第 8 章 表的页面存储结构

8.1 实验介绍

本实验将介绍和实践 openGauss 数据库中表的页面存储结构，即数据在 openGauss 数据库中是以何种方式被存储和组织的。在 openGauss 中，每个表的存储文件都被分成若干个页面，每个页面存储一定量的数据。每个页面的开头都有一个 PageHeaderData 结构体，该结构体包含了关于页面的元数据，例如页面的版本号，检查和标志等。随后的数据存储在页面的主体中，这些数据被存储为一个个元组（tuple），每个元组都有一个对应的 ItemIdData 结构体。该结构体包含了元组的元数据，例如该元组的大小，是否被删除等。每个元组的实际数据存储在 HeapTupleHeaderData 结构体中，该结构体包含了关于该元组的元数据，例如元组的版本号，多元组事务 ID 等。openGauss 表的页面存储结构是一种关系数据的高效物理存储方式，它通过组织元数据和实际数据来实现快速存储和读取。

本实验将首先介绍 openGauss 中的数据文件和堆表存储结构以及元组更新和删除过程；接着通过 pageinspect 插件对表的页面结构进行具体实验与分析。同时，在本实验中，还将建立 railway 示例数据库，用于后续的实验操作。

8.2 实验目的

1. 理解 openGauss 数据文件组织与堆表存储结构、元组更新和删除过程。
2. 了解与堆表数据页面结构相关的重要结构体及其源代码。
3. 掌握使用 pageinspect 插件分析表的页面结构。
4. 掌握实验示例数据库 railway 的表模式、外键约束与数据构成。
5. 理解关系数据库中关系表的物理存储结构。

8.3 实验原理

8.3.1 openGauss 数据文件

openGauss 数据表数据物理存储在非易失性存储设备上面（磁盘、固态硬盘等）。数据表中的数据存储在 N 个数据文件中，每个数据文件有 N 个页面（Page）（大小默认为 8K，可在编译安装时指定）组成。页面为 openGauss 的最小存取单元。

数据文件（堆表、索引）内部被划分为固定长度的页面（page），或者叫区块（block），大小默认为 8192 字节（8KB）。每个文件中的页从 0 开始按顺序编号，这些数字称为区块号。如果数据文件已填满，openGauss 就通过在文件末尾追加一个新的空页来增加文件长度。如图 8.1 所示。

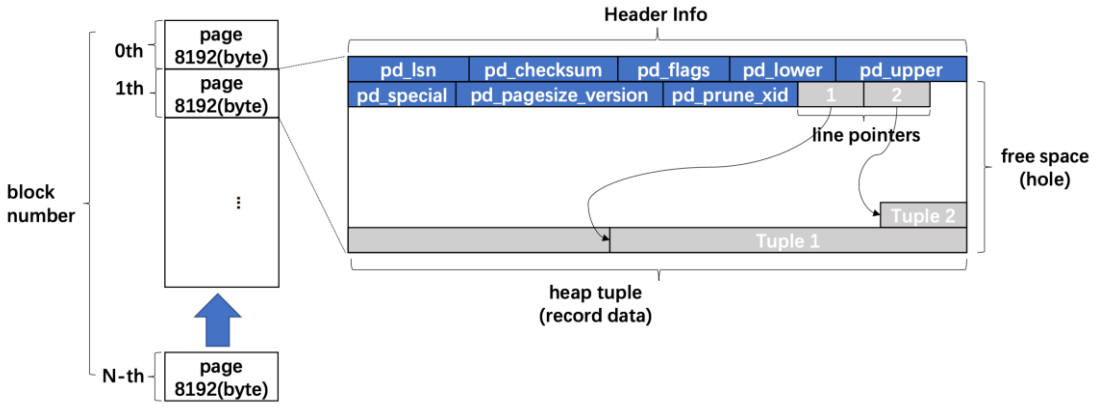

图 8.1 openGauss 数据文件示意图

8.3.2 堆表存储结构

openGauss 的行存储采用堆表存储的方式。所谓堆表，是指元组无序存储，数据按照“先来后到”的方式存储在页面中的空闲位置。由于整体行存储格式默认的介质管理器是磁盘文件系统，因此采用了和文件系统类似的段页式设计，最小 I/O 单元为一个页面，这样可以在大多数场景下获得比较好的 I/O 性能和较低的 I/O 开销。一个堆表页面默认大小为 8KB，其结构如图 8.2 所示。

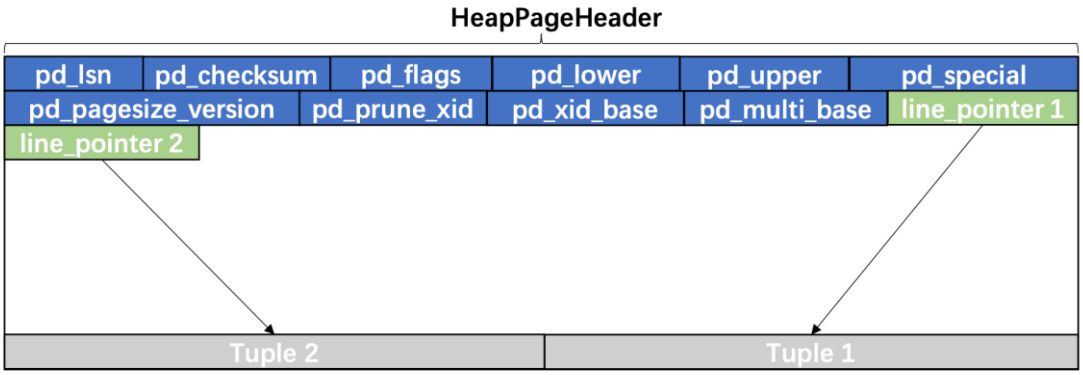

图 8.2 堆表页面示意图

由图 8.2 可见，一个堆表页面分为 4 个部分：

- PageHeaderData/HeapPageHeaderData 页面头/heap 页面头数据（24 字节）。
- line_pointer：行指针数组（指向了元组存储的位置）。
- free space：空闲空间。
- tuple 数据：行数据。

在一个堆表页面中，页面头部分对应 HeapPageHeaderData 结构体。其中，pd_multi_base 以及之前的部分对应定长成员，存储了整个页面的重要元信息；pd_multi_base 之后的部分对应元组（行）指针变长数组，其每个数组成员存储了页面中从后往前的、每个元组的起始偏移和元组长度。真正的元组内容从页面尾部开始插入，向页面头部扩展；相应的，记录每条元组的元组指针从页面头定长成员之后插入，往页面尾部扩展；整个页面中间形成一个空洞，供后续插入的元组和元组指针使用。

8.3.3 元组更新和删除过程

1. 元组更新过程。

   如图 8.3 所示，图中左边是一条新插入的元组，可以看到元组是 xid=100 的事务插入的，没有进行更新，所以 t_xmax=0，同时 t_ctid 指向自己，0 号页面的第 1 号元组。右图是发生 xid=101 的事务更新该元组后的状态，更新在数据库里相当于插入一条新元组，原来的元组的 t_xmax 变为了更新这条事务的 xid=101，同时 t_ctid 指针指向了新插入的元组(0, 2)，0 号页面第 2 号元组，第 2 号元组的 t_xmin=101（插入该元组的 xid），t_ctid=(0, 2)，没有发生更新，指向自己。

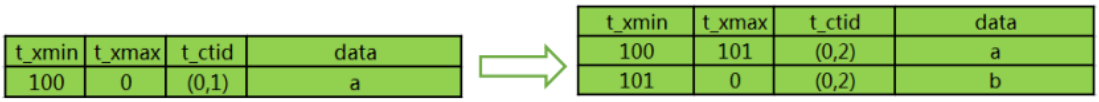

图 8.3 元组更新过程示意图

2. 删除过程

   如图 8.4 所示，代表该元组被 xid=102 的事务删除，将 t_xmax 设置为删除事务的 xid，t_ctid 指向自己。

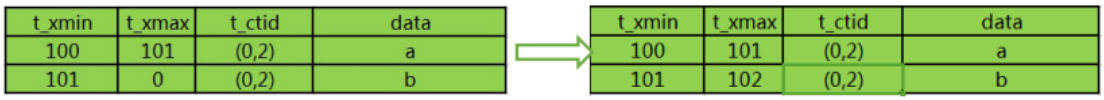

图 8.4 元组删除过程示意图

3. 元组数据（tuple data）有以下特性
   - tuple data 有多个列（属性）组成。
   - 每个属性的长度分定长与变长。
   - 每个属性有不同的字节对齐。
   - 每个属性有不同的存储策略。
   - tuple data 中不存储 NULL 值。
   - 起始位置 8 字节对齐。

   此外，大小超过约 2KB（8KB 的四分之一）的堆元组会使用一种称为 TOAST（The Oversized-Attribute Storage Technique，超大属性存储技术）的方法来存储与管理。TOAST 将自动将其存储在数据库的单独专用区域中，并仅在原始数据页中存储对其的引用。这有助于保持原始数据页的大小在可控范围内，提高性能。

8.3.4 pageinspect 插件及其函数介绍

pageinspect 插件是 openGauss 中继承自 PostgreSQL 的一个 contrib 模块，提供对数据库页内容的低级访问，能够用来检查表或索引中页面的内容，对于诊断损坏、理解存储格式和调试存储层非常有用。pageinspect 模块提供了若干函数，使用这些函数可以检索有关页面结构的信息，包括头信息、元组布局和单个字段的内容。pageinspect 插件为理解 PostgreSQL 如何存储数据提供了一个强大的工具。

pageinspect 插件提供函数对 openGauss 数据库存储页面的结构进行剖析。本实验用到的 pageinspect 插件函数为：

- get_raw_page (relname text, blkno int)：返回 bytea。

返回关系表 relname 的第 blkno 个页面的一个副本。
bytea 是 openGauss 中用于存储二进制数据的数据类型。它可以用于存储任何需要存储在数据库中的二进制数据。

- page_header(page bytea) 返回 record。
  返回一个页面的页头，参数 page 应该是用 get_raw_page 函数返回的页面副本。返回值 record 的各列对应于 PageHeaderData 结构的字段。
- heap_page_items(page bytea) 返回 setof record
  显示一个堆（heap）页面上所有的行指针，参数 page 应该是用 get_raw_page 函数返回的页面副本。返回值是页面内的项（行/item）指针(ItemIdData)以及对页面元组头部结构 HeapTupleHeaderData 的详细信息。

8.3.5 相关结构体

本实验涉及到的几个重要的结构体如下：

1. 结构体 PageHeaderData 及其指针 PageHeader
   PageHeaderData 结构体表示数据库文件中一个数据页面的页头。页头是数据库文件中数据页的第一部分，包含关于该页的信息，包括其大小，该页上可用的空闲空间以及存储在页上的项目数。PageHeaderData 结构提供了一种访问和操纵存储在页头中的信息的方法。
   PageHeaderData 结构体的字段说明：
   - pd_lsn：记录最后一次对页面修改的 xlog 日志记录 id
   - pd_checksum：页面的检查和
   - pd_flags：标志位 flag.
   - pd_lower：空闲空间的起始处（距离页头）
   - pd_upper：空闲空间的结尾处（距离页头）
   - pd_special：页面预留空间的开始处（距离页头）
   - pd_pagesize_version：页面大小及版本号
   - pd_prune_xid：页面清理辅助事务 id（32 位），通常为该页面内现存最老的删除或更新操作的事务 id，用于判断是否要触发页面级空闲空间整理。
   - pd_linp：行（元组/项）指针数组。

   【源码】src/include/storage/buf/bufpage.h：

```
/*
 * disk page organization
 *
 * space management information generic to any page
 *
 *      pd_lsn      - identifies xlog record for last change to this page.
 *      pd_checksum - page checksum, if set.
 *      pd_flags    - flag bits.
 *      pd_lower    - offset to start of free space.
 *      pd_upper    - offset to end of free space.
 *      pd_special  - offset to start of special space.
 *      pd_pagesize_version - size in bytes and page layout version number.
 *      pd_prune_xid - oldest XID among potentially prunable tuples on page.
```

```
 *
 * The LSN is used by the buffer manager to enforce the basic rule of WAL:
 * "thou shalt write xlog before data".  A dirty buffer cannot be dumped
 * to disk until xlog has been flushed at least as far as the page's LSN.
 *
 * pd_checksum stores the page checksum, if it has been set for this page;
 * zero is a valid value for a checksum. If a checksum is not in use then
 * we leave the field unset. This will typically mean the field is zero
 * though non-zero values may also be present if databases have been
 * pg_upgraded from releases prior to 9.3, when the same byte offset was
 * used to store the current timelineid when the page was last updated.
 * Note that there is no indication on a page as to whether the checksum
 * is valid or not, a deliberate design choice which avoids the problem
 * of relying on the page contents to decide whether to verify it. Hence
 * there are no flag bits relating to checksums.
 *
 * pd_prune_xid is a hint field that helps determine whether pruning will be
 * useful.  It is currently unused in index pages.
 *
 * The page version number and page size are packed together into a single
 * uint16 field.  This is for historical reasons: before PostgreSQL 7.3,
 * there was no concept of a page version number, and doing it this way
 * lets us pretend that pre-7.3 databases have page version number zero.
 * We constrain page sizes to be multiples of 256, leaving the low eight
 * bits available for a version number.
 *
 * Minimum possible page size is perhaps 64B to fit page header, opaque space
 * and a minimal tuple; of course, in reality you want it much bigger, so
 * the constraint on pagesize mod 256 is not an important restriction.
 * On the high end, we can only support pages up to 32KB because lp_off/lp_len
 * are 15 bits.
 */
typedef struct {
    /* 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;
    ShortTransactionId pd_prune_xid;          /* oldest prunable XID, or zero if none
*/
```

```
    ItemIdData pd_linp[FLEXIBLE_ARRAY_MEMBER]; /* beginning of line pointer array */
} PageHeaderData;
```

PageHeader 被定义为 PageHeaderData 结构体的指针。

```
typedef PageHeaderData* PageHeader;
```

2. 结构体 HeapPageHeaderData 及其指针 HeapPageHeader

HeapPageHeaderData 结构体用于存储堆页的页面头信息。HeapPageHeaderData 的字段兼容 PageHeaderData。HeapPageHeaderData 的特有字段包括：

- pd_xid_base - 页面上事务 ID 的基础值。
- pd_multi_base - 页面上多重事务 ID 的基础值。

【源码】src/include/storage/buf/bufpage.h：

```
/*
 * HeapPageHeaderData -- data that stored at the begin of each new version heap page.
 *      pd_xid_base - base value for transaction IDs on page
 *      pd_multi_base - base value for multixact IDs on page
 *
 */
typedef struct {
    /* 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;
    ShortTransactionId pd_prune_xid;           /* oldest prunable XID, or zero if none */
    TransactionId pd_xid_base;                /* base value for transaction IDs on page */
    TransactionId pd_multi_base;               /* base value for multixact IDs on page */
    ItemIdData pd_linp[FLEXIBLE_ARRAY_MEMBER]; /* beginning of line pointer array */
} HeapPageHeaderData;
```

HeapPageHeader 被定义为 HeapPageHeaderData 结构体的指针。

```
typedef HeapPageHeaderData* HeapPageHeader;
```

3. 结构体 ItemIdData 及其指针 ItemId

ItemIdData 结构体是堆表的数据结构，用于存储堆表中的每个元素（item）/元组/行的信息。该结构体实际上是一个无符号整型（unsigned）按字段（域）的划分，占 4 字节 32 位。ItemIdData 包含以下字段：

- lp_off：行（元组）在页面中的偏移量（从页头开始），占 15 位（bit）。
- lp_flags：行指针的状态标志位（用于指示元素是否已删除、空闲等），占 2 位。
- lp_len：行（元组）的字节长度，占 15 位。

ItemIdData 行指针结构如图 8.5 所示。

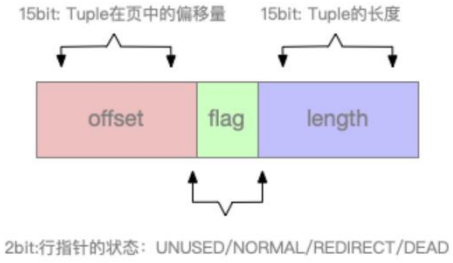

图 8.5 行指针结构 ItemIdData 示意图

【源码】src/include/storage/item/itemid.h：

```
/*
 * An item pointer (also called line pointer) on a buffer page
 *
 * In some cases an item pointer is "in use" but does not have any associated
 * storage on the page.  By convention, lp_len == 0 in every item pointer
 * that does not have storage, independently of its lp_flags state.
 */
typedef struct ItemIdData {
    unsigned lp_off : 15, /* offset to tuple (from start of page) */
        lp_flags : 2,     /* state of item pointer, see below */
        lp_len : 15;      /* byte length of tuple */
} ItemIdData;
```

ItemId 被定义为 ItemIdData 结构体的指针。

```
typedef ItemIdData* ItemId;
```

lp_flags 字段有如下 4 个值：

- LP_UNUSED：该元组未使用（lp_len 应该等于 0）
- LP_NORMAL：该元组已使用（lp_len 应该大于 0）
- LP_REDIRECT：使用 HOT 技术的重定向指针（lp_len 应该等于 0）
- LP_DEAD：死元组（可能占用也可能不占用存储空间）

```
/*
 * lp_flags has these possible states.  An UNUSED line pointer is available
 * for immediate re-use, the other states are not.
 */
#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 */
```

HOT 技术简介：

HOT（Heap Only Tuple）技术是一种数据库优化技术，用于减少不必要的空间和性能开销，并提高并发更新性能。它通过允许多个版本的元组（tuple）共存于同一页面，从而避免了不必要的元组删除和重建的开销。在更新操作时，将原来的元组标记为已删除，并在该页面的空闲空间内创建新版本的元组，使用不同的事务 ID。这样，在读操作时，可以找到没有被删除的元组的最新版本。HOT 技术需要适当的空闲空间来存储新版本的元组，以及更高的页面内搜索开销，因此需要在页面上执行更多的读操作。但是，它可以大大减少页面的版本数，减少不必要的索引空间

与维护开销，并且可以提高并发更新性能。

4. 结构体 HeapTupleHeaderData 和 HeapTupleFields

HeapTupleHeaderData 是表示元组（表中的一行）头部的数据结构。它包含了插入或更新元组的事务 ID、元组的可见性信息以及标志元组状态的信息。其他字段提供了关于元组数据的长度和头部大小的信息。HeapTupleFields 结构体包括 t_xmin、t_xmax 及 t_cid 字段（说明见下面），其嵌入到 HeapTupleHeaderData 结构体中作为其开头部分。

元组由元组头部（tuple header）加上元组数据（tuple data）组成，元组头部是由 23 字节固定大小的前缀和可选的 NULL bitmap 构成的。如下图 8.6 所示。

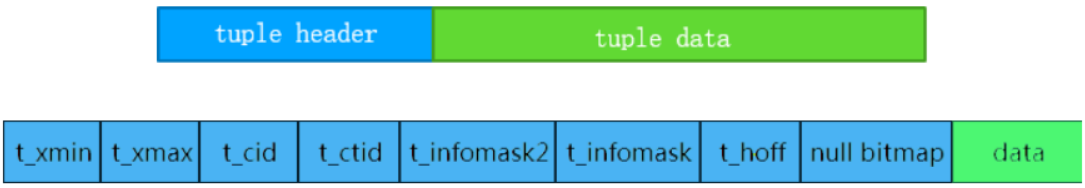

图 8.6 元组头部组成示意图

结构体 HeapTupleHeaderData 和 HeapTupleFields 表示的元组头部的字段说明如下：

- t_xmin：代表插入此元组的事务 id
  typedef uint32 ShortTransactionId; 参见 src/include/c.h
- t_xmax：代表删除（更新）或锁定此元组的事务 id，如果该元组插入后未进行更新或者删除，则 t_xmax=0；（更新被视为先删除再插入）
- t_cid：事务中的插入或删除命令 id
  typedef uint32 CommandId; 参见 src/include/c.h
- t_ctid：保存着指向自身或者新元组的元组标识（TID）。TID 由两个数字组成，第一个数字代表页面号（物理块号），第二个数字代表元组号。在元组更新后 tid 指向新版本的元组，否则指向自己，这样其实就形成了新旧元组之间的“元组链”，这个链在元组查找和定位上起着重要作用。
  关于 TID 的结构，参见：
  src/include/storage/item/itemptr.h 中的 ItemPointerData 结构
  src/include/storage/buf/block.h 中的 BlockIdData 结构
  src/include/storage/off.h 中的 typedef uint16 OffsetNumber;
- t_infomask2：属性数量、各种标志位
- t_infomask：标志位，记录各种信息，如是否存在 NULL 列，是否有变长列，是否有 OID 列等。如果有允许为空的列，则存在 NULL bitmap，可以通过 t_infomask 判断（通过位运算 t_infomask & 0x0001 判断），bitmap 的大小与列个数有关。
  参见 src/include/access/htup.h
- t_hoff：记录元组头部（tuple header）的大小，包含 NULL bitmap 和 padding。元组头部后会有 padding，使元组头部的大小为 8 的整数倍。
  对于一个堆表页面中的一条具体元组，有一个全局唯一的逻辑地址，即元组头部的 t_ctid，其由元组所在的页面号和页面内元组指针数组下标组成。该逻辑地址对应的物理地址，则由 t_ctid 和对应的元组指针成员共同给出。通过页面、对应元组指针数组成员、页面内偏移和元组长度的访问顺序，就可以获取到一条元组的完整内容。

【源码】src/include/access/htup.h：

```
/*
 * Heap tuple header.  To avoid wasting space, the fields should be
 * laid out in such a way as to avoid structure padding.
 *
 * Datums of composite types (row types) share the same general structure
 * as on-disk tuples, so that the same routines can be used to build and
 * examine them.  However the requirements are slightly different: a Datum
 * does not need any transaction visibility information, and it does need
 * a length word and some embedded type information.  We can achieve this
 * by overlaying the xmin/cmin/xmax/cmax/xvac fields of a heap tuple
 * with the fields needed in the Datum case.  Typically, all tuples built
 * in-memory will be initialized with the Datum fields; but when a tuple is
 * about to be inserted in a table, the transaction fields will be filled,
 * overwriting the datum fields.
 *
 * The overall structure of a heap tuple looks like:
 *          fixed fields (HeapTupleHeaderData struct)
 *          nulls bitmap (if HEAP_HASNULL is set in t_infomask)
 *          alignment padding (as needed to make user data MAXALIGN'd)
 *          object ID (if HEAP_HASOID is set in t_infomask)
 *          user data fields
 *
 * We store five "virtual" fields Xmin, Cmin, Xmax, Cmax, and Xvac in three
 * physical fields.  Xmin and Xmax are always really stored, but Cmin, Cmax
 * and Xvac share a field.  This works because we know that Cmin and Cmax
 * are only interesting for the lifetime of the inserting and deleting
 * transaction respectively.  If a tuple is inserted and deleted in the same
 * transaction, we store a "combo" command id that can be mapped to the real
 * cmin and cmax, but only by use of local state within the originating
 * backend.  See combocid.c for more details.  Meanwhile, Xvac is only set by
 * old-style VACUUM FULL, which does not have any command sub-structure and so
 * does not need either Cmin or Cmax.  (This requires that old-style VACUUM
 * FULL never try to move a tuple whose Cmin or Cmax is still interesting,
 * ie, an insert-in-progress or delete-in-progress tuple.)
 *
 * A word about t_ctid: whenever a new tuple is stored on disk, its t_ctid
 * is initialized with its own TID (location).  If the tuple is ever updated,
 * its t_ctid is changed to point to the replacement version of the tuple.
 * Thus, a tuple is the latest version of its row iff XMAX is invalid or
 * t_ctid points to itself (in which case, if XMAX is valid, the tuple is
 * either locked or deleted).  One can follow the chain of t_ctid links
 * to find the newest version of the row.  Beware however that VACUUM might
 * erase the pointed-to (newer) tuple before erasing the pointing (older)
```

```
 * tuple.  Hence, when following a t_ctid link, it is necessary to check
 * to see if the referenced slot is empty or contains an unrelated tuple.
 * Check that the referenced tuple has XMIN equal to the referencing tuple's
 * XMAX to verify that it is actually the descendant version and not an
 * unrelated tuple stored into a slot recently freed by VACUUM.  If either
 * check fails, one may assume that there is no live descendant version.
 *
 * Following the fixed header fields, the nulls bitmap is stored (beginning
 * at t_bits).  The bitmap is *not* stored if t_infomask shows that there
 * are no nulls in the tuple.  If an OID field is present (as indicated by
 * t_infomask), then it is stored just before the user data, which begins at
 * the offset shown by t_hoff.  Note that t_hoff must be a multiple of
 * MAXALIGN.
 */

typedef struct HeapTupleFields {
    ShortTransactionId t_xmin; /* inserting xact ID */
    ShortTransactionId t_xmax; /* deleting or locking xact ID */

    union {
        CommandId t_cid;           /* inserting or deleting command ID, or both */
        ShortTransactionId t_xvac; /* old-style VACUUM FULL xact ID */
    } t_field3;
} HeapTupleFields;

typedef struct DatumTupleFields {
    int32 datum_len_; /* varlena header (do not touch directly!) */

    int32 datum_typmod; /* -1, or identifier of a record type */

    Oid datum_typeid; /* composite type OID, or RECORDOID */

    /*
     * Note: field ordering is chosen with thought that Oid might someday
     * widen to 64 bits.
     */
} DatumTupleFields;

typedef struct HeapTupleHeaderData {
    union {
        HeapTupleFields t_heap;
        DatumTupleFields t_datum;
    } t_choice;
```

```
    ItemPointerData t_ctid; /* current TID of this or newer tuple */

   /* Fields below here must match MinimalTupleData! */

    uint16 t_infomask2; /* number of attributes + various flags */

    uint16 t_infomask; /* various flag bits, see below */

    uint8 t_hoff; /* sizeof header incl. bitmap, padding */

    /* ^ - 23 bytes - ^ */

    bits8 t_bits[FLEXIBLE_ARRAY_MEMBER]; /* bitmap of NULLs -- VARIABLE LENGTH */

    /* MORE DATA FOLLOWS AT END OF STRUCT */
} HeapTupleHeaderData;
typedef HeapTupleHeaderData* HeapTupleHeader;
```

t_infomask 字段不同标志位的含义如下：

```
/*
 * information stored in t_infomask:
 */
#define HEAP_HASNULL 0x0001          /* has null attribute(s) */
#define HEAP_HASVARWIDTH 0x0002      /* has variable-width attribute(s) */
#define HEAP_HASEXTERNAL 0x0004      /* has external stored attribute(s) */
#define HEAP_HASOID 0x0008           /* has an object-id field */
#define HEAP_COMPRESSED 0x0010       /* has compressed data */
#define HEAP_COMBOCID 0x0020         /* t_cid is a combo cid */
#define HEAP_XMAX_EXCL_LOCK 0x0040   /* xmax is exclusive locker */
#define HEAP_XMAX_SHARED_LOCK 0x0080 /* xmax is shared locker */
…
```

t_infomask2 字段不同标志位的含义如下：

```
/*
 * information stored in t_infomask2:
 */
#define HEAP_NATTS_MASK 0x07FF /* 11 bits for number of attributes */
#define HEAP_XMAX_LOCK_ONLY 0x0800 /* xmax, if valid, is only a locker */
#define HEAP_KEYS_UPDATED 0x1000 /* tuple was updated and key cols modified, or tuple
deleted */
#define HEAP_HAS_REDIS_COLUMNS 0x2000 /* tuple has hidden columns added by redis */
#define HEAP_HOT_UPDATED 0x4000 /* tuple was HOT-updated */
#define HEAP_ONLY_TUPLE 0x8000  /* this is heap-only tuple */
```

5. 结构体 HeapTupleData 及其指针 HeapTuple
   结构体 HeapTupleData 是表示元组在内存中的数据结构，其中保存了指向一个元组头和元组数据的指针（HeapTupleHeader）。

结构体 HeapTupleData 字段说明：

- t_len：元组字节长度，包括元组头和元组数据。
- t_self：标识当前元组在页面内所处位置，即页面的块号（block number）和页面内内的偏移量（offset）。
- t_tableOid：当前元组所在的表的 OID。
- t_data：指向一个元组数据的指针（HeapTupleHeader，即元组头+元组数据）

```
/*
 * HeapTupleData is an in-memory data structure that points to a tuple.
 *
 * There are several ways in which this data structure is used:
 *
 * * Pointer to a tuple in a disk buffer: t_data points directly into the
 *     buffer (which the code had better be holding a pin on, but this is not
 *     reflected in HeapTupleData itself).
 *
 * * Pointer to nothing: t_data is NULL.  This is used as a failure indication
 *     in some functions.
 *
 * * Part of a palloc'd tuple: the HeapTupleData itself and the tuple
 *     form a single palloc'd chunk.  t_data points to the memory location
 *     immediately following the HeapTupleData struct (at offset HEAPTUPLESIZE).
 *     This is the output format of heap_form_tuple and related routines.
 *
 * * Separately allocated tuple: t_data points to a palloc'd chunk that
 *     is not adjacent to the HeapTupleData.(This case is deprecated since
 *     it's difficult to tell apart from case #1.  It should be used only in
 *     limited contexts where the code knows that case #1 will never apply.)
 *
 * * Separately allocated minimal tuple: t_data points MINIMAL_TUPLE_OFFSET
 *     bytes before the start of a MinimalTuple. As with the previous case,
 *     this can't be told apart from case #1 by inspection; code setting up
 *     or destroying this representation has to know what it's doing.
 *
 * t_len should always be valid, except in the pointer-to-nothing case.
 * t_self and t_tableOid should be valid if the HeapTupleData points to
 * a disk buffer, or if it represents a copy of a tuple on disk.  They
 * should be explicitly set invalid in manufactured tuples.
 */
typedef struct HeapTupleData {
    uint32 t_len;           /* length of *t_data */
    uint1 tupTableType = HEAP_TUPLE;
    uint1 tupInfo;
    int2   t_bucketId;
    ItemPointerData t_self; /* SelfItemPointer */
```

```
        Oid t_tableOid;         /* table the tuple came from */
        TransactionId t_xid_base;
        TransactionId t_multi_base;
    #ifdef PGXC
        uint32 t_xc_node_id; /* Data node the tuple came from */
    #endif
        HeapTupleHeader t_data; /* -> tuple header and data */
} HeapTupleData;
```

HeapTuple 被定义为 HeapTupleData 结构体的指针。

```
typedef HeapTupleData* HeapTuple;
```

8.4 实验步骤

8.4.1 安装 pageinspect 插件

pageinspect 是 openGauss 继承于 PostgreSQL 的一款扩展插件，其功能是提供函数让用户从低层次观察数据库页面的内容。

1. 以 dblab 用户登录云主机。
2. 进入到 pageinspect 目录下，准备进行编译安装。

   进入到 pageinspect 目录：

   ```
   [dblab@eduog ~] cd opengauss-compile/openGauss-server-v3.0.0/contrib/pageinspect
   ```

   查看 pageinspect 目录下文件内容：

   ```
   [dblab@eduog pageinspect]$ ll
   total 80K
   -rwx------ 1 dblab root  17K Apr  1  2022 btreefuncs.cpp
   -rwx------ 1 dblab root 1.6K Apr  1  2022 fsmfuncs.cpp
   -rwx------ 1 dblab root 9.2K Apr  1  2022 ginfuncs.cpp
   -rwx------ 1 dblab root 6.7K Apr  1  2022 heapfuncs.cpp
   -rwx------ 1 dblab root  458 Apr  1  2022 Makefile
   -rwx------ 1 dblab root 3.5K Apr  1  2022 pageinspect--1.0.sql
   -rwx------ 1 dblab root  173 Apr  1  2022 pageinspect.control
   -rwx------ 1 dblab root 1.3K Apr  1  2022 pageinspect--unpackaged--1.0.sql
   -rwx------ 1 dblab root  18K Apr  1  2022 rawpage.cpp
   ```

3. 编译 pageinspect 插件。

   执行 make，编译 pageinspect，编译过程中出现的 warning 并不影响编译和安装。

   ```
   [dblab@eduog pageinspect]$ make
   ```

```
openeuler_aarch64/zlib1.2.11/comm/include -I/home/dblab/opengauss-compile/openGauss-server-v3.0.0/../binarylibs-v3.0.0/dependency/
ch64/lz4/comm/include -I/home/dblab/opengauss-compile/openGauss-server-v3.0.0/../binarylibs-v3.0.0/dependency/openeuler_aarch64/li
clude -I/home/dblab/opengauss-compile/openGauss-server-v3.0.0/../binarylibs-v3.0.0/component/openeuler_aarch64/dcf/include -I/home
uss-compile/openGauss-server-v3.0.0/../binarylibs-v3.0.0/dependency/openeuler_aarch64/zstd/include  -c -o rawpage.o rawpage.cpp
rawpage.cpp: In function 'Datum page_header(FunctionCallInfo)':
rawpage.cpp:203:23: warning: comparison between signed and unsigned integer expressions [-Wsign-compare]
     if (raw_page_size < GetPageHeaderSize(page))
rawpage.cpp: In function 'Datum page_compress_meta(FunctionCallInfo)':
rawpage.cpp:296:27: warning: comparison between signed and unsigned integer expressions [-Wsign-compare]
         for (int i = 0; i < blknum; ++i) {
                         ~~^~~~~~~
rawpage.cpp: In function 'void parse_compress_meta(StringInfo, char*, Relation)':
rawpage.cpp:434:11: warning: unused variable 'current' [-Wunused-variable]
     char* current = start;
           ^~~~~~~
/home/dblab/opengauss-compile/openGauss-server-v3.0.0/../binarylibs-v3.0.0/buildtools/openeuler_aarch64/gcc7.3/gcc/bin/g++ -std=c+
X_USE_CXX11_ABI=0 -fsigned-char -DSTREAMPLAN -DPGXC -march=armv8-a+crc -O0 -Wall -Wpointer-arith -Wno-write-strings -fnon-call-exc
common -freg-struct-return -pipe -Wendif-labels -Wmissing-format-attribute -Wformat-security -fno-strict-aliasing -fwrapv -g -DENA
fno-aggressive-loop-optimizations -Wno-attributes -fno-omit-frame-pointer -fno-expensive-optimizations -Wno-unused-but-set-variabl
```

图 8.7 编译 pageinspect 插件

4. 安装 pageinspect 插件。

   执行 make install，安装 pageinspect 插件：

```
[dblab@eduog pageinspect]$ make install
```

```
[dblab@eduog pageinspect]$ make install
/usr/bin/mkdir -p '/home/dblab/opengauss-compile/openGauss-server-v3.0.0/dest/lib/postgresql'
/usr/bin/mkdir -p '/home/dblab/opengauss-compile/openGauss-server-v3.0.0/dest/share/postgresql/extension'
/usr/bin/mkdir -p '/home/dblab/opengauss-compile/openGauss-server-v3.0.0/dest/share/postgresql/extension'
/bin/sh ../../config/install-sh -c -m 755  pageinspect.so '/home/dblab/opengauss-compile/openGauss-server-v3.0.0/dest/lib/postgresql/pageinspec
t.so'
/bin/sh ../../config/install-sh -c -m 644 ./pageinspect.control '/home/dblab/opengauss-compile/openGauss-server-v3.0.0/dest/share/postgresql/ex
tension/'
/bin/sh ../../config/install-sh -c -m 644 ./pageinspect--1.0.sql ./pageinspect--unpackaged--1.0.sql  '/home/dblab/opengauss-compile/openGauss-s
erver-v3.0.0/dest/share/postgresql/extension/'
```

图 8.8 安装 pageinspect 插件

8.4.2 创建示例表模式并插入示例数据

本实验以高铁售票场景为例，创建数据库及关系表。

1. 创建 railway 数据库，并连接。

   确保数据库服务器处于启动状态。使用 gsql 连接数据库，执行如下命令：

```
[dblab@eduog openGauss-server-v3.0.0]$ gsql postgres -r
gsql ((openGauss 3.0.0 build ) compiled at 2023-01-29 11:22:50 commit 0 last mr  debug)
Non-SSL connection (SSL connection is recommended when requiring high-security)
Type "help" for help.

openGauss=# CREATE DATABASE railway;
CREATE DATABASE
openGauss=# \c railway
Non-SSL connection (SSL connection is recommended when requiring high-security)
You are now connected to database "railway" as user "dblab".
railway=#
```

2. 按照如下 SQL 语句创建表，插入数据，用于后续实验操作。

   共新建 6 张关系表：用户表、车站表、列车车次表、列车停靠车站表、车次运行表、订单表。

   - 用户表：

```
CREATE TABLE users
(
   u_id varchar(20),                -- 用户 id，用于系统登录账户名（主键）
   u_passwd varchar(20),            -- 密码，用于系统登录密码
   u_name varchar(10),              -- 真实姓名
   u_idnum varchar(20),             -- 证件号码
```

```
   u_regtime timestamp,            -- 注册时间
   CONSTRAINT pk_users PRIMARY KEY (u_id)
);
```

- 用户数据:

```
INSERT INTO users VALUES('1','qweasd','张三','123456789','2000-06-23 12:00:00');
INSERT INTO users VALUES('2','qweasd','李四','123456712','2000-06-24 13:00:00');
INSERT INTO users VALUES('3','qweasd','王五','123456754','2000-06-25 14:00:00');
INSERT INTO users VALUES('4','qweasd','赵六','123456709','2000-06-26 15:00:00');
INSERT INTO users VALUES('5','qweasd','小明','123423709','2000-06-27 15:00:00');
INSERT INTO users VALUES('6','qweasd','小李','123423709','2000-06-27 16:00:00');
INSERT INTO users VALUES('7','qweasd','小赵','123423712','2000-06-27 10:00:00');
INSERT INTO users VALUES('8','qweasd','小红','142423709','2000-06-27 11:00:00');
INSERT INTO users VALUES('9','qweasd','小秦','163423709','2000-06-27 19:00:00');
```

- 车站表:

```
CREATE TABLE station
(
  s_name varchar(20),             -- 车站名称（主键）
  s_city varchar(20),             -- 车站所在城市
  CONSTRAINT pk_station PRIMARY KEY (s_name)
);
```

- 车站数据:

```
INSERT INTO station VALUES('北京南','北京市');
INSERT INTO station VALUES('天津','天津市');
INSERT INTO station VALUES('重庆西','重庆市');
INSERT INTO station VALUES('长沙南','长沙市');
INSERT INTO station VALUES('天津西','天津市');
INSERT INTO station VALUES('天津南','天津市');
INSERT INTO station VALUES('福州','福州市');
INSERT INTO station VALUES('沧州西','沧州市');
INSERT INTO station VALUES('郑州东','郑州市');
INSERT INTO station VALUES('石家庄','石家庄市');
INSERT INTO station VALUES('合肥南','合肥市');
INSERT INTO station VALUES('西安北','西安市');
INSERT INTO station VALUES('武汉','武汉市');
INSERT INTO station VALUES('香港西九龙','香港自治区');
INSERT INTO station VALUES('厦门北','厦门市');
INSERT INTO station VALUES('哈尔滨西','哈尔滨市');
INSERT INTO station VALUES('海口','海口市');
```

- 列车车次表:

```
CREATE TABLE train
(
  t_id varchar(10),               -- 车次 id（主键）
  t_dstation varchar(20),         -- 始发站（外键：参照 station(s_name)）
  t_astation varchar(20),         -- 到达站（外键：参照 station(s_name)）
```

```
  t_dtime time,                    -- 出发时间
  t_atime time,                    -- 到达时间
  CONSTRAINT pk_train PRIMARY KEY (t_id)
);
```

- 列车车次表外键约束:

```
ALTER TABLE train ADD CONSTRAINT fk_train_station_departure FOREIGN KEY (t_dstation)
REFERENCES station(s_name);
ALTER TABLE train ADD CONSTRAINT fk_train_station_arrival FOREIGN KEY (t_astation)
REFERENCES station(s_name);
```

- 列车车次数据:

```
INSERT INTO train VALUES('G321','北京南','厦门北','2022-04-29 08:47','2022-04-29
19:58');
INSERT INTO train VALUES('G2608','天津西','北京南','2022-04-30 05:42','2022-04-30
06:22');
INSERT INTO train VALUES('G2002','天津','北京南','2022-04-29 05:58','2022-04-29 06:28');
INSERT INTO train VALUES('G1709','天津西','重庆西','2022-04-29 08:05','2022-04-29
19:54');
INSERT INTO train VALUES('G305','天津西','香港西九龙','2022-04-29 10:57','2022-04-29
21:07');
```

- 列车停靠车站表:

```
CREATE TABLE trainstop
(
  ts_tid varchar(10),              -- 车次 id（外键: 参照 train(t_id)）
  ts_sname varchar(20),            -- 车站名称（外键: 参照 station(s_name)）
  ts_atime time,                   -- 到达时间
  ts_dtime time,                   -- 出发时间
  CONSTRAINT pk_trainstop PRIMARY KEY (ts_tid, ts_sname) -- （主键）
);
```

- 列车停靠表外键约束:

```
ALTER TABLE trainstop ADD CONSTRAINT fk_trainstop_train FOREIGN KEY (ts_tid)
REFERENCES train(t_id);
ALTER TABLE trainstop ADD CONSTRAINT fk_trainstop_station FOREIGN KEY (ts_sname)
REFERENCES station(s_name);
```

- 列车停靠车站数据:

```
INSERT INTO trainstop VALUES('G321','沧州西','2022-04-29 09:54','2022-04-29 09:56');
INSERT INTO trainstop VALUES('G321','天津南','2022-04-29 09:21','2022-04-29 09:30');
INSERT INTO trainstop VALUES('G321','福州','2022-04-29 17:55','2022-04-29 18:02');
INSERT INTO trainstop VALUES('G321','合肥南','2022-04-29 13:31','2022-04-29 13:44');
INSERT INTO trainstop VALUES('G1709','郑州东','2022-04-29 12:08','2022-04-29 12:16');
INSERT INTO trainstop VALUES('G1709','西安北','2022-04-29 14:21','2022-04-29 14:25');
INSERT INTO trainstop VALUES('G305','石家庄','2022-04-29 12:29','2022-04-29 12:32');
INSERT INTO trainstop VALUES('G305','武汉','2022-04-29 15:56','2022-04-29 16:00');
INSERT INTO trainstop VALUES('G305','长沙南','2022-04-29 17:26','2022-04-29 17:30');
```

- 车次运行表:

```
CREATE TABLE trainrun
(
  tr_date date,                    -- 车次日期
  tr_tid  varchar(10),             -- 车次 id（外键：参照 train(t_id)）
  tr_seat1 smallint,               -- 剩余座位数量：一等
  tr_seat2 smallint,               -- 剩余座位数量：二等
  CONSTRAINT pk_trainrun PRIMARY KEY (tr_date, tr_tid) -- （主键）
);
```

- 车次运行表外键约束:

```
ALTER TABLE trainrun ADD CONSTRAINT fk_trainrun_train FOREIGN KEY (tr_tid)
REFERENCES train(t_id);
```

- 车次运行数据:

```
INSERT INTO trainrun VALUES('2022-04-29','G321',1,10);
INSERT INTO trainrun VALUES('2022-04-29','G2002',0,21);
INSERT INTO trainrun VALUES('2022-04-29','G1709',1,4);
INSERT INTO trainrun VALUES('2022-04-29','G2608',11,30);
INSERT INTO trainrun VALUES('2022-04-29','G305',1,13);
```

- 订单表:

```
CREATE TABLE orders
(
  o_id int,                        -- 订单 id（主键）
  o_uid varchar(20),               -- 用户 id（外键: 参照 users(u_id)）
  o_tdate date,                    -- 发车日期
  o_tid varchar(10),               -- 车次（外键：参照 train(t_id)）
  o_sstation varchar(20),          -- 上车站（外键：参照 station(s_name)）
  o_estation varchar(20),          -- 下车站（外键：参照 station(s_name)）
  o_seattype smallint,             -- 座位类型：一等 1、二等 2
  o_carriage smallint,             -- 车厢号
  o_seatnum smallint,              -- 座位号（排）
  o_seatloc char(1),               -- 座位位置：ABCEF
  o_price money,                   -- 订单金额
  o_ispaid boolean,                -- 是否已支付
  o_ctime timestamp,               -- 订单创建时间
  CONSTRAINT pk_orders PRIMARY KEY (o_id)
);
```

- 订单表外键约束:

```
ALTER TABLE orders ADD CONSTRAINT fk_orders_users FOREIGN KEY (o_uid) REFERENCES
users(u_id);
ALTER TABLE orders ADD CONSTRAINT fk_orders_train FOREIGN KEY (o_tid) REFERENCES
train(t_id);
ALTER TABLE orders ADD CONSTRAINT fk_orders_station_start FOREIGN KEY (o_sstation)
REFERENCES station(s_name);
ALTER TABLE orders ADD CONSTRAINT fk_orders_station_end FOREIGN KEY (o_estation)
REFERENCES station(s_name);
```

- 订单数据:

```
INSERT INTO orders VALUES(1,1,'2022-04-29','G2002','天津',
       '北京南',2,8,7,'F',54,1,'2022-04-27 16:00:12');
INSERT INTO orders VALUES(2,4,'2022-04-29','G321','天津南',
       '福州',1,4,7,'A',742.5,1,'2022-04-27 17:00:12');
INSERT INTO orders VALUES(3,3,'2022-04-29','G1709','天津西',
       '重庆西',2,9,3,'D',929,1,'2022-04-27 18:00:12');
INSERT INTO orders VALUES(4,2,'2022-04-29','G305','天津西',
       '长沙南',4,11,7,'E',657.5,1,'2022-04-27 19:00:12');
INSERT INTO orders VALUES(5,5,'2022-04-29','G321','沧州西',
       '合肥南',3,18,7,'E',325.5,0,'2022-04-27 20:00:12');
INSERT INTO orders VALUES(6,7,'2022-04-29','G1709','郑州东',
       '西安北',7,20,2,'F',206.5,1,'2022-04-27 10:00:12');
INSERT INTO orders VALUES(7,9,'2022-04-29','G305','石家庄',
       '武汉',1,8,2,'B',287.5,0,'2022-04-27 09:00:12');
INSERT INTO orders VALUES(8,8,'2022-04-30','G2608','天津西',
       '北京南',2,8,2,'C',56,1,'2022-04-28 09:00:12');
```

3. 在 railway 数据库中创建 pageinspect 扩展
   执行 CREATE EXTENSION 语句，创建依赖。

```
openGauss=# CREATE EXTENSION pageinspect;
CREATE EXTENSION
```

使用 SELECT * FROM pg_extension;可查看是否安装成功。

```
railway=# SELECT * FROM pg_extension;
     extname     | extowner | extnamespace | extrelocatable | extversion | extconfig |
extcondition
-----------------+----------+--------------+----------------+------------+-----------+---
 plpgsql         |       10 |           11 | f              | 1.0        |           |
 dist_fdw        |       10 |           11 | t              | 1.0        |           |
 file_fdw        |       10 |           11 | t              | 1.0        |           |
 hdfs_fdw        |       10 |           11 | t              | 1.0        |           |
 log_fdw         |       10 |           11 | t              | 1.0        |           |
 hstore          |       10 |           11 | t              | 1.1        |           |
 security_plugin |       10 |           11 | t              | 1.0        |           |
 pageinspect     |       10 |         2200 | t              | 1.0        |           |
(8 rows)
```

8.4.3 使用 pageinspect 插件分析表的页面结构

为了使用 pageinspect 插件分析表的页面结构，在 railway 数据库中创建一个新的表 users2，与 users 表结构完全相同，只是表名不同。

```
railway=# CREATE TABLE users2
(
   u_id varchar(20),              -- 用户 id，用于系统登录账户名（主键）
   u_passwd varchar(20),          -- 密码，用于系统登录密码
```

```
   u_name varchar(10),              -- 真实姓名
   u_idnum varchar(20),             -- 证件号码
   u_regtime timestamp,             -- 注册时间
   CONSTRAINT pk_users2 PRIMARY KEY (u_id)
);
NOTICE:  CREATE TABLE / PRIMARY KEY will create implicit index "pk_users2" for table "users2"
CREATE TABLE
```

1. 查看表页面的页头结构。

   查看空表的页头结构:

   在 users2 表创建后，但还未插入数据之前，执行:

```
railway=# SELECT * FROM page_header(get_raw_page('users2', 0));
ERROR:  block number 0 is out of range for relation "users2"
```

   返回错误信息“block number 0 is out of range for relation "users2"”，说明空表没有添加任何数据，也就没有创建任何页面，编号为 0 的页面不存在。

   执行一条插入语句:

```
railway=# INSERT INTO users2 VALUES(1, 'qweasd', '张三', '123456789', '2000-06-23
12:00:00');
INSERT 0 1
```

   再执行下列语句并查看返回结果:

```
railway=# SELECT * FROM page_header(get_raw_page('users2', 0));
    lsn    | tli | flags | lower | upper | special | pagesize | version | prune_xid
-----------+-----+-------+-------+-------+---------+----------+---------+-----------
0/259762F0 |   0 |     0 |    44 |  8128 |    8192 |     8192 |       6 |     18060
(1 row)
```

   观察可见，此时页面 0 已经创建。页头所占字节数分析如下:

```
lsn: 8 字节
tli: 2 字节
flags: 2 字节
lower: 2 字节
upper: 2 字节
special: 2 字节
pagesize、version: 2 字节
prune_xid: 4 字节
-------------------------- 24 字节 ------------
pd_xid_base: 8 字节
pd_multi_base: 8 字节
------------------------ 40 字节 --------------
一个 ItemIdData pd_linp 元组指针数组项: 4 字节
------------------------ 44 字节 ---------------
```

   可以看到，lower 列的值恰为 44。页面大小为 8192 字节，upper 列的值为 8128，即刚插入的第 1 条元组占用了 64 字节。

   插入第 2 条元组:

```
railway=# INSERT INTO users2 VALUES(2, 'qweasd', '李四', '123456712', '2000-06-24
13:00:00');
```

```
INSERT 0 1
```

再执行下列语句并查看返回结果

```
railway=# SELECT * FROM page_header(get_raw_page('users2', 0));
    lsn    | tli | flags | lower | upper | special | pagesize | version | prune_xid
-----------+-----+-------+-------+-------+---------+----------+---------+-----------
0/259765D0 |   0 |     0 |    48 |  8064 |    8192 |     8192 |       6 |     18060
(1 row)
```

可以看到, lower 列值此时变为了 48, 增加了 4, 验证了一个元组指针占用 4 字节。upper 列值此时变为了 8064，减少了 64，验证了此表的一个元组数据占 64 字节。将剩余的 7 条元组插入:

```
INSERT INTO users2 VALUES(3, 'qweasd', '王五', '123456754', '2000-06-25 14:00:00');
INSERT INTO users2 VALUES(4, 'qweasd', '赵六', '123456709', '2000-06-26 15:00:00');
INSERT INTO users2 VALUES(5, 'qweasd', '小明', '123423709', '2000-06-27 15:00:00');
INSERT INTO users2 VALUES(6, 'qweasd', '小李', '123423709', '2000-06-27 16:00:00');
INSERT INTO users2 VALUES(7, 'qweasd', '小赵', '123423712', '2000-06-27 10:00:00');
INSERT INTO users2 VALUES(8, 'qweasd', '小红', '142423709', '2000-06-27 11:00:00');
INSERT INTO users2 VALUES(9, 'qweasd', '小秦', '163423709', '2000-06-27 19:00:00');
```

再执行:

```
railway=# SELECT * FROM page_header(get_raw_page('users2', 0));
    lsn    | tli | flags | lower | upper | special | pagesize | version | prune_xid
-----------+-----+-------+-------+-------+---------+----------+---------+-----------
0/25977178 |   0 |     0 |    76 |  7616 |    8192 |     8192 |       6 |     18060
(1 row)
```

此时，共有 9 个元组，9 个元组指针占用 36 字节，因此 lower 列值为 76；插入表中的 9 个元组数据占 64*9 = 576 字节，因此 upper 列值为 8192 – 576 = 7616。

2. 查看页面中的元组数据。

   执行

```
railway=# SELECT * FROM heap_page_items(get_raw_page('users2', 0));
```

   返回

```
railway=# SELECT * FROM heap_page_items(get_raw_page('users2', 0));
 lp | lp_off | lp_flags | lp_len | t_xmin | t_xmax | t_field3 | t_ctid | t_infomask2 | t_infomask | t_hoff | t_bits | t_oid
----+--------+----------+--------+--------+--------+----------+--------+-------------+------------+--------+--------+-------
  1 |   8128 |        1 |     64 |  18063 |      0 |        0 | (0,1)  |           5 |       2050 |     24 |        |
  2 |   8064 |        1 |     64 |  18064 |      0 |        0 | (0,2)  |           5 |       2050 |     24 |        |
  3 |   8000 |        1 |     64 |  18065 |      0 |        0 | (0,3)  |           5 |       2050 |     24 |        |
  4 |   7936 |        1 |     64 |  18066 |      0 |        0 | (0,4)  |           5 |       2050 |     24 |        |
  5 |   7872 |        1 |     64 |  18067 |      0 |        0 | (0,5)  |           5 |       2050 |     24 |        |
  6 |   7808 |        1 |     64 |  18068 |      0 |        0 | (0,6)  |           5 |       2050 |     24 |        |
  7 |   7744 |        1 |     64 |  18069 |      0 |        0 | (0,7)  |           5 |       2050 |     24 |        |
  8 |   7680 |        1 |     64 |  18070 |      0 |        0 | (0,8)  |           5 |       2050 |     24 |        |
  9 |   7616 |        1 |     64 |  18071 |      0 |        0 | (0,9)  |           5 |       2050 |     24 |        |
(9 rows)
```

图 8.9 查看页面中的元组数据

从图 8.9 中, 可以看到元组指针中的 lp_off 字段的变化, 元组长度字段 lp_len 为 64, 插入元组操作的事务 id（t_xmin）的递增，t_cid 元组标识（TID）的变化。

3. 查看页面的原始数据。

   执行:

```
railway=# SELECT get_raw_page::text FROM get_raw_page('users2', 0);
```

   返回页面的原始数据以 16 进制形式输出的文本，可以看到前面的页面页头和后面的页面数据，中的“0”都是空闲页面部分（输出过长中间部分省略）。如图 8.10 和图 8.11 所示。

图 8.10 查看页面的原始数据（页头部分）（以 16 进制形式输出数据）

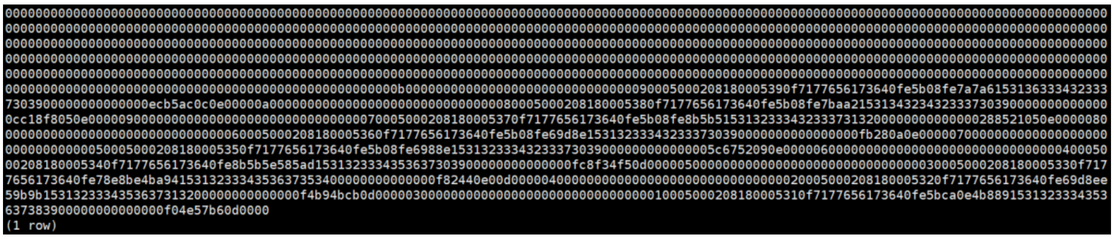

图 8.11 查看页面的原始数据（页尾部分）（以 16 进制形式输出数据）

综合以上信息，可以得到 users 表的页面结构图如图 8.12。

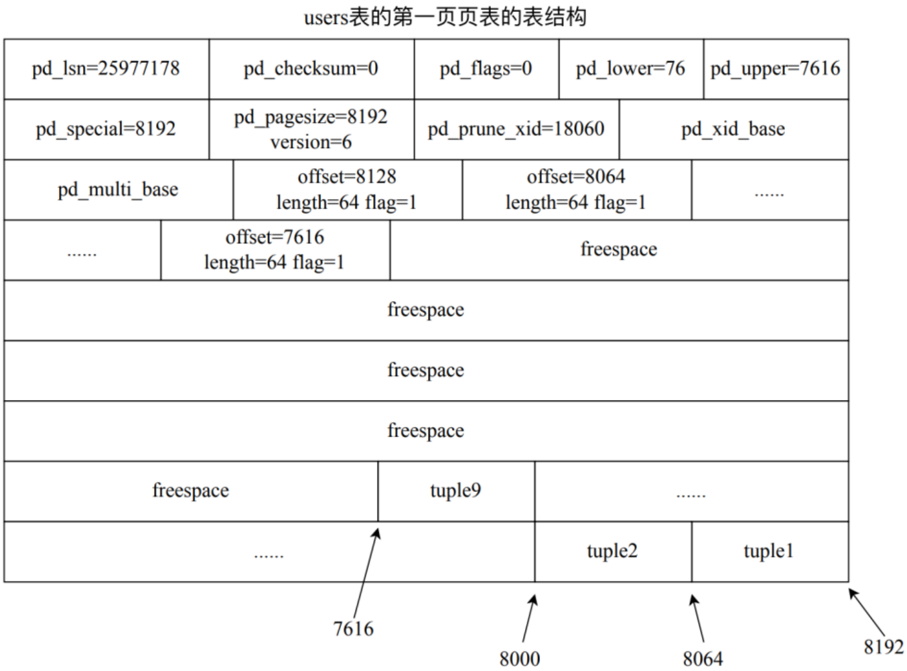

图 8.12 users 表的页面结构示意图

8.5 实验结果

【请按照要求完成实验操作】

1. 完成实验步骤 8.4.1 节，安装 pageinspect 插件。

2. 完成实验步骤 8.4.2 节，使用 gsql 创建表和插入数据。

3. 完成实验步骤 8.4.3 节，根据分析 users 表页面结构的例子，分析 orders 表页面结构，画出 orders 表页面结构图（可使用 Visio 等画图工具）。

4. 使用 pageinspect 插件功能，对 railway 数据库中的其他表调用 page_header 和 heap_page_items 函数，获取对应的页头结构和元组数据描述。

8.6 讨论与总结

【请将实验中遇到的问题描述、解决办法与思考讨论列在下面，并对本实验进行总结。】