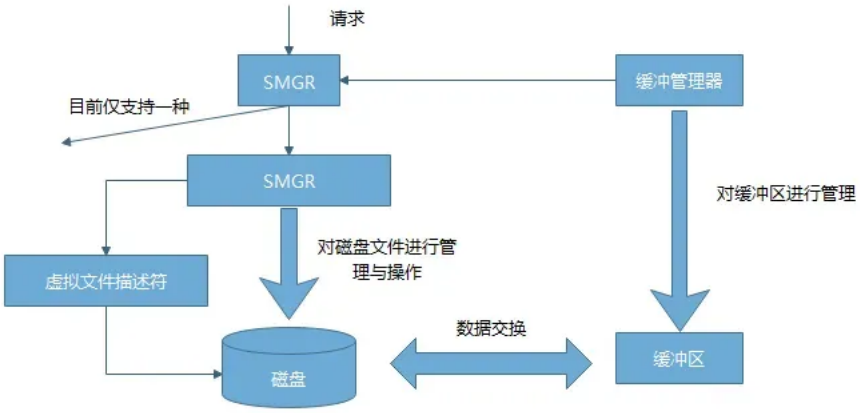
postgres存储管理-外存管理

二、外存管理

外存管理由SMGR （主要代码在smgr. c中） 提供对外存操作的统一接口。SMGR负责统管各种介质管理器，会根据上层的请求选择一个具体的介质管理器进行操作。外存管理体系结构如下图所示：



2.1、表和元组组织方式

2.1.1、堆文件

元组之间不进行关联， 这样的表文件称为堆文件。

a） **普通堆 (ordinary cataloged heap)**

b） **临时堆 (temporary heap)**

仅在会话过程中临时创建， 会话结束会自动删除。

c） **序列 (SEQUENCE relation, 一种特殊的 单行表)**

序列则是一种元组值自动增长的特殊堆。

d） **TOAST 表 (TOAST table)**

在堆中要删除一个元组，理论上有两种方法:

1)直接物理删除:找到该元组所在文件块，并将其读取至缓冲区。然后在缓冲区中删除这个元组，最后再将缓冲块写回磁盘。

2)标记删除:为每个元组使用额外的数据位作为删除标记。当删除元组时，只需设置相应的删除标记，即可实现快速删除。这种方法并不立即回收删除元组占用的空间。

PostgreSQL采用的是第二种方法，每个元组的头部信息HeapTupleHeader 就包含了这个删除标记位，其中记录了删除这个元组的事务D和命令D。如果上述两个D有效，则表明该元组被删除;若无效，则

表明该元组是有效的或者说没有被删除的。在第7章中将会看到这种方法对多版本并发控制也是有好处的。

2.1.2、文件存储形式

1）由于版本更新，在postgres16.2版本下，数据页面的头信息已经达到24个字节，在bufpage.h的文件里看

```
154
155 typedef struct PageHeaderData
156 {
157     /* XXX LSN is member of *any* block, not only page-organized ones */
158     PageXLogRecPtr pd_lsn;      /* LSN: next byte after last byte of xlog
159                                  * record for last change to this page */
160     uint16      pd_checksum;    /* checksum */
161     uint16      pd_flags;       /* flag bits, see below */
162     LocationIndex pd_lower;     /* offset to start of free space */
163     LocationIndex pd_upper;     /* offset to end of free space */
164     LocationIndex pd_special;   /* offset to start of special space */
165     uint16      pd_pagesize_version;
166     TransactionId pd_prune_xid; /* oldest prunable XID, or zero if none */
167     ItemIdData  pd_linp[FLEXIBLE_ARRAY_MEMBER]; /* line pointer array */
168 } PageHeaderData;
169
170 typedef PageHeaderData *PageHeader;
171
```

其头部内容，至少包括但不限于：空闲空间的起始和结束位置；Special space 的开始位置；项指针的开始位置；页面大小和版本信息；标志信息：是否存在空闲项指针、 是否所有的元组都可见、页面修改日志序列号 (LSN)、页面的校验和。用于检测页面是否损坏、页面中最老的可修剪事务ID。 用于垃圾回收。

2）除此之外·，还包括了一系列linp。Linp是 ItemIdData 类型的数组， ItemIdData 类型由 lp_ off 、 lp_ flags 和 lp_ len 三个属性组 成。 每一个 ItemIdData 结构用来指向文件块中的一个元组， 其中 lp_ off 是元组在文件块中的偏移量， 而 lp_ len 则说明了该元组的长度， lp_flages表示元组的状态（分为未使用、 正常使用 、 HOT 重定向和已被删除四种状态）。 每个 Linp 数组元素的长度为6字节。

3）页面中未被使用的空间，用于存储新的数据。 可用空间分为两部分：pd_lower 和 pd_upper 之间的空间(Freespace 空闲空间)、未使用的行指针。其中 Linp 元素从 Freespace 的开头开始分配， 而新元组数据则从尾部开始分配。

4）Special space 是特殊空间， 用于存放与索引方法相关的特定数据。Special space 在普通表文件块中并没有使用， 其内容被置为空。只有设置索引才被使用。

2.2、表文件管理

在PostgreSQL中，每个表都用一个文件（表文件）存储，表文件以表的OI D命名。

2.2.1、分页存储管理

每个表文件由多个 BLCKSZ（一个可配置的常量）字节大小的文件块组成，每个文件块又可以包含多个元组。

表文件以文件块为单位进行IO交换。一个文件块的大小为8k

2.2.2、行式存储

2.2.3、系统表

2.3、磁盘管理器

2.3.1、SMGR实现

磁盘管理器是SMGR的一种具体实现， 它对外提供了管理磁盘介质的接口，其主要实现在文件 md.c中。

```
typedef struct _MdfdVec
{
    File        mdfd_vfd;       /* fd number in fd.c's pool */
    BlockNumber mdfd_segno;     /* segment number, from 0 */
} MdfdVec;

static MemoryContext MdCxt;     /* context for all MdfdVec objects */
```

mdfd_vfd:该文件所对应的 VFD（见 3.2.3 节VFD机制）。mdfd_segno :由于较大的表文件会被切分成多个文件（可以称为段），因此用这个字段表示当前这个文件是表文件的第几段。mdfd_chain:指向同一表文件下一段的MdfdVec, 通过这个字段可以把表文件的各段链接起来，形成链表。

通过mdfd_chain链表使得大于操作系统文件大小（通常为2GB)限制的表文件更易被支持，因 为我们可以将表文件切割成段，使每段小千2GB。

2.3.2、VFD机制

VFD机制通过合理使用有限个实际文件描述符来满足无限的VFD访问需求。

VFD机制的原理类似连接池，当进程申请打开一个文件时，总是能返回一个虚拟文件描述符，对外封装了打开物理文件的实际操作。

VFD C++

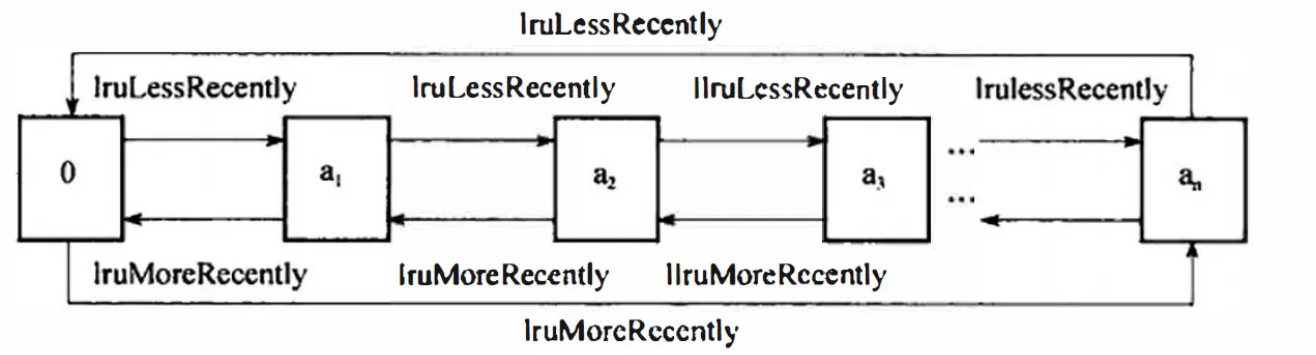
```
typedef struct vfd
{
    int fd; /* current FD, or VFD_CLOSED if none */
    unsigned short fdstate; /* bitflags for VFD's state */
    ResourceOwner resowner; /* owner, for automatic cleanup */
    File nextFree; /* link to next free VFD, if in freelist
*/
    File lruMoreRecently; /* doubly linked recency-of-use list
*/
    File lruLessRecently;
    off_t seekPos; /* current logical file position, or -1 */
    off_t fileSize; /* current size of file (0 if not temporar
y) */
    char *fileName; /* name of file, or NULL for unused VFD */
/* NB: fileName is malloc'd, and must be free'd when closing the VFD
*/
    int fileFlags; /* open(2) flags for (re)opening the file
*/
    int fileMode; /* mode to pass to open(2) */
} Vfd;
```

VFD 机制的实现主要依赖于一个 VFD 数组，其中每个元素对应一个 VFD，而每个 VFD 包含了与打开的文件相关的信息，例如文件描述符、文件状态、文件指针位置等。通过这种方式，PostgreSQL 可以轻松地管理已经打开的文件，并且可以根据需要动态调整 VFD 数组的大小。

在 PostgreSQL 中，每个后台进程都会使用一个 LRU（Least Recently Used，最近最少使用）池来管理所有已经打开的 VFD。LRU 池通过维护一个链表结构，按照 VFD 最近的使用情况进行排序，使得最近使用的 VFD 总是排在链表的前面，而最久未使用的 VFD 则排在链表的末尾。

2.3.3、LRU池

在每一个 PostgreSQL 后台进程中都使用一个 LRU (Last Recently Used, 最近最少使用）池来管理所有已经打开的 VFD,池中每个 VFD 都对应一个物理上已经打开的文件。



从LRU池里VFD的操作主要包括以下三种：

(1)从LRU池删除 VFD

该操作发生在进程使用完一个文件并关闭它时，通过LruDelete函数实现。该操作将指定的VFD从LRU池中删除，并将该VFD对应的文件关闭掉。例如，我们要对Vfd $[a_2]$ 执行LruDelete 操作，则首先将Vfd $[a_2]$ 的lruLessRecently指向Vfd $[a_3]$ ，将Vfd $[a_3]$ 的lruMoreRecently指向Vfd $[a_1]$ 。这样就将Vfd $[a_2]$ 从LRU池中删除了，如果Vfd $[a_2]$ 的fdstate被置为FD_DIRTY, 则要先将Vfd $[a_2]$ 对应的文件同步回磁盘， 再清掉 FD_DIRTY位。接着将Vfd $[a_2]$ 的seekpos置为该文件当前读写的指针位置，最后将Vfd $[a_2]$ 对应的文件关闭掉并将Vfd $[a_2]$ 中的fd置为VFD_ CLOSED。这样Vfd $[a_2]$ 就变为空闲，还需要将其加入到空闲链表中。

(2)将VFD插人LRU池

该操作发生在打开一个新的VFD时，通过LruInsert函数实现。该操作将指定 VFD对应的物理文件打开， 并将该VFD插人到VFD[0]之后的位置。例如，我们要插人的VFD为b, 首先要根据b中的fd字段判断对应的物理文件 是否已经打开了，如果没有，则根据b中的fileName打开此文件， 并插入到LRU池中。插入时要将b的lruMore–

Recently指向Vfd[0]. Vfd[0]的lruLessRecently指向b, 然后 b的lruLessRecently指向Vfd $[a_1]$, Vfd $[a_1]$ 的lruMoreRecently指向b。

(3)删除 LRU池尾的VFD

该操作通过ReleaseLruFile函数实现， 它将LRU池中末尾的那个VFD删除。当LRU池已满而此时又要打开 新的文件时， 就需要执行ReleaseLruFile 操作，将池中末尾的VFD（最少使用的VFD) 删掉，这样新打开的VFD 就可以插入到LRU中。注意，这里被删除的VFD仅仅只是从LRU 池中脱链并关闭其对应的物理文件，VFD结构本身并不做其他修改和删除。因为进程后面的操作还可能会用到该VFD所对应的物理文件 。当再次需要访问一个LRU池之外的VFD时，需要先根据VFD中记录的文件打开标志打开其对应的物理文件，然后根据VFD中记录的读写指针位置将物理文件描述符的读写指针移动到正确的位置，最后还要把该VFD重新插入到LRU池中。

2.4、空间管理

2.4.4、空闲表空间映射

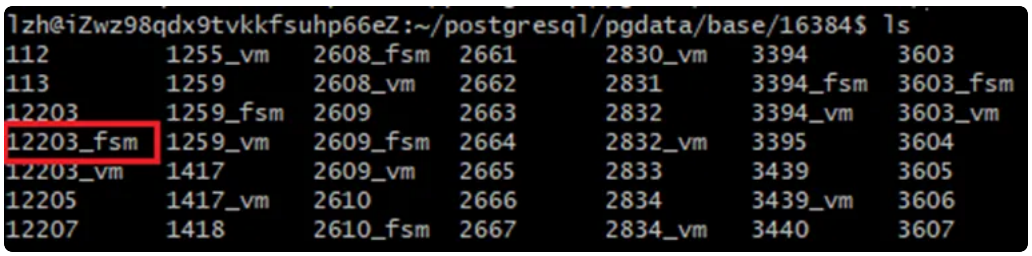
对于每个表文件（包括系统表在内），同时创建一个名为“关系表OID_fsm" 的文件，用千记录该表的空闲空间大小，称之为空闲空间映射表文件(FSM)

FSM 文件的主要作用是跟踪和管理表空间内的空闲区域。这有助于数据库系统更高效地分配和回收空间，尤其是在频繁进行插入、更新和删除操作的场景下。通过使用 FSM，PostgreSQL 可以快速找到足够大的连续空间来满足新数据的需求，而无需进行耗时的磁盘碎片整理。

FSM特点：

1） 每个堆和索引关系（除了哈希索引）都有一个FSM用来跟踪关系中的可用空间

2） FSM作为一个独立的分支存放在主要关系数据旁边，文件名格式为 filenode number 加 _fsm 后缀，如下图oid为16384的数据库里的oid为12203的表，它的FSM文件名为12203_fsm。



3） FSM被组织成一课FSM页的树，底层的FSM页存储了每一个 heap (or index) page 中可用的空闲空间，每个页对应一个字节。上层FSM页面则聚集来自于下层页面的信息

4） 每个FSM页里是一个数组表示的二叉树，每个节点一个字节。每个叶节点表示一个堆页面或者一个下层FSM页面。在每一个非叶节点中存储了它孩子节点中的最大值

FSM中存储的并不是实际的空闲空间大小，而是用一个字节来表示一个范围内的空闲空间大小，如下表所示：

字节	空闲空间范围（字节）
0	0 ~ 31
1	32 ~ 63
…	…
255	8164 ~ 8192

为了更快的查找，FSM文件也不是使用数组存储每个页的空闲空间，而是使用了一个三层树结构。第0层和第1层是辅助层， 第2层用于实际存放各表页中的空闲空间字节位。

FSM页内，有两个基本操作：**查找和更新**

查找：

1） 要在一个FSM页里找一个有 X 空闲空间的页，也就是n >= X的叶子节点，只需要从根节点开始，选一个大于n >= X的孩子作为下一步，一直遍历到叶子节点即可。

2） 但从 fsm_set_avail()看不完全是这样的。在一个FSM页里找一个有 X 空闲空间的页，它会先看根节点是否 >= X，如果有再从 fp_next_slot 开始找：每一步向右移再爬上父节点，在有足够free space的节点停下，停下后在用开始说得方法向下走。(fp_next_slot有两个作用，在找底层FSM页时，每次找到后会指向找到的slot+1，以分散FSM搜索返回的页面。在找上层FSM页时，找到后指向找到的slot，那下次也从这里开始找，可以利用OS的预取和批量写入的优化)

更新：

1） 要更新一个页的空闲空间为 X，首先更新对应的叶子节点，然后不断向上走，维护父节点为两个孩子的最大值，直到一个父节点的值不变即可停下

2.4.5、可见性映射表VM

Postgres 为实现多版本并发控制技术，当事务删除或者更新元组时，并非从物理上进行删除，而是将其进行逻辑删除[具体实现通过设置元组头信息xmax/infomask等标志位信息],随着业务的累增，表会越来越膨胀，对于执行计划的生成/最优路径的选择会产生干扰。为解决这一问题，可以通过调用VACUUM来清理这些无效元组。

但是一个表可能有很多页组成，如何快速定位到含有无效元组的数据页在高并发场景显得尤为重要，pg为表新增对应的附属文件—可见性映射表（VM）,来加速判断heap块是否存在无效元祖。

对于每个表文件，其对应的VM文件命名为:“关系表OID_vm”。对该文件的操作在visibility-map. c文件中进行了定义。

```
13  * INTERFACE ROUTINES
14  *      visibilitymap_clear  - clear bits for one page in the visibility map
15  *      visibilitymap_pin    - pin a map page for setting a bit
16  *      visibilitymap_pin_ok - check whether correct map page is already pinned
17  *      visibilitymap_set    - set a bit in a previously pinned page
18  *      visibilitymap_get_status - get status of bits
19  *      visibilitymap_count  - count number of bits set in visibility map
20  *      visibilitymap_prepare_truncate -
21  *          prepare for truncation of the visibility map
```

以下是几个与可见性映射表操作相关的函数：

visibilitymap_clear：清除指定页在VM中的特定状态标记。

visibilitymap_pin：锁定映射页以便设置比特位。

visibilitymap_pin_ok：验证指定映射页是否已被正确锁定。

visibilitymap_set：在已锁定的映射页中设置比特位。

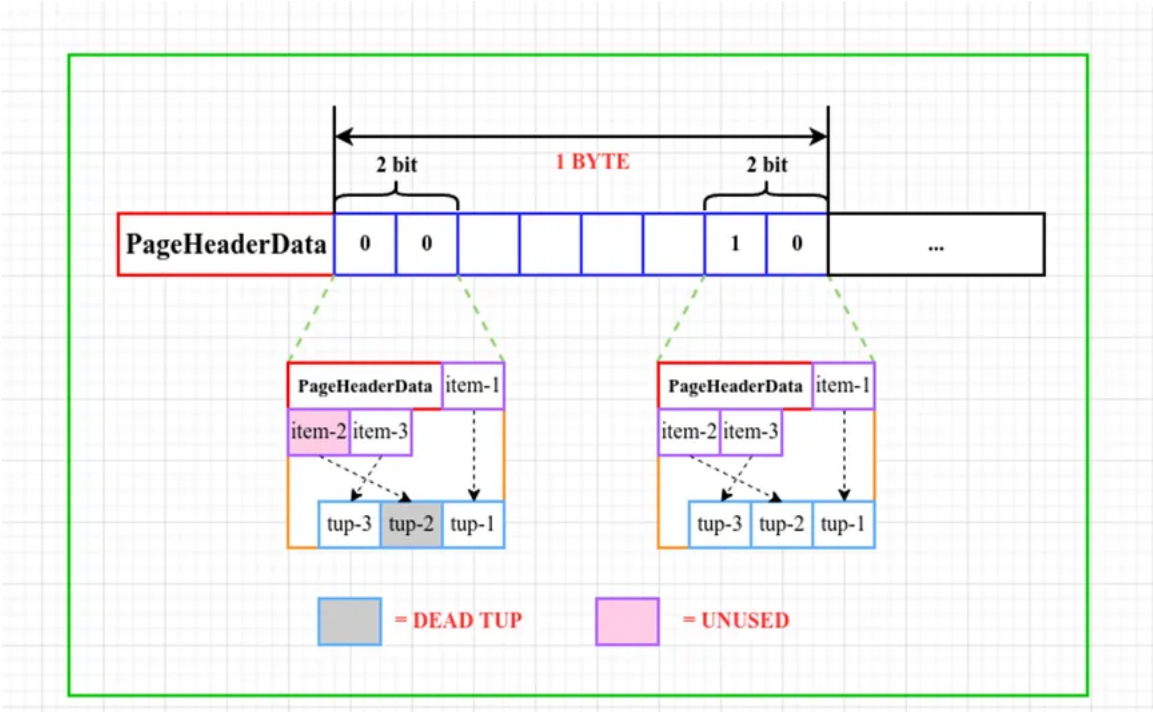
visibilitymap_get_status：查询映射中指定页面的比特位状态。

visibilitymap_count：统计VM中被设置的比特位总数。

visibilitymap_prepare_truncate：为截断VM做好准备工作，包括释放资源或调整大小。

每当事务对表块中的元组进行更新或删除时，对应的VM文件中标志位将被清零，设置标志位前需锁定VM页面，以避免并发修改导致VACUUM判断失误。

VM 文件结构



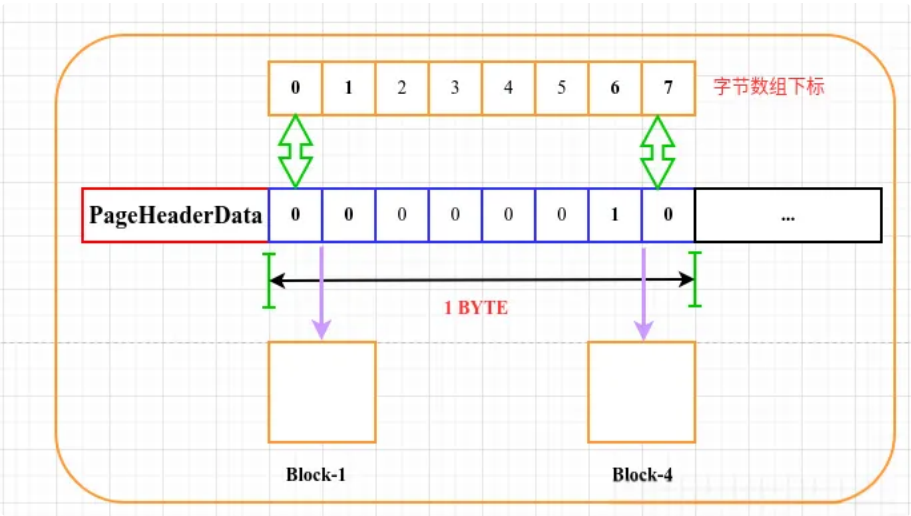
VM中为每个HEAP page设置两个比特位 (all-visible and all-frozen)，分别对应于该页是否存在无效元祖、该页元组是否全部冻结。

all-visible 比特位的设置表明页内所有元组对于后续所有的事务都是可见的，因此该页无需进行vacuum操作；

all-frozen 比特位的设置表明页内所有的元组已被冻结，在进行全表扫描vacuum请求时也无需进行vacuum操作。

NOTES: all-frozen 比特位的设置必须建立在该页已设置过 all-visible比特位。

简单介绍下标识位的写/更新逻辑：



其中比特位的含义如下：

all-visible 比特位： 0 ==> 含有无效元祖　　1 ==> 元组均可见，不含无效元祖

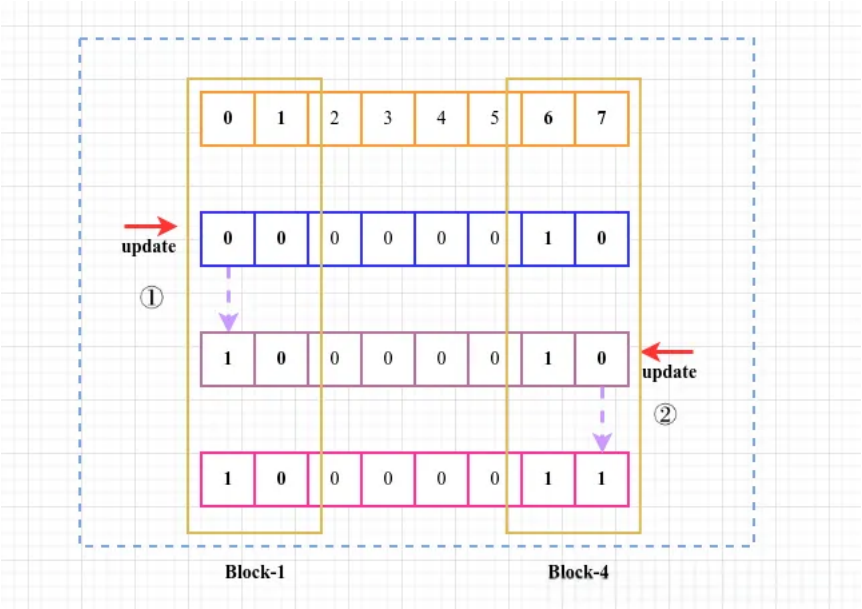
all-frozen 比特位： 0 ==> 含有非冻结元祖　　1 ==> 元组均冻结可见

方便讲述，取自页内的第一个字节示例：

字节对应的二进制信息： 00 00 00 10

根据上述内容可知，heap表的第一页至第三页含有无效元组，第四页没有无效元组

场景：对heap表进行vacuum操作，块1无效元祖被清除，需要设置 all-visible比特位,而块4所有元组冻结



读取数据是以字节为单位，因此通过 char *map数组读取出页内容首地址，通过偏移量确定all-visible 与 all-frozen比特位

1 Block-1对应的比特位为 00， 设置all-visible后更新为 10；

2 Block-4对应的比特位为 10， 设置all-frozen后更新为 11；

宏定义与数据结构

宏定义与数据结构 Plain Text

```
/* 每个heap块在位图中对应的位数 */
#define BITS_PER_HEAPBLOCK 2             // 每个heap块对应 2bits

/* 位图的标志位 */
#define VISIBILITYMAP_ALL_VISIBLE 0x01  // 所有块都可见
#define VISIBILITYMAP_ALL_FROZEN  0x02    // 所有块都被冻结
#define VISIBILITYMAP_VALID_BITS  0x03  /* 所有有效的可见性映射标志位的或运算结果 */

/* 每个可见性映射页上位图的大小（字节），没有额外的头部，所以整个页减去标准的页头都被用于位图 */
#define MAPSIZE (BLCKSZ - MAXALIGN(SizeOfPageHeaderData))    // map页大小

/* 一个字节可以表示的heap块数量 */
#define HEAPBLOCKS_PER_BYTE (BITS_PER_BYTE / BITS_PER_HEAPBLOCK)  // 1 字节对应 4个heap块（假设BITS_PER_BYTE为8）

/* 一个可见性映射页可以表示的heap块数量 */
#define HEAPBLOCKS_PER_PAGE (MAPSIZE * HEAPBLOCKS_PER_BYTE)  // 一个map页对应的heap块数量

/* 将堆块号映射到可见性映射中的正确位 */
#define HEAPBLK_TO_MAPBLOCK(x) ((x) / HEAPBLOCKS_PER_PAGE)    // 计算堆块对应的可见性映射页号
#define HEAPBLK_TO_MAPBYTE(x) (((x) % HEAPBLOCKS_PER_PAGE) / HEAPBLOCKS_PER_BYTE)  // 计算堆块在可见性映射页中的字节偏移
#define HEAPBLK_TO_OFFSET(x) (((x) % HEAPBLOCKS_PER_BYTE) * BITS_PER_HEAPBLOCK)   // 计算堆块在字节中的位偏移

/* 用于计算可见性映射中位集合的掩码 */
#define VISIBLE_MASK64  UINT64CONST(0x5555555555555555) /* 每个位对的低位 */
#define FROZEN_MASK64 UINT64CONST(0xaaaaaaaaaaaaaaaa) /* 每个位对的高位 */

/* 读取不包含行指针的文件页内容的宏 */
/*
 1. PageGetContents：用于在不包含行指针的页的情况下获取页内容。
 2. 注意：在8.3版本之前，这并不保证会返回一个MAXALIGN'd（内存对齐）的结果。
 3. 现在它保证了。请注意旧代码可能认为内容偏移只是SizeOfPageHeaderData，而不是MAXALIGN(SizeOfPageHeaderData)。
*/
#define PageGetContents(page) \
  ((char *) (page) + MAXALIGN(SizeOfPageHeaderData))  // 计算页内容（数据部分）的起始地址
```

当标志位为1时，VACUUM会忽略扫描对应的表块，所以能大大提高VACUUM的效率。由于VM文件不跟踪索引，所以对索引的操作还是需要完全扫描。

当对某个表块中的元组进行更新或删除后，该表块在VM文件中对应的标志位将被置为0。在设置标志位时，需要对其对应的VM页面加锁。

可以通过**create extensiong pg_visibility;select pg_visibility_map(pg_static)；**语句查询状态可见性。**pg_visibility_map（）**函数的实现原理如下：

1） 首先打开 relation即VM文件，随后执行check_relation_relkind函数，此处只支持RELKIND_RELATION 、 RELKIND_INDEX 、 RELKIND_MATVIEW 、 RELKIND_SEQUENCE 、 RELKIND_TOASTVALUE几种类型。

2） 再通过pg_visibility_tupdesc组装出tup的描述 。一般情况下，tupdesc能够直接在系统表里获取。而此处由于数据格式固定，因此需要自行生成 。

3） 然后通过visibilitymap_get_status获取页面的可见性信息。函数逻辑为：传入blocknumber并计算其偏移量，然后读取对应的buffer和内容并返回状态信息。

4） 将返回的状态信息与VISIBILITYMAP_ALL_VISIBLE、VISIBILITYMAP_ALL_FROZEN分别进行对比得到两个布尔值，将布尔值组装成DATUM值返回，并解析成（1,t,t,）、（2,f,f,）形式以显示可见状态。

接口函数

1 visibilitymap_set

该函数的主要功能是设置可见性标识位，其执行流程如下：

1）首先进行安全性校验，判断传入的heap buf 和 vmbuf是否有效以及buf中缓存页是否一一对应；

2）获取VM页内容首地址（跳过PageHeaderData），获取vmbuf的BUFFER_LOCK_EXCLUSIVE；

3）如果之前没有设置过相应的标识位，进行如下操作：

(1) 进入临界区，在指定bit位设置信息，将vmbuf标记为脏；

(2) 写WAL日志，如果开启wal_log_hints，需要将此日志号的LSN更新至heap 页后中；最后更新vmbuf缓存页的LSN，并退出临界。

4）释放vmbuf 持有的排他锁。

visibilitymap_set Plain Text

```
// 设置可见性映射中指定堆块的状态
void
visibilitymap_set(Relation rel, BlockNumber heapBlk, Buffer heapBuf,
          XLogRecPtr recptr, Buffer vmBuf, TransactionId cutoff_xid,
          uint8 flags)
{
  // 计算堆块对应的可见性映射块号
  BlockNumber mapBlock = HEAPBLK_TO_MAPBLOCK(heapBlk);
  // 计算堆块在可见性映射页中的字节偏移
  uint32    mapByte = HEAPBLK_TO_MAPBYTE(heapBlk);
  // 计算堆块在字节中的位偏移
  uint8   mapOffset = HEAPBLK_TO_OFFSET(heapBlk);
  // 可见性映射页
  Page    page;
  // 指向可见性映射数据的指针
  uint8    *map;

#ifdef TRACE_VISIBILITYMAP // 如果定义了TRACE_VISIBILITYMAP，则输出调试信息
  elog(DEBUG1, "vm_set %s %d", RelationGetRelationName(rel), heapBlk);
#endif

  // 断言：在恢复模式下或recptr是无效的XLogRecPtr
  Assert(InRecovery || XLogRecPtrIsInvalid(recptr));
  // 断言：在恢复模式下或heapBuf是一个有效的缓冲区
  Assert(InRecovery || BufferIsValid(heapBuf));
  // 断言：传入的flags是有效的可见性映射标志位
  Assert(flags & VISIBILITYMAP_VALID_BITS);
  //如果这两个条件都不满足，那么断言将失败，并通常会导致程序终止，同时输出一个错误消息，指出断言失败的位置和原因。

  // 检查传入的heapBuf是否指向正确的堆块
  if (BufferIsValid(heapBuf) && BufferGetBlockNumber(heapBuf) != heapBlk)

    elog(ERROR, "wrong heap buffer passed to visibilitymap_set");

  // 检查传入的vmBuf是否指向正确的可见性映射块
  if (!BufferIsValid(vmBuf) || BufferGetBlockNumber(vmBuf) != mapBlock)
    elog(ERROR, "wrong VM buffer passed to visibilitymap_set");

  // 获取可见性映射页
  page = BufferGetPage(vmBuf);
  // 获取可见性映射数据
  map = (uint8 *) PageGetContents(page);
  // 对可见性映射页加排他锁
  LockBuffer(vmBuf, BUFFER_LOCK_EXCLUSIVE);
```

```
44
45      // 如果当前flags与可见性映射中存储的状态不同
46      if (flags != (map[mapByte] >> mapOffset & VISIBILITYMAP_VALID_BITS))
47      {
48        // 开始临界区
49        START_CRIT_SECTION();
50
51        // 设置新的flags到可见性映射中
52        map[mapByte] |= (flags << mapOffset);
53        // 标记可见性映射页为脏页，需要写回磁盘
54        MarkBufferDirty(vmBuf);
55
56        // 如果该关系需要写WAL（Write-Ahead Logging）
57        if (RelationNeedsWAL(rel))
58        {
59          // 如果recptr是无效的，则生成一个新的WAL记录
60          if (XLogRecPtrIsInvalid(recptr))
61          {
62            Assert(!InRecovery);
63            recptr = log_heap_visible(rel->rd_node, heapBuf, vmBuf,
64                                      cutoff_xid, flags);
65
66            // 如果启用了数据校验和（或wal_log_hints=on），我们需要保护堆页不被撕裂
67            if (XLogHintBitIsNeeded())
68            {
69              Page    heapPage = BufferGetPage(heapBuf);
70
71              // 调用者应该先设置PD_ALL_VISIBLE
72              Assert(PageIsAllVisible(heapPage));
73              // 设置堆页的LSN（Log Sequence Number）
74              PageSetLSN(heapPage, recptr);
75            }
76          }
77          // 设置可见性映射页的LSN
78          PageSetLSN(page, recptr);
79        }
80
81        // 结束临界区
82        END_CRIT_SECTION();
83      }
84
85      // 解锁可见性映射页
86      LockBuffer(vmBuf, BUFFER_LOCK_UNLOCK);
87    }
```

2 visibilitymap_get_status

1.首先判断vmbuf是否有效，如果有效，则进一步其缓存的页是否为heap块对应页，若对应关系不匹配，则释放vmbuf pin;

2.若无效，则调用 vm_readbuf 将vm页加载至缓冲块中并返回vmbuf,若返回vmbuf无效，则返回false后退出；

3.紧接着读取vm页首地址，根据偏移量读取相应的标识位信息；

这里只需要pin 机制，无需加 BUFFER_LOCK_SHARE

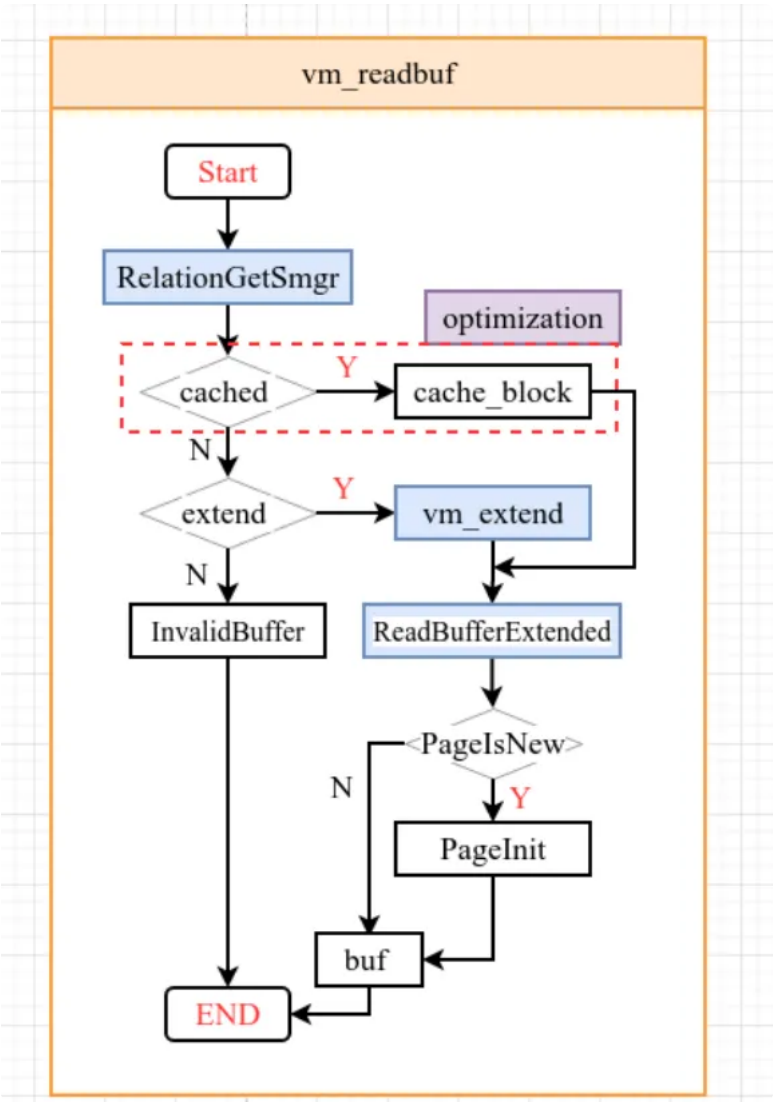
visibilitymap_get_status Plain Text

```
// 函数：visibilitymap_get_status
// 作用：从给定的关系中获取指定堆块的可见性映射状态
// 参数：
//   - rel: 关系（表或索引）的描述符
//   - heapBlk: 堆块的块号
//   - buf: 指向缓冲区的指针，用于存储或重用可见性映射块的缓冲区
// 返回值：
//   - 堆块的可见性状态（uint8 类型）

uint8
visibilitymap_get_status(Relation rel, BlockNumber heapBlk, Buffer *buf)
{
  // 计算可见性映射块号
  BlockNumber mapBlock = HEAPBLK_TO_MAPBLOCK(heapBlk);
  // 计算可见性映射字节在块中的位置
  uint32    mapByte = HEAPBLK_TO_MAPBYTE(heapBlk);
  // 计算偏移量（即位偏移量），用于从字节中提取特定堆块的可见性状态
  uint8   mapOffset = HEAPBLK_TO_OFFSET(heapBlk);
  // 指向可见性映射块的指针
  char     *map;
  // 存储结果的变量
  uint8   result;

  // 如果启用了TRACE_VISIBILITYMAP宏，则输出调试信息
#ifdef TRACE_VISIBILITYMAP
  elog(DEBUG1, "vm_get_status %s %d", RelationGetRelationName(rel), heapBl
k);
#endif

  // 尝试重用旧的已固定缓冲区（如果可能）
  // 如果缓冲区有效但包含的不是我们需要的可见性映射块，则释放它并设置为无效
  if (BufferIsValid(*buf))
  {
    if (BufferGetBlockNumber(*buf) != mapBlock)
    {
      ReleaseBuffer(*buf);
      *buf = InvalidBuffer;
    }
  }

  // 如果缓冲区无效或包含的不是我们需要的可见性映射块，则读取所需的块到缓冲区
  if (!BufferIsValid(*buf))
  {
    *buf = vm_readbuf(rel, mapBlock, false); // 从磁盘读取可见性映射块到缓冲区
```

```
      // 如果读取失败（例如，磁盘错误或块不存在），则返回false
      if (!BufferIsValid(*buf))
        return false;
    }

    // 获取缓冲区中可见性映射块的内容
    map = PageGetContents(BufferGetPage(*buf));

    // 从映射字节中读取特定堆块的可见性状态
    // 这是一个原子操作，因为读取单个字节通常是原子的
    result = ((map[mapByte] >> mapOffset) & VISIBILITYMAP_VALID_BITS);
    // 返回结果
    return result;
}
```

3 vm_readbuf

vm_readbuf 函数的功能是负责将指定VM页加载至缓冲区中，若有需要会进行extend生成新页并进行初始化。其执行流程图如下：



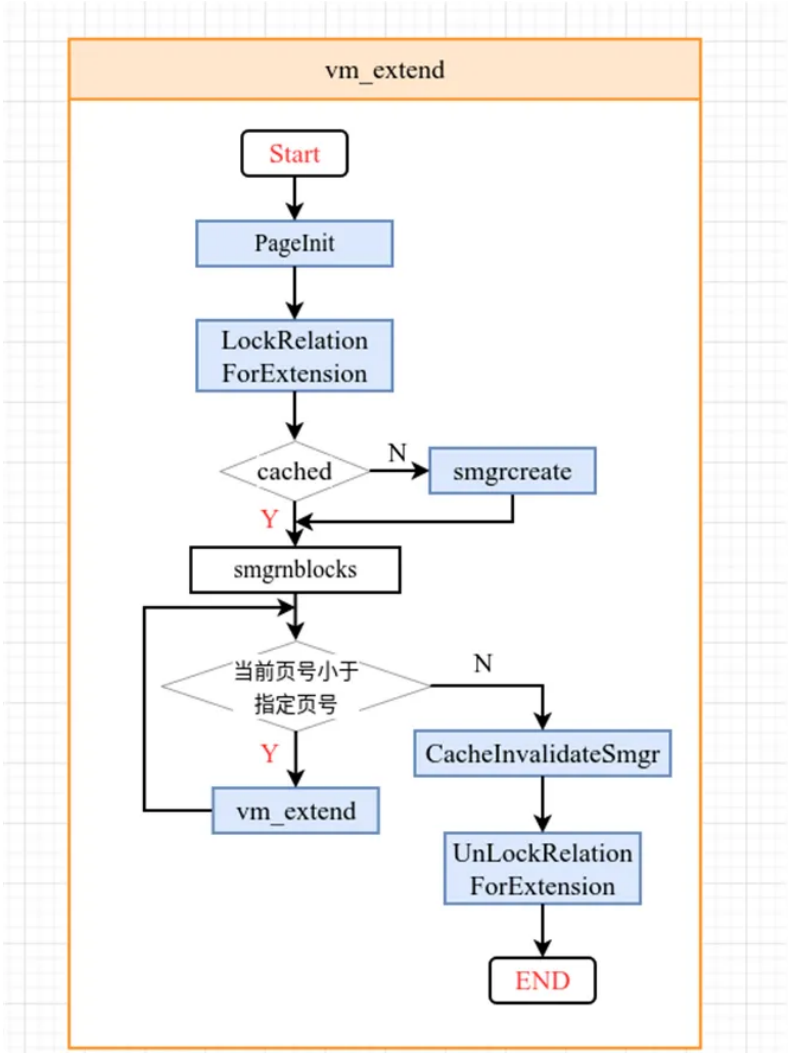
vm_readbuf Plain Text

```
// 函数：vm_readbuf
// 作用：读取或初始化可见性映射（visibility map）的指定块到缓冲区
// 参数：
//   - rel: 关系（表或索引）的描述符
//   - blkno: 要读取的块号
//   - extend: 如果请求的块超出当前范围，是否进行扩展
// 返回值：
//   - 成功时返回包含请求块的缓冲区，失败时返回InvalidBuffer

static Buffer
vm_readbuf(Relation rel, BlockNumber blkno, bool extend)
{
  Buffer    buf; // 缓冲区指针
  SMgrRelation reln; // 存储管理关系的描述符

  // 获取关系对应的存储管理关系
  reln = RelationGetSmgr(rel);

  // 首先检查是否缓存了对应fork（VM）的页
  if (reln->smgr_cached_nblocks[VISIBILITYMAP_FORKNUM] == InvalidBlockNumber)
  {
    // 如果还没有缓存的数量信息，则检查是否存在，并缓存块的数量
    if (smgrexists(reln, VISIBILITYMAP_FORKNUM)) // 判断是否存在VM fork
      smgrnblocks(reln, VISIBILITYMAP_FORKNUM); // 获取并缓存VM fork的块数
    else
      reln->smgr_cached_nblocks[VISIBILITYMAP_FORKNUM] = 0; // 不存在，则设置缓存的块数为0
  }

  // 处理超出EOF（文件末尾）的请求
  // 如果请求的页号超出了VM fork现有的最大页号，并且指定了扩展，则进行扩展
  if (blkno >= reln->smgr_cached_nblocks[VISIBILITYMAP_FORKNUM])
  {
    if (extend)
      vm_extend(rel, blkno + 1); // 调用vm_extend进行扩展，注意参数是blkno+1，因为扩展会创建下一个块
    else
      return InvalidBuffer; // 不扩展，则返回InvalidBuffer表示失败
  }

  // 常规流程：从共享缓冲池中选择一个缓冲块来缓存指定的VM页面
  // 如果是新页面，则获取BUFFER_LOCK_EXCLUSIVE锁后再次检查页面是否为NEW
  //  （进行两次判断是因为可能有其他进程在本进程申请锁时已经完成了初始化）
  buf = ReadBufferExtended(rel, VISIBILITYMAP_FORKNUM, blkno,
```

```
                RBM_ZERO_ON_ERROR, NULL); // 读取或创建缓冲块，并可能进行零初始化

  // 如果页面是新页面（刚刚分配的）
  if (PageIsNew(BufferGetPage(buf)))
  {
    // 获取独占锁以初始化页面
    LockBuffer(buf, BUFFER_LOCK_EXCLUSIVE);

    // 再次检查页面是否仍然是新的（防止其他进程在此期间已经初始化了页面）
    if (PageIsNew(BufferGetPage(buf)))
      PageInit(BufferGetPage(buf), BLCKSZ, 0); // 初始化页面

    // 释放独占锁
    LockBuffer(buf, BUFFER_LOCK_UNLOCK);
  }

  // 返回包含请求块的缓冲区
  return buf;
}
```

4 vm_extend

当访问的vm页在文件中不存在时，此时需调用vm_extend函数扩展新页并完成相应的初始化工作，其执行流程图如下：


vm_extend
Start
PageInit
LockRelation
ForExtension
cached
N
smgrcreate
Y
smgrnblocks
当前页号小于
指定页号
N
Y
CacheInvalidateSmgr
vm_extend
UnLockRelation
ForExtension
END

vm_extend Plain Text

```
// 函数：vm_extend
// 作用：扩展可见性映射（visibility map）以包含更多的块
// 参数：
//   - rel: 关系（表或索引）的描述符
//   - vm_nblocks: 要扩展到的块数
// 返回值：无（void）

static void
vm_extend(Relation rel, BlockNumber vm_nblocks)
{
  BlockNumber vm_nblocks_now; // 当前可见性映射的块数
  PGAlignedBlock pg; // 用于存储新页面数据的对齐块
  SMgrRelation reln; // 存储管理关系的描述符

  // 注意：这里的PageInit调用位置是错误的，因为pg尚未初始化，并且它应该在循环内部为每个新块初始化

  // 锁定关系以进行扩展操作
  LockRelationForExtension(rel, ExclusiveLock);

  // 获取关系的存储管理关系
  reln = RelationGetSmgr(rel);

  // 如果当前没有缓存VM的块数信息或者VM fork不存在
  // 则创建VM fork
  if ((reln->smgr_cached_nblocks[VISIBILITYMAP_FORKNUM] == 0 ||
     reln->smgr_cached_nblocks[VISIBILITYMAP_FORKNUM] == InvalidBlockNumber) &&
     !smgrexists(reln, VISIBILITYMAP_FORKNUM))
     smgrcreate(reln, VISIBILITYMAP_FORKNUM, false); // 创建VM fork

  // 清除缓存的块数信息，因为我们即将更新它
  reln->smgr_cached_nblocks[VISIBILITYMAP_FORKNUM] = InvalidBlockNumber;

  // 获取当前VM fork的块数
  vm_nblocks_now = smgrnblocks(reln, VISIBILITYMAP_FORKNUM);

  // 循环直到当前块数达到或超过请求的块数
  while (vm_nblocks_now < vm_nblocks)
  {
    // 初始化一个新的页面（注意：这里应确保pg在使用前已分配并初始化）
    // PageInit应该在这个循环内部，并且应该在smgrextend之前

    // 假设pg已经初始化，这里仅为页面设置校验和
    PageSetChecksumInplace((Page) pg.data, vm_nblocks_now);
```

```
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
```

```
        // 将新页面扩展到VM fork
        smgrextend(reln, VISIBILITYMAP_FORKNUM, vm_nblocks_now, pg.data, fals
    e);

        // 更新当前块数
        vm_nblocks_now++;
      }

      // 清除与关系相关的存储管理关系的缓存
      // 这确保其他进程或事务可以看到最新的块数信息
      CacheInvalidateSmgr(reln->smgr_rnode);

      // 释放对关系的锁定
      UnlockRelationForExtension(rel, ExclusiveLock);
    }
```

1）首先页面初始化，填充PageHeader结构体pd_lower、pd_upper/和flag初始信息；

2）获取relation的extension锁，防止其他进程进行同样的扩展工作；

3）如果文件不存在，则调用 smgrcreate进行创建，反之进入第4）步；

4）获取当前vm块号，如果当前块号小于指定快号，则需在此调用vm_extend进行扩展（递归调用）；

5）向其他进程发送无效消息强制其关闭对rel的引用，其目的是避免其他进程对此文件的create或者extension,因为这写操作容易发生。

6）最后释放锁资源；

2.5、大的数据存储管理

大数据的存储使用TOAST 机制和大对象机制来实现， 前者主要用千变长字符串， 后者则主要用于大尺寸的文件。

要理解TOAST，我们要先理解**页（BLOCK）**的概念。在PG 中，页是数据在文件存储中的基本单位，其大小是固定的且只能在编译期指定，之后无法修改，默认的大小为8KB。同时，PG 不允许一行数据跨

页存储。那么对于超长的行数据，PG 就会启动TOAST，将大的字段压缩或切片成多个物理行存到另一张系统表中**（TOAST 表）**，这种存储方式叫行外存储。

2.5.1、TOAST表结构

●前4 字节（32bit）称为长度字，长度字后面存储具体的内容或一个指针。

● 长度字的高2bit 位是标志位，后面的30bit 是长度值（表示值的总长度，包括长度字本身，以字节计）。

● 由长度值可知TOAST 数据类型的逻辑长度最多是30bit，即1GB(2^30–1 字节）之内。

● 前2bit 的标志位，一个表示压缩标志位，一个表示是否行外存储，如果两个都是零，那么表示既未压缩也未行外存储。

TOAST 表有三个字段：

● 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（这里的blog ）的TOAST字段的一行数据。

● chunk_seq —— 用来表示该行数据在整个数据中的位置。

● chunk_data —— 该Chunk 实际的数据。

2.5.2、TOAST操作

在PG 中每个表字段有四种TOAST 的策略：

● PLAIN —— 避免压缩和行外存储。只有那些不需要TOAST 策略就能存放的数据类型允许选择（例如int 类型），而对于text 这类要求存储长度超过页大小的类型，是不允许采用此策略的。

● EXTENDED —— 允许压缩和行外存储。一般会先压缩，如果还是太大，就会行外存储。这是大多数可以TOAST 的数据类型的默认策略。

● EXTERNAL —— 允许行外存储，但不许压缩。这让在text 类型和bytea 类型字段上的子串操作更快。类似字符串这种会对数据的一部分进行操作的字段，采用此策略可能获得更高的性能，因为不需要读

取出整行数据再解压。

● MAIN —— 允许压缩，但不许行外存储。不过实际上，为了保证过大数据的存储，行外存储在其它方式（例如压缩）都无法满足需求的情况下，作为最后手段还是会被启动。因此理解为：尽量不使用行外存储更贴切。

实验：见第五实验部分(5.3、TOAST)

2.5.3、大对象

随着信息技术的发展，数据库需要存储的数据对象变得越来越大，为了解决大对象的存储PostgreSQL提供了大对象存储机制。支持三种数据类型的存储:

1. 二进制大对象(BLOB):主要用来保存非传统数据，如图片、视频、混合媒体等。
2. 字符大对象(CLOB):存储大的单字节字符集数据，如文档等。
3. 双字节字符大对象(DBCLOB):用于存储大的双字节字符集数据，如变长双字节字符图形字符串。
   这些类型可以保存诸如音视频、图片以及文本等大数据量的对象，最大可支持2G的尺寸。
   所有的大对象都存在一个名为pg_largeobject的系统表中。每一个大对象还在系统表pg_largeobject_metadata中有一个对应的项。大对象可以通过类似于标准文件操作的读/写API来进行创建、修改和删除。

SQL

```
postgres=# create table image(name text, raster oid);
CREATE TABLE
postgres=# select lo_creat(-1);
lo_creat
----------
49727
(1 row)
```

创建一个新的大对象。其返回值是分配给这个新大对象的OID或者InvalidOid（0）表示失败。

SQL

```
postgres=# select lo_create(43213);
lo_create
-----------
43213
(1 row)
```

尝试创建OID为43213的大对象，返回值是分配给新大对象的OID或InvalidOid（0）表示发生错误。lo_create是从PostgreSQL 8.1版本中开始提供的函数，如果该函数在旧服务器版本上运行，它将失败并返回InvalidOid。

```sql
postgres=# select lo_unlink(43213);
lo_unlink
-----------
1
(1 row)
```

TOAST 和大对象虽然都是对大数据进行存储的技术，但两者有一定的区别。

首先，TOAST 用于存储变长的数据，如 VARCHAR 变量等。只要表中有变长的数据类型，都会自动地创建 TOAST 表，但只有存入的数据长度超过 2KB，才会触发 TOAST 存储机制。而大对象操作是客户端通过代码或者 SQL 命令调用的。因此，两者的第一个区别在于 TOAST 是可变长数据类型的一种大数据存储机制，属于自动触发机制；而大对象属于用户手动调用机制。

其次，TOAST 中的数据不能丢失，一旦丢失则会报错；而大对象中允许数据丢失，如果丢失，则用 0 替代。

第三，文件不适合使用 TOAST 技术进行存储，需要将文件以二进制方式读出，然后将二进制当作字符串存储到变长数据类型的属性中。而读取时则需要将数据读取出来，然后再转变成文件。而大对象操作不一样，它直接将文件作为一个对象存储到大对象表中，读取的时候也直接读取成一个文件，大对象对文件的存储非常容易通过编程来实现。

最后，TOAST 和大对象操作保存大数据时都是采用了将数据切成片段存储到表中的方式。但 TOAST 提供了线外和压缩两种存储机制，而大对象只是对数据不做处理直接存储。