第8讲　开源数据库系统

8.1 概述

8.2 PostgreSQL的学习、使用与定制

8.3 PostgreSQL的主题分析

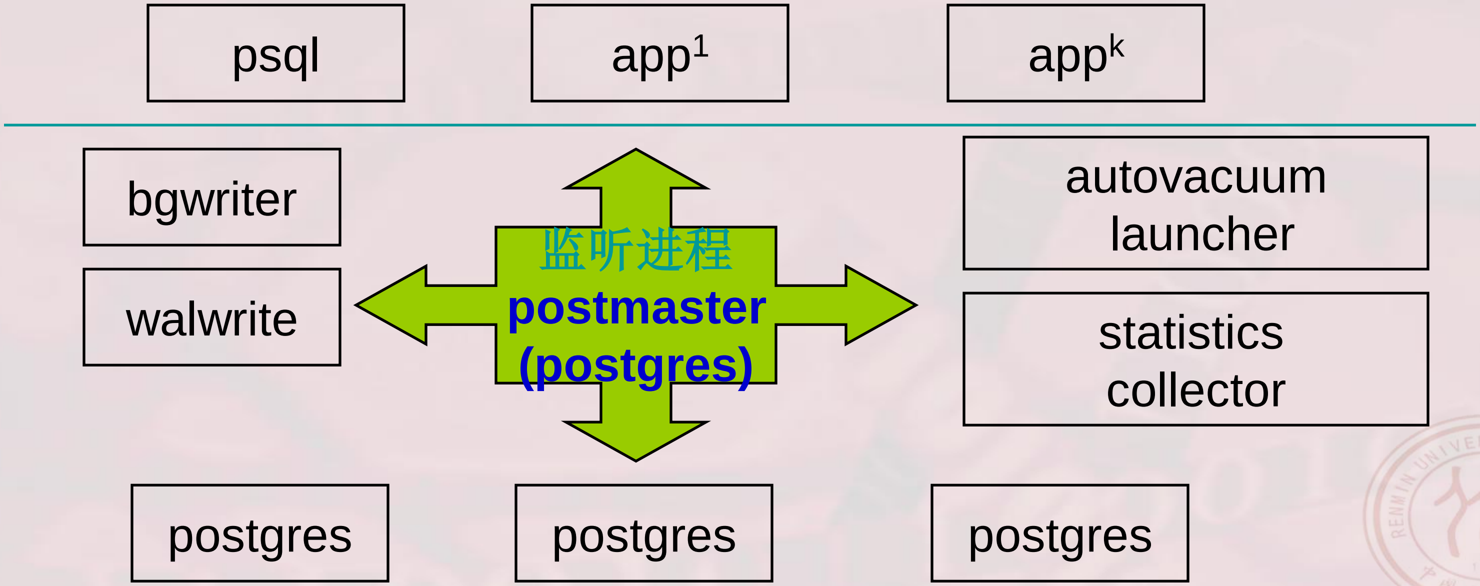
8.3 PostgreSQL的主题分析

- ❖ 进程结构
  - 辅助进程、信号处理器
- ❖ 存储管理器
  - OO设计、空间组织
- ❖ 缓冲区管理器
  - 淘汰算法、多核优化技术
- ❖ 查询处理
  - 执行设计思想
- ❖ 事务处理
  - 多版本并发控制(MVCC)

进程结构

❖ **基本结构---进程**

■ **1:1**



辅助进程

- **核心监听进程**
  - postmaster: 启动服务器集群，监听客户请求，分派后台处理进程
- **主要辅助进程**
  - Syslogger: System logger(系统输出登记进程)
    - 记录服务器运行中各类输出信息
  - BgWriter: Backend writer (后台写进程)
    - 回写“脏”数据缓冲区
  - WalWriter: WAL writer background (WAL日志写进程)
    - 回写WAL日志缓冲区
  - AutoVaccum: autovacuum daemon (自动清理进程)
    - 周期性对数据库进行过时数据的清理
  - PgStat: PostgreSQL statistics collector (统计信息收集进程)
    - 收集各种统计信息，运行状态或数据的

REAPER函数

- 子进程死亡的信号处理函数，完成清理工作。
  - SIGCHLD：子进程=> postmaster

  // pqsignal(SIGCHLD, reaper); /* handle child termination */

- 处理能力
  - 根据子进程类型/信息，完成相应的清理。
  - 例子
    - Walwriter/AutoVacuum：调用HandleChildCrash()处理崩溃；
    - SysLogger：调用SysLogger_Start()重启SysLogger。
    - 辅助进程pid=0，启动辅助进程。

存储管理器

- 存储管理器--smgr
- 大对象--large_object
- 物理页面管理--page
- 空闲空间管理—freespace

存储管理器--smgr

❖ 通用存储管理器(@smgr.c)—OO设计

```
typedef struct f_smgr  {
        void            (*smgr_init) (void);                /* may be NULL */
        void            (*smgr_shutdown) (void);            /* may be NULL */
        void            (*smgr_close) (SMgrRelation reln, ForkNumber forknum);
        void            (*smgr_create) (SMgrRelation reln, ForkNumber forknum,  bool isRedo);
        bool            (*smgr_exists) (SMgrRelation reln, ForkNumber forknum);
        void            (*smgr_unlink) (RelFileNodeBackend rnode, ForkNumber forknum, bool isRedo);
        void            (*smgr_extend) (SMgrRelation reln, ForkNumber forknum,
                                        BlockNumber blocknum, char *buffer, bool skipFsync);
        void            (*smgr_prefetch) (SMgrRelation reln, ForkNumber forknum,  BlockNumber blocknum);
        void            (*smgr_read) (SMgrRelation reln, ForkNumber forknum,
                          BlockNumber blocknum, char *buffer);
        void            (*smgr_write) (SMgrRelation reln, ForkNumber forknum,
                                        BlockNumber blocknum, char *buffer, bool skipFsync);
        BlockNumber (*smgr_nblocks) (SMgrRelation reln, ForkNumber forknum);
        void            (*smgr_truncate) (SMgrRelation reln, ForkNumber forknum,  BlockNumber nblocks);
        void            (*smgr_immedsync) (SMgrRelation reln, ForkNumber forknum);
        void            (*smgr_pre_ckpt) (void);            /* may be NULL */
        void            (*smgr_sync) (void);                /* may be NULL */
        void            (*smgr_post_ckpt) (void);           /* may be NULL */
} f_smgr smgrsw[ ] = { /* magnetic disk */{  mdinit, NULL, mdclose, mdcreate, mdexists, mdunlink, mdextend,  mdprefetch,
        mdread, mdwrite, mdnblocks, mdtruncate, mdimmedsync, mdpreckpt, mdsync, mdpostckpt   } };
```

大对象--large_object(xLOB)

❖ 系统表pg_largeobject

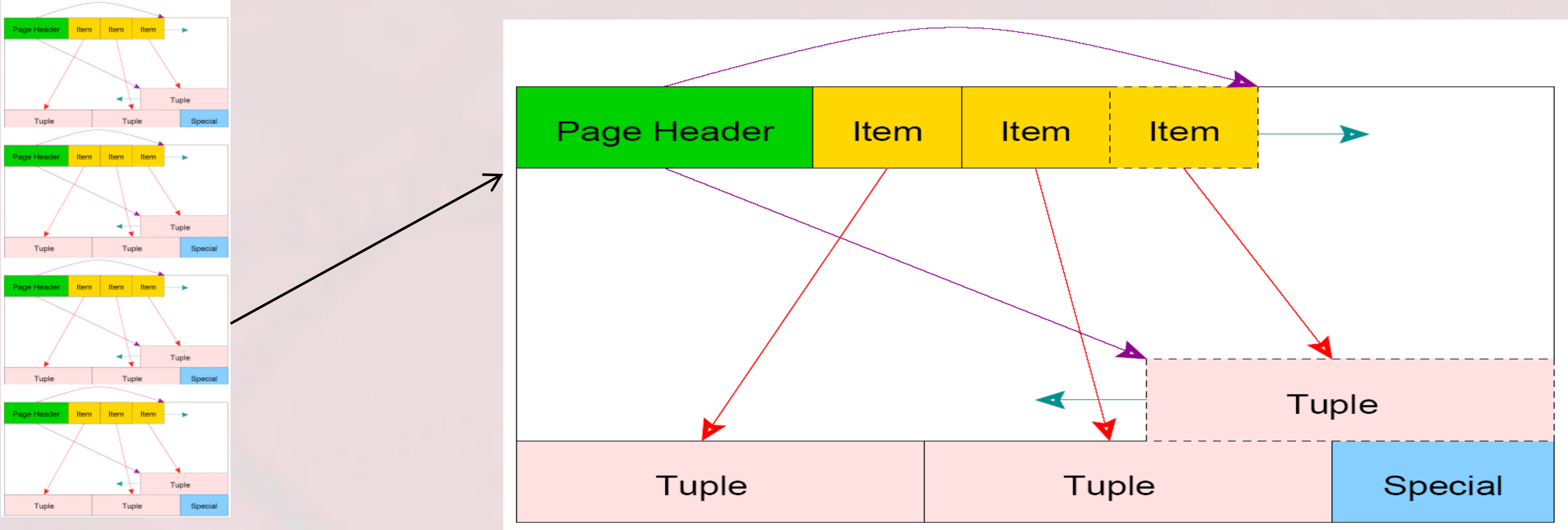
eg. 大的文本文件，比如某个3MB的网页

Name	Type	Description
loid	oid	Identifier of the large object that includes this page
pageno	int4	Page number of this page within its large object (counting from zero)
data	bytea	Actual data stored in the large object. This will never be more than LOBLKSIZE (BLCKSZ / 4) bytes and might be less

loid	pageno	data
2	0	we....
2	1	are...
3	0	next...

物理页面管理--page

- 数据页面= Header + {data_item}



- 记录的地址=>页面物理地址 +页内偏移序号
- 可以很容易扩展为类似**段页式**的存储组织

物理页面管理--page

❖ 数据项索引方式--bufpage.h

Item	Description
PageHeaderData	24 bytes long. Contains general information about the page, including free space pointers.
ItemIdData	Array of (offset,length) pairs pointing to the actual items. 4 bytes per item.
Free space	The unallocated space. New item pointers are allocated from the start of this area, new items from the end.
Items	The actual items themselves.
Special space	Index access method specific data. Different methods store different data. Empty in ordinary tables.

物理页面管理--page

❖ 数据项索引方式

```
/*
 * A postgres disk page is an abstraction layered on top of a postgres
 * disk block (which is simply a unit of i/o, see block.h).
 *
 * specifically, while a disk block can be unformatted, a postgres
 * disk page is always a slotted page of the form:
 *
 * +----------------+---------------------------------+
 * | PageHeaderData | linp1 linp2 linp3 ...           |
 * +-----------+----+---------------------------------+
 * | ... linpN |                                      |
 * +-----------+--------------------------------------+
 * |           ^ pd_lower                             |
 * |                                                  |
 * |             v pd_upper                           |
 * +-------------+------------------------------------+
 * |             | tupleN ...                         |
 * +-------------+------------------+-----------------+
 * |       ... tuple3 tuple2 tuple1 | "special space" |
 * +--------------------------------+-----------------+
```

物理页面管理--page

❖ Item

```
typedef struct ItemIdData          /* C的技巧 */
{
        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;
typedef ItemIdData *ItemId;

#define ItemIdIsValid(itemId)    PointerIsValid(itemId)
#define ItemIdIsUsed(itemId) \
        ((itemId)->lp_flags != LP_UNUSED)
...
```

物理页面管理--page

❖ 磁盘页面布局 --- bufpage.h

```
typedef struct PageHeaderData {
        /* XXX LSN is member of *any* block, not only page-organized ones */
        XLogRecPtr   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[1];      /* beginning of line pointer array */
} PageHeaderData;
// storage/page/bufpage.c  [PageAddItem()]
void PageInit(Page page, Size pageSize, Size specialSize)
{
        ...
        p->pd_lower = SizeOfPageHeaderData;
        p->pd_upper = pageSize - specialSize;
        p->pd_special = pageSize - specialSize;
        PageSetPageSizeAndVersion(page, pageSize, PG_PAGE_LAYOUT_VERSION);
}
```

物理页面管理--page

- **Tuple结构--** ***HeapTupleHeaderData***
  - **src/include/access/htup_details.h**

Field	Type	Length	Description
t_xmin	TransactionId	4 bytes	insert XID stamp
t_xmax	TransactionId	4 bytes	delete XID stamp
t_cid	CommandId	4 bytes	insert and/or delete CID stamp (overlays with t_xvac)
t_xvac	TransactionId	4 bytes	XID for VACUUM operation moving a row version
t_ctid	ItemPointerData	6 bytes	current TID of this or newer row version
t_infomask2	int16	2 bytes	number of attributes, plus various flag bits
t_infomask	uint16	2 bytes	various flag bits
t_hoff	uint8	1 byte	offset to user data

空闲空间管理--freespace

- **FreeSpaceMap-*freespace/freespace.c***
  - 空闲空间映射
  - 快速找到某个关系中的可用空间(eg. 插入元组)
- **持久外存文件---FSM页面(二叉树)**

A non-leaf node stores the max amount of free space on any of its children.

```
          4
      4       2
   3   4    0    2      <-    heap pages
```

缓冲区管理器

- 主要实现技术
  - 存取控制基本机制
  - 缓冲区描述符--BufferDesc
  - 淘汰策略管理器--freelist.c
- 多核优化技术
  - 缓存线(cache line)对齐
  - 缓冲区映射哈希表分区
  - 使用memory barrier(atomic)

存取控制基本机制

@storage/buffer/README

- 引用计数 + 缓冲区封锁
  - 使用状态 vs. 争用/竞争现象

    IPC中的信号量技术--Prodr&Consr

    cnt 可用资源量

    lock 改变状态/信息
- 引用计数(reference count)
  - ReadBuffer vs. ReleaseBuffer
- 缓冲区封锁(buffer content lock)
  - shared vs. exclusive
  - 短锁

缓冲区描述符--BufferDesc

❖ 共享缓冲区描述符或状态

```
typedef struct BufferDesc                    /* 9.6.2 */
{
        BufferTag      tag;                  /* ID of page contained in buffer */
        int            buf_id;               /* buffer's index number (from 0) */

        pg_atomic_uint32 state;              /* state of the tag, containing flags(10), usagecount(4) and refcount (18)*/

        int            wait_backend_pid;     /* backend PID of pin-count waiter */
        int            freeNext;             /* link in freelist chain */

        LWLockId       *content_lock;   /* to lock access to buffer contents */ [SyncOneBuffer()=> locking]
} BufferDesc;

typedef struct buftag
{
        RelFileNode rnode;                   /* physical relation identifier */
        ForkNumber  forkNum;                 /* main/fsm/vm/init */
        BlockNumber blockNum;                /* blknum relative to begin of reln */
} BufferTag;

// BufFlags
#define BM_DIRTY       (1 << 0)              /* data needs writing */
```

缓冲区描述符--BufferDesc

❖ 共享缓冲区状态标记

pg_atomic_uint32 state; /* state of the tag, containing flags, refcount and usagecount */

```
/*
 * Buffer state is a single 32-bit variable where following data is combined.
 *
 * - 18 bits refcount
 * - 4 bits usage count
 * - 10 bits of flags
 *
 * Combining these values allows to perform some operations without locking
 * the buffer header, by modifying them together with a CAS loop.
 *
 * The definition of buffer state components is below.
 */
#define BUF_REFCOUNT_ONE 1
#define BUF_REFCOUNT_MASK ((1U << 18) - 1)
#define BUF_USAGECOUNT_MASK 0x003C0000U
#define BUF_USAGECOUNT_ONE (1U << 18)
#define BUF_USAGECOUNT_SHIFT 18
#define BUF_FLAG_MASK 0xFFC00000U

/* Get refcount and usagecount from buffer state */
#define BUF_STATE_GET_REFCOUNT(state) ((state) & BUF_REFCOUNT_MASK)
#define BUF_STATE_GET_USAGECOUNT(state) (((state) & BUF_USAGECOUNT_MASK) >> BUF_USAGECOUNT_SHIFT)
```

淘汰策略管理器

- 保护锁： BufFreelistLock
- 时钟算法"clock sweep "

  StrategyGetBuffer(BufferAccessStrategy strategy, uint32 *buf_state)[可扩展]
- 基本淘汰策略--确定victim页面

@StrategyGetBuffer()/ @buffer/README

1. 获得buffer_strategy_lock锁。
2. 如果缓冲区空闲链表非空，则将其移出。释放buffer_strategy_lock锁。如果缓冲区被“钉”住或使用计数非零则继续检查下一个，否则“钉”住该页面、并返回缓冲区描述符。
3. 选择策略的 NextVictimBuffer所指向的缓冲区，并循环前进：

   如果缓冲区被“钉”住或使用计数非零，则将使用计数-1，

   重获得buffer_strategy_lock锁，并继续检查下一个
4. “钉”住该页面、并返回缓冲区描述符。

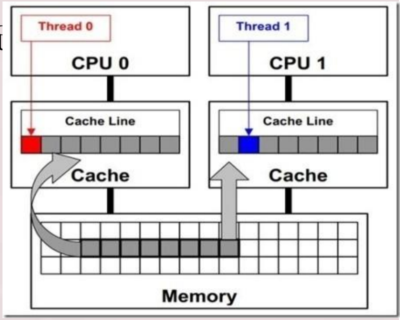
对“脏”页面，需要将数据写出后再重用。

多核优化技术

❖ 缓存线对齐

- BufferDescriptors数组的起始位置在缓存线(cache line)的边界，并使数组元素的大小适配
- 例如64字节的作为缓存线，最好也要对齐到边界，避免伪共享（false sharing）

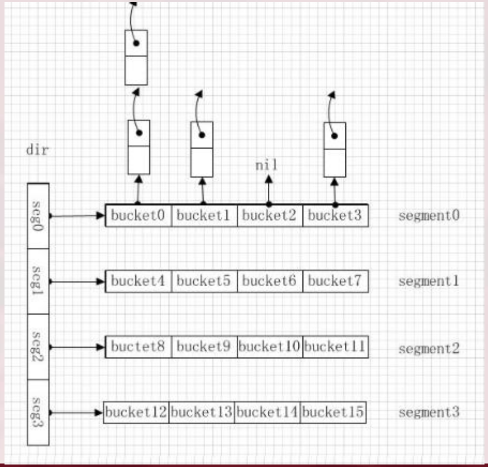
```
typedef union BufferDescPadded
{
    BufferDesc   bufferdesc;
    char         pad[BUFFERDESC_PAD_TO_SIZE];
} BufferDescPadded;
```



- 缓冲区初始化时完成

❖ 缓冲区映射哈希表分区扩充

- 判定是否命中：BufferTag --> hash code
- 分区扩充以减少冲突造成的瓶颈：多核=> 竞争增加



❖ 使用memory barrier(atomic)

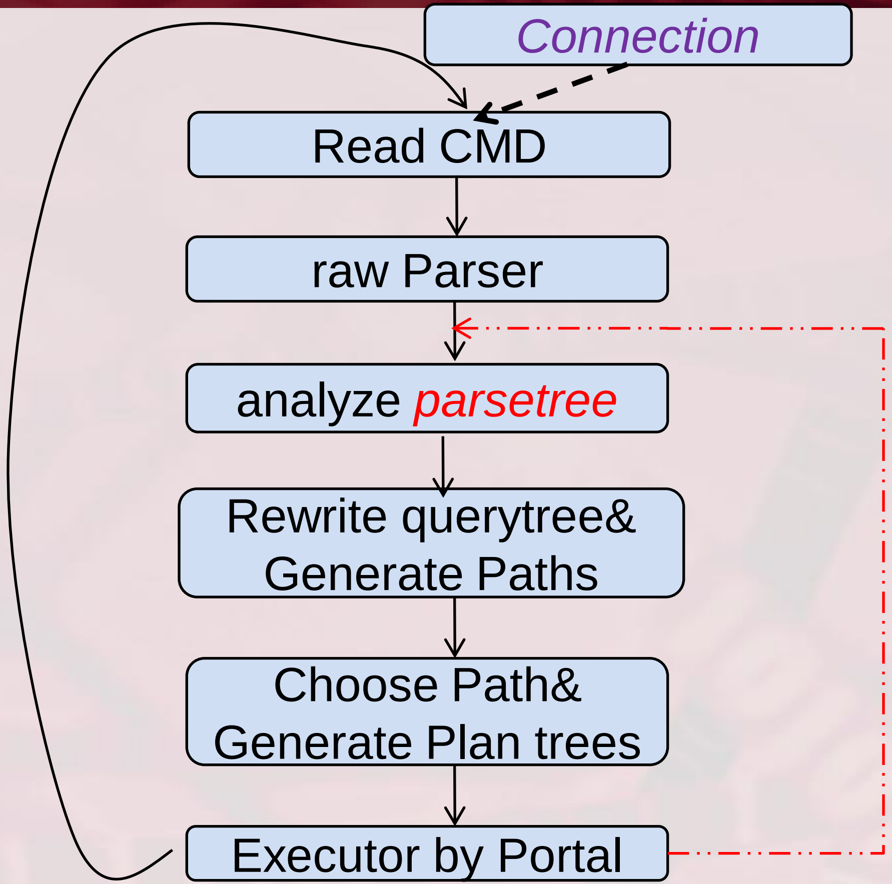
- 控制内存的并发乱序访问--硬件/编译器等级别

```
pg_atomic_uint32 state; // @BufferDesc
...
pg_atomic_compare_exchange_u32(&buf->state,...);
```

查询处理

- 查询执行
- 执行控制采用portal机制
  - 系统设计方法
  - 类似进程控制块PCB
- 代码结构(自学)

查询执行流程


Connection
Read CMD
raw Parser
analyze *parsetree*
Rewrite querytree&
Generate Paths
Choose Path&
Generate Plan trees
Executor by Portal

查询计划执行

❖ 执行控制采用portal机制
- 类似进程控制块PCB => 系统设计方法
- 运行中/可运行查询的执行状态抽象
- /* 支持SQL级游标和协议级的portal */

// include/utils/portal.h[文件头注释 => 自我学习]

```
typedef struct PortalData{
    /* Bookkeeping data */
    const char *name;                 /* portal's name */
    const char *prepStmtName;         /* source prepared statement (NULL if none) */
    ……
} PortalData;

// PortalRun => PortalRunSelect => ExecutorRun => ExecutePlan => ...
      [case] => PortalRunMulti //@PORTAL_MULTI_QUERY -- Util Stmt/crt tb
```

查询处理--代码结构

- src/backend
  - commands：DDL命令及其他命令
  - executor：SQL语法节点命令执行器
  - nodes：语法树节点操作函数
  - optimizer：优化器相关处理
  - parser：SQL词法分析器和语法分析器
  - rewrite：语法树重写形成查询计划树
  - tcop：启动计划执行并返回结果

查询处理--代码结构

❖ 查询执行基本过程--重要代码

[*PostgresMain*()=>exec_simple_query(query_string)]

1. pgstat_report_activity(query_string);
2. start_xact_command(); // 幂等调用
3. parsetree_list = pg_parse_query(query_string);
4. foreach(parsetree_item, parsetree_list)
   1) commandTag = CreateCommandTag(parsetree);
   2) start_xact_command();
   3) querytree_list = pg_analyze_and_rewrite(parsetree, query_string,NULL, 0);
   4) plantree_list = pg_plan_queries(querytree_list, 0, NULL, false);
   5) PortalStart(portal, NULL, InvalidSnapshot);
   6) PortalRun(portal, FETCH_ALL,...);
   7) finish_xact_command();

事务处理

- 多版本并发控制机制

  MVCC: Multi-Version Concurrency Control

  - 基于快照(snapshot)
  - 通过快照动态形成版本

- 代码结构

多版本并发控制机制

例 1：下图在使用 MTVO 的情况下，A 共有三个版本：事务 T_1 开始时的版本 A_0，T_1 写入的版本 A_1，以及 T_2 写入的版本 A_2。事务 T_1，T_2，T_3，T_4 开始的时间戳为 150，225，175 和 200，时间戳的大小代表了事务的先后顺序。

T_1	T_2	T_3	T_4	A_0	A_1	A_2
150	225	175	200			
$r_1(A)$				读		
$w_1(A)$					创建	
	$r_2(A)$				读	
			$w_4(A)$			
			回滚			
	$w_2(A)$					创建
		$r_3(A)$			读	

多版本并发控制机制

❖ 多版本信息@元组

```
 * The overall structure of a heap tuple looks like:
 *                    fixed fields (HeapTupleHeaderData struct)
 *                    ... ...
 * 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. ......
typedef struct HeapTupleFields  { [@ src/include/access/htup.h]
        TransactionId t_xmin;                /* inserting xact ID */
        TransactionId t_xmax;                /* deleting or locking xact ID */
        union       {
                CommandId     t_cid;      /* inserting or deleting command ID, or both */
                TransactionId t_xvac;     /* old-style VACUUM FULL xact ID */
        } t_field3;
} HeapTupleFields;
typedef struct HeapTupleHeaderData { ... ... }
```

多版本并发控制机制

❖ 基于快照(snapshot)--SnapshotData

```
typedef struct SnapshotData{        /* include/utils/snapshot.h */
        SnapshotSatisfiesFunc satisfies;        /* tuple test function */
        TransactionId xmin;        /* all XID < xmin are visible to me */
        TransactionId xmax;        /* all XID >= xmax are invisible to me */
        TransactionId *xip;        /* array of xact IDs in progress */
        uint32      xcnt;        /* # of xact ids in xip[] */
                                   /* all ids in xip[] satisfy xmin <= xip[i] < xmax */
        int32       subxcnt;        /* # of xact ids in subxip[] */
        TransactionId *subxip;     /* array of subxact IDs in progress */
        bool        suboverflowed;        /* has the subxip array overflowed? */
        bool        takenDuringRecovery;        /* recovery-shaped snapshot? */
        bool        copied;        /* false if it's a static snapshot */
        /* all ids in subxip[] are >= xmin, but no filtering out any that are >= xmax */
        CommandId curcid;        /* in my xact, CID < curcid are visible */
        uint32      speculativeToken;
        /*  Book-keeping information, used by the snapshot manager */
        uint32            active_count;/* refcount on ActiveSnapshot stack */
        uint32            regd_count;  /* refcount on RegisteredSnapshots */
        pairingheap_node  ph_node;     /* link in the RegisteredSnapshots heap */
        int64             whenTaken;   /* timestamp when snapshot was taken */
        XLogRecPtr        lsn;         /* position in the WAL stream when taken */
} SnapshotData;
```

多版本并发控制机制

❖ 获得快照GetSnapshotData()

// @ src/backend/storage/ipc/procarray.c [@comments]
Snapshot GetSnapshotData(Snapshot snapshot);

■ 获得运行事务的信息

1. xmin： 仍在运行的最小的事务ID
2. xmax：已经完成的最大的事务ID+1
3. xip： 正在运行的事务ID列表(xmin <= xid < xmax)
   - XID < xmin：已经完成
   - XID >= xmax：仍在运行
   - xmin <= xid < xmax：检查xip列表 => 运行/完成
4. 除快照中列出的事务，所有判定为“较老”事务的执行效果都可见。

■ 获得快照后“运行事务”集合不变

- 包括所有顶级事务及可能多的子事务 => 溢出处理

■ 全局变量

TransactionXmin、RecentXmin、RecentGlobalXmin

多版本并发控制机制

- **设置快照 CurrentSnapshot**

```
// for SelectQuery
PortalStart()
        -> PushActiveSnapshot()
                -> GetTransactionSnapshot() // CurrentSnapshot
        -> GetActiveSnapshot()
        -> ExecutorStart() -> standard_ExecutorStart()
                // estate->es_snapshot =
                        RegisterSnapshot(queryDesc->snapshot);[Ln195]
```

- **元组可见性判断**

HeapTupleSatisfiesVisibility()

```
// @ src/include/utils/tqual.h
#define HeapTupleSatisfiesVisibility(tuple, snapshot, buffer) \
        ((*(snapshot)->satisfies) ((tuple)->t_data, snapshot, buffer))
```

多版本并发控制机制

◆ 示例代码片段

```
// 扫描下一个元组
// @ src/backend/access/heap/heapam.c

HeapTuple
heap_getnext(HeapScanDesc scan, ScanDirection direction)
// heap_getnext
        -> heapgettup
                -> HeapTupleSatisfiesVisibility
                    [HeapTupleSatisfiesMVCC() // @time/tqual.c]
                        [GetTransactionSnapshot设置]
```

事务处理--代码结构

- src/backend/utils/time
  - tqual.c：元组可见性规则
  - combocid.c：组合的命令id(根据cmin和cmax)
- src/backend/storage/ipc/procarray.c
  - backend的运行信息[事务快照数据]
    GetSnapshotData

事务处理--代码结构

- src/backend/access/transam
  - README
  - clog.c/transam.c：事务日志clog管理
  - multixact.c：多事务日志clog管理
  - rmgr.c：resource mgr
  - slru.c：clog bufmgr
  - subtrans.c：子事务日志clog管理
  - twophase.c/twophase_rmgr.c：两阶段提交管理
  - varsup.c：oid&xid变量维护
  - xact.c：事务管理UI
  - xlog.c/xlogfuncs.c/xlogutils.c：xlog manager

小结

- 目标---{知识，能力，素质}
- 如何学习、使用与定制开源数据库
  - 以ORDBMS PostgreSQL为例
  - (尝试)喜欢它--your pet
- PostgreSQL的主题选讲
  - 若干主题:系统结构、存储管理、缓冲区、查询、MVCC
  - 思想菁华：语言特性、OO设计、可扩展、编程风格(职业素养)
- 抛砖引玉---终身学习