



GREENPLUM
DATABASE®



阿里云
aliyun.com



Greenplum内核揭秘之 B树索引

钉钉直播 | 7月24日 16:00 - 17:00





Greenplum中的B树索引

大纲

- B树基础知识
- 存储结构
- 操作算法
- 并发控制
- 相关系统表
- 总结与展望

A light blue triangle with a thin grey border, pointing upwards.

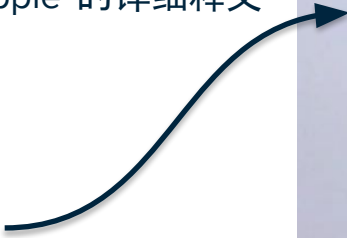
B-Tree
Index

B树基础知识

- 我们上学时在【数据结构】课程中学习了B树, 现在让我们再回忆它一下
- 为什么数据库索引会选用B树这种结构呢?
- Greenplum中的B树索引和课程中的实现完全一样吗?

索引简介

- 什么是索引？**加速**常规操作的数据结构
- 举例
 - B树索引
 - Hash索引, 例如cache
 - 倒排索引, 用于全文检索
- 类比: 一本字典, 我们要查找“apple”的详细释义
 - 可以从头到尾逐页查找
 - 或者使用附录中的索引!
 - apple -> 202页



a		
actor 演員	095	bag 袋子 316
actress 女演員	324	bakery 麵包店 306
afternoon 下午	344	ball 球 320
airplane 飛機	337	balloon 氣球 320
airport 機場	090	banana 香蕉 204
angry 生氣的	241	bank 銀行 106
ant 螞蟻	305	baseball 棒球 127
apple 蘋果	202	basket 籃子 320
April 四月	341	basketball 籃球 128
apron 圍裙	174	bathroom 浴室 193
arm 手臂	039	beach 沙灘 322
art 藝術	123	beach ball 海灘球 322
astronaut 太空人	325	bear 熊 020
		beautiful 美麗的 228
		bed 床 303

B树简介

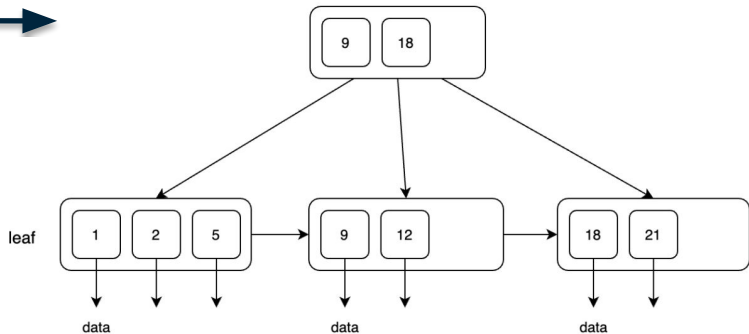
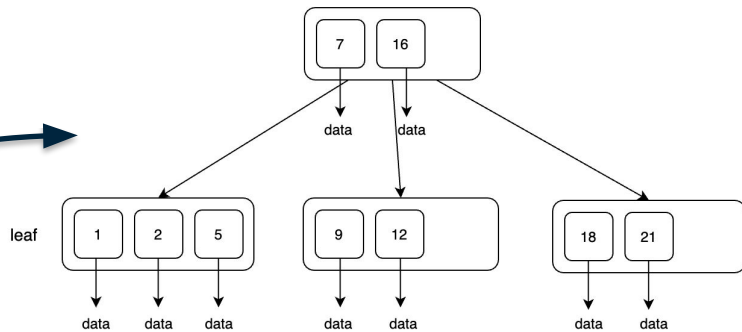
- B树是一个大家族, 可以细分为好多子类别
- 对比二叉树: 它们都是平衡树, 但是B树中的每个节点中都存储大量键值, 即树不会太高

- B树

- 节点中存储若干个键值, 节点有序排列
- 节点中的键值指向目标数据(key->data)

- B+树

- 叶子节点存储了全部键值(key), 这些键值再指向目标数据
- 内部节点中重复存储部分键值, 但不含数据指针
- 叶子节点层有一条正向遍历链表



B树简介

- B+树非常适用于数据库中的索引结构: 减少磁盘IO, 每个节点对应磁盘中的一个页, 访问节点对应一次磁盘IO

- 它是Greenplum中的默认索引类型

```
demo=# select * from big where id=10000;
```

...

Time: 19490.566 ms

```
demo=# create index on big(id);  
CREATE INDEX
```

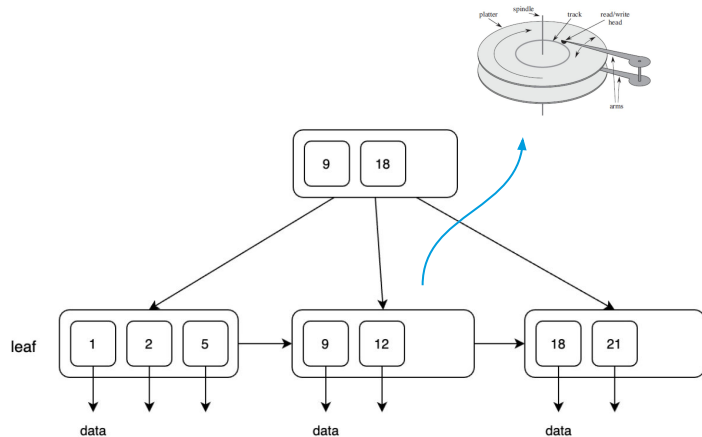
```
demo=# analyze big;  
ANALYZE
```

```
demo=# select * from big where id=10000;
```

...

Time: 210.594 ms

100X boost!



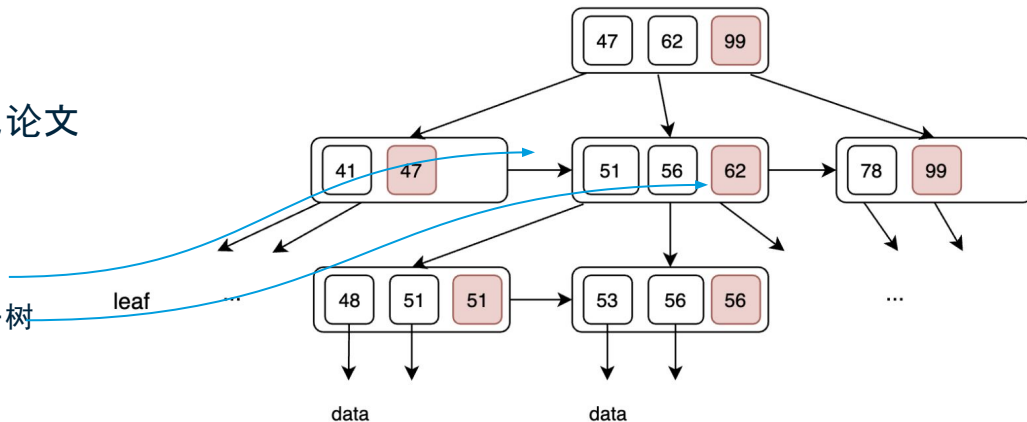
- 用于数据库生产环境中的B树, 我们还需要:
 - 支持故障(宕机)恢复: WAL中记录节点页面和树结构(如页面分裂)的变化
 - 提供高性能的并发控制
- Greenplum中的B+树: **Blink树**
 - 来自Lehman和Yao于1981年的论文《Efficient locking for concurrent operations on B-trees》

引入了右兄弟指针和高键

- Greenplum中的实现参考了此论文

并进行了少许改进

- 右兄弟指针: 内部节点同层间可以向右移动
- 高键(HighKey): 当前节点和以当前节点为根的子树中的最大键值



存储结构

- 现在我们已经了解了B树的基本知识, 接下来让我们看看B树的存储结构
- Greenplum中的B树索引物理上是怎么存储的?
- 放大B树索引, 每个树节点的结构是什么样子的?

索引结构 – 物理存储

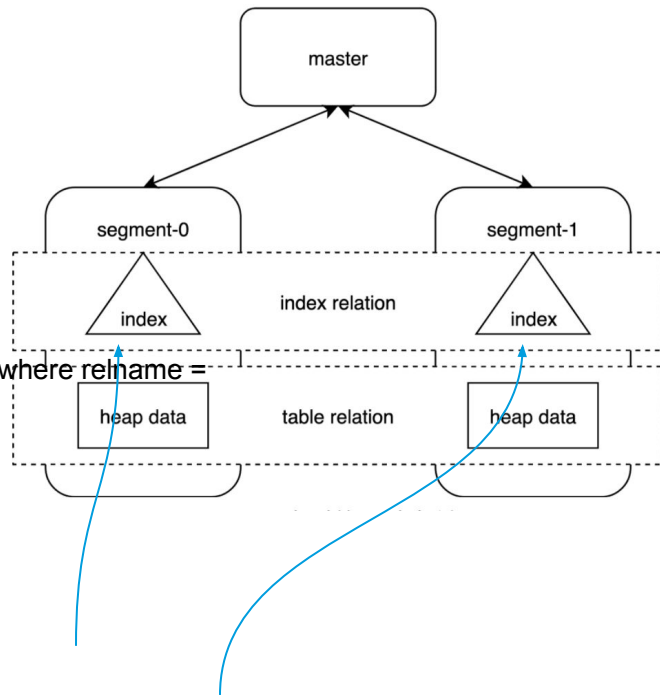
- Greenplum中的索引都是二级索引(非聚集索引)
 - 物理存储上是独立的文件(独立于表中的数据文件)
 - 并且也是按分片存储在每个segment上, 其索引内容对应segment上的数据分片

```
demo=# select relname, relfilenode, gp_segment_id from gp_dist_random('pg_class') where relname = 'student_id_idx';
```

relname	relfilenode	gp_segment_id
student_id_idx	16524	1
student_id_idx	16524	0

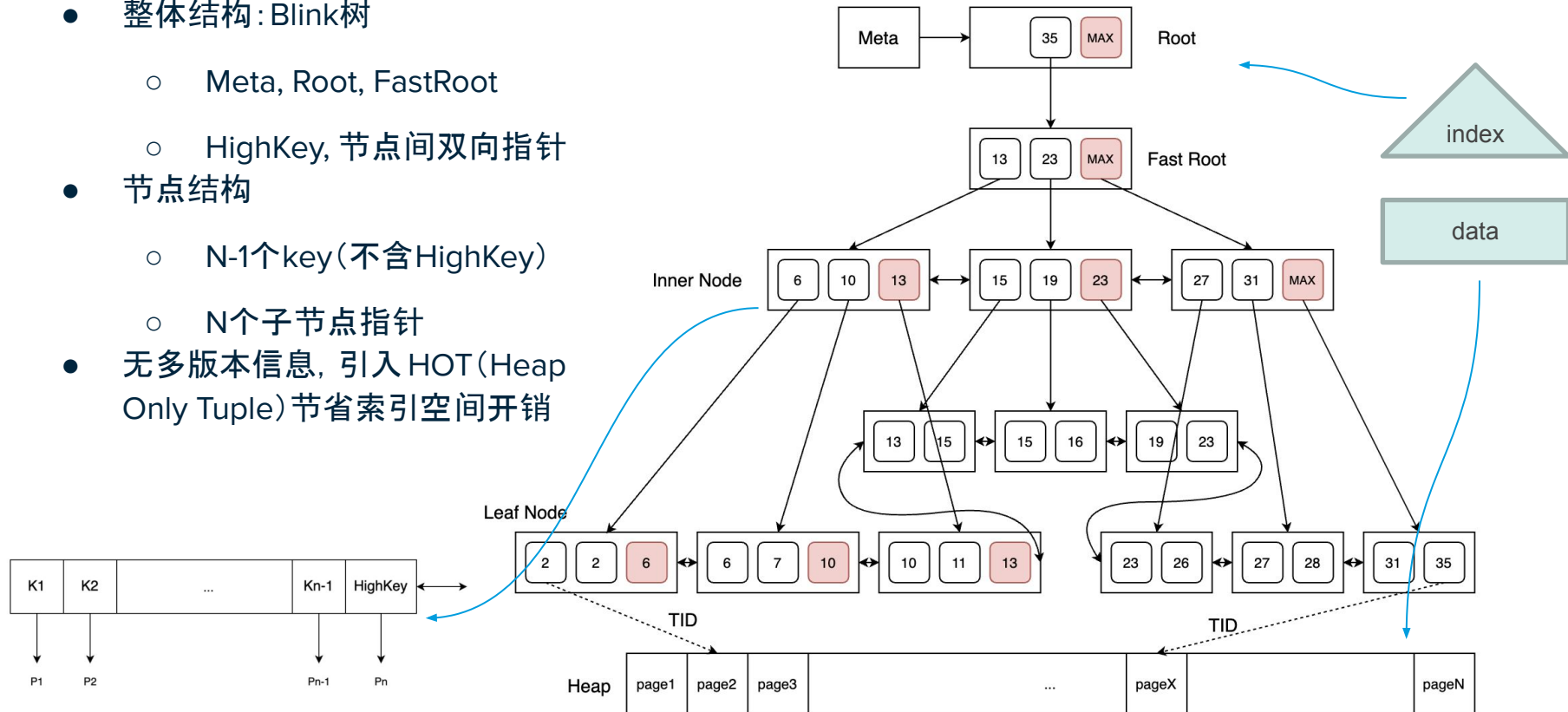
(2 rows)

```
$ find . -name 16524 | xargs ls -l
-rw----- 1 interma staff 1212416 Mar 17 10:44 ./gpseg0/base/16384/16524
-rw----- 1 interma staff 1179648 Mar 17 10:44 ./gpseg1/base/16384/16524
```



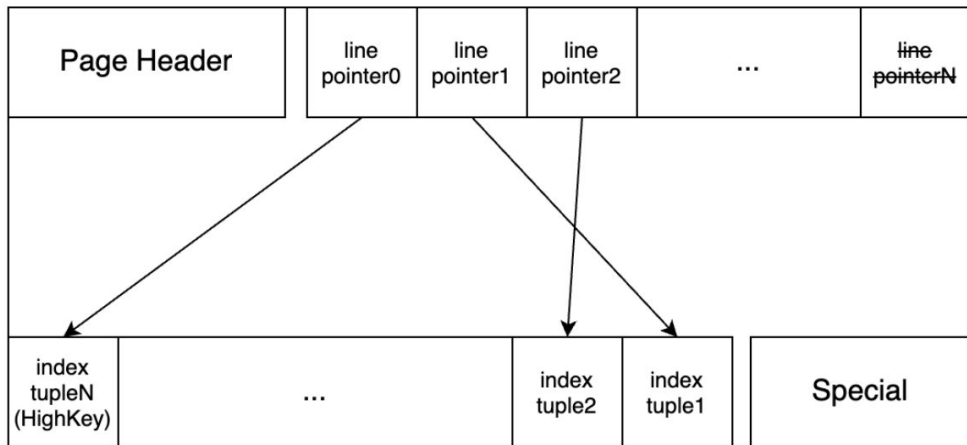
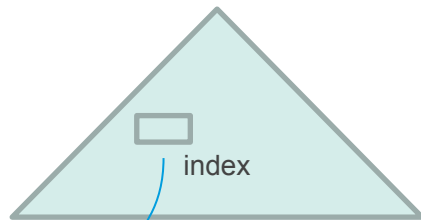
索引结构 – 逻辑结构

- 整体结构: Blink树
 - Meta, Root, FastRoot
 - HighKey, 节点间双向指针
- 节点结构
 - N-1个key(不含HighKey)
 - N个子节点指针
- 无多版本信息, 引入HOT(Heap Only Tuple)节省索引空间开销



索引结构 – 索引节点物理结构

- 每个索引节点对应一个物理页面
- 页内结构和堆表(Heap)的页面结构基本一致
 - 参考: <https://pgconf.ru/media/2016/05/13/tuple-internals.pdf>
- 区别点:
 - 索引元组(key : TID)
 - **Line pointer0指向了HighKey**
 - Special中存储了页面级元信息:
 - 兄弟指针
 - 页面类型
 - 等等



索引结构 – pageinspect示例

Pageinspect是Greenplum的一个扩展, 通过它能查看页面中的内容,
<https://www.postgresql.org/docs/9.5/pageinspect.html>

- **bt_metap** returns information about a B-tree index's metapage
- **bt_page_stats** returns summary information about single pages of B-tree indexes
- **bt_page_items** returns detailed information about all of the items on a B-tree index page

建立测试表

```
demo=# create table test_index (a integer, b text) distributed by (a);  
demo=# insert into test_index(a,b) select s.id, chr((32+random()*94)::integer) from generate_series(1,10000) as s(id)  
order by random();  
demo=# create index on test_index(a);
```

```
demo=# create extension pageinspect;
```

```
PGOPTIONS='-c gp_session_role=utility' psql -p 6000 demo #通过工具模式直接连接到segment上
```

索引结构 – pageinspect示例

Meta页面, 从查询中数据可以看出:

- 这个索引的根层级是1(叶子节点层级是0), 即树的高度为2
- 快速根节点和根节点是同一个节点(块号相同, 均为3)

```
demo=# select * from bt_metap('test_index_a_idx');
 magic | version | root | level | fastroot | fastlevel
```

```
-----+-----+-----+-----+-----+-----
340322 |    2    |    3    |    1    |    3    |    1
```

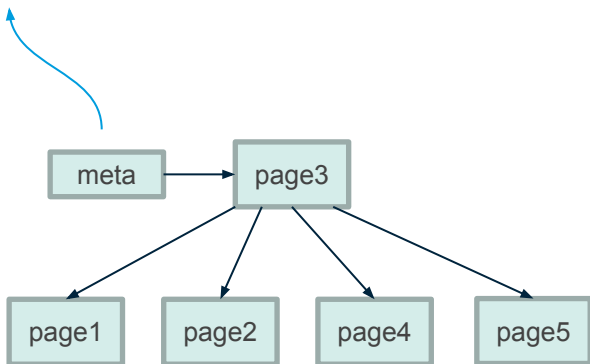


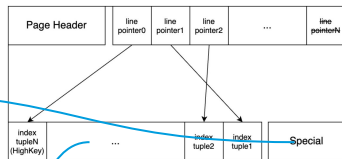
表3-8 BTMetaPageData数据结构

类型	字段	说明
uint32	btm_magic	B-Tree的magic常量
uint32	btm_version	B-Tree的版本常量
BlockNumber	btm_root	根节点的块号
uint32	btm_level	根节点的树层级
BlockNumber	btm_fastroot	快速根节点的块号
uint32	btm_fastlevel	快速根节点的层级

索引结构 – pageinspect示例


根节点(内部节点)

- Stats对应页面中的special结构: BTPageOpaqueData数据结构
 - 各字段的含义如名字, 其中 btpo 表示层号: root节点的层号是1
- Items对应页面中的索引元组: 其中内部节点的第一个索引元组的键值为空
 - 4个索引元组: key => tid



```
demo=# select * from bt_page_stats('test_index_a_idx',3);
 blkno | type | live_items | dead_items | avg_item_size | page_size | free_size | btpo_prev | btpo_next | btpo | btpo_flags
-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----
  3 | r |      4 |         0 |        14 |    32768 |    32652 |         0 |         0 |    1 |          2
(1 row)
```

```
demo=# select * from bt_page_items('test_index_a_idx',3);
 itemoffset | ctid  | itemlen | nulls | vars | data
-----+-----+-----+-----+-----+-----
      1 | (1,1) |      8 | f | f | 
      2 | (2,1) |     16 | f | f | 9d 0b 00 00 00 00 00 00
      3 | (4,1) |     16 | f | f | 23 17 00 00 00 00 00 00
      4 | (5,1) |     16 | f | f | 78 22 00 00 00 00 00 00
(4 rows)
```



4个子节点指针分别指向了1, 2, 4, 5号页面
3个键值的值分别是: 2973, 5923, 8824, 计算过程如下:

```
$ python
Python 2.7.10 (default, Aug 17 2018, 19:45:58)
>>> int('0000000000000b9d', 16)
2973
>>> int('1723', 16) # 去除前缀0
5923
>>> int('2278', 16) # 去除前缀0
8824
```

索引结构 – pageinspect示例

叶子节点 stats

- type都等于l, 表明leaf页; btpo都等于0, 表示第0层
- live_items总共: $1473 \times 3 + 597 = 5016$ 个索引元组, 而另一个segment上有4984个索引元组, 这与表中共10000个记录吻合
- 前3个leaf页面已经填满, 填充率约为 $(1 - 3264 / 32768) = 90\%$, 这也是B-Tree的叶子节点的默认填充因子(内部节点的填充因子是70%)
- btpo_prev和btpo_next页面表明leaf页的顺序是(由左到右): 1 <-> 2 <-> 4 <-> 5

```
demo=# select blkno,type,live_items,page_size,free_size,btpo_prev,btpo_next,btpo from  
bt_page_stats('test_index_a_idx',1) union select blkno,type,live_items,page_size,free_size,btpo_prev,btpo_next,btpo  
from bt_page_stats('test_index_a_idx',2) union ... ; -- 依次再查询4, 5页
```

blkno	type	live_items	page_size	free_size	btpo_prev	btpo_next	btpo
-------	------	------------	-----------	-----------	-----------	-----------	------

1	l	1473	32768	3264	0	2	0
2	l	1473	32768	3264	1	4	0
4	l	1473	32768	3264	2	5	0
5	l	597	32768	20784	4	0	0

(4 rows)

索引结构 – pageinspect示例

叶子节点items

- 第一个条目: ctid=(1,628), 如前边B-Tree逻辑示例图中所述, 每个页面内存储的第一个键值是HighKey, 而从第二个元素开始, 才是真正叶子节点存储的键值
- 同时这也是第二页的第一个普通键值(这里不是巧合, B-Tree构建的细节将在后边小节中详述)。而第一页的第2个键值为2, 它也是这个B-Tree索引的最小值。

```
demo=# select * from bt_page_items('test_index_a_idx',1) limit 5;
```

itemoffset	ctid	itemlen	nulls	vars	data
1	(1,628)	16	f	f	9d 0b 00 00 00 00 00 00
2	(3,613)	16	f	f	02 00 00 00 00 00 00 00
3	(3,430)	16	f	f	03 00 00 00 00 00 00 00
4	(2,534)	16	f	f	04 00 00 00 00 00 00 00
5	(1,372)	16	f	f	06 00 00 00 00 00 00 00

(5 rows)

```
demo=# select * from bt_page_items('test_index_a_idx',2) limit 5;
```

itemoffset	ctid	itemlen	nulls	vars	data
1	(1,312)	16	f	f	23 17 00 00 00 00 00 00
2	(1,628)	16	f	f	9d 0b 00 00 00 00 00 00
3	(3,520)	16	f	f	9f 0b 00 00 00 00 00 00
4	(2,662)	16	f	f	a2 0b 00 00 00 00 00 00
5	(0,369)	16	f	f	a3 0b 00 00 00 00 00 00

(5 rows)

```
demo=# select * from test_index where ctid='(1,628)' and gp_segment_id=0;
```

```
a | b
```

```
-----+-----
```

```
2973 | H
```

```
demo=# select * from test_index where ctid='(3,613)' and gp_segment_id=0;
```

```
a | b
```

```
----+----
```

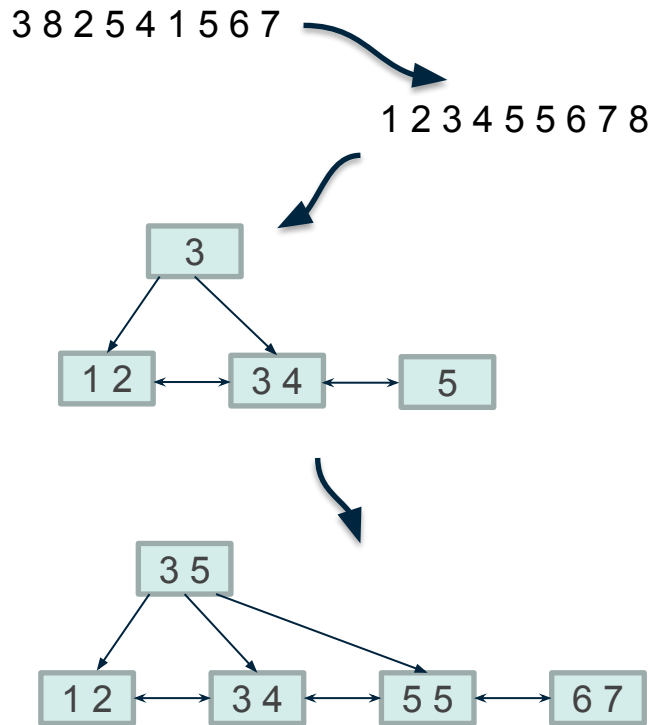
```
2 | #
```

操作算法

- 简要介绍索引的增删改查算法
- 先不考虑并发控制

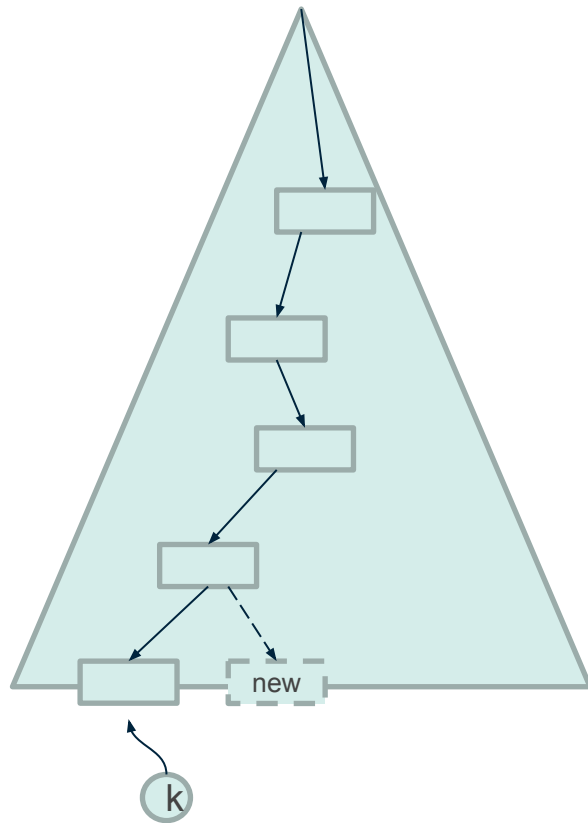
B树算法 – 构建

- 由create index触发, 从表中现有数据创建出一颗B树索引, 算法可以分2大阶段: 排序+构建
- 第一步是将表中数据元组有序化
 - 对比不排序的逐个插入方案: 提升效率
 - 处理唯一索引
- 第二步是遍历有序的数据元组, **由下向上**来构建整个B树
 - 当节点页面已满(实际上还有部分空闲空间): 生成当前节点高键; 生成右兄弟节点; 插入键值到父节点中
 - 最后由下向上补全B树, 并填充meta信息



B树算法 – 插入

- 由insert语句触发: 首先插入数据元组, 然后插入索引元组, 算法如下:
- 首先从根节点开始向下查找, 目标是定位要插入索引元组的叶子节点
 - 记录从根节点到叶子节点的这条路径, 供后续反向插入父节点使用
- 随后在叶子节点中定位要插入的具体偏移位置, 并插入索引元组到这个叶子节点中。
 - 如果叶子节点已满, 则需要分裂节点并插入一个新键值到父节点中, 此过程会沿着查找路径递归向上执行(有可能导致父节点继续分裂)。



B树算法 – 查找/删除

- 查找
 - 普通索引扫描: 非常类似插入算法, 一条下降路径
 - 位图索引扫描: page => bitmap
 - 普通索引扫描执行随机IO: key => TID
 - 于是引入了位图索引扫描, 将随机IO转化为顺序IO
- 删除
 - 非delete语句触发, 而是索引扫描时发现死亡的数据元组后, 对相应的索引元组进行“标记删除”
 - vacuum时进行最终删除, 也是二阶段算法: 首先删除已标记的索引元组, 然后删除空页面并调整树结构(由下向上调整)

并发控制

- 现实中应用的数据库系统, 必须考虑并发控制
- 引入Blink树的原因:《Efficient locking for **concurrent** operations on B-trees》

B树算法 – 并发控制

- 朴素思想: 读节点加读锁, 写节点加写锁, 对分裂操作进行特殊处理
- 朴素的并发控制算法:

```
Search(Key) {  
    currentNode = Root  
    LockRead(currentNode) // 加读锁  
    while (currentNode.level() != LEAF) {  
        idx = SearchNode(currentNode, Key)  
        LockRead(currentNode.Child[idx])  
        UnLock(currentNode) // 释放锁  
        currentNode = currentNode.Child[idx]  
    }  
    tid = GetTID(currentNode, Key)  
    UnLock(currentNode)  
    return tid  
}
```

```
Insert(Key) {  
    currentNode = Root  
    LockWrite(currentNode) // 加写锁  
    while (currentNode.level() != LEAF) {  
        idx = SearchNode(currentNode, Key)  
        LockWrite(currentNode.Child[idx])  
        currentNode = currentNode.Child[idx]  
        if (currentNode.isSafe()) // 判断是否是安全节点  
            UnLockAllParents(currentNode) // 释放各父节点中的锁  
    }  
    InsertAllParents(currentNode, Key) // 插入当前节点, 并递归插入到各父节点中 (如果需要)  
    UnLock(currentNode)  
    UnLockAllParents(currentNode)  
}
```

当在某个节点上插入一个新索引元组后, 不会触发它的分裂, 那么这个节点就叫做安全节点

B树算法 – 并发控制

朴素的并发控制算法

- Search操作由根节点出发, 逐层下降加**读锁**并完成查找操作。留意: 每次只锁当前的一个节点, 先获取当前节点锁, 然后再释放上层锁
- Insert操作同样由根节点出发, 逐层下降加**写锁**并完成插入操作。留意: 下降路径中的一条不安全节点段有可能被锁住(如果存在不安全节点的话)
- 正确性说明
 - 读读操作: 只涉及读锁, 完全并发, 得出正确
 - 写写操作/读写操作: 操作路径每次都是先拿写锁再释放锁, 只可能锁住查找路径上的一条末端路径; 另外涉及到全部操作节点上都加了写锁, 因此不存在并发修改, 得出正确
 - 是否死锁: 加锁顺序是有序的: 都是由根向下, 因此不会死锁
- **问题:** 但由于每次路径下降都需要锁操作, 所以在靠近树根的位置上的锁冲突率较高; 另外在路径下降时加的这些锁, 大概率都会被马上释放掉(无冲突+安全节点)

B树算法 – 并发控制

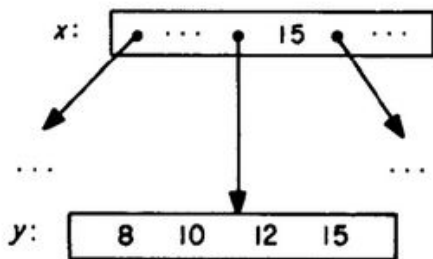
- Blink树并发控制算法, 引入了一个 **moveright**操作, 利用到了 HighKey和右兄弟指针, 用于及时发现节点已经被分裂: 如果分裂, 所查找的键值一定在右兄弟节点上

```
// 右移函数, 执行结束后在查询到的节点上保持读锁
_bt_moveright() {
    while {
        // 修复路径中发现的不完全分裂
        [...]

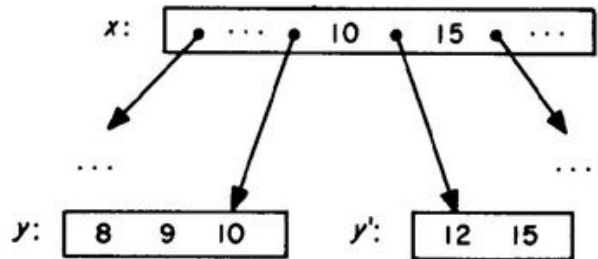
        // ScanKey大于HighKey, 因此需要右移
        if (P_IGNORE(opaque) || _bt_compare(rel, keysz, scankey, page, P_HIKEY)
            >= cmpval)
        {
            // 向右移动一页, 先释放当前节点上的读锁, 再获取右兄弟节点上的读锁
            buf = _bt_relandgetbuf(rel, buf, opaque->btpo_next, access);
            // 继续循环: 可能需要连续右移
            continue;
        }
        else
            break;
    }
}
```

B树算法 – 并发控制

- 通过moveright操作, 可以处理如下这种分裂问题
- 因此可以放松下降过程中的锁操作: Insert操作的下降过程中也加读锁



(a)



```
// 演示“找不到已存在键值”的异常情况  
// 操作A, Search(15)
```

```
node = read(x);  
检查node, 发现15在叶子y节点中
```

```
// y已经分裂, 目前只包含(8,9,10)!  
node = read(y);  
无法在y节点中找到15!
```

```
// 操作B, Insert(9)  
node2 = read(x);
```

检查node2, 发现9应该插入叶子节点y

准备插入9到y中, 但是发现y已经满了
也是将y分裂并插入9:

$y = (8, 9, 10)$; $y' = (12, 15)$

插入y'对应的键值到x中

B树算法 – 并发控制

- Blink树并发控制算法 – Search

- 从根节点开始逐层下降, 加读锁
- 每次下移一层后, 都调用 moveright操作, 检查节点是否分裂
- 在下移或者右移操作时, **都是先释放锁, 然后再去申请新锁**, 即申请新读锁操作时并不持有锁, 因此可以避免死锁(和Insert操作并发时)
- 最后到达叶子节点, 加读锁, 并读取内容

- Blink树并发控制算法 – Insert

- 开始阶段同Search操作一样, **逐层下降加读锁并配合 moveright**, 因此并行佳
- 逐层下降到达叶子节点后, 需要将读锁升级为写锁
- 如果需要节点分裂
 - 新建右兄弟页面, 加写锁, 随后将它挂入到B-Tree中
 - 随后递归向上插入键值: **由下向上为父节点申请写锁**, 随后插入到父节点的操作, 然后再释放下层节点的写锁

B树算法 – 并发控制

- Blink树并发控制算法 – 总结
- 解决了朴素算法的2个问题
 - 树根的位置上的锁冲突率较高 => 读写均加读锁
 - 另外在路径下降时加的这些锁, 大概率都会被马上释放掉 => 下降时加读锁, 写锁由下向上申请
- 死锁考虑
 - Insert操作中写锁的申请顺序都是由下向上, 由左到右, 不会发生死锁
 - Search操作是可能和Insert操作出现不同申请顺序(前者由上向下;后者由下向上), 但是Search操作每次都是先释放再申请, 因此也不会发生死锁

索引相关系统表

- 简介相关系统表
- Greenplum中的类型系统是可以自定义/扩展的, 同一套B树算法如何支持不同的数据 类型?

索引相关系统表

- 类似C++中的泛型算法
 - <https://www.postgresql.org/docs/9.5/xindex.html>
- B树索引的7个接口, 分为2类
 - 策略操作符, 5个
 - 支持函数, 2个
 - 本质上都是自定义函数


Table 35-2. B-tree Strategies

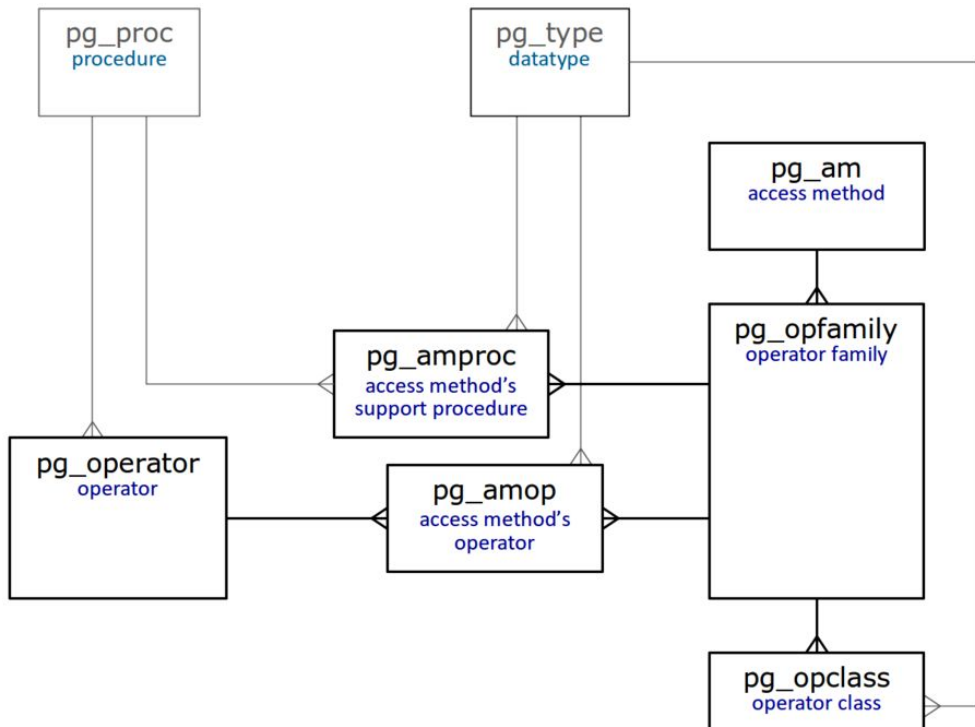
Operation	Strategy Number
less than	1
less than or equal	2
equal	3
greater than or equal	4
greater than	5

Table 35-8. B-tree Support Functions

Function	Support Number
Compare two keys and return an integer less than zero, zero, or greater than zero, indicating whether the first key is less than, equal to, or greater than the second	1
Return the addresses of C-callable sort support function(s), as documented in <code>utils/sortsupport.h</code> (optional)	2

索引相关系统表

- 相关系统表 
- 对外通过2个数据库对象定义
 - Operator class, 同类型上的函数接口
 - Operator family, 跨相近类型上的函数接口
- <https://www.postgresql.org/docs/9.5/xindex.html>



最终目的: 为自定义类型添加索引的支持

- Greenplum中的B树索引实现是非常经典的
 - 一个具体的B树实现
 - Blink树的并发控制算法具有深远的影响
- 数据库索引新发展
 - LSM Tree, 如LevelDB, 聚簇索引(按id有序存储)
 - 全文检索支持: FTS, Greenplum Text
 - 更适合OLAP的索引: 轻量级索引, 统计信息(如Brin index)等
 - 更好的并发控制, 适用多核, 如 Bw-Tree等



GREENPLUM DATABASE®

扫码加入Greenplum技术讨论群



钉钉群: <https://dwz.cn/23XPHVOD>

欢迎访问Greenplum问答平台 ask.greenplum.cn

A group of people in a meeting room. A man on the left is pointing at a whiteboard. Several people are seated in the center, and two more are standing on the right. The room has a whiteboard, a desk with a computer, and a bag on the floor.

Q&A