数据库并发控制 实验报告

1 WKDB 并发框架

在wkdb并发测试框架体系之内，所有的并发控制算法都需要针对两个不同的主体来进行修改和编写：

- 数据记录。可以简单理解为数据库中的一个元组，表中的一行。对于数据记录的相关变量信息和函数操作全部都是通过类RowData来实现和维护的。通过这个类中的变量和方法，可以获取或者设置这一条记录的主码、所在的表名、模式名等等。其中对于并发控制算法实现来说可能需要使用和修改的部分如下所示：

  ```
  RC RowData::get_row(access_t type, TxnMgr * txn, RowData *& row)
  ```

  get_row()是事务与数据项进行交互的主要接口之一。一个事务通过调用该函数，请求访问特定的数据项。txn是请求访问的事务指针，RowData是请求访问的数据项指针，type标志这次请求访问的类型是读操作还是写操作。在这个函数内部需要对是否允许事务进行访问做出判断，如果允许访问，就要使用事务内部txn的cur_row作为指针，申请一个RowData空间的大小并深拷贝row到事务自己内部的cur_row。最终使用返回值来标识这次访问的状态是RCOK（成功访问）还是Abort（失败）。

  ```
  void RowData::return_row(RC rc, access_t type, TxnMgr * txn, RowData * row)
  ```

  return_row()也是事务与数据项进行交互的主要接口之一，在事务访问完成之后会对这一数据项进行写回和回收处理，有一点类似于将修改刷回磁盘。其中rc是access之后的状态，也就是get_row的返回值，type代表这次读写的类型，txn是进行操作的事务，而row是这个事务进行操作和修改之后某一行的新的值。这个新的值跟据rc的状态和type的类型来决定要不要写回到当前的RowData，也就是this中去，同时，无论是否写回需要将row的内存释放掉。

  ```
  void copy(RowData * src);
  ```

  copy()是将数据项进行深拷贝的函数，往往是txn的cur_row调用这个函数将某一个RowData深拷贝到自身。是数据访问所必需的。

  ```
  #if ALGO == MVCC
      Row_mvcc * manager;
  #elif ALGO == OCC
      Row_occ * manager;
  #endif
  ```

  manager是在每一个数据项上进行并发控制操作的类指针，指向相应算法的manager。依据使用的并发控制算法的不同，会声明不同的manager，我们要实现的重要一部分就是完成Row_mvcc/Row_occ类的编写。

- 事务。具体而言就是TxnMgr类，对于每一个事务，都有其自身的manager，利用这个manager来维护其信息。其中对于我所实现的算法而言，最重要的信息主要是时间戳：

```
void set_start_timestamp(uint64_t start_timestamp);
ts_t get_start_timestamp();
void set_end_timestamp(uint64_t end_timestamp)
uint64_t get_end_timestamp()
```

除此之外还记录了这个事务所有的访问记录，并提供了这些访问记录的类型、数量：

```
uint64_t access_get_cnt() {return txn->row_cnt;}
uint64_t sizeof_write_set() {return txn->write_cnt;}
uint64_t sizeof_read_set() {return txn->row_cnt - txn->write_cnt;}
access_t access_get_type(uint64_t access_id) {return txn->access[access_id]-
>type;}
RowData * access_get_original_row(uint64_t access_id) {return txn-
>access[access_id]->orig_row;}
```

基本上实现或者修改一个并发控制算法只需要修改以上几个部分就可以了，以及可能需要在global.cpp，global.h之中加入头文件并声明全局变量即可

2 MaaT

在实现我自己的乐观控制法之前，我首先调研了wkdb框架之中给出来的OCCTEMPLATE算法，经过我的文献搜集和资料调查，我发现这个OCCTEMPLATE模板算法实质上并不是最初OCC的初版论文 *Kung, Hsiang-Tsung, and John T. Robinson. "On optimistic methods for concurrency control."*所提出来的原始乐观控制法，而是对OCC的改进和变种。我按照OCCTEMPLATE算法的主体思想进行搜索，找到了另外一篇在VLDB上边的论文 *MaaT: Effective and scalable coordination of distributed transactions in the cloud*。经过确认，OCCTEMPLATE大体上是MaaT的一个简单实现，下面简单介绍MaaT的算法原理和流程

一个并发控制算法和核心是“定序”和“检验”，也就是人为地确定一个事务执行的顺序，使得并发算法的执行效果等同于按照这个顺序进行串行执行的效果。而MaaT很特殊的一点是它采用的是“动态定序”，也就是一个事务的时间戳并不是在事务开始或者执行读写操作时确定的，而是在事务的验证阶段，根据事务所做过的读写操作，人为地尽可能确定一个不引发冲突的时间戳。

具体而言，在一个事务进行提交时，维护一个lowerbound和upperbound，分别界定这个事务的时间戳可以位于的上界和下界：

- 首先要找到他所有读过的记录的last write时间戳，当前事务的时间序必须要大于这个时间戳，也就是要规范下界
- 其次要找到这个事务在执行read读取操作时，所有的记录所涉及到的uncommitted write的其他事务，要求这些事务的提交时间序必须要大于当前事务（否则，当前事务所读到的就应当是这些事务修改之后的记录，而不是现在读到的记录），为此，可能需要提高上界upperbound。
- 然后对于这个事务所执行过的write操作（论文中称为prewrite），要保证当前事务的时间序应当在这些记录的last read之后，否则那些事务所读取到的记录版本就是错误的，因此需要检查并提高下界lowerbound。
- 最后，对于写写冲突，规定和当前事务一块写同一些记录，可能存在写写冲突的uncommitted write，当前事务的时间戳应当在他们后边，也就是当前事务的lowerbound要高于他们的upperbound
- 规定好当前事务的lower和upper之后，检查是否存在upper大于lower。如果满足就可以直接返回验证通过，直接提交事务。否则，如果是upper反而要小于lower，就代表无论如何都找不到一个当前事务的合理的时间戳，因此应当将当前进行验证的事务返回验证结果为不通过，直接进行abort

3 silo

3.1 算法思路

我所分配到的算法是乐观控制法（Optimistic Concurrency Control，OCC）。乐观控制法本质上与两阶段锁，时间戳算法有很大的区别，总体来说，乐观控制法只会在事务进行提交时进行验证，验证一个事务所处理过的所有的记录和数据是否会与之前提交的事务发生读写冲突、写读冲突、写写冲突。如果不存在冲突现象，那么事务可以被正常提交，否则验证不通过，事务只能回滚。这是最原始的OCC算法，出自 *On Optimistic Methods for Concurrency Control*.

而我所实现的算法实际上更加类似于silo算法，出自*Speedy Transactions in Multicore In-Memory Databases.* 在论文中提出的算法运行伪代码如下：

```
Data: read set R, write set W, node set N,
      global epoch number E
// Phase 1
for record, new-value in sorted(W) do
    lock(record);
compiler-fence();
e ← E;                                  // serialization point
compiler-fence();
// Phase 2
for record, read-tid in R do
    if record.tid ≠ read-tid or not record.latest
          or (record.locked and record ∉ W)
    then abort();
for node, version in N do
    if node.version ≠ version then abort();
commit-tid ← generate-tid(R, W, e);
// Phase 3
for record, new-value in W do
    write(record, new-value, commit-tid);
    unlock(record);
```

- 第一阶段，一个事务进入验证阶段之后，首先对他的写集按照某种特定的顺序进行排序，将排好序的写集按照顺序加锁。
- 第二阶段，对于这个事务的所有读集，检验一下这个数据项上的最后写入时间戳wts与当前待检验事务的开始时间戳之间的关系。

  如果最后写入时间戳wts要大于当前事务的开始时间戳ts，就说明在当前事务读完之后有其他事务修改，并且，因为wts已经修改过了，就说明那个事务已经验证通过了，按照OCC的定序顺序，当前事务应当在后，但是却读到了过早的版本，就应当回滚。

  如果其读集之中的某一个数据项已经被加锁了，而且这个数据项不在当前事务的写集之中（也就是说并不是被当前事务自己加的锁）。那么说明已经有另外一个线程已经开始验证并且会对这一个数据项进行修改。在这种情况下，直接进行回滚。
- 第三阶段，如果事务验证通过了，那么就将该事务的写集中的数据实际写回，并释放在写的数据项上所加的锁。

对这个算法进行总结，一个并发控制算法主要分为两大部分，定序和检验。

- 定序：按照事务的开始验证时间
- 检验：在验证的时候进行检验，主要检验事务读集之中的每一个数据项，检验他们是否存在写写冲突、读写冲突、写读冲突，如果检验不通过，直接回滚。具体检验如下：
- 写写冲突：由于按照验证时间戳排序，验证顺序即为修改顺序，所以写写不会冲突，会直接吸入
- 读写冲突：如果先写后读，那么后读的事务在检验的时候会因为wts已经大于当前事务的开始时间戳而导致回滚
- 写读冲突：如果如果先读后写，那么读的时候在检验的时候会因为同样的原因而回滚。

3.2 实现步骤

对于在数据项上的信息，我仿照OCCTEMPLATE的实现，设计了一个Row_occ类，这个类的框架结构如下所示：

```
class Row_occ {
public:
    void                init(RowData * row);
    RC                  access(TxnMgr * txn,access_t type);
    void                latch();
    bool                validate(uint64_t valid_ts);
    void                write(RowData * data, uint64_t ts);
    void                release();
    bool                locked();
private:
    sem_t               _semaphore;
    RowData *           _row;
    ts_t                wts;
};
```

private部分：

- 设置了一个信号量，_semaphore,用于一个事务在验证的时候对写集进行加锁，分别对每一个涉及到的数据项等待该锁。其实我是把这个信号量当成了互斥锁来使用，此处用互斥锁替代也是同样没有问题的。
- _row，这个manager所对应的RowData类型的数据。每一个RowData之中包含了一个manager，但是这个manager却没有办法来直接访问这个RowData，为了可以访问，在manager之中又声明了另外一个RowData，在初始化的时候直接进行指针的赋值。这里没有必要进行深拷贝，直接赋值指针即可。
- wts，表示最近修改这个数据项的事务的开始时间戳。

public部分，所有的函数具体实现全部放在了Row_occ.h里边，内容如下所示：

```
void Row_occ::init(RowData * row) {
    _row = row;
    sem_init(&_semaphore, 0, 1);
    wts = 0;
}
```

- init()主要用于对Row_occ结构进行初始化，并不需要太多复杂的操作，只需要将内置的_row设进行指针赋值，以及初始化信号量并且设置最后修改时间为0即可。

```
void Row_occ::latch() {
  sem_wait(&_semaphore);
}
void Row_occ::release() {
  sem_post(&_semaphore);
}
```

- latch()和release()都比较简单，只是为了对私有变量信号量进行操作，分别进行加锁和解锁工作，也就是信号量的wait和post两个操作。

```
RC Row_occ::access(TxnMgr * txn) {
    RC rc = RCOK;
    sem_wait(&_semaphore);
    if (txn->get_start_timestamp() < wts) {
        INC_STATS(txn->read_thd_id(),occ_ts_abort_cnt,1);
        rc = Abort;
    }
    else {
        txn->cur_row->copy(_row);
        rc = RCOK;
    }
    sem_post(&_semaphore);
    return rc;
}
```

- access是实际对于各个事务对于该数据项元组的读和写操作进行访问控制的结构。进入函数首先需要获得加在这一数据结构上面的信号锁，然后然后判断发起请求的事务的开始时间戳。如果这个时间戳要早于所要访问的元组的最后一次修改的时间，说明按照MyOCC的定序，这个事务应该看到的是修改之前的值，他不能看到这个数据项“未来”的值，因此直接将状态设置为Abort并返回，提前回滚实现了Early Abort。

```
bool Row_occ::validate(uint64_t ts) {
    return ts>=wts;
}
```

- validate函数是最终当一个事务进入验证阶段时，就它所访问过的每一行进行验证，这个事务的开始时间必须要早于这个元组最后一次被修改的时间才能够验证成功。

```
void Row_occ::write(RowData * data, uint64_t ts) {
    _row->copy(data);
    wts = ts;
}
```

- write函数是在修改修改当前元组时所需要使用到的，将数据进行修改之后更新last write timestamp，同时在此之前要确保ts>wts，否则不能够修改。

由于OCC需要使用全局验证，所以我设计了MyOCC类，专门用于进行全局的验证。

```
class MyOCC {
public:
    void init();
    RC validate(TxnMgr * txn);
    void finish(RC rc, TxnMgr * txn);
private:
    volatile uint64_t tnc;
    sem_t   valid_semaphore;
};
```

对于private部分：

- tnc是对进行验证的事务的一个计数值
- valid_semaphore是信号量，如果使用的是silo无 临界区的验证方法，这个信号量不需要使用

对于public部分：

```
void MyOCC::init() {
    sem_init(&_semaphore, 0, 1);
    tnc = 0;
}
```

- init()对于全局的MyOCC类型进行初始化，只需要做两件事情， 初始化信号量然后将事务计数值设置为0

```
RC MyOCC::validate(TxnMgr * txn) {
    RC rc=RCOK;
    uint64_t starttime = acquire_ts();
    for (uint64_t i = txn->access_get_cnt() - 1; i > 0; i--) {
        for (uint64_t j = 0; j < i; j ++) {
            int tabcmp = strcmp(txn->access_get_original_row(j)-
>get_table_name(),
                txn->access_get_original_row(j+1)->get_table_name());
            if (tabcmp > 0 || (tabcmp == 0 && txn->access_get_original_row(j)-
>get_primary_key() > txn->access_get_original_row(j+1)->get_primary_key())) {
        txn->swap_accesses(j,j+1);
            }
        }
    }
    std::set<uint64_t> write_rowid;
    std::vector<uint64_t> read_accessid;
    for(uint64_t i=0;i<txn->access_get_cnt();i++){
        if(txn->access_get_type(i)==WR){
            write_rowid.insert(txn->access_get_original_row(i)->get_row_id());
            txn->access_get_original_row(i)->manager->latch();
        }
        else if(txn->access_get_type(i)==RD){
            read_accessid.push_back(i);
        }
    }

    for(uint64_t i=0;i<read_accessid.size();i++){
        if(!txn->access_get_original_row(read_accessid[i])->manager->validate(
txn->get_start_timestamp())){
            rc=Abort;
            break;
```

```
        }
        if(write_rowid.find(txn->access_get_original_row(read_accessid[i])-
>get_row_id())==write_rowid.end()
        && txn->access_get_original_row(read_accessid[i])->manager->locked()){
            rc=Abort;
            break;
        }
    }

    return rc;
}
```

- validate(TxnMgr*)是MyOCC进行全局验证的主要实现部分，首先通过一个冒泡排序对所有的access集中的内容进行排序，排序的准则是按照表明的字典序，如果字典序相同的话就按照列名的字典序。这样做的目的是使得各个事务在检验的时候对锁的获取顺序是相同的，从而保证不会出现死锁的情况。接下来使用write_rowid和read_accessid，来记录读集和写集，之后的验证操作在前面已经有介绍，这里不再赘述。

```
void MyOCC::finish(RC rc, TxnMgr * txn) {
  if(rc == RCOK) {
        txn->set_end_timestamp(global_manager.get_ts( txn->read_thd_id() ));
  }
}
```

- finish()函数会给事务设置一个结束的时间戳，一开始构想可以利用这个时间戳来进行一些调度，后来在后续的函数实现之中并没有用到。

4 Cicada

Cicada算法是OCC+MVCC进行结合的一个范例，我依据助老师和两位助教师兄提供的算法整理完成了Cicada一个比较原始的实现。

4.1 算法思路

- 定序：同原始的OCC不同，使用开始时间戳而不是验证时间戳
- 检验：在验证阶段对三种冲突进行检验
- 写写冲突：因为在验证阶段，写事务需要写入PENDING版本并检验有没有更新的版本，所以后写入PENDING版本的事务会回滚的。
- 写读冲突：如果读到了PENDING的版本，会阻塞直至PENDING版本变成COMMITED版本或者是ABORTED版本再读；如果读到的写之前的旧版本，那么读在检验的时候会因为具有更新的版本而回滚
- 读写冲突：如果先写后读，读事务会自动地在版本链中寻找合适的版本，而不会导致冲突

4.2 实现步骤

首先定义了一个结构体，RowVersionEntry，用来存储每一个RowData的每一个版本，相当于是版本链条中的一个节点。

```
struct RowVersionEntry {
    ts_t wts;
    ts_t rts;
    sem_t status_lock;
    version_t status;
    RowData *version_data;
    RowVersionEntry * past;
    RowVersionEntry * future;
};
```

其中wts和rts分别是最后修改事务的开始时间戳和最后阅读事务的开始时间戳。status_lock用以控制状态，避免读和写的时候获取到的是PENDING版本。version_data是一个指针，指向这个版本所存储的RowData。past和future分别是指向旧一点的版本和新一点的版本的指针。

同样需要对于每一行，设置一个Row_mvcc的类，用来对一个数据项进行控制。

```
class Row_mvcc {
public:
    void init(RowData* initrow); // init function
    RC access(access_t type, TxnMgr * txn, RowData * row);
    RC commit(access_t type, TxnMgr * txn, RowData* root, RowData * data);
    RC abort(access_t type, TxnMgr * txn, RowData* root);
    ~Row_mvcc();
    sem_t list_sem;
    std::vector<RowVersionEntry*> *row_version_list;    // version chain
    std::map<ts_t,int> *rvm;
    std::map<ts_t,int> *wvm;
    std::map<ts_t,int> *cvm;
    uint64_t row_version_len;       // length of version chain

};
```

- list_sem作为信号量，来保证只能由一个线程来访问某一个数据项的版本链条。
- rvm 是读版本映射，将一个事务的开始时间戳映射到这个事务在这个数据项上所读的版本，也就是对应在版本链之中的下标。
- wvm是写版本映射，同rvm，只对写过该数据项的事务有映射
- cvm是提交版本映射，如果一个事务在某一个数据项的版本链中提交了PENDING的记录，那么就会有此映射，将该事务的开始时间戳映射到版本下标
- row_version_list是版本链，组织成了vecotor的形式。

```
void Row_mvcc::init(RowData* initrow) {
    row_version_list=new std::vector<RowVersionEntry*>();
    rvm=new std::map<ts_t,int>();
    wvm=new std::map<ts_t,int>();
    cvm=new std::map<ts_t,int>();
    struct RowVersionEntry* new_version=(RowVersionEntry*)
alloc_memory.alloc(sizeof(RowVersionEntry));
    new_version->rts=new_version->wts=0;
    new_version->status=COMMITED;
    new_version->version_data=initrow;
    sem_init(&(new_version->status_lock),0,0);
    row_version_list->push_back(new_version);
    row_version_len = 1;
    sem_init(&list_sem,0,1);
}
```

- init()函数，主要进行初始化，对类中的四个容器全部分配新空间，并将原始版本作为版本链条上的第一个节点链入版本链之中。

```
RC Row_mvcc::access(access_t type, TxnMgr * txn, RowData * row) {
    RC rc = Abort;
    int i;
    sem_wait(&list_sem);
    for(i=row_version_list->size()-1;i>=0;i--){
        if(row_version_list->at(i)->wts>txn->get_start_timestamp())
            continue;
        if(row_version_list->at(i)->status==PENDING)
            sem_wait(&(row_version_list->at(i)->status_lock));
        if(row_version_list->at(i)->status==ABORTED)
            continue;
        if(row_version_list->at(i)->status==COMMITED){
            rc=RCOK;
            row->copy(row_version_list->at(i)->version_data);
            if(type==RD)
                (*rvm)[txn->get_start_timestamp()]=i;
            else if(type==WR)
                (*wvm)[txn->get_start_timestamp()]=i;
            break;
        }
    }
    sem_post(&list_sem);
    return rc;
}
```

- access()函数，用于判断读和写请求能否访问数据。首先在版本链之中找到正确的版本，也就是写入时间戳小于当前事务开始时间戳的最新版本。然后判断这个版本当前的状态，如果仍处于PENDING状态，就等待直至状态改变；如果处于ABORTED状态，就说明这个版本已经废弃，应当寻找下一个版本；如果处于COMMITED状态，说明已经找到合适的版本，可以直接将指针赋值给txn->cur_row,然后返回。

```
RC Row_mvcc::commit(access_t type, TxnMgr * txn, RowData* root, RowData * data)
{
    RC rc = RCOK;
    int semvalue;
    if(type==RD)
        return rc;
    //std::shared_lock lock(list_mutex);
    sem_wait(&list_sem);
    size_t index=(*cvm)[txn->get_start_timestamp()];
    row_version_list->at(index)->version_data=
(RowData*)alloc_memory.alloc(sizeof(RowData));
    row_version_list->at(index)->version_data->init(root->get_table(), root-
>get_part_id());
    row_version_list->at(index)->version_data->copy(data);
    row_version_list->at(index)->status=COMMITED;
    while(!sem_getvalue(&(row_version_list->at(index)->status_lock),&semvalue)
&& semvalue<0)
        sem_post(&(row_version_list->at(index)->status_lock));
    //std::shared_lock unlock(list_mutex);
    sem_post(&list_sem);
    return rc;
}
```

- commit()函数，将之前事务在这个数据项的版本链上写入的PENDING版本修改为COMMITED状态，同时重新开辟空间存储这一版本的RowData。注意这个地方不可以直接使用指针来赋值，这个指针在函数返回之后马上就会被释放掉，会让这个这个版本的指针直接指飞，造成bug。正确的做法应当是重新开辟一个RowData的空间并进行深拷贝。

```
RC Row_mvcc::abort(access_t type, TxnMgr * txn, RowData* root){
    RC rc = RCOK;
    int semvalue;
    if(type==RD)
        return rc;
    //std::shared_lock lock(list_mutex);
    sem_wait(&list_sem);
    size_t index=(*cvm)[txn->get_start_timestamp()];
    row_version_list->at(index)->status=ABORTED;
    while(!sem_getvalue(&(row_version_list->at(index)->status_lock),&semvalue)
&& semvalue<0)
        sem_post(&(row_version_list->at(index)->status_lock));
    //std::shared_lock unlock(list_mutex);
    sem_post(&list_sem);
    return rc;
}
```

- abort()相比之下简单一点，只需要修改状态为ABORTED，通过sem_getvalue()函数获得当前信号量的值，signal信号量让所有在此PENDING版本上正在等待的线程可以继续运行即可。

除了Row_mvcc之外，还需要一个类，用于进行全局的事务验证，为此，我设计了MVOCC类，内容如下：

```
class MVOCC {
public:
    void init();
    RC validate(TxnMgr * txn);
    void finish(RC rc, TxnMgr * txn);
private:
    volatile uint64_t tnc;
    sem_t   valid_semaphore;
};
```

MVOCC类本身就是为了验证而设的，所以其最为核心的内容就是验证函数validate

```
RC MVOCC::validate(TxnMgr * txn){
    RC rc=RCOK;
    for(size_t i=0;i<txn->access_get_cnt();i++){
        if(txn->access_get_type(i)!=WR)
            continue;
        sem_wait(&(txn->access_get_original_row(i)->manager->list_sem));
        std::map<ts_t,int>* wvmp=txn->access_get_original_row(i)->manager->wvm;
        std::map<ts_t,int>* cvmp=txn->access_get_original_row(i)->manager->cvm;
        std::vector<RowVersionEntry*> * list_ptr=txn-
>access_get_original_row(i)->manager->row_version_list;
        size_t index=(*wvmp)[txn->get_start_timestamp()];
        if( ( (index!=list_ptr->size()-1) && (list_ptr->at(list_ptr->size()-1)-
>rts!=txn->get_start_timestamp()) )
        || list_ptr->at(index)->rts > txn->get_start_timestamp()){
            sem_post(&(txn->access_get_original_row(i)->manager->list_sem));
            return Abort;
```

```
        }
        else{
            RowVersionEntry* new_version=
(RowVersionEntry*)alloc_memory.alloc(sizeof(RowVersionEntry));
            new_version->status=PENDING;
            new_version->rts=new_version->wts=txn->get_start_timestamp();
            list_ptr->push_back(new_version);
            (*cvmp)[txn->get_start_timestamp()]=list_ptr->size()-1;
            sem_post(&(txn->access_get_original_row(i)->manager->list_sem));
        }
    }
    //update read timestamp
    for(size_t i=0;i<txn->access_get_cnt();i++){
        if(txn->access_get_type(i)!=RD)
            continue;
        sem_wait(&(txn->access_get_original_row(i)->manager->list_sem));
        std::map<ts_t,int>* rvmp=txn->access_get_original_row(i)->manager->rvm;
        std::vector<RowVersionEntry*> * list_ptr=txn-
>access_get_original_row(i)->manager->row_version_list;
        size_t index=(*rvmp)[txn->get_start_timestamp()];
        if(list_ptr->at(index)->rts<txn->get_start_timestamp())
            list_ptr->at(index)->rts=txn->get_start_timestamp();
        sem_post(&(txn->access_get_original_row(i)->manager->list_sem));
    }
    //check version consistency
    for(size_t i=0;i<txn->access_get_cnt();i++){
        sem_wait(&(txn->access_get_original_row(i)->manager->list_sem));
        std::map<ts_t,int>* rvmp=txn->access_get_original_row(i)->manager->rvm;
        std::vector<RowVersionEntry*> * list_ptr=txn-
>access_get_original_row(i)->manager->row_version_list;
        std::map<ts_t,int>* wvmp=txn->access_get_original_row(i)->manager->wvm;
        if(txn->access_get_type(i)==RD){
            size_t index=(*rvmp)[txn->get_start_timestamp()];
            for(size_t j=index+1;j<list_ptr->size();j++){
                if(list_ptr->at(j)->wts<=txn->get_start_timestamp()){
                    sem_post(&(txn->access_get_original_row(i)->manager-
>list_sem));
                    return Abort;
                }
            }
        }
        else if(txn->access_get_type(i)==WR){
            size_t index=(*wvmp)[txn->get_start_timestamp()];
            if(list_ptr->at(index)->rts>txn->get_start_timestamp()){
                sem_post(&(txn->access_get_original_row(i)->manager->list_sem));
                return Abort;
            }
        }
        sem_post(&(txn->access_get_original_row(i)->manager->list_sem));
    }
    return rc;
}
```

整个validate的工作逻辑可以概括如下：

- 写入新版本，对于写集之中的每一个数据项，检查是否有更新的版本或者是否被后面的事务读过，如果是，就直接回滚，否则插入PENDING版本，但不必为这个PENDING版本的version_data赋

值，因为没有事务可以读到PENDING版本
- 更新读时间戳，对于每一个读事务，使用其开始时间戳来更新数据项对应版本的rts
- 检验一致性。对于每一个读事务，如果存在更新的版本，而且其wts要小于当前事务的rts，就进行回滚；对于每一个写事务，如果他写之前的版本被之后的事务读过了，就进行回滚。全部验证通过之后，就可以提交了。

5 总结与反思

- 在所有的Row_occ/Row_mvcc结构体之中，不可以直接声明可变大小的容器如vector，list等，这些东西全部都只能在类内部声明一个指针，然后在初始化的时候使用new来得到一个可用的容器；
- 内存分配时出现段错误，往往是因为上一次内存分配时有越界现象；
- 很多时候传参都是一个类的指针，首先要检查代码逻辑，看这个指针有没有分配空间，才可以通过这个指针调用这个类的变量和成员函数。