**Btree索引介绍**

BTree：最常用的索引， BTree索引适合用于处理等值查询和范围查询。 PostgreSQL的查询规划器会在使用以下运算符时考虑使用B-tree索引：<、<=、=、>=、>、BETWEEN、IN、IS NULL、IS NOT NULL。

另外，查询规划器在涉及模式匹配运算符LIKE、~、ILIKE、~\*的查询时考虑使用B-tree索引。

B树具有一些重要的特征：

•B树是平衡的，也就是说，每个叶子节点与根节点都由相同数量的中间节点分隔开。因此，搜索任何值都需要花费相同的时间。

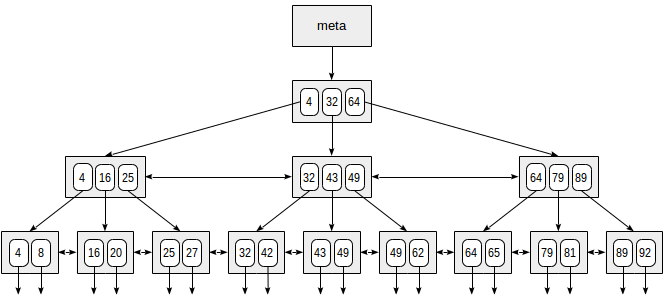
• B树是多分支的，即每个页面（通常为8 KB）包含许多（数百个）ctid。因此，B树的深度很小，对于非常大的表，实际上可以达到4–5的深度。当字段超过单个索引页的1/3时，不适合b-tree索引。

•索引中的数据按非递减顺序排序（在页面之间和每个页面内部），并且同一级别的页面通过双向列表相互连接。

因此，可以仅通过列表的一个方向或另一个方向获得有序数据集，而不必每次都返回到根。

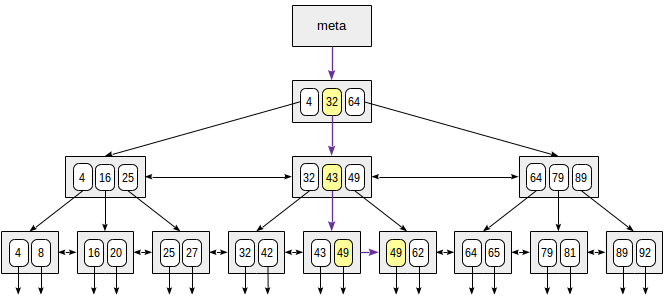
**索引结构**

下面是带整数键的一个字段上的索引的简化示例：



**等值搜索**

通过条件“索引字段=表达式 ” 在树中搜索值。假设对49的键感兴趣。

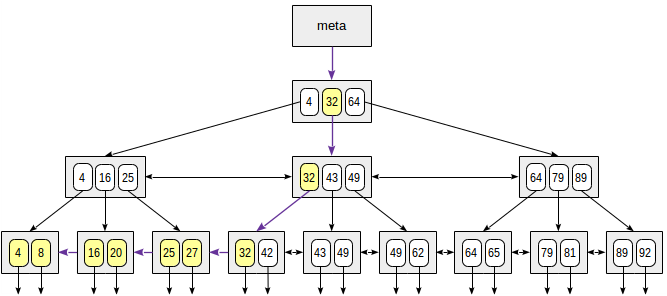


搜索从根节点开始，然后需要确定要降到哪个中间节点。由于知道根节点（4、32、64）中的键，因此可以计算出中间节点的值范围。由于32 < 49 < 64，所以需要下降到第二个中间节点。接下来，递归重复相同的过程，直到到达叶子节点，可以从该叶叶子节点获得所需的TID。

如果一个索引包含非唯一值，并且可能有多个相等的值以至于它们没法存放在一页。因此，一旦在内部页面中找到一个完全相等的值，就必须向左移一个位置，然后从左到右查看底层的索引行以寻找所需的键。

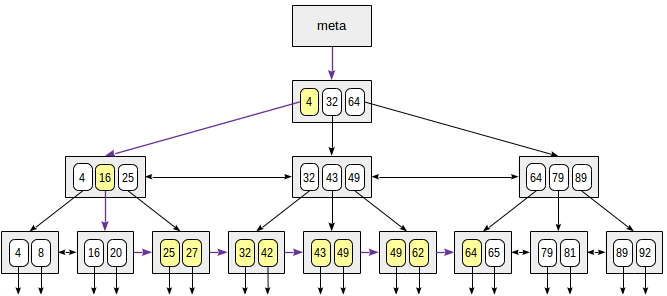
**不等搜索**

当通过条件“索引字段 <= 表达式 ”（或"索引字段 >= 表达式 “）进行搜索时，首先通过等式条件"索引字段=表达式" 在索引中找到一个值（如果有的话），然后按照适当的方向遍历叶子节点直到结束。该图说明了n <= 35的过程：



以类似的方式支持 >= 运算符。
**范围搜索**

示例：23<=n<=64，首先通过条件“n=23”找到一个值，然后遍历叶页面直到“n=64"被满足; 反之亦然：从第二个表达式开始，然后朝相反的方向走，直到到达第一个表达式。



对于任何类型的扫描（索引、仅索引或位图），btree 访问方法都会返回有序数据。因此，如果表在排序条件上具有索引，则优化器将考虑使用表的索引扫描（该索引扫描很容易返回排序后的数据）。

示例

explain select \* from t1 where id = 3000;

explain select \* from t1 where id < 3000;

explain select \* from t1 where id between 3000 and 5000;

**索引排序**

**1.排列顺序**

创建索引时，可以显式地指定排列顺序。

create index on t1(id desc);

在这种情况下，较大的值将出现在左侧的树中，而较小的值将出现在右侧。

**2.列顺序**

使用多列索引时出现的另一个问题是索引中列的顺序。对于B树，此顺序非常重要：页面内的数据将按第一个字段排序，然后按第二个字段排序，以此类推。

**3.空值**

Btree 可以索引NULL，并支持按条件IS NULL和IS NOT NULL进行搜索。

explain  select \* from t1 where age is null;

NULL位于叶节点的一端或另一端，具体取决于创建索引的方式（NULLS FIRST或NULLS LAST）。如果SELECT命令后的ORDER BY子句中指定的NULL顺序与创建索引时指定的顺序相同（NULLS FIRST或NULLSLAST），则可以使用索引。示例：

顺序相同，优化器选择索引扫描

explain select \* from t1 order by id NULLS LAST;--不走索引

顺序不同，优化器选择顺序扫描和排序：

explain select \* from t1 order id NULLS FIRST;--走索引

要使用索引，必须在创建索引时指定 NULLS FIRST：

create index on t1(id NULLS FIRST);

Btree 索引支持两种获取值的技术：索引扫描和位图扫描。访问方法既可以“向前” forward 又可以“向后” backward遍历树。

**内部结构**

**1.B-Tree索引有4种类别的页面：**

meta page：存放的是索引的元数据信息（描述索引本身的信息），每个索引文件的第0页都是meta page

root page：meta page的下一页叫做root page，对于索引量小的情况（一页root page即可存放所有索引数据），只有一个root page

branch page：用于连接root page和leaf page（不存放实际的数据，存放索引数据）

leaf page：存放指向tuple的物理位置

其中mete paga和root page是必备，meta page有个字段指向root page的page no。随着记录的增加，一个root page可能存不下所有的heap items，所以就需要分裂成leaf page,如果继续分裂就存在了branch page和leaf page.

使用 pageinspect 来查看索引的内部结构。

create extension pageinspect;

**2.查看元页面信息：**

select \* from bt\_metap('idx\_t1\_id');

version：版本号

root：root page页的page no（对应bt\_page\_stats.blkno）

level：表示索引的层数，level为0表示只有root page。level=1表示有一个root page和多个leaf page。2表示包含1级的branch和1级leaf,以此类推(存在多级的branch，只有1级的leaf)。

**3.单个页面的信息：**

select \* from bt\_page\_stats('idx\_t1\_id',1);--1代表第1页索引（第0页存放的是meta page，不能用这个函数查看）

blkno：block no，即索引文件的页号

type：索引类型。r:表示root、l:表示leaf、i:表示branch

live\_items：对于leaf page表示有效索引的数量，对于root/branch page表示子索引页的数量

dead\_items：无效索引的数量，或者是无效子索引页

avg\_item\_size：平均字段大小

page\_size：索引页的大小

free\_size：空闲的大小

btpo\_prev：指向同一级的当前索引页的上一页的的page no（同一级索引是双向链表）

btpo\_next: 指向同一级的下一个索引页的的page no，如果没有上一页或者下一页则指向meta page

btpo\_level:表示当前在第几层。最底层为0。

btpo\_flags:指示索引页的类型，0：branch page，1：leaf page，2：root page， 3：既是 root page 又是 leaf page

**4.查看索引页上所有entry（行）的详细信息：**

select \* from bt\_page\_items('t1\_pkey',1);

ctid：如果btpo\_level=0（leaf page）,则值指向tuple的物理位置，可以使用类似select \* from t1 where ctid='(0,1)';进行查询。否则为子索引页的页号（忽略后一项），如(3,1)表示blkno=3的页为子索引页。

 data：

              leaf page存放的是索引数据的值（非最右页的第一项存放的是下一索引页的最小值，最右页第一项存放的是tuple数据），

              root page存放的是子索引页的最小值（第一项为空，因为最左页不存放最小值），

              branch page非最右页第一项存放的是下一个索引页的最小值，第二项为空，最右页第一项为空（因为没有下一个索引页），第二项指向第一个子索引页的最小值

**5.0级索引(int类型 max：407条)**

0层结构，只有meta page和root page。root page最多可以存储的item数，取决于索引字段数据的长度、以及索引页的大小。

create table t1(id int primary key, info text);

insert into t1 select n,md5(n::text) from generate\_series(1,407) n;

select \* from bt\_metap('t1\_pkey');--查看主键索引，显示索引级别level=0:表示只有root层，没有branch和leaf，root=1:表示root的page no=1.（当前页为meta page，是第0页，固定值）

select \* from bt\_page\_stats('t1\_pkey',1);--查看root页的统计信息，live\_items=407表示其索引了407条数据（因为没有子索引页），btpo\_level=0表示已经是最后一层的页，btpo\_flags=3表示既是root页，又是leaf页。

select \* from bt\_page\_items('t1\_pkey',1) limit 5;--查看root页的数据内容，data为索引的值。

select count(\*) from bt\_page\_items('t1\_pkey',1);--root页的确有407条数据，每个索引项一条数据。

insert into t1 select 408, md5(408::text);

select \* from bt\_metap('t1\_pkey');--再插入一条数据，查看索引级别变为了1，说明产生了leaf page。

**6.1级索引（max：149369条）**

包括meta page, root page, leaf page。

create table t1(id int primary key, info text);

insert into t1 select n,md5(n::text) from generate\_series(1,1000) n;--向表中插入1000条数据

select \* from bt\_metap('t1\_pkey');--索引级别level=1:表示有root层和leaf，root=3:表示root的page no=3

select \* from bt\_page\_stats('t1\_pkey',3);--live\_items=3可以看出root页有三个子索引页（此时root本身不在存放索引内容），btpo\_flags=2:表示这个是root层，btpo\_level=1:表示第一层（最底层btpo\_level=0）。

select \* from bt\_page\_stats('t1\_pkey',1);

select \* from bt\_page\_stats('t1\_pkey',2);

select \* from bt\_page\_stats('t1\_pkey',4);--btpo\_flags=1 说明是leaf层，btpo\_level=0 说明是最后一层了，btpo\_next=2 说明下一个page no=2，btpo\_prev=0 说明上一个page no=0,指向meta page。查看页1,2,4分别有367,367,268条数据（多了两条？？）。

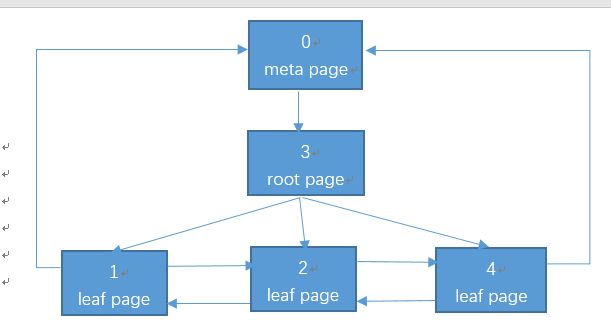
select \* from bt\_page\_items('t1\_pkey',3);--可以看出root page有3条记录，代表有3个子索引页，子索引页的页号为1,2,4（对于非叶子索引，忽略ctid的后一个字段）。data项存放的是子索引页的最小值，第一项（1号页）为空，因为最左页不保存最小值。第二页的最小值为016f=367,第四页的最小值为02dd=733=366+366+1（366为上面的live\_items-1）。

select \* from bt\_page\_items('t1\_pkey',1);--可以看出leaf page页的第一项存放的是页2的最小值016f

select \* from bt\_page\_items('t1\_pkey',2);--页2的第一项存放的是页4的最小值

select \* from bt\_page\_items('t1\_pkey',4);--页4的第一项存放的是正常的数据（因为没有下一页）。

总结：总数据数量367+367+268=1002，因为前两页，分别有一条数据用来保存下一页的最小值。



**7.2级索引（max：4200w左右）**

记录数超过1层结构的索引可以存储的记录数时，会分裂为2层结构，除了meta page和root page，还可能包含1层branch page以及1层leaf page。

create table t2(id int primary key);

insert into t2 select generate\_series(1,1000000);

select \* from bt\_metap('t2\_pkey');--level=2:表示有root、branch、leaf，root=412:表示root的page no=421.

select \* from bt\_page\_stats('t2\_pkey',412);----live\_items=10表示root页有10个子索引页（root本身不在存放索引内容），btpo\_flags=2:表示这个是root层，btpo\_level=2:表示第2层（最底层btpo\_level=0）。

select \* from bt\_page\_items('t2\_pkey',412);----可以看到10条记录，表示root页有10个子索引页，子索引页的页号为3、411、698、984、1270、1556、1842、2128、2414、2700。

select \* from bt\_page\_stats('t2\_pkey',3);--live\_items=286可以看出branch page有286个子索引页（branch page本身不在存放索引内容），btpo\_flags=0表示当前页为branch page，btpo\_level=1表示当前页处于第1层。

select \* from bt\_page\_items('t2\_pkey',3);--查看branch page的索引条目，上述的229条记录代表branch层包含229个page，这些页存放相应的leaf

select \* from bt\_page\_stats('t2\_pkey',1);--btpo\_flags=1 表示leaf，btpo\_prev=0表示最左指向meta page，live\_item=374表示对应到heap的记录

select \* from bt\_page\_items('t2\_pkey',1);

**索引维护**

**amcheck**

postgresql 中有一个很有用用的扩展是 amcheck ，它提供的函数让用户能验证B树索引的逻辑结构一致性，并能够提前检测到故障。如果结构有效，则不会发生错误。

amcheck函数只能由超级用户使用。

create extension amcheck;

bt\_index\_check(index regclass, heapallindexed boolean)

参数heapallindexed表示函数验证堆表的所有元组是否都有对应的索引项存在。

select bt\_index\_check('idx\_t1\_name',false);

bt\_index\_parent\_check(index regclass, heapallindexed boolean, rootdescend boolean)

函数会验证索引结构中的父子关系，会检查父节点是否有缺失。

当可选参数rootdescend值为true时，函数会通过从根页面开始搜索来查找叶子节点的元组。

bt\_index\_parent\_check执行的检查是bt\_index\_check执行检查的超集。

bt\_index\_parent\_check对目标索引及其所属的堆表加锁，会阻塞INSERT、UPDATE以及DELETE的并发数据修改，同时防止所属堆表被VACUUM。

bt\_index\_parent\_check的额外验证有更大可能检测到索引结构的异常，但与bt\_index\_check不同，函数不能在开启hot\_standby的备机上使用。

**索引膨胀**

索引膨胀，主要是针对B-tree而言。B-tree索引页是在整个页完全空了以后才能被重用，比堆表更容易膨胀。

索引膨胀的原因：

•大量删除发生后，导致索引页面稀疏，降低了索引使用效率。

•长时间运行的事务，禁止vacuum对表的清理工作，因而导致页面稀疏状态一直保持。

reindex 是可以解决索引膨胀的，需要注意的是：

•当表的数据小于20万则对表做 reindex index xxx 操作，也可以并发重建

•否则先删除索引，然后再创建该索引

vacuum full也可以解决膨胀问题。