**TID扫描**

PostgreSQL 自带的表是堆表，数据按行存储在HEAP PAGE中，在btree索引中，除了存储字段的键，还会存储对应的ctid(即物理位置)，检索记录也是通过ctid进行检索的。

因此通过ctid是可以快速检索到记录的。

ctid的写法是(page\_number, item\_number)，数据块从0开始编号，行号从1开始编号。

例子 :

查找0号数据块的第10条记录，走tid扫描，是非常快的，因为已经给出了page号和item号，直接定位block和item。

select \* from t1 where ctid='(0,10)'::tid;
explain select \* from t1 where ctid='(0,10)';

要使用tid进行快速的行扫描，必须开启参数enable\_tidscan。否则就会走全表扫描的哦，那是非常慢的。

set enable\_tidscan=off;

explain select \* from t1 where ctid='(0,10)'::tid;

**顺序扫描**

执行计划使用Seq Scan表示

在PG中也称为顺序扫描 或 全表扫描，把表的所有数据块从头到尾读一遍，然后从数据块中找到符合条件的数据块。

什么情况下会走顺序扫描？

**1.没有索引**

explain select \* from t1 where id=100;

**2.数据量小**，因为 index scan 至少要发生两次I/O，一次是 读取索引块， 一次是读取数据块。

select relpages, reltuples from pg\_class where relname = 't1';

relpages | reltuples

----------+-----------

1 | 100 |

数据量为 100条记录。

预估成本：

set enable\_seqscan=false;

explain select \* from t1 where id=50;

set enable\_seqscan=true;

explain select \* from t1 where id=50;

数据量大的时候：

数据为1000条记录时，已经跨越了7个page：

analyze;

select relpages, reltuples from pg\_class where relname = 't1';

relpages | reltuples

----------+-----------

7 | 1000

再次预估成本，此时seq scan 已经开始变得不划算了：

set enable\_seqscan=false;
explain select \* from t1 where id=50;

set enable\_seqscan=true;
explain select \* from t1 where id=50;

set enable\_indexscan=false;
explain select \* from t1 where id=50;

set enable\_bitmapscan=false;
explain select \* from t1 where id=50;

**3.条件选择性不好**

explain select \* from t1 where age <=28;

在机械硬盘上，顺序扫描比随机扫描更快，因为将磁头带到轨道的机械操作比读取数据本身花费更多的时间。SSD的效果不太明显，因为随机读写性能都很好。有两个参数可用于考虑访问成本的差异，seq\_page\_cost和random\_page\_cost，我们不仅可以在全局设置，还可以在表空间级别设置，这种方法可以调整不同磁盘子系统的特性。

**索引扫描**

执行计划使用Index Scan表示

首先在索引中找到需要的数据行的物理位置，然后再到表的数据块中把对应数据读出来的过程，也就是先查索引找到匹配记录的ctid，再通过ctid查堆表。

示例：

**等值查询：**

explain  select \* from t1 where id = 1;

**范围查询：**

explain select \* from t1 where id <=500;

是否走索引和存储相关性有关

select attname, correlation from pg\_stats where tablename = 't1';

**位图扫描**

位图扫描也是索引扫描的一种方式，先扫描索引把符合条件的记录的ctid在内存中构建一个位图，然后根据位图到把对应的数据读出来，每个页面只读取一次，避免重复访问相同的数据页。

Bitmap Index相对于Index Scan的劣势在于，它必须建立一个 bitmap, 而 bitmap会产生一些代价。但Bitmap Index的好处在于它把对于数据存储的随机读取变成了顺序读取，在数据量比较大的时候具有一定优势。

如果每次Bitmap Index Scan都建立一个无损的bitmap，bitmap可能非常大。而bitmap是一个内存中的数据结构，如果太大将导致内存不够用，因此bitmap是可以压缩的，压缩的方式也十分简单。当一个block被命中的Tuple比较多时，就可以直接用一个bit来代表整个block，从而把 bitmap压缩到很小。

如果走两个索引，可以把两个索引的位图进行and或者or计算合并成一个位图，再到表中把对应的数据读出来。

执行计划名称说明：

Bitmap Index Scan：在索引中查找符合条件的行，在内存建立位图

Bitmap Heap Scan：根据位图再到表中扫描

Recheck Cond：当从索引中找到行从表中读出来，还需要检查下条件

BitmapOr/BitmapAnd：位图合并

当只处理几个值时，优化器会走索引扫描。但是，随着检索到的行数增加，很有可能多次返回同一个表的页。因此，优化器切换到位图扫描。

explain select \* from t1 where id <= 100;

1.返回所有与条件匹配的TID（位图索引扫描），并从这些TID构建行的位图。

2.从表中读取行（位图堆扫描）。

注意：在第二步中，可能会重新检查条件（Recheck Cond）。

如果对多个表字段施加条件并对这些字段建立索引，位图扫描允许同时使用多个索引。对于每个索引，构建行的位图，然后执行位图合并（AND连接 或 OR连接））。例如：

explain select \* from t1 where id <= 100 and age <= 22;

这里BitmapAnd节点通过按位与操作连接两个位图。

但是如果表页面中数据的物理排序方式与索引记录完全相同，优化器会怎么选择呢？  --走索引扫描

但是这种情况非常少，如果在执行CLUSTER命令之后不进行更改。在这种情况下，构建位图是一个过度的步骤，常规索引扫描也同样好。因此，在选择索引时，优化器会查看一个特殊的统计信息，该统计信息显示了物理行排序和列值逻辑排序之间的相关性:

select attname, correlation from pg\_stats where tablename = 't1';

接近1的值表示高相关性（对于列id），而接近于0的值则相反，表示随机分布（对于列age）。

**仅索引扫描**

如果索引字段中包含了所有返回字段，并且在可见性映射 (vm)表中所查记录为可见的数据，不查堆表直接返回索引中的值。

通常，索引的主要任务是返回匹配行的标识符，以便从这些行读取数据。但是，如果索引已包含查询所需的所有数据呢？这样的索引称为覆盖，在这种情况下，优化器可以应用仅索引扫描：

explain select id from t1 where id < 100;

仅索引扫描可能让人觉得数据库根本不会去访问表，而是仅从索引中获取所有必要信息。但事实并非如此，因为PostgreSQL中的索引并不存储能够判断行可见性的信息。因此，仅索引扫描只会返回与搜索条件匹配的行版本，而不考虑它们在当前事务中的可见性。

但是，如果数据库每次都需要查看表以获得可见性，那么这种扫描方法与常规索引扫描没有什么不同。为解决这个问题，PostgreSQL为表维护了一个所谓的可见性映射表，在这个映射表中，记录着行对当前事务是否可见的信息，如果可见，则可以避免访问该表。

Index-Only扫描有些时候比普通Index扫描还慢 ，尤其当无法通过可见性映射判断结果记录是否可见时（原因是未及时执行vacuum更新可见性映射表）。

因此，定期vacuum可以提高覆盖指数的效率。此外，优化器会考虑死元组的数量，如果预测可见性检查的开销很高，则可以决定不使用仅索引扫描。我们可以使用EXPLAIN ANALYZE命令了解对表的访问次数：

explain analyze select age from t where a < 100;

在这种情况下，不需要访问表（Heap Fetches：0）。一般来说，这个数字越接近零越好。