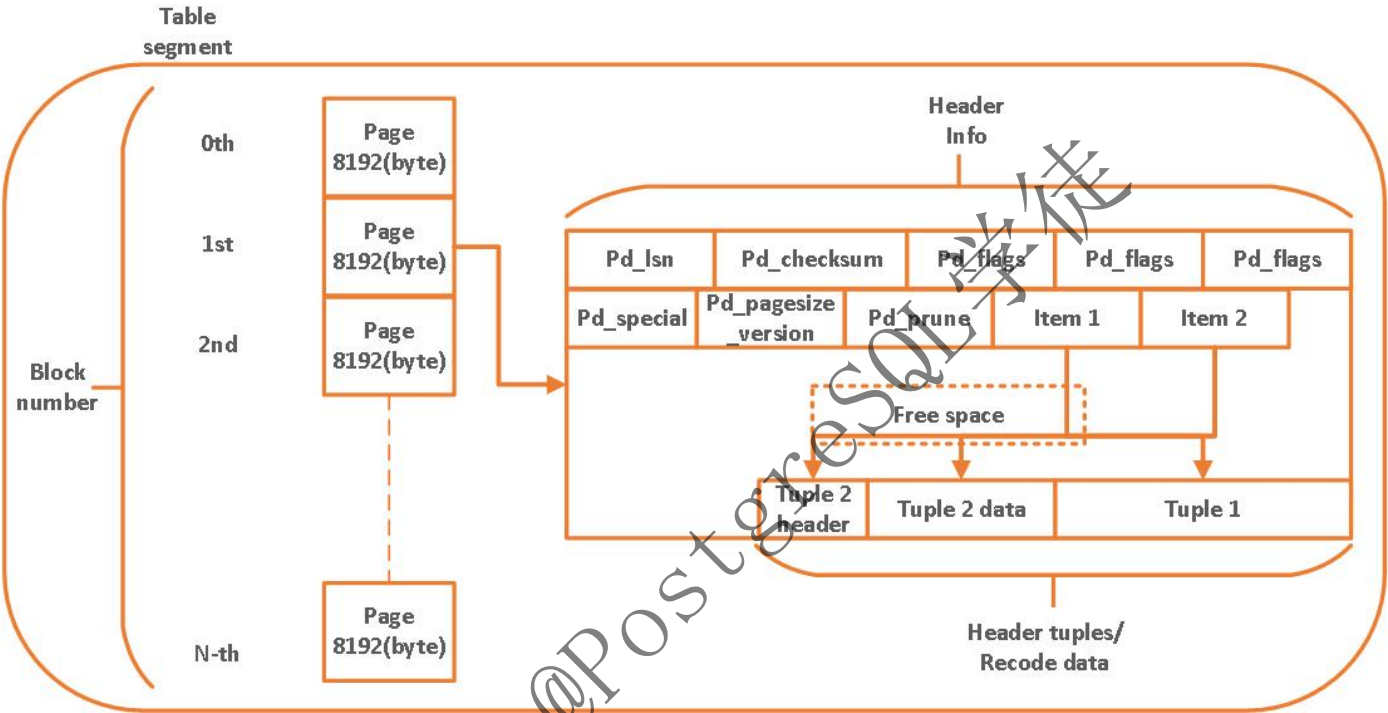


前言

对于从事PostgreSQL相关的，不管你是开发还是运维，肯定都或多或少听到过标志位这个东东，云里雾里，大概了解这个标志位可以加速事务的获取、标识某些状态信息等等，为了弥补这一块的知识空缺。

何为infomask

首先我们需要简单了解一下PostgreSQL的内部结构，数据块的结构如下👉，默认8KB，最大32KB，在此不过多介绍，度娘一堆。

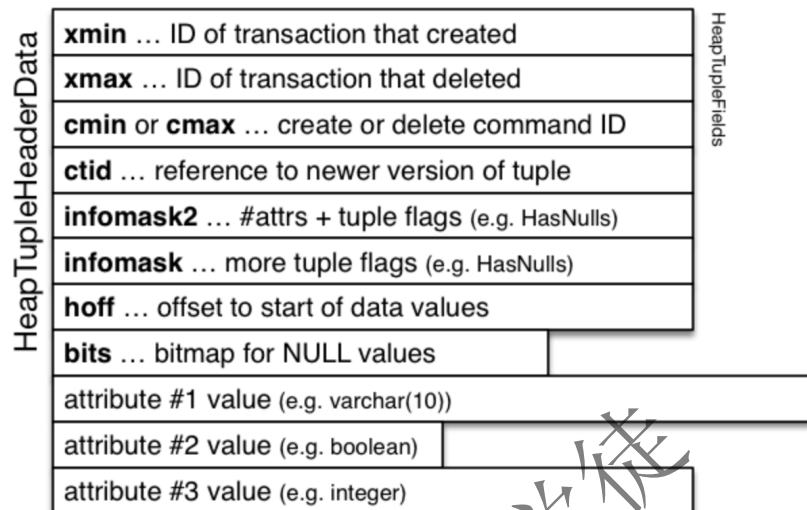


下面是Tuple的具体结构，Tuple头部是由23byte固定大小的前缀和可选的NullBitMap构成，当元组中存在空值时，会出现空值位图，每个字段占一位，远远大于Oracle的3字节，也比Mysql略大。所以在PostgreSQL里面，存储的额外开销要略大于Oracle，另外值得注意的是可能还会有字节对齐，在此表过不提。

我们关心的是infomask，more tuple flags，可以看到，是关于元组Tuple的一些标志位，用于标识元组的一些属性和状态。

PostgreSQL Tuples (cont)

Tuple structure:



在源码里面，infomask有如下这么多状态位，可以看到，就是通过比特位来标识某些属性，比如是否具有空属性、是否具有变长的属性

```
/*
 * information stored in t_infomask:
 */

#define HEAP_HASNULL      0x0001 /* has null attribute(s) */
#define HEAP_HASVARWIDTH  0x0002 /* has variable-width attribute(s) */
#define HEAP_HASEXTERNAL  0x0004 /* has external stored attribute(s) */
#define HEAP_HASOID_OLD   0x0008 /* has an object-id field */
#define HEAP_XMAX_KEYSHR_LOCK 0x0010 /* xmax is a key-shared locker */
#define HEAP_COMBOCID     0x0020 /* t_cid is a combo cid */
#define HEAP_XMAX_EXCL_LOCK 0x0040 /* xmax is exclusive locker */
#define HEAP_XMAX_LOCK_ONLY 0x0080 /* xmax, if valid, is only a locker */

/* xmax is a shared locker */
#define HEAP_XMAX_SHR_LOCK (HEAP_XMAX_EXCL_LOCK | HEAP_XMAX_KEYSHR_LOCK)

#define HEAP_LOCK_MASK (HEAP_XMAX_SHR_LOCK | HEAP_XMAX_EXCL_LOCK | \
    HEAP_XMAX_KEYSHR_LOCK)

#define HEAP_XMIN_COMMITTED 0x0100 /* t_xmin committed */
#define HEAP_XMIN_INVALID  0x0200 /* t_xmin invalid/aborted */
#define HEAP_XMIN_FROZEN   (HEAP_XMIN_COMMITTED|HEAP_XMIN_INVALID)
#define HEAP_XMAX_COMMITTED 0x0400 /* t_xmax committed */
#define HEAP_XMAX_INVALID  0x0800 /* t_xmax invalid/aborted */
#define HEAP_XMAX_IS_MULTTI 0x1000 /* t_xmax is a MultiXactId */
#define HEAP_UPDATED       0x2000 /* this is UPDATEDed version of row */
#define HEAP_MOVED_OFF     0x4000 /* moved to another place by pre-9.0
    * VACUUM FULL; kept for binary
    * upgrade support */
```

```

#define HEAP_MOVED_IN      0x8000 /* moved from another place by pre-9.0
    * VACUUM FULL; kept for binary
    * upgrade support */
#define HEAP_MOVED (HEAP_MOVED_OFF | HEAP_MOVED_IN)

#define HEAP_XACT_MASK     0xFFFF /* visibility-related bits */

/*
 * A tuple is only locked (i.e. not updated by its Xmax) if the
 * HEAP_XMAX_LOCK_ONLY bit is set; or, for pg_upgrade's sake, if the Xmax is
 * not a multi and the EXCL_LOCK bit is set.
 *
 * See also HeapTupleHeaderIsOnlyLocked, which also checks for a possible
 * aborted updater transaction.
 *
 * Beware of multiple evaluations of the argument.
 */

/*
 * information stored in t_infomask2:
 */
#define HEAP_NATTS_MASK     0x07FF /* 11 bits for number of attributes */
/* bits 0x1800 are available */
#define HEAP_KEYS_UPDATED   0x2000 /* tuple was updated and key cols
    * modified, or tuple deleted */
#define HEAP_HOT_UPDATED    0x4000 /* tuple was HOT-updated */
#define HEAP_ONLY_TUPLE     0x8000 /* this is heap-only tuple */

#define HEAP2_XACT_MASK     0xE000 /* visibility-related bits */

```

这里分享一个实用函数，可以直接获取infomask和infomask2里面的标志位信息

```

create type infomask_bit_desc as (mask varbit, symbol text);

create or replace function infomask(msk int, which int) returns text
language plpgsql as $$
declare
    r infomask_bit_desc;
    str text = '';
    append_bar bool = false;
begin
    for r in select * from infomask_bits(which) loop
        if (msk::bit(16) & r.mask)::int <> 0 then
            if append_bar then
                str = str || '|';
            end if;
            append_bar = true;
            str = str || r.symbol;
        end if;
    end loop;
end function;

```

```

        end if;
    end loop;
    return str;
end;
$$ ;

create or replace function infomask_bits(which int)
returns setof infomask_bit_desc
language plpgsql as $$
begin
    if which = 1 then
        return query values
            (x'8000'::varbit, 'MOVED_IN'),
            (x'4000', 'MOVED_OFF'),
            (x'2000', 'UPDATED'),
            (x'1000', 'XMAX_IS_MULTI'),
            (x'0800', 'XMAX_INVALID'),
            (x'0400', 'XMAX_COMMITTED'),
            (x'0200', 'XMIN_INVALID'),
            (x'0100', 'XMIN_COMMITTED'),
            (x'0080', 'XMAX_LOCK_ONLY'),
            (x'0040', 'EXCL_LOCK'),
            (x'0020', 'COMBOCID'),
            (x'0010', 'XMAX_KEYSHR_LOCK'),
            (x'0008', 'HASOID'),
            (x'0004', 'HASEXTERNAL'),
            (x'0002', 'HASVARWIDTH'),
            (x'0001', 'HASNULL');
    elsif which = 2 then
        return query values
            (x'2000'::varbit, 'UPDATE_KEY_REVOKED'),
            (x'4000', 'HOT_UPDATED'),
            (x'8000', 'HEAP_ONLY_TUPLE');
    end if;
end;
$$;

```

下面让我们一个个来分析这些标志位！

HEAP_HASNULL & HASVARWIDTH

这个标志位很明显，就是用于判断是否具有空列，见如下样例，可以看到第一行因为是 text 变长列，所以置了 HASVARWIDTH 的标志位，第二行和第三行插入了 null，所以置了 HASNULL 的标志位。

```

#define HEAP_HASNULL      0x0001  /* has null attribute(s) */
#define HEAP_HASVARWIDTH  0x0002  /* has variable-width attribute(s) */

```

```

postgres=# create table nullt1(id int,info text);

```

```
CREATE TABLE
postgres=# insert into nullt1 values(1,'xiongcc');
INSERT 0 1
postgres=# insert into nullt1 values(1,null);
INSERT 0 1
postgres=# insert into nullt1 values(null,null);
INSERT 0 1
postgres=# select lp, t_xmin, t_xmax, t_ctid,
        infomask(t_infomask, 1) as infomask,
        infomask(t_infomask2, 2) as infomask2
from heap_page_items(get_raw_page('nullt1', 0));
```

lp	t_xmin	t_xmax	t_ctid	infomask	infomask2
1	3643895	0	(0,1)	XMAX_INVALID HASVARWIDTH	
2	3643896	0	(0,2)	XMAX_INVALID HASNULL	
3	3643897	0	(0,3)	XMAX_INVALID HASNULL	

(3 rows)

HEAP_HASEXTERNAL

元组是否包含外部存储的字段

```
#define HEAP_HASEXTERNAL    0x0004 /* has external stored attribute(s) */
```

```
postgres=# create table blog(id int, title text, content text);
```

```
CREATE TABLE
```

```
postgres=# \d+ blog
```

```
Table "public.blog"
 Column | Type      | Collation | Nullable | Default | Storage  | Stats target |
Description
-----+-----+-----+-----+-----+-----+-----+
 id      | integer   |           |          |         | plain    |              |
 title   | text      |           |          |         | extended |              |
 content | text      |           |          |         | extended |              |
Access method: heap
```

可以看到int类型默认策略为plain，而text为extended，目前PostgreSQL支持的存储方式共有4种：

1. PLAIN：避免压缩和行外存储。
2. EXTENDED：先压缩，后行外存储。
3. EXTERNAL：允许行外存储，但不许压缩。
4. MAIN：允许压缩，尽量不使用行外存储。

那么，何时压缩数据？何时行外存储呢？

Tuple压缩：当Tuple大小超过大概2KB时，大概1/4个Block大小时，PostgreSQL会尝试基于LZ压缩算法进行压缩，另外在v14里面直接对表支持了LZ4压缩算法

```
postgres=# CREATE TABLE tab_compression (
    a text COMPRESSION pglz,
    b text COMPRESSION lz4);
CREATE TABLE
postgres=# \d+ tab_compression
```

Column	Type	Collation	Nullable	Default	Storage	Compression	Stats target
a	text				extended	pglz	
b	text				extended	lz4	

Access method: heap

行外存储 (TOAST): Toast属性的本意是The Oversized-Attribute Storage Technique，主要用来应对物理数据行超过数据块大小的场景，对于某个超长的属性单独存储，因为PostgreSQL不允许Tuple跨页存储，因此页的大小就是行大小的硬上限，当某行数据超过PostgreSQL页大小 (8K) 后，会将这个页放到系统命名空间pg_toast下的一个单独的表中，而在原表中存储一个TOAST pointer

```
typedef struct
{
    uint8      va_header;      /* Always 0x80 or 0x01 */
    uint8      va_tag;         /* Type of datum */
    char       va_data[FLEXIBLE_ARRAY_MEMBER]; /* Type-specific data */
} varattrib_1b_e;

typedef struct varatt_external
{
    int32      va_rawsize;     /* Original data size (includes header) */
    int32      va_extsize;     /* External saved size (doesn't) */
    Oid        va_valueid;     /* Unique ID of value within TOAST table */
    Oid        va_toastrelid;  /* RelID of TOAST table containing it */
} varatt_external;
```

看一下blog表的Toast存储在哪里

```
postgres=# select relname,relfilenode,reltoastrelid from pg_class where relname='blog';
 relname | relfilenode | reltoastrelid
-----+-----+-----
 blog   |    49444    |    49447
(1 row)
```

```
postgres=# \d pg_toast.pg_toast_49444
TOAST table "pg_toast.pg_toast_49444"
  Column   | Type
-----+-----
```

```

chunk_id | oid
chunk_seq | integer
chunk_data | bytea
Owning table: "public.blog"
Indexes:
    "pg_toast_49444_index" PRIMARY KEY, btree (chunk_id, chunk_seq)

```

1. chunk_id：用来表示特定TOAST值的OID，可以理解为具有同样 chunk_id 值的所有行组成原表 (也就是此处的blog表) 的TOAST字段的一行数据
2. chunk_seq：用来表示该行数据在整个数据中的位置
3. chunk_data：实际存储的数据。

```

postgres=# insert into blog values(1, 'title', '0123456789');
INSERT 0 1
postgres=# select * from blog;
 id | title | content
----+-----+-----
  1 | title | 0123456789
(1 row)

postgres=# select * from pg_toast.pg_toast_49444;
 chunk_id | chunk_seq | chunk_data
-----+-----+-----
(0 rows)

postgres=# update blog set content=content||content where id=1;
UPDATE 1
postgres=# select id,title,length(content) from blog;
 id | title | length
----+-----+-----
  1 | title |      20
(1 row)

postgres=# select * from pg_toast.pg_toast_49444;
 chunk_id | chunk_seq | chunk_data
-----+-----+-----
(0 rows)

postgres=# select id,title,length(content) from blog;
 id | title | length
----+-----+-----
  1 | title |      40
(1 row)

---反复执行，不断扩大
postgres=# select id,title,length(content) from blog;
 id | title | length
----+-----+-----
  1 | title | 41943040

```

(1 row)

---可以看到, 当content 的长度为41943040时, pg_toast里面有了数据, 并且长度都略小于2K, 说明在extended策略下, 先启用了压缩, 然后才使用行外存储。

```
postgres=# select chunk_id,chunk_seq,length(chunk_data) from pg_toast.pg_toast_49444;
```

chunk_id	chunk_seq	length
49458	0	1996
49458	1	1996
49458	2	1996
49458	3	1996
49458	4	1996
49458	5	1996
49458	6	1996
49458	7	1996
49458	8	1996
49458	9	1996
49458	10	1996
49458	11	1996
49458	12	1996
49458	13	1996
49458	14	1996
49458	15	1996
49458	16	1996
49458	17	1996
49458	18	1996
49458	19	1996
...		

我们再来看看标志位, 这次就很清楚的看到了HASEXTERNAL。

```
postgres=# select lp, t_xmin, t_xmax, t_ctid,
```

```
        infomask(t_infomask, 1) as infomask,
```

```
        infomask(t_infomask2, 2) as infomask2
```

```
from heap_page_items(get_raw_page('blog', 0)) limit 2;
```

lp	t_xmin	t_xmax	t_ctid	infomask
1				
2	3643921	3643922	(0,3)	

UPDATED	XMAX_COMMITTED	XMIN_COMMITTED	HASEXTERNAL	HASVARWIDTH
HOT_UPDATED	HEAP_ONLY_TUPLE			

(2 rows)

HEAP_HASOID_OLD

这个就很好理解了，是否具有oid，PostgreSQL的系统表中大多包含一个叫做OID的隐藏字段，这个OID也是这些系统表的主键。所谓OID，中文全称就是"对象标识符"，oid的分配来自一个实例的全局变量，每分配一个新的对象，对这个全局变量加一。当分配的oid超过4字节整形最大值的时候会重新从0开始分配，但这并不会导致类似于事务ID回卷那样严重的影响，值得注意的是，从v12开始default_with_oids参数就没了，The parameter default_with_oids is gone, it had been [disabled by default](#) since after PostgreSQL 8.0, 并且the default_with_oids parameter cannot be changed to 'on', 可能也是为了性能吧，毕竟每次都去检验重试一遍还是十分耗时的。

```
#define HEAP_HASOID_OLD      0x0008  /* has an object-id field */
```

```
postgres=# show default_with_oids ;
default_with_oids
-----
off
(1 row)

postgres=# select version();
version
-----
PostgreSQL 11.9 on x86_64-pc-linux-gnu, compiled by gcc (GCC) 4.8.5 20150623 (Red Hat 4.8.5-44), 64-bit
(1 row)

postgres=# create table tb3(id int) with oids;
CREATE TABLE
postgres=# insert into tb3 values(1);
INSERT 16440 1
postgres=# select lp, t_xmin, t_xmax, t_ctid,
postgres-#         infomask(t_infomask, 1) as infomask,
postgres-#         infomask(t_infomask2, 2) as infomask2
postgres-# from heap_page_items(get_raw_page('tb3', 0));
 lp | t_xmin | t_xmax | t_ctid | infomask | infomask2
-----+-----+-----+-----+-----+-----
  1 |    574 |      0 | (0,1) | XMAX_INVALID | HASOID
(1 row)
```

HEAP_XMAX_KEYSHR_LOCK & HEAP_XMAX_SHR_LOCK

这仨好基友就是行锁的体现了，在PostgreSQL里面，行锁是存储在磁盘上的，表锁我们可以很方便的通过pg_locks查看，行锁就只能pgrowlocks和pageinspect了。

```
#define HEAP_XMAX_KEYSHR_LOCK 0x0010  /* xmax is a key-shared locker */
#define HEAP_XMAX_EXCL_LOCK   0x0040  /* xmax is exclusive locker */
#define HEAP_XMAX_LOCK_ONLY   0x0080  /* xmax, if valid, is only a locker */
```

```

postgres=# create table mytest2(id int);
CREATE TABLE
postgres=# insert into mytest2 values(1);
INSERT 0 1
postgres=# insert into mytest2 values(2);
INSERT 0 1
postgres=# begin;
BEGIN
postgres=# select * from mytest2 where id = 1 for key share;
 id
----
  1
(1 row)

postgres=# select * from mytest2 where id = 2 for update;
 id
----
  2
(1 row)

```

再开一个会话查看

```

postgres=# select lp, t_xmin, t_xmax, t_ctid,
        infomask(t_infomask, 1) as infomask,
        infomask(t_infomask2, 2) as infomask2
from heap_page_items(get_raw_page('mytest1', 0));
ERROR:  relation "mytest1" does not exist
postgres=# select lp, t_xmin, t_xmax, t_ctid,
        infomask(t_infomask, 1) as infomask,
        infomask(t_infomask2, 2) as infomask2
from heap_page_items(get_raw_page('mytest2', 0));
 lp | t_xmin | t_xmax | t_ctid | infomask | infomask2 |
-----+-----+-----+-----+-----+-----+
  1 | 3643931 | 3643934 | (0,1) | XMIN_COMMITTED | XMAX_LOCK_ONLY | XMAX_KEYSHR_LOCK |
  2 | 3643932 | 3643934 | (0,2) | XMIN_COMMITTED | XMAX_LOCK_ONLY | EXCL_LOCK |
UPDATE_KEY_REVOKED
(2 rows)

```

HEAP_COMBOCID

元组的t_cid是组合command id，这个标记，常用来标记本事务内的老记录，在介绍combo cid前，需要先了解下cmin和cmax。

cmin和cmax位于数据页中每个tuple的头部，用于判断tuple在一个事务内可见性的判断，cmin是产生该条tuple的command id，cmax是删除该tuple的command id。在一个事务内，command id是从0开始递增。一般来说，在PostgreSQL中，判断一条tuple的可见性的时候，通常使用事务当前的snapshot(xmin, xmax, xip_list)，其中xmin是当前活跃的最小事务id，比这个xmin还是小的事务id要么commit要么rollback。xmax是本事务取snapshot时，还没有分配的最小事务id，也就是说，大于等于xmax的事务id所做的修改，对于当前事务来说都不可见。xip_list是当前活跃事务id列表，当前事务看不到该list中的事务所做修改。但是在同一个事务内的时候，并读取时，就会有点问题了：

```
postgres=# create table test1 (c1 int, c2 int);
CREATE TABLE
postgres=# begin;
BEGIN
postgres=# insert into test1 values (1,2);
INSERT 0 1
postgres=# select * from test1;
 c1 | c2 
----+---
  1 |  2 
(1 row)

postgres=# commit;
COMMIT
```

如果仅仅使用xmin，xmax就无法判断tuple的可见性，因为插入的事务跟查询的事务在同一个事务中。所以，此时使用query的command id去比较tuple上的cmin和cmax。上面的例子中，query的cid为1，tuple(1,2)上cmin为0，cmax为invalid。因此，cid > cmin，同时cmax为invalid，所以query对该tuple可见。

因此，为了对一个tuple进行事务内部可见性的判读，需要在每个tuple的头部存储两个uint32类型的字段，cmin和cmax。但是，一个tuple在一个事务内被插入然后马上被更新或删除的场景一般比较少，所以一般MVCC判断的时候，比较少走到使用cmin和cmax的逻辑。

因而，为了减少cmin和cmax对heap page空间的占用，在PG8.3后，将cmin和cmax合并成一个字段，使用1个uint32类型，即combo cid。那么combo cid是如何实现，仅依靠一个字段的存储，在需要cmin的时候提供cmin，而在需要cmax的时候提供cmax呢？对于这个问题，相信通过了解combo cid的逻辑，就可引刃而解。

combo cid逻辑的介绍，这里通过下面几个问题的回答来介绍：

1. 何时会产生combo cid？

在一个事务，当新插入一个tuple的时候，实际是不需要使用combo cid的，因为此时只有cmin有效。而只有在一个tuple被update或者delete时，cmax才会产生，此时就需要使用cmax。所以，在刚插入一个tuple的时候，cmin/cmax这个字段就是指示的cmin，只有，在update或者delete的时候，这个域才会是combo cid。

```
postgres=# begin;
BEGIN
postgres=# insert into test1 values (1,2);
INSERT 0 1
postgres=# update test1 set c2 = 99 ;
```

UPDATE 2

```
postgres=# select lp, t_xmin, t_xmax, t_ctid,
               infomask(t_infomask, 1) as infomask,
               infomask(t_infomask2, 2) as infomask2
from heap_page_items(get_raw_page('test1', 0));
```

lp	t_xmin	t_xmax	t_ctid	infomask	infomask2
1	3643936	3643937	(0,3)	XMIN_COMMITTED	HOT_UPDATED
2	3643937	3643937	(0,4)	COMBOCID	HOT_UPDATED
3	3643937	0	(0,3)	UPDATED XMAX_INVALID	HEAP_ONLY_TUPLE
4	3643937	0	(0,4)	UPDATED XMAX_INVALID	HEAP_ONLY_TUPLE

(4 rows)

HEAP_XMIN_COMMITTED

```
#define HEAP_XMIN_COMMITTED    0x0100 /* t_xmin committed */
#define HEAP_XMIN_INVALID     0x0200 /* t_xmin invalid/aborted */
#define HEAP_XMIN_FROZEN      (HEAP_XMIN_COMMITTED|HEAP_XMIN_INVALID)
#define HEAP_XMAX_COMMITTED    0x0400 /* t_xmax committed */
#define HEAP_XMAX_INVALID     0x0800 /* t_xmax invalid/aborted */
```

这几个好基友就是用来加速事务获取和冻结相关的，冻结可以参照我之前的文章，在此就不再演示了。

当查询一条数据的时候，需要去判断行的可见性，需要去查询相应事务的提交状态，我们只能从CLOG中或者PGXACT内存结构中(未结束的或未清除的事务信息内存)得知该tuple对应的事务提交状态，显然如果每条tuple都要查询pg_clog的话，性能一定会很差，当然还要根据隔离级别、事务快照来综合判断行的可见性，在此不再赘述，在PostgreSQL中提供了TransactionIdInProgress、TransactionIdDidCommit和TransactionIdDidAbort用于获取事务的状态，这些函数被设计为尽可能减少对CLOG的频繁访问(假如把freeze相关参数设置为20亿的话，那么clog最多可能达到500多MB，每一个事务占2bit)。尽管如此，如果在检查每条元组时都执行这些函数，也可能会成为瓶颈。

所以，为了解决这个问题，PostgreSQL在t_infomask中使用了相关标志位，如下：

```
#define HEAP_XMIN_COMMITTED    0x0100 /* t_xmin committed */
#define HEAP_XMIN_INVALID     0x0200 /* t_xmin invalid/aborted */
#define HEAP_XMAX_COMMITTED    0x0400 /* t_xmax committed */
#define HEAP_XMAX_INVALID     0x0800 /* t_xmax invalid/aborted */
```

在读取或写入元组时，PostgreSQL会择机将提示为设置到t_infomask中，比如PostgreSQL检查了元组的t_xmin对应事务的状态，结果为committed，那么就会在元组的t_infomask中置位一个HEAP_XMIN_COMMITTED，表示这条元组已经提交了，如果设置了标志位，那么就不再需要去调用TransactionIdDidCommit和TransactionIdDidAbort去获取事务的状态，可以高效地检查每个元组xmin和xmax对应的事务状态。所以，和Oracle一样，一些select操作也会产生写IO，原因就是设置标志位。

```
postgres=# create table test2(id int);
CREATE TABLE
postgres=# insert into test2 values(1);
INSERT 0 1
```

```
postgres=# select t_xmin,t_xmax,t_infomask,t_infomask2 from
heap_page_items(get_raw_page('test2', 0));
 t_xmin | t_xmax | t_infomask | t_infomask2
-----+-----+-----+-----
 3643942 |      0 |      2048 |           1
(1 row)
```

```
postgres=# select lp, t_xmin, t_xmax, t_ctid,
infomask(t_infomask, 1) as infomask,
infomask(t_infomask2, 2) as infomask2
from heap_page_items(get_raw_page('test2', 0));
lp | t_xmin | t_xmax | t_ctid | infomask | infomask2
---+-----+-----+-----+-----+-----
 1 | 3643942 |      0 | (0,1) | XMAX_INVALID |
(1 row)
```

---注意，此处仅有XMAX_INVALID标志位

```
postgres=# select * from test2;
id
---
 1
(1 row)
```

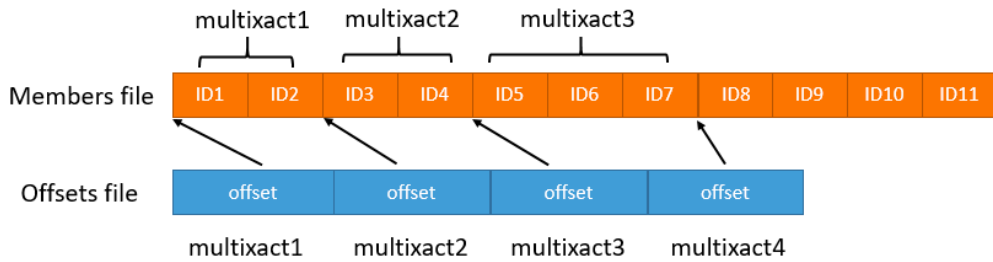
```
postgres=# select lp, t_xmin, t_xmax, t_ctid,
infomask(t_infomask, 1) as infomask,
infomask(t_infomask2, 2) as infomask2
from heap_page_items(get_raw_page('test2', 0));
lp | t_xmin | t_xmax | t_ctid | infomask | infomask2
---+-----+-----+-----+-----+-----
 1 | 3643942 |      0 | (0,1) | XMAX_INVALID | XMIN_COMMITTED |
(1 row)
```

---添加了XMIN_COMMITTED的标志位

但是需要注意的是，并不是在事务结束时设置t_infomask的标志位，而是等到后面的DML或者DQL，VACUUM等SQL扫描到对应的TUPLE时，触发置位的操作，是不是大有前人拉屎后人擦屁股的赶脚？？

HEAP_XMAX_IS_MULTI

也就是multixact，因为对于FOR SHARE和FOR KEY SHARE这一类的行级锁，一行上面可能会被多个事务加锁，Tuple上动态维护这些事务代价很高，为此引入了multixact机制，将多个事务记录到MultiXactId，再将MultiXactId记录到tuple的xmax中。



Members文件中每4个一组，会有一个4byte的flag,每组bytes.offsets中记录的offset要经过计算才能得到真实members中的起始位置
 源码位置
[Backend/access/transam/multixact.c](#)

```

/*
 * The situation for members is a bit more complex: we store one byte of
 * additional flag bits for each TransactionId. To do this without getting
 * into alignment issues, we store four bytes of flags, and then the
 * corresponding 4 Xids. Each such 5-word (20-byte) set we call a "group", and
 * are stored as a whole in pages. Thus, with 8kB BLCKSZ, we keep 409 groups
 * per page. This wastes 12 bytes per page, but that's OK -- simplicity (and
 * performance) trumps space efficiency here.
 *
 * Note that the "offset" macros work with byte offset, not array indexes, so
 * arithmetic must be done using "char *" pointers.
 */
/* We need eight bits per xact, so one xact fits in a byte */
#define MXACT_MEMBER_BITS_PER_XACT 8
#define MXACT_MEMBER_FLAGS_PER_BYTE 1
#define MXACT_MEMBER_XACT_BITMASK ((1 << MXACT_MEMBER_BITS_PER_XACT) - 1)

/* how many full bytes of flags are there in a group? */
#define MULTIXACT_FLAGBYTES_PER_GROUP 4
#define MULTIXACT_MEMBERS_PER_MEMBERGROUP \
    (MULTIXACT_FLAGBYTES_PER_GROUP * MXACT_MEMBER_FLAGS_PER_BYTE)
/* size in bytes of a complete group */
#define MULTIXACT_MEMBERGROUP_SIZE \
    (sizeof(TransactionId) * MULTIXACT_MEMBERS_PER_MEMBERGROUP + MULTIXACT_FLAGBYTES_PER_GROUP)
#define MULTIXACT_MEMBERGROUPS_PER_PAGE (BLCKSZ / MULTIXACT_MEMBERGROUP_SIZE)
#define MULTIXACT_MEMBERS_PER_PAGE \
    (MULTIXACT_MEMBERGROUPS_PER_PAGE * MULTIXACT_MEMBERS_PER_MEMBERGROUP)

```

```

postgres=# begin;
BEGIN
postgres=# select txid_current();
txid_current
-----
3643949
(1 row)

postgres=# select * from test3 where id = 1 for key share;
id
---
1
(1 row)

```

查看行锁

```

postgres=# select * from test3 as t,pgrowlocks('test3') as lc where t.ctid =
lc.locked_row;
id | locked_row | locker | multi | xids | modes | pids
---+-----+-----+-----+-----+-----+-----
1 | (0,1) | 3643949 | f | {3643949} | {"For Key Share"} | {18215}
(1 row)

```

再开一个事务，加上行锁

```

postgres=# begin;
BEGIN
postgres=# select txid_current();
 txid_current
-----
      3643951
(1 row)

postgres=# select * from test3 where id = 1 for share;
 id
----
   1
(1 row)

```

再次查看行锁，可以看到multi字段为true了

```

postgres=# select * from test3 as t,pgrowlocks('test3') as lc where t.ctid =
lc.locked_row;
 id | locked_row | locker | multi |      xids      |      modes      |      pids
-----+-----+-----+-----+-----+-----+-----
  1 | (0,1)      |      1 | t      | {3643949,3643951} | {"Key Share",Share} |
{18215,18267}
(1 row)

postgres=# select lp, t_xmin, t_xmax, t_ctid,
infomask(t_infomask, 1) as infomask,
infomask(t_infomask2, 2) as infomask2
from heap_page_items(get_raw_page('test3', 0));
 lp | t_xmin | t_xmax | t_ctid | infomask
-----+-----+-----+-----+-----
  1 | 3643944 |      1 | (0,1) | XMAX_IS_MULTI|XMIN_COMMITTED|XMAX_LOCK_ONLY|EXCL_LOCK|XMAX_KEYSHR_LOCK |
  2 | 3643945 |      0 | (0,2) | XMAX_INVALID|XMIN_COMMITTED
  3 | 3643946 |      0 | (0,3) | XMAX_INVALID|XMIN_COMMITTED
(3 rows)

```

可以通过pg_get_multixact_members获取multixact

```
postgres=# select * from pg_get_multixact_members('1');
   xid   | mode
-----+-----
 3643949 | keysh
 3643951 | sh
(2 rows)
```

其他

其他几个infomask标志位就不再赘述了，下面这两标志位保留用于pg_upgrade使用

```
#define HEAP_MOVED_OFF      0x4000 /* moved to another place by pre-9.0
    * VACUUM FULL; kept for binary
    * upgrade support */
#define HEAP_MOVED_IN      0x8000 /* moved from another place by pre-9.0
    * VACUUM FULL; kept for binary
    * upgrade support */
#define HEAP_MOVED (HEAP_MOVED_OFF | HEAP_MOVED_IN)
```

update产生的元组

```
#define HEAP_UPDATED      0x2000 /* this is UPDATED version of row */
```

这个和可见性有关了，不过愚笨的我暂未复现什么条件会触发这个标志位的设置，再琢磨琢磨。

```
#define HEAP_XACT_MASK      0xFFFF /* visibility-related bits */
```

小结

另外我们需要注意的是，标志位还和wal_log_hints有关

1. 假设未开启checksum，开启了wal_log_hints,第一次使页面变脏的操作是修改hint bints,会记录整页到wal中
2. 假设开启了checksum，不管wal_log_hints如何，checkpoint后第一次修改的页面都会记录整页到wal中。即使是hint bits
3. 假设未开启wal_log_hints，第一次使页面变脏的操作是修改hint bints，不会记录full page image
4. 可见，在启用 checksum 的情况下，checkpoint 后页面的第一次修改如果是更新 Hint Bits，会写 Full Page Image 至 WAL 日志，这会导致 WAL 日志占用更多的存储空间。

同时，还有一种可能性，因为SetHintBits是针对单条tuple的，所以当有并行的会话在对一个数据页的多个tuple进行SetHintBits操作时，可能导致这个PAGE在多次checkpoint时被写多次到WAL。或者在2个checkpoint之间，多次被bgwriter刷到OS dirty page，可能造成多次OS IO。

以上种种演示了hint bits的作用，可以看到好处很多，并且可以用这个去面试小白：select是否会产生写IO? (坏笑ing...)

参考

<https://cloud.tencent.com/developer/article/1004455>

https://github.com/digoal/blog/blob/master/201610/20161002_03.md

https://www.commandprompt.com/blog/decoding_infomasks/

<https://webcms3.cse.unsw.edu.au/COMP9315/20T1/resources/40359>

<https://zhuanlan.zhihu.com/p/67725967>

!

xiongcc@PostgreSQL学徒