《数据库系统原理》大作业实验报告

PostgreSQL 分析

题目名称： Postgresql 索引与内外存管理

学号及姓名：21371455 宁然

21371249 潘卓然

21371173 张博豪

21373216 李嘉鹏

2023 年 12 月 22 日

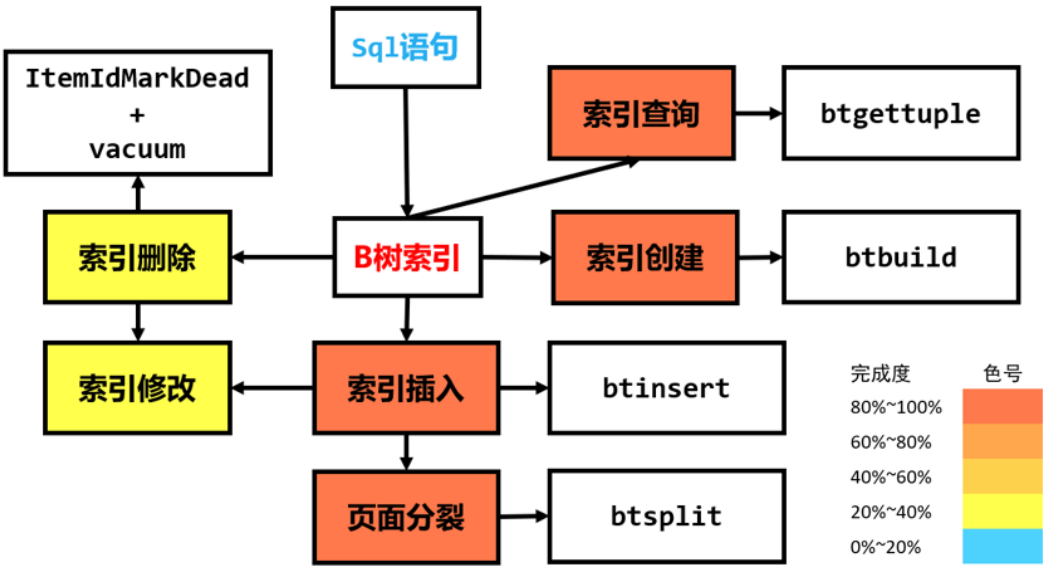
组内同学承担任务说明

学生姓名	工作内容			工作量占比（组内同学总和为 1）
	子任务 1：	子任务 2：	子任务 3：	
宁然	B 树索引相关的重要数据结构解析	B 树索引创建插入删除修改等操作的核心逻辑以及一些与其他部分的对接接口	索引相关示例 SQL 语句的执行过程解析	
潘卓然	内存物理描述结构解析	内存逻辑描述结构解析	内存相关 SQL 语句执行流解析	
张博豪	内存管理架构总览，内存上下文源码分析	Drop table persons 语句执行流程分析	SMGR 存储管理器源码分析，串接内外存	
李嘉鹏	外存总体框架速览，VFD 机制，大数据存储机制	VM 文件与 Vacuum 机制解析	外存相关 SQL 语句执行流解析，内外存交互的数据结构视角与数据流视角解析，外存管理与操作系统的关系	

一．子系统结构与功能概述

（一）、索引

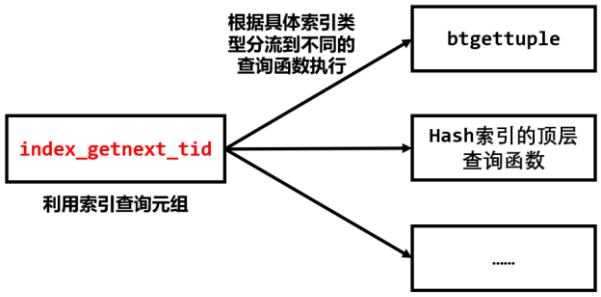
下面这张图可以概括索引部分的主要工作内容，这张图本身全都是索引解析工作的主要覆盖模块。下面结合下图概括性地说一下索引子系统的大体功能。



当执行一个 sql 语句时，如果其涉及到了和 b 树索引有关的内容。那么执行流就会来到和 B 树索引有关的代码处，也就是图中标记为红色的地方。然后根据不同的索引操作要求执行不同部分的索引相关代码：索引插入、查询、创建、修改、删除。正如图中所示，插入由 btinsert 这一顶层函数完成，查询用 btgettuple（也可以认为是 index_get_nexttid，这个在报告后面会讲到）顶层函数完成，创建用 btbuild 函数完成，插入由 btinsert 函数完成，其中可能涉及到的分裂过程用 btsplit 函数实现，索引修改由“删除+插入”的组合操作实现，索引的删除通过将索引元组标记为“dead”，并且在之后触发 vacuum 操作时统一清除的方式实现。

可以注意到，索引删除部分的解析深度可能还不够，比如还没有对 btvacuumcleanup 等与“vacuum-索引”相关的函数进行细腻的代码阅读和分析。也因此，作为“删除+插入”结合的索引修改部分的逻辑并没有解析地很透彻。不过，索引查询、创建、插入、分裂这些部分均有着自认为较高的完成度，不过其中也还尚有着没有完全解析的一些细节，比如遍历时的快速路径优化等部分。此外，除了 b 树索引本身的内容外，可能还可以解析的和

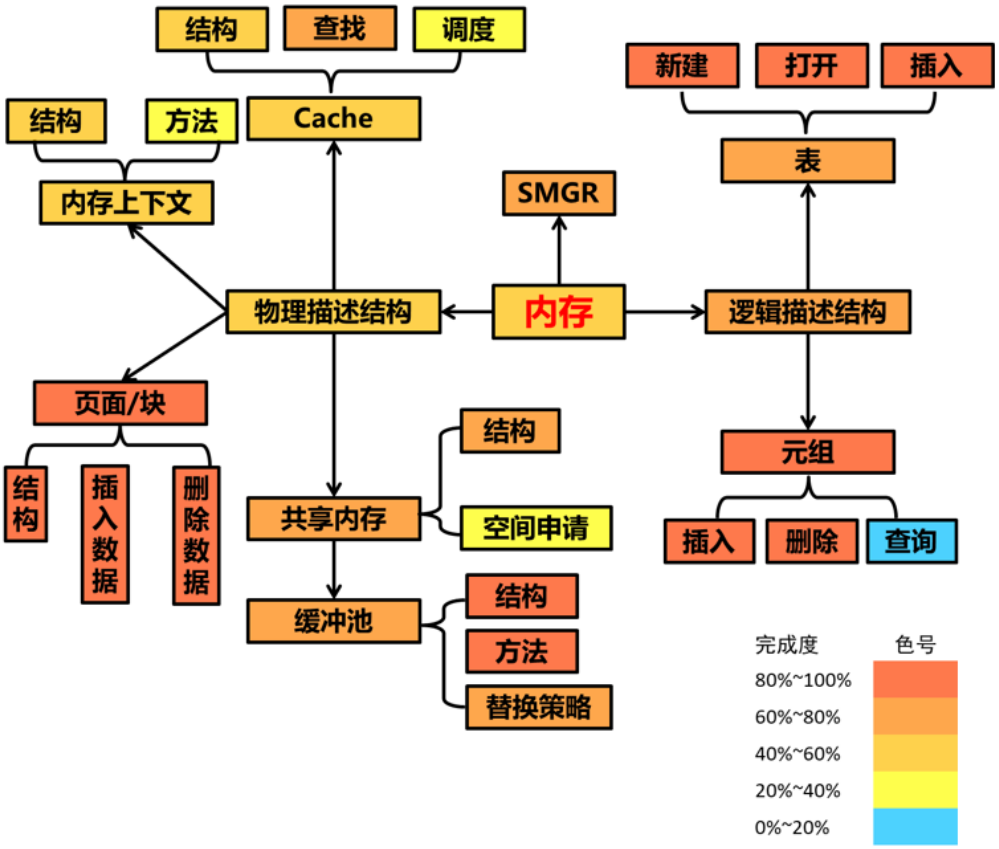
索引相关的内容有：hash 索引等其他类型的索引各方面的核心逻辑，各类型索引“共用代码”的逻辑，比如下图，做索引查询时会先调用 index_getnext_tid（命名中带“index”而不带和具体索引类型相关的字眼的，就是各类型索引共享的公共代码），然后才到 btgettuple（带有 bt 相关字眼的命名，才是和 btree 索引有关的）。“分流”（实际上是用了一个函数指针来调用和具体索引类型相关的函数）之前的代码尚未详细解析。



（二）、内存

内存是 PostgreSQL 在运行时不可或缺的底层支持模块；内存空间中，具有不同组织架构的模块和数据（内存上下文，缓冲池，Cache …），但它们在底层物理结构基元上又是高度统一的（页/块）；由于内存模块的各个组件功能性较为独立，各个模块详细的功能描述和解析将放在第二节（解析结果）部分详细描述。

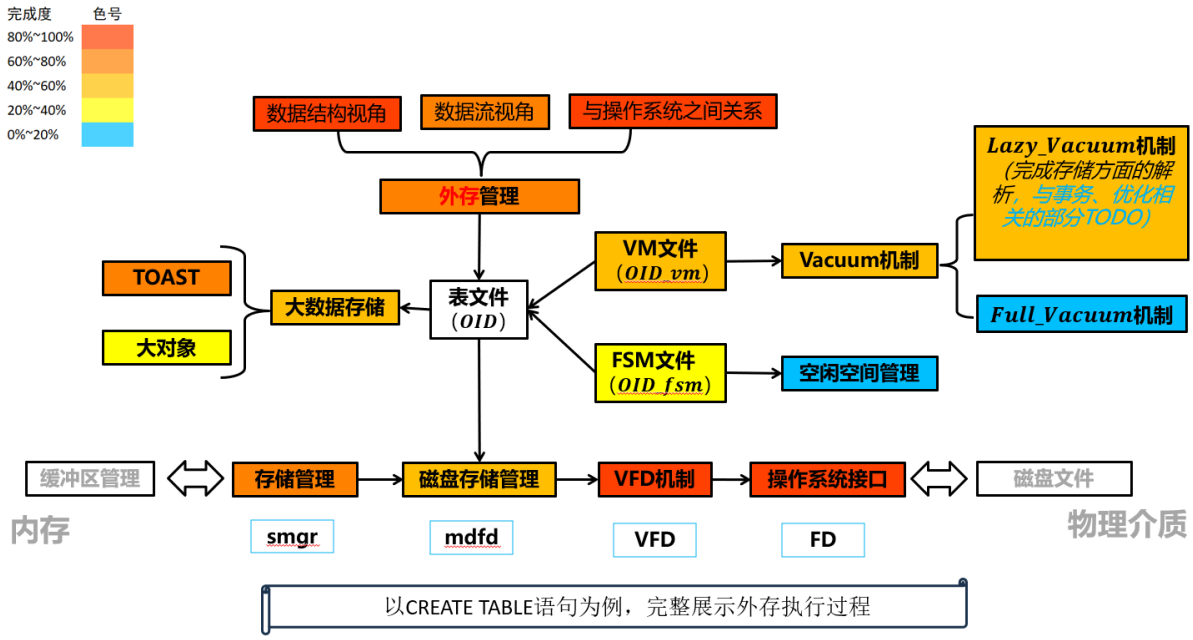
以下是内存部分较为重要的结构的概览和我们的解析工作的完成情况：



在本学期的解析工作中，我们对内存模块中的物理描述结构中的页面/块、共享内存、缓冲区部分做了较为详细的解析，对于内存上下文、Cache 两模块的解析并未过于深入，其中内存上下文封装的内存申请和使用方法、Cache 结构和调度的具体细节还待探索；在逻辑描述结构模块，我们对表和元组的解析较为深入，但对于元组的查询操作这一块并未深究，这一方面后续可以和负责查询的小组进行联动；在 SMGR (存储管理器)部分，解析了 SMGR 控制内外存交互的部分和动态 Hash 机制，但 SMGR 涉及外存与操作系统部分的调用细节还需进一步探究。

（三）、外存

外存管理，衔接着 DBMS 内存管理和操作系统对物理文件的操作，是数据交互的桥梁，从与物理介质最接近的视角来管理数据。该模块的解析图如下以及解析工作的完成情况如下:



外存管理的解析工作，可以从**数据管理**与**数据交互**两大板块来理解。

1） 数据管理：在解析图中围绕“表文件”横向展开。主要包括两个辅助表文件，FSM（空闲空间映射表）和 VM（可见性映射表）以及借助这两个辅助表工具进行的“空闲空间管理”与“Vacuum 机制”（垃圾回收以及事务冻结等）。还包括大数据存储管理。

2） 数据交互：在解析图中围绕“表文件”纵向展开，以及图的中下部横向的数据交互流图。主要包括存储管理，磁盘存储管理，虚拟文件管理等。

后续工作：

1） 数据管理方面：

- 可以对 FSM（空闲空间映射表）与“空闲空间管理”进行解析探究。
- Lazy_vacuum 机制的存储方面：可以在与内存管理删除操作的串联上进行完善补充；
- Lazy_vacuum 机制的非存储方面：比如事务可见与冻结，更新表相关信息以备优化，可以进行解析探究。
- 可以对 Full_ vacuum 机制进行解析探究，可以梳理其函数调用脉络，但深入探究似乎可能有些细枝末节。
- 可以对大数据存储的操作层面进行解析，但可能有些细枝末节。

2）数据交互方面：

- 可以进一步解析 mdfd，smgr

- 可以选择更多的语句操作，来进行解析，展示外存管理在其中的作用

二．解析结果

（一）、索引

（1）. 重要数据结构解释

这里先放一些解析过程中遇到的一些比较关键的数据结构：

- 索引元组 IndexTupleData，其中包含了 t_tid 和 t_info 两个属性，其中 t_tid 中含有一个块号 ip_blkid 和一个块内偏移号 ip_posid，这个结构体描述的就是索引元组所指向的位置，也就是索引元组的指针。t_tid 中的 t_info 中的各个二进制位描述了一些关于本索引元组的信息，具体哪些位表示什么样的信息见下图中的注释。这个索引元组本身的键值信息实际上是存储在 IndexTupleData 这个结构体之后的空间中的（尾随其后），调用宏函数 #define index_getattr(tup, attnum, tupleDesc, isnull)，可以以“基地址+偏移”的方法，拿出这个 IndexTupleData 结构体后面的该索引元组的各个键值具体值的信息。

```
typedef struct IndexTupleData
{
        // 这个结构体存了该索引指向的目标(可能是表中的数据也可能是低一级索引块)的信息
        ItemPointerData t_tid;          /* reference TID to heap tuple */

        /* ---------------
         * t_info is laid out in the following fashion:
         *
         * 15th (high) bit: has nulls          该元组是否为null
         * 14th bit: has var-width attributes  标记是否有可变长度的属性
         * 13th bit: AM-defined meaning        未使用
         * 12-0 bit: size of tuple                          索引元组的大小
         * ---------------
         */

        unsigned short t_info;          /* various info about tuple */ // 关于索引元组的一些信息

} IndexTupleData;                               /* MORE DATA FOLLOWS AT END OF STRUCT */
```

其中的 t_tid 的类型的全貌如下，也就是可以理解为是一个用来定位元组的指针。

```
typedef struct ItemPointerData // tid是元组的id
{
        BlockIdData ip_blkid; // 块号
        OffsetNumber ip_posid; // 偏移，由块号定位到具体块，然后由块内偏移定位到具体元组
}
```

- 扫描键 BTScanInsertData，在扫描 b 树时使用这个数据结构的键值来和树上的索引元组的键值比较。其中相对关键的属性为 scankeys(索引列属性的值数组)

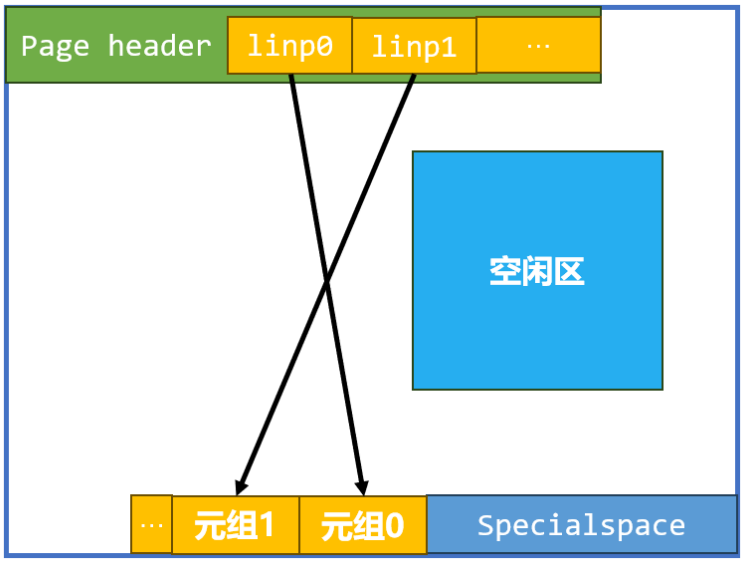
```
typedef struct BTScanInsertData
{
        bool            heapkeyspace;
        // true表明重复删除数据是安全的
        bool            allequalimage;
        // keys中是否有null值
        bool            anynullkeys;
        // 决定_bt_search和_bt_binsrch的比较条件
        // false -> 大于等于   true -> 大于, 决定是>high key时右移还是>=high key时右移
        bool            nextkey;
        bool            pivotsearch; // 大多情况都是false
        ItemPointer scantid;            /* tiebreaker for scankeys */ // 存着Tid, 唯一标志物理元组
        int                     keysz;                  /* Size of scankeys array */
        // 这是存储了查询时要关注的, 要去比较的信息,
        // 在哪上面创建索引, 这里面就包含什么的信息, 因为别的属性没创建索引,
        // 在查询的时候也不会被用到
        ScanKeyData scankeys[INDEX_MAX_KEYS];   /* Must appear last */
} BTScanInsertData;
```

- ScanKeyData 这是扫描键中的单个属性对应的结构体，其中值得重点关注的属性是 sk_argument 和 sk_func。前者是一个 Datum 类型的变量，其就是一个指针，指向该属性的具体的值。后者是一个 FmgrInfo 结构体（全称应是 function manger infomation），其中比较关键的属性是一个函数地址，可以简单地将其理解成是一个函数指针，其指向的函数就定义了这个属性列的值之间的比较规则（比如对于字符串类型的变量按字典序大小比较等）

```
// [key1, key2, key3, ...] 这个结构体的一个就对应一个key, 插入扫描键中有多个key当然也就要有一个scankey的数组
typedef struct ScanKeyData
{
        int     sk_flags;               /* flags, see below */
        AttrNumber      sk_attno;               /* table or index column number */
        // 比较策略号: < <= > >= ... 经常采用默认比较方式, 少数情况下需要手动指定比较方式
        StrategyNumber sk_strategy; /* operator strategy number */
        Oid     sk_subtype;             /* strategy subtype */
        Oid     sk_collation;   /* collation to use, if needed */
        // 比较函数, 定义了在该属性的两个值之间的比大小规则
        FmgrInfo sk_func;               /* lookup info for function to call */
        // 真实的数值, 都是用Datum指针的方式存的
        Datum sk_argument;      /* data to compare */
} ScanKeyData;
```

- PageHeaderData。这是一个很常用到的结构体，对于每一个页面，其开头都有一个这样的数据结构用来描述和这个页面相关的一些信息，其中比较重要的属性为 pd_linp 数组，这个数组中的元素和本页面上存在的索引元组一一对应，每个元素都指向页内一个索引元组的位置，也即下面第二幅图展示的结构。

```c
typedef struct PageHeaderData
{
        /* XXX LSN is member of *any* block, not only page-organized ones */
        PageXLogRecPtr pd_lsn;          /* LSN: next byte after last byte of xlog
                                                * record for last change to this page */
        uint16          pd_checksum;    /* checksum */
        uint16          pd_flags;               /* flag bits, see below */
        // 空闲空间的开始位置的偏移(这个东西就是一个整数而已，表示偏移量)
        LocationIndex pd_lower;         /* offset to start of free space */
        // 空闲空间的结束位置的偏移
        LocationIndex pd_upper;         /* offset to end of free space */
        // special空间位置的偏移
        LocationIndex pd_special;       /* offset to start of special space */
        uint16          pd_pagesize_version;
        TransactionId pd_prune_xid; /* oldest prunable XID, or zero if none */
        // 页内一个个数据项的id数组
        ItemIdData      pd_linp[FLEXIBLE_ARRAY_MEMBER]; /* line pointer array */
} PageHeaderData;

typedef PageHeaderData *PageHeader;
```



- BTPageOpaqueData 同样是对于每个页面都有一个的结构体，其中的属性也是对本页面相关信息的描述。比如其中比较常用到的两个属性，btpo_prev 和 btpo_next。他们分别表示该页面的左右兄弟的位置，也就是 b 树上的左右指针。

```c
// btpo->b tree page opaque
typedef struct BTPageOpaqueData // B-Tree中，左右节点的信息 和 节点的类型
{
        BlockNumber btpo_prev;          /* left sibling, or P_NONE if leftmost */ // 前一块块号
        BlockNumber btpo_next;          /* right sibling, or P_NONE if rightmost */ // 后一块块号
        uint32          btpo_level;             /* tree level --- zero for leaf pages */ // 页面在索引树上的层级
        uint16          btpo_flags;             /* flag bits, see below */ // 这里存了很多表示页面特征的标记位
        BTCycleId       btpo_cycleid;   /* vacuum cycle ID of latest split */ // 页面对应的最新的
} BTPageOpaqueData;

typedef BTPageOpaqueData *BTPageOpaque;
```

- BTPageState 结构体。其中，btps_page 是比较关键的属性，即为创建索引树时每一层的，未插满索引元组的，最右边的那一个节点页面，当 btps_page 表示的页

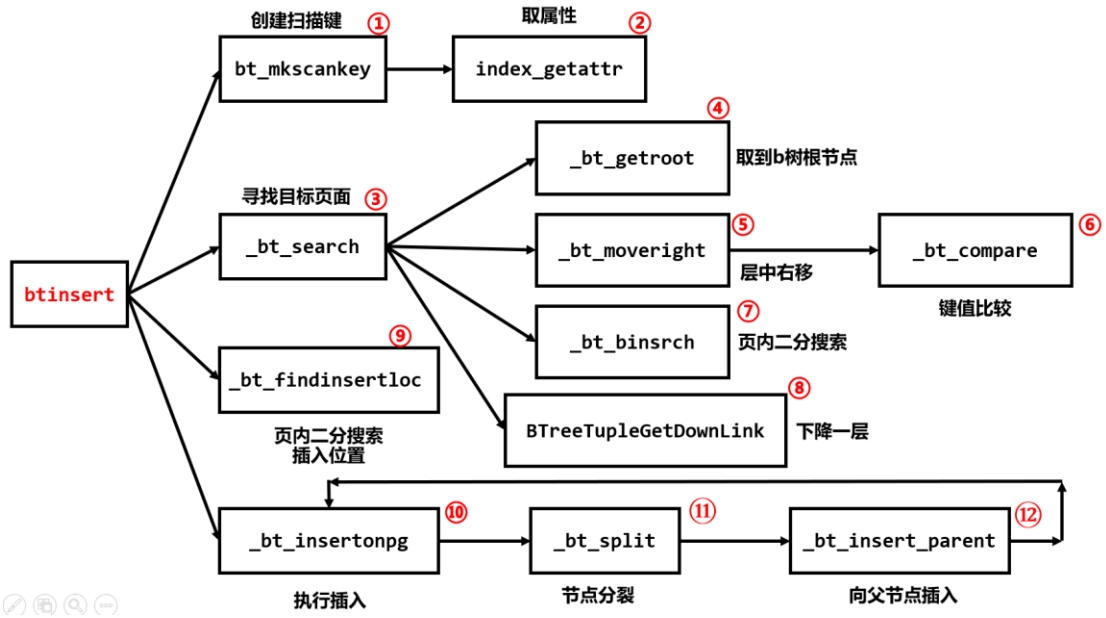
面插满后，该属性就会指向新的 new 出来的页面，始终保持自己指向本层有剩余可插入空间的页面。

```
// 在创建索引树的过程中，树上的每一层都有且仅有一个BTPageState这样的结构
typedef struct BTPageState
{
        // 当前进行插入操作的页面
        Page            btps_page;              /* workspace for page building */
        // 页内的偏移号
        BlockNumber btps_blkno;         /* block # to write this page at */
        // 页中键值最小的索引元组
        IndexTuple btps_lowkey; /* page's strict lower bound pivot tuple */
        // 页中最后插入的索引元组的块号
        OffsetNumber btps_lastoff;      /* last item offset loaded */
        Size btps_lastextra; /* last item's extra posting list space */
        // 该页在树中的层数
        uint32 btps_level;              /* tree level (0 = leaf) */
        // 页面允许的最小空闲空间，和空闲因子有关
        Size btps_full;         /* "full" if less than this much free space */
        // 指向更高一层的BTPagestate结构的指针
        struct BTPageState *btps_next;  /* link to parent level, if any */
} BTPageState;
```

（2）. 模块/函数运行的数据流和处理流剖析与说明。

1) . 索引树的插入函数 btinsert

完成 b 索引树插入索引元组任务的顶层函数是 btinsert(…), 从该函数起的函数调用图如下所示：



插入的流程解释如下：

Step1: 首先用 bt_myscankey 函数创建扫描键(BTScanKey 结构体)，在这个过程中

我们需要循环用 index_getattr 这个宏函数去取出元组的和索引有关的属性值来将它们赋给扫描键中的属性，以便后面用扫描键去和索引树上的索引元组做比较。同时在构建扫描键的过程中也会给每个索引属性列一个比较函数，这个比较函数就是定义某种类型的属性的值应该如何进行比较（比如：定义字符串类型的属性值按照字典序来进行比较等）。

Step2: 调用_bt_search 函数在 b 树上进行遍历搜索，以找到在保持 b 树的有序的性质下刚才构建出来的扫描键应该插入到的目标页面节点。其中具体的算法会在后面剖析算法的部分详细解释。

Step3: 通过_bt_search 找到目标插入页面后，用_bt_findinsertloc 函数来确定目标页面内部应该插在的具体偏移号。由于 b 树节点页面中的若干个索引元组是有序排列的，因此为了追求更高的查找效率，这个过程通过二分查找这一常见算法实现。

Step4: 最后如果找到的目标页面有空余空间以供插入，则用_bt_insertonpg 函数在上面找到的具体插入位置处，执行将待插入的索引元组加入到目标位置的操作。这个函数其实就是调用了名为 PageAddItem 的宏函数（这个函数在本报告中也是做了详细解释了的），其将待插入的索引元组的内容 memcpy 到页面的对应位置上，同时维护页面 PageHeaderData 结构中的 linp 数组，将新插入索引元组的 ItemIdData 结构插入到 linp 数组对应的位置上，完成插入元组的操作。

Step5: 最后如果找到的目标页面没有空余空间以供插入，则此时调用_bt_insertonpg 函数会触发页面分裂，分裂的重要函数是_bt_split 函数。触发分裂时，原页面上的所有索引元组和原本待插入的索引元组会被分配到分裂后的左右页面上，然后用_bt_insert_parent 函数将指向分裂后的右页面的索引元组插入到父页面中，维护好父子之间的指针，彻底完成分裂。分裂的算法会在后文进行详细解析。

2) . 索引树创建函数 btbuild

完成 b 索引树创建任务的顶层函数是 btbuild(…), 从该函数起的函数调用图（该图同时能展示出函数嵌套关系）如下所示：

```
IndexBuildResult *btbuild(...);
    _bt_spools_heapscan(heap, index, &buildstate,
    indexInfo);
    void _bt_leafbuild(BTSpool *btspool,...)
        tuplesort_performsort(btspool->sortstate);
        _bt_load(&wstate, btspool, btspool2);
          if (state == NULL)
              state = _bt_pagestate(wstate, 0);
            _bt_buildadd(wstate, state, itup, 0);
```

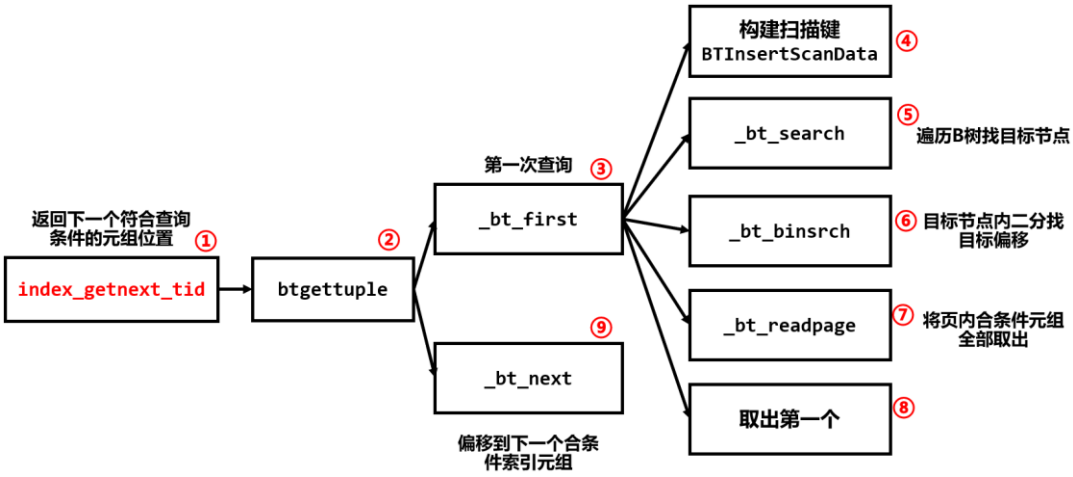
对该函数的执行流进行解释如下：

a. 在 btbuild 函数内部，首先执行_bt_spools_heapscan 函数来扫描一个个的表元组然后把他们封装成一个个索引元组的数据结构(IndexTuple)。

b. 然后 btbuild 内部再调用_bt_leafbuild 函数，并于其中调用 tuplesort_performsort 函数按照索引键值对索引元组进行排序为之后将他们一个个插入索引树上做准备。

c. 接着在_bt_leafbuild 函数中调用_bt_load 函数，这个函数就基本是我们建树的核心逻辑了，它的功能可以大体理解为：将单个索引元组插入到树上对应的位置。那么循环调用这个函数就可以将所有的索引元组都插入到树上了.我们的核心算法就体现在将这一个个索引元组循环插入树上的过程中。具体的插入算法在之后的部分会详解。

3). 索引树的查询过程



走 B 树索引查询的顶层函数是 index_get_next_tid，其定义为：

```
ItemPointer
index_getnext_tid(IndexScanDesc scan, ScanDirection direction)
```

顾名思义，其作用就是返回 B 树上下一个符合查询条件的索引元组的 ItemPointer,（这

个结构体的定义在上面已经展示过了）。通过该 ItemPointer 指针我们就可以找到拿到符合查询条件的索引元组了，继而便可以通过此索引元组中的指针拿到其所指向的表元组。结合上面的从 index_getnext_tid 开始的函数调用图来对这个函数的主干逻辑进行解析如下：

- 调用 btgettuple 函数（看 index_getnext_tid 函数名中并不带和 btree 有关的字眼便不难得知这个函数其实是各种类型的索引都会调用的，而对于 btree 索引来讲，该函数其中用函数指针来调用的下一层函数便是 btgettuple，这个函数才是仅属于 btree 索引的）。开始主干逻辑的执行。
- btgettuple 函数第一次被调用，内部执行_bt_first 函数。
- _bt_first 函数首先依据查询条件构建扫描键，和前面讲的索引插入中的扫描键的含义相同。在后续的 b 树遍历过程中会拿这个扫描键来不断和树上索引元组的键值进行比较，从而决定是要往什么样的方向（向右移还是向下移）去向目标页面靠近。
- 构建扫描键后调用函数_bt_search，这个函数在索引插入的讲解中也已经进行了解释，其会依据扫描键来找到树上的目标页面。
- 调用_bt_binsrch 函数在目标页面内二分查找，找到目标页面内符合查询条件的第一个索引元组的偏移 offnum。
- 调用_bt_readpage 函数从上一步得到的第一个位置开始往后遍历，将当前页面后面的所有符合查询条件的索引元组全部都 save(保存)到 scan->opaque 这个 BTScanOpaque 结构体变量中。

  其核心代码如下所示。就是遍历取出 offnum 偏移处及之后的每个索引元组，用函数_bt_checkkeys 来判断遍历到的索引元组是否符合查询条件(比较键值，这个过程在前面的索引插入部分也讲解过)。符合条件那么就存下来，否则结束循环。

```
static bool _bt_readpage(IndexScanDesc scan, ScanDirection dir, OffsetNumber offnum)
{
    while (offnum <= maxoff) // maxoff就是页内最大偏移号
    {
        ItemId          iid = PageGetItemId(page, offnum);
        IndexTuple      itup;
        itup = (IndexTuple) PageGetItem(page, iid); // itup是此轮遍历到的索引元组

        if (_bt_checkkeys(scan, itup, indnatts, dir, &continuescan))
        {
            /* tuple passes all scan key conditions */
            _bt_saveitem(so, itemIndex, offnum, itup);
            itemIndex++;
        }
        if (!continuescan)
            break; // 未通过检查，也就是当前遍历到的索引元组不满足查询条件，那就break了
        offnum = OffsetNumberNext(offnum);
    }
}
```

- 执行如下代码。

```
scan->xs_itup = (IndexTuple) (so->currTuples + currItem->tupleOffset);
```

令得 scan 结构体中的 xs_itup 等于刚才存下来的那些符合条件的索引元组中的第一个元组。以“保存在结构体中”这种形式返回，调用它的函数可以从此结构体中读出这个被修改后的属性，取出第一个符合条件的索引元组。

- 至此，btgettuple 第一次执行完毕。当再次针对 btree 调用 index_get_nexttid 函数，进而再次调用 btgettuple 函数时，直接将上一步中的 scan -> xs_itup 元素向后偏移一项， 即可直接获取到下一个符合条件的索引元组。（由于第一次调用 btgettuple 函数时，已经将所有符合条件的索引元组都存进了 scan->opaque 中，所以现在再次调用 btgettuple 函数不需要再次花费时间来遍历 b 树了，直接向后偏移一个单位即可）。

4). 索引删除

这部分没有很独特的算法。其大体流程如下：

- 每当利用索引去取出一个表元组时，会根据表元组状态标记 scan->kill_prior_tuple。

```
bool index_fetch_heap(IndexScanDesc scan, TupleTableSlot *slot)
{
        bool            dead = false;
        bool            found;
        found = table_index_fetch_tuple(scan->xs_heapfetch,
                &scan->xs_heaptid, scan->xs_snapshot, slot,
                        &scan->xs_heap_continue, &dead);
        // all_dead为true, 表示该索引元组需要被标记为dead
        scan->kill_prior_tuple = dead;
        return found;
}
```

- 如果上一步的 kill_prior_tuple 标记位被标记成了 true，那么之后将其对应的索引元组加入到 so->killedItems 数组中，顾名思义，killItems 就是要杀死的索引元组的清单。

```
bool btgettuple(IndexScanDesc scan, ScanDirection dir)
{
    BTScanOpaque so = (BTScanOpaque) scan->opaque;
    if (scan->kill_prior_tuple) // 为真, 则说明此元组应该被kill
    {
        so->killedItems[so->numKilled++] = so->currPos.itemIndex;
    }
}
```

- 调用_bt_killitems 函数为索引元组加上 LP_DEAD 的标记。下面的代码的核心在于下图中红框框出的一句代码。ItemIdMarkDead 是一个函数，其将参数传进的某个索引元组对应的 ItemId 变量中的标志位置为 lp_dead，表示这个索引元组已经死亡，不过不会立即做出处理，会待到之后 vacuum 机制触发后统一对死亡的元组进行清楚，以获得更好的效率。

```
void _bt_killitems(IndexScanDesc scan)
{
    for (i = 0; i < numKilled; i++)
    { // 遍历killitems数组
        int itemIndex = so->killedItems[i];
        BTScanPosItem *kitem = &so->currPos.items[itemIndex];
        while (offnum <= maxoff)
        { // 遍历索引元组项，来找和kill item一样的那个索引元组，找到后将其标记为lp_dead
            ItemId              iid = PageGetItemId(page, offnum);
            IndexTuple  ituple = (IndexTuple) PageGetItem(page, iid);
            bool                killtuple = false;
            if (ItemPointerEquals(&ituple->t_tid, &kitem->heapTid))
                killtuple = true;
            if (killtuple)
            {
                ItemIdMarkDead(iid);
                killedsomething = true;
                break;                  /* out of inner search loop */
            }
            offnum = OffsetNumberNext(offnum);
        }
    }
}
```

5). 索引修改

索引的修改是通过“删除+插入”这个组合实现的。删除和插入操作上文均已经进行了解释。故而，修改就是通过标记索引元组状态为 LP_DEAD 来实现对索引元组的删除，再通过 btinsert 函数来插入新的索引元组，如此一来，便可达到修改的效果。

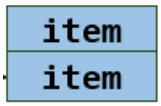
（3）. 关键索引算法剖析。

1) . b 索引树创建算法详解

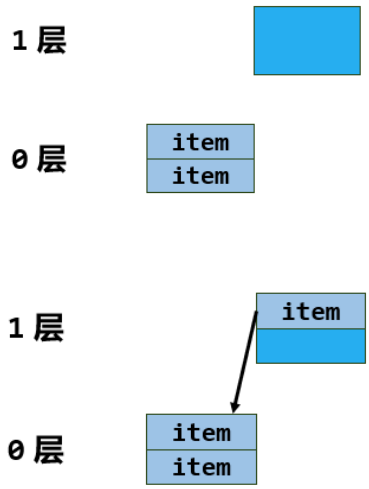
这里详细讲解将一个个索引元组往树上插入的流程，这是一个自底向上构建树的过程，也是创建 b 索引树过程中的算法核心所在。具体如下。

Step1: 首先, new 出一个树节点(树上第一个被创建的节点)，为插入第一个元组做准备。(其实是 new 一个 PageState 结构体,这个结构体中有一个 Page 类型的属性，所以也可以粗浅地先理解为是 new 出了一个页面节点，这个 PageState 结构体对于树的每一层都会有且仅有一个, 类似于是一个“层管理员”)

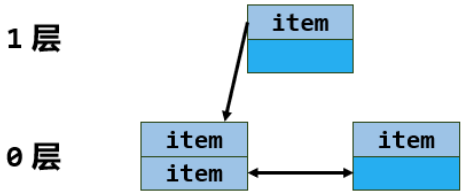
Step2: 按顺序将元组插入到新 new 出的节点上，直到插满（在示意图中，我们假设一个节点最多只能插两个元组）。



Step3: 树中任何层的任意一个页面节点插满后还想继续插入（记插满的这个节点名字为 full 节点），我们都访问相比本层更高一层的“层管理员”，如果没有更高一层，则 new 出一个新的更高一层的“层管理员” PageState 结构体（如下面图一）。然后我们向更高一层的“层管理员”PageState 结构体中的 Page 指针属性指向的页面中插入以 full 节点的 low key 为键值（注意！如果这一步也因为插满了而难无法进行时，重新开始执行 Step3 的逻辑，这里是递归自下至上建树的核心!），指向 full 节点的索引元组（如下面图二，直观理解即为，创建树上父节点指向子节点的指针）



Step4: 维护好父节点后，new 一个右兄弟 Page，将待插入的元组插入到右兄弟 Page 中，并将这一层的“层管理员”PageState 结构体中的 Page 指针指向新 new 出来的兄弟 Page（表示下次再在这一层插入，就插这个新的没满的页面）



Step5: 重新回到第二步，其中描述的“新 new 出的节点”始终指的是第 0 层的“层管理员”PageState 中的 Page 指针属性指向的节点。

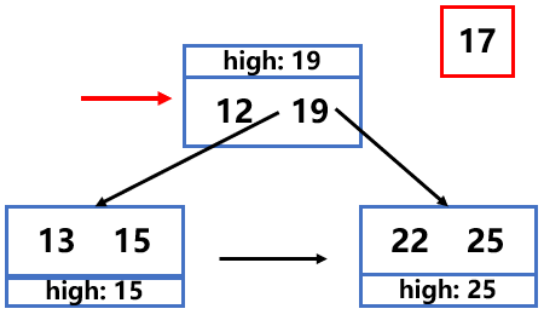
2). b 索引树遍历算法详解

本部分主要讲解无论插入还是查询时都会涉及到的 b 树的遍历过程，也就主要是

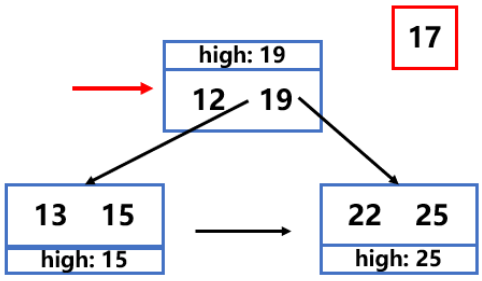
_bt_search 这个函数的执行过程。具体如下。

注意！当下降到叶子页面时，该算法终止。

Step1: _bt_getroot 函数获取 b 树的树根页面 Page 指针（指向树根页面的位置）。认为当前遍历到的页面 curPage = rootPage

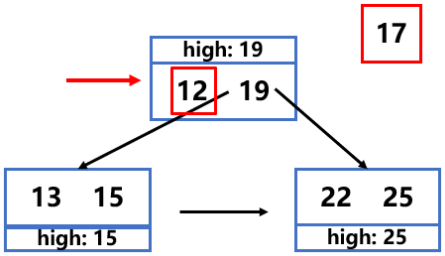


Step2: 在 curPage 所在的树层中，将 curPage 不断右移，即执行 curPage = curPage->right_next（伪代码）。右移的条件是，curPage 并不是当前层的最右边的节点（即，curPage->right_next != null），而且，扫描键中的键值比 curPage 的 high key 索引元组的键值更大（页面内的索引元组按照自身的键值大小有序排放，high key 就是某个页面中键值最大的那一个索引元组）。注意，上面这里有“比较”二字，我们如何对两个键值进行“比较”呢，这里就是用到了我们之前提到过的 ScanKeyData 结构中的 sk_func 函数指针，该指针指向一个定义了单个属性列上两值比较规则的比较函数，不同的属性列可能会有不同的比较函数，向他们传入两个待比较的某个属性的具体值，该函数会返回比较结果。那么对索引扫描键中的属性依次调用该函数去与树上的索引元组中的相对应的属性值比较，即可得出两个键之间的大小关系（循环比较两个键的各个属性值的过程中，如果双方的相应属性值相同，那么就比较下一个属性值，如果不同，那就终止循环，得出比较结果）。



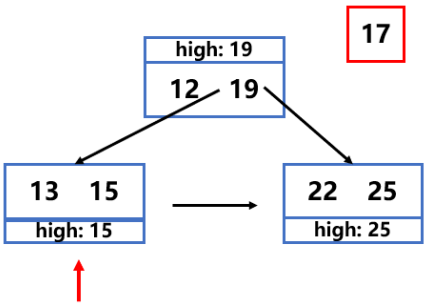
（根节点已经是当层的最右节点，不需再右移）

Step3: 执行_bt_binsrch，在 curPage 页面内的众多索引元组之中进行二分搜索，找到页内从小到大，最后一个键值小于扫描键键值的索引元组，为了方便描述，将此索引元组记为 downIndexTuple。



（找到的目标索引元组就是这个键值为 12 的元组）

Step4: 执行 BTreeTupleGetDownLink，将当前遍历到的页面 curPage 切换为其 downIndexTuple 的指针指向的页面，即：curPage = downIndexTuple -> pointerPage（伪代码）。重新开始执行 Step2。

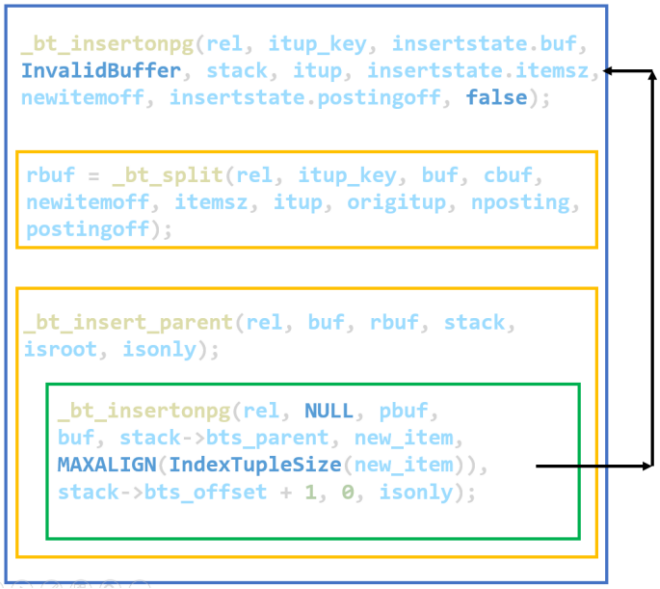


（curPage 指针指向下一层的页面）

至此，我们通过_bt_search 函数遍历 b 树，得到了在保持 b 树有序性的状态下扫描键所应该在的 b 树页面，将这个目标页面作为函数返回值返回。

3). b 索引树节点分裂算法详解

这部分我们主要解释，在调用_bt_insertonpg 函数准备向一个已经没有空间的叶子节点（为了方便表述，将之记为 full_page）中插入索引元组时将要发生的事情。给出与此有关的主要逻辑的函数调用图如下：



结合这个图详细解释一下。

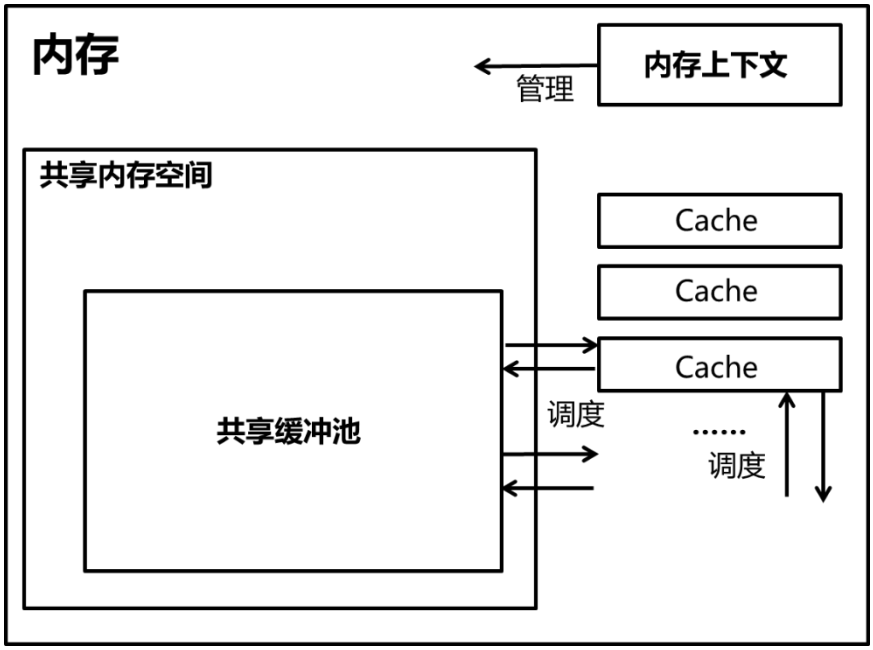
Step1: 首先执行_bt_split 函数，将 full_page 分裂成左右两个 page。

- 首先调用_bt_findsplitloc 函数在 full_page 中选择一个分裂点（用偏移号表示，记为 firstrightoff），后续页面将以此点为界分裂开来。这个函数选取分裂点的标准是要保证分裂后的左右页面索引元组分布尽可能均衡。
- 调用_bt_getbuf 函数 new 出新的左右子页面，分别记为 leftpage 和 rightpage
- 以偏移号遍历取出原始 full_page 上的所有索引元组，将它们和待插入的元组按照有序的原则，以 firstrightoff 为分隔点，分别分配到左右子页面上。
- 调用 PageRestoreTempPage 函数，将之前 new 出来的 leftpage 复制到原本的 full_page 的起始地址上将其覆盖，full_page 成为了分裂后的左子页面。
- _bt_split 至此完毕，注意此时的左子页面由于覆盖了原本的 full_page，其与父页面的指针是不用再维护的，但是右子页面是新 new 出来的，还没有构建其与父节点之间的链接关系，下面的步骤需要构建。

Step2: 调用_bt_insert_parent 函数，创建一个以左子页面的 high key 为键值，指向右子页面的索引元组，将该索引元组调用_bt_insertonpg 函数插入到父节点的相应位置中，以维护父节点和右子页面之间的链接关系。注意！我们这里向父节点中插入索引元组调用的也是_bt_insertonpg 函数，这就形成了递归，意味着如果检测到父节点已经满了，那么就会对父节点使用_bt_split 函数将其进行分裂，这就是我们实现自底向上分裂的核心思想所在。

（二）、内存

以下是与内存相关的一些主要的实体及其功能概述，由于 PostgreSQL 内存部分模块解耦性较为优良，故先在此简述模块间的关系，后续对各个模块进行分别描述：



内存上下文（Memory Context）：

功能简述： 内存上下文是一种用于管理内存分配和释放的机制（一种工具）。它允许 PostgreSQL 在执行查询、事务等操作时有效地分配和管理内存。

描述： 内存上下文是一种内存分配的工作单元，当一个任务完成时，整个上下文可以轻松地被释放，以释放关联的内存。这有助于减少内存泄漏的风险。

缓冲池（Buffer）：

功能简述： 缓冲池是一块划分出的内存区域，用于缓存数据库中的表和索引的数据块，以提高数据访问的速度。

描述： 缓冲池存储了最近被访问过的数据块，如果需要再次访问相同的块，就可以避免从磁盘重新读取。这有助于减少对磁盘的访问，提高数据库性能。

页面（Page）：

功能简述： Page 是缓冲池和外存中的基本单位，用于存储表和索引的数据。

描述： 每个 Page 通常有一个固定的大小，通常是 8KB。当数据库需要读取或写入数据时，它以 Page 为单位进行操作。缓冲池中的 Block 就是这些 Page。

块（Block）

功能简述：Block 是内存中的基本单位，单个块内的空间以内存片的形式进行分配，用于存储表或索引的数据。

描述：每个 Block 大小默认为 8 KB（可以认为改动），这也是内存中 Block 和 Page 通常可以通用的原因。

共享内存（Shared Memory）：

功能简述： 共享内存（shmem）是一种允许不同进程之间共享数据的机制，对于多进程的数据库系统尤为重要。

描述： 在 PostgreSQL 中，共享内存用于存储全局状态信息、锁、进程通信等。这对于多用户并发访问数据库是必要的，因为各个进程需要协调和共享资源；上述的缓冲池就位于共享内存中。

缓存（Cache）：

功能简述： 缓存是一种用于存储常用查询的结果以提高性能的机制。

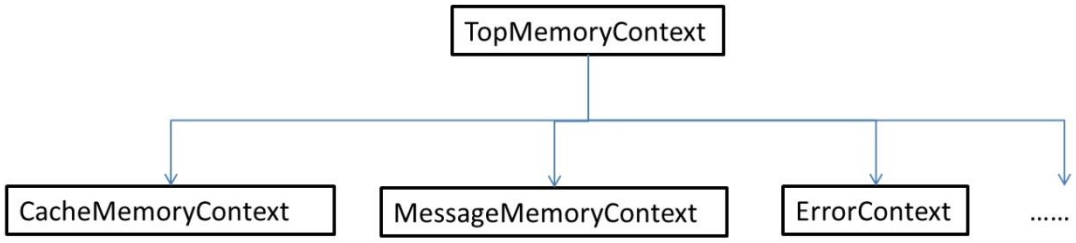
描述： PostgreSQL 中有各种缓存，包括查询结果缓存、计划缓存等。这些缓存存储了先前查询的结果，以便在相同或相似的查询发生时，可以直接返回缓存的结果，而不必重新执行查询。

内存上下文

我们需要强调的是：内存上下文这个机制基本上只服务于内存空间的分配和释放；它不是一种存储实际的表或其他的实体的数据的数据结构，而是一种标准化 PostgreSQL 内存的申请和释放的组织结构，这种设计减少了内存泄露的风险。这是正确理解上下文机制的基础。

内存上下文的外部结构

PostgreSQL 的每个子进程都有多个私有的内存上下文，下面展示的是子进程内存上下文的构成。首先子进程的所有内存上下文构成一个树形结构：



各个种类的内存上下文功能简述：

- TopMemoryContext：

  为上下文树的根节点，在这里进行内存分配相当于 malloc 函数，因为该上下文不会被删除或者重置（存放有可能会一直在内存中的内容，或者控制模块在适当的时候将要删除的内容）

- PostmasterContext：

  是 postmaster 进程的执行环境，当生成一个 postgres 后端进程，该上下文就可以删除，释放不需要的内存。

- CacheMemoryContext：

  负责 relcache、catcache 和相关模块的长期存储，和 TopMemoryContext 一样，也不会被删除或者重置。是因为调试才和 TopMemoryContext 区分开来。有许多的生命周期比较短的子上下文。

- MessageContext：

  负责存储来自前端的当前的命令消息，以及任何生命周期和当前消息一样长的派生存储（在简单查询中，解析树和计划树可以存放于此）。该上下文可以被重置或删除。

- TopTransactionContext：

  保存了所有有关 top-level 事务的的内容，直到 top-level 事务结束。并且结束时，会重置该上下文，所有子上下文都会被删除。

- CurTransactionContext：

  保存了和当前事务相关的数据，直到 top-level 事务结束才会重置，并删除所有子上下文。

- PortalContext：

  实际上不是一个单独的上下文，而是一个指向当前活跃的执行计划的 portal 的每个 portal 上下文的全局变量。当需要分配一个和执行计划生命周期一样长的 portal 时，可

以使用该上下文。关于 portal，我在下面放了解释。

- ErrorContext：

  在错误恢复处理时会切换到这个上下文，恢复完成后，会将该上下文重置。这里总是会保留至少 8KB 的空闲内存，在这种情况下，即便是后端的内存空间不足，也有内存用于错误恢复。这使得内存不足由 fatal error 变为普通的 error。

内存上下文的内部结构

内存上下文内部结构的数据结构记载了上下文关键的信息，如空间分配情况、分配方法等；值得一提的是，该数据结构的组织方法体现了很多面向对象的思想。

（1） 内存上下文头部

存储外部结构的链接和该上下文基本信息

```
typedef struct MemoryContextData *MemoryContext;

typedef struct MemoryContextData
{
    NodeTag     type;           //内存上下文节点类型
    /* 下面两个变量是为了减少内存对齐浪费: */
    bool        isReset;        //在上次重置后是否分配内存空间
    bool        allowInCritSection; //是否允许在临界区使用palloc
    const MemoryContextMethods *methods;
    //用于存放虚拟函数表（ADT的实现），会始终指向AllocSetMethods这个全局变量，原因放在下面的内存上下文操作中。
    MemoryContext parent;       //上下文树中该节点的父节点指针
    MemoryContext firstchild;   //上下文树中该节点的第一个孩子指针
    MemoryContext prevchild;    //上下文树中该节点的左边的兄弟节点的指针
    MemoryContext nextchild;    //上下文树中该节点的右边的兄弟节点的指针
    const char *name;           //上下文的名称（调试时使用）
    const char *ident;          //上下文的ID（调试时使用）
    MemoryContextCallback *reset_cbs;   //重置或删除的回调列表
} MemoryContextData;
```

（2） 内存上下文主体

可以看作继承自 MemoryContextData 的类，存储了详细的空间分配情况和分配的指导性信息：

```
typedef AllocSetContext *AllocSet;//类似与MemoryContext,AllocSet是AllocSetContext的指针

typedef struct AllocSetContext
{
    MemoryContextData header;    //内存上下文的头信息，即上面分析的MemoryContext
    //关于内存分配的信息
    AllocBlock  blocks;          //内存上下文中所有内存块的链表
    AllocChunk  freelist[ALLOCSET_NUM_FREELISTS];    //内存上下文中空闲内存片的数组
    //内存上下文分配时用到的参数
    Size        initBlockSize;   //初始内存块的大小
    Size        maxBlockSize;    //最大内存块的大小
    Size        nextBlockSize;   //下一个要分配的内存块的大小
    Size        allocChunkLimit;     //分配内存片的尺寸阈值，在分配内存片时会遇到
    AllocBlock  keeper;          //保存在这个变量中的内存块在内存上下文重置时会被保留，不被释放。
    //可以将该上下文放入到context_freelists中
    int         freeListIndex;   //即该上下文在context_freelists中的序号，如果未放入则为-1。
} AllocSetContext;
```

- initBlockSize、maxBlockSize：这两个变量会在内存上下文创建的时候就给定，然后会将 nextBlockSize 设置为和 initBlockSize 相同的值。在进行内存分配时，如果需要分配一个新的内存块，那么这个新内存块的大小就会采用 nextBlockSize 的值。
- allocChunkLimit：前面提到的内存片分配受 allocChunkLimit 限制：如果一个内存片的尺寸超过了 allocChunkLimit 时，将会为该内存片单独分配一个独立的内存块，从而避免以后进行内存回收时产生过多的碎片
- Keeper 机制：在内存上下文进行重置时不会对 keeper 中记录的内存块进行释放（free），而是对其内容进行清空（reset）。这样可以保证内存上下文重置结束后就已经包含一定的可用内存空间，避免了重复的 malloc。

（3） 内存上下文操作函数

以函数指针的形式存储内存上下文，可以看做该上下文类的“方法”

```
typedef struct MemoryContextMethods
{
    void       *(*alloc) (MemoryContext context, Size size);//分配内存的函数指针
    void        (*free_p) (MemoryContext context, void *pointer);//释放内存的函数指针
    void       *(*realloc) (MemoryContext context, void *pointer, Size size);//重分配内存的函数指针
    void        (*reset) (MemoryContext context);//重置内存上下文的函数指针
    void        (*delete_context) (MemoryContext context);//删除内存上下文的的函数指针
    Size        (*get_chunk_space) (MemoryContext context, void *pointer);//检查内存片段大小的的函数指针
    bool        (*is_empty) (MemoryContext context);//检查内存上下文是否为空的的函数指针
    void        (*stats) (MemoryContext context,
                          MemoryStatsPrintFunc printfunc, void *passthru,
                          MemoryContextCounters *totals);//打印内存上下文状态的的函数指针

} MemoryContextMethods;
```

内存上下文的关键存储结构

（1） 内存块链表节点：

```
typedef struct AllocBlockData
{
    AllocSet    aset;           //该内存块所位于的AllocSet,即上面的数据结构
    AllocBlock  prev;           //指向链表中上一个内存块的指针
    AllocBlock  next;           //指向链表中下一个内存块的指针
    char       *freeptr;        //指向该内存块中空闲区域的首地址
    char       *endptr;         //指向该内存块的尾地址
}          AllocBlockData;
```

（2） 内存片头部信息：

```
typedef struct AllocChunkData
{
    Size        size;//内存片中的大小
    void       *aset;//链接指针
}          AllocChunkData;
```

其中的链接指针 aset 有两种不同时生效的用途：

- 如果内存片为空闲，则该结构在某一空闲链表（共 11 个空闲链表位于 FreeList 中，每一条中存储的单位空闲内存片大小为 2^(3 + n), n = 0…10，n 为数组下标）中,aset 指向链表中的下一单元
- 如果内存片非空闲，指向该内存片所在的 AllocSet（上文提到的内存上下文主题的指针）

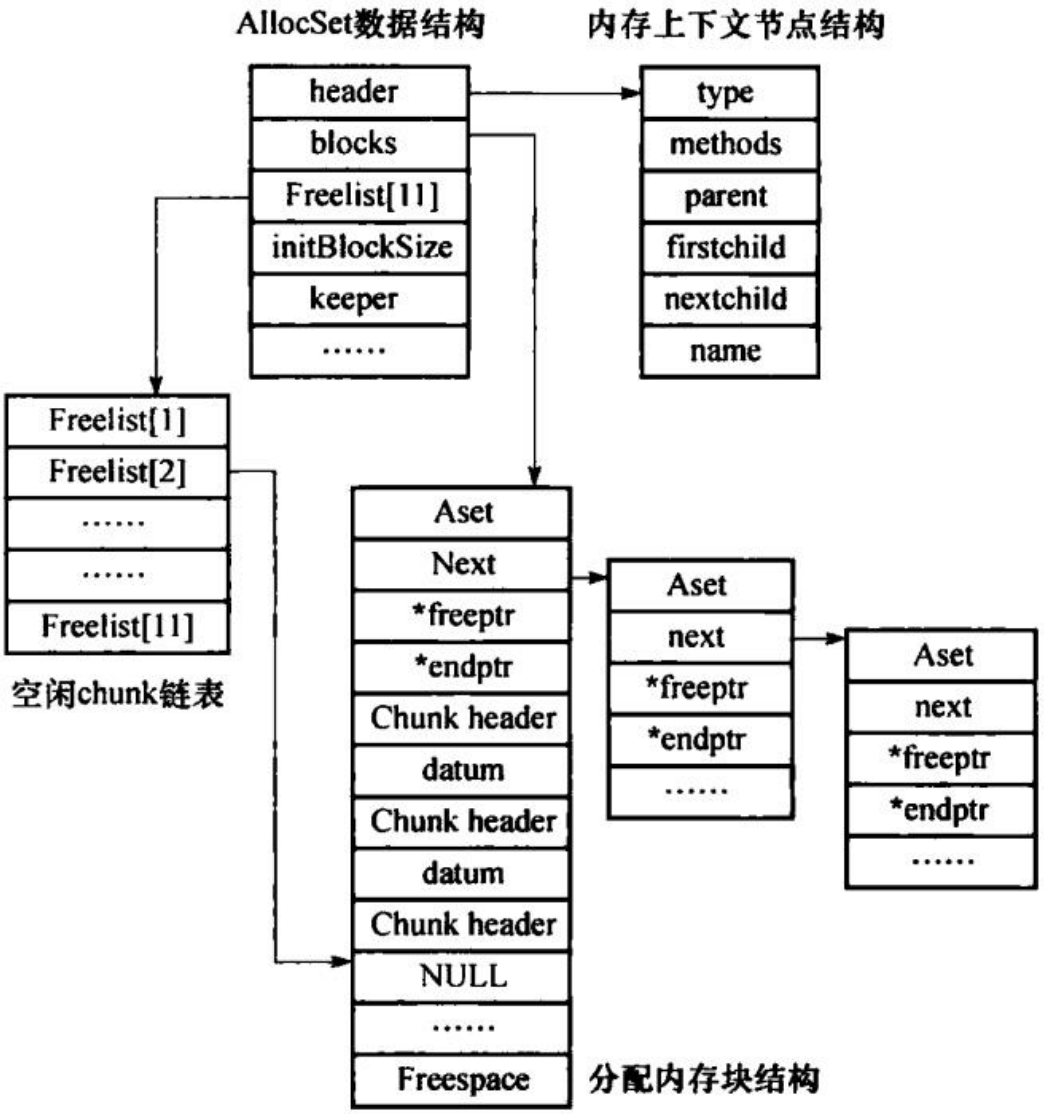
（3） 空闲内存片链表数组：

FreeList 数组是一个 AllocChunkData 类型（上面的结构体）的数组。用于维护在内存块中被回收的空闲内存片，这些空闲内存片将用于再分配。

FreeList 数组中的每一个元素都指向一个由特定大小空闲内存片组成的链表，大小和该元素在数组中的位置有关。假设元素序号为 N，则该元素所指向的链表的每个空闲内存片的大小为 2^(N + 3) 字节。这意味着指向最小的空闲内存片大小为 8B,最大的空闲内存片大小为 8KB；当一个分配空间的请求发起时：

- 若请求大小不足 8KB 则分配一个 8 KB 的内存片
- 若请求大小大于 8KB 则直接将整个内存块分配
- 其余大小均区能容纳该请求大小的最小内存片进行分配

综上，内存上下文体系的总结构如下：



可以直观地看出，内存上下文统筹了两块空间——已分配空间和空闲空间，并设置了指针对相应的结构进行监控，这种机制方便在申请或释放空间时集中管理空间，不易造成内存空间的泄露。

缓冲池

缓冲池这个概念，听起来有些抽象，实际上就是内存中划分出来的一块特定区域，在 PostgreSQL 中缓冲池的作用是利用内存访问的局部性特点，加速存取和访问。还需要明确的一点是：缓冲池空间不仅有为各个进程独占的，也有所有进程共享的（这与后续提到的 Cache 表现不同），这就表明其必需具备一定的访问控制机制。

缓冲池的关键数据结构

缓冲池的基本存储单元是一个个块（Block），也被称为“缓冲区”，每个缓冲区具有一个缓冲区描述符（BufferDesc），这个数据结构中存储了缓冲区的位置信息和状态信息：

```
typedef struct BufferDesc
{
    BufferTag   tag;            //缓冲区页面ID，指明了该缓冲区对应表块的物理信息
    int         buf_id;         //缓冲区的索引号（从0开始）
    //state这里是一个结构体，即当前缓冲区的状态，
    //包含了标志位（缓冲区是否为脏），引用数，
    //用于缓冲区替换的最近缓冲区的使用次数
    pg_atomic_uint32 state;
    int         wait_backend_pid;   //用于记录等待所有pin结束的进程PID
    //等待pin归0的目的就是要修改该缓冲区，这里可以理解为记录请求修改该缓冲区的进程号。

    int         freeNext;       //用于链接FreeList（和前面分析过的的FreeList类似）
    LWLock      content_lock;   //进程对缓冲区内容访问时加锁
} BufferDesc;

typedef struct buftag
{
    //该结构体可作为 hash 表的索引
    RelFileNode rnode;          //物理表的OID，由表所在的表空间OID、数据库OID以及表本身的OID构成
    ForkNumber  forkNum;        //枚举类型，标记缓冲区中是什么类型的文件块
    BlockNumber blockNum;       //块号
} BufferTag;
```

具体的，在内存阶段值得关注的属性有：

- tag: 存储物理信息，指示了缓冲区对应的块号
- buf_id: 可以理解为缓冲区的编号，获取缓冲区的地址时只需使用一个起始地址加上缓冲区编号乘以缓冲区大小（8KB）即可：

  ```
  (Block) (BufferBlocks + ((Size) ((buffer) - 1)) * BLCKSZ)
  ```

  关于起始地址 BufferBlocks 是如何得到的，后续与共享内存相关的描述会提及
- state: 状态标志位，其中脏位在缓冲区内容被换出时需要写回；引用次数和最近使用次数决定了缓冲区替换时该缓冲区能不能被换出，换出合不合适

缓冲池的常见函数

（1） InitBufferPool

初始化共享缓冲池，该函数的大致过程为：

- 从共享内存内初始化一系列空间
- 缓冲池若已存在则结束
- 初始化每个缓冲区对应的描述符

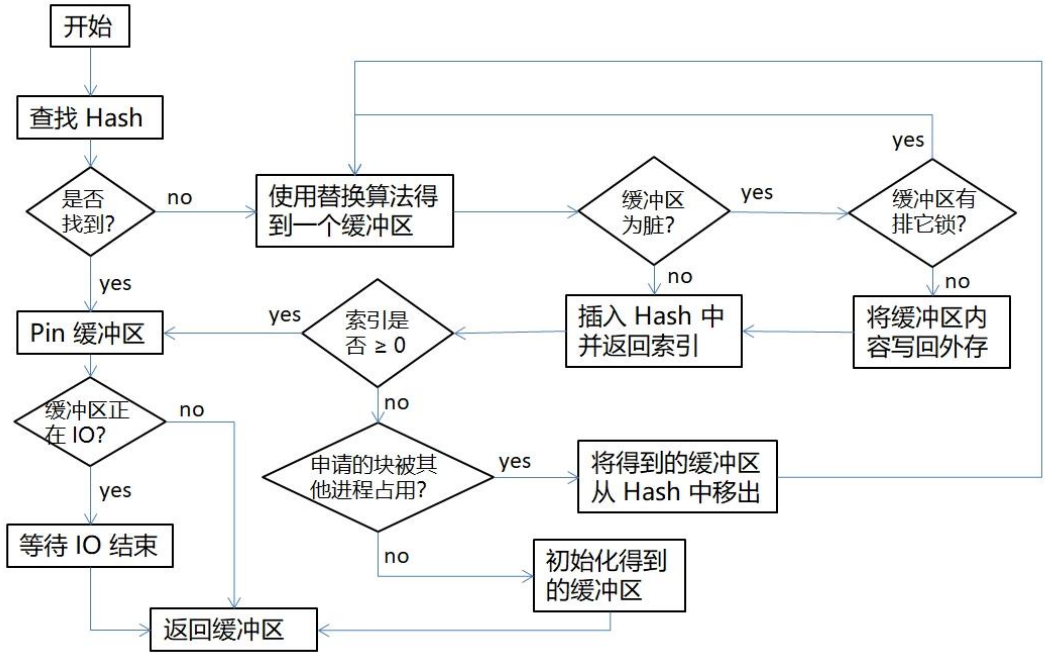
- 调用 StrategyInitialize 函数初始化其他共享缓冲区管理的条目

（2） ReadBuffer_common

读取指定缓冲区号的缓冲区，这里的读取意思是返回这个缓冲区的缓冲区描述符，其中对共享缓冲区的查找较为重要，调用核心是 BufferAlloc 函数。

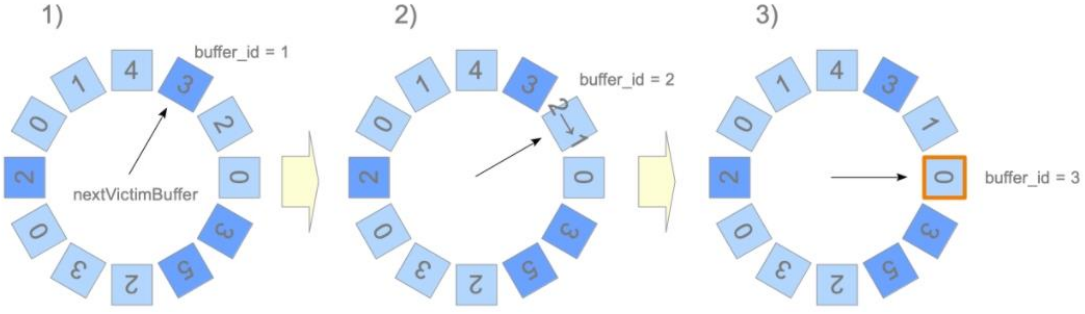
（3） BufferAlloc

该函数虽然看起来功能是为缓冲区分配一个空间，但实际上含义是：为某一部分数据申请一个缓冲区，但该函数的功能不止于此，它还是执行缓存区替换策略的核心函数；一下是该函数的大致执行流程：



缓冲池的替换策略

缓冲池的替换策略是我们在 OS 中熟悉的 Clock 算法，结合示例简要分析：



- 在 1）中，nextVictimBuffer 指向 buffer_id=1 的描述符；但是深蓝色表示该缓冲区被其他进程使用，因此跳过该缓冲区去找下一个。
- 在 2）中，nextVictimBuffer 指向 buffer_id=2 的描述符；该缓冲区没有被其他进程使用，但是上面的 usage_count（最近使用次数）为 2，因此将其减 1 后并将 nextVictimBuffer 指向下一个。
- 在 3）中，nextVictimBuffer 指向 buffer_id=3 的描述符；该缓冲区没有被 pin 住，并且 usage_count 为 0，因此该缓冲区被选中为替换对象。

注意，该算法内置一个计数器，若一直无法找到引用计数为 0 的缓冲区，其最终会递减为 0 报错，但这种情况概率极小。

页面 & 块

在缓冲池的介绍中，我们指出缓冲池的基本单位是缓冲区，也就是一个内存块，这就是我们接下来要介绍的结构：

页面（Page）和块（Block）一直是内存部分源码中共同出现最多的两个 token，在内存部分，二者在绝大多数情况下（至少是我们分析的源码范围内）是可以通用的，它们表示的内存空间大小都默认为 8 KB，它们的指针相互转换也是没有问题的。

二者最直观的不同点是页面是内存和外存都具有的结构，而块则一般只在内存语境下使用。

二者在内存范围内的不同点主要集中在不同使用情境的使用频率上，具体的：在涉及物理空间的分配和管理方面，页面结构出现较多；在涉及空间分配，函数传参等方面，块的结构出现得较多。

由于二者的相似性，下面我们仅以页面为主进行分析：

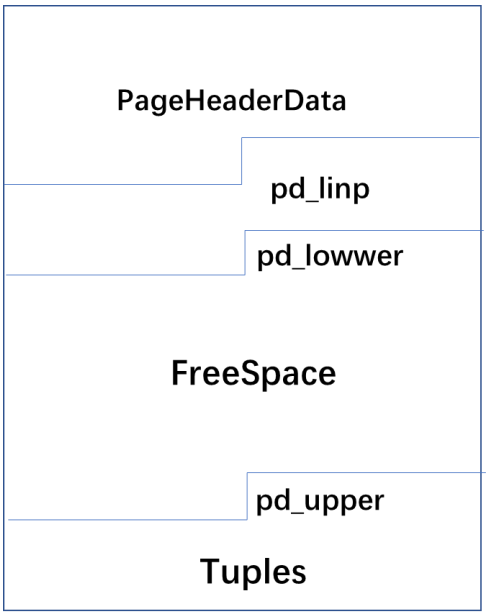
页面的数据结构和物理结构

一个页面头部的数据结构安排如下：

```c
typedef struct PageHeaderData
{
    /* XXX LSN is member of *any* block, not only page-organized ones */
    PageXLogRecPtr pd_lsn;      /* LSN: next byte after last byte of xlog
                                 * record for last change to this page */
    uint16      pd_checksum;    /* checksum */

    uint16      pd_flags;       /* flag bits, see below */

    LocationIndex pd_lower;     /* offset to start of free space */
    LocationIndex pd_upper;     /* offset to end of free space */
    LocationIndex pd_special;   /* offset to start of special space */

    uint16      pd_pagesize_version;
    TransactionId pd_prune_xid; /* oldest prunable XID, or zero if none */

    ItemIdData  pd_linp[FLEXIBLE_ARRAY_MEMBER]; /* line pointer array */

} PageHeaderData;
```

```c
typedef struct ItemIdData
{
    unsigned    lp_off:15,      /* offset to tuple (from start of page) */
                lp_flags:2,     /* state of line pointer, see below */
                lp_len:15;      /* byte length of tuple */
} ItemIdData;

typedef ItemIdData *ItemId;
#define LP_UNUSED       0       /* unused (should always have lp_len=0) */
#define LP_NORMAL       1       /* used (should always have lp_len>0) */
#define LP_REDIRECT     2       /* HOT redirect (should have lp_len=0) */
#define LP_DEAD         3       /* dead, may or may not have storage */
```

为了说明的直观，我们结合以下草图对上述结构进行分析：



PageHeaderData 占据每个页面地址最低的一部分空间，其可以看做整个页面的“中控”，其具体组件解释如下：

- pd_flags 标志页面状态
- pd_lowwer: 空闲空间的最低地址
- pd_upper: 空闲空间的最高地址
- pd_special: 特殊空间的开始地址，一般和 pd_lowwer 相等
- pd_linp: 行指针数组

关于行指针数组，先解释行指针的意义：相当于相应的元组数据在该页面位置的索引，外

界通过访问行指针来定位具体的目标数据；我们需要关注行指针结构的三个参数：

- lp_off: 对应的数据相对页面首地址的偏移量，这里和 lp_len 设置为 15 位是为了将一个 ItemIdData 结构设置为 32 位，便于地址对齐
- lp_flags: 对应数据的状态，和数据的删除、版本链控制相关
- lp_len: 对应数据的长度

行指针数组地址结束处即为 pd_lowwer。

值得注意的是 pd_linp 结构大小是不定的，则 PageHeaderData 大小也是可变的，故真正数据存储的空间从高地址开始分配，当 pd_lowwer 与 pd_upper 间的空间无法再容纳新的数据插入，则该页面对待插入数据“已满”。

页面中数据的操作

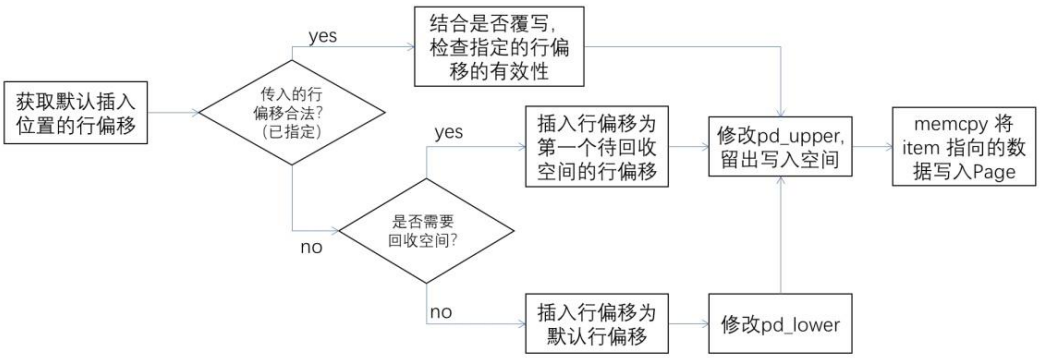
在分析具体页面操作前，我们需要明确一个比较重要的概念——行偏移（OffsetNumber），在很多对页面数据存取的函数中，往往是传入这个值，将数据写入或读出，需要注意的是，这里的偏移并不是目标数据相对页面首地址的偏移，而是目标数据对应的行指针数组的下标（从 1 开始，0 预留位 Invalid），访问一般先由行偏移定位到行指针，再由行指针索引到目标数据。

这么设计的原因可能是，外部的参数只需要关心与行指针的对接，一个页面内部的数据的控制和操作由行指针数组所在的 PageHeader 控制，这样的解耦性使系统更加易于扩展维护，也方便了数据的管理。

以下是两个常见的页面数据操作：

（1） PageAddItemExtended

这个函数多用于页面上数据的插入，函数指定要插入的页面，指定的行偏移（0 则为默认紧密模式），需要插入的数据首地址，数据长度；返回值是插入的数据对应的行偏移。此函数的执行流程大致如下：



该函数的传入的偏移合法是指：行偏移大于 0，小于等于目前最大的行偏移。

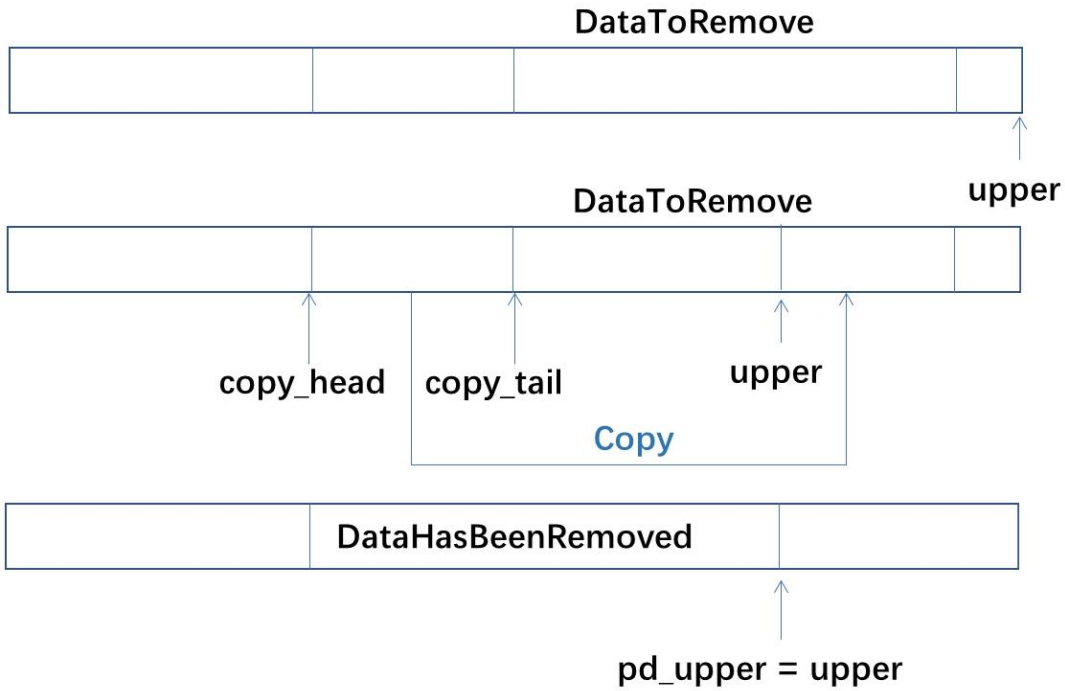
若传入的行偏移大于最大行偏移，则直接抛出错误；若等于 0，则为默认策略，在最大行偏移对应的行指针后为新插入的数据新建行指针并修改 pd_lowwer，或直接使用已无效、需要回收的行指针，将其指向新的数据地址。

该函数的“覆写”情况是指对指定的行指针进行覆写，将其更新为指向新插入的数据的地址，这里新插入数据也对应修改 pd_upper 为新数据腾出空间；可能产生的疑惑是，只是覆写了行指针，但正在的数据未被覆写，也未被删除，这如何处理？其实这是下面介绍的 compactify_tuples 函数的工作。

（2） compactify_tuples

在 PostgreSQL 的内存中，“删除”往往是指将某些标志位置为特定值的懒惰删除操作，但这样必定会导致空间内许多无效空间组成的碎片；一个有效的处理机制是 VACCUM，这在外存部分会进行解析，人为或自动触发该机制时会整合空间内的碎片和无效数据，其中在内存阶段调用的函数之一就是 compactify_tuples。

该函数的作用是合并页面中的碎片空间，“消除”已经无效的数据，这里的“消除”其实更像是将无效的数据移动到和空闲空间相接，并扩展空闲空间，这是该函数核心执行的示意图：



对于某一个已无效的连续空间 DataToRemove，通过双指针移动的方式，截取下一段连续的有效数据的长度，将这段有效数据拷贝到指定的高地址，重复这个过程，直到页面上所有的无效数据空间被转化为空闲空间即可；最后同步更新空闲空间的 pd_upper 指针，实现有效空间紧缩和碎片回收的功能。

共享内存

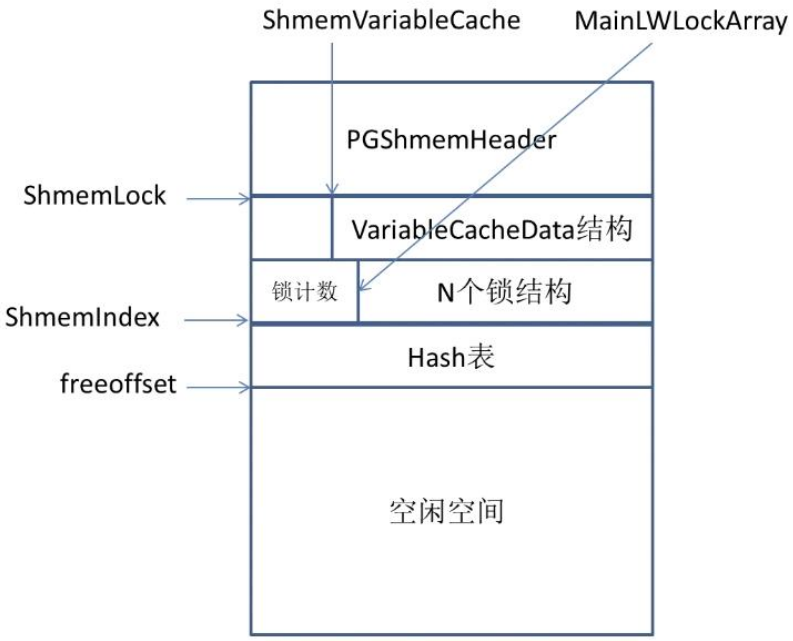
共享内存，顾名思义是所有进程都可以进行访问的一大块内存空间，上文提到的共享缓冲池就位于该空间中。

共享内存空间的初始化

共享内存在系统启动的时候被分配空间并初始化，其中主要涉及的函数如下：

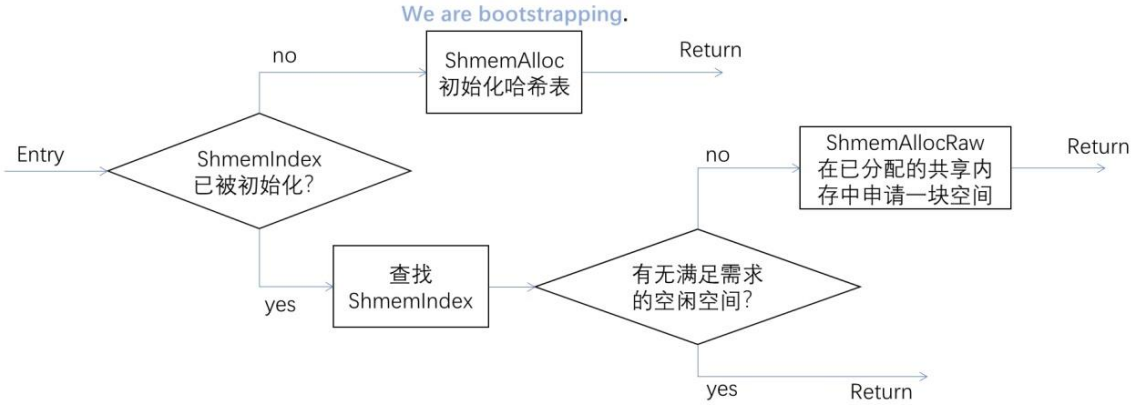
- PGSharedMemoryCreate: 申请指定大小的内存空间作为共享内存（一般比较大，上百甚至上千页的量级）
- InitShmemAccess: 使用全局变量（如下图的 ShmemLock, ShmemIndex）对共享内存的指标进行记录，使记录可以被访问到
- InitShmemAllocation: 完成共享内存 spinlock 初始化
- InitShmemIndex: 创建 hash 表，该 hash 表用来记录该共享内存空间中已申请空间的结构，实现快速检索同时避免重复申请

在空间初始化结束后，内存中的一片区域就被划分出来作为了共享内存，其结构简图如下：



共享内存空间的申请

在 PostgreSQL 系统初始化时，现为共享内存分配内存空间，再为共享缓冲池分配共享内存空间，其中后者使用的函数 ShmemInitStruct(const char *name, Size size, bool *foundPtr)，函数接受待分配区域的名字、大小，返回分配后的区域首地址，工作流程如下：



首先判断共享内存空间指示空间分配情况的 Hash 表是否已经构建了，若没有构建，证明还处在启动系统的阶段，分配一部分空间给 Hash 表并初始化；若已构建，则在 Hash 表中查找满足要求的大小的已申请的空间（但是空闲）的首地址，若已经有满足的，则直接返回，若没有满足需求的，则调用 ShmemAllocRaw 申请共享内存中的一部分空间，该函数的逻辑十分简单：返回当前 freeOffset 指向的地址，并更新 freeOffset。

上文提到的共享缓冲池的起始地址 BufferBlocks 就是这样得到的。

缓存

缓存（Cache）虽然在操作系统的术语中是独立的物理器件，但在 PostgreSQL 中其与内存（主存）使用的元件单元是相同的，Cache 更像是 PostgreSQL 中专门用于缓存系统关键数据的一种加速机制；在 PostgreSQL 中，缓存实际上由两种缓存构成，分别是 SysCache（系统表元组缓存）和 RelCache（表模式信息缓存），注意，每个进程都拥有自己的缓存，且普通表的元组是不会被放入 Cache 的（但其可以被放入缓冲池），这样设计的原因是因为系统表元组和表模式信息在数据库操作时是会被频繁使用的，所以为它们提供更快速的机制—— Cache。

这里我们主要关注的是 SysCache，其存储的系统表元组所在的各个系统表存储了数据库管理系统的元数据信息，这些元数据描述了数据库中的对象（如表、索引、约束等）。系统表中的元组（或称为行）包含了关于这些数据库对象的信息。

SysCache 的数据结构

```
static CatCache *SysCache[SysCacheSize];
```

SysCache 在代码形式上来看是一个数组，其中每个元素为 CatCache 指针，每个元素对应数据库中的一张系统表。以下是 CatCache 的具体数据结构，具体各个元素的意义可见详细注释：

```
typedef struct catcacheheader
{
    slist_head  ch_caches;      //指向catcache链表的头部，该链表串联了SysCache数组的元素，只是为了方便查找
    int         ch_ntup;        //链表中所有catcache中缓存元组的总数
} CatCacheHeader;

typedef struct catcache
{
    int         id;             //catcache的ID
    int         cc_nbuckets;    //该catcache中hash桶的个数
    TupleDesc   cc_tupdesc;     //对应的系统表元组描述符
    dlist_head *cc_bucket;      //hash桶构成的dlist链表，hash桶中实际缓存着系统表元组
    CCHashFN    cc_hashfunc[CATCACHE_MAXKEYS];  //用于每一个关键字的hash函数
    CCFastEqualFN cc_fastequal[CATCACHE_MAXKEYS];   //每个关键字的快速判断相等函数
    int         cc_keyno[CATCACHE_MAXKEYS]; //每个关键字的属性数量
    dlist_head  cc_lists;       //由catclist结构构成的dllist链表，用于缓存部分匹配的元组，每一个catclist结构保存一次部分匹配的结果元组
    int         cc_ntup;        //缓存在该catcache中的元组数量
    int         cc_nkeys;       //关键字的个数，取值范围为1~CATCACHE_MAXKEYS
    const char *cc_relname;     //catcache对应的系统表名称
    Oid         cc_reloid;      //catcache对应的系统表的OID
    Oid         cc_indexoid;    //catcache查找关键字上的索引OID
    bool        cc_relisshared; //对应的系统表是否是数据库之间共享的
    slist_node  cc_next;        //即链表的next，用于连接到SysCache中的下一个catcache
    ScanKeyData cc_skey[CATCACHE_MAXKEYS];  //为键预先填充好的ScanKey结构
-
    //运行catcache使用的统计信息
#ifdef CATCACHE_STATS
    long        cc_searches;    //总共查询该catcache的次数，每查询一次都会加一，不论是否成功。
    long        cc_hits;        //该catcache的命中次数
    long        cc_neg_hits;    //负元组的命中次数：如果一次查找在catcache和物理表中都没有找到匹配的元组，则会将这个未找到的元组作为一个负元组加入到catcache中
    long        cc_newloads;    //将catcache中没有的元组载入其中的次数
    //查询失败的次数即 cc_searches - (cc_hits + cc_neg_hits + cc_newloads)

    long        cc_invals;      //该catcache中无效元组的个数
    long        cc_lhits;       //在cc_lists中命中的次数
#endif
} CatCache;
```

其中 cc_bucket 数组中的每一个元素都表示一个 Hash 桶，元组的键值通过为其指定的 Hash 函数映射到相应的桶中，每个桶是双向链表，链表项为 dlist_node，这个属性也包含在 catctup 结构（存储单个元组）中来实现双向链表:

```
typedef struct catctup
{
    int         ct_magic;       //用于表示catctup这个结构体
#define CT_MAGIC   0x57261502   //ct_magic会被设置为这个值

    uint32      hash_value;     //该元组的Hash键值

    Datum       keys[CATCACHE_MAXKEYS];//该元组的查找关键字

    //缓存中的每个元组都是存储其哈希桶元素的 dlist 的成员。每个dlist保持LRU顺序以提升重复查找的速度。
    dlist_node  cache_elem;     //指向该元组所在的桶链表
    int         refcount;       //对该缓存元组的引用计数
    bool        dead;           //标记该元组是否死亡
    //后续搜索不会返回死亡的元组。但是，在其引用计数refcount变为零之前，它不会从CatCache中物理删除。
    //如果死亡元组同时也属于一个CatClist,则必须等到CatCList和CatCTup的refcount都变为0时才能将其从CatCache中清除。
    bool        negative;       //标记该元组是否为负元组
    HeapTupleData tuple;        //缓存元组的头部信息，实际的元组数据存放在接下来的内存中(类似于内存上下文中的内存块头
    和内存块数据)。

    struct catclist *c_list;    //指向该元组所在的catclist，如果该元组不属于任何catclist则设置为NULL

    CatCache   *my_cache;       //链接到该元组所属的catcache
} CatCTup;
```

其中属性 tuple 是 HeapTupleData 类型，这是元组数据在 PostgreSQL 内存中的表现形式，在后文会有详细分析。

其中 catlist 不同于 catctup，其存储的是一组元组，服务于部分查询，其具体数据结构如下：

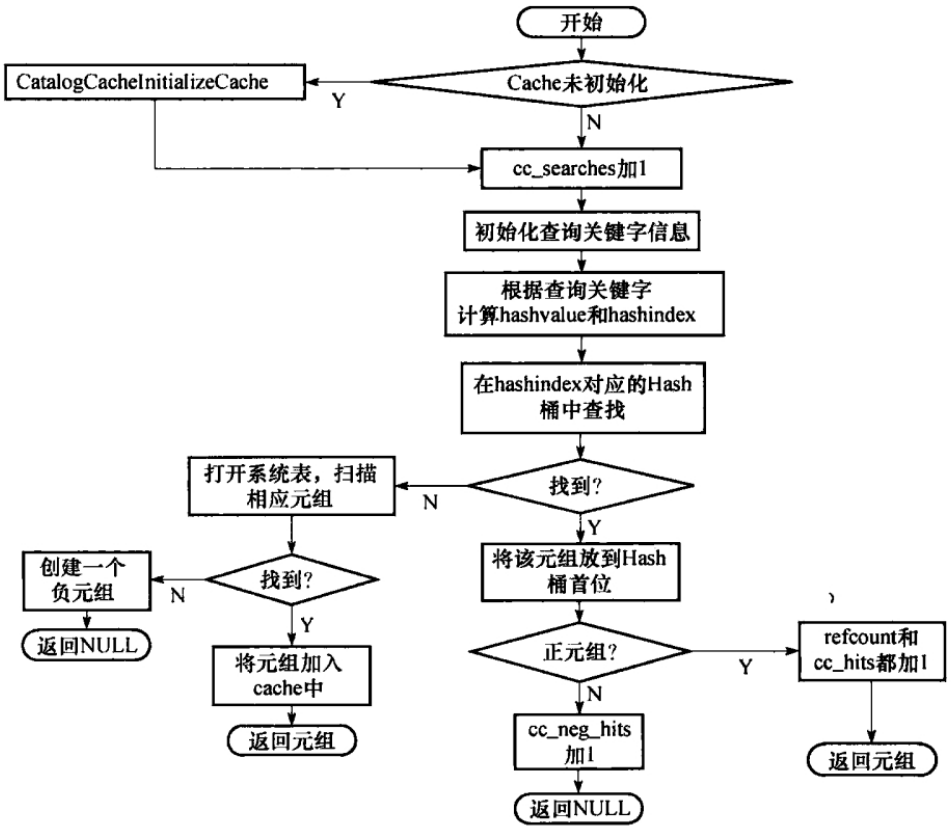
```
typedef struct catclist
{
    int         cl_magic;       //用于表示 catclist
#define CL_MAGIC   0x52765103   //cl_magic 被设置为这个值

    uint32      hash_value;     //查找关键字的 hash 值

    dlist_node  cache_elem;     //指向cc_list中的节点

    Datum       keys[CATCACHE_MAXKEYS]; ////查找关键字，只有前 nkeys 个有效

    int         refcount;       //对该元组列表的引用计数

    bool        dead;           //该元组列表是否死亡
    bool        ordered;        //包含的各元组是否按索引排序
    short       nkeys;          //匹配的键值个数
    int         n_members;      //该元组列表的元组个数
    CatCache    *my_cache;      //指向所在的 catcache
    CatCTup     *members[FLEXIBLE_ARRAY_MEMBER]; //元组列表
} CatCList;
```

Cache 中元组的查询

Cahce 中元组的查询可以分为两类：

- 精确查找：返回一个符合所给键值（组）的元组
- 部分查找：返回所有符合所给键值（组）的元组

两种查找在查找流程上区别不大，大概如下图所示：



查找根据元组的键值信息通过 Hash 函数装换位一定的值在 cc_bucket 中查找，若找到且为正元组（即表示数据确实在 Cache 中），则维护相关状态量并将其返回，若为负元组，这里我们跳出来解释负元组机制的含义：

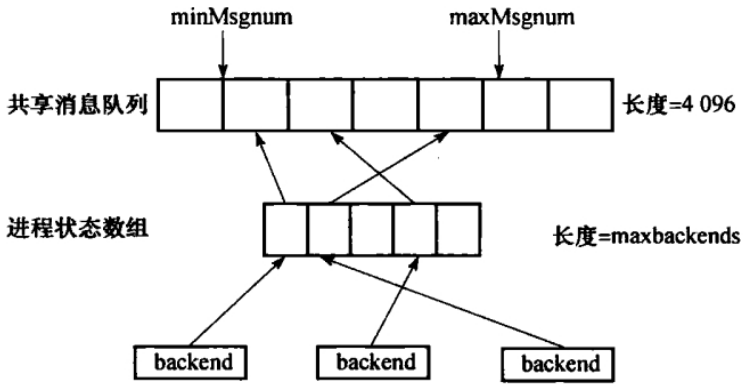
在 PostgreSQL 的 Cache 中，负元组表示一个待查找数据根本就不在数据库中，负元组一般是在一个目标元组经过扫描式查找（Cache，内存，磁盘）都没有在表内找到后由系统自动在 Cache 内产生的；当下一次在查找相同的元组时，直接命中 Cache 中对应的负元组，这样系统直接得知该元组不在数据库中，直接返回未找到，大大降低了扫描式查找的次数。

因此，在流程中，负元组的生成和查找也不难理解了。

Cache 的消息同步

我们注意到，每个进程都有自己独立的 Cache，虽然不用考虑进程间的资源冲突问题，但资源间数据的完整性、一致性应该被重点维护；具体的，一个系统表可能在不同的进程中都有对应的 Cache 来缓存它的元组；一个进程进行元组的更新和删除操作需要与其他进

程同步。在 PostgreSQL 中主要通过串行化隔离事务（SI Message）队列机制进行进程间信息的同步，这方面与事务处理关系紧密，现只作简单介绍：



当一个进程对元组数据进行了修改后，会将这次动作作为事务添加进队列；在该队列的 minMsgnum 和 maxMsgnum 之间，存储了一系列进程待办的事务，各个进程（图中的 backend）在同个时间点处理事务的进度不一样，系统会定时对各个进程处理事务的进度进行检查，并向最慢的进程发送信号，将之后的进程调度偏向该进程，达到各个进程处理事务的进度不会差距过大的目的。

SMGR

总览

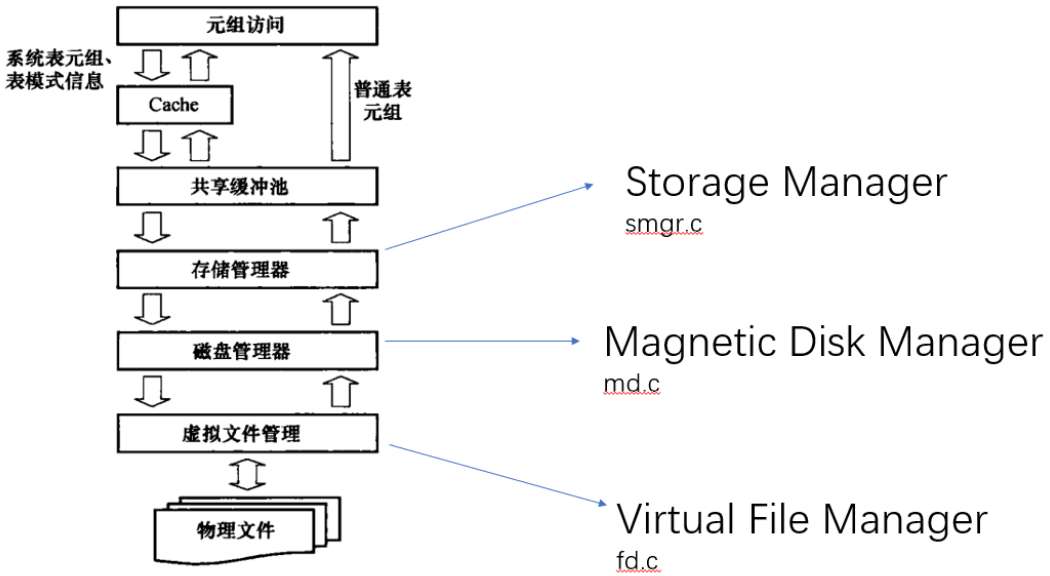


图 3-2　读写元组的过程

smgr 是 storage manager 的缩写，接收上层发送的请求，再去调用合适的存储管理器进行处理。在最初的 Berkeley Postgres 系统中，有多个存储管理器，现在只剩下“磁盘”管理

器(magnetic disk manager，md.c)，保留存储管理器(smgr.c)作为中间层，原因是：一来保证向后兼容性，使之平滑地过渡到新的存储管理技术；二来删除中间层也不会节省明显的开销，因为对存储介质的操作，比一层 c 语言函数调用昂贵的多。

具体来说：

- smgr.c :存储管理器中间层，负责分发调度，PG 中，所有对文件系统的操作都是通过这个文件中的函数进行分发。该文件中的函数调用适当的存储管理器来执行上层代码发起的存储访问请求。
- md.c：磁盘存储管理器，它实际上只是内核文件系统操作的接口。md.c 依赖于 fd.c(vfd)，调用 fd.c 相应函数进行 open，close，read，write 等操作，我们先看一下 smgr 的初始化

```
void
smgrinit(void)
{
    int         i;

    for (i = 0; i < NSmgr; i++)
    {
        if (smgrsw[i].smgr_init)
            (*(smgrsw[i].smgr_init)) ();
    }

    /* register the shutdown proc */
    on_proc_exit(smgrshutdown, 0);
}
```

```
md.c
typedef struct f_smgr
{
    void        (*smgr_init) (void);
    void        (*smgr_shutdown) (void);
    void        (*smgr_close) (SMgrRelation
reln,
ForkNumber forknum);
}
```

exit -> hook <- clean

✧ 解析：smgrsw 数组保存了一套接口函数，当前只有一套实现方法，就是磁盘控制系列函数，定义在 md.c 中，这里我们只展示一小部分

初始化函数主要包括初始化、启动、停止、关闭、创建、读、写等方法，构体内的所有函数都采用了函数指针，是为了实现各种介质的存储管理器而创建的，具备了较高的扩展性。

这里的钩子函数 ==> 在后端（postgres）进程终止时，这个函数会先 hook 住，直到 smgr 完成清理工作（终止函数）后，才继续后端的终止。

相关数据结构

```
typedef struct RelFileNode
{
        Oid     spcNode;/* tablespace */
        Oid     dbNode;/* database */
        Oid     relNode;/* relation */
} RelFileNode;

typedef enum ForkNumber
{
        InvalidForkNumber = -1,
        MAIN_FORKNUM = 0,
        FSM_FORKNUM,
        VISIBILITYMAP_FORKNUM,
        INIT_FORKNUM
} ForkNumber;
```

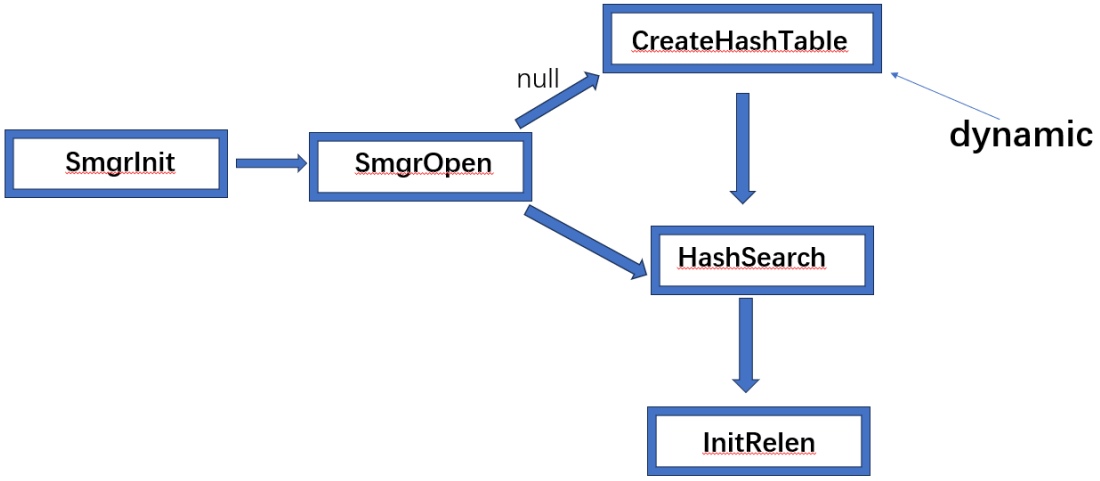
```
typedef struct SMgrRelationData
{
        RelFileNodeBackend smgr_rnode;
        struct SMgrRelationData **smgr_owner;
        BlockNumber smgr_targblock;
        BlockNumber smgr_fsm_nblocks;
        BlockNumber smgr_vm_nblocks;
        int                     smgr_which;
        int
        md_num_open_segs[MAX_FORKNUM + 1];
        struct _MdfdVec *md_seg_fds[MAX_FORKNUM
+ 1];
        dlist_node      node;
} SMgrRelationData;
```

- SMgrRelationData，为单个关系的文件结构。主要属性是 smgr_rnode，它是关系的物理标识，同时作为哈希表的 key（方便快速查找）。他的类型是 RelFileNodeBackend，就是在 RelFileNode 的基础上添加了一个 BackendId 用于标记后端的 oid。smgr_owner 是指向表头的指针，smgr_targblock 是写的块的数量，smgr_fsm_nblocks 是最近的 fsm 块的数量，smgr_vm_nblocks 是最近的 vm 块的数量。

- ForkNumber，是一个枚举类型的结构体，用于定义文件的类型。而 PG 中数据表文件有以下三种类型：main 负责存储数据、fsm 存储 main 中空闲块的大小，vm 则是方便 vacuum 回收已删除数据的空间。因此，MAIN_FORKNUM 为存储实际的数据的文件类型，FSM_FORKNUM 为存储空闲位置的文件类型，VISIBILITYMAP_FORKNUM 为存储 VM 文件类型，INIT_FORKNUM 用于数据库的初始化。

- RelFileNode 这个数据结构，它是关系文件节点，存放三个数据，分别是表空间节点（为关系所在的表空间），数据库空间节点（关系所在的数据库），关系节点（识别具体的关系）。有了这三个数据，就可以物理访问这个关系。

Open 文件过程

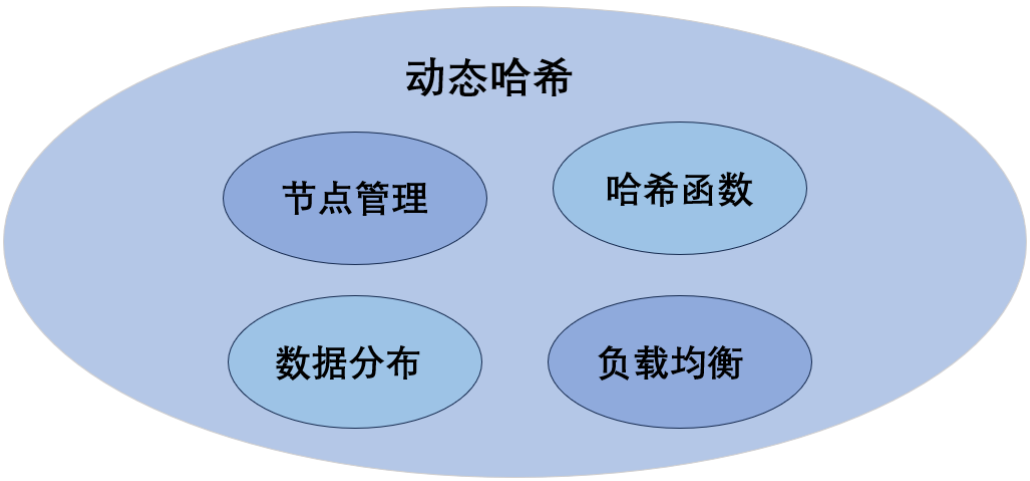


smgropen 方法非常重要，能够返回一个 SMgrRelation，也就是指向关系的指针，或者说表头，PG 中用它来表示一个被打开的表文件。

在第一次使用这个方法时，会创建一个 SMgrRelationHash，即关系哈希表，仅对后台进程可见，能够存储所有已有的表头，然后初始化没有属主的链表，节点之间没有前驱或后继的关联关系，然后在哈希表中根据参数查找对应的关系，找到则直接返回表头；未找到则将 reln 初始化，并将各种类型文件的分段数量都设为 0（意思是没有打开这个关系文件），并归入到没有属主的链表中，然后返回 reln。

然后设置属主、清理属主的方法，检测关系对应文件类型是否存在的方法，关闭文件，关闭哈希表以及关闭节点的方法。

这里的哈希表属于我们课上提到的动态哈希表，因为时间关系，老师上课没有展开讲，这里我想做一点小小的补充：PG 通过 4 个模块实现动态哈希方式：



- 节点管理模块：该模块负责管理集群中的所有节点，并维护其状态信息。当新节点加入或旧节点离线时，该模块会自动进行节点状态的更新和调整。节点就是我们说的桶。
- 哈希函数模块：该模块定义了哈希函数的计算方法。就是说我们可以选择不同的哈希函数来适应不同的应用场景或性能需求。
- 数据分布模块：该模块负责将数据分散到不同的节点上。数据的分布采用一致性哈希算法（Consistent Hashing）的方式实现。具体来说，该算法将数据的哈希值映射到一个环形空间中，每个节点在环形空间中占据一个位置。当需要访问某个数据时，根据其哈希值找到其在环形空间中的位置，然后沿着环形空间顺时针方向查找下一个节点，直到找到存储该数据的节点为止。
- 负载均衡模块：该模块负责将客户端请求分发到不同的节点上。负载均衡采用轮询（Round Robin）的方式实现。具体来说，每个节点都会依次接收到相同数量的请求，以实现负载均衡。

➢ 优势：

1. 增量扩展：动态哈希允许在运行时动态地添加或删除节点，而不会对整个哈希表进行重新哈希。这使得动态哈希在分布式系统中更加适用，因为可以方便地进行节点的扩容和缩容操作，而无需停机或重新计算哈希值。
2. 负载均衡：动态哈希能够根据节点的负载情况自动进行数据的迁移和重新分布，以实现负载均衡。当系统中某个节点的负载过高或过低时，动态哈希可以自动调整分布，使得每个节点负载相对均衡，提高系统的性能和可伸缩性。
3. 容错性：动态哈希具有较好的容错性，当系统中某个节点发生故障或离线时，动态哈希可以自动将其数据迁移到其他节点上，保证数据的可用性和一致性。

➢ 相比之下，静态哈希的主要优势在于简单性和预测性。静态哈希在构建时需要提前确定节点数量，并计算出每个节点的哈希范围。这种静态配置使得静态哈希在规模较小、节点固定的环境中更加适合，动态哈希适用于需要频繁进行扩展、缩减的情况。

mdcreate

```
if (isRedo && reln->md_num_open_segs[forkNum] > 0)
   return;              /* 检查是否早已创建和打开 */

path = relpath(reln->smgr_rnode, forkNum);//获取文件目录

fd = PathNameOpenFile(path, O_RDWR | O_CREAT | O_EXCL | PG_BINARY);//获取虚拟文件描述符
if (fd < 0)//获取描述符失败，即文件打开失败。
{
   int        save_errno = errno;
   if (isRedo)//如果是已经创建过
      fd = PathNameOpenFile(path, O_RDWR | PG_BINARY);//重新获取虚拟文件描述符
}


 Init:
 _fdvec_resize(reln, forkNum, 1);//设置此类型文件的分段数目为1，并且截断分段数组
    mdfd = &reln->md_seg_fds[forkNum][0];//设置分段数组的第一个分段
    mdfd->mdfd_vfd = fd;//添加虚拟文件描述符
    mdfd->mdfd_segno = 0;//段号设置为0
 }
```

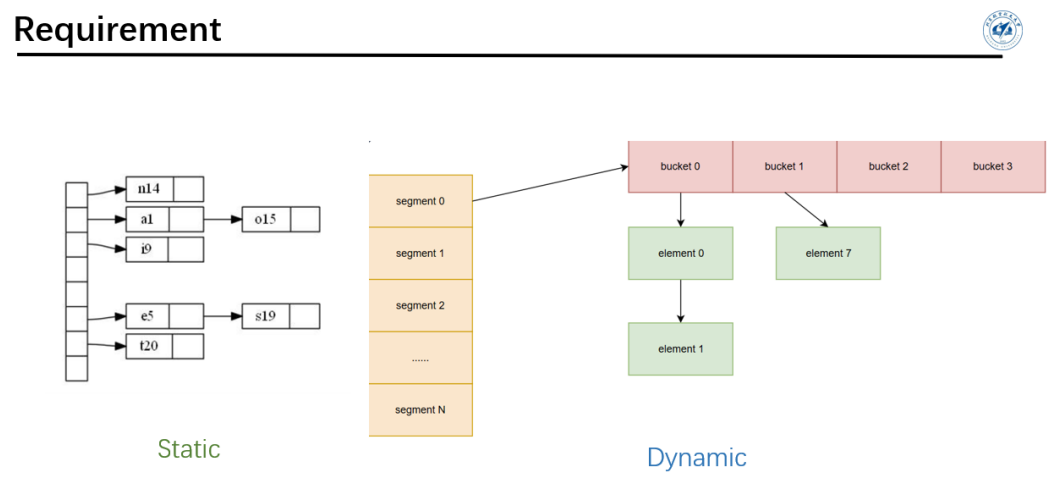
- 大致流程就是：

  检测重复->获取文件目录->获取虚拟文件描述符->初始化

- 关于分段：

  磁盘存储管理器在自己的描述符池中跟踪打开的文件描述符。这样会使得那些有文件大小限制（通常 2G 字节）的操作系统更容易实现关系表。为了做到这一点，我们将关系分成“段”文件，每个文件都比操作系统文件大小限制小。段大小由 pg_config.h 中的 RELSEG_size 配置常量设置。在磁盘上，关系表必须由连续编号的段文件组成。打开的时候也是一段一段打开的，当有多个段时，只打开第一个。

动态哈希



为什么需要动态哈希？

平常的 hash，大多是下面这样一副面孔:这种 Hash 维护着一些桶，就是图上左边的部分，每一个桶中装着 hash 值相同的数据。这些具有相同 hash 值的数据形成一个链表。这种 hash 的一个最主要缺点就是桶的数目是一定的，不易扩展，随着插入数据增多，查找效率会急剧下降。动态 hash 就是用来解决这个问题的,postgresql 实现的动态 hash 保证填充因子不超过一个预定值的情况下动态地增长 hash 表的容量。同时每一次扩容所作的改动不大，空间利用率也比较地高。

哈希结构

Postgresql 与动态 hash 相关的代码分布在 dynahash.c 和 hashfn.c 这两个文件之中，hashfn.c 主要是一些 Hash Function，而 dynahash.c 才是动态 hash 的主要实现。

与普通 hash 表相比，动态 hash 多了一个新的行政单位: 目录 dir。dir 是一个大小可变的数组，初始长度可以在创建时指定，以后每一次扩展其长度都 *2。dir 中的每一项都指向一个长度固定的 Segment, 这些 Segment 的长度都相同且必须是 2 的整数次幂，Segment 数组中的元素是 Bucket(桶) ，每一个桶中存放着一个链表，动态 hash 将所有具有相同 hash 值的元素都放在同一个桶中。

现在来看一下 pg 中 这些基本概念的定义:

Requirement

```
typedef struct HASHELEMENT
{
    struct HASHELEMENT *link;   /* link to next entry in same bucket */
    uint32      hashvalue;      /* hash function result for this entry */
} HASHELEMENT;

/* A hash bucket is a linked list of HASHELEMENTs */
typedef HASHELEMENT *HASHBUCKET;


/* A hash segment is an array of bucket headers */
typedef HASHBUCKET *HASHSEGMENT;
```

HCTL ➔ 创建可扩展哈希表时的信息表
HTAB ➔ 哈希表结构
HASHHDR ➔ 哈希表头结构,内含哈希表的所有可变信息

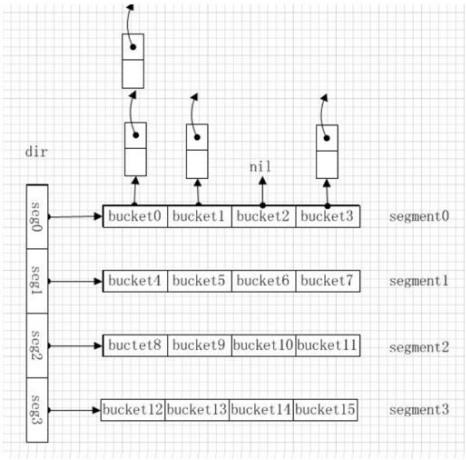
可扩展哈希表和几个数据结构有关，一个是 HCTL ，这个是创建可扩展哈希表时的信息表，一个是 HTAB ，这个就是哈希表结构，一个是 HASHHDR ，这个是哈希表头结构，包含了哈希表的所有可变信息 ，还有 HashSegment 、 HashBucket 、 HashElement 等结构，

一个 HashSegment 是 HashBucket 数组的头，一个 HashBucket 是 HashElement 链表。

搜索

给定 hash value，如何找到与其对应的 Bucket?

Search




```
/* Convert a hash value to a bucket number */
static inline uint32
calc_bucket(HASHHDR *hctl, uint32 hash_val)
{
    uint32      bucket;

    bucket = hash_val & hctl->high_mask;
    if (bucket > hctl->max_bucket)
        bucket = bucket & hctl->low_mask;

    return bucket;
}
```

- hctl->max_bucket 指的是 bucket 总数减 1，对于图 2 来说，这个值为 15
- hctl->low_mask 是<= (hctl->max_bucket + 1)的最大的 2^K 减 1， 对于图 2 来说，这个值是 16 - 1 = 15 (0000 1111)
- hctl->high_mask 是 2^(K + 1)减 1, 对于上图来说，这个值是 32 - 1 = 31 (0x0001 1111)

这几个变量要注意的是，hctl->max_bucket 在 hash 表创建好以后，会变化，一般情况下每次增加一个，就需要更新 hctl->low_mask 和 hctl->high_mask，后面我们会看到具体如何进行扩容

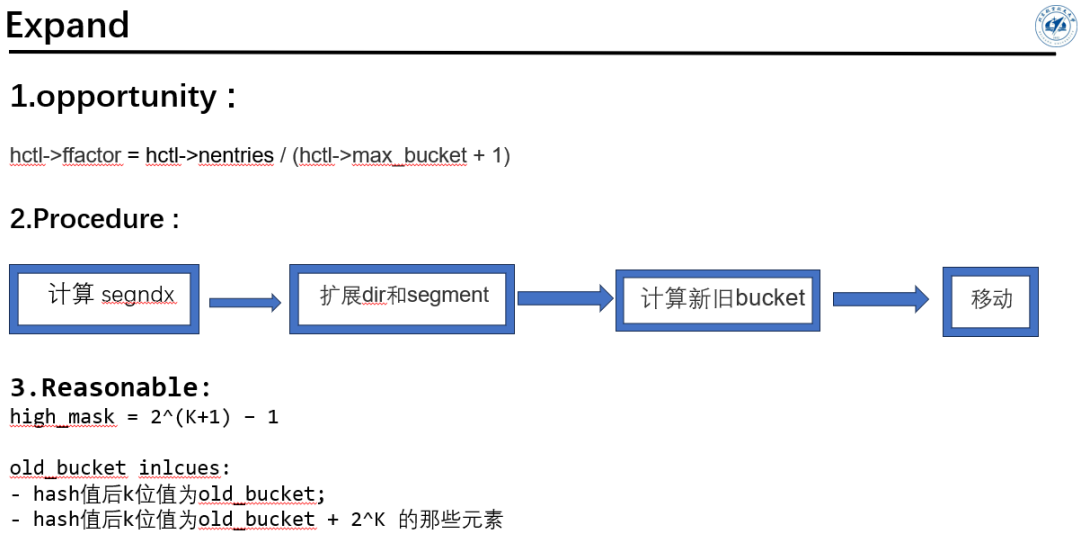
我们回头继续看那个 calc_bucket 函数, 因为理解这个函数是理解动态扩展的关键。从上面关于 max_bucket,low_mask 和 high_mask 的介绍，可以得出下面的结论:

hctl->high_mask >= hctl->max_bucket >= hctl->low_mask

反应在它们的二进制表示上，如果 hctl->low_mask 占 m 位(2^m - 1)，则 hctl->high_mask 占 m + 1 位。而 hctl->max_bucket 是在闭区间[low_mask, high_mask]之间的。所以要取得 hash_val 的 bucket 值应优先&上 high_mask,如果发现得到的 bucket 的序号比 max_bucket 还大，则再 &上 low_mask。这一步为后面的扩展埋下了伏笔。

动态扩展

在弄清楚动态 hash 的结构以及如何从 hash 值得到其所在的 bucket 后，hash 表的查找，删除以及通常情况下的插入操作就非常地容易啦。比如删除，就先是调用 cal_bucket 找到所在的桶，然后在这个桶中一个个地找，找到具有相同键值的元素，就从桶所对应的链表中删除。下面来说一下动态扩展的问题，毕竟这才是 pg 动态 hash 的核心。



扩展时机

说到动态扩展，就得说扩展时机，什么时候我们的 hash 表需要增大容量呢？postgresql 只会在负载率过高的情况下，发生扩容操作。并且它的一次扩容只会增加一个 bucket。 我们增大容量是为了保证其查找的效率。而 hash 表的查找效率是与一个叫做填充因子东西有很大关系的。

pg 的填充因子定义为:

hctl->ffactor = hctl->nentries / (hctl->max_bucket + 1)

在 PostgreSQL 中，nentries 是一个用于动态哈希表实现的计数器，用于记录哈希表中当前存储的元素数量。

nentries 记录了哈希表中当前的元素个数，而 max_bucket 则表示哈希表中最大桶的索引。通过这两个参数，可以计算出哈希表的填充因子（fill factor），即哈希表中元素占用的比例。

在给哈希表分配内存或者调整哈希表大小时，填充因子是一个重要的参考指标。填充因子过高可能导致哈希冲突率增加，从而影响哈希表的性能，此时我们需要扩展；填充因子过低则可能导致哈希表空间浪费，从而影响系统的内存使用效率，此时我们可能需要缩

小哈希表大小以节省内存空间。

从这个公式可以看出填充因子会随着插入元素的增多而增加， 当增大到比 hctl->ffactor 还要大的时候就需要扩展了，这里的扩展的意思是指 hctl->max_bucket 要增加 1 个。

扩展方式

当发生扩容时，会计算新增 bucket 的元素和原先哪个 bucket 有关系，需要将旧有的 bucket 数据重新分开。并且还会更新相关属性，也就是 low_mask 和 high_mask

具体的计算过程如下：

- 计算出 max_bucket + 1 所在的 segment 索引值 segndx
- <<扩展 dir 和 segment>>
    - 如果 segndx 没有超出已分配的 segment 的容量，是则计算新旧 bucket,否则继续;
    - 检查 segndx 是否超出了现有的 dir 大小，如是，将 dir 的大小扩展为以前的 2 倍；
    - 给 dir[segndx]分配一个新的 segment;
- <<计算新旧 bucket>>
    - 计算 max_bucket + 1 在没有扩展前的 bucket 号 old_bucket;
    - max_bucket++后检查 max_bucket 是否成了 2 的整数次幂，若是，修改 low_mask 和 high_mask;
    - 计算新的 bucket 号 new_bucket;
- <<将 old_bucket 中满足条件的元素移动 new_bucket 中>>
    - 遍历 old_bucket 链表，重新计算他们所应该在的 bucket,将需要移动的移到到 new_bucket 中。

这个过程中最后一步需要说明一下，为什么只是本次拆解扩展前这一个的 old_bucket，其他 bucket 不变。这是由 calc_bucket 的实现来决定的。假定 high_mask = 2^K - 1, 这个函数告诉我们，old_bucket 中存放着两类元素:

1. hash 值后 k 位值为 old_bucket;
2. hash 值后 k 位值为 old_bucket + 2^(k - 1)的那些元素

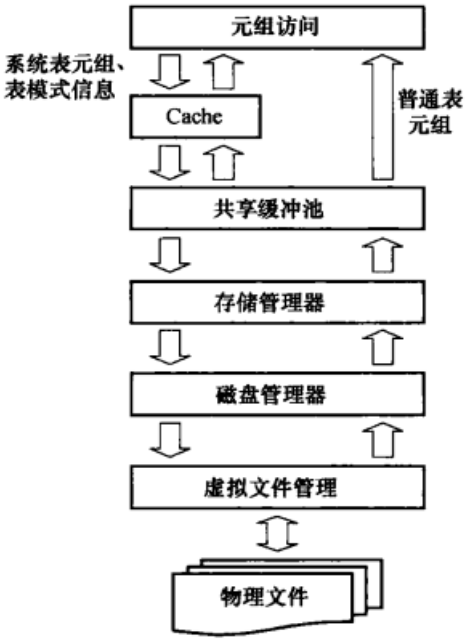
在扩展后，hash 表中只有第 2)元素的桶号应该变为 new_bucket。所以只要移动一个 old_bucket 中第 2)类到 new_bucket 中就可以啦。

哈希总结

postgresql 在设计扩容时是很精妙的，只是使用了 high_mask 和 low_mask 两个数字进行 & 操作，就可以快速的找到对应的 bucket。同时，pg 实现的这个动态 hash 保证填充因子不超过设定值，其检索效率不会因插入元素的增多而降低，同时其扩展的代价也不是十分地大，每次只需要移动一个 bucket 中的某些项到新的 bucket 中

（三）、外存

1.外存框架



Cache miss，查缓冲区，可以归属于内存管理的任务。

缓冲区 miss，调磁盘，可以归属于外存管理的任务。

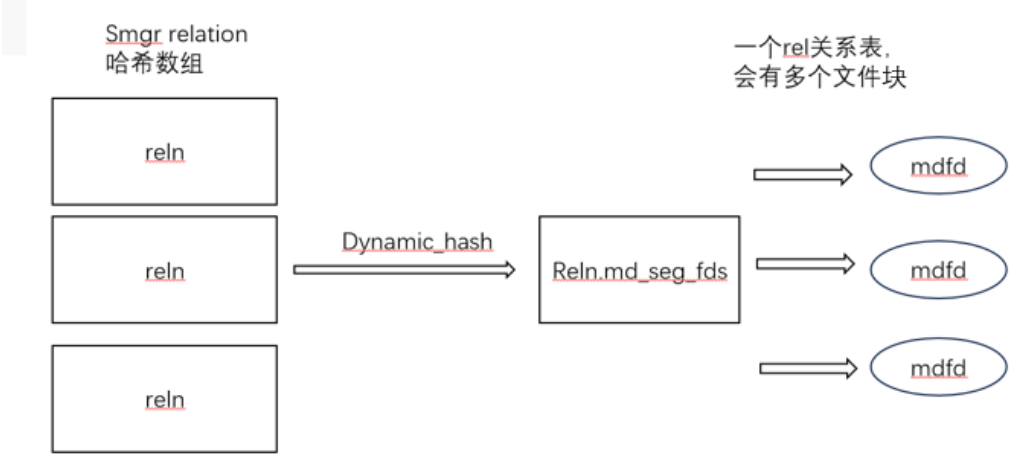
1.1 数据结构中的框架层次关系

这一部分主要抓住三个映射关系：

1）smgrRelation 到 mdfd 的映射（从存储管理器到磁盘管理器）：

```
typedef struct RelFileNode
{
        Oid     spcNode;/* tablespace */
        Oid     dbNode;/* database */
        Oid     relNode;/* relation */
} RelFileNode;

typedef enum ForkNumber
{
        InvalidForkNumber = -1,
        MAIN_FORKNUM = 0,
        FSM_FORKNUM,
        VISIBILITYMAP_FORKNUM,
        INIT_FORKNUM
} ForkNumber;
```

```
typedef struct SMgrRelationData
{
        RelFileNodeBackend smgr_rnode;
        struct SMgrRelationData **smgr_owner;
        BlockNumber smgr_targblock;
        BlockNumber smgr_fsm_nblocks;
        BlockNumber smgr_vm_nblocks;
        int                     smgr_which;
        int
        md_num_open_segs[MAX_FORKNUM + 1];
        struct _MdfdVec *md_seg_fds[MAX_FORKNUM
+ 1];
        dlist_node      node;
} SMgrRelationData;
```



在内存部分，会维护记录每个打开的表文件，这些表文件都存储在 smgrRelation 结构体形成的动态哈希数组中，其中每个 smgrRelation 结构体类型的 reln 代表一个从外存打开的表文件。一个表文件可能会由不止一个文件块组成，因此对于物理存储介质是磁盘的情况，reln.md_seg_fds 数组中存储着每个文件块对应的磁盘描述符的编号 mdfd。

2）mdfd 到 vfd 的映射（从磁盘管理器到虚拟文件管理）：

磁盘管理器

```
typedef struct _MdfdVec{
    File mdfd_vfd; // mdfd -- vfd
    BlockNumber mdfd_segno; // segment number, from 0
    struct _MdfdVec *mdfd_chain;
}MdfdVec;
```

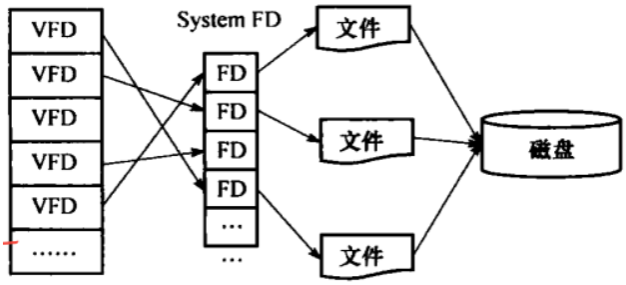
通过_MdfdVec 数据结构来存储映射关系

2） vfd 到 fd 映射（从虚拟文件管理到真实操作系统的文件系统）：

VFD结构

```
typedef struct vfd
{
    int         fd;             // vfd -- 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       fileSize;       /* current size of file (0 if not temporary) */
    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 */
    mode_t      fileMode;       /* mode to pass to open(2) */
} Vfd;
```

通过 vfd 数据结构来存储映射关系



具体是如何建立映射关系，函数调用流程是什么，可见（“示例 SQL 语句的执行过程展示”部分）

1.2 函数执行中的框架层次关系

下面我们以 create 表文件为例，在函数执行过程中概览外存管理的框架层次关系（已移动到“示例 SQL 语句的执行过程展示”部分）

1.3 postgreSQL DBMS 外存管理与操作系统之间关系的讨论

通过 1.2，我们对两者之间的关系，已经有了大体的认识。与此类似地，我们再举一举非常核心的外存操作 read/write 的例子：

Read

```
void
mdread(SMgrRelation reln, ForkNumber forknum, BlockNumber blocknum,
       char *buffer)
{
    ...

    nbytes = FileRead(v->mdfd_vfd, buffer, BLCKSZ, seekpos,
                      WAIT_EVENT_DATA_FILE_READ);

    ...
}
```

```
int
FileRead(File file, char *buffer, int amount, off_t offset,
         uint32 wait_event_info)
{
    int         returnCode;
    Vfd        *vfdP;
...
retry:
    pgstat_report_wait_start(wait_event_info);
    returnCode = pg_pread(vfdP->fd, buffer, amount, offset);
    pgstat_report_wait_end();

...
}
```

```
//fileapi.h

WINBASEAPI
_Must_inspect_result_
BOOL
WINAPI
ReadFile(
    _In_ HANDLE hFile,
    _Out_writes_bytes_to_opt_(nNumberOfBytesToRead, *lpNumberOfBytesRead)
    __out_data_source(FILE) LPVOID lpBuffer,
    _In_ DWORD nNumberOfBytesToRead,
    _Out_opt_ LPDWORD lpNumberOfBytesRead,
    _Inout_opt_ LPOVERLAPPED lpOverlapped
    );
```

函数调用关系：

smgrread--函数指针指向-->mdread->FileRead->pg_pread->ReadFile

Write

```
void
mdwrite(SMgrRelation reln, ForkNumber forknum, BlockNumber blocknum,
        char *buffer, bool skipFsync)
{
...

    nbytes = FileWrite(v->mdfd_vfd, buffer, BLCKSZ, seekpos,
                       WAIT_EVENT_DATA_FILE_WRITE);
...
}
```

```
int
FileWrite(File file, char *buffer, int amount, off_t offset,
          uint32 wait_event_info)
{
    int         returnCode;
    Vfd        *vfdP;
...

retry:
    errno = 0;
    pgstat_report_wait_start(wait_event_info);
    returnCode = pg_pwrite(VfdCache[file].fd, buffer, amount, offset);
    pgstat_report_wait_end();
...
}
```

```
//fileapi.h

WINBASEAPI
BOOL
WINAPI
WriteFile(
    _In_ HANDLE hFile,
    _In_reads_bytes_opt_(nNumberOfBytesToWrite) LPCVOID lpBuffer,
    _In_ DWORD nNumberOfBytesToWrite,
    _Out_opt_ LPDWORD lpNumberOfBytesWritten,
    _Inout_opt_ LPOVERLAPPED lpOverlapped
    );
```

函数调用关系(从磁盘管理器层次开始)：

smgrwrite--函数指针指向-->mdwrite->FileWrite->pg_pwrite->WriteFile

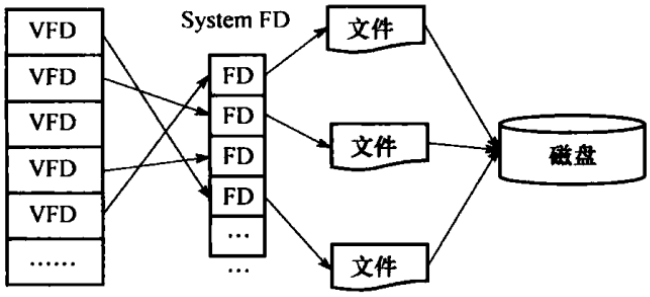
这些例子都清楚的让我们看到， postgreSQL DBMS 的外存管理追根溯源下去，并没有非常底层地直接操作磁盘介质， 最终都通过调用 WINAPI 操作系统接口的方式实现操作。但 postgreSQL DBMS 对操作系统的文件系统又进行了层层封装，以适应 DBMS 本身管理数据，频繁与数据交互的特性，完备本身文件系统的功能，同时可以让 DBMS 很好的支持跨平台运行，不依赖于底层系统实现。

2.VFD 机制

2.1 功能概述：

操作系统对于单个进程中能打开的文件数是有限制的，而数据库需要频繁的操作大量的文件。为了满足数据库的需求又不违背操作系统的约束限制，postgreSQL 实现了虚拟文件描述符 VFD（Virtual File Description）机制。本质上是在 postgreSQL 软件内，对操作系统的文件系统进行了又一层封装。这样所有需要操作文件的地方可以调用 VFD，VFD 分配不受操作系统数量限制，同时也隐藏了对操作系统管理的细节。

虚拟文件描述符 VFD、真实文件描述符 FD、磁盘文件之间的关系如下图所示。概括起来：操作系统中，真实文件描述符 FD 机制实现了对磁盘文件的封装；postgreSQL 软件中，建立虚拟文件描述符 VFD 和真实文件描述符 FD 的映射，实现了对操作系统文件系统的又一次封装。



2.2 实现方式

2.2.1 数据结构组织

VFD 这层封装主要是采用了 vfd 结构体的两个链表：free 链表和 LRU 链表来实现。

```
typedef struct vfd
{
    int         fd;             // vfd -- fd   /*current FD, or VFD_CLOSED if none */
    unsigned short fdstate;     /* 脏位，写回时判断标记 */
    ResourceOwner resowner;     /* owner, for automatic cleanup */
    File        nextFree;       /* 指向下一个空闲的 虚拟文件描述符 */
    File        lruMoreRecently;    /* 指向比该 VFD 最近更常用的虚拟文件描述符 */
    File        lruLessRecently;        /* 指向比该 VFD 最近更不常用的虚拟文件描述符  */
    off_t       fileSize;       /* current size of file (0 if not temporary) */
    char       *fileName;       /* 表示该 VFD 对应文件的文件名，如果是空闲的 VFD 则 fileName 为空值 */
    /* NB: fileName is malloc'd, and must be free'd when closing the VFD */
    int         fileFlags;      /* 表示该文件打开时标志，包括只读、只写、读写等 */
    mode_t      fileMode;       /* 表示文件创建时所指定模式 */
} Vfd;
```

Vfd 结构体中核心关注点是，fd 字段，为每个 vfd 建立了 vfd—fd 映射。如果当前 VFD 没有打开文件描述符（即没有对应的物理文件描述符），则其值为 VFD_CLOSED（为 -1）。

1）首先，VfdCache 连续数组记存储了所有的 Vfd 结构体，VFD 在 VFD 数组中的下标

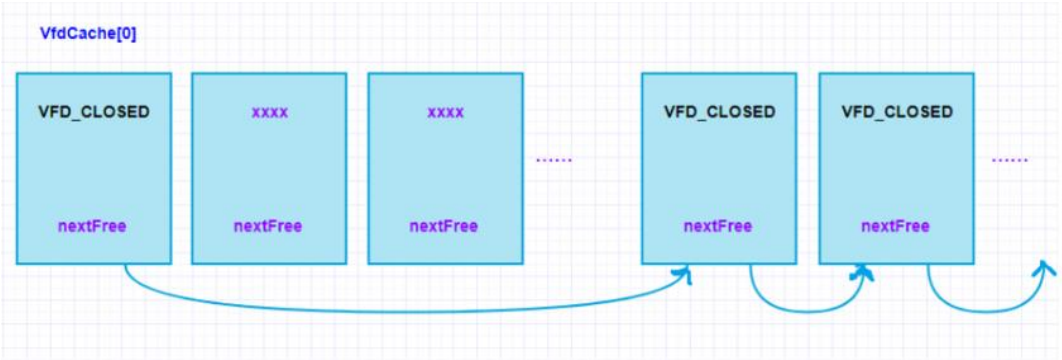
即为虚拟文件描述符 VFD 的数值。vfd 结构体为虚拟文件管理模块中存储的数据类型；而在磁盘管理器等顶层模块使用时，虚拟文件描述符通过 file 数据类型来代指一个文件。值得注意的是，file 数据类型为 postgreSQL 自定义的数据类型，实际上就是 int。不要将其与 C 语言中的 FILE 类型，以及我们操作系统课程中的 file 结构体类型混淆。

```
//postfreSQL  fd.h
typedef int File;    //虚拟文件描述符
```

```
// 摘自操作系统课程源码
struct File {
 char f_name[MAXNAMELEN]; // filename
 uint32_t f_size; // file size in bytes
 uint32_t f_type; // file type
 uint32_t f_direct[NDIRECT];
 uint32_t f_indirect;

 struct File *f_dir; // the pointer to the dir where this file is in, valid only in memory.
 char f_pad[BY2FILE - MAXNAMELEN - (3 + NDIRECT) * 4 - sizeof(void *)];
 } __attribute__((aligned(4), packed));
```

因此，概括起来，在我们的 postgreSQL 源码中，虚拟文件描述符 VFD 和真实文件描述符 FD，都是通过一个 int 数字标号来代指一个文件。（fd 的 int 数字标号，通过调用 C 语言文件相关库函数获取）

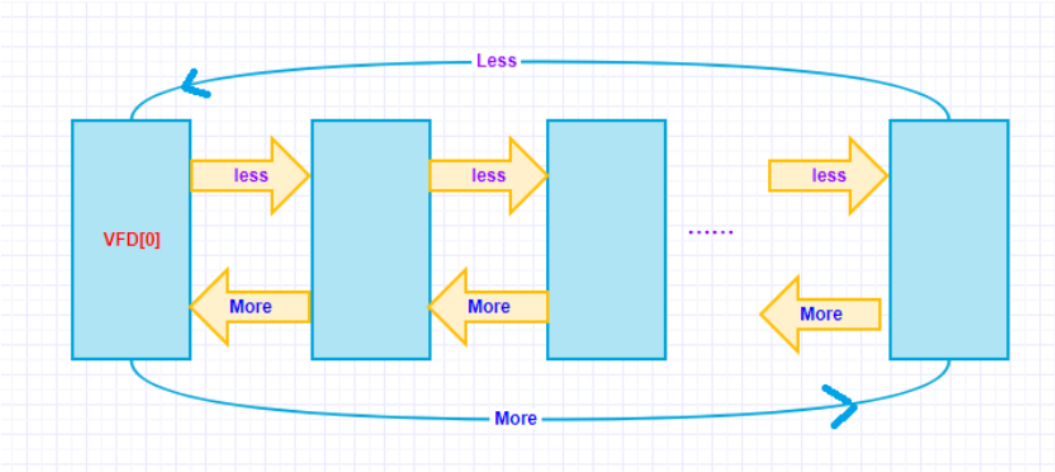
2）然后，关注 vfd 结构体的的 nextFree 字段。nextFree 字段指向下一个空闲的 VFD。形成了一个 free 单向链表。在 VfdCache 数组中，由头结点 VfdCache[0]开始，通过访问 nextFree 指针，可以遍历目前所有空闲节点



3）最后，再来关注 vfd 结构体的的 lruMoreRecently 和 lruLessRecently 字段。分别指向比该 VFD 最近更不常用的 VFD（这里的 More 做打开时长更长（即更旧）来理解）；和指向比该 VFD 最近更常用的 VFD（这里的 Less 做打开时长更短（即更新）来理解）。形成了一个 LRU 双向循环链表。在 VFD 节点中，lruMoreRecently 和 lruLessRecently 两个字段，作为双链表的指针，将所有打开的文件链接两个闭合的环，形成一个 LRU 池。

当 LRU 池未满时，进程可以直接申请一个 VFD 用来打开一个物理文件；而当 LRU 池

已满后，进程需要先从池中删除一个 VFD 并关闭其物理文件，这样打开新的文件时就不会因为超出操作系统的限制而打开文件失败。



2.2.2 维护 vfd 链表的核心函数举例：（位于 fd.c 文件中）

InitFileAccess():当后台进程启动时,会调用该函数，初始化 VfdCache 数组，创建 VfdCache 头。

AllocateVfd()：函数从空闲 vfd 链表中分配 vfd 节点，并将节点索引返回。当 Freelist 链表上没有空闲节点时，就会新申请一块更大的内存（在原数组的基础上扩大一倍），并把原有的 VfdCache 数组搬迁过来。

Insert()：将索引 file 的 Vfd 结构体插入环形队列的 front

LruInsert()：将索引 file 的 Vfd 结构体插入环形队列

Delete()：将索引 file 的 Vfd 结构体从环形队列中删除

ReleaseLruFile()：删除 lru 池中最近最少使用的 vfd，使用 LruDelete 函数处理，不仅从 Lru 中剔除还将打开文件的当前指针位置保存到 vfd，并且关闭文件，设置 vfd 的状态为无效。

3. FSM 文件与空闲空间管理

3.1 概述

随着数据表中不断插入和删除元组，页内必然会产生空闲空间。当我们需要插入新的元组时，需要优先将元组放到已有页内的空闲空间内，以节约存储空间。为了避免反复遍

历页寻找空闲空间，PostgreSQL 使用了空闲空间映射表（FSM）来记录所有表文件的空闲空间信息。

在 PostgreSQL 中，对于每一个表文件（包括系统表）都会同时存在一个名为 OID_fsm 的空闲空间映射表，其中 OID 是对应表的 OID。

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：

映射值	数据块大小
1	0-31
2	32-63
...	...
255	8146-8192

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。

3.2 详细代码解析（这部分尚未深入探索，留给学弟学妹）

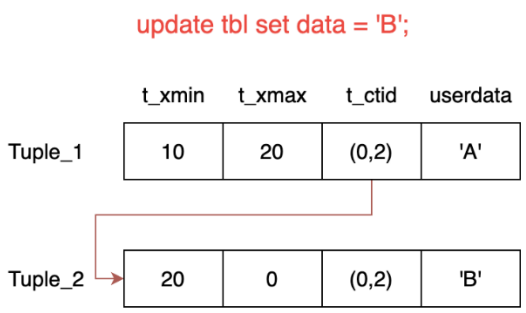
4. VM 文件与 Vacuum 机制

4.1 VM 文件与 Vacuum 机制

4.1.1 背景

1）死元组和表空间膨胀

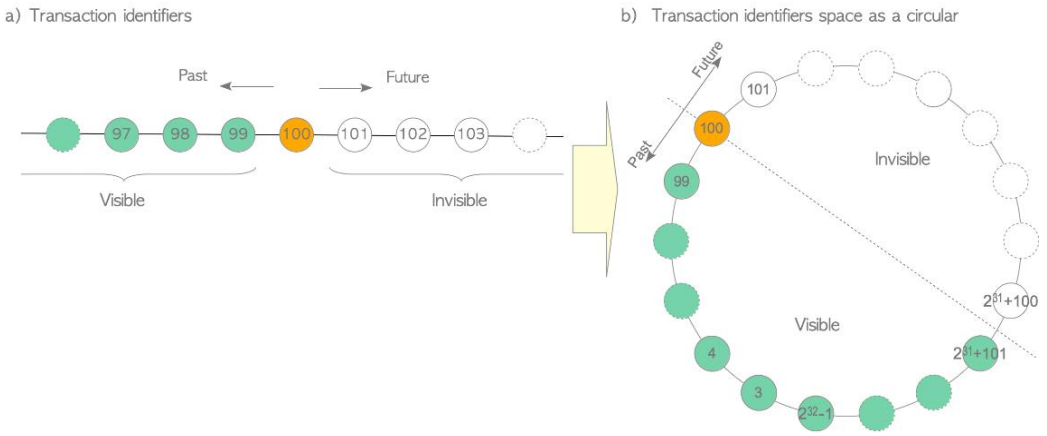
在 PG 中，update/delete 语句的实现通过 MVCC 机制的多版本链实现。如下图所示，更新一条元组时，会将原来的元组标记，并新增一条元组。后续的事物通过快照来判断元组的可见性。



对于一条已经被更新/删除的元组来说，当这条元组对所有事物都不可见后，它的存在就没有意义了，理应被删除，对于这种元组，我们称之为“死元组”。当一张表有大量更新/删除时，如果不做清理的话，表里面就会积攒很多这样的“死元组”，占用大量的空间，造成表空间膨胀。

2）事物号回卷

PG 的事物号采用 32 位无符号整数 xid 来表示，约能表示 40 亿数字。借用下图来说，对于任意一个 xid ，它在环中的前 20 亿对它来说是过去，后 20 亿对它来说是未来。



以上图为例，对于 xid 为 100 的事物来说，本来 99 是它的过去，当事物号耗尽的时候，99 会回卷重新使用。这时候就无法判断一条 xmin 或 xmax 为 99 的元组到底是过去还是未来。

4.1.2 Vacuum 机制

1）Vacuum 的功能：

①清理死元组：删除死元组和其索引，回收表**空间**；

②冻结元组的**事务**号：如果一个元组的事物号已经很老了，对于当前所有的事务都可见，那么就直接将它的事物号改为 FrozenTransactionId （注：FrozenTransactionId 对于所有事物号都属于过去）

③更新表的统计信息，方便**优化**器工作。

可以看到，vacuum 在 pg 中用处不小，可以同时服务于存储管理、事务管理与优化。

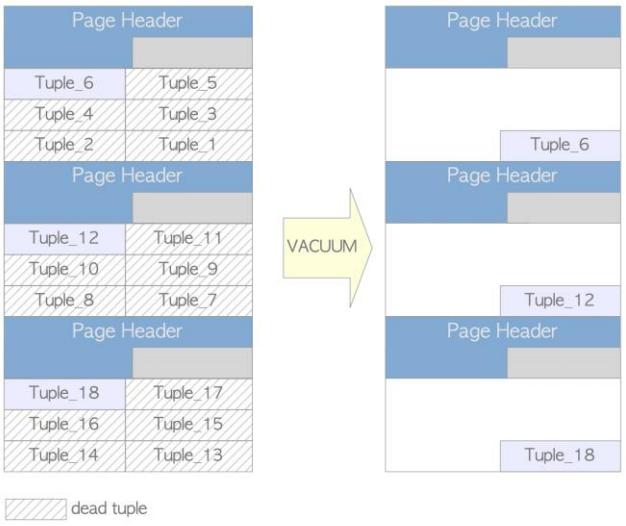
2）两种 Vacuum：

VACUUM 有两种：***Lazy VACUUM*** 和 ***Full VACUUM***

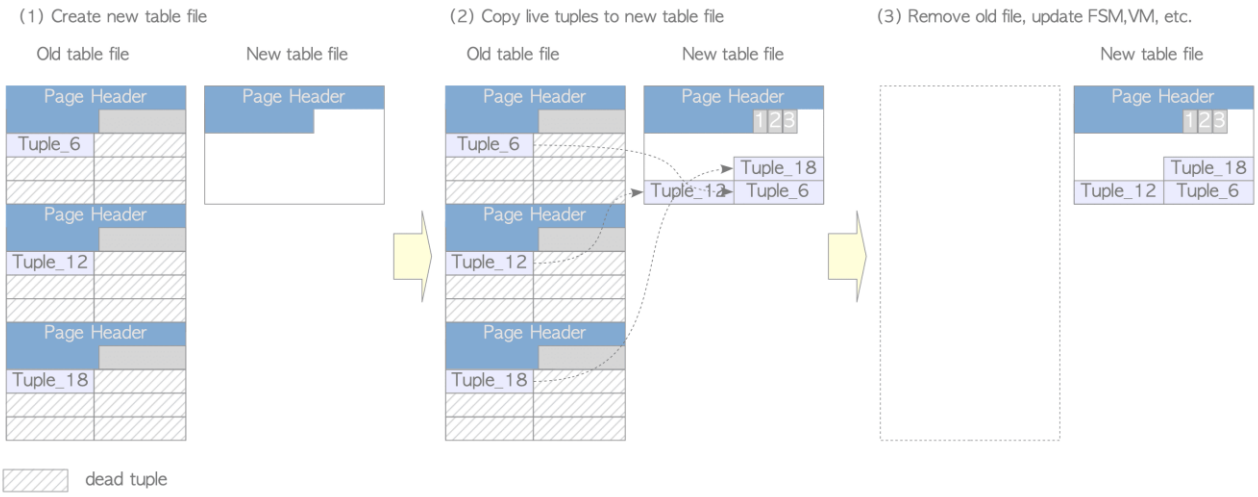
Lazy VACUUM 基本可以并行其他操作（DML 运行正常，但不能执行 ALTER TABLE），但能回收的磁盘空间很少。

Full VACUUM 能回收更多的磁盘空间，但运行速度要慢得多；它需要对表加独占锁，因此不能与该表的其他操作并发进行；此外还需要额外空间存储表副本。

Lazy VACUUM 清理效果图例



Full VACUUM 清理效果图例

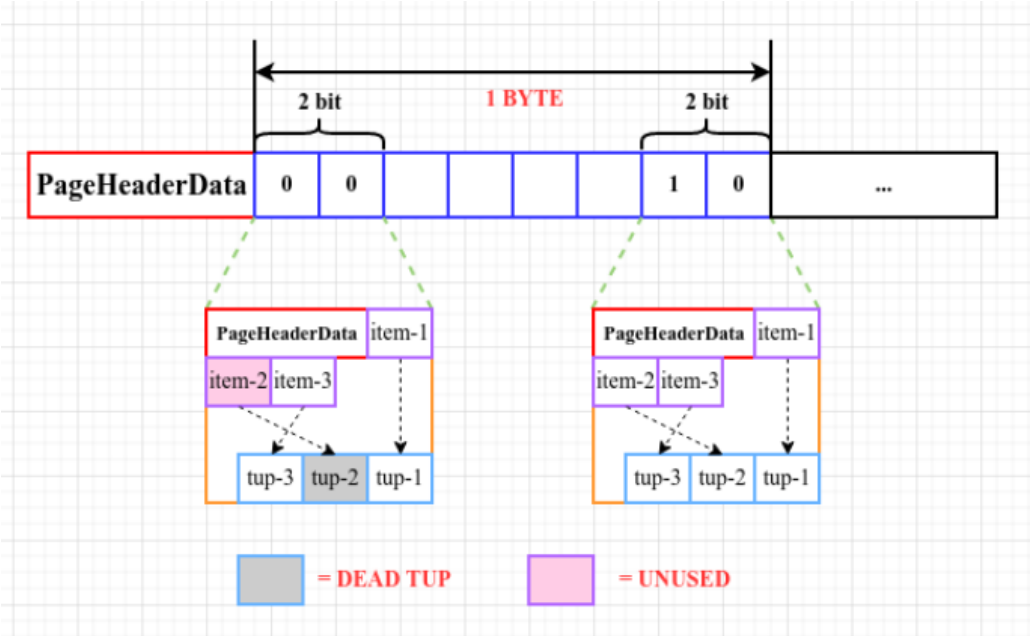


4.1.3 VM 表（可见性映射表）

在 Vacuum 过程中，有很多张表，一个表有很多页组成，如何快速定位到含有无效元组的数据页在高并发场景非常关键。因此出现 VM 表。 每个编号为"***OID***"的表文件对应一个可见性映射表“***OID_vm***”。

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。

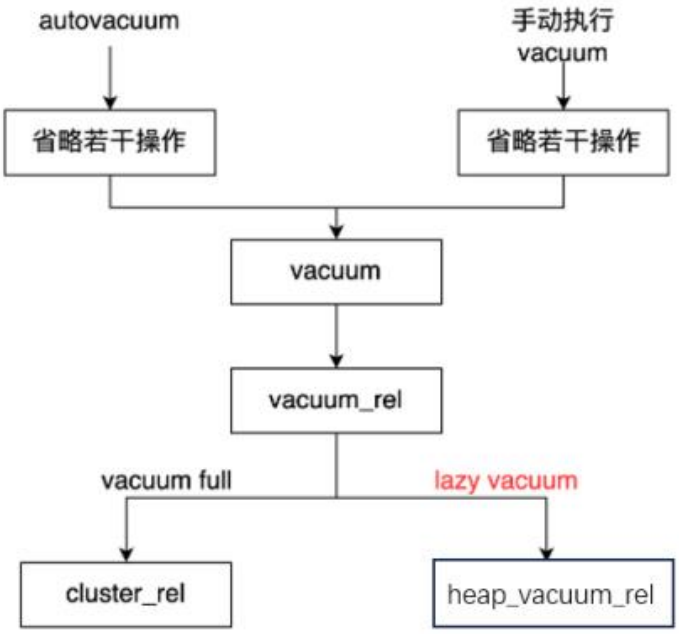
all-visible 比特位的设置表明页内所有元组对于后续所有的事务都是可见的，因此该页无需进行 vacuum 操作；all-frozen 比特位的设置表明页内所有的元组已被冻结，在进行全表扫描 vacuum 请求时也无需进行 vacuum 操作。

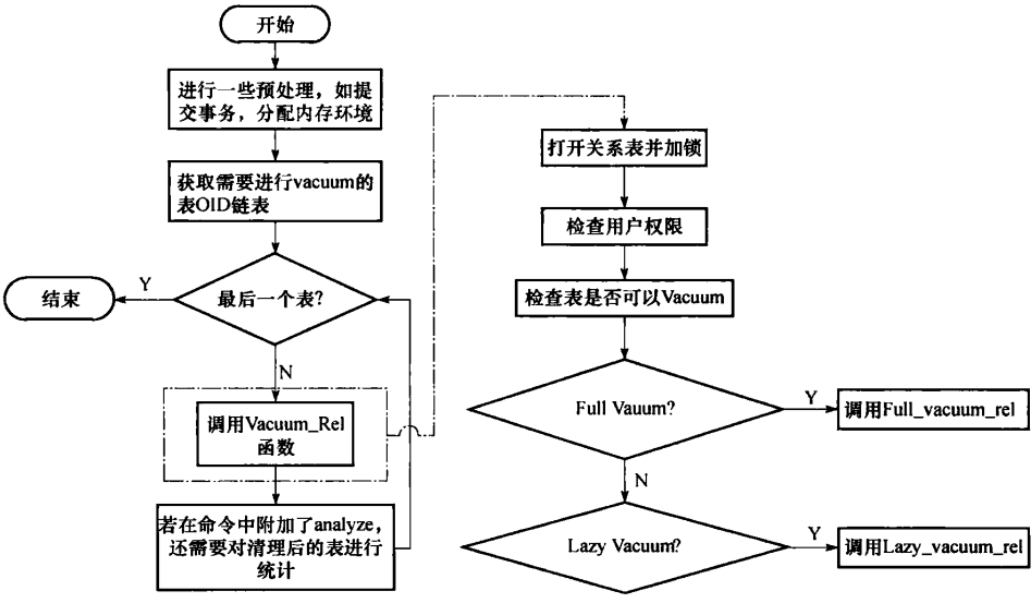


4.1.4 Vacuum 机制实现流程

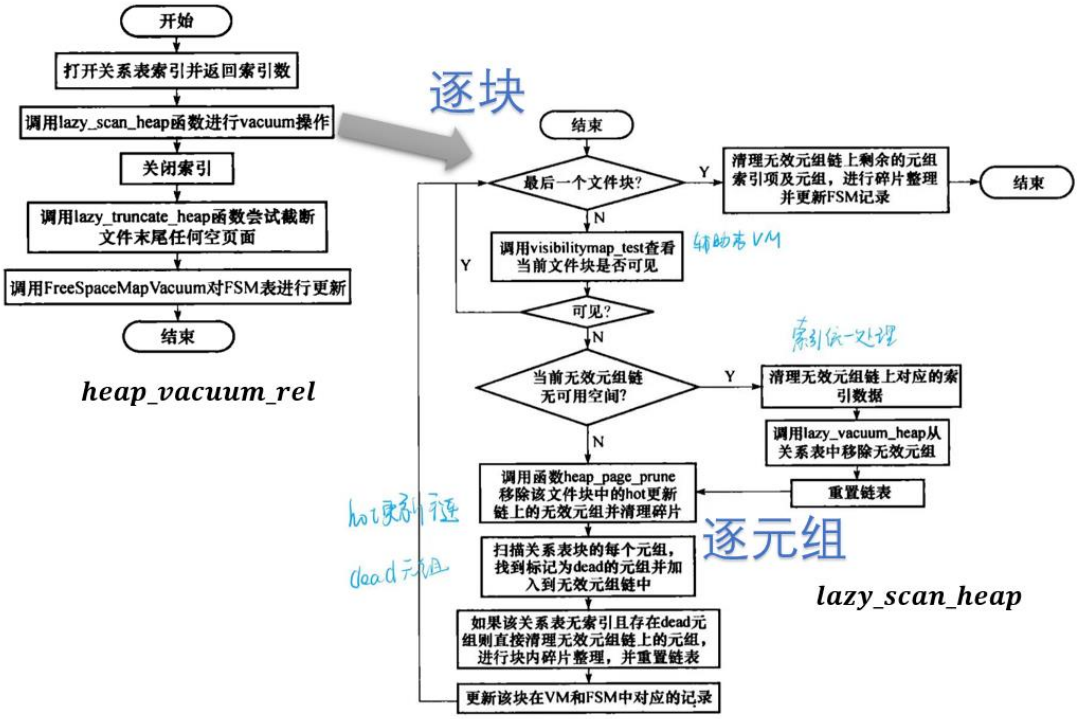
1）总体流程

Vacuum 可以在满足条件时，系统自动触发；也可以手动输入 SQL 语句 Vacuum 来进行制定模式的 Vacuum

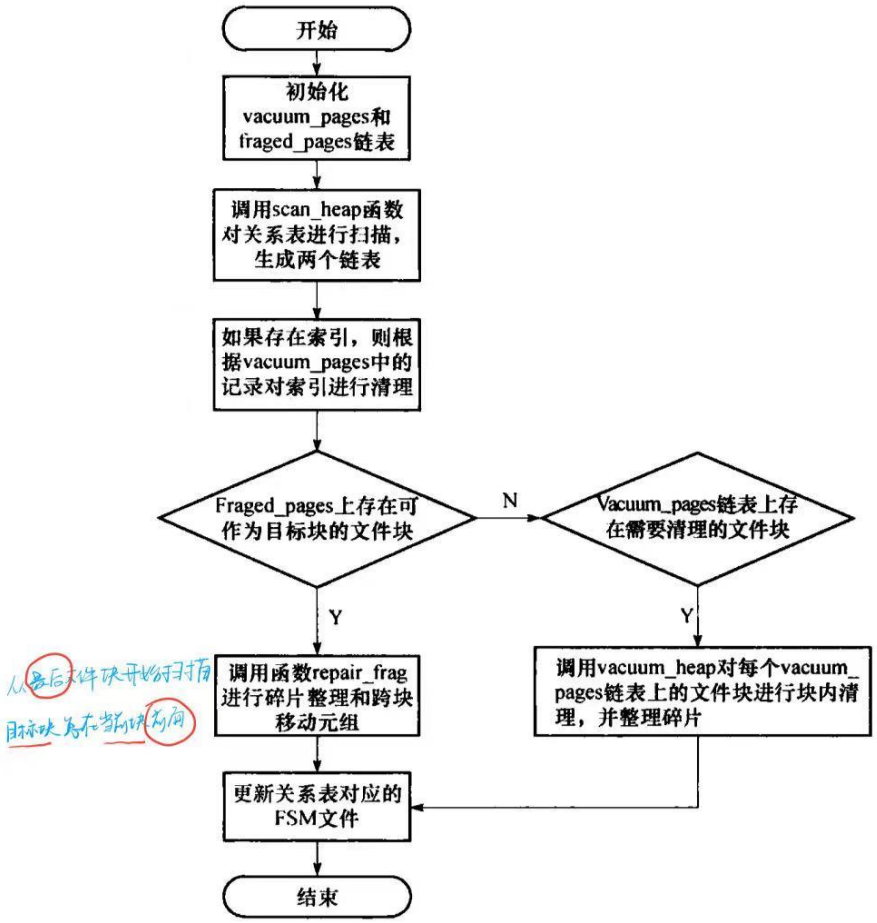




2）Lazy_Vacuum



3）Full_Vacuum



4.2 Lazy_Vacuum 核心代码的解析

对于 Lazy_Vacuum 中存储相关部分，核心内容分为以下三个操作：HOT-update chains 标记，维护 dead_items 无效元组链，索引和元组的统一清理。

（下面各图中，标黄的部分，表示该操作涉及的函数调用路径）

1） HOT-update chains 标记

heap_vacuum_rel

- lazy_scan_heap [backend\access\heap\vacuumlazy.c] **进行lazyvacuum操作**
  - 多次调用lazy_scan_skip函数 **使用VM表（visibility map）记录不需要扫描的块**
  - lazy_scan_prune 被调用于line:1048（逐块_for循环里） [backend\access\heap\vacuumlazy.c]
    - heap_page_prune 被调用于line:1590 [backend\access\heap\pruneheap.c] **页面清理和修复碎片化**

      (heap_page_prune会遍历指定页面的所有元组，对每个元组调用heap_prune_chain)
      - heap_prune_chain

        步骤1：遍历元组的HOT链

        步骤2：判断元组是否过期

        需要注意的是，HOT链上的元组是按照版本的先后顺序排列，所以只需要找到最后一条过期的原数，它之前的所有元组都过期了。

        步骤3：记录需要设置为LP_UNUSED、LP_DEAD、LP_REDIRECT的元组
      - heap_page_prune_execute

        步骤1.依据heap_prune_chain的处理结果，将相关元组设置为LP_UNUSED、LP_DEAD、LP_REDIRECT。

        步骤2.调用PageRepairFragmentation对页面内进行空间整理，即删除过期元组的元组实体部分

2） 维 护 dead_items 无 效 元 组 链

heap_vacuum_rel

- lazy_scan_heap [backend\access\heap\vacuumlazy.c] **进行vacuum操作**
  - 多次调用lazy_scan_skip函数 **使用VM表（visibility map）记录不需要扫描的块**
  - lazy_vacuum(LVRelState *vacrel) 被调用于line:953（逐块_for循环里，如果当前dead_items死元组列表快满了，调用该函数），line:1264（逐块_for循环结束之后，执行dead_items死元组列表剩余元组，最后调用一次） **调用该函数清理索引和元组**
  - lazy_scan_prune 被调用于line:1048 [backend\access\heap\vacuumlazy.c]
    - heap_page_prune 被调用于line:1590 [backend\access\heap\pruneheap.c] **页面清理和修复碎片化**

      line:1644
    - ItemIdIsDead宏函数 被调用于line:1644 [include\storage\itemid.h]

      line:1907

相关重要代码如下：

相关代码

```
backend\access\heap\vacuumlazy.c
lazy_scan_prune函数：


//line:1644
if (ItemIdIsDead(itemid))
        {
            deadoffsets[lpdead_items++] = offnum;
            continue;
        }


//line:1907
    /*    Now save details of the LP_DEAD items from the page in vacrel*/
    if (lpdead_items > 0)
    {
        VacDeadItems *dead_items = vacrel->dead_items;
        ItemPointerData tmp;

        vacrel->lpdead_item_pages++;

        ItemPointerSetBlockNumber(&tmp, blkno);

        for (int i = 0; i < lpdead_items; i++)
        {
            ItemPointerSetOffsetNumber(&tmp, deadoffsets[i]);
            dead_items->items[dead_items->num_items++] = tmp;
        }
```

```
include\storage\itemid.h
识别元组为dead的宏
/*
 * ItemIdIsDead
 *      True iff item identifier is in state DEAD.
 */
#define ItemIdIsDead(itemId) \
    ((itemId)->lp_flags == LP_DEAD)
```

3） 索引和元组的统一清理

heap_vacuum_rel

- lazy_scan_heap [backend\access\heap\vacuumlazy.c] **进行vacuum操作**
  - 多次调用lazy_scan_skip函数 **使用VM表（visibility map）记录不需要扫描的块**
  - lazy_vacuum(LVRelState *vacrel) 被调用于line:953（逐块_for循环里，如果当前dead_items死元组列表快满了，调用该函数），line:1264（逐块_for循环结束之后，执行dead_items死元组列表剩余元组，最后调用一次） **调用该函数清理索引和元组**
    - lazy_vacuum_all_indexes(vacrel) **对索引进行清理**
    - lazy_vacuum_heap_rel(LVRelState *vacrel) **对元组进行清理**
  - lazy_scan_prune 被调用于line:1048（逐块_for循环里） [backend\access\heap\vacuumlazy.c]

5.大数据存储--TOAST 和大对象

5.1 TOAST 机制

5.1.1 概述

TOAST 是 PostgreSQL 用于存储超大字段值的机制，它只适用于变长类型数据的存储。

在 PostgreSQL 中，页面大小通常是 8KB，并且不允许元组跨多个页面进行存储，所以在 PostgreSQL 无法直接存储内存占据过大的值。那么对于超长的行数据，PostgreSQL 会将其进行压缩或者切片成多个物理行存到另一张系统表(TOAST 表)中。

TOAST 使用的四个存储 TOASTed 数据的策略有四个：

1）plain：既不进行压缩也不进行行外存储，这也是不可变类型唯一可以使用的存储策略；

2）extended：允许压缩以及行外存储，这也是默认的 TOAST 存储策略；

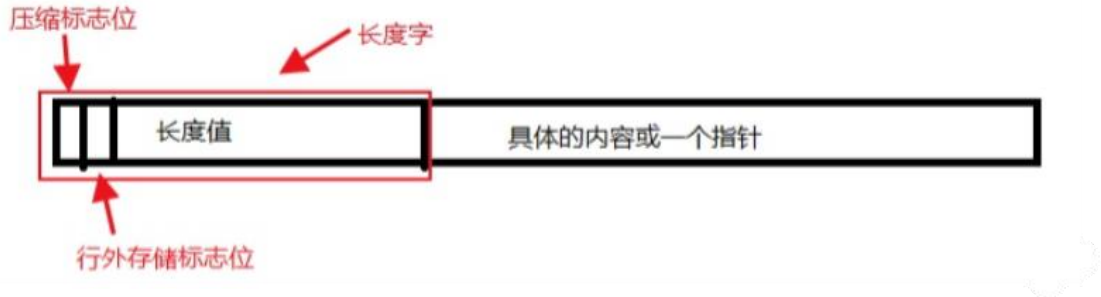
3）external：允许行外存储，但不允许压缩。这种方式可以使大字段的数据读写的速度更快(不需要经过压缩与解压处理)；

4）main：允许压缩，但不允许行外存储。(事实上行外存储还是会进行，但是只有当没有其它的方法可以使数据更小以存储在一个页面中)。

5.1.2 存储结构

1.原来的表

在原表中，对于可 TOAST 的变长类型数据的记录字段如下图：



TOAST 占用使用变长类型的长度字的最高两个二进制位（大端存储机器上的高位，小端存储机器上的低位）作为标记位，这样把任何可 TOAST 值的逻辑长度限制在 1GB。

- 当两个二进制位为 00 时，那么数值是该数据类型一个普通的未 TOAST 的值，并且长度字的剩余位给出整个数据以字节计的大小（包括长度字）。
- 当两个二进制位为 01 时，则表示该数据的内容被压缩过并且在使用前必须先解压。在这种情况中四字节长度字的剩余位指出了压缩过的数据的大小，而不是原始数据的大小。
- 当最高位或者最低位为 1 时，该值只是有一个单字节头部而不是通常的四字节头部，并且该字节的剩余位数给出了以字节计的总数据尺寸（包括长度字节）。这种节省空间的方案支持对低于 127 字节的值的存储，不过需要时仍然允许数据类

型增长到 1GB。带有单字节头部的值不会按照任何特别的边界对齐，反之带有四字节头部的值会按照至少一个四字节边界对齐。这种对齐填充的省略，额外地节省了空间，这种节省比起短值来说更加显著。

- ✧ 作为一种特殊情况，如果一个单字节头部的剩余位全是零（对于一个数据本身来说，这个长度是不可能的），该值就是一个线外数据的指针。这样一个*TOAST 指针*的类型和尺寸由该数据的第二个字节中存储的一个代码决定。（注意对于线外数据也可能存在压缩，但是变长数据的头部不会告诉我们压缩是否发生—TOAST 指针的内容将说明这个问题。）

原表中行外存储的 TOASTED 的值中需要存储的信息包括指向 TOAST 表的指针，TOAST 指针的数据定义如下：

```
typedef struct varatt_external
{
    int32       va_rawsize;     /* Original data size (includes header) */
    uint32      va_extinfo;     /* External saved size (without header) and
                                 * compression method */
    Oid         va_valueid;     // ①线外存储数据的OID
    Oid         va_toastrelid;  // ②TOAST表的OID
} varatt_external;
```

2.系统表

同时，任何存在 TOAST 属性字段的表都会存在一个关联的 TOAST 表格，该表格的 OID 保存在 pg_class.reltoastrelid 中。

3. TOAST 表

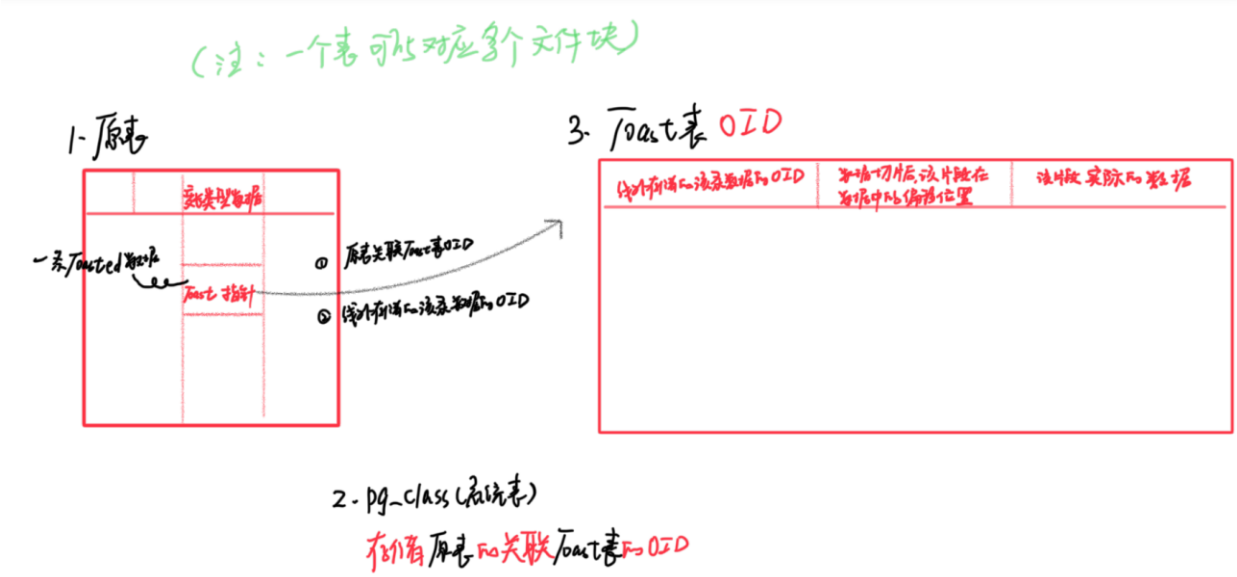
TOAST 表中仅含有三个字段：

表 3-2 TOAST 表属性

属性名称	类型	描述
chunk_id	oid	线外存储时为整个 TOAST 数据分配的 OID
chunk_seq	int4	一个序列号，存储该片段在整个 TOAST 数据中的位置
chunk_data	bytea	该片段实际的数据

4.总结梳理

总结起来，TOAST 机制存储结构的整体关系如下图所示：



5.2 大对象机制

TOAST 是一种自动触发的大数据存储机制，且不适合存储文件。而大对象属于用户手动调用机制。

一个大对象将会被分成若干元组存放在系统表 pg_largeobject 中，每一个元组也称为一个页面。该系统表含有以下三个字段：

```
CATALOG(pg_largeobject,2613,LargeObjectRelationId)
{
    Oid         loid BKI_LOOKUP(pg_largeobject_metadata);    //大对象的标识符
    int32       pageno;          //页面号码
    bytea       data BKI_FORCE_NOT_NULL;     //存储在大对象中的实际数据
} FormData_pg_largeobject;
```

每条大对象数据是使用其创建时分配的 OID 标识的，即 OID 是 PostgreSQL 数据库识别大对象的唯一标记。在对大对象进行实际操作时，需要先为其创建一个实例 LargeObjectDesc

```
typedef struct LargeObjectDesc
{
    Oid         id;             /* LO's identifier */
    Snapshot    snapshot;       /* snapshot to use */
    SubTransactionId subid;     /* owning subtransaction ID */
    uint64      offset;         /* current seek pointer */
    int         flags;          /* see flag bits below */
} LargeObjectDesc;
```

5.2 TOAST 和大对象机制的对比

TOAST 只适用于存储变长类型数据，不适合存储文件。而且是一种自动触发的大数据

存储机制。

大对象支持的数据类型更加广泛，且属于用户手动调用机制。

详细来讲：

1）调用时间：TOAST 用于存储变长字段的数据，只要表中含有变长的数据类型，都会自动创建 TOAST 表，但只有存储的元组字段数据长度超过指定值时才会促发 TOAST 存储机制。而大对象的操作是客户端通过代码或者 SQL 命令调用的，由用户手动调用；

2）数据丢失处理：TOAST 中的数据不能丢失，一旦丢失则会报错；而大对象的数据允许丢失，如果丢失则用 0 代替；

3）文件存储：TOAST 存储文件时，需要先将文件读取出来(一般以二进制的方法)，然后将二进制当作字符串存储到变长的数据类型中。而读取时则需要将数据读取出来，再转化为文件。大对象则直接将文件作为一个对象存储到大对象表中；

4）数据处理：TOAST 中会对存储的数据进行压缩处理，然后再分片进行线外存储。而大数据则直接进行分片并存储。

三. 示例 SQL 语句的执行过程展示

（一）、索引部分

重点展示索引树的创建、遍历、节点分裂算法。

(1). 这里举一个 create index 语句的例子，来帮助更好地了解索引树的创建流程。

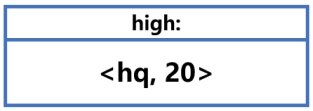
假设有一个表 s，其中按顺序依次有四个属性列：<name，age，gender，height>，我们在<name，age>两个属性列上创建了索引。现在表中有 3 个表元组如下：

Name	Age	Gender	Height
hq	20	男	176
iy	22	男	177
pj	15	男	180
pj	24	男	183

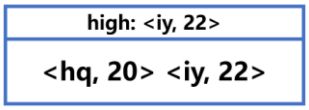
现在我们执行 create index index_name on s (name, age); 这条 sql 语句会针对 s 表和<name, age>两个目标属性列调用函数 btbuild。在这个函数中，首先四个表元组分别被封装成 4 个索引元组，他们的键值依次为：<hq, 20> <iy, 22> <pj, 15> <pj, 24>。

- 首先为第 0 层（树上最低的一层）创建一个 BTPageState 数据结构，并给其

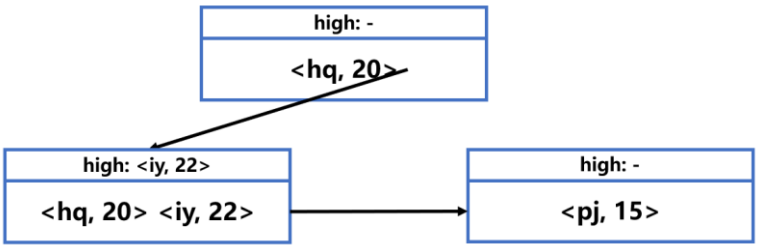
中的 Page 属性分配一个空间。然后将第一个索引元组<hq, 20>插入到该 Page 属性页中。



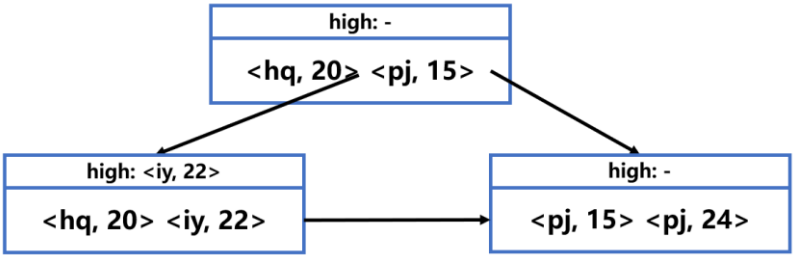
- 将<iy, 22>键值对应的索引元组往第 0 层的 BTPageState 结构体里的 Page 属性对应的页面中插，得到下图。



- 将<pj, 15>键值对应的索引元组往第 0 层的 BTPageState 结构体里的 Page 属性中插。但是发现这个 page 插满了（认为一个 page 最多插 2 个索引元组），此时，为第 1 层创建一个 BTPageState 结构并为其中的 Page 指针分配一个空间，将以 page 的 low key 也就是<hq, 20>为键值，指向此 page 的索引元组插入到第 1 层的 BTPageState 的 page 属性中。同时为第 0 层 new 出一个页面，将<pj, 15>插入到这个页面上，将第 0 层的 BTPageState 的 Page 指针更新为指向这个新的 0 层页面。效果如下图：



- 将<pj, 24>插入到第 0 层 BTPageState 结构的 page 属性中。当之后再要插入新的索引元组时发现第 0 层的 page 都满了，才会创建第 0 层第二个 page 和父节点之间的链接（同样是第 0 层第二个页面的 low key 作为键值，指针指向自己，插入到父节点上），这里我们直接把这个过程也体现出来了。



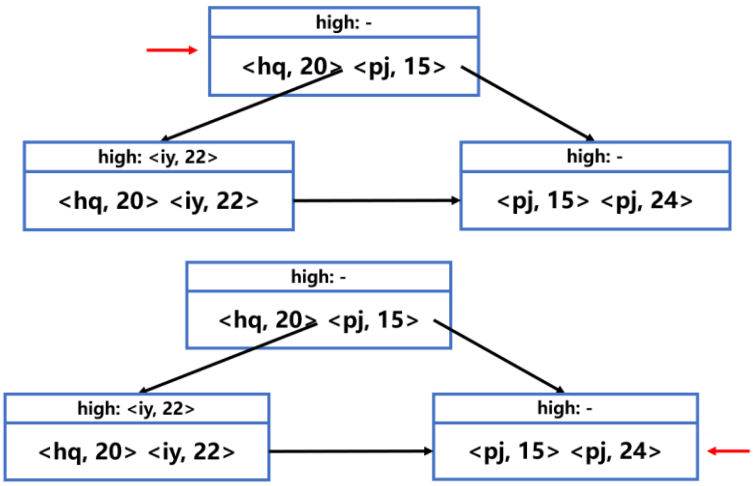
至此我们完成了 b 树索引的创建。

(2). 这里举一个 insert 语句作为例子来向上面 create 出来的索引树上插入新的索引元组，以此来展示索引树的遍历和递归分裂的逻辑。

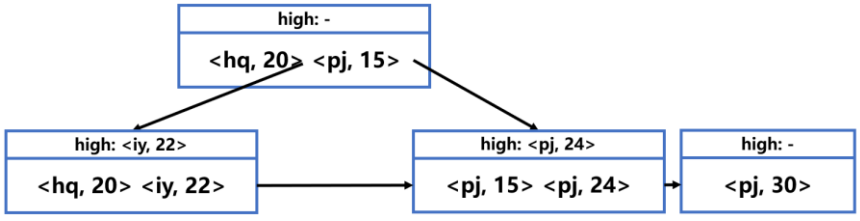
insert into s values('pj', 30, '男', 176);

执行上述语句插入元组时由于要维护索引树所以会触发索引相关的逻辑。代码执行到索引相关部分的入口函数是 btinsert，将新的索引元组<pj, 30>插入到树中。首先进行树的遍历。

- getroot 函数获取到根 Page 指针
- 接下来需要层内右移，不过 root 节点在层内是最右节点了，所以直接右移结束。在该 Page 进行二分搜索，发现比待插入的索引元组键值小的最后一个元组是<pj, 15>，所以我们应该下降一层，来到<pj, 15>指向的页面（将表示遍历到的页面的 Page 指针指向该下层页面）。



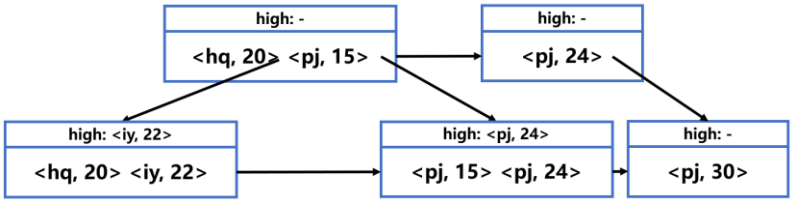
- 从<pj, 15>指向的页面开始向右移，由于该页面也已经是最右节点了因此直接右移结束。页内二分，发现这个待插入的索引元组应该插在本页面中<pj, 24>右边的位置。但是这个页面已经满了。所以就要触发分裂的逻辑了。
- 将此页面分裂成两个，将此页面的两个元组和待插入元组分别插在分裂后的两个子页面上（我们这里选取<pj, 24>为分裂点， 此分裂点归属左页面上）



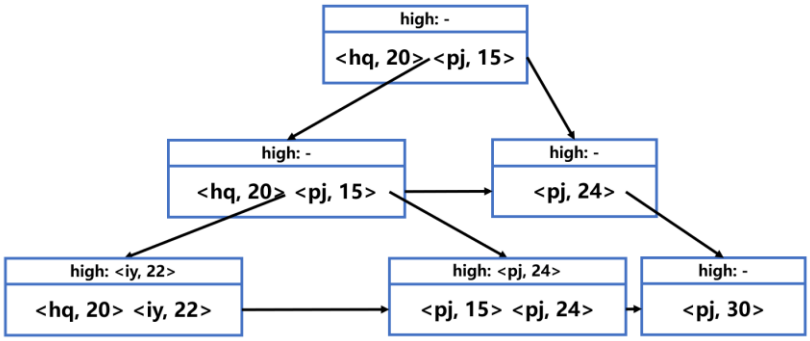
- 将以分裂后的子左页面 high key<pj, 24>为键值，指向右子页面的索引元组用 _bt_insertonpg 函数插入到原本页面的父节点上。但是发现父节点也插满了。

触发父节点的分裂逻辑。

- 父节点分裂出两个页面，将其上的两个元组和待插入的<pj, 24>元组分配到这两个左右页面上（以<pj, 15>为分裂点）。



然后将以左页面 high key<pj, 15>为键值指向自己的索引元组插入到自己的父节点上（这个插入是递归分裂的要求）。但是自己是根，没有父节点，此时 new 一个父节点。同时将以自己的 low key<hq, 20>为键值指向自己的索引元组加入到这个新的父节点（也是新的根节点）上（这个插入是 new 新根的要求）。



- 完成递归分裂，也完成了因为分裂的完成而顺利完成了插入操作。

（二）、内存部分

在该部分，对于数据库中两大重要的逻辑结构——表和元组分别介绍其 PostgreSQL 源码中在内存中的数据结构，然后结合具体的 SQL 语句展示相关执行流程。

表

在 PostgreSQL 源码中，含有 Relation 字眼的结构一般与表有关。

表的数据结构

在内存中，表使用结构体 RelationData 表示，其具体数据结构如下：

```
typedef struct RelationData
{
    RelFileNode rd_node;       /* 关系的物理标识符 */
    SMgrRelation rd_smgr;      /* 缓存的文件句柄，或为NULL */
    int rd_refcnt;             /* 引用计数 */
    BackendId rd_backend;      /* 拥有此关系的后端ID，如果是临时关系 */
    bool rd_islocaltemp;       /* 关系是当前会话的临时关系 */
    bool rd_isnailed;          /* 关系在缓存中被固定 */
    bool rd_isvalid;           /* relcache条目是否有效 */
    bool rd_indexvalid;        /* rd_indexlist是否有效？（也包括rd_pkindex和rd_replidindex） */
    bool rd_statvalid;         /* rd_statlist是否有效 */

    SubTransactionId rd_createSubid;               /* 关系在当前事务中创建 */
    SubTransactionId rd_newRelfilenodeSubid;       /* 更改rd_node的最高子事务 */
    SubTransactionId rd_firstRelfilenodeSubid;     /* 更改rd_node的最高子事务 */
    SubTransactionId rd_droppedSubid;              /* 与其他Subid设置一起删除 */

    Form_pg_class rd_rel;          /* RELATION元组 */
    TupleDesc rd_att;              /* 元组描述符 */
    Oid rd_id;                     /* 关系的对象ID */
    LockInfoData rd_lockInfo;      /* 锁定关系的锁管理器信息 */
    RuleLock *rd_rules;            /* 重写规则 */
    MemoryContext rd_rulescxt;     /* 用于rd_rules的私有内存上下文，如果有的话 */
    TriggerDesc *trigdesc;         /* 触发器信息，如果关系没有触发器则为NULL */
```

其中的重要参数解释如下：

- rd_node: 表的物理标识符，这是在外存中唯一确定一个关系的标志
- rd_refcnt: 引用计数，和缓冲区的调度，事务处理等相关
- rd_rel: 关系元组指针，记录了表的名称、类型等信息，存储在系统表中
- rd_id: 关系对象标识符，用于在内存中唯一标识该关系
- rd_att: 元组描述符，数据结构是 TupleDesc，用于描述关系中存储的元组的详细信息，包括元组各个属性的描述、元组的键的信息等，可以看做是元祖各个方面的详细记录，方便元组在该表中的存取，和对该表中元组的扫描；其具体数据结构如下（注意，该结构存在于表的数据结构中，而不是下文提到的元组的数据结构中）：

```
//元组描述符
typedef struct TupleDescData
{
        int     natts;          //元组属性个数
        Oid     tdtypeid;       //元组的复合类型（行类型）ID
        int32 tdtypmod;         //元组模式
        int     tdrefcount;     //元组描述符引用次数
        TupleConstr *constr;    //元组约束条件
        //attrs[N]为属性号N+1的属性描述
        FormData_pg_attribute attrs[FLEXIBLE_ARRAY_MEMBER];
}                       TupleDescData;
typedef struct TupleDescData *TupleDesc;
```

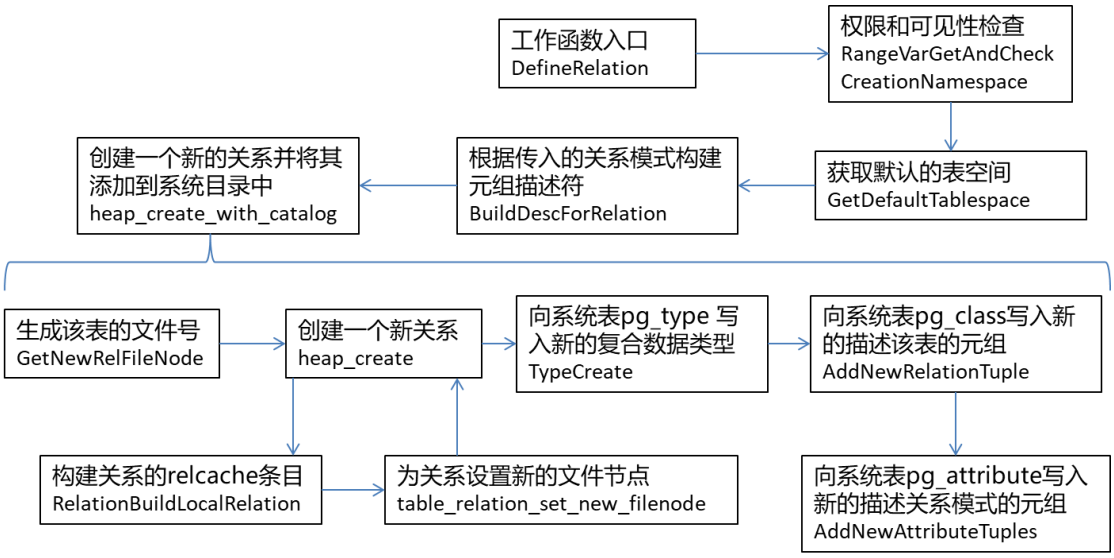
表的 SQL 操作实例

（1） 表的创建

使用的 SQL 语句是：

```
CREATE TABLE Persons (
  Id_P int PRIMARY KEY NOT NULL, LastName varchar(255),
  FirstName varchar(255), Address varchar(255),
  City varchar(255) );
```

经代码调试，省略一些较为细碎的步骤，得到该 SQL 语句在内存部分的执行步骤如下：



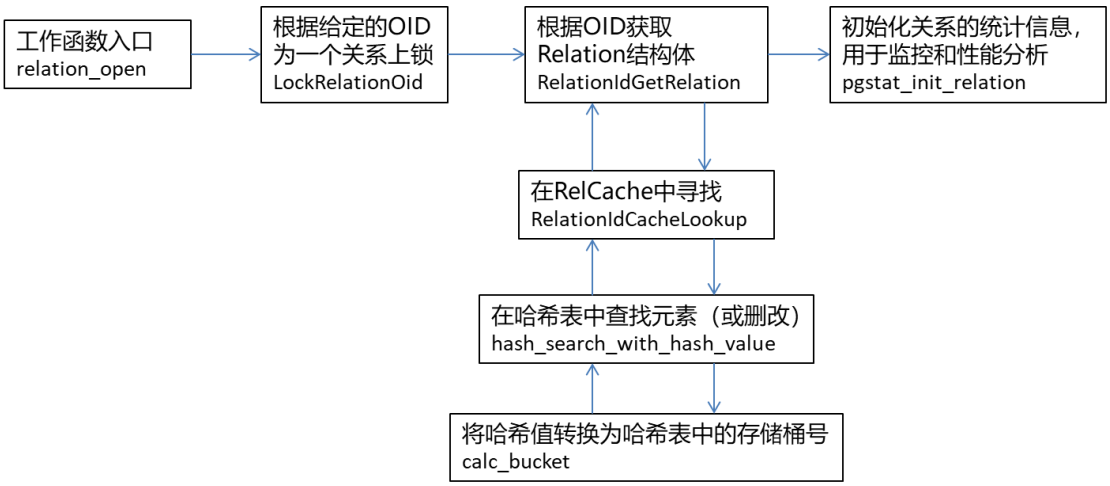
表的创建，在内存部分可以抽象地理解为分配空间和在相关的系统表内登记该表的一系列信息，具体的：

- 从函数入口出发，会先对当前用户的权限，空间的大小等进行检查，满足即可进入正式创建表的步骤
- 根据 SQL 语句编译阶段传来的关系模式，构建元组描述符，将描述符传入创建表的核心函数 heap_create_with_catalog
- 在该核心函数中，先对内外存的存储进行操作：为表分配物理标识符以写入外存的文件中，将表的元组写入 relCache 中以加速后续对该表的访问
- 后在系统表中登记该表的一系列后续使用表需要查询的信息，包括表的复合数据类型信息、关系模式信息等。

（2） 表的打开

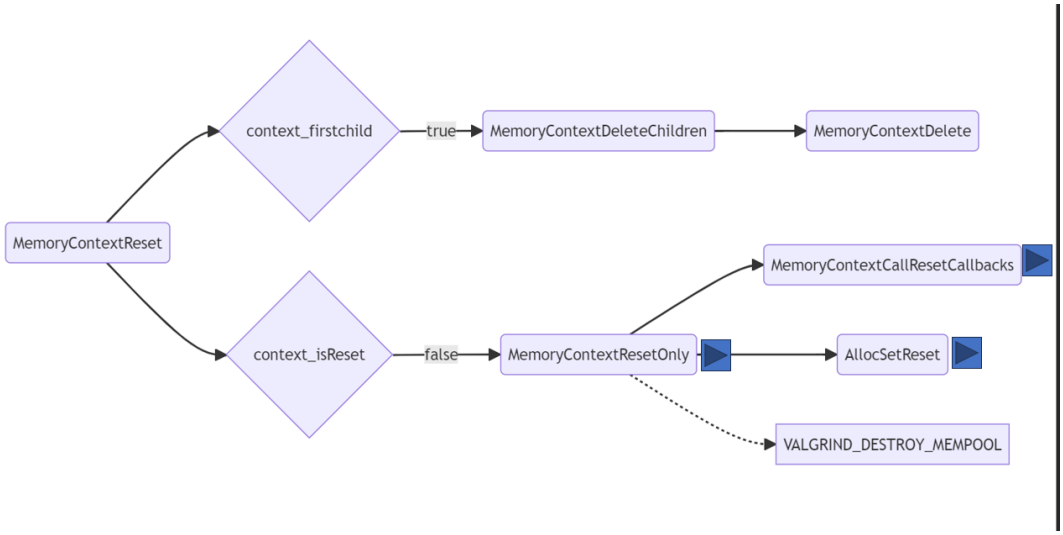
打开表这一操作没有专门与之对应的 SQL 语句，因为所有关于表的操作（插

入，创建，删除）过程中都需要打开表；所谓打开表，在内存中来说就是根据表的 Oid （前文结构中的 rd_id）将表的结构体 RelationData 指针返回给相应的处理函数，让其能根据表的 RelationData 信息对表进行一些特定的操作；以下是打开表的执行流程：



从 relation_open 进入，为保证数据的完整性和一致性，先对该表上锁防止进程冲突，后进入表打开的核心函数 RelationIdGetRelation ，该函数会先在 RelCache 中利用 Oid 结合 Hash 函数进行快速查找，如果无法找到（图中的执行流程没有表示），会一次检查主存和外存，并将其写入 Cache。最后做一些和内存部分关系不大的收尾分析，返回表结构体。

（3） 表的删除



drop 的执行过程函数调用如上，下面我们进行各模块拆解分析。

MemoryContextResetonly

MemoryContextResetOnly

```
if (!context->isReset)
    {
        MemoryContextCallResetCallbacks(context);

        context->methods->reset(context);
        context->isReset = true;
        VALGRIND_DESTROY_MEMPOOL(context);
        VALGRIND_CREATE_MEMPOOL(context, 0, false)
    }
```

- MemoryContextResetonly: 用于重置给定内存上下文（MemoryContext）中的分配器，释放所有已分配内存

具体处理流程如下：

1.如果自上次重置以来没有任何 palloc 分配，则无需执行任何操作。这里的 isReset 表示自上次重置以来是否有新的内存分配

2.如果有新的内存分配，则调用 MemoryContextCallResetCallbacks 函数，执行与重置相关的回调函数。

3.在重置之前，如果上下文标识符（ident）指向上下文的内存区域，那么它将成为一个悬空指针。为了避免这种情况，可以将其设为 NULL，但这会破坏一些合法的编码模式，例如将标识符存储在父上下文中。因此，这里假设程序员已经处理好了这个问题。

4.调用上下文的 reset 方法，重置分配器并释放所有已分配内存。

5.将 isReset 标记设置为 true，表示上下文已被重置。

6.用 VALGRIND_DESTROY_MEMPOOL 和 VALGRIND_CREATE_MEMPOOL 宏，更新 Valgrind 的内存池信息，以便在调试时能够更好地追踪内存泄漏。

MemoryContextCallResetCallbacks

MemoryContextCallResetCallbacks

```
MemoryContextCallback *cb;

while ((cb = context->reset_cbs) != NULL)
{
    context->reset_cbs = cb->next;
    cb->func(cb->arg);
}
```

MemoryContextCallResetCallbacks :用于调用上下文中注册的所有重置回调函数。

具体实现如下：

1.定义一个指针变量 cb，用于遍历上下文中的重置回调链表。

2.在循环中，每次从链表头部弹出一个回调对象，并将其保存到 cb 变量中。如果链表为空，则退出循环。

3.调用回调函数的 func 成员，并传递回调函数的参数 arg。

4.在调用回调函数之前，将回调从链表中移除。这样做的目的是，如果在回调函数内部发生错误，我们不会尝试在稍后重置或删除上下文时再次调用它。

5.总的来说，这段代码的作用是调用上下文中注册的所有重置回调函数。用于释放与上下文相关的资源，例如文件句柄、锁等

AllocSetReset

AllocSetReset

```
AllocSet  set = (AllocSet) context;
AllocSetCheck(context);
MemSetAligned(set->freelist, 0, sizeof(set->freelist));
set->blocks = set->keeper;
if (block == set->keeper)
{
	char  *datastart = ((char *) block) + ALLOC_BLOCKHDRSZ;
	wipe_mem(datastart, block->freeptr - datastart);
	block->freeptr = datastart;
} else {
	context->mem_allocated -= block->endptr - ((char *) block);
	wipe_mem(block, block->freeptr - ((char *) block));
		free(block);
}
set->nextBlockSize = set->initBlockSize;
```

- AllocSetReset：重置给定内存集合（AllocSet）中分配的所有内存。

具体实现如下：

1.将传入的上下文转换为 AllocSet 类型，并将其保存到 set 变量中。

2.对 set 进行有效性检查，确保其为有效的 AllocSet。

3.（可选）在释放内存之前，对上下文进行一些完整性检查，以检测内存破坏或泄漏。这一步是可选的，可能会在编译时根据宏 MEMORY_CONTEXT_CHECKING 的定义决定是否启用。

4.清空内存块的空闲列表。

5.将 blocks 指针指向 keeper 块，即保留一个块不释放。这样做的目的是避免在重复进行小内存分配后，每次重置上下文都频繁调用 malloc()，从而减少性能损耗。keeper 块与上下文头部共享一个 malloc 块。

6.遍历所有的内存块，并逐个处理。

7.如果当前块是 keeper 块，则重置该块但不返回给 malloc，即不释放该块。重置操作包括将块中已使用的内存区域标记为可重用状态，并将 freeptr 指针重置为块的起始位置。

8.如果当前块不是 keeper 块，则释放该块。释放操作包括减少上下文的 mem_allocated 记录的内存使用量，并调用 free() 函数释放内存块。

9.重置完所有内存块后，确保上下文的 mem_allocated 记录的内存使用量与 keepersize 相等。

10.最后，重置内存块大小的分配序列。

总的来说，这段代码的作用是重置给定内存集合中分配的所有内存。重置操作包括将已使用的内存标记为可重用状态，并释放除了一个特殊的保留块之外的所有内存块。这样做可以提高性能，避免在重复进行小内存分配时频繁调用 malloc()。

元组

在 PostgreSQL 源码中，含有 Tuple 字眼的结构一般与元组有关。

元组的数据结构

元组在内存中出现的形式是 HeapTupleData 结构体：

```
//元组在内存中的存在形式
typedef struct HeapTupleData
{
    uint32      t_len;          //元组长度
    ItemPointerData t_self;     //该结构自身所在块号及偏移量
    Oid         t_tableOid;     //元组所在表的OID
#define FIELDNO_HEAPTUPLEDATA_DATA 3
    HeapTupleHeader t_data;     //元组头信息
} HeapTupleData;

typedef HeapTupleData *HeapTuple;
```

该结构体中，记录了元组的长度、元组在内存中的位置、元组所属的表的 Oid，元组头结构信息；关于描述元组在内存中的位置的记录 t_self 的数据结构，具体如下：

```
typedef struct ItemPointerData
{
    BlockIdData ip_blkid;       //元组内数据所在块号
    OffsetNumber ip_posid;      //块内偏移量
}

/* If compiler understands packed and aligned pragmas, use those */
#if defined(pg_attribute_packed)
 && defined(pg_attribute_aligned)
            pg_attribute_packed()
            pg_attribute_aligned(2)
#endif
ItemPointerData;
```

在该结构中，指明了元组数据所在的块号，使用内存地区分块的首地址加上该块号乘以块大小（8KB）的结果即可锁定该元组所在的块；结合块内行偏移 ip_posid（上文 Page 部分提到的概念），可以确定元组相对块首地址的偏移量；在该地址上，依次存储的是 HeapTupleData 结构体和元组存储的真正的数据库内的数据。

关于描述元组头部信息 t_data 的数据结构，具体如下：

```
//元组头信息
struct HeapTupleHeaderData
{
    union
    {
        HeapTupleFields t_heap;     //常见形式元组
        DatumTupleFields t_datum;   //紧凑形式元组
    }           t_choice;

    ItemPointerData t_ctid; //当前元组的 TID (Tuple ID)

    /* Fields below here must match MinimalTupleData! */

#define FIELDNO_HEAPTUPLEHEADERDATA_INFOMASK2 2
    uint16      t_infomask2;    //属性信息

#define FIELDNO_HEAPTUPLEHEADERDATA_INFOMASK 3
    uint16      t_infomask;     //标志位信息

#define FIELDNO_HEAPTUPLEHEADERDATA_HOFF 4
    uint8       t_hoff;         //元组头大小，包括位图和填充

    /* ^ - 23 bytes - ^ */

#define FIELDNO_HEAPTUPLEHEADERDATA_BITS 5
    bits8       t_bits[FLEXIBLE_ARRAY_MEMBER];
    //长度不定的位图数组，用于表示元组中哪些属性为 NULL
    /* MORE DATA FOLLOWS AT END OF STRUCT */
};
```

在该结构中，记载了元组的类型，元组中属性的标记；注意这里的两组掩码（infomask）记载的详细信息值得深挖，两组掩码在元组的修改和删除、元组的版本控制起到标记的作用，可以认为只需修改掩码可以标记一个元祖已被删除或修改，以提高数据库的操作效率。

元组的 SQL 操作实例

（1） 元组的插入

使用的 SQL 语句是：

```
insert into Persons
values (123, 'Trump', 'Donald', 'US', 'Washington.D.C');
```

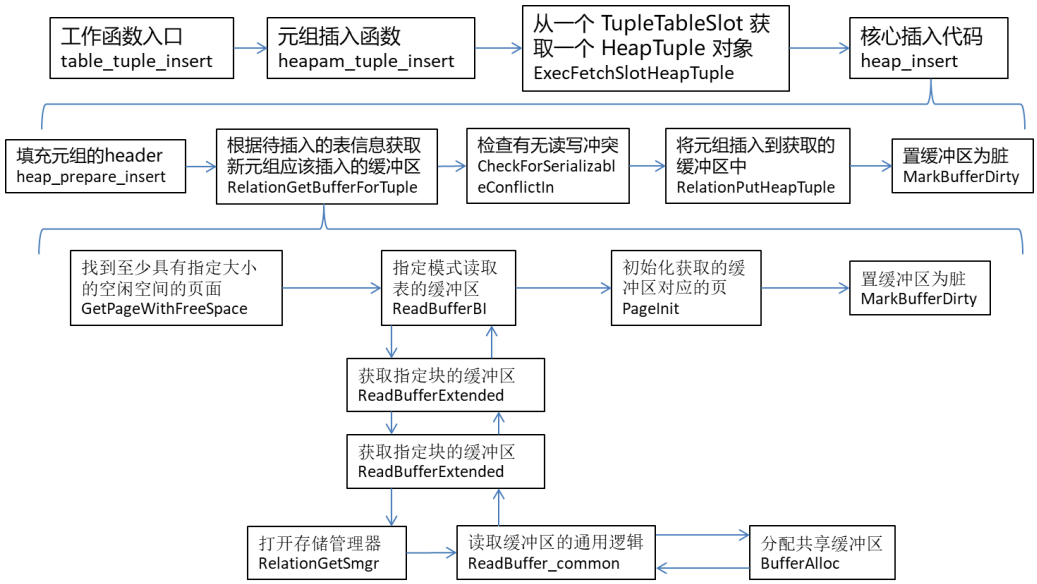
这里我们要先搞清楚，元组信息从一个 SQL 语句到内存中可进行操作的 HeapTupleData 结构中间经历了什么；这涉及到编译阶段和计划节点的生成，在计划节点生成后，ExecProcNode 函数将其中的元组信息和数据提取暂存在 planslot 中；

如果操作成功则 planslot 转化为 slot 返回供下一步插入操作使用；其中 planslot 和 slot 的数据结构都是 TupleTableSlot：

```
typedef struct TupleTableSlot
{
    NodeTag     type;
#define FIELDNO_TUPLETABLESLOT_FLAGS 1
    uint16      tts_flags;      /* 标志位，包含是否为空的标志 */
#define FIELDNO_TUPLETABLESLOT_NVALID 2
    AttrNumber  tts_nvalid;     /* # of valid values in tts_values */
    const TupleTableSlotOps *const tts_ops; /* implementation of slot */
#define FIELDNO_TUPLETABLESLOT_TUPLEDESCRIPTOR 4
    TupleDesc   tts_tupleDescriptor;    /* 存储的元组的描述符 */
#define FIELDNO_TUPLETABLESLOT_VALUES 5
    Datum       *tts_values;    /* 当前每个属性对应的值 */
#define FIELDNO_TUPLETABLESLOT_ISNULL 6
    bool        *tts_isnull;    /* 当前各个属性是否为空 */
    MemoryContext tts_mcxt;     /* 这个结果本身所在的内存上下文 */

    ItemPointerData tts_tid;    /* 存储的元组的 tid */

    Oid         tts_tableOid;   /* 存储元组所在表的 oid */
} TupleTableSlot;
```

该结构可以看作对 HeapTuple 结构的一种包装，用于在指令执行的不同阶段传递元组数据，无需复制实际数据，从而提高性能和减少内存消耗；该结构与 HeapTuple 结构十分类似，在目前内存涉及的部分中，二者的区别是：TupleTableSlot 多用于元组作为参数参与一系列的操作，HeapTuple 则偏向元组存储结构信息的描述，用于页面上的静态存储。

下面是元组插入的调用流程，省略了一些细碎的函数调用和事务处理的步骤：



从 table_tuple_insert 开始分析，从一个 TupleTableSlot 中构建出一个 HeapTuple 对象，进入插入操作的核心函数 heap_insert，执行步骤如下：

- 登记元组的头部信息
- 根据待插入元祖的目标表信息获取应该插入的缓冲区

- ✧ 预先找到符合空间要求的页面（因为后面只是向缓冲区写，这里预先获取一个主存中的页面以供后续写回）
- ✧ 获取指定的表的缓冲区
- ✧ 如果没有读取到，则需要申请一个缓冲区空间，使用刚刚预先查找的页面
- ✧ 初始化缓冲区对应的页面（填充 Page 的头部信息，登记表的 smgr 信息等）
- ✧ 置该缓冲区为脏

- 进程读写冲突检查，冲突则等待调度
- 将元组插入到刚刚获取的缓冲区中
- 置缓冲区为脏

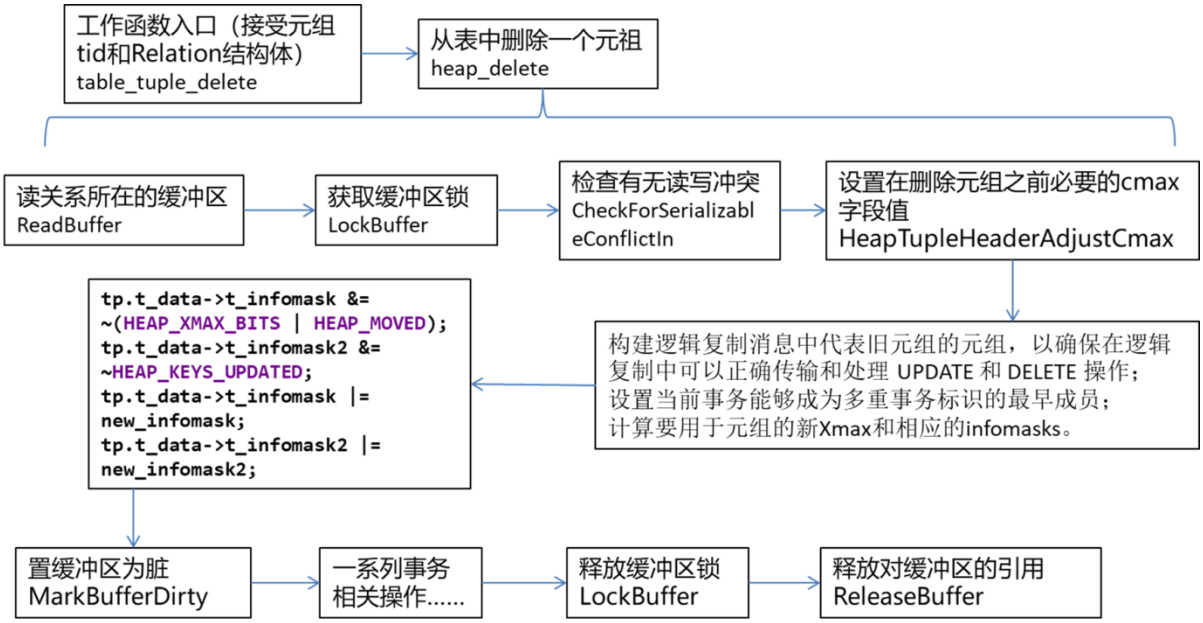
这些操作中较为关键的是指定表的缓冲区是如何确定的，其实在 PostgreSQL 中，每个关系（表、索引等）都与一个物理文件相关联，用于存储数据；为了高效地处理文件 I/O 操作，PostgreSQL 使用文件句柄缓存（Smgr）来跟踪和管理这些文件。获取表对应的缓冲区，需要查找表的 Smgr 结构中指示当前插入该表的目标块（即下图中的 smgr_targblock），将该块返回给插入的核心函数，即可实现定位和插入；关于 Smgr 结构的详细说明和维护可见下文。

```
typedef struct SMgrRelationData
{
    RelFileNodeBackend smgr_rnode;    /* 关系的物理标识符 */

    // 指向拥有者的指针，或者如果没有则为 NULL
    struct SMgrRelationData **smgr_owner;

    // 以下字段在缓存刷新事件后重置为 InvalidBlockNumber
    BlockNumber smgr_targblock; /* 当前插入目标块 */
    BlockNumber smgr_cached_nblocks[MAX_FORKNUM + 1];    /* 最后已知的大小 */

    /*
     * 下面的字段旨在是 smgr.c 及其子模块私有的。请不要从其他地方访问它们。
     */
    int         smgr_which;         /* 存储管理器选择器 */

    // 每个 fork 的打开段数量（md_num_open_segs）和段本身（md_seg_fds）的每个 fork 数组。
    int         md_num_open_segs[MAX_FORKNUM + 1];
    struct _MdfdVec *md_seg_fds[MAX_FORKNUM + 1];

    // 如果没有所有者，以链表方式链接到所有未归属的 SMgrRelations 中
    dlist_node node;
} SMgrRelationData;

typedef SMgrRelationData *SMgrRelation;
```

（2） 元组的删除

在内存中，元组的删除一般是懒惰的，依赖于上文提到的元组数据结构的两组

infomask，进行标记删除，具体流程如下：



从入口出发，读取表所在的缓冲区并获得锁，进行一系列事务处理信息的登记操作，其后进入关键操作：修改元组数据结构中的掩码，将其相应标志位修改为已删除，将缓冲区置为脏，释放锁和引用即可；这样在这个块要被从缓冲区中换出时，相应的数据也会因为置为脏而被写回主存。

但这里可能会产生一个疑惑，如果一直懒惰删除，那么空间中的碎片和无效元组会越来越多，如何回收这些空间呢；关于这点可详见下文关于 VACUUM 机制的描述，其中使用的回收并紧缩内存部分空间的函数见上文提到的 compactify_tuples。

（三）、外存部分

1.表的创建

使用的 SQL 语句是：

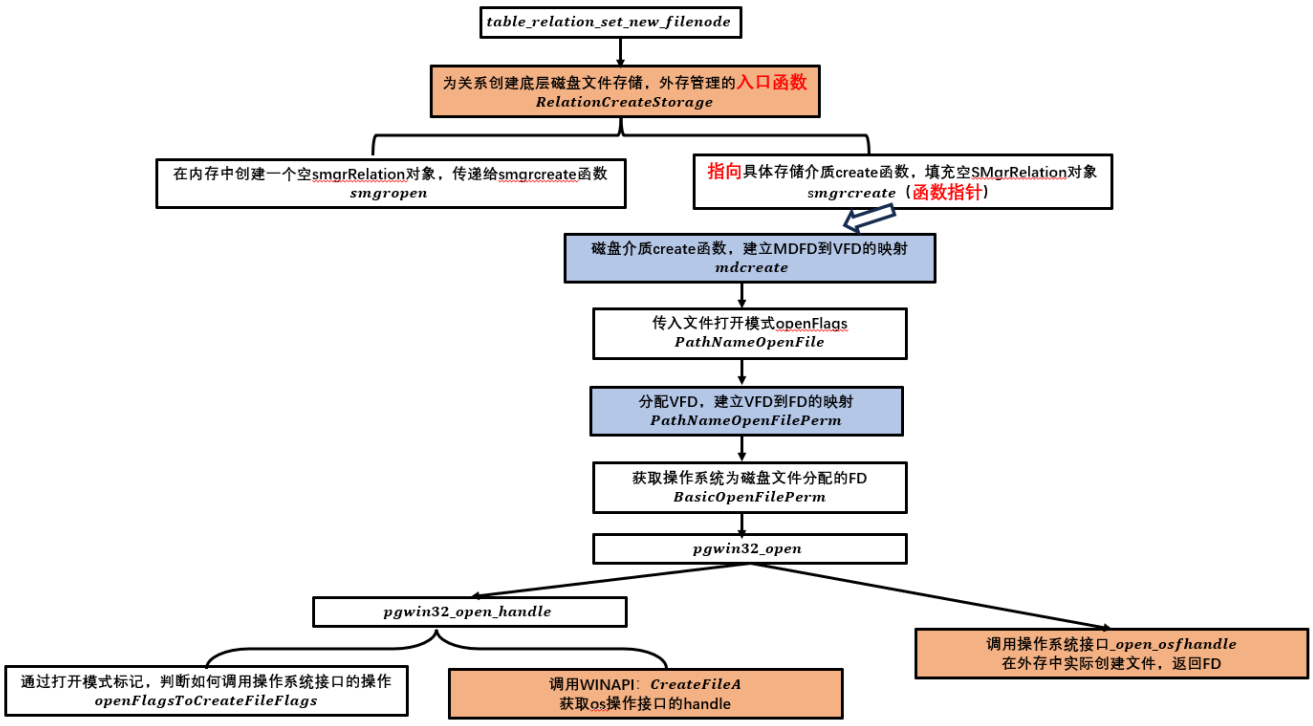
```
CREATE TABLE Persons (
    Id_P int PRIMARY KEY NOT NULL, LastName varchar(255),
    FirstName varchar(255), Address varchar(255),
    City varchar(255) );
```

1.1 调试上述语句的函数调用栈

```
postgres.exe!pgwin32_open_handle(const char * fileName, int fileFlags, bool backup_semantics) 行 68  C
postgres.exe!pgwin32_open(const char * fileName, int fileFlags, ...) 行 165  C
postgres.exe!BasicOpenFilePerm(const char * fileName, int fileFlags, unsigned short fileMode) 行 1125  C
postgres.exe!PathNameOpenFilePerm(const char * fileName, int fileFlags, unsigned short fileMode) 行 1603  C
postgres.exe!PathNameOpenFile(const char * fileName, int fileFlags) 行 1569  C
postgres.exe!mdcreate(SMgrRelationData * reln, ForkNumber forkNum, bool isRedo) 行 208  C
postgres.exe!smgrcreate(SMgrRelationData * reln, ForkNumber forknum, bool isRedo) 行 372  C
postgres.exe!RelationCreateStorage(RelFileNode rnode, char relpersistence, bool register_delete) 行 151  C
postgres.exe!heap_create(const char * relname, unsigned int relnamespace, unsigned int reltablespace, unsigned int relid, unsig...  C
postgres.exe!heap_create_with_catalog(const char * relname, unsigned int relnamespace, unsigned int reltablespace, unsigned i...  C
postgres.exe!DefineRelation(CreateStmt * stmt, char relkind, unsigned int ownerId, ObjectAddress * typaddress, const char * que...  C
postgres.exe!DefineSequence(ParseState * pstate, CreateSeqStmt * seq) 行 220  C
postgres.exe!ProcessUtilitySlow(ParseState * pstate, PlannedStmt * pstmt, const char * queryString, ProcessUtilityContext context...  C
postgres.exe!standard_ProcessUtility(PlannedStmt * pstmt, const char * queryString, bool readOnlyTree, ProcessUtilityContext co...  C
postgres.exe!ProcessUtility(PlannedStmt * pstmt, const char * queryString, bool readOnlyTree, ProcessUtilityContext context, Par...  C
postgres.exe!ProcessUtilitySlow(ParseState * pstate, PlannedStmt * pstmt, const char * queryString, ProcessUtilityContext context...  C
postgres.exe!standard_ProcessUtility(PlannedStmt * pstmt, const char * queryString, bool readOnlyTree, ProcessUtilityContext co...  C
postgres.exe!ProcessUtility(PlannedStmt * pstmt, const char * queryString, bool readOnlyTree, ProcessUtilityContext context, Par...  C
postgres.exe!PortalRunUtility(PortalData * portal, PlannedStmt * pstmt, bool isTopLevel, bool setHoldSnapshot, _DestReceiver * ...  C
postgres.exe!PortalRunMulti(PortalData * portal, bool isTopLevel, bool setHoldSnapshot, _DestReceiver * dest, _DestReceiver * a...  C
postgres.exe!PortalRun(PortalData * portal, long count, bool isTopLevel, bool run_once, _DestReceiver * dest, _DestReceiver * alt...  C
postgres.exe!exec_simple_query(const char * query_string) 行 1258  C
postgres.exe!PostgresMain(const char * dbname, const char * username) 行 4600  C
postgres.exe!BackendRun(Port * port) 行 4512  C
postgres.exe!SubPostmasterMain(int argc, char * * argv) 行 5009  C
postgres.exe!main(int argc, char * * argv) 行 194  C
```

1.2 执行流程

内存管理函数 table_relation_set_new_filenode 会调用 heapam_relation_set_new_filenode 并调用 RelationCreateStorage，进行外存管理操作。可以将 RelationCreateStorage 视为 create 操作外存部分的入口函数。SQL 语句在外存部分的执行步骤如下：



RelationCreateStorage：为关系创建底层磁盘文件存储，外存管理的入口函数

smgropen ：在内存中创建一个空 SMgrRelation 对象，传递给 Smgrcreate 函数

smgrcreate（函数指针）：指向具体存储介质 create 函数，填充空 SMgrRelation 对象

mdcreate：磁盘介质 create 函数，建立磁盘文件描述符(mdfd)到虚拟文件描述符(VFD)的映射

PathNameOpenFile：传入文件打开模式 openFlags

PathNameOpenFilePerm：分配虚拟文件描述符(VFD)，建立 VFD 到 FD 的映射

BasicOpenFilePerm：获取操作系统为磁盘文件分配的物理文件描述符(FD)

pgwin32_open

pgwin32_open_handle

openFlagsToCreateFileFlags：通过打开模式标记，判断如何调用操作系统接口的操作

调用 WINAPI：$\boldsymbol{CreateFileA}$ 获取 os 操作接口的 handle

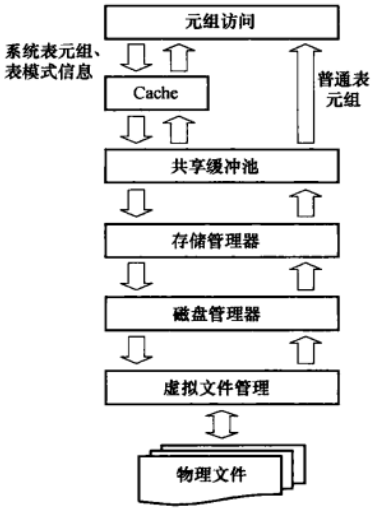
调用操作系统接口$\boldsymbol{_open_osfhandle}$，在外存中实际创建文件，返回 FD

1.3 核心代码的解析：

1.3.1 概述

上述执行流程中，可以核心关注**一个指针**：smgrcreate（函数指针），它指向了具体存储介质 create 函数。**两个函数**：mdcreate 函数，建立了磁盘文件描述符(mdfd)到虚拟文件描述符(VFD)的映射；PathNameOpenFilePerm：建立了虚拟文件描述符(VFD)到真实文件描述符(FD)的映射。

这些则展现了从存储管理器到具体介质管理器；从磁盘管理器到虚拟文件管理；从虚拟文件管理到操作系统真实文件的外存管理精巧架构设计。



1.3.2 具体解析

核心函数的具体解释见下图的注释部分

```
void mdcreate(SMgrRelation reln, ForkNumber forkNum, bool isRedo)//创建底层文件
{
    MdfdVec    *mdfd;
    char       *path;
    File        fd;

    if (isRedo && reln->md_num_open_segs[forkNum] > 0)
        return;                 /* 检查是否早已创建和打开 */
    Assert(reln->md_num_open_segs[forkNum] == 0);
    path = relpath(reln->smgr_rnode, forkNum);//获取文件目录

    fd = PathNameOpenFile(path, O_RDWR | O_CREAT | O_EXCL | PG_BINARY);//获取虚拟文件描述符

    if (fd < 0)//获取描述符失败，即文件打开失败。
    {
        int         save_errno = errno;
        if (isRedo)//如果是已经创建过
            fd = PathNameOpenFile(path, O_RDWR | PG_BINARY);//重新获取虚拟文件描述符
        if (fd < 0)//仍旧获取失败
        {
            errno = save_errno;
            ereport(ERROR,
                    (errcode_for_file_access(),
                     errmsg("could not create file \"%s\": %m", path)));
        }
    }

    pfree(path);

    _fdvec_resize(reln, forkNum, 1);//设置此类型文件的分段数目为1，并且截断分段数组
    mdfd = &reln->md_seg_fds[forkNum][0];//设置分段数组的第一个分段
    mdfd->mdfd_vfd = fd;//添加虚拟文件描述符
    mdfd->mdfd_segno = 0;//段号设置为0
}
```

```
//  为文件分配VFD的函数
File PathNameOpenFilePerm(const char *fileName, int fileFlags, mode_t fileMode)
{
    char       *fnamecopy;
    File        file;
    Vfd        *vfdP;
/* We need a malloc'd copy of the file name; fail cleanly if no room.*/
fnamecopy = strdup(fileName);
if (fnamecopy == NULL)
    ereport(ERROR,
            (errcode(ERRCODE_OUT_OF_MEMORY),
             errmsg("out of memory")));

file = AllocateVfd();   //从空闲vfd链表中分配vfd节点
vfdP = &VfdCache[file];

/* Close excess kernel FDs. */
ReleaseLruFiles();  //若超过则从lru中选择一个关闭，保证不会从操作系统多请求fd

vfdP->fd = BasicOpenFilePerm(fileName, fileFlags, fileMode);    //打开物理文件，并返回取自操作系统的真实文件描述符FD，建立VFD与FD映射

if (vfdP->fd < 0)
{
    int         save_errno = errno;
    FreeVfd(file);
    free(fnamecopy);
    errno = save_errno;
    return -1;
}
++nfile;
Insert(file);   //将索引file的Vfd结构体插入LRU环形队列的front
return file;
}
```

```
//  获取操作系统为文件分配的FD的函数
BasicOpenFilePerm(const char *fileName, int fileFlags, mode_t fileMode)
{
    int         fd;
...
#else
    fd = open(fileName, fileFlags, fileMode);
...
}
```

```
//fileapi.h

WINBASEAPI
HANDLE
WINAPI
CreateFileA(
    _In_ LPCSTR lpFileName,
    _In_ DWORD dwDesiredAccess,
    _In_ DWORD dwShareMode,
    _In_opt_ LPSECURITY_ATTRIBUTES lpSecurityAttributes,
    _In_ DWORD dwCreationDisposition,
    _In_ DWORD dwFlagsAndAttributes,
    _In_opt_ HANDLE hTemplateFile
    );
```

2.表的删除

使用的 SQL 语句是：

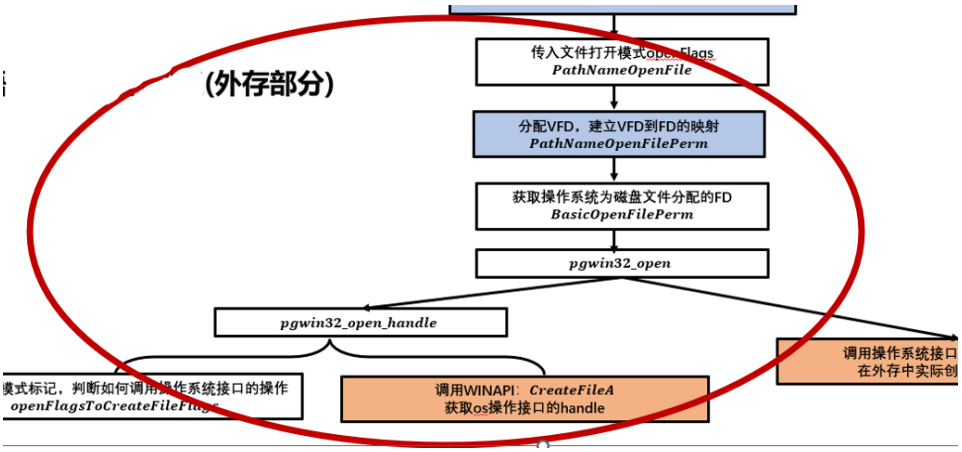
```
DROP TABLE Persons;
```

（这里仅粗略展现表的删除的执行，目的仅是与表的创建语句形成对比。）

执行 DROP 语句的函数调用栈

```
postgres.exe!PathNameOpenFilePerm(const char * fileName, int fileFlags, unsigned short fileMode) 行 1603
postgres.exe!PathNameOpenFile(const char * fileName, int fileFlags) 行 1569
postgres.exe!mdopenfork(SMgrRelationData * reln, ForkNumber forknum, int behavior) 行 529
postgres.exe!mdexists(SMgrRelationData * reln, ForkNumber forkNum) 行 173
postgres.exe!smgrexists(SMgrRelationData * reln, ForkNumber forknum) 行 250
postgres.exe!DropRelFileNodesAllBuffers(SMgrRelationData * * smgr_reln, int nnodes) 行 3244
postgres.exe!smgrdounlinkall(SMgrRelationData * * rels, int nrels, bool isRedo) 行 439
postgres.exe!smgrDoPendingDeletes(bool isCommit) 行 706
postgres.exe!CommitTransaction() 行 2345
postgres.exe!CommitTransactionCommand() 行 3049
postgres.exe!finish_xact_command() 行 2761
postgres.exe!exec_simple_query(const char * query_string) 行 1276
postgres.exe!PostgresMain(const char * dbname, const char * username) 行 4600
postgres.exe!BackendRun(Port * port) 行 4512
postgres.exe!SubPostmasterMain(int argc, char * * argv) 行 5009
postgres.exe!main(int argc, char * * argv) 行 194
[外部代码]
```

删除表操作，首先需要查看该表在外存中是否还存在，调用 mdexsists 函数。此时，触发外存管理涉及 open 文件的相关的函数与“表的创建”语句基本一致。



区别在于会在调用 PathNameOpenFile 时传入不同的文件打开模式，最终在 openFlagsToCreateFileFlags 处判断，从而打开文件的第一个 fork，并进行其他操作。

```
bool
mdexists(SMgrRelation reln, ForkNumber forkNum)
{
    ...
    return (mdopenfork(reln, forkNum, EXTENSION_RETURN_NULL) != NULL);
}
```

```
static MdfdVec *
mdopenfork(SMgrRelation reln, ForkNumber forknum, int behavior)
{
    MdfdVec    *mdfd;
    char       *path;
    File        fd;
    ...
    path = relpath(reln->smgr_rnode, forknum);

    fd = PathNameOpenFile(path, O_RDWR | PG_BINARY);
    ...
}
```

```
static int
openFlagsToCreateFileFlags(int openFlags)
{
    switch (openFlags & (O_CREAT | O_TRUNC | O_EXCL))
    {
            /* O_EXCL is meaningless without O_CREAT */
        case 0:
        case O_EXCL:    //删除很久之前的表文件操作，会返回的情况
            return OPEN_EXISTING;

        case O_CREAT:
            return OPEN_ALWAYS;

            /* O_EXCL is meaningless without O_CREAT */
        case O_TRUNC:
        case O_TRUNC | O_EXCL:
            return TRUNCATE_EXISTING;

        case O_CREAT | O_TRUNC:
            return CREATE_ALWAYS;

            /* O_TRUNC is meaningless with O_CREAT */
        case O_CREAT | O_EXCL:
        case O_CREAT | O_TRUNC | O_EXCL:
            return CREATE_NEW;  //创建表，会返回的情况
    }

    /* will never get here */
    return 0;
}
```

四. 感想与总结

1. 心得感想

宁然：本次大作业我主要负责 b 树索引相关部分逻辑的解析。通过本次大作业，我对 postgresql 这个数据库系统有了一个大体的认识。同时，我对其中和 b 树索引有关的部分有了较为深入的理解。这主要体现在我通过对 b 树索引插入、删除、查询、创建等方面代码的认真阅读，理解了这些 sql 语句的底层执行逻辑，理解了其中的一些重要算法的执行逻辑，比如 b 索引树遍历、创建等算法。这些知识是我之前所不曾接触过的，所以对我而言

也是学到了很多新的内容。同时我感觉在本次大作业上花费的精力算是中等偏上的级别（肯定不如编译实验这种花费的精力大，但是会比安卓大作业这种花费的时间更多），因为面对如此庞大（因为好奇，上网查了下据说源码规模达到了百万行的级别）的由 c 语言编写的系统，我刚接触时肯定是一头雾水，不知从何做起，而且经常出现“想读懂一个函数的时候，发现其中调用了很多自己不懂的其它函数，以及涉及到了很多自己不懂的数据结构”的情况，这个慢慢地把这一头雾水“擦拭”下去的过程于我而言还是有着一定的挑战性的，当然，与痛苦作伴的，无疑是收获与提升，这是知识、心理素质、自学能力（自学能力的提升在我看来非常非常的宝贵）甚至团队协作能力等多方位的提升。总而言之，虽然 postgresql 的源码规模实在是过于庞大，以至于还有很多可以解析的内容，但是本次大作业最后的完成效果也算是达到了我的预期，算是圆满完成了本次大作业。

潘卓然：在内存部分的分析工作中，我从内存上下文开始，先逐步理清各个模块的静态数据结构，严格界定各个内存模块的使用场景，对原本繁杂的内存模块有了较为清晰的了解；后结合 SQL 语句，动态监测各个模块在指令生命周期中的调用关系和数据流向；最后我回归内存模块的最底层结构：页和块，详细地描述了其结构和指令对其上数据的操作效果，真正做到了由面到点，由浅入深。在分析过程中，我也结合操作系统的基础知识，认识到 DBMS 和操作系统在内存结构上的异同。这次难得的 PostgreSQL 源码解析过程，使我对 DBMS 有了更加深入的了解，也提升了我的系统能力、代码阅读能力、团队合作能力和表达能力等。

张博豪：最后一节课上，老师说，时间过得真快啊，仿佛上节课我们还在讲数据库概论，今天就要结课了。深以为然，加之我对数据库本有的浓厚兴趣，全身心投入其中，更是觉得美好的时光总是那么短暂。一学期的课程结束了，PostgreSQL 源码分析的任务也将收尾，回顾和老师，队友一起上课，讨论的时光，我深深感受到被团队包裹的力量感，在这个积极向上的集体中，我有幸和大家一起蜕变，得到能力的提升。回到最初选择 PostgreSQL 任务的那刻，来审视现在的自己是否诚保初心，实现了最初的愿景。首先我确实发挥了在数据结构方面的擅长：在学习 PG 源码过程当中，大量复杂的数据结构，继承关系，函数调用并没有成为拦路虎，我用已有的数据结构知识趁热打铁，从中学习 postgreSQL 的设计思维，这样的正反馈我认为也在一定程度上加深了我对 PostgreSQL 数据库的浓厚兴趣。让我感触最深的一点是：编码过程中对于数据结构的处理和组织十分关键，稍有不慎就会因为疏忽导致数据不能同步，这对之后的访问和修改操作都是潜在的隐患，而这里的“同步”和 PG 系统内存管理确保文件数据一致的“同步”相比，小巫见大巫，起初我在这方

面的直觉也有了更深刻的认识。期间，我的算法能力也如愿以偿获得了提高，最终在学习PG 实现动态哈希的过程中，掌握了哈希处理的精妙算法，高位掩码和低位掩码互相配合，能够保证查找过程中迅速找到数据所在 bucket，增，删，改的操作也类似，更加方便。而高低位掩码的好处却又不止于此，在新增节点实现动态哈希的过程中，由于 old_bucket 中存放着两类元素：① hash 值后 k 位值为 old_bucket;② hash 值后 k 位值为 old_bucket + 2^(k - 1)的那些元素 ，所以只需要裂旧有的一个桶即可，大大提高效率。总结来看，通过本学期 PG 的源码分析，无论是学习能力方面亦或是团队合作方面，我都取得了很大进步，由衷推荐学弟学妹们继续学习 PG 源码。

李嘉鹏：外存管理部分较其他部分解析起来更加细碎，所以在整个解析过程中，如果抓住几个脉络线索，可以把看似凌乱的各个部分统一起来，形成更有条理，更有层次化的理解。我主要从数据管理与数据交互两大板块来梳理了整个外存管理部分的职责与关系。数据管理方面，给我印象深的是空间换时间思想。通过维护的两个辅助表 FSM 和 VM，存储并管理数据的关键信息，将在关键时候极大的提高操作效率和性能。这也让我联想到，我们在解析的过程中，随时记录解析过程，将会对我们后续完整性地回顾整个解析过程，系统性地梳理总结整个体系模块，有很大帮助。数据交互方面，给我印象最深的是，外存管理不同层次间映射、抽象、封装的精巧架构设计。明明操作系统已经有了一套完整的文件系统，可是 postgreSQL 的 DBMS 为什么还要搞一套外存管理的模块呢？DBMS 的关键特性就是对大批量数据进行管理。pg DBMS 通过对 OS 的文件系统进行层层封装，可以更好地适应 DBMS 本身管理数据，频繁与数据交互的特性，完备操作系统的文件系统的功能。同时，这也可以让 DBMS 不依赖于底层系统实现，更好的在多种平台上运行，具有更好的跨平台性质。

2. 建议

- 或许助教学长可以定期发放更为具体的解析要求（比如具体到：前三周完成对 btinsert 函数的解析，然后三周完成对 btbuild 函数的解析…）？这样能给大作业一个更为确定的工作量，而且明确需求相比让同学们自己探索或许更能减轻同学们的负担？
- 在后续的解析工作中，可以向同学们提供我们的文档，先让同学们选出其中我们描述得模糊或有欠缺的地方，使大家能够找到自己感兴趣的部分进行深挖，逐步

完善整个源码解析的课程实验体系。

- 可以引导同学们参加开源社区，PostgreSQL 是一个活跃的开源项目，有着庞大的开发者社区。参与社区讨论、邮件列表或开发者会议，与其他开发者交流，可以获取更多关于 PostgreSQL 源码的洞见和经验。可以考虑让同学们参与逐步实践与调试：尝试修改和扩展 PostgreSQL 源码，运行自己的代码，并通过调试来验证和理解代码的执行过程，这样可以更好地加深对源码的理解。

3. 总结

通过本次大作业，我们都获得了或同或异的专业知识的收获（比如 B 树索引底层代码逻辑及重要算法原理，内存各组件的功能差异和在底层结构上的高度统一，内存存储架构与 SMGR 协调内外存交互，外存管理不同层次间映射、抽象、封装的精巧架构设计等）。同时除了专业知识之外，我们的自学能力、团队协作能力、沟通能力和表达能力还得到了很大的锻炼。因此，我们都觉得我们的大作业做下来很有意义。