参考文档：

https://developer.aliyun.com/article/111793

btree

原理

《深入浅出PostgreSQL B-Tree索引结构》

应用场景

b-tree适合所有的数据类型，支持排序，支持大于、小于、等于、大于或等于、小于或等于的搜索。索引与递归查询结合，还能实现快速的稀疏检索

例子

```
postgres=# create table t_btree(id int, info text);
CREATE TABLE
postgres=# insert into t_btree select generate_series(1,10000), md5(random()::text) ;
INSERT 0 10000
postgres=# create index idx_t_btree_1 on t_btree using btree (id);
CREATE INDEX
postgres=# explain (analyze,verbose,timing,costs,buffers) select * from t_btree where id=1;
                                                              QUERY PLAN
---------------------------------------------------------------------------------------------------------------------------------------
 Index Scan using idx_t_btree_1 on public.t_btree  (cost=0.29..3.30 rows=1 width=37) (actual time=0.027..0.027 rows=1 loops=1)
   Output: id, info
   Index Cond: (t_btree.id = 1)
   Buffers: shared hit=1 read=2
 Planning time: 0.292 ms
 Execution time: 0.050 ms
(6 rows)
```

二、hash

应用场景

hash索引存储的是被索引字段VALUE的哈希值，只支持等值查询。

hash索引特别适用于字段VALUE非常长（不适合b-tree索引，因为b-tree一个PAGE至少要存储3个ENTRY，所以不支持特别长的VALUE）的场景，例如很长的字符串，并且用户只需要等值搜索，建议使用hash index。

```
postgres=# create table t_hash (id int, info text);
CREATE TABLE
postgres=# insert into t_hash select generate_series(1,100), repeat(md5(random()::text),10000);
INSERT 0 100

-- 使用b-tree索引会报错，因为长度超过了1/3的索引页大小
postgres=# create index idx_t_hash_1 on t_hash using btree (info);
ERROR:  index row size 3720 exceeds maximum 2712 for index "idx_t_hash_1"
HINT:  Values larger than 1/3 of a buffer page cannot be indexed.
Consider a function index of an MD5 hash of the value, or use full text indexing.

postgres=# create index idx_t_hash_1 on t_hash using hash (info);
CREATE INDEX

postgres=# set enable_hashjoin=off;
SET
postgres=# explain (analyze,verbose,timing,costs,buffers) select * from t_hash where info in (select info from t_hash limit 1);
                                                              QUERY PLAN

-----------------------------------------------------------------------------------------------------------------------------
 Nested Loop  (cost=0.03..3.07 rows=1 width=22) (actual time=0.859..0.861 rows=1 loops=1)
   Output: t_hash.id, t_hash.info
   Buffers: shared hit=11
   ->  HashAggregate  (cost=0.03..0.04 rows=1 width=18) (actual time=0.281..0.281 rows=1 loops=1)
         Output: t_hash_1.info
         Group Key: t_hash_1.info
         Buffers: shared hit=3
         ->  Limit  (cost=0.00..0.02 rows=1 width=18) (actual time=0.012..0.012 rows=1 loops=1)
               Output: t_hash_1.info
```

```
29                Buffers: shared hit=1
30                ->  Seq Scan on public.t_hash t_hash_1  (cost=0.00..2.00 rows=100
  width=18) (actual time=0.011..0.011 rows=1 loops=1)
31                      Output: t_hash_1.info
32                      Buffers: shared hit=1
33    ->  Index Scan using idx_t_hash_1 on public.t_hash  (cost=0.00..3.02 rows=1 width=22)
   (actual time=0.526..0.527 rows=1 loops=1)
34          Output: t_hash.id, t_hash.info
35          Index Cond: (t_hash.info = t_hash_1.info)
36          Buffers: shared hit=6
37  Planning time: 0.159 ms
38  Execution time: 0.898 ms
39 (19 rows)
```

三、gin

原理

gin是倒排索引，存储被索引字段的VALUE或VALUE的元素，以及行号的list或tree。

（ col_val:(tid_list or tid_tree)， col_val_elements:(tid_list or tid_tree) ）

《PostgreSQL GIN索引实现原理》

《宝剑赠英雄 - 任意组合字段等效查询, 探探PostgreSQL多列展开式B树 (GIN)》

应用场景

1、当需要搜索多值类型内的VALUE时，适合多值类型，例如数组、全文检索、TOKEN。（根据不同的类型，支持相交、包含、大于、在左边、在右边等搜索）

2、当用户的数据比较稀疏时，如果要搜索某个VALUE的值，可以适应btree_gin支持普通btree支持的类型。（支持btree的操作符）

3、当用户需要按任意列进行搜索时，gin支持多列展开单独建立索引域，同时支持内部多域索引的bitmapAnd, bitmapOr合并，快速的返回按任意列搜索请求的数据。

四、gist

应用场景

GiST是一个通用的索引接口，可以使用GiST实现b-tree, r-tree等索引结构。

不同的类型，支持的索引检索也各不一样。例如：

1、几何类型，支持位置搜索（包含、相交、在上下左右等），按距离排序。

2、范围类型，支持位置搜索（包含、相交、在左右等）。

3、IP类型，支持位置搜索（包含、相交、在左右等）。

4、空间类型（PostGIS），支持位置搜索（包含、相交、在上下左右等），按距离排序。

5、标量类型，支持按距离排序。

《PostgreSQL 百亿地理位置数据 近邻查询性能》

例子

1- 几何类型检索

```
postgres=# create table t_gist (id int, pos point);
CREATE TABLE
postgres=# insert into t_gist select generate_series(1,100000),
point(round((random()*1000)::numeric, 2), round((random()*1000)::numeric, 2));
INSERT 0 100000
postgres=# select * from t_gist  limit 3;
 id |       pos
----+-----------------
  1 | (325.43,477.07)
  2 | (257.65,710.94)
  3 | (502.42,582.25)
(3 rows)
postgres=# create index idx_t_gist_1 on t_gist using gist (pos);
CREATE INDEX

postgres=# explain (analyze,verbose,timing,costs,buffers) select * from t_gist where
circle '((100,100) 10)'  @> pos;
                                                          QUERY PLAN


-------------------------------------------------------------------------------------------
-------------------------------
 Bitmap Heap Scan on public.t_gist  (cost=2.55..125.54 rows=100 width=20) (actual
time=0.072..0.132 rows=46 loops=1)
   Output: id, pos
   Recheck Cond: ('<(100,100),10>'::circle @> t_gist.pos)
   Heap Blocks: exact=41
   Buffers: shared hit=47
   ->  Bitmap Index Scan on idx_t_gist_1  (cost=0.00..2.53 rows=100 width=0) (actual
time=0.061..0.061 rows=46 loops=1)
         Index Cond: ('<(100,100),10>'::circle @> t_gist.pos)
         Buffers: shared hit=6
 Planning time: 0.147 ms
 Execution time: 0.167 ms
```

```
(10 rows)

postgres=# explain (analyze,verbose,timing,costs,buffers) select * from t_gist where circle '((100,100) 1)' @> pos order by pos <-> '(100,100)' limit 10;
                                                              QUERY PLAN
                                                              
------------------------------------------------------------------------------------------------------------------------------------------------
 Limit  (cost=0.28..14.60 rows=10 width=28) (actual time=0.045..0.048 rows=2 loops=1)
   Output: id, pos, ((pos <-> '(100,100)'::point))
   Buffers: shared hit=5
   ->  Index Scan using idx_t_gist_1 on public.t_gist  (cost=0.28..143.53 rows=100 width=28) (actual time=0.044..0.046 rows=2 loops=1)
         Output: id, pos, (pos <-> '(100,100)'::point)
         Index Cond: ('<(100,100),1>'::circle @> t_gist.pos)
         Order By: (t_gist.pos <-> '(100,100)'::point)
         Buffers: shared hit=5
 Planning time: 0.092 ms
 Execution time: 0.076 ms
(10 rows)
```

五、sp-gist

SP-GiST类似GiST，是一个通用的索引接口，但是SP-GIST使用了空间分区的方法，使得SP-GiST可以更好的支持非平衡数据结构，例如quad-trees, k-d tree, radis tree.

《Space-partitioning trees in PostgreSQL》

《SP-GiST for PostgreSQL User Manual》

应用场景

1、几何类型，支持位置搜索（包含、相交、在上下左右等），按距离排序。

2、范围类型，支持位置搜索（包含、相交、在左右等）。

3、IP类型，支持位置搜索（包含、相交、在左右等）。

六、brin-块级索引

BRIN 索引是块级索引，有别于B-TREE等索引，BRIN记录并不是以行号为单位记录索引明细，而是记录每个数据块或者每段连续的数据块的统计信息。因此BRIN索引空间占用特别的小，对数据写入、更新、删除的影响也很小。

BRIN属于LOSSLY索引，当被索引列的值与物理存储相关性很强时，BRIN索引的效果非常的好。

例如时序数据，在时间或序列字段创建BRIN索引，进行等值、范围查询时效果很棒。

应用场景

《BRIN (block range index) index》
《PostgreSQL 物联网黑科技 - 瘦身几百倍的索引(BRIN index)》
《PostgreSQL 聚集存储 与 BRIN索引 - 高并发行为、轨迹类大吞吐数据查询场景解说》
《PostgreSQL 并行写入堆表，如何保证时序线性存储 - BRIN索引优化》

七、rum-- GIN增强版本

https://github.com/postgrespro/rum
rum 是一个索引插件，由Postgrespro开源，适合全文检索，属于GIN的增强版本。
增强包括：
1、在RUM索引中，存储了lexem的位置信息，所以在计算ranking时，不需要回表查询（而GIN需要回表查询）。
2、RUM支持phrase搜索，而GIN无法支持。
3、在一个RUM索引中，允许用户在posting tree中存储除ctid（行号）以外的字段VALUE，例如时间戳。
这使得RUM不仅支持GIN支持的全文检索，还支持计算文本的相似度值，按相似度排序等。同时支持位置匹配，例如（速度与激情，可以采用"速度" <2> "激情" 进行匹配，而GIN索引则无法做到）

应用场景

《PostgreSQL 全文检索加速 快到没有朋友 - RUM索引接口(潘多拉魔盒)》
《从难缠的模糊查询聊开 - PostgreSQL独门绝招之一 GIN , GiST , SP-GiST , RUM 索引原理与技术背景》
《PostgreSQL结合余弦、线性相关算法 在文本、图片、数组相似 等领域的应用 - 3 rum, smlar应用场景分析》

八、bloom

原理
bloom索引接口是PostgreSQL基于bloom filter构造的一个索引接口，属于lossy索引，可以收敛结果集(排除绝对不满足条件的结果，剩余的结果里再挑选满足条件的结果)，因此需要二次check，bloom支持任意列组合的等值查询。
bloom存储的是签名，签名越大，耗费的空间越多，但是排除更加精准。有利有弊。

应用场景

bloom索引适合多列任意组合查询。

九、zombodb

原理

zombodb是PostgreSQL与ElasticSearch结合的一个索引接口，可以直接读写ES。

https://github.com/zombodb/zombodb

应用场景

与ES结合，实现SQL接口的搜索引擎，实现数据的透明搜索。