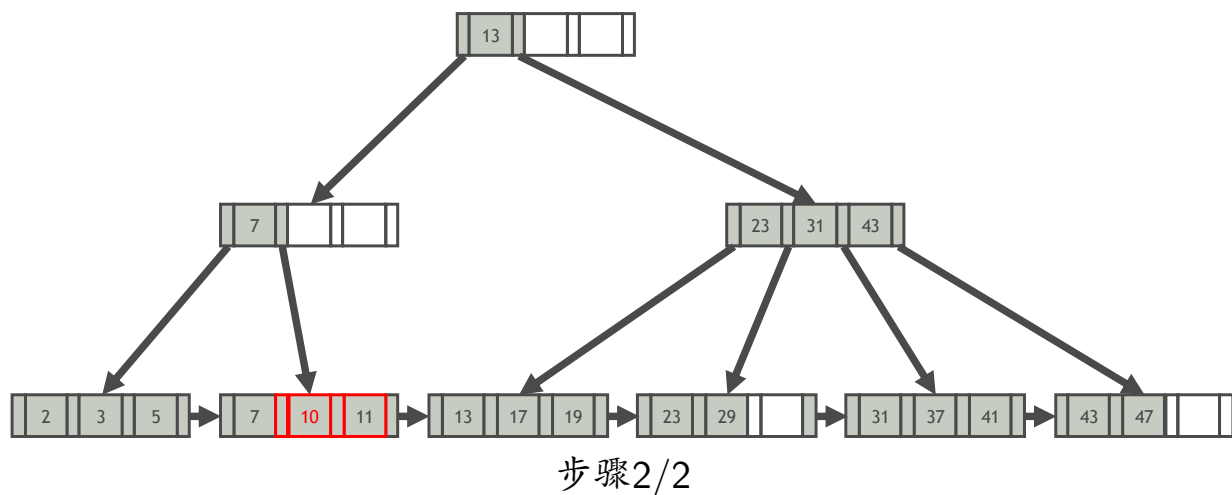
例1: $K = 10$ (无节点分裂)



步骤1/2

插入索引项(续)

例1: $K = 10$ (无节点分裂)



叶节点分裂

分裂叶节点 L

- ① 创建一个新的叶节点 L_2
- ② 将 L 中的索引项平分, 前半留在 L 中, 后半移入 L_2 中
- ③ 将 L_2 中最小的键存入“中间键(middle key)”变量
- ④ 在叶节点链表中, 将 L_2 插到 L 的右边
- ⑤ 在 L 的父节点 N 中插入middle key及指向 L_2 的指针
- ⑥ 如果 N 不溢出, 则完成对 L 的分裂; 否则, 继续分裂 N

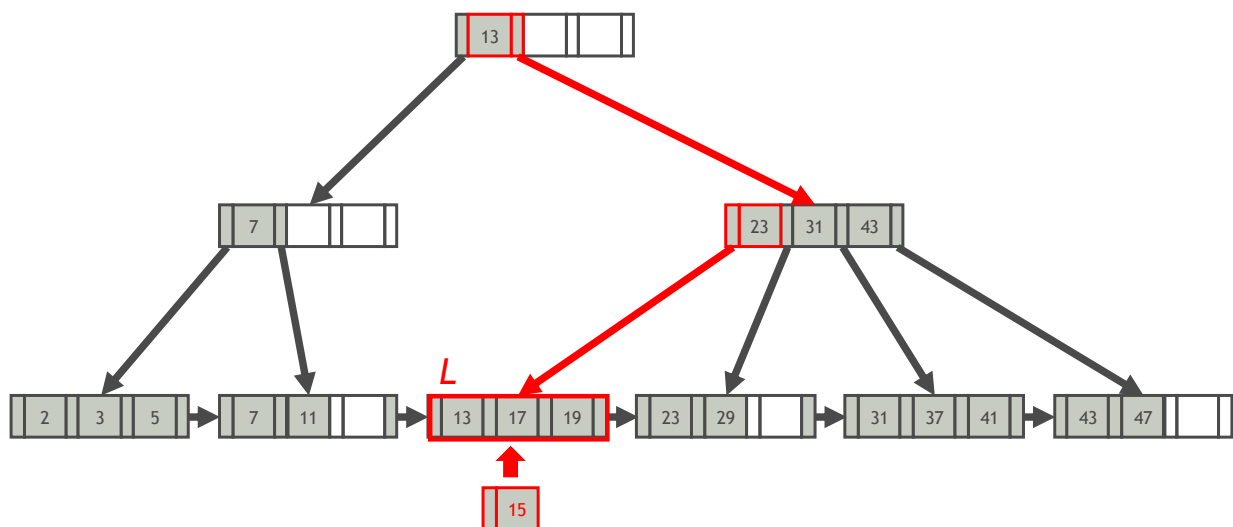
内节点分裂

分裂内节点 N

- ① 创建一个新的内节点 N_2
- ② 将 N 中的指针平分，前一半留在 N 中，后一半移入 N_2 中
- ③ 将 N 中多余的键存入“中间键(middle key)”变量
- ④ 如果 N 是根节点，则创建一个新的根节点 N' ，并在 N' 中插入一个指向 N 的指针
- ⑤ 在 N 的父节点 N' 中插入middle key及指向 N_2 的指针
- ⑥ 如果 N' 不溢出，则完成对 N 的分裂；否则，继续分裂 N'

插入索引项(续)

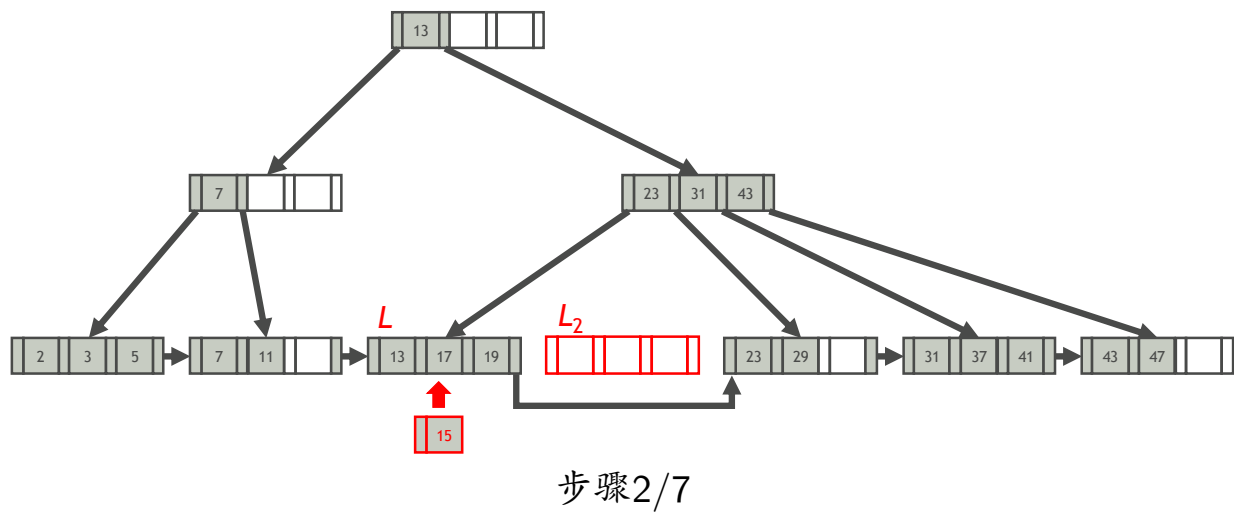
例2: $K = 15$ (有节点分裂)



步骤1/7

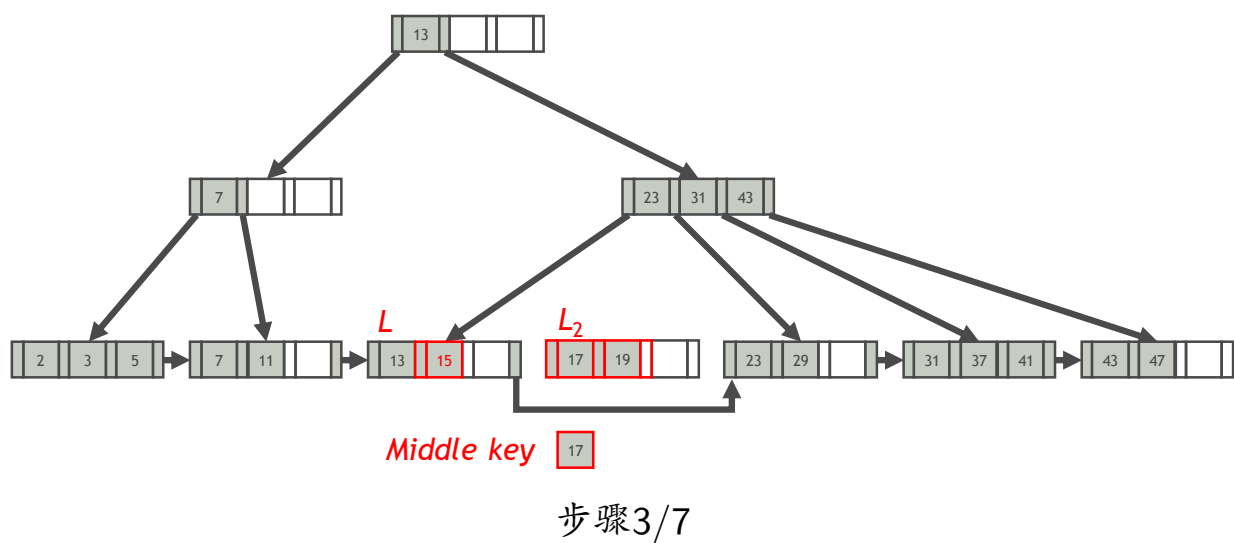
插入索引项(续)

例2: $K = 15$ (有节点分裂)



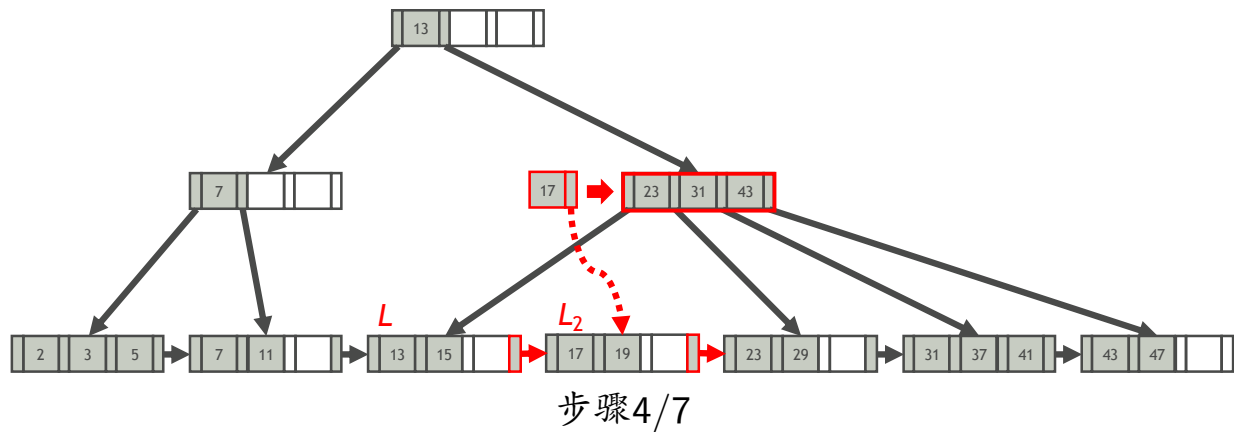
插入索引项(续)

例2: $K = 15$ (有节点分裂)



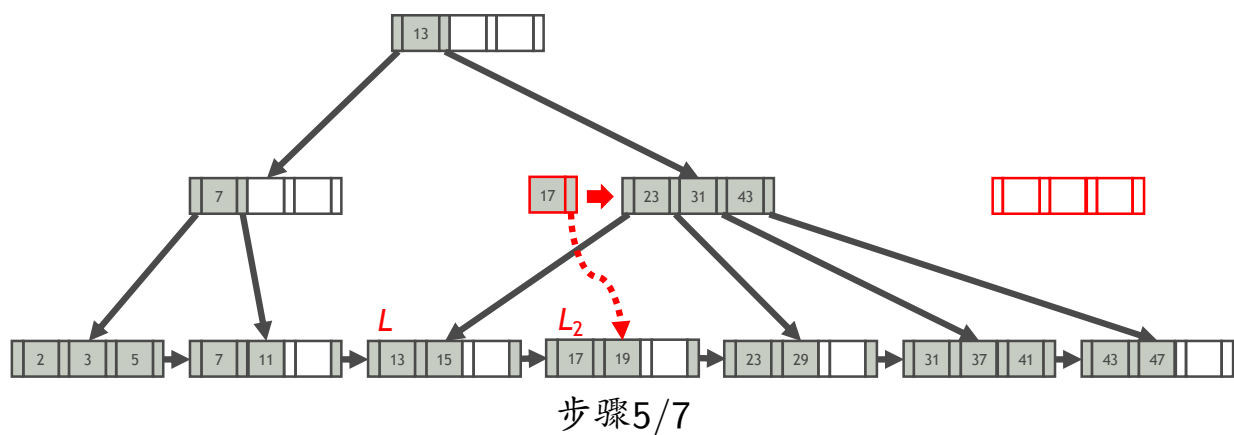
插入索引项(续)

例2: $K = 15$ (有节点分裂)



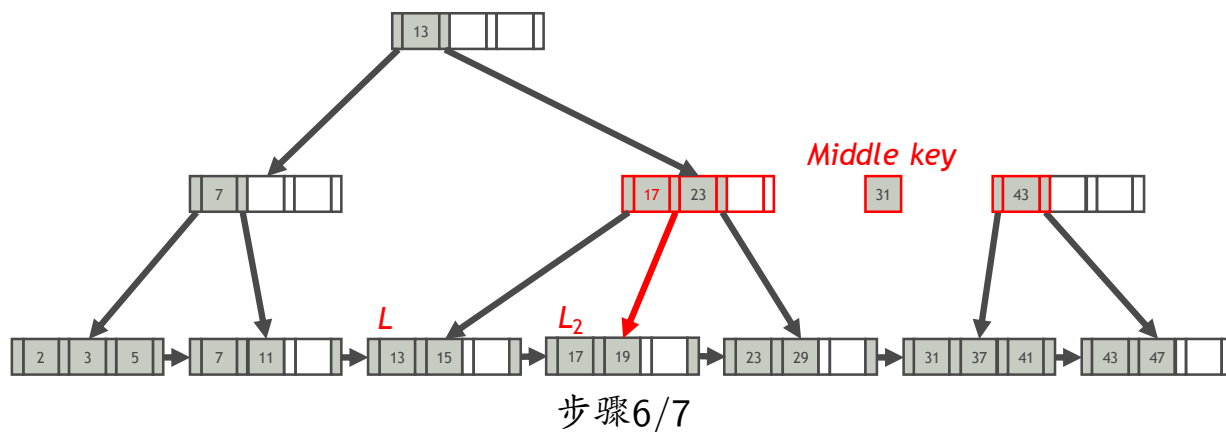
插入索引项(续)

例2: $K = 15$ (有节点分裂)



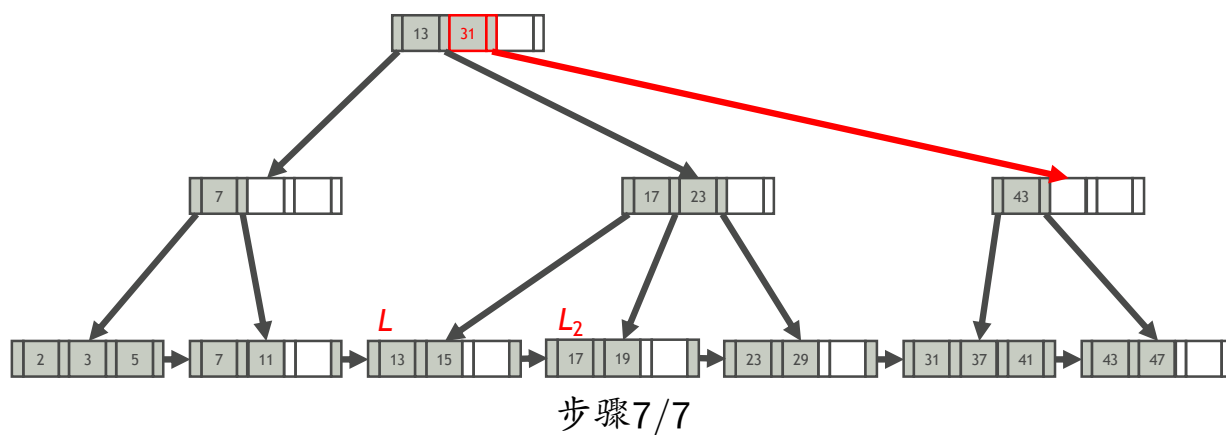
插入索引项(续)

例2: $K = 15$ (有节点分裂)



插入索引项(续)

例2: $K = 15$ (有节点分裂)



Rucbase中B+树插入函数

```
bool IxIndexHandle::insert_entry(  
    const char *key, const Rid &value,  
    Transaction *transaction);
```

```
IxNodeHandle *IxIndexHandle::Split(IxNodeHandle *node);
```

```
void IxIndexHandle::InsertIntoParent(  
    IxNodeHandle *old_node, const char *key,  
    IxNodeHandle *new_node, Transaction *transaction);
```

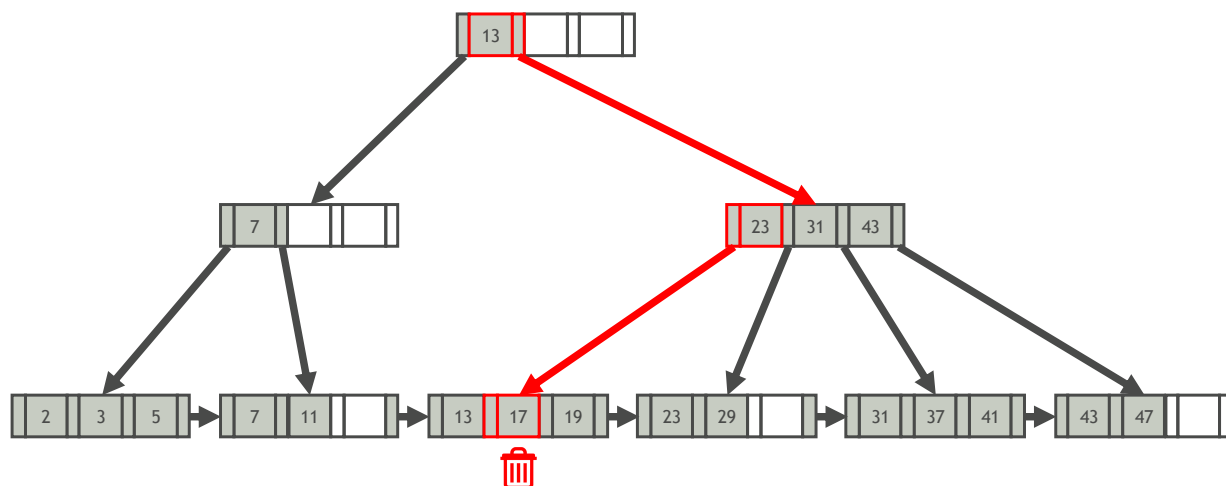
删除索引项

删除键为 K 的索引项

- ① 找到 K 所在的叶节点 L
- ② 从 L 中删除键为 K 索引项
- ③ 如果 L 至少半满, 则完成删除; 否则, 处理 L , 使 L 至少半满

删除索引项(续)

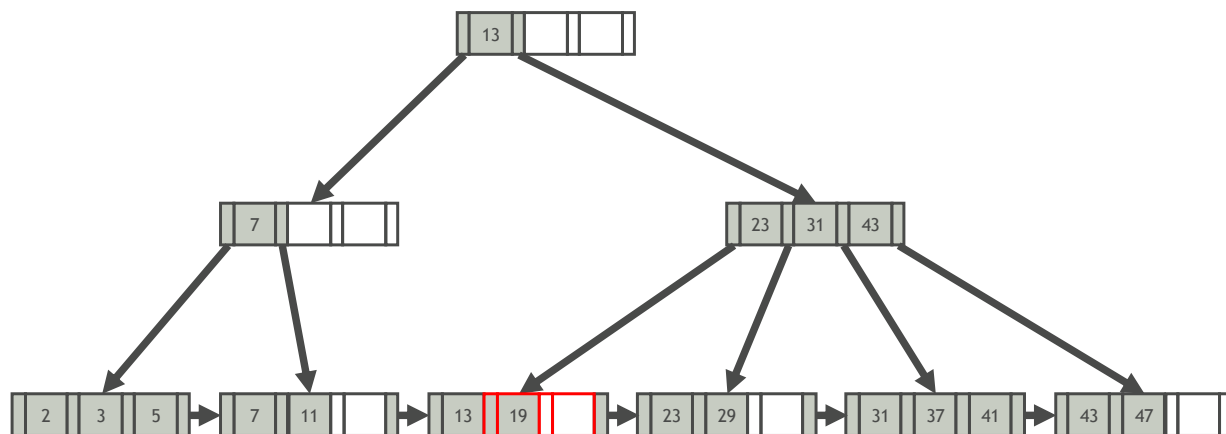
例1: $K = 17$ (没有键重分布及节点合并)



步骤1/2

删除索引项(续)

例1: $K = 17$ (没有键重分布及节点合并)



步骤2/2

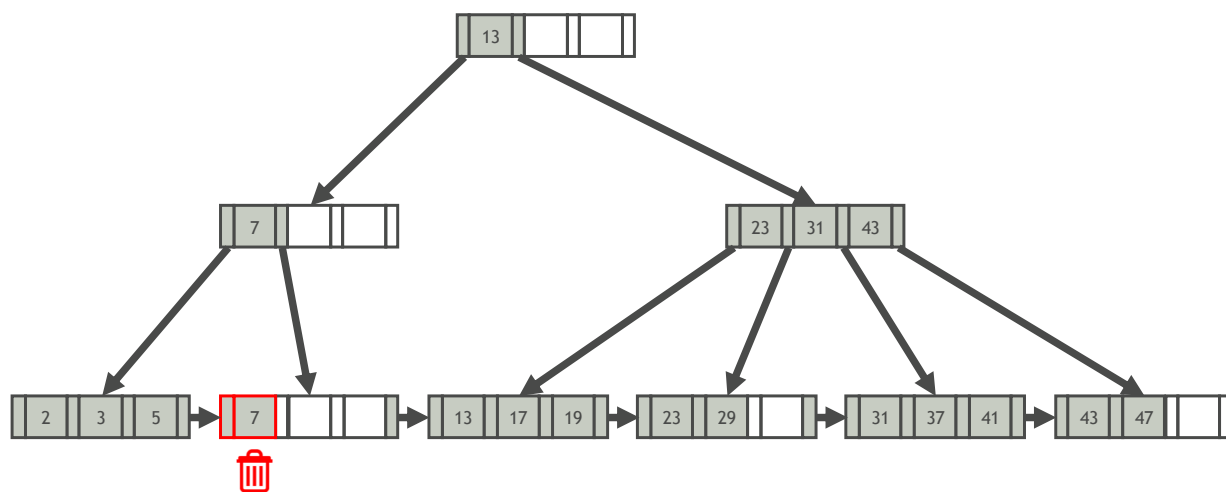
删除索引项(续)

使叶节点 L 至少半满的处理方法

- ① 尝试从 L 相邻的兄弟节点借一个索引项，使两者均至少半满(如果 L 向左右兄弟都能借，则优先向左兄弟借)
- ② 如果借不到，则将 L 与其兄弟节点合并(merge)

删除索引项(续)

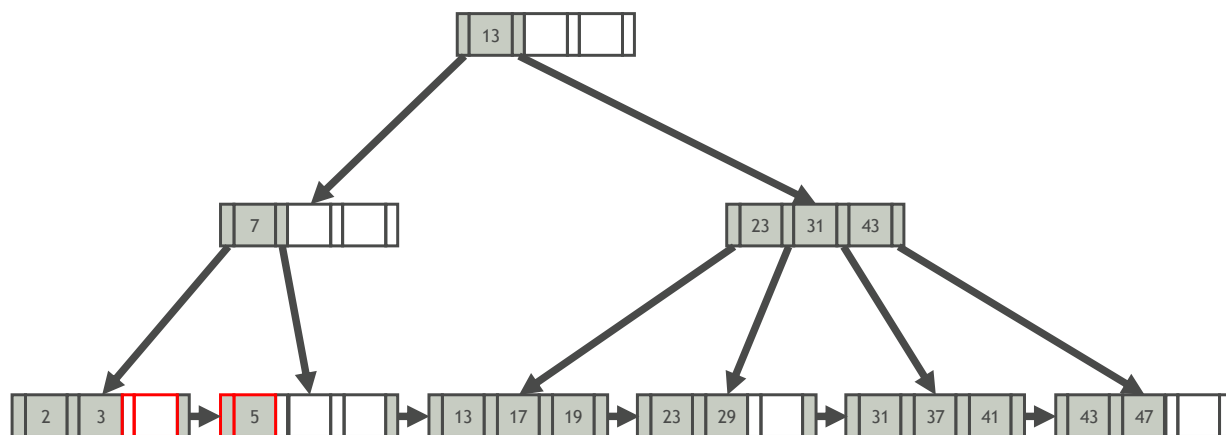
例2: $K = 7$ (需要重分布键)



步骤1/3

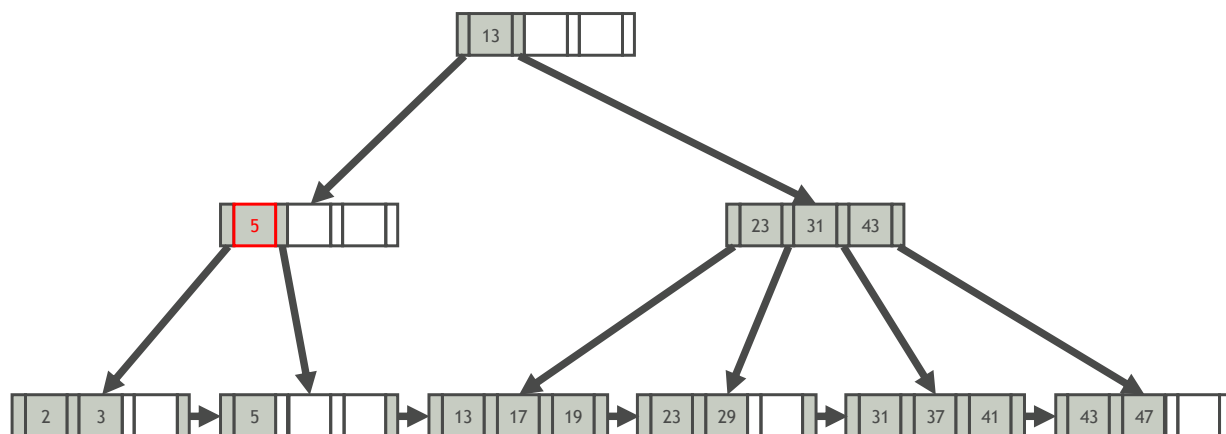
删除索引项(续)

例2: $K = 7$ (需要重分布键)



删除索引项(续)

例2: $K = 7$ (需要重分布键)



- 如果一个节点的最小键发生了变化, 需要更新其父节点中相应的键
- 如果父节点中最小键也发生了变化, 需要递归更新父节点的父节点中相应的键
- ...

删除索引项(续)

如果 L 借不到，则将 L 与其兄弟节点合并(merge)

节点合并

- ① 如果 L 与左侧兄弟节点 L_1 合并，则从 L 的父节点中删除指向 L 的指针及相应的键；
如果 L 与右侧兄弟节点 L_2 合并，则从 L 的父节点中删除指向 L_2 的指针及相应的键
如果 L 与左右兄弟都能合并，则优先与左兄弟合并
- ② 如果 L 的父节点 N 至少半满，则完成合并；否则，处理 N ，使 N 至少半满
 - ▶ 如果 N 是根节点，且 N 中只有一个指针，则删除 N
 - ▶ 如果 N 是内节点，则处理 N ，使 N 至少半满

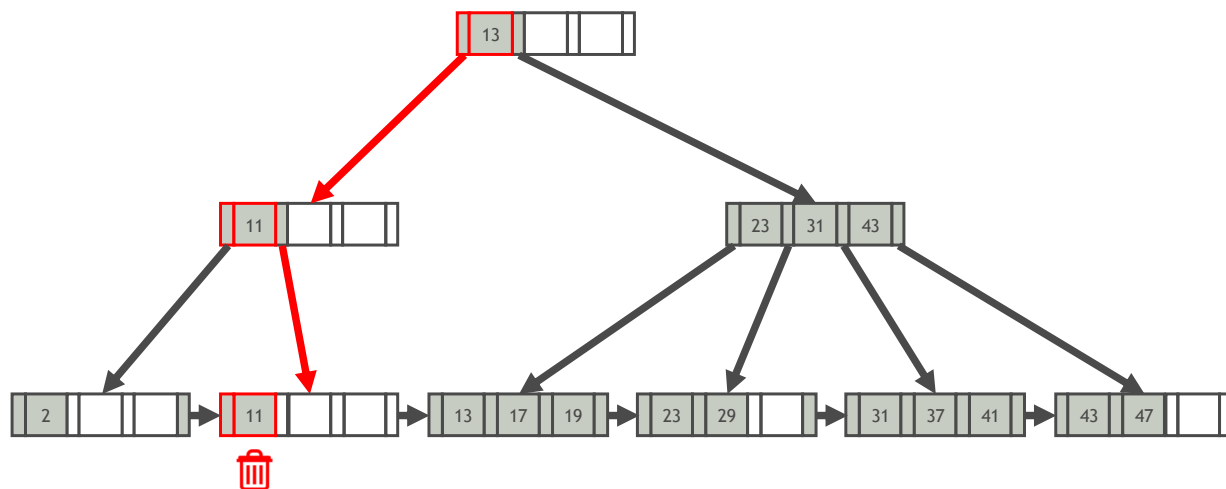
删除索引项(续)

使内节点 N 至少半满的处理方法

- ① 尝试从 N 相邻的兄弟节点借一个指针及键，使两者均至少半满
- ② 如果借不到，则将 N 与其兄弟节点合并(merge)

删除索引项(续)

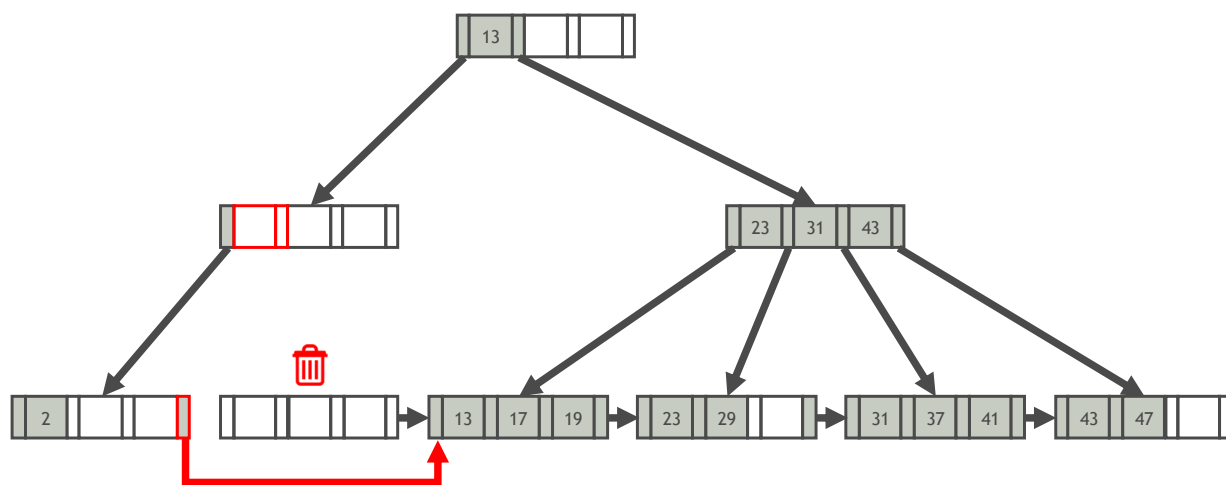
例3: $K = 11$ (需要合并节点)



步骤1/4

删除索引项(续)

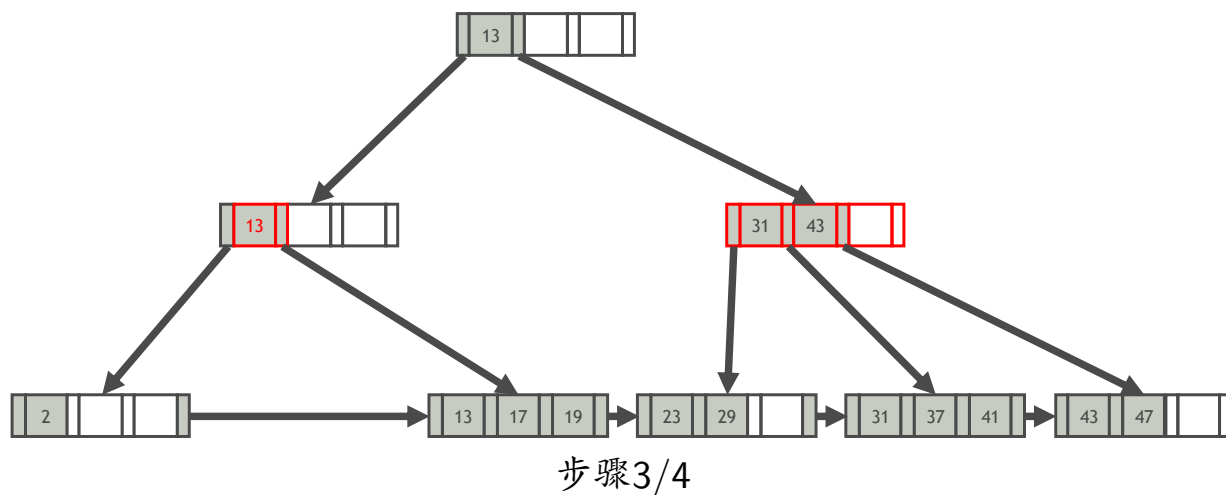
例3: $K = 11$ (需要合并节点)



步骤2/4

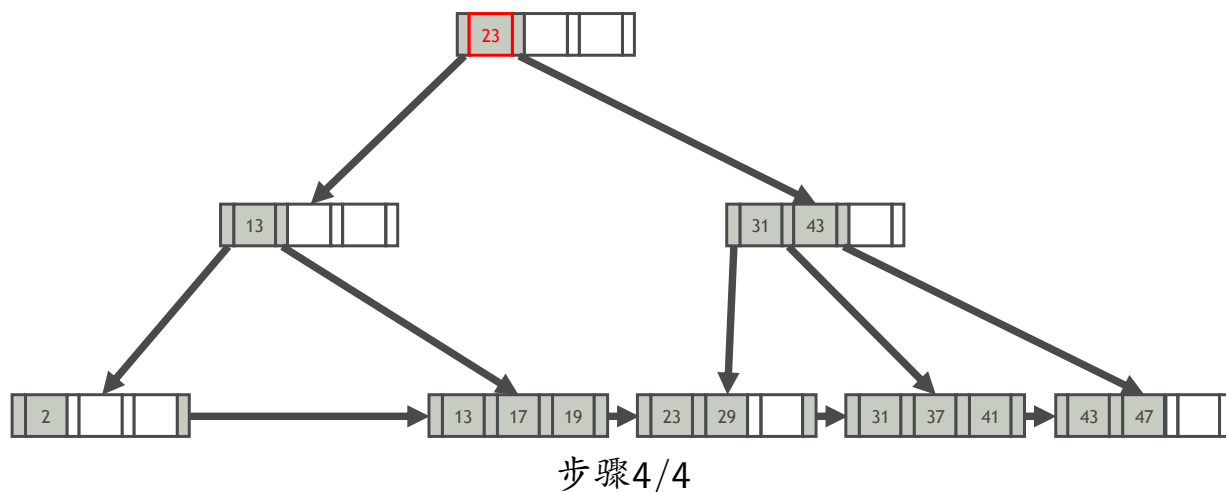
删除索引项(续)

例3: $K = 11$ (需要合并节点)



删除索引项(续)

例3: $K = 11$ (需要合并节点)



- 如果一个节点的最小键发生了变化, 需要更新其父节点中相应的键
- 如果父节点中最小键也发生了变化, 需要递归更新父节点的父节点中相应的键
- ...

Rucbase 中 B+ 树插入函数

```
bool IxIndexHandle::delete_entry(  
    const char *key, Transaction *transaction);  
  
bool IxIndexHandle::CoalesceOrRedistribute(  
    IxNodeHandle *node, Transaction *transaction);  
  
bool IxIndexHandle::AdjustRoot(IxNodeHandle *old_root_node);  
  
void IxIndexHandle::Redistribute(  
    IxNodeHandle *neighbor_node, IxNodeHandle *node,  
    IxNodeHandle *parent, int index);  
  
bool IxIndexHandle::Coalesce(  
    IxNodeHandle **neighbor_node, IxNodeHandle **node,  
    IxNodeHandle **parent, int index,  
    Transaction *transaction);
```

B+ 树演示

<https://cmudb.io/btree>

键压缩(Key Compression)

对键进行压缩，尽可能减少键的长度

- 从B+树中查找一个索引项所需的磁盘I/O数 = B+树的高度 $\approx \log_{fan_out}(\# \text{ of index entries})$
- 索引键越长 \implies 扇出数越小 \implies B+树越高 \implies 查询时间越长

前缀压缩(Prefix Compression)

- 同一叶节点中的键很可能具有相同的前缀(prefix)
- 提取键的公共前缀，只存储每个键的后缀(suffix)

Example (前缀压缩)

Microphone	Microsoft	Microwave
------------	-----------	-----------

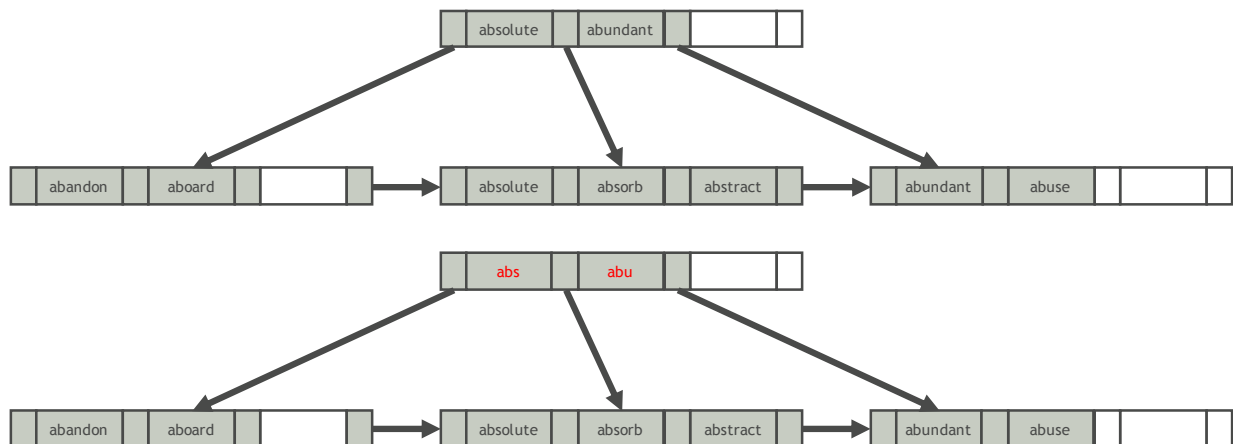
↓ 前缀压缩

前缀: Micro

phone	soft	wave
-------	------	------

后缀截断(Suffix Truncation)

- 内节点中的键仅用于导航
- 不需要在内节点中存储整个键
- 在保证正确导航的前提下，只需存储每个键的最短前缀即可



批量加载(Bulk Loading)

在一组页中的索引项上建立B+树

自顶向下的方法

- 从一棵空的B+树开始，每次插入一个索引项
- 缺点：插入每个索引项都需要从根节点向下走到叶节点

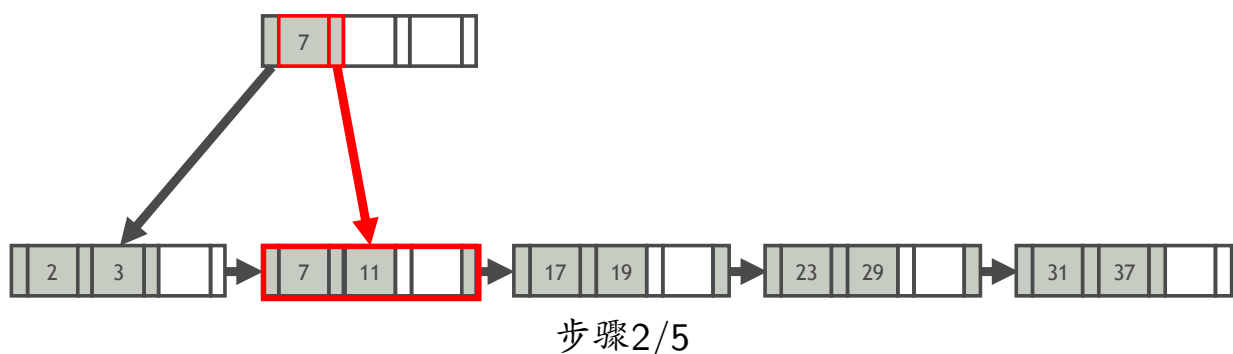
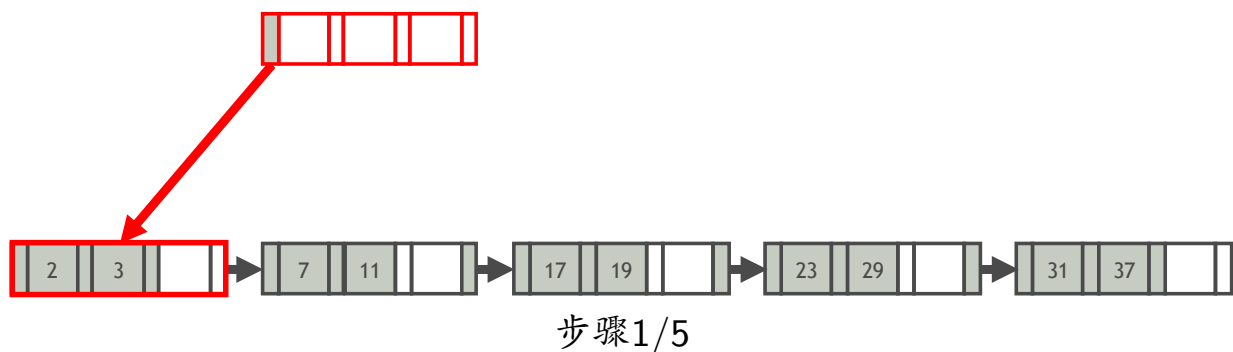
批量加载(续)

自底向上的方法

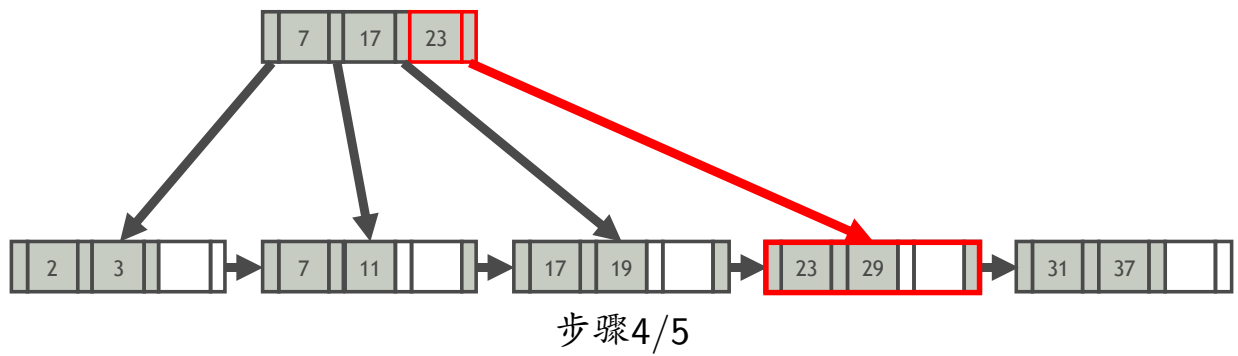
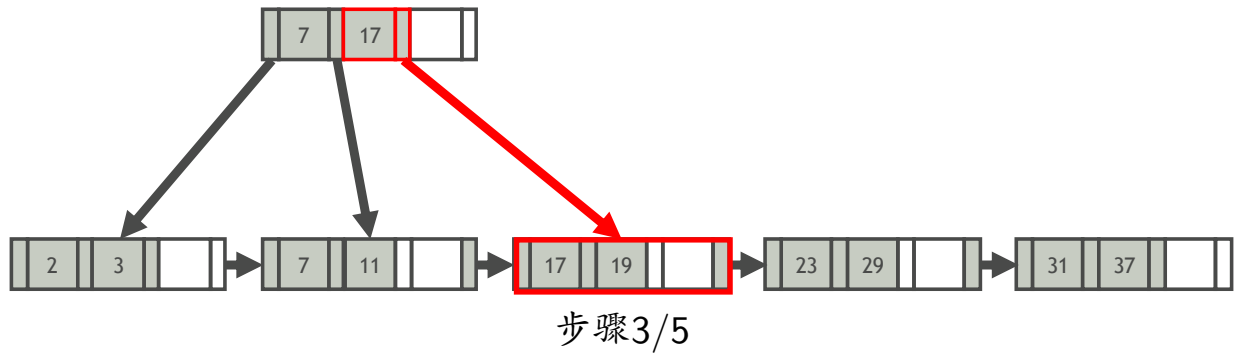
- ① 使用外存归并排序算法对所有页中的索引项排序
- ② 将每个页作为一个叶节点，建立叶节点链表
- ③ 创建一个空的内节点作为根，并插入一个指针，指向第一个叶节点
- ④ 对叶节点链表中的下一个叶节点 L ，向叶节点上层最右边的内节点插入 L 中最小的键及指向 L 的指针；如果内节点溢出，则分裂
- ⑤ 重复第4步，直至所有叶节点都插入B+树为止

批量加载(续)

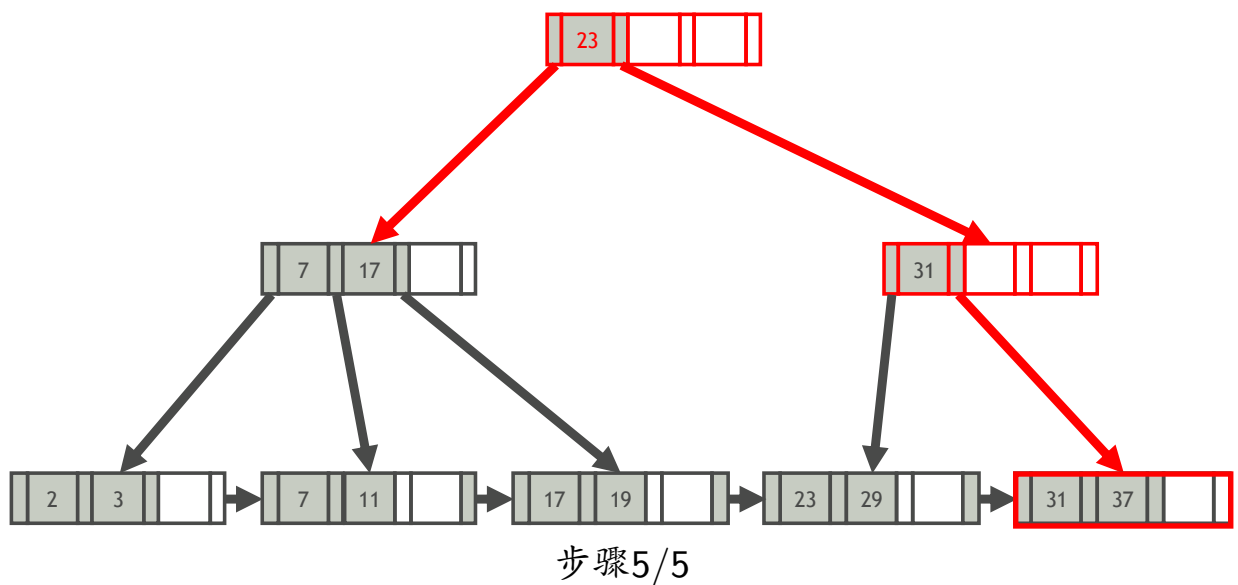
例: 在排好序的键2, 3, 7, 11, 17, 19, 23, 29, 31, 37上建立B+树



批量加载(续)



批量加载(续)



Log-Structured Merge-Trees (LSM-Trees)

B+树的原地更新(In-Place Updates)

B+树使用原地更新

- 直接使用新数据覆盖旧数据
- 数据更新过程产生大量随机磁盘I/O

日志结构合并树(Log-Structured Merge-Trees, LSM-Trees)

LSM树被广泛用于NoSQL数据库系统的存储层

- LevelDB、RocksDB、HBase、Cassandra、TiDB等

LSM树执行**异地更新(out-of-place updates)**

- 写操作首先缓存在内存中
- 内存缓冲区中的写操作后续会刷写到磁盘文件，并与现有文件合并
- 数据更新过程只使用顺序磁盘I/O

LSM树的基本结构

LSM树由两部分构成

- **Memtable**: 内存B+树或内存哈希表
- **不可变文件(immutable file)**: 磁盘上不可更新的文件

Example (LSM树)

Memtable (3, 333), (7, 777) 内存

Immutable file (2, 222), (3, 123), (5, 555), (8, 888) 磁盘

LSM树的查找操作

查找键为 K 的索引项

- ① 首先在memtable中查找键为 K 的索引项
- ② 如果找到，则返回索引项；否则，在不可变文件中查找索引项

Example (LSM树的查找操作)

$K = 3$

Memtable (3, 333), (7, 777) 内存

Immutable file (2, 222), (3, 123), (5, 555), (8, 888) 磁盘

Example (LSM树的查找操作)

$K = 5$

Memtable (3, 333), (7, 777) 内存

Immutable file (2, 222), (3, 123), (5, 555), (8, 888) 磁盘

缺点：当不可变文件非常大时，在文件上查找的效率非常低

LSM树的更新操作

更新键为 K 的索引项

- ① 首先将更新操作缓存在memtable中(原地更新)
- ② 当memtable写满后，将memtable的内容与immutable文件的内容合并(compact)，合并后的内容写入新文件，并用新文件替换旧文件

Example (LSM树的更新操作)

合并前

Memtable (3, 333), (Delete 8) 内存

Immutable file (2, 222), (3, 123), (5, 555), (8, 888) 磁盘

合并后

Memtable \emptyset 内存

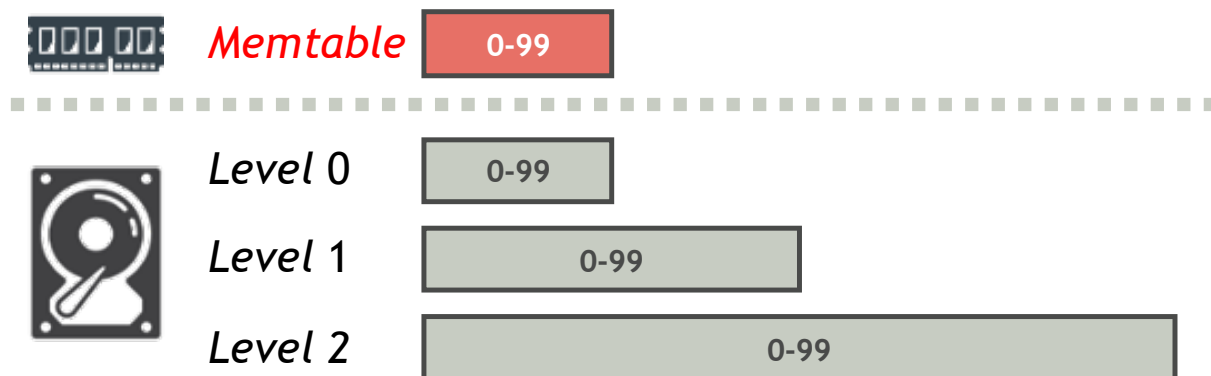
Immutable file (2, 222), (3, 333), (5, 555) 磁盘

缺点：当不可变文件非常大时，合并过程非常慢

分层LSM树

分层LSM树包含多个层

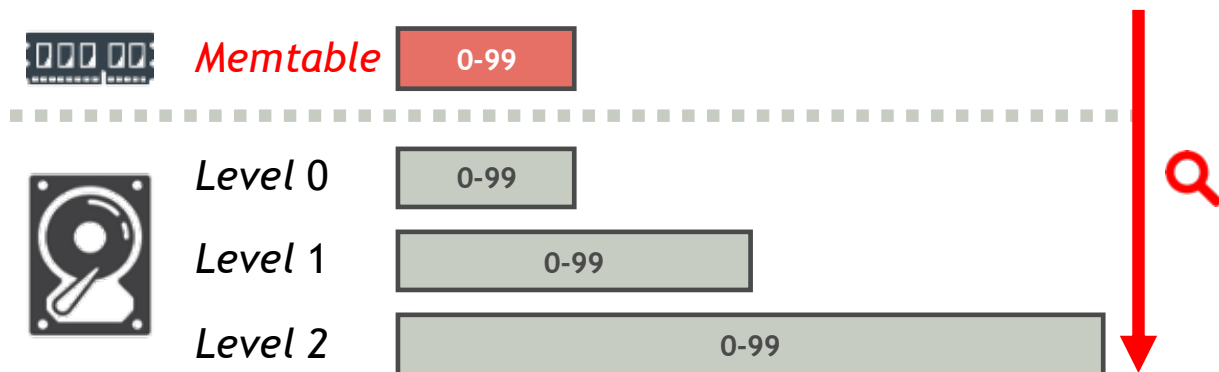
- **Memtable**: 内存B+树或内存哈希表
- **Level 0**: memtable在磁盘上的不可变副本(键-值对按键排序)
- **Level i ($i \geq 1$)**: 磁盘上的不可变有序文件(键-值对按键排序)
 - ▶ 第 $i+1$ 层的键-值对比第 i 层的键-值对旧
 - ▶ 第 $i+1$ 层的文件比第 i 层的文件大 T 倍



分层LSM树的查找操作

在分层LSM树上, 从上向下查找键为 K 的键-值对

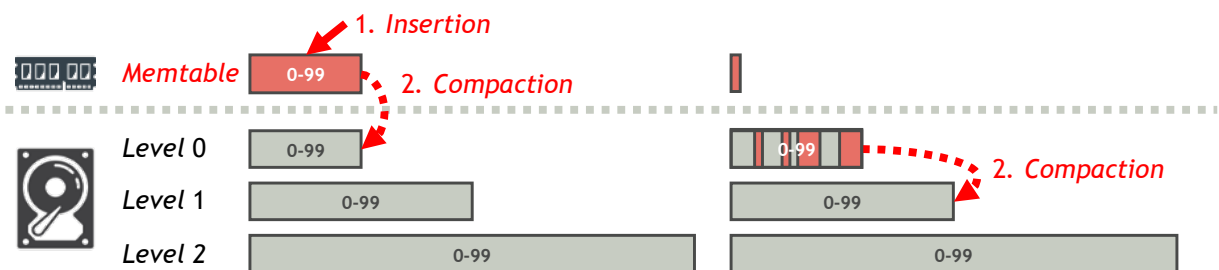
- ① 如果找到了键为 K 的键-值对, 则返回该键-值对
- ② 如果找到了一个墓碑(tombstone), 则返回“不存在”
- ③ 如果在所有层上均未找到, 则返回“不存在”



分层LSM树的插入操作

插入键-值对(K, V)

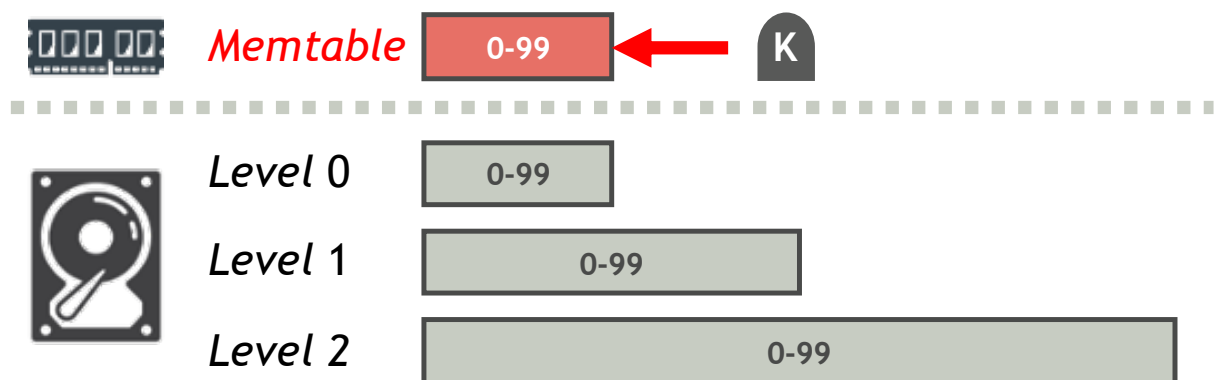
- ① 将(K, V)插入memtable (内存中, 原地更新)
- ② 如果memtable未溢出, 则完成插入; 否则, 将memtable中的键-值对写入第0层, 成为不可变的有序文件(磁盘中, 顺序I/O)
- ③ 如果第 i 层溢出, 则将第 i 层的键-值对合并到第 $i+1$ 层(磁盘中, 异地更新, 顺序I/O)



分层LSM树的删除操作

删除键为 K 的键-值对

- ① 在memtable中为 K 插入一个墓碑(tombstone)
- ② 在合并时, 删除键为 K 且比墓碑旧的键-值对



B+树VS LSM-树

	B+树	LSM-树
更新方法	原地更新	异地更新
空间放大(space amplification)	低(一个键只有1个副本)	高(一个键有多个副本)
写性能	低(随机I/O)	高(顺序I/O)
空间利用率	空间碎片化(平均一个页有1/4空闲)	高(键-值对在不可变文件中有序存储)
并发控制与故障恢复	复杂	简单(文件不可更改且合并操作只进行异地更新)

Advanced Topics in Indexing

位图索引(Bitmap Index)

当属性的取值较少时, 可以使用位图索引加快多属性上的选择查询

- 对关系 R 的属性 A 的每个取值 v 建一个长度为 $|R|$ 的位图 I_v
- I_v 的第 i 位为1当且仅当 R 中第 i 个元组的 A 属性值为 v

Example (位图索引)

编号	Student关系				
	Sno	Sname	Ssex	Sage	Sdept
0	CS-001	Elsa	F	19	CS
1	CS-002	Ed	M	19	CS
2	MA-001	Abby	F	18	Math
3	PH-001	Nick	M	20	Physics

Ssex上的位图索引

'F': 1010

'M': 0101

Sdept上的位图索引

'CS': 1100

'Math': 0010

'Physics': 0001

SELECT * FROM Student WHERE Ssex = 'F' AND Sdept = 'CS';

'F': 1010 bitwise AND 'CS': 1100 = 1000, 因此元组0是查询结果

空间索引(Spatial Index)

空间索引用于索引空间数据(spatial data)

- kd树(kd-tree)
- R树(R-tree)

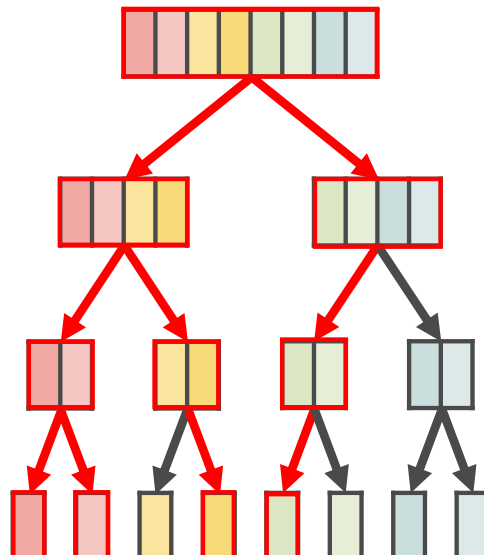
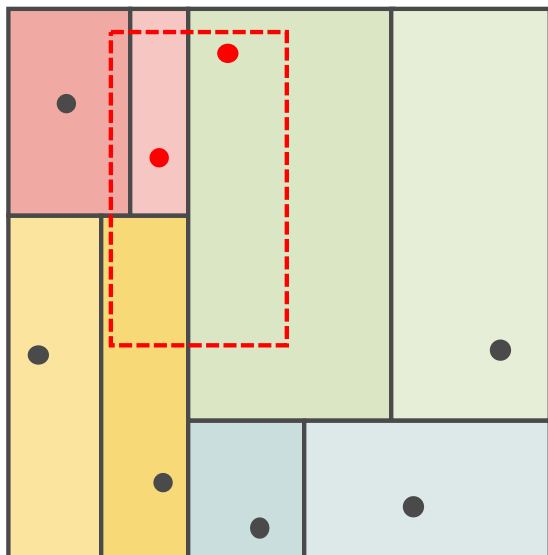
空间数据查询

- 范围查询(range query): 查询位于给定范围内的数据对象
- 近邻查询(nearest neighbor query): 查询与一个给定对象距离最近的一个或多个其他对象

kd树(kd-Tree)

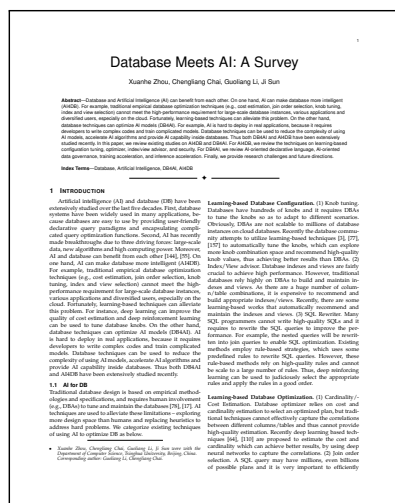
kd树是一种多索引结构

- kd树的每一层都把数据空间分成两部分
- 如果kd树的一个节点代表的子空间与查询区域不相交，则该节点中的所有数据对象都不可能是查询结果



基于人工智能的索引结构

使用人工智能技术，优化索引结构设计或设计基于模型的索引结构



X. Zhou, C. Chai, G. Li, J. Sun. **Database Meets Artificial Intelligence: A Survey.** *IEEE Transactions on Knowledge and Data Engineering*, 34(3):1096–1116, 2022.

Section 2.3.1 “Learned Data Structure”

总结

- ① Indexes
- ② Hash-based Index Structures
 - Extensible Hash Tables
 - Linear Hash Tables
- ③ Tree-based Index Structures
 - B+ Trees
- ④ Log-Structured Merge-Trees (LSM-Trees)
- ⑤ Advanced Topics in Indexing

Q&A

- ① 当B+树进行删除操作时，若一个节点不足半满，是优先向左兄弟借，还是优先向右兄弟借呢？
答: 都可以，取决于B+树的具体实现方法。

致谢

- 感谢詹儒彦(1190202307)、金彦铮(1190200418)、杨宇辰(1190300611)、蔡思娣(1190201925)同学指出课件中的错误