针对ams3.8的在获取推荐列表的执行sql慢，原因分析与查询

1.执行sql

SELECT software.id AS softwareId ,software.NAME AS NAME

,software.package\_name AS packageName ,software.tag AS tag

,software.tag\_end\_time AS tagEndTime,software.adjust\_download AS adjustDownload

,software\_version.size AS size ,software.download AS download ,software\_version.version AS versionCode,software.star AS star,software\_version.NAME AS versionName

,software.update\_time AS updateTime ,software\_version.icon AS icon

FROM (

SELECT package\_name AS packageName,count(id) AS installCount

FROM installed\_app

WHERE mac IN (

SELECT mac FROM installed\_app

WHERE package\_name = 'air.dzg.android.common.A123qibu.com')

GROUP BY package\_name

) ins

INNER JOIN software ON packageName = package\_name

INNER JOIN software\_version ON software\_id = software.id

WHERE software.STATUS = 1

AND software\_version.STATUS = 1

AND software\_version.is\_new = 1

AND packageName NOT IN (

'air.dzg.android.common.A123qibu.com'

,'com.chinanetcenter.appmarket'

)

ORDER BY installCount DESC limit 6 offset 0

2.采用explain来分析



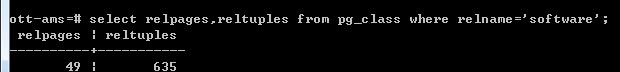
2.1 Cost

cost:表示评估的花费，是以plan cost constants，计算出来的值

seq\_page\_cost：连续块扫描操作的单个块的cost. 例如全表扫描

random\_page\_cost： 随机块扫描操作的单个块的cost. 例如索引扫描  
cpu\_tuple\_cost：处理每条记录的CPU开销（tuple：关系中的一行记录）  
cpu\_index\_tuple\_cost：扫描每个索引条目带来的CPU开销  
cpu\_operator\_cost：操作符或函数带来的CPU开销.(需要注意函数以及操作符对应的函数的三态, 执行计划会根据三态做优化, 关系到多条记录时三态对应的调用次数是需要关心的)

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



可以看出relpages磁盘页数有49，条数635条的记录。

2.1.1 seq scan的cost(全表扫描)

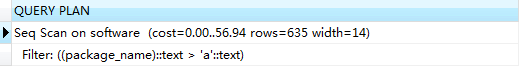
select name from software



cost = relpages\*1.0(seq\_page\_cost的值) + reltuples\*0.01(cpu\_tuple\_cost)=49 + 635\*0.01=55.35

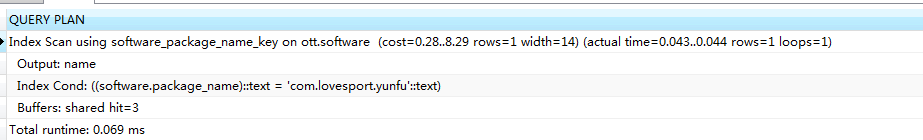
2.1.2 加上条件扫描，得出的cost值，会偏大一些

select name from software where package\_name > ‘a’;



cost = 2.1 + cpu\_operator\_cost \* 条数 = 55.35 + 635\*0.0025= 56.94

2.1.3 索引扫描(Index Scan)



select \* from bt\_metap(‘key’);//无法执行，报错

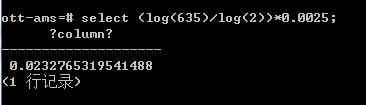
查看索引的深度，但是就是报错。还不知道为什么？？

问题原因：由于没有将pageinspect加载进来

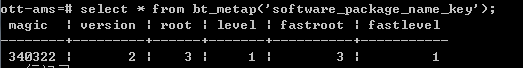
解决方法：create extension pageinspect;

图2.1.3—1 bt\_metap

1.计算ceil(log(数据条数) / log(2)) \* 0.0025(cpu\_operator\_cost) = 0.023276，ceil向上取整数，得10\*0.0025=0.025



2.可以看出software\_package\_name\_key的level为1。(1+1)\*50\*0.0025=0.25



3.计算出indexStartupCost=(0.25+0.025)=0.275=0.28

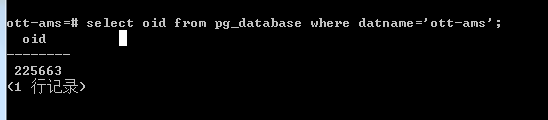
rows:该搜索计划输出的行数

width:该搜索计划输出的字节数

查看数据库的占用空间

通过本地文件查看(有很多系统表中oid字段默认是隐藏的)

1.可以在postgresql的安装目录下,data/base的文件中查看。首先查看数据库对应的oid



2.到postgresql/data/base/225663中查看

3.通过select \*oid from pg\_class 中对应的relfilenode就是数据库目录下对应的表文件名称。刚开始以为oid就是对应的文件名，但是经过了vacuum full之后，oid不变，但是relfilenode的值却变了

通过数据库语句可以查看大小

select pg\_size\_pretty(pg\_database\_size('ott-ams'));

pg\_relation\_size(table\_name)可以查看表空间

pg\_column\_size(column) from table;可以查看列的空间

# B-Tree Index

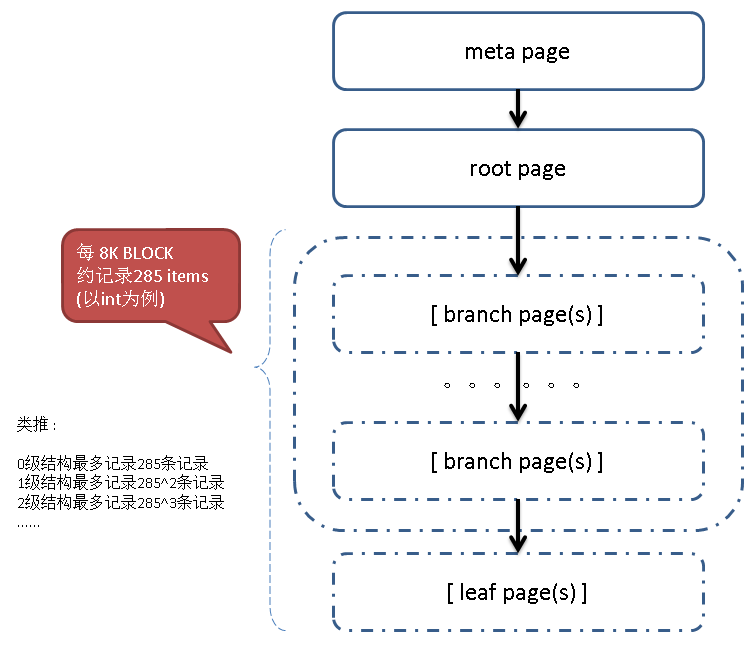
meta page

root page # btpo\_flags=2

branch page # btpo\_flags=0

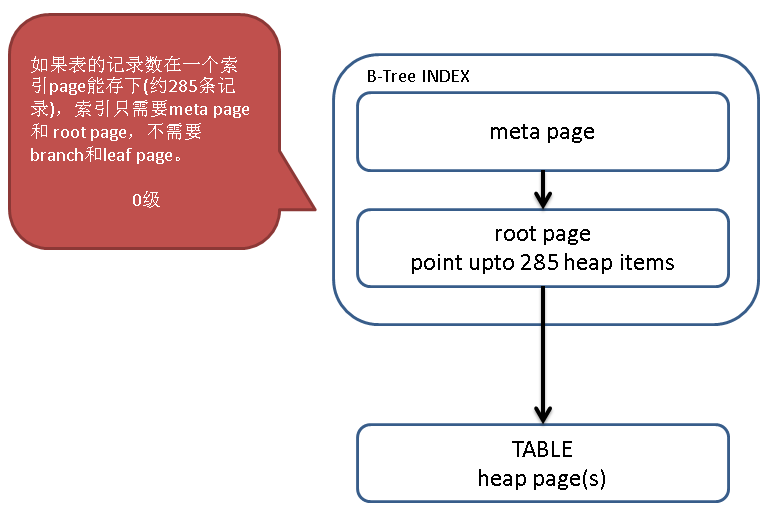
leaf page # btpo\_flags=1

如果即是leaf又是root则 btpo\_flags=3。



0层结构

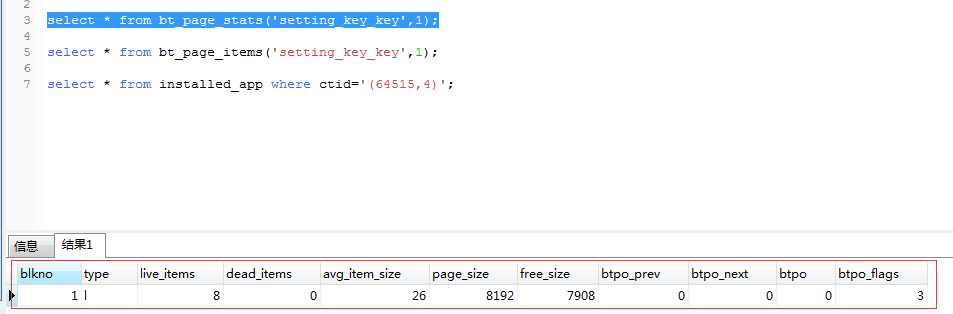
0层结构，只有meta和root页。   
root页最多可以存储的item数，取决于索引字段数据的长度、以及索引页的大小。



1.通过bt\_metap的方法，查看level=0，那么说明没有branch和page，也可以看出root的位置是在1



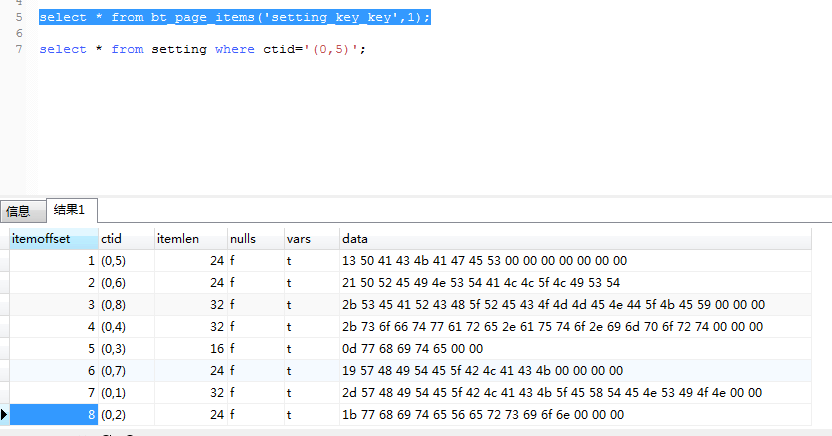
2.通过bt\_page\_stats来查看btpo=0，表示该页为最底层和btpo\_flags=3，表示是root也是leaf,live\_items表示活跃记录数8，所以有8条的



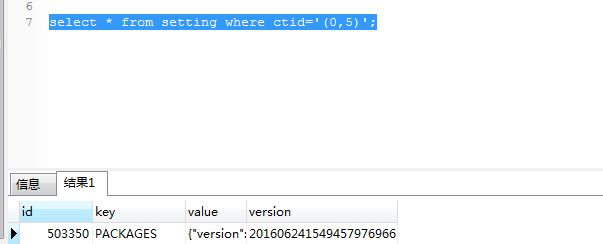
3.通过bt\_page\_items，可以具体查看items的ctid，

0级ctid 表示存储的是 heap页的寻址。 （如果是多层结构，那么branch page中的ctid， 它表示的是同级btree页(链条项ctid)或者下级btree页的寻址） 。   
当ctid指向heap时， data是对应的列值。(多级结构的data意义不一样，后面会讲)

这里的ctid的对应的记录，就是排序的，按照索引创建的索引(非最右页的第一条都是右页第一条记录的地址)



4.最后可通过，select \* from setting where cid=’’，来得到记录

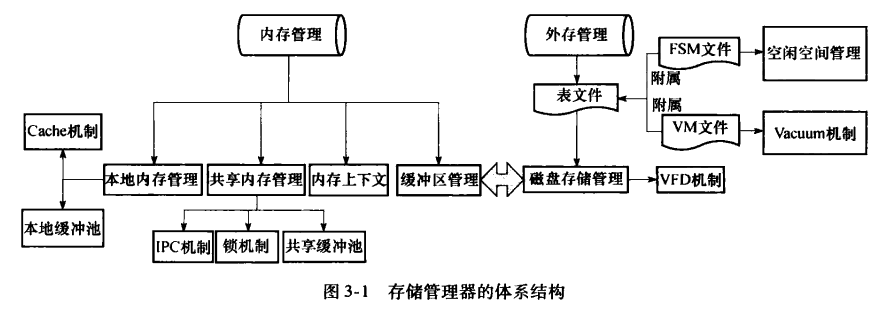


cost\_index is only responsible for the heap-access part of the charges.

The index-access part is in btcostestimate and genericcostestimate in

utils/adt/selfuncs.c.

# Caching



查看查询计划中，有多少记录是命中缓存的。

pg\_buffercache(共享内存)，属于上面的哪一块呢？ 属于共享内存，所有数据库共享

pg\_prewarm预加载(os cache、数据库缓存)

pg\_fincore加载数据到os cache

CREATE EXTENSION pg\_buffercache;

SELECT

C.relname,

COUNT(CASE WHEN B.isdirty THEN 1 ELSE NULL END) As dirty\_buffers,

COUNT(\*) As num\_buffers

FROM

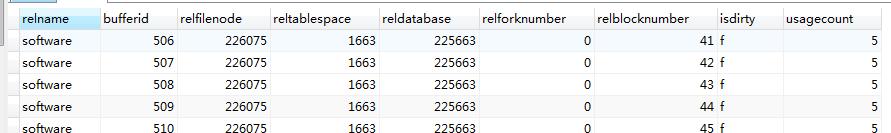
pg\_class AS C INNER JOIN

pg\_buffercache B ON C.relfilenode = B.relfilenode INNER JOIN

pg\_database D ON B.reldatabase = D.oid AND D.datname = current\_database()

WHERE C.relname IN ('software')

从上面的查询中，可以看出在pg\_buffercache有多少条缓存记录,其中relblocknumber就是快的id(也就是blockId)，relforknumber就是行的id(itemId，下标从0开始)。那么组成的ctid的值(blockId.itemId)



show shared\_buffers;用于查询缓存的占用空间

bt\_metap、bt\_page\_status，这些信息是存在哪张表里面呢？

# 扫描方式：

## index scan

当数据量分布<1.7%的时候，采用该索引

## bitmap index scan

当数据量分布>1.7%，则采用该索引

## seq scan

当数据量占用在70%以上直接使用了

# 对于Join的扫描

EXPLAIN (analyse,BUFFERS )

SELECT

this\_. ID AS id17\_3\_,

this\_.create\_time AS create2\_17\_3\_,

this\_.create\_user AS create3\_17\_3\_,

this\_.update\_time AS update4\_17\_3\_,

this\_.update\_user AS update5\_17\_3\_,

this\_.app\_id AS app6\_17\_3\_,

this\_.bind\_time AS bind7\_17\_3\_,

this\_.package\_id AS package8\_17\_3\_,

this\_.provider\_id AS provider9\_17\_3\_,

this\_.status AS status17\_3\_,

this\_.video\_id AS video11\_17\_3\_,

video1\_. ID AS id22\_0\_,

video1\_.create\_time AS create2\_22\_0\_,

video1\_.create\_user AS create3\_22\_0\_,

video1\_.update\_time AS update4\_22\_0\_,

video1\_.update\_user AS update5\_22\_0\_,

video1\_.actor AS actor22\_0\_,

video1\_.classify\_id AS classify28\_22\_0\_,

video1\_.classify\_tags AS classify7\_22\_0\_,

video1\_.cp\_id AS cp8\_22\_0\_,

video1\_.definition AS definition22\_0\_,

video1\_.director AS director22\_0\_,

video1\_.drama\_index AS drama11\_22\_0\_,

video1\_.introduction AS introdu12\_22\_0\_,

video1\_.label AS label22\_0\_,

video1\_.max\_number AS max14\_22\_0\_,

video1\_.max\_publish\_time AS max15\_22\_0\_,

video1\_. MODE AS mode22\_0\_,

video1\_. NAME AS name22\_0\_,

video1\_.name\_initial AS name18\_22\_0\_,

video1\_.pay\_type AS pay19\_22\_0\_,

video1\_.plays AS plays22\_0\_,

video1\_.poster AS poster22\_0\_,

video1\_.provider\_id AS provider29\_22\_0\_,

video1\_.release\_time AS release22\_22\_0\_,

video1\_.score AS score22\_0\_,

video1\_.status AS status22\_0\_,

video1\_.total\_number AS total25\_22\_0\_,

video1\_. TYPE AS type22\_0\_,

video1\_.watch AS watch22\_0\_

FROM

video\_lib\_video this\_

INNER JOIN video video1\_ ON this\_.video\_id = video1\_. ID

AND video1\_.status = 1

--AND video1\_.pay\_type IN (

--'FREE'

--,'BASIC\_PACKAGE'

--'ADDED\_PACKAGE'

--)

INNER JOIN classify classifypo4\_ ON video1\_.classify\_id = classifypo4\_. ID

INNER JOIN provider provider5\_ ON video1\_.provider\_id = provider5\_. ID

WHERE

this\_.app\_id = 500050

AND this\_.status = 1 --and this\_.valid=1

and this\_.package\_id in (-1,1,2,3)

and this\_.pay\_type in ('BASIC\_PACKAGE')

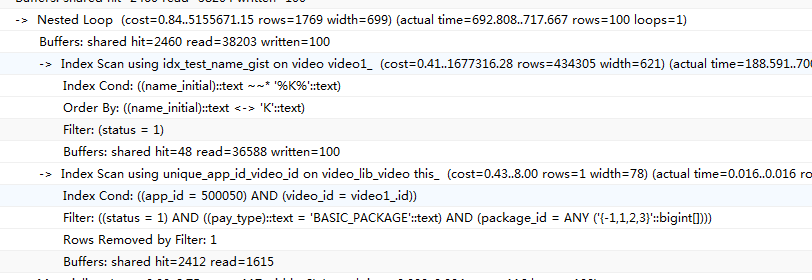
--AND POSITION ('K' IN video1\_.name\_initial) >= 0

and video1\_.name\_initial ILIKE '%K%'

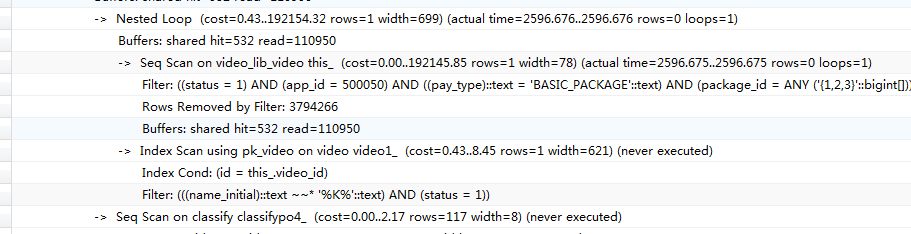
ORDER BY video1\_.name\_initial <-> 'K'

LIMIT 100

这边执行之后，执行计划中会有Nested Loop，那么会则是先执行完一条，往下走一条。在执行计划中，会先找出数据量少的，然后去join大表的过滤数据。由于包含了package为-1，因为video\_lib\_video中对于package\_id为-1的数据量很大，所以默认先去查询了video中的数据满足的数据量(则执行name\_initial的gist索引，走了匹配度的比较索引)。那么再由video去join video\_lib\_video，那么就会走unique\_appid\_videoid的索引



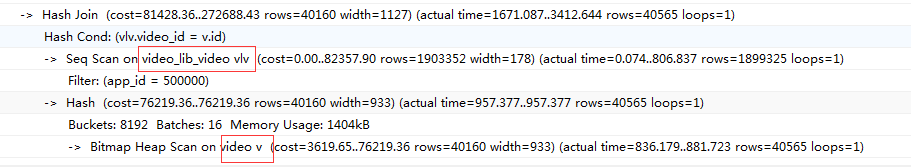
去掉package\_id的-1，会导致video\_lib\_video查询出来的结果集很少，那么就会先查询video\_lib\_video(无法走索引)，接着去连接video表的时候，由于id的索引最高，那么连接的时候，首先采用video的pk\_id索引。那么也会导致name\_initial的gist索引无法执行



## HashJoin

可通过enable\_hashjoin=off来启用或者禁用

不同的表做了全表扫描，然后对结果按照hash值再进行合并。大表对大表之间的join

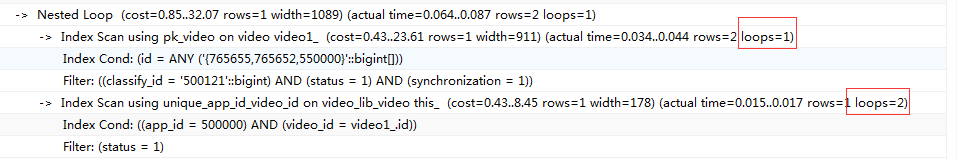


按videoId的hashJoin

## Nest Loop

可通过enable\_nestloop=off来启用或者禁用

大表与小表之间的join，并以小表作为外表，大表作为内表



loops表示应该循环几次，never executed则表示，没有执行

上图执行计划可以看出，video表的loops为1次，表示外表

vlv的loops为2，表示执行两次，因为video有两条记录

## Merge Join

set enable\_mergejoin=on;



按照join的列都进行排序，然后进行合并

# postgresql操作

## 初始化数据库

initDb –D dir

碰到问题：

在linux上面，安装了9.3版本，并在9.3的安装目录下，执行了该命令(环境上面还有9.1版本)，结果却还是初始化成了9.1的数据目录

解决方法：

su - postgres -c "LD\_LIBRARY\_PATH=$start\_dir/lib:$LD\_LIBRARY\_PATH $start\_dir/bin/initdb -D \"$data\_dir\" "

修改临时环境变量LD\_LIBRARY\_PATH

## 修改postgres密码

ALTER ROLE postgres WITH PASSWORD '123456'

## 备份数据库

pg\_dumpall -U postgres -p 6543 >/tmp/db.sql

## 备份单个数据库

pg\_dump –U postgres –p 6543 ott-ams > test.sql

## 还原数据库

psql -U postgres -p 6543 -f /tmp/db.sql

## 升级数据库

su - postgres -c "/cache1/PostgreSQL/9.3/bin/pg\_upgrade -c --link -b /cache1/PostgreSQL/9.2/bin -B /cache1/PostgreSQL/9.3/bin -d /cache1/PostgreSQL/9.2/data -D /cache1/PostgreSQL/9.3/data"

碰到问题(可查看/home/postgres/目录下的日志文件进行定位)：

1. fe\_sendauth: no password supplied

修改pg\_hba.conf，将对应的md5认证，都改成trust

1. cannot load library …

编译pg\_upgrade\_support模块

su - postgres -c "LD\_LIBRARY\_PATH=/usr/local/pgsql\_9\_6/lib:$LD\_LIBRARY\_PATH /usr/local/pgsql\_9\_6/bin/pg\_upgrade -c --link -b \"/usr/local/pgsql/bin\" -B \"/usr/local/pgsql\_9\_6/bin\" -d \"/cache2/ott-db\" -D \"/cache2/db/ott-9-6\""

## 全文索引：

Create index idx\_test on test gin(to\_vsvector(‘english’, ‘…’));

### 查询

Select \* from test where to\_vsvector(‘english’,‘test’) && to\_tsquery(‘english’,’good’)

## 占用空间

### 数据库

**select** pg\_size\_pretty(pg\_database\_size('playboy'));

### 表

不包含索引

**select** pg\_size\_pretty(pg\_relation\_size('test'));

包含索引

select pg\_size\_pretty(pg\_total\_relation\_size('video\_lib\_video'));

### 索引

**select** pg\_size\_pretty(pg\_relation\_size('playboy\_id\_pk'));

## 查看占用情况

SELECT pg\_terminate\_backend(pid) FROM pg\_stat\_activity WHERE datname='数据库名' AND pid<>pg\_backend\_pid();

# gin与gist索引

create extendsion pg\_trgm;

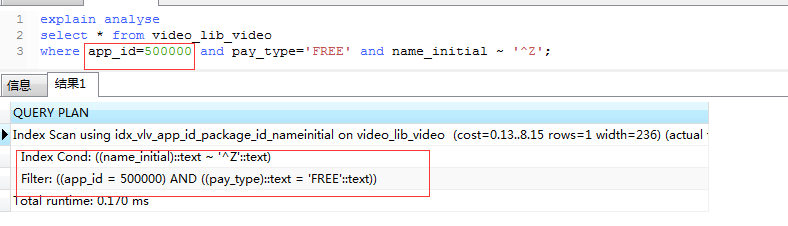
gin索引

## Gist trgm的~不生效

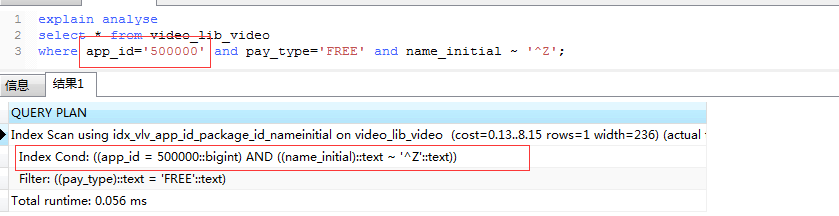
由于该操作只有在9.3.15之后才能起到索引效果

## Gist的复合索引不生效

app\_id为bigint的类型，建立了gist（app\_id， name\_initial gist\_trgm\_ops）



可以看出只走了name\_initial的索引，并没有走app\_id和name\_initial同时作用的复合索引



将app\_id用字符串的方式去等于，结果却能够走正常的gist复合索引

# 查看数据库执行情况

## pg\_stat\_activity

当state=’active’ and wait\_event is not null说是执行sql正在等待,pid就是对应正在执行的进程

## pg\_terminate\_backend

select pg\_terminate\_backend(pid)可执行关闭执行sql

## pg\_cancel\_backend

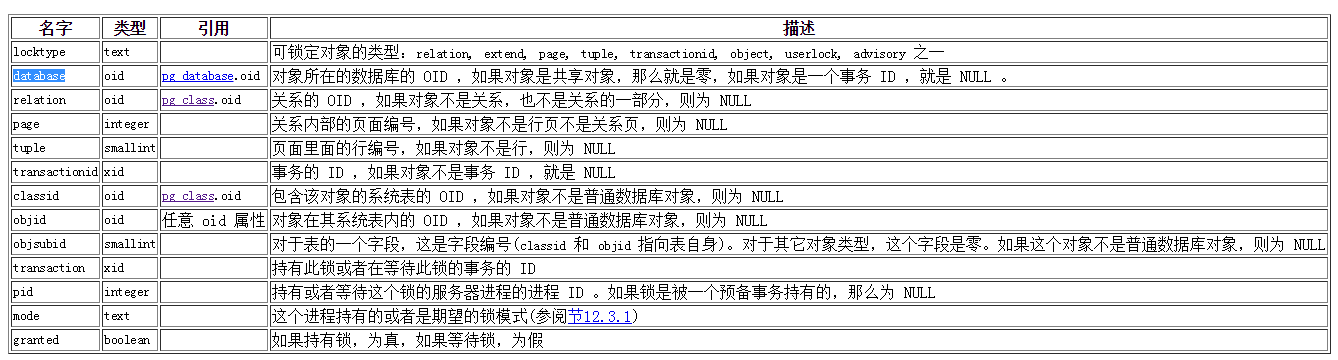
select pg\_terminate\_backend(pid)可关闭执行查询sql

## pg\_locks

可用于查看数据库的记录锁，或者表锁。

pg\_class：查询所有表名

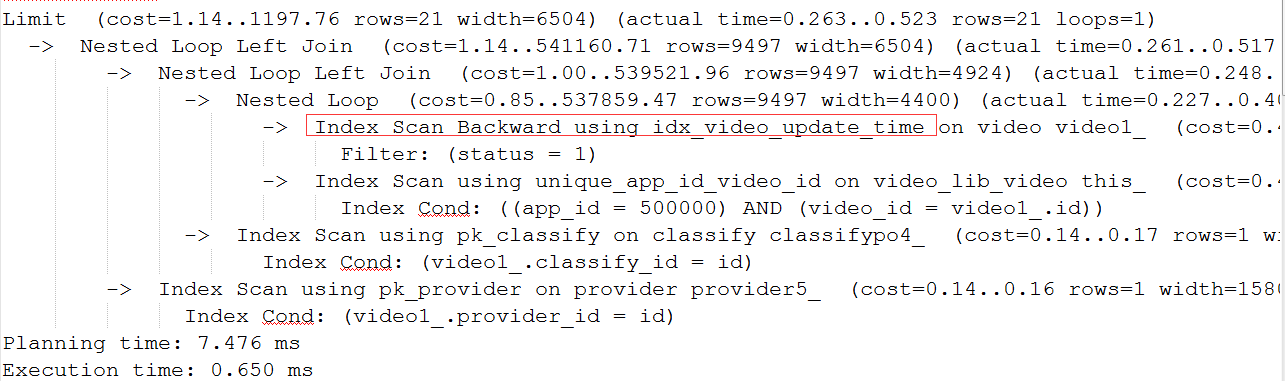
pg\_database:查看数据库



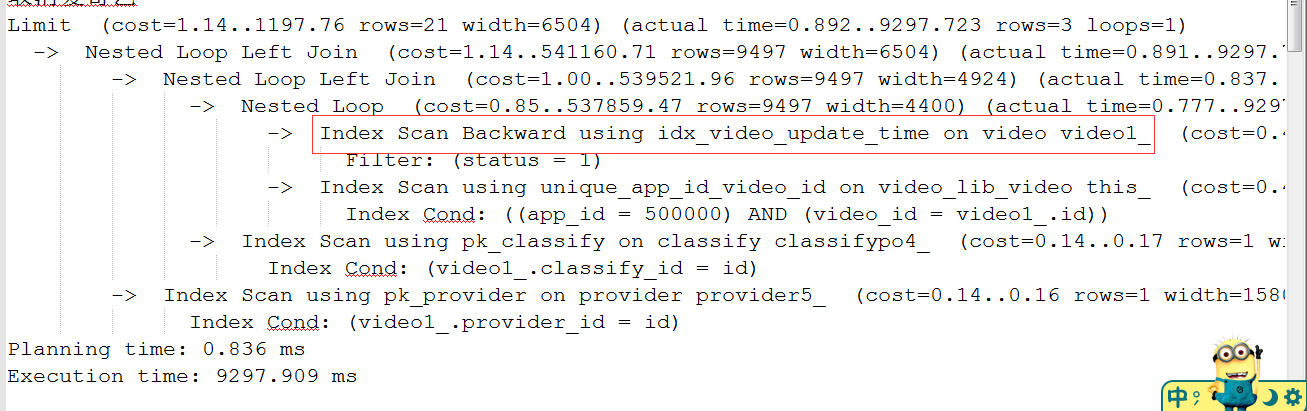
# 数据库的vacuum

类似于oracle的高水线

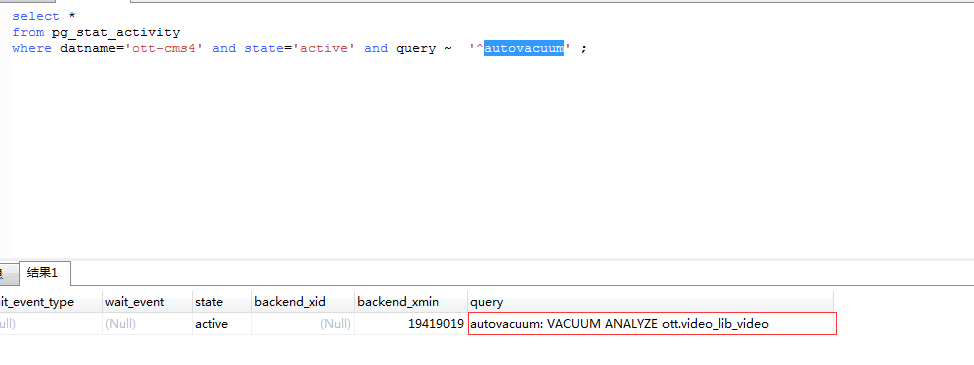
在视频库中，首先设置了video\_lib\_video关联(爱奇艺，180w)，然后查询视频库的数据，



接着再将video\_lib\_video的数据去掉关联(爱奇艺)，再去查询视频库的数据



大概一分钟左右会触发autovacuum，也就是autovacuum\_naptime ，postgresql.conf中的配置周期，默认是1分钟(可通过下列sql查看)。可以看出它是针对表进行vacuum，发现其时间大概需要8分钟左右



手动的去执行vacuum的话，时间大概在70s左右

## Vacuum analyse

更新统计信息，使得优化器有更好的方案去执行sql

## autovacuum

是否启动自动清理

## autovacuum\_naptime

每次清理的时间，默认1分钟

## autovacuum\_vacuum\_threshold

## autovacuum\_analyze\_threshold

默认为50，

## autovacuum\_analyze\_scale\_factor

默认为0.1

## 问题：

1. drama对synchronization建了表达式索引，也就是synchronization=0的索引，如craete index idx\_test on (drama) where synchronization=0;会走idx\_test索引，但是bigMap,而不是正常的idx\_test。

执行select \* from drama inner join video where synchronization=0 order by v.update\_time desc limit 10;

但是没有走idx\_test的索引，而是走了update\_time的索引

问题原因：由于没有对drama进行vacuum analyse的统计信息分析。

执行完之后，接着执行了update 10000条的数据，通过explain analyse的分析，发现其预估的统计分析却没有改变，也没有触发auto\_vacumm的。

问题原因：drama表有200w的数据，而触发自动vacuum的条数>= 200w\*0.1（analyse\_scale\_factor） + 50，所以没有执行auto\_vacuum\_analyse

# 数据库的配置优化

## shared\_buffers

数据库缓冲区，减少内核和磁盘打交道

## effective\_cache\_size

数据库缓存数据

## work\_mem

EnterpriseDB在执行排序操作时,会根据work\_mem的大小决定是否将一个大的结果集拆分为几个小的  
      和work\_mem查不多大小的临时文件。显然拆分的结果是降低了排序的速度。  
      因此增加work\_mem有助于提高排序的速度。及散列表在散列连接,散列为基础的聚集,以及散列为基础的 IN 子查询处理中都要用到。  
      通常设置为实际RAM的2% -4%,根据需要排序结果集的大小而定,比如81920(80M)

比如有100个查询，用到了排序，那么就会占用内容100\*work\_mem的大小

## maintenance\_work\_mem

create index和vacuum时候使用

# 碰到问题：

1. 使用navicat的工具，进行创建索引，结果在执行搜索的时候，却没有按照想要的进行索引。但是使用sql创建，却能够正常。

原因：采用工具的话，需要重新保存多个索引。而不像sql，直接刷新一个索引

1. 当新建一个索引idx1依赖于col1，这时候把col1删除，会造成idx1也被删除。
2. video\_lib\_video新增了字段app\_channel\_id，所以这次唯一索引变成了app\_id,app\_channel\_id,video\_id三个字段组成。本来以为：通过video\_id和app\_channel\_id能够反查出app\_id，应该也能走唯一索引。但是事实却没有。目前的另外一种方式是通过join app，来查出记录数

# Postgresql vs mysql

|  |  |  |
| --- | --- | --- |
| 数据库/不同点 | Mysql | Postgresql |
| 引擎 | 多引擎数据库  myisam：没有事务  innodb:支持acid | 只有一种引擎 |
| 锁 | 当搜索条件不是索引字段时，会锁表，不支持行锁 | 支持行锁 |
| 模式 | 无 | 有schema |
| 权限 | 粒度较大，作用于表上或者数据库上 | 由于有了schema的关系，可以控制得粗细更好 |

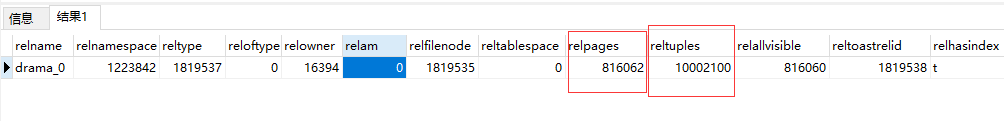
# Postgresql中Page存储和Btree存储分析

## Page存储

根据select pg\_relation\_filenode和pg\_relation\_filepath，可以查看数据表的具体文件和node信息。数据记录的存储不是一次性进行存储，而是分成了很多个小块（block\_size）来存储。但是这个只能在编译文件的时候生效，无法在安装后的数据库进行操作。

1. 查看表有多少个分页

select \* from pg\_class where relname='drama\_0';



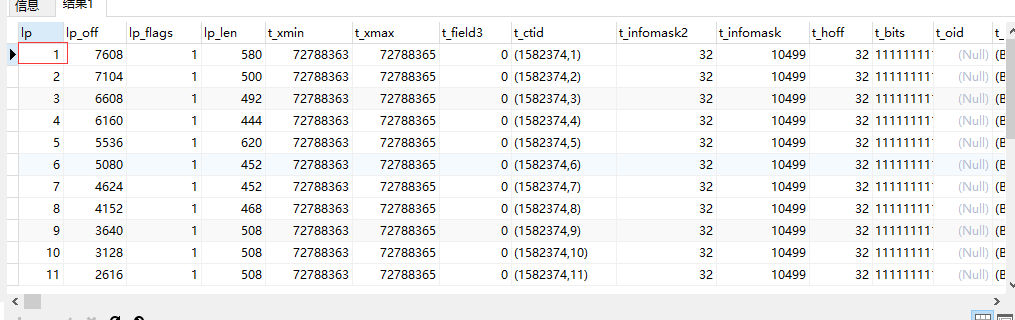
可以看到总页数是816062，总条数10002100条记录

1. 查看分页的信息和条目信息

以下是查看第816062的页条目信息，因为页数是从0编号开始

select \* from heap\_page\_items(get\_raw\_page('drama\_0', 816061));

，那么ctid(816061,1)对应就是代表逻辑位置第826062页，第一条记录的信息，x\_ctid是物理地址



Lp\_len:记录长度，tuple header (24字节)+ data，并且取8的倍数，一个页面可存储多少页就是block size / lp\_len。空行的话，则是24字节+4字节=28字节

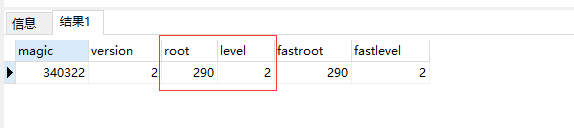
## Btree索引存储结构分析

首先索引默认有meta-page、root-page，如果不够存储，才会加入leaf page，再者有branch page。所以首先查看meta-page的信息，查看索引 idx\_create\_time\_drama\_0

1. 查看bt\_metap(‘idx\_create\_time\_drama\_0’)

可以看到有root-page的block-no为290（root），有两层(level=2)

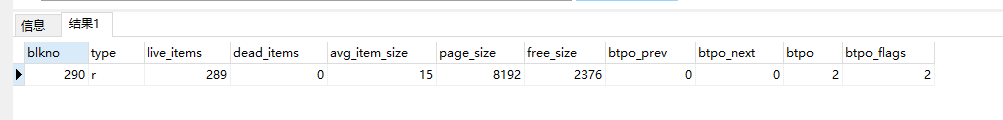
SELECT \* FROM bt\_metap('idx\_create\_time\_drama\_0');



1. 查看root-page=290的page信息

有289个branch-page(live\_items)，没有无效的branches(dead\_items)，没有左右节点(btpo\_prev、btpo\_next)。Btpo为2表示以下还有两层，btpo\_flags为2，表示root page

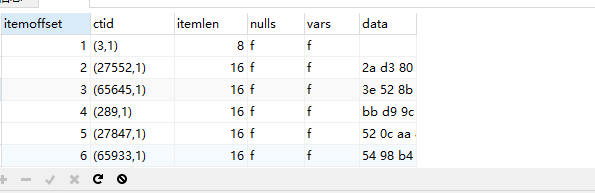
select \* from bt\_page\_stats('idx\_create\_time\_drama\_0', 290);



1. 查看root-page=290的所有子信息(branch-page)

可以看出第一个branch-page的blockno为3

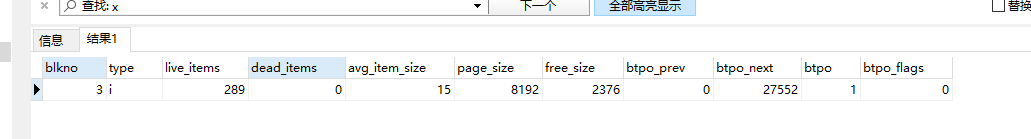
select \* from bt\_page\_items('idx\_create\_time\_drama\_0', 290);



1. 查看branch-page=3的分页信息

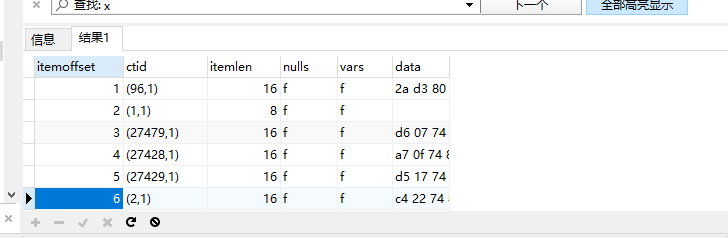
Btpo为1，说明往下还有一层，btpo\_flags为0表示branches-page

select \* from bt\_page\_stats('idx\_create\_time\_drama\_0', 3);



1. 查看branch-page=3的分页条目信息

由4步中可以看出当前页有btpo\_next，所以，第一条数据(96,1)不是起始数据，而是btpo\_next的第一条数据(即27552的btpo\_next第一条数据)，不信可以看27552的数据。这样索引数据的范围就能够找出是(1,1)的data到(96,1)的data之间范围



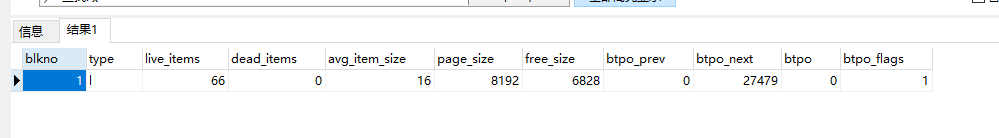
select \* from bt\_page\_stats('idx\_create\_time\_drama\_0', 3);

1. 查看leaf-page=1的分页信息

Btpo=0，表示没有再下一层了，说明这一个分页存储的条目就是对应的数据记录(ctid)

Btpo\_flags=1，表示leaf

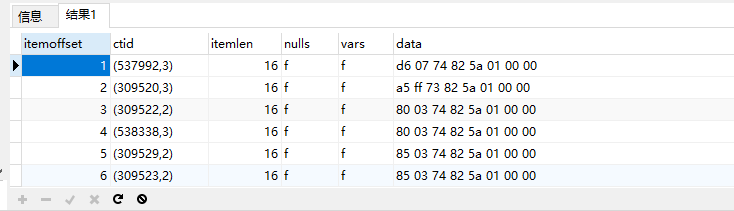
select \* from bt\_page\_stats('idx\_create\_time\_drama\_0', 1);



1. 查看leaf-page=1的分页条目

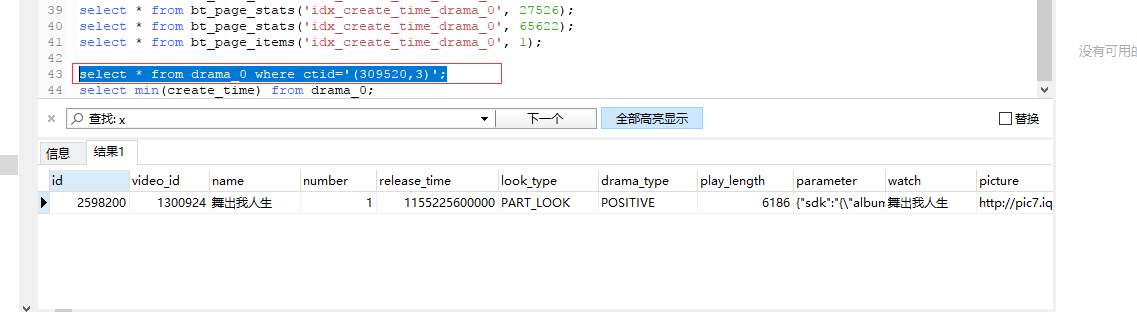
由于6中的btpo\_next不为空，所以第一条数据应该是(309520,3)

select \* from bt\_page\_items('idx\_create\_time\_drama\_0', 1);



1. 查找(309520,3)的数据信息

可以看出在7中的data为a5 ff 73 82 5a 01，对应的16进制是01 5a 82 73 ff a5，刚好就是drama的create\_time为1488247324581



## Postgresql的插入、更新、删除

更新：新增一条数据记录，并将原来的数据变更为已删除，t\_xmax为新记录的t\_xmin的，代表被新数据替代

删除：在记录中的t\_xmax的值标志。不会做记录删除

Vacuum: 清理t\_xmax不为空的记录，将其所有的值清空，将对应的index记录也清除

Vacuum full:清空所有t\_xmax的记录，并移动记录。整理空间

# Postgresql中一些统计信息的视图

# pg\_stat\_database

|  |  |  |
| --- | --- | --- |
| **列** | **类型** | **描述** |
| datid | oid | 一个数据库的 OID |
| datname | name | 这个数据库的名称 |
| numbackends | integer | 当前连接到这个数据库的后端数量。这是在这个视图中唯一一个返回反映当前状态值的列。所有其他列返回从上次重置以来积累的值。 |
| xact\_commit | bigint | 在这个数据库中已经被提交的事务的数量 |
| xact\_rollback | bigint | 在这个数据库中已经被回滚的事务的数量 |
| blks\_read | bigint | 在这个数据库中被读取的磁盘块的数量 |
| blks\_hit | bigint | 磁盘块被发现已经在缓冲区中的次数，这样不需要一次读取（这只包括 PostgreSQL 缓冲区中的命中，而不包括在操作系统文件系统缓冲区中的命中） |
| tup\_returned | bigint | 在这个数据库中被查询返回的行数 |
| tup\_fetched | bigint | 在这个数据库中被查询取出的行数 |
| tup\_inserted | bigint | 在这个数据库中被查询插入的行数 |
| tup\_updated | bigint | 在这个数据库中被查询更新的行数 |
| tup\_deleted | bigint | 在这个数据库中被查询删除的行数 |
| conflicts | bigint | 由于与恢复冲突而在这个数据库中被取消的查询的数目（冲突只发生在后备服务器上，详见[pg\_stat\_database\_conflicts](http://www.postgres.cn/docs/9.6/monitoring-stats.html" \l "PG-STAT-DATABASE-CONFLICTS-VIEW)）。 |
| temp\_files | bigint | 在这个数据库中被查询创建的临时文件的数量。所有临时文件都被统计，不管为什么创建这些临时文件（如排序或哈希），并且不管[log\_temp\_files](http://www.postgres.cn/docs/9.6/runtime-config-logging.html" \l "GUC-LOG-TEMP-FILES)设置。 |
| temp\_bytes | bigint | 在这个数据库中被查询写到临时文件中的数据总量。所有临时文件都被统计，不管为什么创建这些临时文件（如排序或哈希），并且不管[log\_temp\_files](http://www.postgres.cn/docs/9.6/runtime-config-logging.html" \l "GUC-LOG-TEMP-FILES)设置。 |
| deadlocks | bigint | 在这个数据库中被检测到的死锁数 |
| blk\_read\_time | double precision | 在这个数据库中后端花费在读取数据文件块的时间，以毫秒计 |
| blk\_write\_time | double precision | 在这个数据库中后端花费在写数据文件块的时间，以毫秒计 |
| stats\_reset | timestamp with time zone | 这些统计信息上次被重置的时间 |

## pg\_stat\_all\_tables

又可细分pg\_stat\_user\_tables来获取用户表、pg\_stat\_sys\_tables来查看系统表

| **列** | **类型** | **描述** |
| --- | --- | --- |
| relid | oid | 一个表的 OID |
| schemaname | name | 这个表所在的模式的名称 |
| relname | name | 这个表的名称 |
| seq\_scan | bigint | 在这个表上发起的顺序扫描的次数 |
| seq\_tup\_read | bigint | 被顺序扫描取得的活着的行的数量 |
| idx\_scan | bigint | 在这个表上发起的索引扫描的次数 |
| idx\_tup\_fetch | bigint | 被索引扫描取得的活着的行的数量 |
| n\_tup\_ins | bigint | 被插入的行数 |
| n\_tup\_upd | bigint | 被更新的行数（包括 HOT 更新的行） |
| n\_tup\_del | bigint | 被删除的行数 |
| n\_tup\_hot\_upd | bigint | 被更新的 HOT 行数（即不要求独立索引更新的行更新） |
| n\_live\_tup | bigint | 活着的行的估计数量 |
| n\_dead\_tup | bigint | 死亡行的估计数量 |
| n\_mod\_since\_analyze | bigint | 从这个表最后一次被分析后备修改的行的估计数量 |
| last\_vacuum | timestamp with time zone | 上次这个表被手动清理的时间（不统计VACUUM FULL） |
| last\_autovacuum | timestamp with time zone | 上次这个表被自动清理守护进程清理的时间 |
| last\_analyze | timestamp with time zone | 上次这个表被手动分析的时间 |
| last\_autoanalyze | timestamp with time zone | 上次这个表被自动清理守护进程分析的时间 |
| vacuum\_count | bigint | 这个表已被手工清理的次数（不统计VACUUM FULL） |
| autovacuum\_count | bigint | 这个表已被自动清理守护进程清理的次数 |
| analyze\_count | bigint | 这个表已被手工分析的次数 |
| autoanalyze\_count | bigint | 这个表已被自动清理守护进程分析的次数 |

## pg\_stat\_all\_indexes

又可细分pg\_stat\_user\_indexes用户索引、pg\_stat\_sys\_indexes系统索引

Idx\_tup\_fetch是查找表记录的个数，比如如果只查询了索引字段或者查询的记录已经不存在则这个值就会小于idx\_tup\_read

| **列** | **类型** | **描述** |
| --- | --- | --- |
| relid | oid | 这个索引的基表的 OID |
| indexrelid | oid | 这个索引的 OID |
| schemaname | name | 这个索引所在的模式的名称 |
| relname | name | 这个索引的基表的名称 |
| indexrelname | name | 这个索引的名称 |
| idx\_scan | bigint | 在这个索引上发起的索引扫描次数 |
| idx\_tup\_read | bigint | 在这个索引上由扫描返回的索引项数量 |
| idx\_tup\_fetch | bigint | 被使用这个索引的简单索引扫描取得的活着的表行数量 |

## pg\_stat\_progress\_vacuum

查看当前的vacuum的进程

| **列** | **类型** | **描述** |
| --- | --- | --- |
| pid | integer | 这个后端的进程ID。 |
| datid | oid | 这个后端连接到的数据库的OID。 |
| datname | name | 这个后端连接到的数据库的名称。 |
| relid | oid | 当前被VACUUM的表的OID。 |
| phase | text | 当前VACUUM的处理阶段。详见[表 28-21](http://www.postgres.cn/docs/9.6/progress-reporting.html" \l "VACUUM-PHASES)。 |
| heap\_blks\_total | bigint | 当前被VACUUM的表堆数据块的总数。这个数字在扫描开始时被报告；之后增加的数据块 不会（也不必）被当前VACUUM访问。 |
| heap\_blks\_scanned | bigint | 已扫描的堆数据块数。由于[可见性图](http://www.postgres.cn/docs/9.6/storage-vm.html)被用于优化 扫描，因此某些块将被忽略而不进行检查；跳过的块也被包括在这个总数中，所以当 VACUUM完成时，这个数字最终将等于heap\_blks\_total。这个计数 器只有在VACUUM的扫描堆阶段才会增长。 |
| heap\_blks\_vacuumed | bigint | 已被VACUUM的堆数据块数。除非表中没有索引，否则这个计数器只有在VACUUM的 清理堆阶段才会增长。不包含任何死亡元组的数据块将被跳过，因 此该计数器的值有时会大幅跳跃式增加。 |
| index\_vacuum\_count | bigint | 已完成的索引VACUUM数量。 |
| max\_dead\_tuples | bigint | 在必须执行一次索引VACUUM前可存储的死亡元组数量。 这基于[maintenance\_work\_mem](http://www.postgres.cn/docs/9.6/runtime-config-resource.html#GUC-MAINTENANCE-WORK-MEM)的值。 |
| num\_dead\_tuples | bigint | 自上一次索引VACUUM结束后，一共清理的死亡元组数量。 |