**GIST索引介绍**

GiST：（generalized search tree，通用搜索树），这是一个平衡的搜索树，不是单独一种索引类型，而是一种架构， 可以在这种架构上实现很多不同的索引策略。 GiST索引定义的特定操作符可以用于特定索引策略。用于解决一些B-tree, GIN难以解决的数据相交问题，例如，范围是否相交，是否包含，地理位置中的点面相交，或者按点搜索附近的点。

PostgreSQL的标准发布中包含了用于空间数据类型的GiST操作符类，比如， 一个图形包含另一个图形的操作符“@>”， 一个图形在另一个图形的左边且没有重叠的操作符“<<”， 等等。 GiST索引在空间数据类型和全文检索时很有用。它允许定义规则来将任意类型的数据分布到一个平衡的树中，并且允许定义一个方法使用此表示形式来让某些运算符访问。

对于point类型， 有两种索引操作类“kd\_point\_ops”和“quad\_point\_ops”， 其中quad\_point\_ops是默认的索引操作类， 在某些场景下， kd\_point\_ops索引操作类会更高效一些。

示例：

create table t1(p point);

CREATE INDEX ON t1 USING gist (p);

通常，叶子节点的每一行(leafrow)都包含一些谓词（布尔表达式）和对表行（TID）的引用。索引数据(key)必须符合此谓词。

中间节点的每一行(internal row)还包含一个谓词和对页叶子节点的引用，并且子树的所有索引数据都必须满足此谓词。换句话说，内部行的谓词包括所有子行的谓词。GiST索引的这一重要特征取代了 B-tree的简单排序。

如何搜索？

搜索从根节点开始，检查搜索谓词确定是否需要下降到相应的中间节点，对于叶子节点，只需确定索引的数据是否满足谓词。

GiST索引示例：

（1）点（和其他几何实体）和邻近搜索。

（2）区间和排除约束。

（3）全文搜索。

**GIST应用案例1-R树**

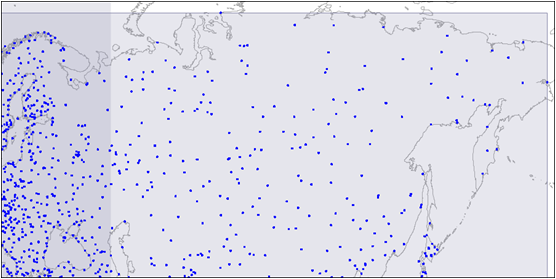
常规的B-tree不适合空间数据类型，因为没有为点定义比较运算符。

R-tree的想法是将平面拆分为矩形，这些矩形总共覆盖所有索引的点。索引行存储一个矩形，可以这样定义谓词：“所寻找的点位于给定的矩形内”。

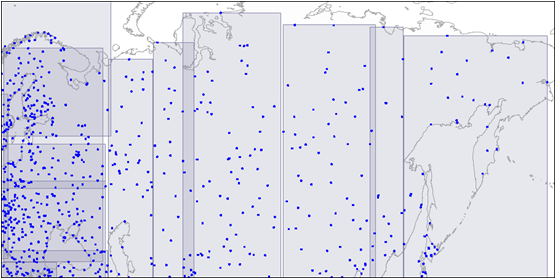
R-tree的根节点将包含几个最大的矩形（可能相交）。叶子节点将包含较小尺寸的矩形，这些矩形嵌入到父节点中，并且覆盖所有基础点。

示例：全球机场坐标图，下面提供了R-tree的三个层次的图像。

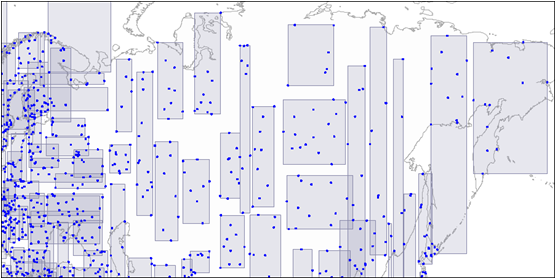
一级：两个大的相交的可见矩形。



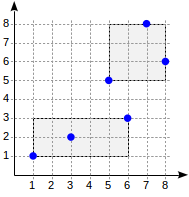
第二级：将大矩形分割成较小的区域。



第三级：每个矩形包含的点数最多可以容纳一个索引页。



“一级”示例：

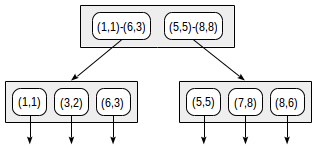


create table points(p point);

insert into points(p) values (point '(1,1)'), (point '(3,2)'), (point'(6,3)'),  (point '(5,5)'), (point '(7,8)'), (point '(8,6)');

create index on points using gist(p);

通过拆分，索引结构将如下所示：



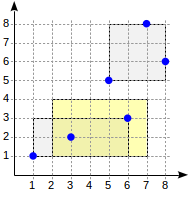
查找给定矩形中包含的所有点：

p <@ box '(2,1),(7,4)'：

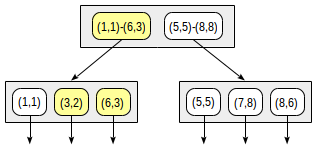
set enable\_seqscan = off;

explain select \* from points where p <@ box '(2,1),(7,4)';  --IndexOnly Scan

其中索引字段是一个点，表达式是一个矩形，对于内部行，如果其矩形与表达式定义的矩形相交，则返回 yes。对于叶子行，如果它的点包含在表达式的定义矩形中，则该函数返回 yes。



搜索从根节点开始。矩形（2,1）-（7,4）与（1,1）-（6,3）相交，但不与（5,5）-（8,8）相交，因此不需要下降到第二个子树。



到达叶节点后，经过其中包含的三个点，并返回其中两个作为结果：（3,2））（6,3）。

select \* from points where p <@ box '(2,1),(7,4)';  -- (3,2) (6,3)

**内部结构**

返回gist索引页面的信息

SELECT \* FROM gist\_page\_opaque\_info(get\_raw\_page('points\_p\_idx', 0));

返回关于存储在gist索引的页面中的数据的信息

SELECT \* FROM gist\_page\_items(get\_raw\_page('points\_p\_idx', 0), 'points\_p\_idx');

与gist\_page\_items相同，但将密钥数据作为原始字节blob返回。由于它不尝试解码密钥，因此不需要知道涉及哪个索引。

SELECT \* FROM gist\_page\_items\_bytea(get\_raw\_page('points\_p\_idx', 0));

**GIST应用案例2-近邻搜索**

<@ 称为搜索运算符，因为它们在查询中指定了搜索条件。

<-> 排序运算符，用于ORDER BY子句中：

select \* from points order by p <-> point '(4,7)' limit 2;  --(5,5) (7,8)

使用排序运算符<->表示从一个点到另一个点的距离。查询的意思是返回最接近点（4,7）的两个点。这样的搜索称为近邻搜索。

为了支持此类查询，必须要指定一个包含排序的运算符类（例如，用于点的 points\_ops 类）。以下查询显示了运算符及其类型：

select amop.amopopr::regoperator, amop.amoppurpose,amop.amopstrategy

from pg\_opclass opc, pg\_opfamily opf, pg\_am am, pg\_amop amop

where opc.opcname = 'point\_ops'

and opf.oid = opc.opcfamily

and am.oid = opf.opfmethod

and amop.amopfamily = opc.opcfamily

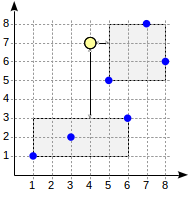
and am.amname = 'gist'

and amop.amoplefttype = opc.opcintype;

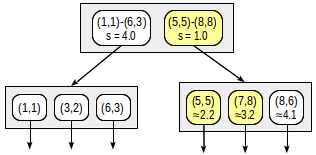
距离函数是为索引元素调用的，它根据排序运算符定义的值来计算到给定元素距离。对于叶子元素，直接计算距离。对于内部元素，计算到子树的最小距离。

点到矩形边界的距离被认为是点到该矩形的最小距离；如果点位于矩形内，则为零。无需遍历子节点即可轻松计算该值，并且该值肯定不大于到任何子树的距离。

搜索从根节点开始。该节点包含两个边界矩形。点 '(4,7)' 到（1,1）-（6,3）的距离是4.0，到（5,5）-（8,8）的距离是1.0。



按照距离的顺序遍历叶子节点，首先下降到最近的叶子节点并计算到这些点的距离：

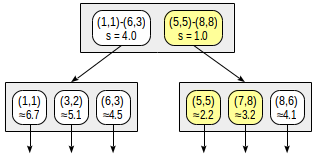


返回前两个点（5,5）和（7,8）。由于知道到矩形（1,1）-（6,3）内的点的距离为4.0或更大，因此不必下降到第一个叶子节点。

如果需要找到前三个点呢？

select \* from points order by p <-> point '(4,7)' limit 3;  -- (5,5) (7,8) (8,6)

尽管第二个叶子节点包含所有这些点，但是如果不查看第一个叶子节点，我们将无法返回（8,6），因为该节点可以包含更近的点（因为4.0 <4.1）。



通过为第二行选择较小的距离（4.0而不是实际的4.5），降低了效率，但并未破坏算法的正确性。

**GIST应用案例3-区间索引**

例如时间区间（“ tsrange”类型）。不同的是，中间节点将包含边界区间，而不是边界矩形。

示例：出租一间小屋，并将预订时间区间存储在一张表中：

create table room(during tsrange);

insert into room(during) values ('[2022-01-01,2022-02-01)'),('[2022-02-01, 2022-03-01)'),('[2022-03-01,2022-04-01)');

create index on room using gist(during);

查看执行计划：

explain  select \* from room where during && '[2022-01-01, 2022-03-12)';  --Index Only Scan 查询返回与给定区间相交的所有区间。

注意：尽管为区间定义了比较运算符，但这不是以一定顺序获取区间。我们可以为区间使用 btree 索引，但是在这种情况下，我们将必须在不支持以下操作的情况下进行操作：

select amop.amopopr::regoperator, amop.amoppurpose,amop.amopstrategy

from pg\_opclass opc, pg\_opfamily opf, pg\_am am, pg\_amop amop

where opc.opcname = 'range\_ops'

and opf.oid = opc.opcfamily

and am.oid = opf.opfmethod

and amop.amopfamily = opc.opcfamily

and am.amname = 'gist'

and amop.amoplefttype = opc.opcintype;

**内部结构**

返回索引页面的信息

SELECT \* FROM gist\_page\_opaque\_info(get\_raw\_page('room\_during\_idx', 0));

返回存储在索引的页面中的数据的信息

SELECT \* FROM gist\_page\_items(get\_raw\_page('room\_during\_idx', 0), 'room\_during\_idx');

SELECT \* FROM gist\_page\_items\_bytea(get\_raw\_page('room\_during\_idx', 0));

**排除约束**

GiST索引可用于支持排他约束（EXCLUDE）。

排他约束确保任何两行的值不能重叠。

排他约束示例：

alter table room add exclude using gist(during with &&);

insert into room(during) values ('[2022-04-01,2022-05-01)');

insert into room(during) values ('[2022-04-15,2022-05-15)');  --ERROR

**btree\_gist 扩展**

在出租的基础上，添加房屋门牌号：

alter table room add house\_no integer default 1;

更改排除约束，以便考虑房屋门牌号。但是，GiST不支持整数的相等运算：

alter table room drop constraint room\_during\_excl;

alter table room add exclude using gist(during with &&,house\_no with =);  --ERROR

解决方案：

安装 btree\_gist 扩展，它添加了对B树的GiST支持，可以让GiST支持任何运算符

create extension btree\_gist;

alter table room add exclude using gist(during with &&,house\_no with =);

insert into room(during, house\_no) values ('[2022-04-15, 2022-05-15)',1);  --ERROR

insert into room(during, house\_no) values ('[2022-04-15, 2022-05-15)', 2);

**GIST应用案例4-RD树**

RD树是将键的HASH签名作为存储键存放在索引里。这类HASH签名算法的查询效率比较差。

create table t5(info text, info\_tsv tsvector);

create index on t5 using gist(info\_tsv);

insert into t5(info) values

('Can a sheet slitter slit sheets?'),

('How many sheets could a sheet slitter slit?'),

('I slit a sheet, a sheet I slit.'),

('Upon a slitted sheet I sit.'),

('Whoever slit the sheets is a good sheet slitter.'),

('I am a sheet slitter.'),  ('I slit sheets.'),

('I am the sleekest sheet slitter that ever slitsheets.'),

('She slits the sheet she sits on.');

update t5 set info\_tsv = to\_tsvector(info);

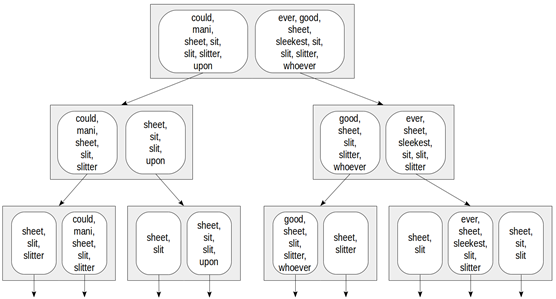
select \* from t5;

索引如何组织？

使用RD树（RD代表“Russian Doll”），RD树的思想是用边界集（即包含子集的所有元素的集）替换边界矩形，集合被理解为词素集合。

如何在索引行中表示集合？

最直接的方法就是枚举集合中的所有元素。



文档签名是所有文档词素按照按位或进行签名。

假设以下词素签名：

could 1000000

ever 0001000

good 0000010

mani 0000100

sheet 0000100

sleekest 0100000

sit 0010000

slit0001000

slitter 0000001

upon 0000010

whoever 0010000

然后，文档签名如下：

Can a sheet slitter slit sheets?   0001101

How many sheets could asheet slitter slit?   1001101

slit a sheet, a sheetI slit.   0001100

Upon a slitted sheet Isit.   0011110

Whoever slit the sheetsis a good sheet slitter.   0011111

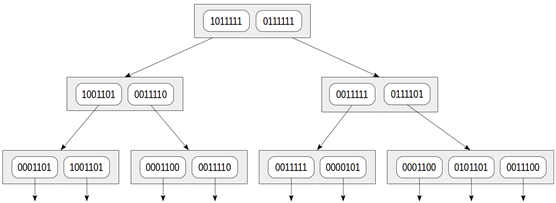
I am a sheetslitter.   0000101

I slit sheets.   0001100

I am the sleekest sheetslitter that ever slit sheets.   0101101

She slits the sheet shesits on.                         0011100

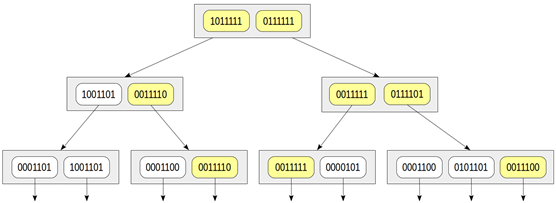
索引树可以表示如下：



这种方法的优点：索引行的大小相等，索引很紧凑。但缺点也很明显：为了紧凑而牺牲了准确性。

info\_tsv @@ to\_tsquery(‘sit’)如何搜索？

先搜索查询的签名：0010000。



树变成了黄色，意味着出现误报，并且在搜索过程中经过了过多的节点。

whoever 词素的签名与 sit 词素的签名相同。

此外，可能会发生这样的情况，不同的文档也将获得相同的签名：I slit a sheet, asheet I slit 和 I slit sheets（都具有0001100的签名）。在这种情况下，将要求索引引擎使用表重新检查结果，因此用户不会看到这些误报。但是搜索效率可能会受到影响。